Source Encoding and Compression

Jukka Teuhola

Computer Science
Department of Information Technology
University of Turku
Spring 2016

Lecture notes

Table of Contents

1. Introduction 3
2. Coding-theoretic foundations 6
3. Information-theoretic foundations 12
4. Source coding methods 15
   4.1. Shannon-Fano coding 15
   4.2. Huffman coding 17
   4.2.1. Extended Huffman code 20
   4.2.2. Adaptive Huffman coding 20
   4.2.3. Canonical Huffman code 23
   4.3. Tunstall code 26
   4.4. Arithmetic coding 29
   4.4.1. Adaptive arithmetic coding 35
   4.4.2. Adaptive arithmetic coding for a binary alphabet 36
   4.4.3. Viewpoints to arithmetic coding 39
5. Predictive models for text compression 40
   5.1. Predictive coding based on fixed-length contexts 42
   5.2. Dynamic-context predictive compression 48
   5.3. Prediction by partial match 52
   5.4. Burrows-Wheeler Transform 58
6. Dictionary models for text compression 62
   6.1. LZ77 family of adaptive dictionary methods 63
   6.2. LZ78 family of adaptive dictionary methods 66
   6.3. Performance comparison 71
7. Introduction to Image Compression 72
   7.1. Lossless compression of bi-level images 72
   7.2. Lossless compression of grey-scale images 75
   7.3. Lossy image compression: JPEG 77

Literature (optional)

- T. C. Bell, J. G. Cleary, I. H. Witten: *Text Compression*, 1990.
- R. W. Hamming: *Coding and Information Theory*, 2nd ed., Prentice-Hall, 1986.
- K. Sayood: *Introduction to Data Compression*, 3rd ed., Morgan Kaufmann, 2006.
- K. Sayood: *Lossless Compression Handbook*, Academic Press, 2003.
- I. H. Witten, A. Moffat, T. C. Bell: *Managing Gigabytes*: compressing and indexing documents and images, 2nd ed., Morgan Kaufmann, 1999.
- Miscellaneous articles (mentioned in footnotes)

1. Introduction

This course is about *data compression*, which means reducing the size of source data representation by decreasing the *redundancy* occurring in it. In practice this means choosing a suitable coding scheme for the source symbols. There are two practical motivations for compression: Reduction of storage requirements, and increase of transmission speed in data communication. Actually, the former can be regarded as transmission of data 'from now to then'. We shall concentrate on *lossless compression*, where the source data must be recovered in the *decompression* phase exactly into the original form. The other alternative is *lossy compression*, where it is sufficient to recover the original data *approximately*, within specified error bounds. Lossless compression is typical for alphabetic and other symbolic source data types, whereas lossy compression is most often applied to numerical data which result from digitally sampled continuous phenomena, such as sound and images.

There are two main fields of coding theory, namely

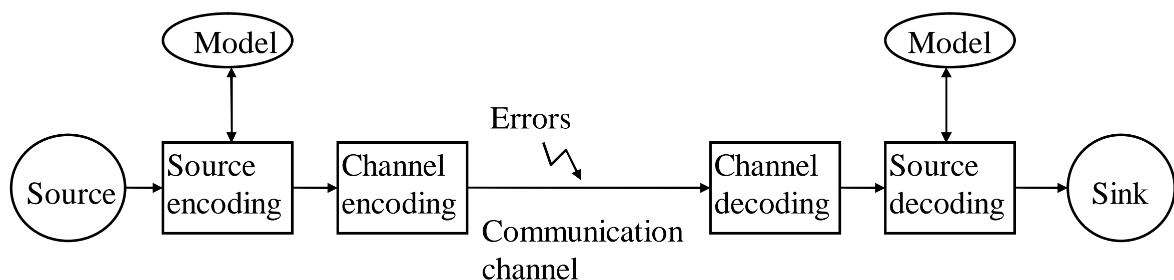
1. *Source coding*, which tries to represent the source symbols in minimal form for storage or transmission efficiency.
2. *Channel coding*, the purpose of which is to enhance detection and correction of transmission errors, by choosing symbol representations which are far apart from each other.

Data compression can be considered an extension of source coding. It can be divided into two phases:

1. *Modelling* of the information source means defining suitable units for coding, such as characters or words, and estimating the probability distribution of these units.
2. *Source coding* (called also *statistical* or *entropy coding*) is applied to the units, using their probabilities.

The theory of the latter phase is nowadays quite mature, and optimal methods are known. Modelling, instead, still offers challenges, because it is most often approximate, and can be made more and more precise. On the other hand, there are also practical considerations, in addition to minimizing the data size, such as compression and decompression *speed*, and also the size of the model itself.

In data transmission, the different steps can be thought to be performed in sequence, as follows:



We consider source and channel coding to be independent, and concentrate on the former. Both phases can also be performed either by hardware or software. We shall discuss only the software solutions, since they are more flexible.

In compression, we usually assume two *alphabets*, one for the source and the other for the target. The former could be for example ASCII or Unicode, and the latter is most often binary. There are many ways to classify the huge number of suggested compression methods. One is based on the grouping of source and target symbols in compression:

1. *Fixed-to-fixed*: A fixed number of source symbols are grouped and represented by a fixed number of target symbols. This is seldom applicable; an example could be reducing the ASCII codes of numbers 0-9 to 4 bits, if we know (from our model) that only numbers occur in the source.

2. *Variable-to-fixed*: A variable number of source symbols are grouped and represented by fixed-length codes. These methods are quite popular and many of the commercial compression packages belong to this class. A manual example is the *Braille code*, developed for the blind, where 2x3 arrays of dots represents characters but also some combinations of characters.

3. *Fixed-to-variable*: The source symbols are represented by a variable number of target symbols. A well-known example of this class is the *Morse code*, where each character is represented by a variable number of dots and dashes. The most frequent characters are assigned the shortest codes, for compression. The target alphabet of Morse code is not strictly binary, because there appear also inter-character and inter-word spaces.

4. *Variable-to-variable*: A variable-size group of source symbols is represented by a variable-size code. We could, for example, make an index of all words occurring in a source text, and assign them variable-length binary codes; the shortest codes of course for the most common words.

The first two categories are often called *block coding* methods, where the block refers to the fixed-size result units. The best (with respect to compressed size) methods today belong to category 3, where extremely precise models of the source result in very high gains. For example, English text can typically be compressed to about 2 bits per source character. Category 4 is, of course, the most general, but it seems that it does not offer notable improvement over category 3; instead, modelling of the source becomes more complicated, in respect of both space and time. Thus, in this course we shall mainly take example methods from categories 2 and 3. The former are often called also *dictionary methods*, and the latter *statistical methods.*

Another classification of compression methods is based on the availability of the source during compression:

1. *Off-line* methods assume that all the source data (the whole message[1]) is available at the start of the compression process. Thus, the model of the data can be built before the actual encoding. This is typical of storage compression trying to reduce the consumed disk space.

2. *On-line* methods can start the coding without seeing the whole message at once. It is possible that the tail of the message is not even generated yet. This is the normal situation in data transmission, where the sender generates source symbols one-by-one and compresses them on the fly.

---

[1] We shall use the word *message* to represent the unit which is given as input to one execution of the compression program. In storage compression, the message usually means a *file*.

Still another classification, related to this one, is based on the emergence and status of the source model:

1. *Static* methods assume a fixed model, which is used by both the encoder and decoder, and which is common for all messages from the source. This is acceptable, if there is not much variation between distinct messages, for example, if they are all English texts from the same application domain.

2. *Semi-adaptive* methods construct a model (for example a *codebook*) separately for each source message. The model is based on the whole message and built before starting the actual encoding. Thus, two passes through the data are needed. The encoder must first send the model and then the encoded message to the decoder. This presupposes an off-line method. In some data communication situations this kind of delay is not acceptable.

3. *Adaptive* methods construct the model on the fly. When encoding a certain input symbol (or group of symbols), the model is based on the previous part of the message, already encoded and transmitted to the decoder. This means that *the decoder can construct the same model intact*. Both partners gradually *learn* the properties of the source. The process starts with some initial model, for example a uniform distribution for all possible symbols, and then changes the model to converge towards the actual distribution.

Even though semi-adaptive methods can in principle create a more precise model of the source, it has turned out that adaptive methods can achieve the same compression efficiency. One reason is that they do not have to transmit the model from the encoder to the decoder. Another advantage is that they may adapt to local changes in the statistical properties within the message. Adaptive methods are quite common today in general-purpose compression software.

Compression efficiency is normally measured by the *compression ratio*, i.e. the ratio of the source message size to its compressed size, both measured in bits. In text compression, an objective measure is *bits per source character* (*bpc*), because it is independent of the original coding. In image compression, *bits per pixel* is a suitable measure.

Earlier, character codes were normally of fixed length, such as ASCII (7 or 8 bpc), and ISO-8859 character sets (8 bits per character). The more recent *Unicode* has been adopted in many contexts (Java, XML). Some Unicode versions have a fixed number of bits per character: UCS-2 with 16 bpc, and UCS-4 with 32 bpc. The others are of variable length: UTF-8 with 8, 16, 24 or 32 bpc, and UTF-16 with 16 or 32 bpc. Even though our source text characters may have a variable number of bits, we can still talk about e.g. fixed-to-variable coding, if we encode one source character at a time because a character is logically a fixed unit.

2. Coding-theoretic foundations

Here we study the coding of messages, consisting of *symbols* from a given *source alphabet*, denoted $S = \{s_1, ..., s_q\}$. In practice, the actual alphabet, used in coding, can be an *extension* of the original, so that also groups of symbols are assigned their own codes. A *codebook* contains a *codeword* w_i for each (extended) symbol s_i. Here we assume that the codewords can be of variable length. In adaptive compression, the codebook can also be *dynamic*, so that it may change during the coding of the source message. However, for each position of the message, there is a unique, discrete sequence of bits for every possible symbol.

The codebook may also be *implicit* if the code is a computable function of the source symbols. The function is determined from our model of the source. As mentioned, it is possible to encode a group of source symbols as a unit. Since the codebook grows exponentially with the group size, large groups can only be coded with implicit methods. An extreme is a method which treats the whole message as one group, and computes a *single* codeword for it. In such coding, there is no precise correspondence between individual source symbols and coded bits: For a certain bit in the coded sequence, we cannot say exactly, which source symbol has produced it. *Arithmetic coding*, to be discussed later, is an example of this kind of method.

Let us study the desirable properties what the codebook (explicit or implicit), denoted $W = \{w_1, ..., w_q\}$, should have. The first necessary condition is, of course, uniqueness: For each $s_i, s_j \in S$: $s_i \neq s_j \Rightarrow C(s_i) \neq C(s_j)$, where C is the coding function. This condition would be sufficient if we should transmit only a single symbol (and if end-of-code can be signalled by some means). However, this is not usually the case. For multi-symbol messages (and variable-length codewords) we have two options:

(a) use some special symbols (so called *commas*) between the codes,
(b) develop *self-punctuating* codes.

The former would be inefficient use of the special symbols, thus we concentrate on the latter.

Let X and Y be any two sequences of source symbols. If $X \neq Y \Rightarrow C(X) \neq C(Y)$, then code C is called *uniquely decodable (decipherable)*. The following is an example of an ill-designed codebook

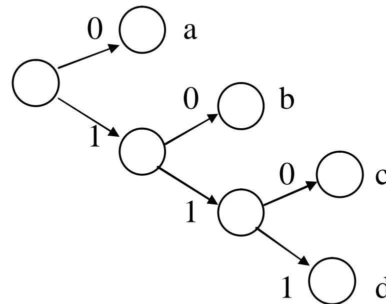
a = 0
b = 01
c = 11
d = 00

Now consider the encoded sequence "00110". It can result either from message "dca" or "aaca". The codebook (or *code*, for short) is not uniquely decodable. The following codebook is, instead, such:

a = 0
b = 10
c = 110
d = 111

Here the only interpretation of “00110” is “aac”. This code is uniquely decodable and, moreover, a *prefix-free* code (*prefix* code, for short): No codeword is the prefix of any other codeword.

Codebooks are often illustrated by binary trees (called also *decoding trees* or *decision trees*). The latter of the above codebooks can be visualized as follows:



It is easy to see that a prefix-free code is represented by a binary tree where each source symbol corresponds to a *leaf* node. Most often this kind of coding trees are also *complete*, i.e. all internal nodes have two children. The prefix-free code is always an *instantaneous* code, i.e. we know the source symbol as soon as all its code bits have been received. Instantaneous decoding is extremely useful, but not absolutely necessary for unique decodability. For example, the following code is uniquely decodable, even though it is not prefix-free:

a = 0
b = 01
c = 011
d = 111

When decoding the bit string 00110, we cannot say after the first bit, whether it is part of ‘a’, ‘b’, or ‘c’. After the second bit we can, since there is no code with “00”. The next source symbol can be decided only after the last ‘0’ has been seen. A more general worst case is the sequence “0111...111”, for which the first source symbol can be decided only after seeing the very last bit. The reader might notice that each codeword is the *reverse* of the corresponding prefix-free codeword in the earlier codebook. Unique decodability results from the property that the coded sequence can be decoded backwards as if the codebook were prefix-free (actually it is *suffix-free*).

If the decoding tree is fully balanced, it means that all codewords are of the same length, i.e. we have a block code. If the size of the source alphabet is not a power of 2, at least the lowest level is incomplete. If the difference between the longest and shortest codeword length is 1, then the code is called *shortened block code*. For example, the 5-symbol alphabet {a, b, c, d, e} can be encoded as

a = 00
b = 01
c = 10
d = 110
e = 111

A rather simple condition has been proved about the lengths l_1, ..., l_q of an instantaneous code, namely the *Kraft inequality*:

Theorem 2.1. An instantaneous codebook W of q codewords exists if and only if the lengths $\{l_1, ..., l_q\}$ of the codewords satisfy $K = \sum_{i=1}^{q} \frac{1}{2^{l_i}} \leq 1$.

Proof. For an instantaneous code we have the symbols in the leaves of the decoding tree, and any binary tree with q leaves can be interpreted into an instantaneous code. Thus we show that for any binary tree the depths of leaves satisfy the inequality. We apply induction: For q=1, there is one leaf with depth = 1, so the sum K is equal to ½ ≤ 1. (Note that the code length cannot be zero, so a root-only tree is illegal.) For q=2 we have two leaves with depth 1, so the sum is ½ + ½ ≤ 1. Now assume an arbitrary tree with q nodes, with sum $K \leq 1$.

(a) Extend any leaf (with depth d) with one new leaf (making the old leaf internal). The new leaf has depth d+1. Thus the sum of the theorem becomes $K-1/2^d+1/2^{d+1} \leq K \leq 1$.

(b) Extend any leaf (with depth d) with two new leaves (making the old leaf internal). The new leaves have depth d+1. Now the number of leaves becomes q+1, and the sum value is $K-1/2^d+1/2^{d+1}+1/2^{d+1} = K \leq 1$.

Since any binary tree can be constructed step by step using extensions (a) and (b), the theorem holds.

■

In general form, the Kraft inequality has radix r in the denominator, instead of 2, but we restrict ourselves to the binary target alphabet. Moreover, it can also be shown that, for binary target alphabet, strict equality should hold; otherwise the code is inefficient.

Let us examine the validity of the theorem by studying the previous reduced block code, with codeword lengths {2, 2, 2, 3, 3}. We obtain the sum ¼ + ¼ + ¼ + 1/8 + 1/8 = 1. Thus it is easy to check for a given set of lengths, whether a codebook can be constructed or not, and if yes, whether the code is inefficient. Consider the set of lengths {1, 3, 3, 3}, which produces the sum 7/8. We can directly say that the code is inefficient. The related tree has a leaf with no brother; therefore the leaf can be omitted, and the related symbol gets one bit shorter code-word (corresponding to the parent). Now, the set {1, 2, 3, 3} produces K = 1.

The Kraft inequality holds for instantaneous codes. The *MacMillan inequality* says that the same holds for any uniquely decodable code. The proof is not quite as straightforward as above. Important is that, when choosing a codebook, we can calmly restrict ourselves to instantaneous codes, because for any set of lengths in a uniquely decodable code, we can derive an instantaneous code with the same lengths.

In some applications, the source alphabet can in principle be *infinite* (but yet *countable)*. One such situation is *run-length coding*, where we should encode the number of similar successive symbols in a sequence. Of course, all runs are of finite length, but no upper bound can be given. In this case we need a general coding system for all integers; the traditional fixed-length representation cannot be used. A coding system is in a sense *universal*, if it works for all possible sources in some class. Usually, an additional requirement is set for true universality, such that the code satisfies a certain efficiency constraint (to be explained later). Since the alphabet is infinite, we cannot store an explicit codebook, but we must provide an algorithm (function), which produces the code for any integer. Two desirable properties for such a code are:

- *Effectiveness*: There is an effective procedure that can decide whether a given sequence of bits belongs to the codebook or not.

- *Completeness*: adding any new code would create a codebook that is not uniquely decipherable.

There are various ways of designing the universal code; Peter Elias[1] has described and analysed potential coding approaches for positive integers. A simplified description of them is as follows:

1. *α-coding*: This is unary coding, where number *n* is represented by *n*-1 zeroes plus an ending one-bite: 1→1; 2→01; 3→001; 4→0001; 5→00001; ... [Actually, α-coding is not universal in the strict sense, because it does not satisfy the efficiency constraint.]

2. *β-coding*: This is the traditional (positional) representation of numbers, excluding the leading zeroes. Since its codes are not self-punctuative, we need an end symbol '∗', and thus our code alphabet is *ternary*. The leading 1-bit can be omitted: 1→∗; 2→0∗; 3→1∗; 4→00∗; 5→01∗; 6→10∗; 7→11∗; 8→000∗; ... This can be transformed to the normal binary representation e.g. by further coding: 0→0; 1→10; ∗→11. The final encoding would thus be 1→11; 2→011; 3→1011; 4→0011; 5→01011; 6→10011; 7→101011; 8→00011. Notice that the lengths of the codewords are *not* monotonically increasing with the number to be encoded.

3. *γ-coding*: To avoid using the end symbol, the positional representation is prefixed by an α-coded length. Again, the leading 1-bit is omitted from the number – actually we can think that the 1-bit that ends the unary code is simultaneously the leading 1-bit. The small integers are now coded as follows: 1→1; 2→010; 3→011; 4→00100; 5→00101; 6→00110; 7→00111; 8→0001000; ...

4. *δ-coding*: similar as γ-coding, but the length is not represented with unary coding, but γ-coding. Examples: 1→1; 2→0100; 3→0101; 4→01100; 5→01101; 6→01110; 7→01111; 8→00100000; ...

δ-coding is theoretically better than γ-coding for large numbers (cf. next section).

There are many other useful properties of codes. One such property is *robustness* against transmission errors. In the worst case, inverting a single bit in a sequence of variable-length codes may scramble the whole tail of the coded sequence. The *synchronization property* refers to the behaviour of the decoding process in the context of errors (add/drop/complement a bit). Synchronization means, how fast the decoder will catch again the correct codeword boundaries.

An example of an elegant, robust, universal coding scheme for integers is so called *Zeckendorf* representation (called also *Fibonacci* representation)[2]. Its simplest form encodes

---

[1] Peter Elias: "Universal Codeword Sets and Representations of the Integers", *IEEE Trans. on Inf. Theory*, Vol. IT-21, No. 2, 1975.

[2] A. S. Fraenkel, S. T. Klein "Robust Universal Complete Codes for Transmission and Compression ", *Discrete Applied Mathematics*, Vol. 64, 1996, pp 31–55.

numbers by bit-weighted sums of Fibonacci numbers …, 34, 21, 13, 8, 5, 3, 2, 1. For example, the value 17 can be represented by the sum $0·34 + 0·21 + 1·13 + 0·8 + 0·5 + 1·3 + 0·2 + 1·1$. The robustness of Zeckendorf code results from the fact that every number can be represented with a code with no pairs of adjacent 1-bits (results immediately from the Fibonacci system). Thus, by extending the code with two additional 1-bits we can make the code self-delimiting. Moreover, by transmitting the least significant bits first, up to the most significant 1-bit, *only a single extra* 1*-bit is needed to delimit the code.* The codebook for small integers looks like follows.

Number	**Weights for** **8 5 3 2 1**	**Code**
1	0 0 0 0 1	11
2	0 0 0 1 0	011
3	0 0 1 0 0	0011
4	0 0 1 0 1	1011
5	0 1 0 0 0	00011
6	0 1 0 0 1	10011
7	0 1 0 1 0	01011
8	1 0 0 0 0	000011
9	1 0 0 0 1	100011
10	1 0 0 1 0	010011
11	1 0 1 0 0	001011
12	1 0 1 0 1	101011
…	…	…

Since no two consecutive 1-bits can exist within a codeword, excluding the end, single-bit errors can only propagate to the next codeword, but not further. What is more, it has been shown that basic arithmetic operations can be performed directly with Zeckendorf-coded integers[1].

However, transmission errors are not our main concern in this course, so we shall later only mention some results and observations from this topic.

As an example of a *parametric* code, we present the so called *Golomb code*[2], which is optimal for specific types of numeric sources. As a preliminary step, we introduce a code for *uniform* distribution of a finite set of symbols (i.e. all q symbols are equally likely). If q is a power of two ($q = 2^k$), then the codes for the alphabet naturally consist of all possible k-bit combinations. However, if q is not a power of 2, a fixed-length code of $\lceil \log_2 q \rceil$ bits would be redundant (some code values unused). In this case we choose a so-called *semi-fixed-length* code, where some of the codewords have length $\lfloor \log_2 q \rfloor$, and the rest have length $\lceil \log_2 q \rceil$. The following tables shows semi-fixed-length codebooks for q in 2…8.

x	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$	$q = 7$	$q = 8$
0	0	0	00	00	00	00	000

[1] P. Fenwick: "Zeckendorf Integer Arithmetic", *Fibonacci Quarterly*, Nov 2003, pp. 405 – 413.
[2] S. W. Golomb: "Run-Length Encodings", *IEEE Trans. on Inf. Theory*, Vol. 12, 1966, pp. 399-401.

1	1	10	01	01	01	010	001
2		11	10	10	100	011	010
3			11	110	101	100	011
4				111	110	101	100
5					111	110	101
6						111	110
7							111

Golomb coding is applied to numeric data (i.e. integers). It has one parameter m, which is used in encoding of an integer x as follows. Encode the quotient $a = \lfloor x/m \rfloor$ with unary coding, and the remainder $x - ma$ with semi-fixed-length coding, where the alphabet size $q = m$. The two parts are concatenated to codewords. An example with m=3 is given in the table below. The parts of the codewords are separated by a space, for clarity. In the special case where m is a power of 2 (denote 2^k) we obtain a so called *Rice code*[1], which is structurally simpler, as shown in the table for $m = 2$.

x	Golomb $m = 3$	Rice $m = 2$
0	0 0	00
1	0 10	01
2	0 11	100
3	10 0	101
4	10 10	1100
5	10 11	1101
6	110 0	11100
7	110 10	11101
8	110 11	111100
9	1110 0	111101

Both are very efficient in practice, if the distribution of source numbers is close to geometric. Such a distribution results e.g. from repeating binary events (sequences of 'success'/'failure') where a success is more probable than a failure, and we encode the number of successes up to the next failure. Another interpretation of the same situation is run-length encoding of a binary sequence. If $p = P(0) > P(1)$, then m should be chosen such that the probability of m 0-bits before the next 1-bit is about 0.5. From this it follows that $m \approx -1/\log_2 p$ is the best choice. The next chapter studies the connection between symbol probabilities and optimal codes more deeply.

[1] R. F. Rice: "Some Practical Universal Noiseless Coding Techniques", Jet Propulsion Laboratory, JPL Publication 79-22, Pasadena, CA, 1979.

3. Information-theoretic foundations

Information theory, mainly founded by Claude Shannon[1] at Bell Labs, studies both noise protection and efficient use of the communication channel. It gives bounds for the ultimate transmission rate of the channel, and for the ultimate data compression. Of these two we concentrate on the latter. First we shall define a quantitative *measure of information* for both single symbols and the whole alphabet. Notice that the word 'information' is here a quite concrete technical concept, and bears only loose analogy with the 'normal' concept of information in human communication.

Suppose that we have a source alphabet $S = \{s_1, \dots s_q\}$, with probabilities $P = \{p_1, \dots p_q\}$. The question is, how much information the receiver gets when seeing a certain symbol s_i. The principle is that a less frequent character gives more information than a more frequent one. Thus, the amount of information measures the degree of *surprise* (uncertainty). If some $p_i = 1$ then all symbols to be sent are $= s_i$, and there is no surprise – no information. For constructing an information function $I(p)$, three properties should be satisfied:

1. $I(p) \geq 0$,
2. $I(p_1 p_2) = I(p_1) + I(p_2)$ for independent events,
3. $I(p)$ is a continuous function.

From condition 2 it follows that $I(p^n) = nI(p)$. We see that I satisfies the same properties as the logarithm function. In fact, the solution is essentially unique, and we define $I(p) = k \log p$. From property 1 (and noticing that $0 \leq p \leq 1$), we deduce the value $k = -1$. Hence we have:

$$I(p) = -\log p = \log (1/p)$$

The base of logarithm can be chosen. Normally we take base = 2, and define the *unit of information* to be a *bit*. This word has now two meanings: a digit in 2-base representation and a unit in measuring information. Of course, this is not a coincidence, as seen from the following example: Consider tossing a (fair) coin with P(heads) = P(tails) = ½. Then I(heads) = I(tails) = –log ½ = 1 (bit), which is consistent with the idea that with one binary digit we can tell the outcome. If the coin is unfair, such that P(heads) = 1/8 and P(tails) = 7/8, then we have I(heads) = –log (1/8) = 3 and I(tails) = –log (7/8) ≈ 0.193 bits. Thus getting heads is a much bigger surprise than getting tails. Notice also that the information amount is a real-valued quantity, and thus more general than the binary digit. For studying the additive property, consider tossing a fair coin three times. The information amount of the sequence: heads-tails-heads (or any other 3-sequence), is: I(½½½) = I(1/8) = –log (1/8) = 3 bits = I(½) + I(½) + I(½), as expected.

Next we define the *average* information when receiving a symbol from alphabet S, with probability distribution P. Since we get amount $I(p_i)$ with probability p_i, the average (expected) information per symbol, called *entropy*, is a weighted sum:

$$H(P) = \sum_{i=1}^{q} p_i I(p_i) = \sum_{i=1}^{q} p_i \log\left(\frac{1}{p_i}\right)$$

---

[1] C. E. Shannon: "A Mathematical Theory of Communication", *Bell System Technical Journal*, Vol. 27, pp. 379-423, 623-656, 1948.

Example. Let $P = \{0.1, 0.2, 0.3, 0.4\}$. Then $H(P) = -0.1 \log 0.1 - 0.2 \log 0.2 - 0.3 \log 0.3 - 0.4 \log 0.4 \approx 0.332 + 0.464 + 0.521 + 0.529 =$ **1.846**. Compare this with uniform distribution $P = \{0.25, 0.25, 0.25, 0.25\}$, which has entropy $H(P) = -4 \times 0.25 \log 0.25 =$ **2**. It is easy to prove the observation generally: a uniform distribution gives the maximum entropy. Notice the analogy with physical phenomena e.g. in thermodynamics, where the same word is used.

Shannon has proved the following fundamental "noiseless source encoding theorem":

Theorem 3.1. Entropy $H(S)$ gives a lower bound to the average code length L for any instantaneously decodable system. (Proof skipped)

Even though precise symbol probabilities are seldom known, entropy is an excellent reference measure when studying the compression power of a source coding method. The difference $L-H(S)$ between the average codeword length and entropy is called *redundancy* of the code. The equality in Theorem 3.1 holds if the probabilities p_i are powers of 2, i.e. $p_i = 1/2^{l_i}$. In this case we reach the theoretic optimum with redundancy = 0. Even if this were not the case, L can always be limited to at most $H(S) + 1$ by choosing

$$\log_2\left(\frac{1}{p_i}\right) \le l_i < \log_2\left(\frac{1}{p_i}\right) + 1$$

A code is called *universal* if $L \le c_1 H(S) + c_2$ asymptotically for some constants c_1 and c_2 and for nonzero entropy $H(S)$. The code is *asymptotically optimal* if $c_1 = 1$.

Consider the encoding of natural numbers $n = 1, 2$, etc. Often we do not know their exact probabilities, and they are hard to find out experimentally. However, the following fundamental result has been proved[1]:

Theorem 3.2. There exist codes, which satisfy the above universality condition for *any* distribution satisfying $P(1) \ge P(2) \ge P(3) \ge \ldots \ge P(n) \ge \ldots$ (Proof skipped).

The codeword lengths for such encodings are non-decreasing with n. Of the coding methods for numbers (Chapter 2), it can be shown that α-code is not universal in the above sense, whereas γ- and δ-codes are. Moreover, γ-code has constant $c_1 = 2$, while δ-code has $c_1 = 1$, the latter being asymptotically optimal. Zeckendorf code (called also Fibonacci code) is also universal, but not asymptotically optimal; its c_1-value is about 1.440. Nevertheless, for a large subset of integers, the Zeckendorf codewords are actually shorter than δ-codes.

Even though α-code (= unary code) is not universal, it is *optimal* if the probabilities are:
$P(1) = 1/2, P(2) = 1/4, P(3) = 1/8, \ldots, P(n) = 1/2^n, \ldots$ In fact, this kind of distribution is not uncommon. Note that the sum of probabilities $1/2 + 1/4 + 1/8 + \ldots = 1$, as required.

Above we assumed the successive symbols of the message to be independent. In practice this is not the case, so we should extend the concept of entropy. This leads us to the modelling part of compression, which we shall study later more deeply. Here we mention the natural extension to *conditional probabilities*. Suppose that we know m previous symbols before s_i,

[1] See: Peter Elias: "Universal Codeword Sets and Representations of the Integers", *IEEE Trans. on Inf. Theory*, IT-21, No 2, pp. 194-203, 1975

namely $s_{i_1},\cdots,s_{i_m}$, and the related conditional probability $P(s_i \mid s_{i_1},\cdots,s_{i_m})$. The conditional information of s_i in this *m-memory source* is naturally $\log_2(1/P(s_i|s_{i_1},\cdots,s_{i_m}))$ and the conditional entropy

$$H(S|s_{i_1},\cdots,s_{i_m}) = \sum_S P(s_i|s_{i_1},\cdots,s_{i_m})\log_2\left(\frac{1}{P(s_i|s_{i_1},\cdots,s_{i_m})}\right)$$

The situation can be described by an *m*'th order *Markov process*, which is a finite automaton with states for all *m*-grams of symbols, and *transitions* corresponding to successors of *m*-grams, with the conditional probabilities. In an *ergodic* Markov process the state probabilities approach the so called *equilibrium*. The global entropy is

$$H(S) = \sum_{S^m}\sum_S P(s_{i_1},\cdots,s_{i_m})P(s_i|s_{i_1},\cdots,s_{i_m})\log_2\left(\frac{1}{P(s_i|s_{i_1},\cdots,s_{i_m})}\right)$$

$$= \sum_{S^{m+1}} P(s_{i_1},\cdots,s_{i_m},s_i)\log_2\left(\frac{1}{P(s_i|s_{i_1},\cdots,s_{i_m})}\right)$$

There is another measure for the complexity of a message which is called *Kolmogorov complexity*, and which can be considered more general than entropy. It is defined as the length of the shortest (binary) *program* for generating the message. If the sequence of symbols is drawn at random from a distribution that has entropy *H*, then the Kolmogorov complexity is approximately equal to *H*. However, consider a pseudo random number generator, which produces a seemingly random sequence of symbols. The generating *algorithm* can be regarded as the compressed representation of the message, because the sequence can be recovered by applying it.

Unfortunately, Kolmogorov complexity has been shown to be *non-computable*, i.e. there is no algorithm to determine it (cf. Gödel's incompleteness theorem). This makes the concept rather impractical. However, a somewhat related idea is used in *fractal compression* of images, where the compressor tries to find self-repeating components from the image, i.e. tries to find the 'hidden algorithm' that has generated the image. However, this can only be done approximately (i.e. the method is lossy), unless the image is a pure fractal.

4. Source coding methods

In this section we shall study the second part of compression, namely source coding, and leave the first, 'harder' part (= modelling) to the next sections. Source coding is based on the assumed probability distribution (P) of the source symbols (S). For a finite alphabet it is usually possible to give at least estimates of the probabilities. For an infinite alphabet the distribution is often implicit. For example, we could assume that the probability of a number is inversely proportional to its magnitude. At least, it must hold that $p_i \rightarrow 0$ as $i \rightarrow \infty$, because $\Sigma p_i = 1$. We shall concentrate on finite alphabets.

4.1. Shannon-Fano coding

The theory of Section 3 gives a symbol (s_i) with probability p_i the information $\log_2(1/p_i)$ bits. An optimal coding solution would give the symbol a code of this length. However, the length is an integer only if p_i is a negative power of 2. If this is not the case, we can take as the actual length the next larger integer, i.e. $\lceil \log_2(1/p_i) \rceil$. It is easy to see that this choice satisfies the Kraft inequality:

$$l_i \geq \log_2(1/p_i) \Rightarrow 2^{l_i} \geq 1/p_i \Rightarrow p_i \geq 1/2^{l_i} \Rightarrow \sum p_i \geq \sum(1/2^{l_i}) \Rightarrow 1 \geq \sum(1/2^{l_i})$$

Thus, there is a prefix code with lengths $\lceil \log_2(1/p_i) \rceil$ for i = 1, ..., q. Since every codeword length is at most one larger than the theoretical optimum, it also holds that the average code length L satisfies:

$$H(S) \leq L \leq H(S) + 1$$

Example. Let $p_1 = p_2 = 0.3$, and $p_3 = p_4 = p_5 = p_6 = 0.1$. From $\log_2(1/0.3) \approx 1.737$ and $\log_2(1/0.1) \approx 3.322$, we get the code lengths {2, 2, 4, 4, 4, 4}. The codes can be assigned in lexicographic order for the list of lengths sorted into ascending order:

$s_1 = 00$, $s_2 = 01$, $s_3 = 1000$, $s_4 = 1001$, $s_5 = 1010$, $s_5 = 1011$

which has the average code length of **2.8**, while the entropy is $H(S) \approx$ **2.371**. If we study the decoding tree, we notice that it is not complete (not *succinct*): The subtree 11... is missing. Checking the Kraft inequality confirms the property (the sum is not 1):

$$\sum_{i=1}^{6} \frac{1}{2^{l_i}} = \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^4} + \frac{1}{2^4} + \frac{1}{2^4} = \frac{3}{4} < 1$$

The situation is more obvious if we take a binary source alphabet with $p_1 = 0.99$ and $p_2 = 0.01$, for which the above rule gives code lengths $l_1 = 1$ and $l_2 = 7$, although the only sensible possibility is to choose both lengths to be = 1 (corresponding to codes 0 and 1). Nevertheless, for lengths 1 and 7, the average code length would yet be only 0.99×1 + 0.01×7 = 1.06, while the entropy is

$$-0.99 \cdot \log_2 0.99 - 0.01 \cdot \log_2 0.01 \approx 0.08 \text{ bits}$$

The above description is the theoretical basis for Shannon-Fano coding (presented independently by Shannon & Weaver and Fano in 1949), but the actual algorithm is more constructive, and also avoids non-complete decoding trees. Its idea is to divide the 'probability mass' (with related symbols) repeatedly into two approximately same-sized halves, and simultaneously construct the decoding tree in a top-down fashion. With no loss of generality, we assume here that probabilities $\{p_1, ..., p_q\}$ are in *descending* order. *Sorting* may thus precede the following codebook generation algorithm:

Algorithm 4.1. Shannon-Fano codebook generation.
Input: Alphabet $S = \{s_1, ..., s_q\}$ with probability distribution $P = \{p_1, ..., p_q\}$, where $p_i \geq p_{i+1}$.
Output: Decoding tree for S.
begin

Create a root vertex r and associate alphabet S with it.
If S has only one symbol then return a tree with r as the only node (root).
Choose index j ($\neq 0$ and $\neq q$), for which the sums $\sum_{i=1}^{j} p_i$ and $\sum_{i=j+1}^{q} p_i$ are the closest.
Find decoding trees, rooted by r_1 and r_2, for the sub-alphabets $\{s_1, ..., s_j\}$ and $\{s_{j+1}, ..., s_q\}$ recursively and add them as subtrees of r. Attach labels 0 and 1 to the corresponding parent-child relationships $r \rightarrow r_1$ and $r \rightarrow r_2$.
Return the tree rooted by r.

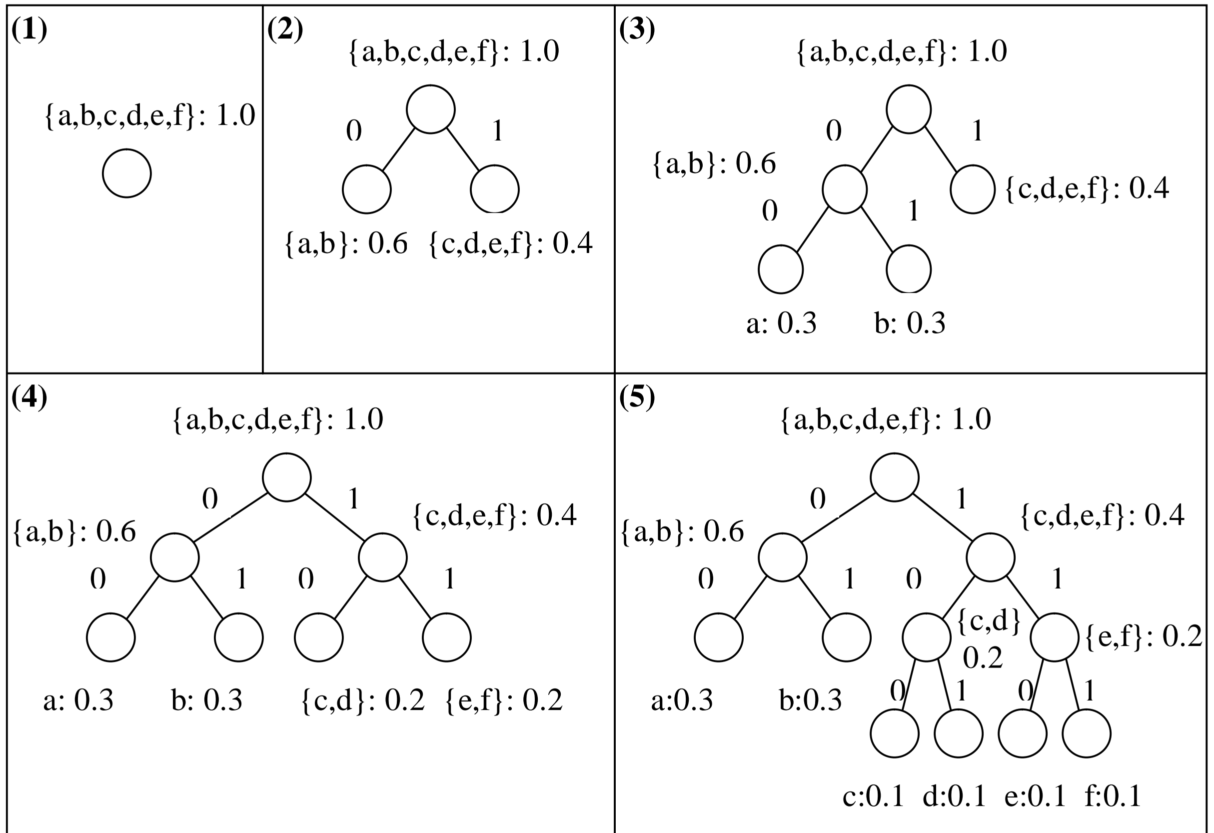
end

It can be proved that the same properties hold for this code as for the previous code based on direct computation of the codeword lengths, i.e. the redundancy is at most 1. To be precise, the selection of an optimal split index j is not trivial (generally, optimal partitioning is NP-hard). In practice the suggested condition (based on closest sums) is sufficient.

Example. $S = \{a, b, c, d, e, f\}$, $P = \{0.3, 0.3, 0.1, 0.1, 0.1, 0.1\}$. The codebook generation proceeds step by step, as shown in the figure on the next page, with subsets of symbols and related sums of probabilities marked for nodes.

The first split is done into probability subsets {0.3, 0.3} and {0.1, 0.1, 0.1, 0.1}, attached with bits 0 and 1 (to be prefixes for the codes in the subsets). The former is split again, and produces singleton subsets {0.3} and {0.3} with final codes 00 and 01 for a and b. Subset {0.1, 0.1, 0.1, 0.1} is split first into {0.1, 0.1} and {0.1, 0.1}, producing prefixes (= paths from the root) 10 and 11. The final splits produce four singleton subsets, each with probability 0.1. The related codes are 100, 101, 110 and 111. For the codebook {a=00, b=01, c=100, d=101, e=110, f=111} the average code length is **2.4** bits, having redundancy $\approx$ 0.029 bits per symbol. In fact, the obtained code is optimal (among methods assigning distinct codewords for symbols), although generally Shannon-Fano code does not guarantee optimality.

The tree structure is suitable for the decoder: by branching according to the received bits it is easy to pick up the correct original symbol from the leaf. The encoder, instead, needs some other data structure, presumably a table with an entry for each symbol and the related codeword.



4.2. Huffman coding

The best-known source coding method is without doubt Huffman coding. The generation of a Huffman code resembles the Shannon-Fano method, but the direction is opposite: Shannon-Fano generates the decoding tree top-down, while Huffman method does it *bottom-up*. The average codeword length of the Huffman code reaches the entropy if the probabilities are integer powers of ½ (cf. Chapter 3). It also holds that, even in cases where the entropy is not reached, Huffman coding is optimal among the methods that assign distinct codewords for separate, independent symbols of the source alphabet.

The idea of Huffman code is developed as follows. First assume, without loss of generality, that the probabilities of symbols are sorted into decreasing order: $p_1 \geq p_2 \geq ... \geq p_q$. Obviously, this implies that the codeword lengths must satisfy: $l_1 \leq l_2 \leq ... \leq l_q$. Since all non-root nodes in an efficient (complete) decoding tree have a sibling, and symbol s_q is on the lowest level (leaf), also s_{q-1} should be a leaf, more precisely, the sibling of s_q, and have the same codeword length. (The choice may not be unique, if there are equal probabilities, but then we can make an arbitrary choice.) Therefore, we can assign *suffixes* 0 and 1 to the last two symbols; their final codewords will be of the same length and differ only in the last bit position. The pair can be considered a single 'meta-symbol' (= sub-alphabet $\{s_{q-1}, s_q\}$), which has probability $p_{q-1} + p_q$, and becomes the parent of the last two symbols. From this we can continue by repeating the process for the reduced alphabet of q-1 symbols, and attaching new parents to already generated subtrees. More formally, the algorithm proceeds as follows:

Algorithm 4.2. Huffman codebook generation.

Input: Alphabet $S = \{s_1, ..., s_q\}$ with probability distribution $P = \{p_1, ..., p_q\}$, where $p_i \geq p_{i+1}$.
Output: Decoding tree for S.
begin
Initialize forest F to contain a distinct single-node tree T_i for each symbol s_i and set $weight(T_i) = p_i$.
while $|F| > 1$ **do**
begin
Let X and Y be two trees in the forest which have the lowest weights.
Create a binary tree Z, with X and Y as subtrees (equipped with labels 0 and 1).
The root of Z is a new node representing the combined symbol set of X and Y.
Set $weight(Z) = weight(X) + weight(Y)$.
Add Z to forest F and remove X and Y from it.
end
Return the single remaining tree of forest F.
end

The resulting code is a prefix code, because the original symbols will be leaves of the final tree. The proof of optimality is skipped here, but essentially we proved it already in the above derivation. As Huffman coding is optimal, and Shannon-Fano satisfies

$$H(S) \leq L \leq H(S) + 1$$

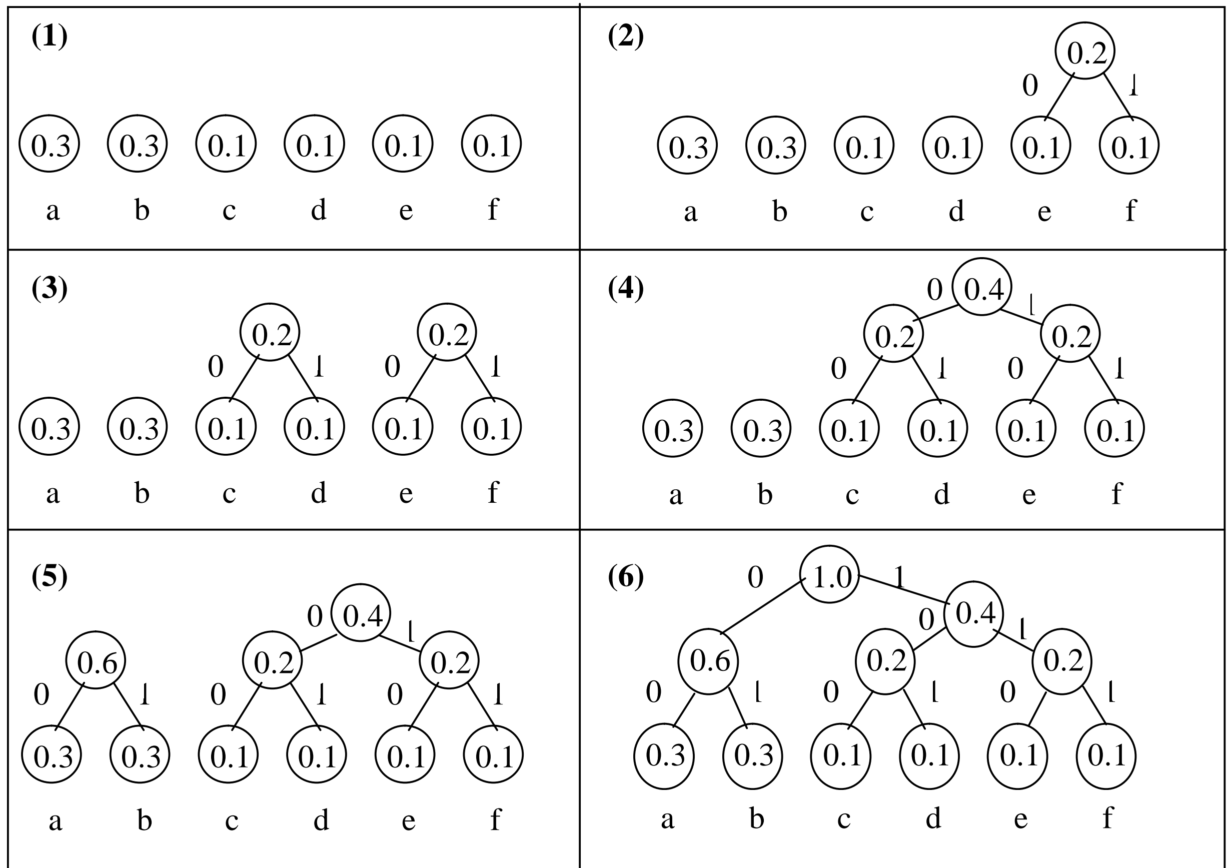
then also Huffman code satisfies it. In fact, it has been proved that an upper bound for the redundancy is $p_1 + 0.086$, where p_1 is the highest probability among the symbols. To be precise, the upper bound is $\text{Min}(p_1 + 0.086, 1)$.

The generated codebook is not unique, if there occur equal probabilities during the process; ties in choosing the two smallest weights can be broken arbitrarily, without affecting the average length L of codewords. However, if we always favour the smallest trees, we obtain a code, the longest codeword of which is minimal. This may have practical advantage. When we still notice that, in each combining step, 0 and 1 can be assigned to subtrees in either way, it is clear that the number of different optimal codebooks is exponential.

Example. Determine the Huffman code for alphabet $S = \{a, b, c, d, e, f\}$ where $P = \{0.3, 0.3, 0.1, 0.1, 0.1, 0.1\}$. Forest F develops in the steps shown in the picture on the following page. We obtain the same codebook {00, 01, 100, 101, 110, 111} as with Shannon-Fano method, which in this case is also optimal. Technically the Huffman algorithm can be implemented e.g. in the following ways:

(1) Build and maintain a *heap* (more precisely, a min-heap) structure for the symbols (both original and meta-symbols), ordered by weight. The heap can be built in linear time using *Floyd's algorithm.* Extracting the smallest and adding a new meta-symbol need both $O(\log q)$ operations for q nodes, and thus the total complexity is $O(q \log q)$.

(2) If the alphabet is already sorted by probability, we can build the tree in linear time by keeping a *queue* of the created metasymbols: they will be generated in increasing order of weight. The two smallest weights are thus found either among the sorted list of original symbols or from the queue of metasymbols.



There are some special probability distributions worth looking at. First, if all symbols are equally likely, and if the number of symbols q is a power of two, we get a perfectly balanced binary tree and a block code with each codeword having length $\log_2 q$. If symbols are equally likely but q is not a power of two, we obtain a shortened block code, where the lengths differ by at most one. A somewhat weaker condition is sufficient for this: If the sum of two smallest probabilities ($p_{q-1} + p_q$) is greater than the biggest probability (p_1), then we also end up to a (shortened) block code. This is seen immediately from technique (2) above, because all actual symbols are combined before any of the metasymbols, and the metasymbols are combined in the order they were generated.

Another extreme is the (negative) exponential distribution $p_i = 1/2^i$, the Huffman tree of which is a degenerated tree where each internal node has a leaf child (two leaves on the lowest level). This kind of code is similar to unary code (11...10), except for one codeword that has all its bits = 1.

A distribution that often occurs in practice, e.g. for symbols in natural language, is the *Zipf distribution*, where the i'th symbol in the alphabet, ordered in decreasing order of probability, has probability proportional to $1/i$. Since the sum of $1/i$ is not limited when i approaches infinity, the Zipf distribution is valid only for finite alphabets. A typical compression result is about 5 bits per character.

In static compression, the Huffman code can be determined once for all inputs. In semi-adaptive coding, however, the code is determined for each input message separately. Therefore, in transmitting the message to the decoder, one has to send *both* the codebook *and* the encoded message. This reduces the compression gain but, having a finite alphabet, the loss is negligible for long messages. Anyway, it is worthwhile to consider the compression of the

codebook (= decoding tree) itself. The shape of the tree can be expressed with $2q$-1 bits: The whole tree contains always this number of nodes, and for each node we must tell whether it is a leaf or not. In addition, we have to transmit the order of symbols in the leaves from left to right. In total, the representation amounts to $2q-1 + q\lceil \log_2 q \rceil$ bits. It is also possible to express the codebook by transmitting only the *lengths* of codewords (in alphabetic order), or *counts* of different lengths plus symbols in probability order. We shall return to the latter alternative later.

4.2.1. Extended Huffman code

For large alphabets, with the largest probability not too big, Huffman coding is generally efficient. However, for very small alphabets with a skew distribution it can be rather bad, compared to the entropy. For example, if we have $S = \{a, b, c\}$ with $P = \{0.8, 0.18, 0.02\}$, the optimal codebook will be {0, 10, 11}, with average codeword length = 1.2 bits, while the entropy is only 0.816 bits. The redundancy (0.384 bits per character) can be reduced by extending the alphabet S to $S^{(n)}$, consisting of n-symbol blocks, called *n-grams*. Their count is q^n, and the probability of each can be computed as the product of elementary probabilities. Then we can construct the Huffman code for the extended alphabet. As before, we have the following condition for the average codeword length: $H(S^{(n)}) \leq L^{(n)} \leq H(S^{(n)}) + 1$. Therefore, the number (L) of bits per original symbol is bounded as follows:

$$H(S^{(n)})/n \leq L \leq (H(S^{(n)}) + 1)/n \;\Rightarrow\; H(S) \leq L \leq H(S) + 1/n$$

In the limit, we get arbitrarily close to the entropy. In the above example, by taking pairs of symbols (aa, ab, ac, ba, bb, bc, ca, cb, cc) as the new coding alphabet ($n = 2$), the average codeword length will reduce to 0.8758 bits, having a redundancy of only 0.06 bits per symbol.

Of course, the size of the codebook, as well as the effort of computing it, will explode when n gets larger. This restricts the applicability of the method. Moreover, many of the n-grams will not even occur in the actual message. It should somehow be possible to compute the needed codes on the fly, without building an explicit decoding tree. This idea will be used later, especially in *arithmetic coding*.

4.2.2. Adaptive Huffman coding

Normal Huffman coding needs knowledge of the symbol probabilities before starting the encoding. This means that the compression is done off-line, in two passes. If this is not applicable, we should develop a modification where the code adapts to the probabilities symbol by symbol, while the message is being transmitted. Then the decoder can maintain the same code synchronously with the encoder, and there is no need to transmit the decoding tree.

A naive solution would be to start with some default, e.g. uniform distribution, maintain symbol frequencies, and compute the whole code repeatedly; once per each transmitted symbol. Although this is in a sense optimal, it is clearly too laborious. It should be noticed that small local changes in the symbol probabilities do not usually cause global changes in the Huffman tree structure. We shall take advantage of this observation. In what follows we talk about symbol and node *weights* (corresponding to *frequencies*), instead of probabilities. This

has no effect on the resulting code, only the scaling of numbers is different. Weights are thus integers, and a parent has always a weight which is the sum of the weights of its children. A Huffman tree has the ***sibling property*** if the following two conditions hold:

(1) Each node, except the root, has a sibling (i.e. the binary tree is complete).
(2) The tree nodes can be listed in non-decreasing order of weight in such a way that each node is adjacent to its sibling in the list.

Theorem. A binary tree with weights associated with its nodes, as defined above, is a Huffman tree if and only if it has the sibling property.

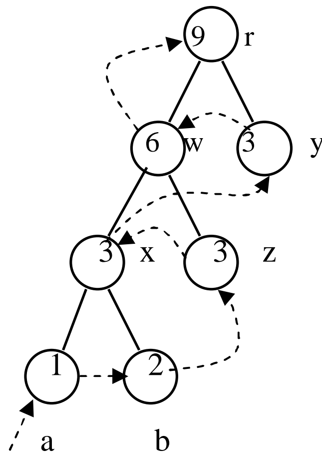
The proof is skipped. Obviously, it must be possible to swap nodes during the course of adaptation, as the order of weights (frequencies) changes. To do this, we maintain a *threaded list of nodes* in non-decreasing order. In this list, the adjacent nodes with equal weights are called a *block*. When, during encoding, a new symbol x has been transmitted, the tree must be updated: The related leaf weight is increased by one. Now it may happen that the node is not anymore in the correct place. Therefore, it is swapped with the *rightmost* node y of the block; the next block must have weights at least as large as the increased value, so the order is correct. If $x = y$ then x was the only node in its block, and no swapping is required. Next assume that $x \neq y$ and denote the sibling of x by z, and the sibling of y by w. Now we check that the sibling property holds. Since before the update $weight(x) = weight(y)$, then after the swap the sibling property holds for y and z. For w, two cases can be distinguished:

1) $weight(x) = weight(w)$ before the update, and the order of nodes is $x...wy...$ After the swap and increase of $weight(x)$ it holds: $weight(w) < weight(x)$ and the order is $y...wx...$ Thus the sibling property holds.

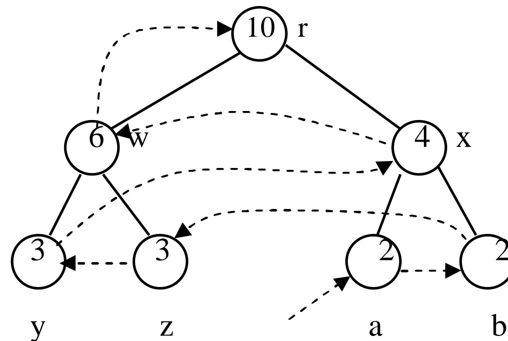
2) $weight(x) < weight(w)$ before the update, and the order of nodes is $x...yw...$ After the swap and increase of $weight(x)$ it holds: $weight(x) \leq weight(w)$ and the order is $y...xw...$ Again, the sibling property holds.

Notice also that in swapping, the possibly attached subtrees are carried along. The procedure continues to the parent of x, the weight of which is also increased by one, and the same test plus possible swap are done. The process stops at the root.

The following example shows the evolution (including swap $x \leftrightarrow y$) of a sample Huffman tree, after increase of $weight(a)$. Node a maintains its position, but the increase of the weight of its parent x from 3 to 4 makes it out of order, and relocation of x follows, whereby its children a and b move along.



List: (a, b, z, x, y, w, r)



List: (a, b, z, y, x, w, r)

A modification to the above algorithm has been suggested, where one starts with an empty alphabet, and a tree with only one node as a placeholder. As new symbols are encountered during the message, the placeholder splits and the new symbol occupies the other half. Of course, the code of the placeholder, plus the ASCII (or corresponding) code of the new symbol must be transmitted to the decoder.

The swap can be made in constant time, but finding the rightmost node in the block of equal weights can be more time-consuming. This can be improved by maintaining a second list for the last nodes of each block of equal-weighted nodes. Each tree node must contain a pointer to the related block node, from which we directly find the last node of the block. If the block list is two-way linked, it can be updated in constant time per one weight increase. Thus, the overall complexity is proportional to the depth of the related leaf, and also proportional to the number of output bits.

It can be proved that the compression result is at most $2h + n$ bits, where h is the length of the corresponding result from static two-phase Huffman coding, and n is the length of the source message. In practice, the lengths of the two codes are very close to each other.

In the form presented, adaptive Huffman coding does not adapt well to local changes in the probability distribution within a (long) message. Some researchers have suggested modifications, where all the weights are periodically changed, e.g. nullified or halved, so that old symbol occurrences will have less weight than the new ones.

Adaptive Huffman coding has been used also in practice: The Unix *compact* command is based on it. Its practical advantage is, however, limited by the fact that in many modelling techniques for the source, a separate decoding tree is needed for each possible *context*. For 2- or 3-memory sources for ASCII characters, the number of trees (plus related lists) becomes huge. Arithmetic coding, to be described later, is much more applicable to adaptive coding.

4.2.3. Canonical Huffman code

The *decompression* speed is usually more important than compression speed. In Huffman decoding, the normal procedure is to follow a path in the decoding tree from the root to the correct leaf. At each node, we have to inspect the transmitted bit and make a choice between the children. For large alphabets (consisting of multi-character words, for example), the size of the tree will be quite large, and the probing may become inefficient. *Canonical Huffman coding* is a scheme which avoids the explicit storage of the tree, and has much better decoding speed.

As mentioned before, the number of alternative Huffman codebooks is exponential, due to tie situations and alternatives in bit allocation. Canonical Huffman coding will use the same *lengths* of codewords for the symbols as normal Huffman coding, but may assign the bits in a different, systematic way. The following table shows three possible Huffman codes for a six-symbol alphabet, with given frequencies.

Symbol	Frequency	Codebook I	Codebook II	Codebook III
a	10	000	111	000
b	11	001	110	001
c	12	100	011	010
d	13	101	010	011
e	22	01	10	10
f	23	11	00	11

Codebooks I and II are generated with the normal Huffman technique; their difference is just in the assignment of bits: the codes are complements of each other. Codebook III is, however, different. Although it is a prefix code and optimal (the codeword lengths being equal to those in codebooks I and II), it is not possible to generate it with the Huffman algorithm. Thus, we suggest the following relaxed definition:

> *A Huffman code is any prefix-free assignment of codewords, where the lengths of codewords are equal to the depths of corresponding symbols in a Huffman tree.*

What is so special in codebook III is its nice ordering of codeword values: The 3-bit codewords are in increasing sequence, and the same holds for 2-bit codewords. Moreover, if we discard the last bit from 3-bit codewords, the resulting sequence of 2-bit codewords is still non-decreasing. This property enables simple decoding: If the integer value of the first two bits is 2 or 3, the codeword has been uniquely identified. If the value is 0 or 1, we know that the two bits are a prefix of a 3-bit codeword, so the next bit is read, and the corresponding numerical value (0..3) computed, which uniquely identifies the related symbol.

In a general case, we do similarly as above. First we have to determine the length of each codeword by using the Huffman algorithm. Then we compute the number of codewords per each length from 1 to *maxlength* (= maximum depth of the Huffman tree). Using these we can determine the *firstcode* value for each length, whereafter the symbols can be assigned their codewords one by one. A more formal algorithm is given as follows.

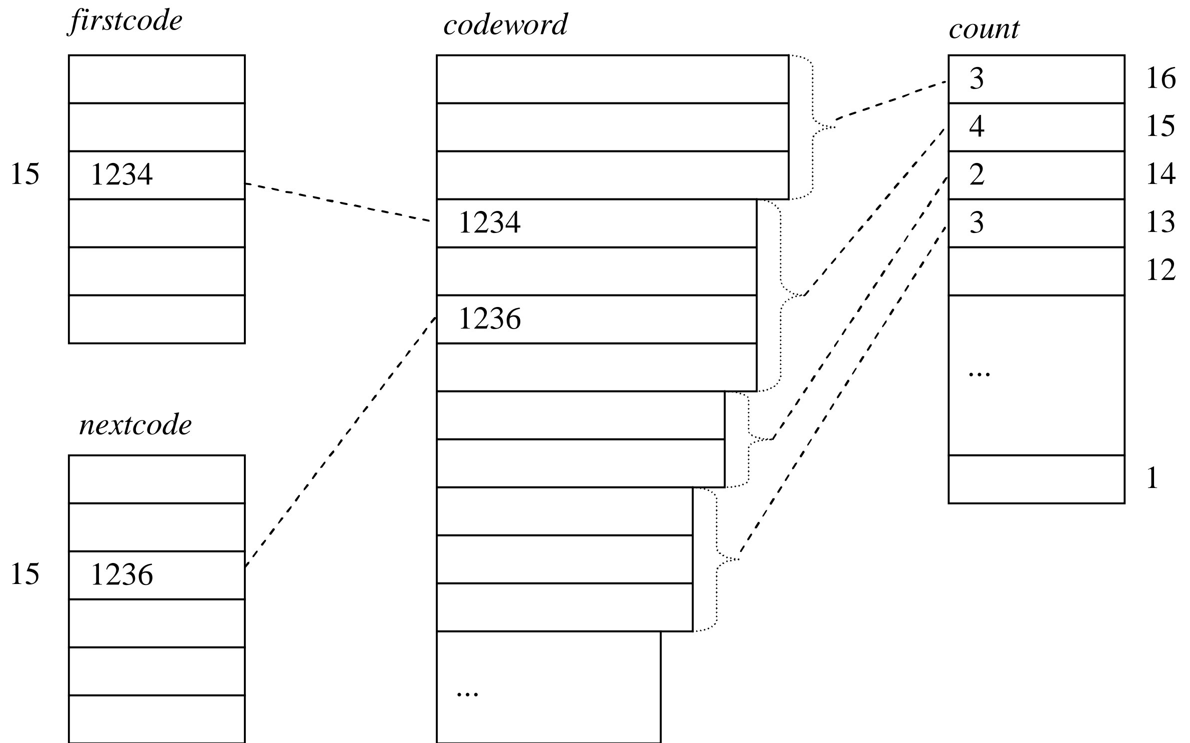
Algorithm 4.3. Assignment of canonical Huffman codewords.
Input: Length l_i for each symbol s_i of the alphabet, determined by the Huffman method.

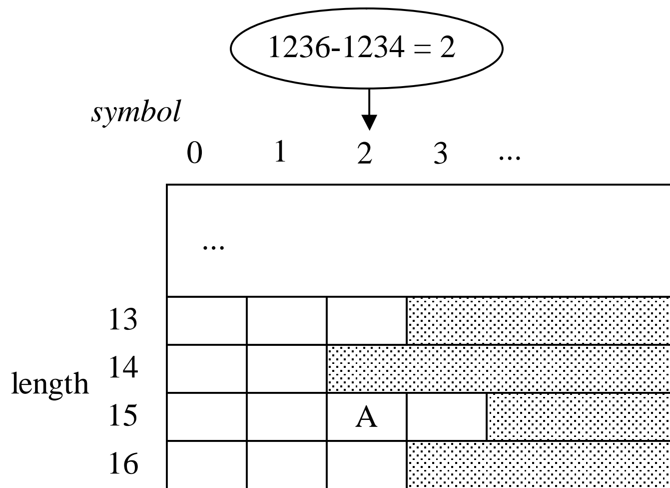
Output: Integer values of codewords assigned to symbols, plus the order number of each symbol within same-length symbols.

```
begin
    Set maxlength := Max{l_i}
    for l := 1 to maxlength do
        Set countl[l] := 0

    /* Count the number of codes for each length */
    for i := 1 to q do
        Set countl[l_i] := countl[l_i] + 1

    /* Compute the first code for each length, starting from longest      */
    /* Shifting to one bit shorter means division of the next code by 2 */
    Set firstcode[maxlength] := 0
    for l := maxlength − 1 downto 1 do
        Set firstcode[l] := (firstcode[l+1] + countl[l+1] ) / 2

    /* Initialize the assignment of codewords; nextcode[l] loops within length l. */
    for l := 1 to maxlength do
        Set nextcode[l] := firstcode[l]

    /* Assign the codewords to characters and create the symbol table for decoding */
    for i := 1 to q do
    begin
        Set codeword[i] := nextcode[l_i]
        Set symbol[l_i, nextcode[l_i] − firstcode[l_i] ] := i
        Set nextcode[l_i] := nextcode[l_i] + 1
    end
end
```

The two-dimensional array *symbol* is used in decoding, which is based on the fundamental property of the canonical Huffman code: All k-bit prefixes of an l-bit codewords are always strictly smaller (of their numerical value) than any k-bit codeword. By inspecting the transmitted bit sequence, we take one bit at a time until the numerical value of the bit string is larger than the *firstcode* of the respective length. Then we have reached a legitimate code, and the related symbol can be looked up from the created *symbol* table.

The following picture illustrates the data structures used. Data about an example symbol 'A' with code 1236 having length 15 is given. The code is supposed to be the 3rd in the group of codes of length 15. The situation is just after assigning the character *i* to the *symbol* table.





Algorithm 4.4. Decoding of canonical Huffman code.

Input: The numerical value of the first code for each codeword length, plus the symbol for each order number within the set of codewords of equal length.

Output: Decoded symbol.

begin

set *value* := **readbit**( ); set l := 1

while *value* < *firstcode*[l] **do**

begin Set *value* := 2 ∗ *value* + **readbit**()

Set $l := l + 1$

end

return *symbol*[l, *value* − *firstcode*[l]]

end

The loop is extremely fast; multiplication by 2 can be implemented by shift-left. The memory consumption of the decoding process is quite small: The *firstcode* table has an entry only for each different length. The *symbol* array is larger; it is shown as a 2-dimensional array, but the actual implementation can be an array of arrays, requiring only q entries, plus cumulative counts for each length.

4.3. Tunstall code

All the previous coding methods were of the type 'fixed-to-variable', i.e. a fixed number (often 1) of source symbols were mapped to bit strings of variable length. Now we take the opposite approach, namely map *blocks* of a variable number of source symbols into fixed-length codewords. Later we shall study many practical algorithms belonging to this category ('LZ78 family', among others), which gather frequent substrings of the source message as coding units. Here we restrict ourselves to pure source coding, i.e. we assume that only the symbol probabilities to be given, not probabilities of substrings. The probability of a substring $<s_{i1}s_{i2}...s_{ik}>$ is thus computed as the product $p_{i1}p_{i2}...p_{ik}$ of probabilities of the (independent) symbols involved.

When using fixed-length codewords, the optimum selection for the extended alphabet would be such where the substrings are equiprobable (i.e. have uniform distribution). In practice, we may only approximate the optimum situation. Of course, an additional condition is that any source message can be parsed into the substrings of the extended alphabet. Tunstall has developed a technique (1968), which produces the required codebook. It can be characterized as 'reverse-Huffman' coding, since the two are in a way dual methods. Note also the resemblance to Shannon-Fano coding.

The Tunstall method builds a *trie*, where each branch is labelled with a source symbol, and each node has a weight equal to the product of probabilities of symbols on the path to the root. For simplicity, we restrict the discussion to prefix-free substrings, i.e. the trie must be complete, so that every non-leaf node has precisely q children; one for each symbol of the source alphabet. This enables extremely simple encoding: we just follow the path of branches where the labels correspond to those in the source message. The fixed-length codeword is attached to the leaf. Decoding is still simpler: The transmitted codewords can be used as indices to a lookup table of related strings. In the following, we denote the codeword length by k. Thus, the number of different codewords is 2^k, and we should assign them to the leaves of the trie.

The trie is built top-down, by starting from a 1-level trie with leaves for all symbols. Then we repeatedly extend the trie by adding q children for the leaf having the largest probability, until the number of leaves grows beyond 2^k-q+1. Formally presented, the codebook generation is done as follows.

Algorithm 4.5. Tunstall codebook generation.

Input: Symbols s_i, $i = 1, ..., q$ of the source alphabet S, symbol probabilities p_i, $i = 1, ..., q$, and the length k of codewords to be allocated.

Output: Trie representing the substrings of the extended alphabet, with codewords $0, ..., 2^k-u$ attached to the leaves ($0 \le u \le q-2$), plus the decoding table.

begin

Initialize the trie with the root and q first-level nodes, equipped with labels $s_1, \ldots s_q$, and weights $p_1, \ldots, p_q$.

$n := 2^k - q$ -- Number of remaining codewords

while $n \geq q - 1$ **do**

Find leaf x from the trie having the biggest weight among leaves.

Add q children to node x, equipped with labels $s_1, \ldots s_q$, and weights $weight(x) \cdot p_1, \ldots, weight(x) \cdot p_q$.

Set $n := n - q + 1$

end

for each leaf l_i in preorder **do**

Assign $codeword(l_i) := i$.

Denote $path(l_i) =$ sequence of labels from the root to l_i.

Add pair $(i, path(l_i))$ to the decoding table.

end

end

If $q > 2$, it may happen that some codewords are left unused. More precisely, the number of unused codewords is

$$u = 2^k - (q-1)\left\lfloor \frac{2^k - 1}{q-1} \right\rfloor - 1$$

We could use them by extending some leaf only partially, but that would destroy the prefix-free property of the trie.

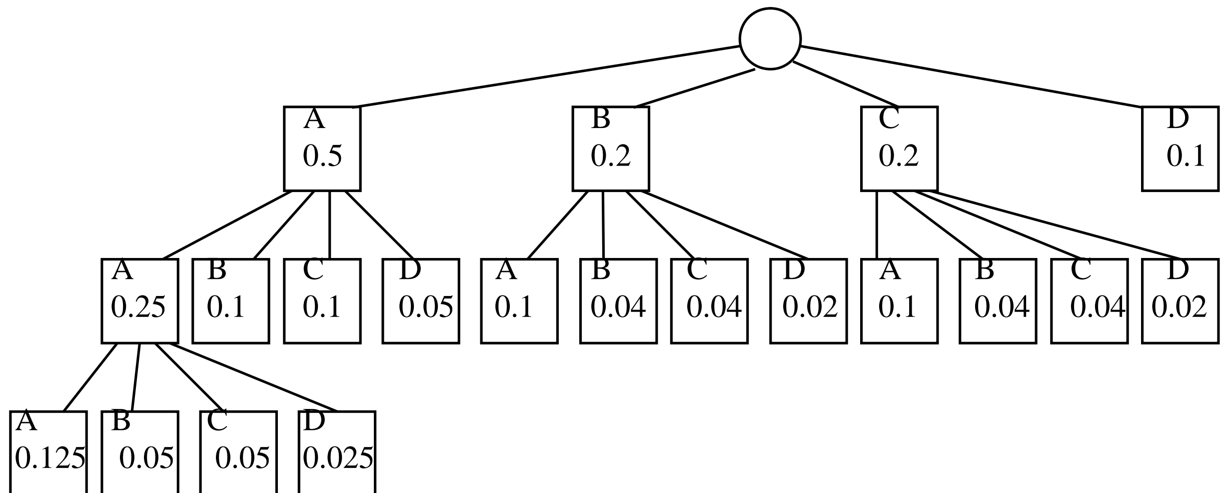
When implementing the algorithm, we obviously need a *priority queue* to effectively find the largest weight among the leaves. By using a *heap* for this purpose, the codebook generation takes time $O(2^k + k \cdot 2^k/q)$, because the depth of the heap is $O(k)$, but we need the heap only for the $O(2^k/q)$ internal nodes. The complexity also includes the assumption that the fixed-length codewords can be manipulated with constant-time operations, i.e. $k <$ number of bits in a machine word. This is a rather non-restrictive assumption, because the size of the decoding table can hardly be larger than, say, 2^{32} entries. When walking the final trie in preorder, we naturally maintain a *stack* of the ancestors of the current node, so that the path is easily determined.

The last substring of a message may represent an incomplete path of the trie, but this is easily handled by taking any complete path, and sending the message length before the coded message. This is possible, because we assume a semi-adaptive coding environment.

The average number of bits consumed by an input symbol is

$$\frac{k}{\sum_{path \in Trie} (P(path) Length(path))}$$

Example. Let S = {A, B, C, D}, P = {0.5, 0.2, 0.2, 0.1}, and k = 4.



The decoding table for this trie is as follows:

0000 → AAA	0100 → AB	1000 → BB	1100 → CB
0001 → AAB	0101 → AC	1001 → BC	1101 → CC
0010 → AAC	0110 → AD	1010 → BD	1110 → CD
0010 → AAD	0111 → BA	1011 → CA	1111 → D

All possible codeword values were used in this case, because 2^k-1 = 15 is divisible by q-1 = 3. The average number of bits per source symbol is $4/(0.25\cdot3+0.65\cdot2+0.1\cdot1) \approx 1.86$, whereas the entropy is $H(S) \approx 1.76$ bits per symbol. One possible Huffman code for this example would be A=0, B=10, C=110, and D=111, which results in 1.80 bits per character; somewhat better than with Tunstall coding.

It should be noticed that the Tunstall trie is optimal among prefix-free tries. In chapter 2 we noticed that in *decoding* of variable-length codes, the prefix-free property was invaluable, because it guaranteed unique decoding. Here in *encoding*, on the other hand, the prefix-free property is not mandatory: It is also possible to assign codeword values to internal nodes, and allow the number of children to be $< q$. At each step of encoding, we can take the longest substring of the remaining part of the source message, so that the substring matches a path in the trie, and send the attached codeword. This modification may improve the compression gain somewhat, but it makes codebook generation more complicated, and slows down the encoding process.

4.4. Arithmetic coding

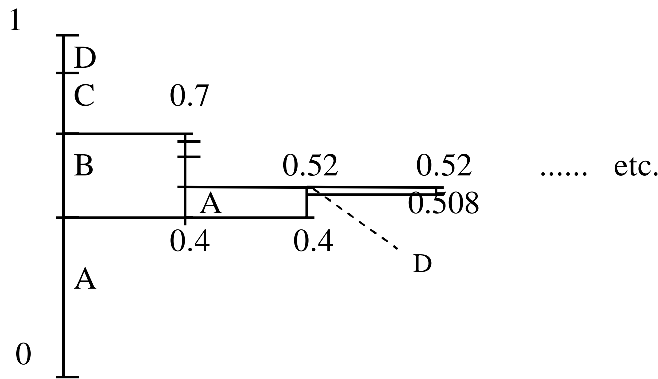
One of the most popular source coding algorithms (in addition to Huffman code) in modern compression methods is *arithmetic coding*. It is in most respects better than the traditional Huffman coding: It *reaches the information-theoretic lower bound* (within used precision), i.e. entropy, and is very flexible in both static and adaptive coding situations. It represents a pure separation of source coding from building and maintaining the model of the source. Therefore, it is used in many of the best compression programs as the final coding technique. With respect to compression performance, it is superior to Huffman coding especially for small alphabets and skewed probability distributions. It is computationally relatively efficient, although usually somewhat slower than Huffman coding. There are, however, faster suboptimal rearrangements of the basic method, especially for binary source alphabets.

The origins of arithmetic coding can be traced back to the pioneers of information theory and source coding, namely Shannon and Elias, who invented the basic ideas about 50 years ago. However, modern arithmetic coding was discovered 1976, independently by two researchers (Pasco and Rissanen). Since then, several papers have appeared describing the practical implementation of arithmetic coding. Among the many, we mention here the paper written by I.H. Witten, R.M. Neal, and J.G. Cleary: "Arithmetic Coding for Data Compression", in Communications of the ACM, Vol. 3, No. 6, 1987, pp. 520-540. This paper contains also a detailed C-code of a simple implementation.

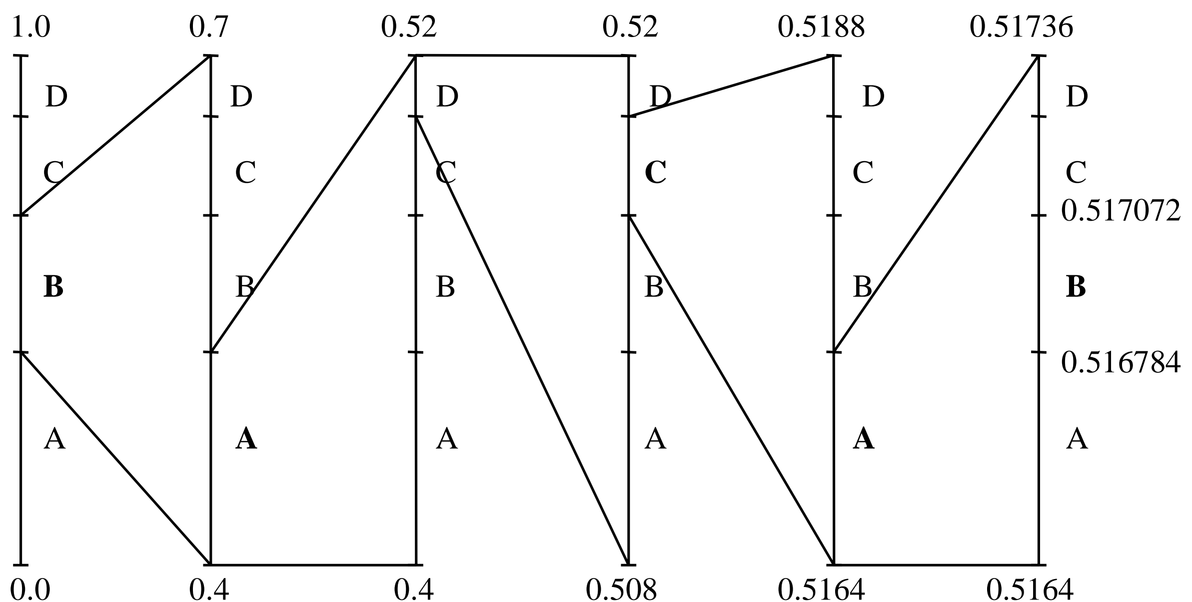
Arithmetic coding is hard to classify among other coding methods. In principle, it generates only a single codeword for the whole message. Therefore, it can be regarded as an extreme case of extended Huffman code, except that arithmetic coding does not create any explicit codebook. The size of the codebook for extended Huffman is exponential in the block size, and becomes prohibitive for large blocks. It is peculiar to arithmetic coding that there is no clear correspondence between symbols of the source message and bits of the code; we cannot say that a certain bit resulted from a certain symbol, as is the case with Huffman code.

The key idea of arithmetic coding is to represent the message by an *interval of real numbers between* 0 *and* 1. Longer messages are represented with smaller intervals (and higher precision) than shorter messages, and vice versa. The size of the interval should be in accordance with the probability of the message. The encoding process starts with the whole [0, 1) interval, and successive symbols of the message reduce the interval size in proportion to their probabilities. At an arbitrary position, the interval is partitioned into subintervals, one for each symbol of the alphabet, and then the next symbol of the message determines, which subinterval is selected. Notice that the intervals are half-open; the upper bound value belongs always to the next higher subinterval.

In the following example, the alphabet is {A, B, C, D} with given probabilities {0.4, 0.3, 0.2, 0.1}, and the message to be encoded is "BADCAB". First, the starting interval [0, 1) is divided into four subintervals: [0, 0.4), [0.4, 0.7), [0.7, 0.9), [0.9, 1), of which we select the second one, corresponding to the first letter 'B' of the message. Interval [0.4, 0.7) is then divided again into four intervals in the proportion of the original symbol probabilities: [0.4, 0.52), [0.52, 0.61), [0.61, 0.67), [0.67, 0.7). The next message symbol is A, so we select the first subinterval. This process is repeated up to the last symbol of the message; the final interval is [0.516784, 0.517072). The process can be graphically described as follows.



Notice how each succeeding interval is contained in the preceding interval, the final interval size being equal to the product of probabilities of the message symbols. For visibility, the intervals can be rescaled for each subinterval as follows:



After computing the final interval, *any* real value from that interval is a valid representative of the whole message. For simplicity, we could take the lower bound 0.516784 of the interval as the code. The *midpoint* of the interval is another popular choice, and we shall use it in our analysis of the method.

Decoding is as simple as encoding, and proceeds as follows: From the first division into subintervals, we notice that the value 0.516784 belongs to the second one corresponding to symbol B. The interval is thus reduced to [0.4, 0.7), *similarly as in encoding*. Then we compute the next division into four subintervals, and notice that the value 0.516784 belongs to the first one, corresponding to A, namely [0.4, 0.52), and so on. The encoder and decoder create the same hierarchical (nested) division, only the basis for choosing the correct subinterval at each step is different (encoder: symbol at hand; decoder: code value).

In the following tentative algorithms, we assume that the symbols of our alphabet are numbered from 1 to q, and the numbers are used instead of the actual symbols.

Algorithm 4.6. Arithmetic encoding.
Input: Sequence $x = x_i$, i=1, ..., n; probabilities p_1, ..., p_q of symbols 1, ..., q.
Output: Real value between [0, 1) that represents X.

```
begin
    cum[0] := 0
    for i := 1 to q do cum[i] := cum[i−1] + p_i
    low := 0.0
    high := 1.0
    for i := 1 to n do
    begin  range := high − low
           high := low + range * cum[x_i]
           low := low + range * cum[x_i−1]
    end
    return (low + high) / 2
end
```

Algorithm 4.7. Arithmetic decoding.
Input: v: Encoded real value; n: number of symbols to be decoded;
probabilities p_1, ..., p_q of symbols 1, ..., q.
Output: Decoded sequence x.

```
begin
    cum[0] := 0
    for i := 1 to q do cum[i] := cum[i−1] + p_i
    low := 0.0
    high := 1.0
    for i := 1 to n do
        begin  range := high − low
        z := (v − low) / range
        Find j such that cum[j−1] ≤ z < cum[j]
        x_i := j
        high := low + range * cum[j]
        low := low + range * cum[j−1]
    end
    return x = x_1, ..., x_n
end
```

These algorithms are not applicable to practical coding as such, for several reasons: They assume arbitrary-precision real-valued arithmetic, and decoding needs the number of symbols. The major question is, what *representation* is chosen for the final code. We can take any value from the final interval, but its exact representation is not necessary, as long as the approximation is good enough for determining the correct interval. If we look at the binary representations of the endpoints of the final interval, the best choice is to take all *significant* bits of the midpoint, up to the first position which makes a distinction from both lower and upper endpoints. In the above example, the binary representations are:

- *upper*: 0.517072 = .10000100010111101...
- *lower*: 0.516784 = .10000100010010111...
- *midpoint*: 0.516928 = .10000100010101010...

A sufficient code is 1000010001010, i.e. 13 bits; the last bit is needed to ensure a prefix-free code. From compression point of view, the code takes 2.17 bits per symbol, while the entropy is only 1.846 bits per symbol. However our input message "BADCAB" did not have the distribution assumed. Instead, the message "ABCDABCABA" would have had the final interval length $0.4^4 \cdot 0.3^3 \cdot 0.2^2 \cdot 0.1 = 0.000002764$. The amount of information for this probability is $\log_2(1/0.000002764) \approx 18.46$ bits, which is *exactly the value determined by the entropy*. However, we round this value up, and add one bit position, to guarantee prefix-free property, so the final code will be 20 bits long. We give the general result formally below (proof skipped). Notice especially that we restrict the discussion here to *prefix-free messages*, so the intervals of different messages will be disjoint. We return to this issue later.

Theorem 4.2. Let *range* = *upper* − *lower* be the final probability interval in Algorithm 4.6 when encoding message x. The binary representation of *mid* = (*upper* + *lower*) / 2 truncated to $l(x) = \lceil \log_2(1/range) \rceil + 1$ bits is a uniquely decodable code for *prefix-free* messages.

Using Theorem 4.2, we can derive the average number of bits used per source symbol. Notice that $l(x)$ is the number of bits to encode the *whole* message x. The average code length of a message containing n symbols is obtained as follows.

$$\begin{aligned} L^{(n)} &= \sum P(x) l(x) \\ &= \sum P(x) \left[\left\lceil \log_2 \frac{1}{P(x)} \right\rceil + 1 \right] \\ &\le \sum P(x) \left[\log_2 \frac{1}{P(x)} + 2 \right] \\ &= \sum P(x) \log_2 \frac{1}{P(x)} + 2 \sum P(x) \\ &= H(S^{(n)}) + 2 \end{aligned}$$

where $S^{(n)}$ the extended alphabet of n-symbol blocks from S. On the other hand, we know that $L^{(n)} \ge H(x^{(n)})$. Thus we get the following bounds for the number of bits per source symbol:

$$\frac{H(x^{(n)})}{n} \le L \le \frac{H(x^{(n)})}{n} + \frac{2}{n}$$

which for independent source symbols gives

$$H(S) \le L \le H(S) + \frac{2}{n}$$

Thus, for long messages, the compression efficiency gets extremely close to the entropy. This is a major result, and with good reason arithmetic coding can be called optimal. The common conception that Huffman coding is optimal does not hold generally; only among the methods that give distinct codewords for distinct symbols of the alphabet.

In the above discussion, we assumed that the set of studied source messages was prefix-free, i.e. message pairs like "TO BE" and "TO BE OR NOT TO BE" were not accepted. The reason is that the related intervals will not be disjoint, and hence we cannot guarantee the uniqueness of the code. If we used the *lower* value as our codeword, then sequences "A", "AA", "AAA", ... would all result in the same codeword (value 0), if 'A' is the first symbol in

partitioning. There are two possibilities: either transmit the length of the message before the actual code, or extend the alphabet with an *end-of-message* symbol. The former is not useful in adaptive (1-phase) coding, so the latter is most often used. Actually, it makes the messages prefix-free. The only question is, what probability we should assign to the new symbol. Optimal would be $1/n$ for an n-symbol message, but generally n is unknown. Fortunately, an estimate of the average message length is quite satisfactory.

We now turn to the practical implementation of arithmetic coding. The following problems can be observed in the above tentative algorithms:

- The encoded message is created only when the last input symbol has been received. *Incremental* transmission and decoding would be highly desirable in practice.

- The precision of normal floating-point arithmetic is insufficient for all but the shortest messages. *Fixed-point* arithmetic should be applied, most preferably *integer* operations.

The key idea to solving the above problems is to apply *scalings* (*zoomings*, *renormalizations*) of the interval as soon as we know enough about the next bit to be transmitted. In other words, the common bits in the front of *lower* and *upper* cannot change, anymore, and can therefore be transmitted. The following cases can be distinguished:

1. If the *upper* bound of the interval is below 0.5, we know that the next bit in the codeword will be 0. We can transmit it immediately, and in a way *move the binary point* one position to the right in the representation of the interval, i.e.

$$\begin{aligned} lower &:= 2 \cdot lower \\ upper &:= 2 \cdot upper \end{aligned}$$

2. If the *lower* bound gets > 0.5, we know that the next code bit is 1, which is transmitted. Removing the 1-bit after the binary point corresponds to subtraction of 0.5, and then moving the binary point right corresponds to multiplication by 2 (which in practice is implemented by shift-left):

$$\begin{aligned} lower &:= 2 \cdot (lower - 0.5) \\ upper &:= 2 \cdot (upper - 0.5) \end{aligned}$$

   We call these two operations *half-scalings.* Observe that we may have to apply half-scaling operations 1 and 2 possibly in mixed order, until $lower < 0.5$ and $upper > 0.5$. For example, if $lower = 0.4$ and $upper = 0.45$, the scalings proceed as follows:

   Transmit 0, $lower = 0.8$, $upper = 0.9$
   Transmit 1, $lower = 0.6$, $upper = 0.8$
   Transmit 1, $lower = 0.2$, $upper = 0.6$

   The stated problems are not, however, completely solved with half-scalings. A situation may arise, where the nested intervals keep in the middle, so that $lower < 0.5$ and $upper > 0.5$ even though $upper–lower$ gets extremely small; possibly creating a floating-point underflow. This problem can be solved with *quarter-scaling* as follows:

3. If *lower* > 0.25 and *upper* < 0.75, then we know that the next bits to be transmitted are either 011...1 (if *upper* finally drops below 0.5), or 100...0 (if *lower finally* gets > 0.5). Now we scale the interval by

$$lower = 2 \cdot (lower - 0.25)$$
$$upper = 2 \cdot (upper - 0.25)$$

   but we cannot send any bits yet. Instead, we have to remember, how many *pending bits* are waiting, i.e. how many quarter-scalings have been done before the next bit *b* is finally resolved. The pending bits will be complements of *b*, and sent after *b*. Notice that the interval size is doubled, so not very many repetitions of scaling are required, at a time.

As an example of quarter-scaling, consider the interval where *lower* = 0.45 and *upper* = 0.55. The following adjustments are now made:

$$lower = 0.4,\ upper = 0.6$$
$$lower = 0.3,\ upper = 0.7$$
$$lower = 0.1,\ upper = 0.9$$

Thus, three pending bits will wait for transmission. There may become still more, if the same situation reappears after the next shrinking of the interval (with the next source symbol). If not sooner, the final code value (midpoint of the last interval) will ultimately decide the value of the pending bits.

The feature that all bits cannot be immediately sent means that the sender and receiver may not operate quite synchronously. However, in practice, the delay will not be very long. Commercial compression programs contain some adjustments to the basic algorithm, to avoid the worst case (but at the cost of some additional redundancy).

The decoder starts from interval [0, 1), and then restricts it according to received bits. It reads as many bits as needed to uniquely determine the next symbol: If a 0-bit is received, the lower half of the interval is selected, if a 1-bit is received, the upper half of the interval is selected. This is repeated until the interval is contained in the interval of some symbol. This will be the next decoded symbol, and the interval is scaled similarly as in encoding. This means that the 'used' bits are discarded, and new ones are taken to again narrow down the interval in searching for the next symbol. Notice also that the problem of pending bits does not occur in decoding.

Above we suggested that, due to speed, integer arithmetic would be more preferable than floating-point arithmetic. It is relatively easy to convert the above calculations to operate on integers. We just replace the [0,1) interval with a suitable $[0, 2^k-1]$ integer interval. In the scalings and other calculations, we just replace 0.5 by $2^{k-1}-1$(= *half)* 0.25 by $2^{k-2}-1$ (= *first-quarter)* and 0.75 by $3{\cdot}2^{k-2}-1$ (= *third-quarter*). Instead of probabilities, we should use *frequencies* of symbols. In reducing the interval, we have to perform the following computations:

$$upper := lower + (range \cdot cum[symbol] \,/\, total_freq) - 1$$
$$lower := lower + (range \cdot cum[symbol-1] \,/\, total_freq)$$

In order to avoid integer overflow in multiplication, we set a restriction on both range and frequency. For *w*-bit machine words, the maximum cumulative frequency should be $< 2^{w-k}$.

Due to scalings, the range is always larger than one quarter. Therefore, to avoid underflow (*lower* = *upper*) for minimum frequency 1, the *total_freq* should be less than 2^{k-2}. Witten, Neal and Cleary (see reference earlier) suggest the selection $0..2^{16}-1 = 0..65535$ for the initial range, and maximally $2^{14}-1$ for *total_freq*, when using 32-bit machine words. It makes the probability model somewhat imprecise for highly skewed distributions, but this is not usually a severe problem.

Decoding can be similarly modified for integer arithmetic. The process can be started by reading the first k bits (= *value*) sent by the encoder. The first symbol x is determined by the following condition:

$$cum(x-1) \le \left\lfloor \frac{(value - lower + 1) \cdot total_freq - 1}{upper - lower + 1} \right\rfloor < cum(x)$$

Since the precision of *value* is two bits more than the precision of *cum*, adding more bits to *value* would not change the interval, so the selection of x is unique and correct. After every scaling, one front bit is dropped from *value* and the next one read in to the rear, creating a new *value* for the next symbol to be decoded.

4.4.1. Adaptive arithmetic coding

Arithmetic coding is especially flexible in the respect that the model, i.e. probability distribution, can be changed at any time, even after every transmitted symbol. This is because the nested reductions of intervals can be based on any distribution, without affecting the correct functioning of the method. In principle, adapting the model does not belong to pure source encoding, but in order to develop a one-pass method, which has no knowledge of the source beforehand, adaptation is needed. We assume that successive symbols of the message are independent. The approach is analogous to adaptive Huffman coding, see Section 4.2.2.

Basically, adapting the model means maintaining the symbol frequencies; after each transmitted symbol we increase its frequency, as well as the total frequency, by one. As this is done by both encoder and decoder, the two models are kept synchronized. As for the initial model, there are two alternatives:

(a) Initialize the frequencies of all symbols to one.

(b) Start with an ‘active’ alphabet consisting of only a placeholder symbol, called NYT (Not Yet Transmitted). When some symbol is encountered for the first time, the NYT is transmitted (using the distribution of the active alphabet), followed by the actual symbol (using an even distribution for the ‘passive’ alphabet, consisting of all symbols that are not yet encountered). This symbol is then transferred from the passive alphabet to the active, equipped with frequency one.

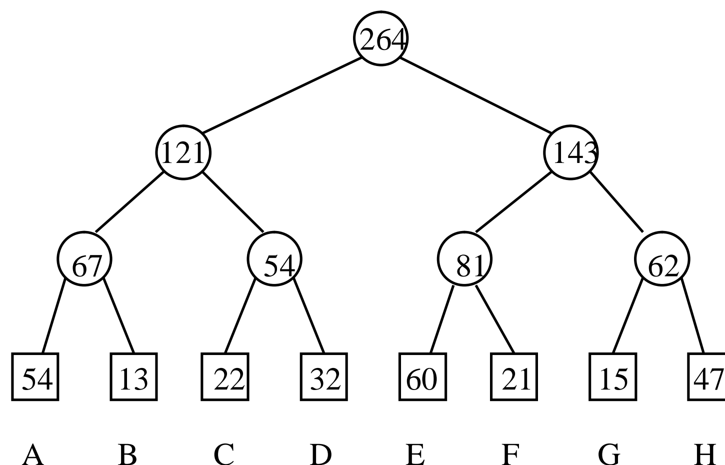
The former approach is simpler, and sufficient in most situations, at least for long messages. The latter has the advantage that the unused symbols do not distort the active distribution; i.e. waste the probability space and cause an unnecessary reduction of intervals. This is especially appropriate for large alphabets (e.g. consisting of words), where only a small fraction of all symbols are actually used in a single message. The problem with alternative (b) is, what frequency we should assign the NYT symbol. Actually, its probability *decreases* after every new active symbol. Thus, we should initialize the frequency to the estimated size of the final

active alphabet, and reduce the frequency by one after every new symbol. However, the value should not be reduced to zero, unless the whole alphabet becomes active.

Another practical problem with adaptive arithmetic coding is the computation of cumulative frequencies, which are needed in both encoding and decoding. Keeping a simple array of cumulative frequencies is quite inefficient; for example, adding 1 to the frequency of symbol 'A' will cause addition to all subsequent cumulative frequencies. One possibility is to keep the alphabet ordered according to increasing probability. Then, adding the frequency of the most probable symbol would affect only one cumulative frequency, adding the frequency of the next probable symbol would affect only two cumulative frequencies, and so on. In addition, we need a mapping table for the order of symbols in the cumulative frequency table.

The above approach is applicable for small alphabets and skew distributions, but generally we should have a more effective data structure. Here we suggest a balanced tree, which is a modification of the structure presented by A. Moffat[1].

We suggest that the frequencies of symbols are maintained in the leaves of a balanced binary tree. Each internal node contains the sum of frequencies of its children. An example tree looks like follows; the actual storage could be implicit in an array, such as for heaps.



The cumulative frequency of a symbol s is obtained as follows: Add the leaf frequency of s to the frequencies of all *left siblings* on the path to the root. For example, the cumulative frequency of 'G' is 15 + 81 + 121 = 217. The cost of this computing is obviously at most $O(\log_2 q)$ for an alphabet of size q. Maintaining the structure is as efficient: If some frequency is increased by one, all nodes up to the root should get an increment of one, as well. Thus, the complexity is again $O(\log_2 q)$.

4.4.2. Adaptive arithmetic coding for a binary alphabet

A binary alphabet (symbols 0 and 1) presents no problem to arithmetic coding; the above algorithms are directly applicable to this case. However, now we can tailor the method and develop faster encoding and decoding programs. The first observation is that the cumulative frequency table has only one relevant entry, expressing the breakpoint between the probabilities of 0 and 1. In other words, we do not need any complicated data structure for main-

---

[1] A. Moffat: "Linear Time Adaptive Arithmetic Coding", *IEEE Trans. on Information Theory*, 36, 2, 1990, pp. 401-406.

taining cumulative frequencies. In fact, in some cases it might be reasonable to decompose longer symbol representations (e.g. ASCII) into bits, and do the encoding bitwise.

The main source of slowness in both arithmetic encoding and decoding is the multiplication operations needed in reducing the intervals. Scalings, instead, need only shift-left operations. For a binary alphabet, several techniques have been developed, which avoid the multiplications by some kind of approximations. We study here two such approaches.

(1) The first technique (due to Langdon & Rissanen, 1981), is based on approximating the smaller of the two probabilities by the nearest integral power of ½, denoted $1/2^Q$, where Q is an integer. The related symbol is called LPS (Less Probable Symbol), while the other one is called MPS (More Probable Symbol). The encoding and decoding routines contain the following expression:

$$low + range * cum$$

The multiplication can be calculated by shifting *range* Q positions to the right. Notice that we can freely choose either 0 or 1 as the ‘first’ symbol of our alphabet, whichever happens to have a smaller probability. The order can even be changed during the coding, as long as both the encoder and decoder apply the same rule of deciding the order.

Now the problem is, how we should choose the value Q for a given probability. Denote the actual value of the smaller probability by p. The related symbol (LPS) is coded as if it had probability $1/2^Q$, occupying $-\log_2(1/2^Q) = Q$ bits (based on information theory). MPS has true probability $1-p$, and occupies $-\log_2(1-1/2^Q)$ bits. The average number of bits per symbol is then

$$pQ - (1-p)\log_2(1-\frac{1}{2^Q})$$

We should find optimal values of Q for each p in interval [0, 0.5). It is sufficient to determine sub-intervals, within which a certain Q-value is optimal. The breakpoint value between, say, $Q = r$ and $Q = r+1$ can be determined by solving the following equation:

$$pr - (1-p)\log_2(1-\frac{1}{2^r}) = p(r+1) - (1-p)\log_2(1-\frac{1}{2^{r+1}})$$

which gives

$$p = \frac{z}{1+z} \quad \text{where} \quad z = \log_2 \frac{1-1/2^{r+1}}{1-1/2^r}$$

The following table gives a few probability ranges, related Q-values, and approximated probabilities ($1/2^Q$):

Probability range	Q	Effective probability
0.3690 – 0.5000	1	0.5
0.1820 – 0.3690	2	0.25
0.0905 – 0.1820	3	0.125
0.0452 – 0.0905	4	0.0625
0.0226 – 0.0452	5	0.03125
0.0113 – 0.0226	6	0.015625

The maintenance of the actual probability is the responsibility of the model. For a given probability, supplied by the model, the encoder (and decoder) chooses the correct range, and applies the related Q-value in the shift-right operation, avoiding multiplication.

It is surprising how little compression performance is lost due to the approximation. The loss can be measured by the ratio of entropy to the average number of bits per symbol:

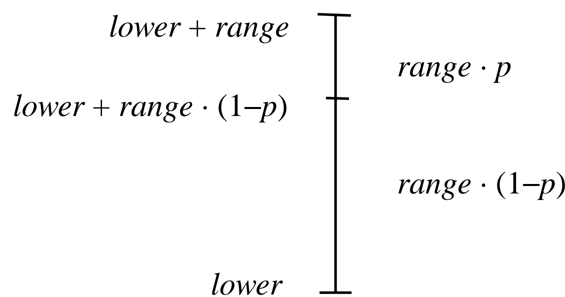
$$\frac{-p\log p-(1-p)\log(1-p)}{pQ-(1-p)\log(1-1/2^{Q})}$$

The worst case is for the first breakpoint, namely $p = 0.369$, which gives 95% effectiveness. In practice a typical observed value has been around 98.5%. Normally, such a small loss is quite acceptable.

(2) E.g in the compression of black-and-white images, another technique is used for approximating the coding formulas. Several variations of the approach have been suggested, e.g. Q-coder (original), QM-coder (refined; studied here), MQ-coder (used in JBIG standard for compressing bi-level images), and M-coder (used in H.264/AVC video compression standard). The QM-coder involves a sophisticated technique for maintaining the model (= probabilities of LPS and MPS) during the coding. The model is based on a fixed Markov model (i.e. finite automaton), where certain events in coding trigger a transition to the next state. Here we concentrate on the coding part of the method.

In QM-coder, MPS is regarded as the first symbol of the alphabet, and LPS is the second. The roles of the original symbols may have to be changed during the coding. For example, when coding a black-and-white image, in mostly-white areas the 'white' symbol is the MPS, whereas in mostly-black area the opposite holds. Again, we denote the probability of LPS by p, and the probability of MPS by $1-p$.

QM-coder maintains two values for the intervals: *lower* and *range*. The ideal subdivision of the interval can be described as follows:



Again, independent of the chosen subinterval, multiplication is needed to update the *range* value. To eliminate multiplication, QM-coder operates (virtually) in the overall interval [0, 1.5). Scaling (renormalization, zooming) is done each time when the *range* drops below 0.75. After scaling, $0.75 \le range < 1.5$. Now, the following crucial approximation is made:

$$range \approx 1$$

This results in the following approximations of multiplications:

$$range \cdot p \approx p$$
$$range \cdot (1-p) = range - range \cdot p \approx range - p$$

When coding the next symbol, the interval is updated as follows:

- For MPS, *lower* is unchanged, and *range* := *range* − *p*.
- For LPS, *lower* := *lower* + *range* − *p*, and *range* := *p*.

Scalings involve only multiplications by 2, which can be implemented by a shift-left operation. Thus all multiplications are eliminated. The practical implementation of QM-coder is done with integer arithmetic, replacing 1.5 by $2^{16} = 65536$, and 0.75 by $2^{15} = 32768$. Notice also that QM-coder does not use the midpoint of the interval as the code value, but the lower bound. When the *lower* value is scaled, we do not have to subtract 32768, even if the value was larger than that (interval in the upper half). The bits that flow over are just the correct 1-bits of the code. They are sent to the output procedure as 8-bit chunks, i.e. bytes. The problem of pending bits is solved in lower-level routines, and we skip its discussion here. The details can be found from: W.B. Pennebaker & J.L. Mitchell: *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993. A slightly different *MQ-coder* is used in JPEG 2000 image compression standard.

4.4.3. Viewpoints to arithmetic coding

There are some practical problems with arithmetic coding. First, since it produces one codeword for the entire message, the code is *not partially decodable* nor *indexable*; we always have to start from the beginning in order to recover a portion of the encoded message. Second, arithmetic coding is *vulnerable*. One-bit errors usually result in a totally scrambled data for the rest of the decoded message, and usually an incorrect number of decoded symbols. Codeword-based methods like Huffman tend to be *self-synchronizable*, although no upper bound for the length of the disrupted sequence can be given.

A version called *Arithmetic Block Coding*[1] (ABC) avoids these problems, assuming that the symbol distribution is static. Its principle is to apply the original idea of arithmetic coding within the limits of machine words (32 or 64 bits), and restart a new coding loop when the bits are used. The coding technique resembles Tunstall code, but does not construct an explicit codebook. ABC-coding is quite fast, because it avoids the scalings and bit-level operations. The code length is not optimal, but rather close to it.

[1] J. Teuhola & T. Raita: "Arithmetic Coding into Fixed-Length Codewords", *IEEE Trans. on Inf. Theory*, Vol. 40, No. 1, 1994, pp. 219-223.

5. Predictive models for text compression

In the previous sections we have studied source encoding methods which are based on a given probability distribution of symbols. Now we change our focus to the methods, using which the distribution is determined. This task is the responsibility of the *model* of the source. The best models are those which at each step assign the highest possible probability to the symbol which will actually appear in the text[1]. This is then reflected in the smallest possible number of bits 'used' in actual coding. We can characterize the model as making *predictions* of what will follow in the message. The concept of prediction is generalized so that the model does not predict only the most probable symbol, but gives all possible symbols some probabilities on which they a predicted to occur. The model can be determined statically before coding, or dynamically, based on the processed prefix of the message.

Predictive techniques are currently the most powerful models in text compression, with respect to compression efficiency. Their drawback is relatively high consumption of resources – both time and working storage (for the model). Many of the practical text compression programs use so called *dictionary methods*, where frequent substrings are units of coding. The subsequent 'entropy coding' technique is usually very simple – even fixed-length coding of substrings. Dictionary methods are relatively fast, but their compression power is clearly lower than with the best predictive methods. This may be somewhat surprising, since one would assume that taking larger units as the basis for modelling would enable bigger gains in compression. Dictionary techniques are described in Section 6.

We can distinguish different kinds of predictive models, depending on the information that is used as the basis of prediction. Three typical approaches are the following:

(1) Finite-context models

In normal text, especially in natural language, there is a high interdependency between successive symbols. In English, for example, the probability of '*u*' is 0.024, but the probability of '*u*' to occur after '*q*' is as high as 0.991. Thus, if coding has proceeded to an occurrence of '*q*', the following '*u*' will involve only $-\log_2 0.991$ bits ≈ 0.013 bits of information. On the other hand, the prediction is not certain, and the other symbols must also be assigned non-zero probabilities, and their occurrence would result in a very high number of bits. This idea can be generalized to more than one preceding symbol. If we have last processed the symbols '*th*', then there is a high probability for '*e*' to follow, although other plausible alternatives exist, as well.

We thus need *conditional probabilities* of symbols, knowing a few previous symbols. The lower bound of compression is given by *conditional entropy* (see Section 2), which is (almost) reached by arithmetic coding, because it functions as well for distributions that vary from symbol to symbol. The conditional probabilities are usually determined empirically, by gathering statistics from the message text itself.

The previous symbols used in prediction are called the *context* (or *prediction block*; called also *conditioning class*) of the next symbol, and the length of the context is called the *order* of the model. Order 0 means a model where no context is used, but the symbols are predicted

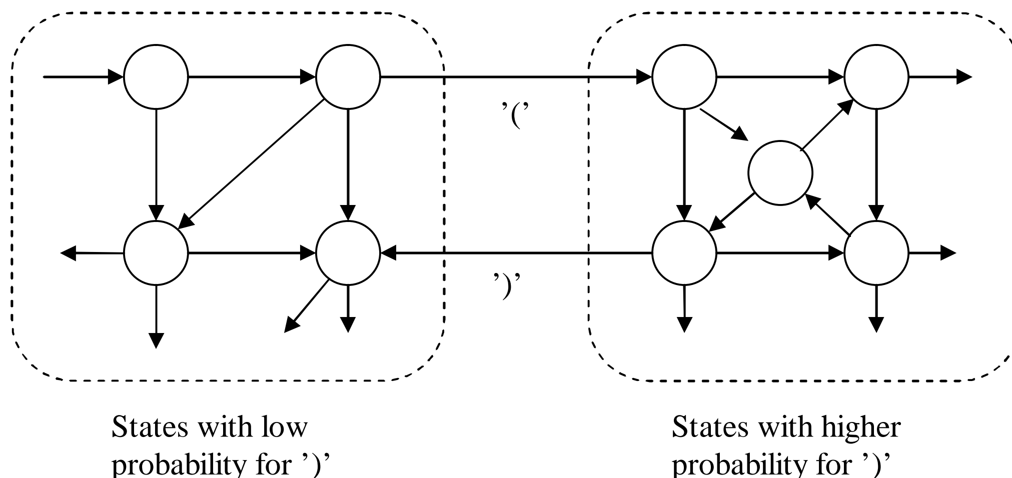
[1] Actually, the techniques discussed in this chapter apply to any symbol sequences in general, not just text.

on the basis of their (global) probabilities. We even define an order '−1' to denote prediction of all q symbols with the same probability ($1/q$). In natural language, the highest correlations are between adjacent symbols, although some kind of dependencies may extend even over 30 symbols, but weaken considerably at word boundaries. In principle, a longer context should give a more precise (= more skewed) distribution and thus result in more effective compression. In practice, it has been observed that extending the order above 5 or 6 will not usually pay. The problem is that collecting reliable statistics becomes difficult, because longer substrings occur more seldom. Also the model size grows rather large, although only the observed contexts need to be recorded.

(2) Finite-state models

There may be dependencies between symbols, which cannot be represented by finite-context models. Consider, for example, a source program written in some structured programming language. If the keyword **if** has been encountered (in a Pascal-like programming language), it is very probable that the word **then** will follow sooner or later, even though the number of intervening symbols may be arbitrarily large. Thereafter, we would expect the word **else**, although it may never appear. On the other hand, it is highly exceptional to see then word **then** or **else** if no **if** has been met. Another example is the matching of parentheses in arithmetic expressions. The occurrence of '(', without an enclosing ')' is infrequent. This, as well as expecting quotes to match, holds also for natural language.

A finite-state model is described by a finite-state machine, with *states*, *transitions*, and symbol *labels* for transitions, together with transition probabilities. The model is also called a *Markov model* or *finite-state probabilistic model*. Compressing a message means traversing through the machine from state to state, choosing transitions according to input symbols. A symbol is encoded (e.g. by arithmetic coding) using the distribution of transitions leaving the current state. To adapt to the **if ... then ... else** structure, the machine should lead, after seeing '**i**' and '**f**', to such a subset of states which gives the transition sequence '**t**'→'**h**'→'**e**'→'**n**' a higher probability than normal. The model thus *remembers* that **if** has appeared. Another example where a finite-state model is useful is such data which contains certain size of repeating blocks (alignments) of symbols. Such data could be a table, an image, a DNA sequence, etc. A sketch of a machine for encoding matching (non-nested) parentheses is given below, showing partitioning of the states.



Although finite-state models are in principle stronger than finite-context models, it is very difficult to take advantage of this possibility, unless the special features of the messages to be compressed are known beforehand. Notice that the finite-state machine need not be static during the compression; in a one-phase adaptive compression scheme, we can add states and transitions, as well as update the transition probabilities during compression, as long as the decoder is able to do exactly the same changes to its own model, intact. An efficient compression technique using this approach was presented for a binary alphabet by Cormack & Horspool[1].

Note still that any finite-context model can be represented by a finite-state model. The related machine has a state for each possible context, and transitions for all possible successors of that context. It is clear that finite-state models suffer from the same problem as finite-context models, namely high demand of working storage for the model. A machine corresponding to context length n would in principle have q^n states and q transitions from each state. In practice, we only have to create states according to observed contexts. As for speed, walking from state to state is rather fast, but maintaining the model can be time-consuming.

(3) Grammar models

Even though finite-state models are able to capture non-contiguous properties of the source, they are not able to notice arbitrary *nestings* of grammatical structures. For example, if a mathematical expression (or a LISP program) can contain an arbitrarily deep nesting of balanced parentheses, there is no finite-state machine that could 'check' the correctness. A more powerful machine is needed, which has a *stack* as an auxiliary data structure. The same holds also for the '**if ... then ... else**' example above, because these structures can also be arbitrarily nested.

A suitable model for these kinds of formal languages is a *context-free grammar*. A syntactically correct source program could be compressed by first parsing the program and then expressing the parse tree in some compact way. There exist compression programs based on this approach, but they are far from general-purpose. Syntactically incorrect programs cannot be handled, nor comments within programs (the proportion of which should be very large). As for natural language, it is very hard to find a suitable grammar. Automatic induction of the grammar, on the basis of the source message only, is not possible, because the system should learn also the features which are *not* allowed in the language. Moreover, the level of nesting is not usually very deep in natural language, implying that finite-context and finite-state models are sufficient for all practical purposes. Notice that as long as an upper bound for nesting is known, a finite-state model can be built to capture the structure.

5.1. Predictive coding based on fixed-length contexts

We start the description of predictive methods by introducing three simple techniques[2] which all use a fixed-length (k) context as the basis of prediction. These techniques are examples of different kinds of couplings between modelling and coding, and also the trade-off between speed and compression efficiency. All three methods are fully adaptive, one-pass techniques.

---

[1] G.V. Cormack and R.N.S. Horspool: "Data Compression Using Dynamic Markov Modelling", *The Computer Journal*, Vol. 30, No. 6, 1987, pp. 541-550

[2] T. Raita and J. Teuhola: "Predictive Encoding in Text Compression", *Information Processing & Management*, Vol. 25, No. 2, 1989, pp. 151-160.

At each step of coding, we should be able to define an approximated distribution of successor symbols in *any* context of length k. Since we cannot have statistics of contexts before we have seen them, we must resort to some default predictions during the learning phase of coding. Even after having seen the context earlier, it may be that the current successor has not occurred before in this context. Therefore, we must have default codings for the not-yet-occurred successors, as well.

An important technical question is, what kind of data structure we should use for storing information about contexts (so called *prediction blocks*) and their successors. Any data structure that enables searching by a string-valued key is sufficient. Most of the predictive methods, presented in the literature, use a *trie* structure for storing and searching context strings. Here we apply a much faster technique, namely *hashing*: The context string of k symbols is converted to a number, from which the *home address* is calculated by a (randomizing) hash function. The successor information, based on the processed part of the message, is stored in the home address. There are two aspects that make hashing especially suitable here:

- The hash table is *faster* and *more compact* than a trie, because the contexts need not be represented explicitly. This means that, given a certain amount of main memory, we can use longer contexts with hashing, compared to the trie.

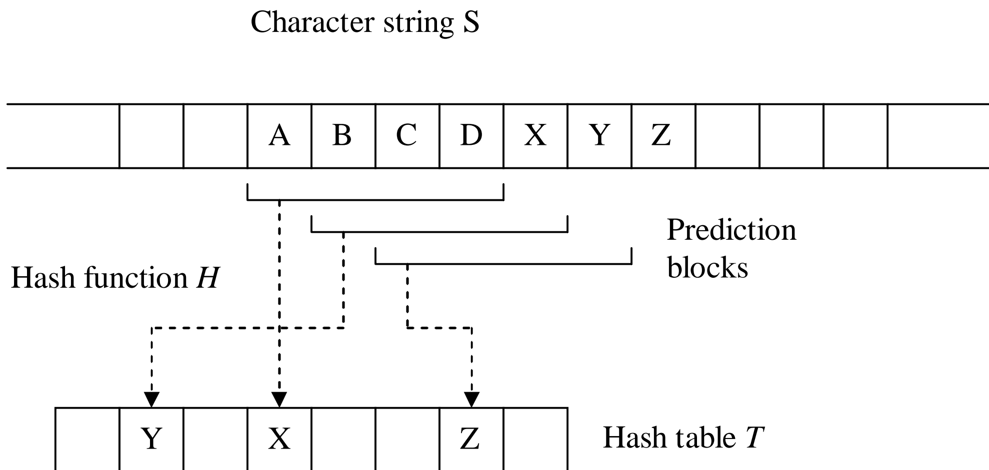
- *Collisions can be ignored.* Their only effect is to deteriorate the compression efficiency slightly, because two different contexts may be interpreted to be the same. However, if the table is not too full, collisions are so rare that they do not play a significant role. In adaptive compression, the table need not be transmitted to the decoder, because the decoder is able to maintain its own table identically with the encoder.

The three methods are here presented in increasing order of compression gain (and thus in decreasing order of speed):

A. Simple success/failure prediction

Given a source message $X = x_1x_2 \dots x_n$ and a prefix $x_1x_2 \dots x_{i-1}$ of x, where $i > k$, we denote the context of x_i by $C = x_{i-k} \dots x_{i-1}$. We predict that x_i is the same as the successor of the previous occurrence of C before this. The success/failure is encoded with one bit; in case of a failure we also have to transmit the actual successor x_i[1]. If C has not occurred before, a default prediction is made (e.g. space for English sources). A hash table $T[0..m-1]$, which is an array of m symbols, is maintained to keep track of the latest successors. For each i from k+1 to n we store x_i into the slot indexed by $H(x_{i-k} \dots x_{i-1})$. The hash function H maps k-grams into $0..m-1$. The data structure is illustrated as follows:

[1] Thus we do not use any of the ‘entropy coders’ of Chapter 4, but a simple yes/no coding.



The encoding algorithm is formulated as follows:

Algorithm 5.1. Predictive success/failure encoding using fixed-length contexts.
Input: Message $X = x_1x_2 \dots x_n$, context length k, hash table size m, default symbol d.
Output: Encoded message, consisting of bits and symbols.

```
begin
    for i := 0 to m–1 do T[i] := d
    Send symbols x1, x2, ..., xk as such to the decoder
    for i := k+1 to n do
    begin
        addr := H(x(i–k) ... x(i–1))
        pred := T[addr]
        if pred = xi
            then Send bit 1                      /* Prediction succeeded */
            else begin
                Send bit 0 and symbol xi         /* Prediction failed */
                T[addr] := pred
                end
    end
end
```

It is obvious that this extremely simple method is exceptionally fast, although its compression gain is modest. On the other hand, it is able to react to local changes of successors for a given context. Due to its speed, this technique has actually been used in some network protocols. Decoding is as simple as encoding, and we skip it here.

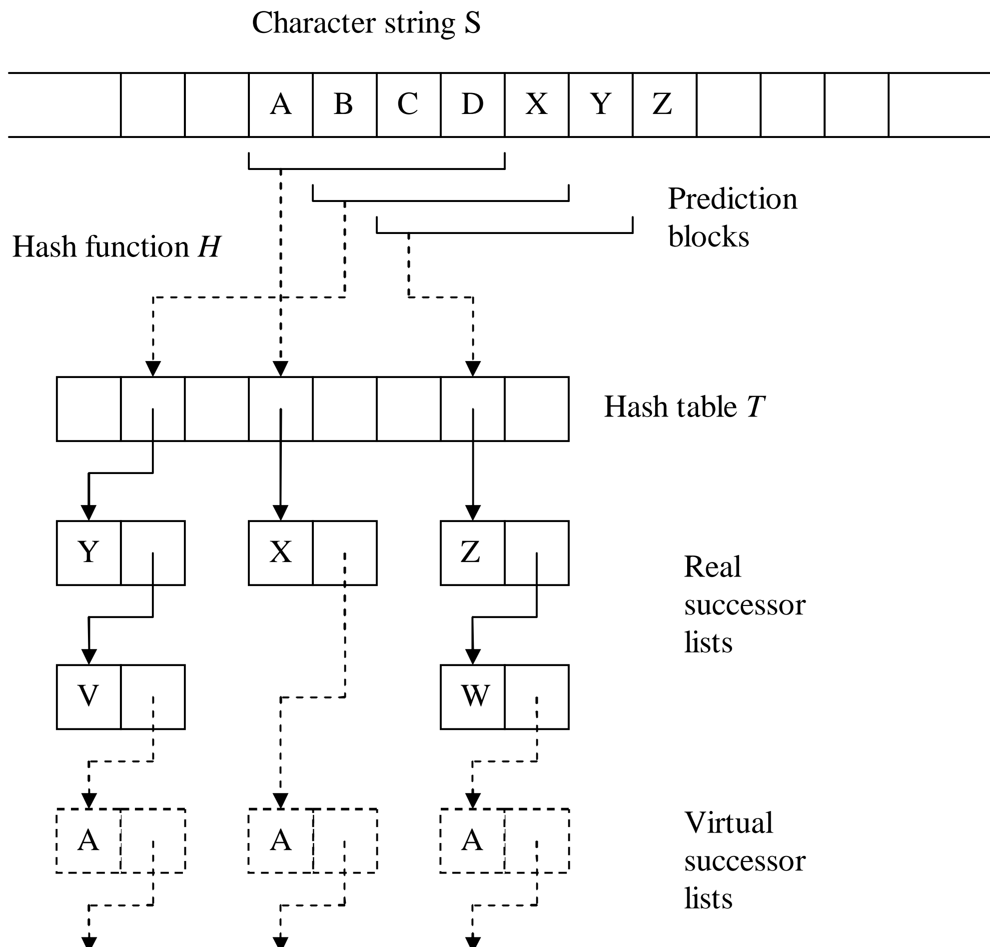
One interesting technical detail is the selection of the hash function, and the computation of numerical representatives of successive (overlapping) k-grams. If we are encoding ASCII characters, then for $k \le 4$ we can consider the k-gram as an unsigned 32-bit integer as such. Moreover, the next k-gram is obtained from the previous by *shift-left* and logical *or* operations. The common and good choice for the hash function is $H(C) = C \bmod m$.

Note that sending the first k symbols as such (to get the process started) can be avoided by assuming a virtual string of k symbols (e.g. spaces) to be at the front of the message. These

then constitute an artificial context for the first actual symbol x_1. The same technique can be applied in the methods to follow, as well.

B. Universal coding of successor numbers.

Now we generalize the prediction to more than one alternative symbol. For each context, every symbol of the alphabet is given an order number, in decreasing order of probability (approximately). However, due to speed, we want to avoid explicit maintenance of frequencies, and therefore we encode the order number of the actual successor within the list of successors for the current context. The order of probability in the list is approximated by applying the *move-to-front* principle: The actual successor is always moved to the front of the list. This means the same as in method A; the latest observed successor of the context is our primary prediction. The symbols which have never occurred as successors for a context, appear virtually at the rear, but in practice they do not have to be stored. In most cases, the order number of the real successor should be small, and we can apply any of the universal coding schemes (α to δ) for integers, presented in Section 2. Hash table T now contains the heads of the successor lists. Decoding is again analogous, and skipped.



The encoding algorithm is given below.

Algorithm 5.2. Prediction of symbol order numbers using fixed-length contexts.

Input: Message $X = x_1x_2 \dots x_n$, context length k, hash table size m.

Output: Encoded message, consisting of the first k symbols and γ-coded integers.

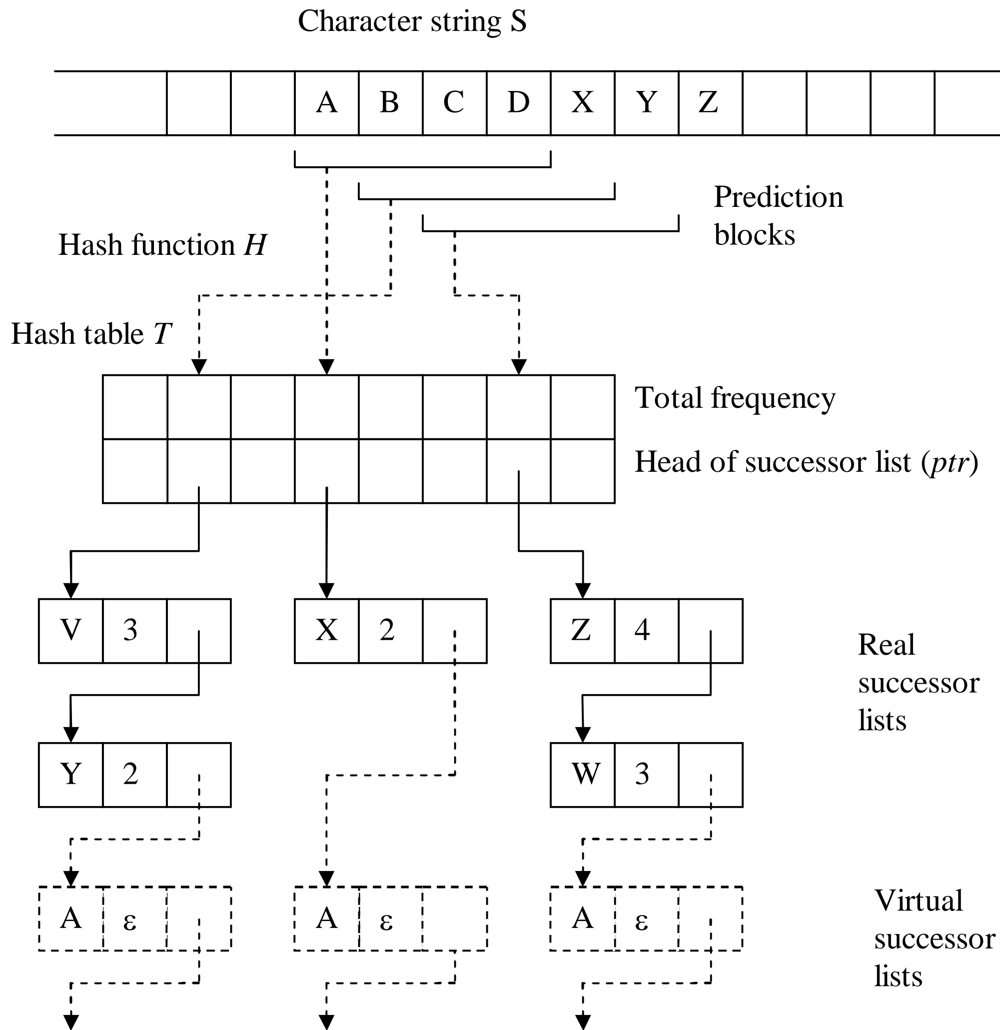
```
begin
    for i := 0 to m–1 do T[i] := NIL
    Send symbols x1, x2, ..., xk as such to the decoder
    for i := k+1 to n do
    begin
        addr := H(x_{i–k} ... x_{i–1})
        if xi is in list T[addr]
        then begin
            r := order number of xi in T[addr]
            Send γ(r) to the decoder
            Move xi to the front of list T[addr]
            end
        else begin
            r := order number of xi in alphabet S, ignoring symbols in list T[addr]
            Send γ(r) to the decoder
            Create a node for xi and add it to the front of list T[addr]
            end
    end
end
```

C. Statistics-based prediction.

By maintaining the frequencies of successors we get a more precise estimate of the actual successor probabilities for each context. By extending the data structure of Method B, we can store the frequencies in the list elements, together with the corresponding characters. The total frequency of the list items is stored in the list head (field $T[i]$.*total* of the hash table entry). Since we now can compute probability estimates for successors, we use *arithmetic coding* for them.

The main problem, concerning all predictive methods, is what we should do with the symbols that have not occurred with the current context C before. We cannot give them zero probability, because coding would fail if the symbol is met, anyway. Here we assume that some small initial frequency (ε) is assigned to all symbols in each context. The value of ε should be optimized empirically. When using integer frequencies, we must choose $\varepsilon = 1$, but probably use increments larger that one for the actual successors.

The data structure extended by frequency fields and total frequencies for successor lists is given below, followed by the encoding algorithm.



Algorithm 5.3. Statistics-based coding of successors using fixed-length contexts.
Input: Message $X = x_1x_2 \ldots x_n$, context length k, alphabet size q, hash table size m.
Output: Encoded message, consisting of the first k symbols and an arithmetic code.

begin

- **for** i := 0 **to** m–1 **do**
- **begin** $T[i].head$:= NIL; $T[i].total$:= $\varepsilon \cdot q$;
- Send symbols $x_1, x_2, \ldots, x_k$ as such to the decoder
- Initialize arithmetic coder
- **for** i := k+1 **to** n **do**
- **begin**
  - $addr := H(x_{i-k} \ldots x_{i-1})$
  - **if** x_i is in list $T[addr].head$ (node N)
  - **then** F := sum of frequencies of symbols in list $T[addr].head$ before ***N***.
  - **else** **begin**
    - F := sum of frequencies of real symbols in list L headed by $T[addr].head$.
    - $F := F + \varepsilon \cdot$(order number of x_i in the alphabet, ignoring symbols in list L)
    - Add a node N for x_i into list L, with $N.freq = \varepsilon$.
    - **end**
  - Apply arithmetic coding to the cumulative probability interval

```
            [F / T[i].total),  (F+N.freq) / T[i].total)
        T[i].total := T[i].total + 1
        N.freq := N.freq + 1
    end  /* of for i := … */
    Finalize arithmetic coding
end
```

The compression efficiency of Method A for English source has been observed to be about 4.5 bits per symbol, while Method B produces from 3.5 to 4 bits per symbol and Method C from 3 to 3.5 bits per symbol. The ratios of both compression and decompression times are roughly 1:5:11 for the three methods.

The presented methods exemplify the separation of modelling and coding steps in compression, although choosing one may affect the choice of the other. Next we shall study more sophisticated modelling techniques, resulting in better compression performance.

5.2. Dynamic-context predictive compression

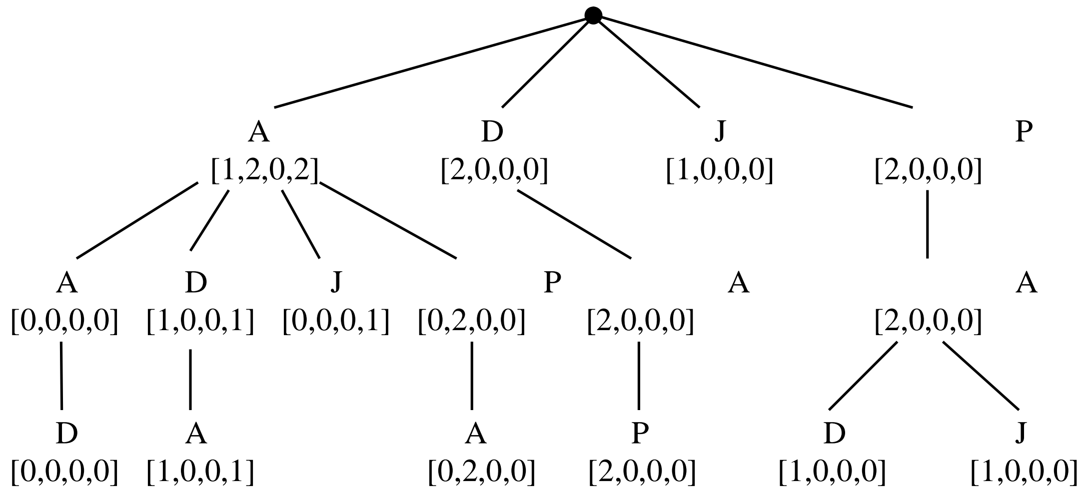
If we perform adaptive compression, where the model gets more precise all the time during the process (assuming an ergodic source), then it is clear that we cannot predict with a context that has not occurred earlier in the text. In other words, we should choose the longest context that has appeared before. This implies that our contexts tend to grow, when the coding gets further. A sophisticated implementation of this simple idea has been presented by R. N. Williams[1]. He made adjustments especially concerning the following questions:

- How to store the observed contexts to enable effective finding of the longest possible context at each step?
- How long contexts should we store?
- When is a context considered 'reliable' to make a prediction?
- What to do if the prediction fails, so that the current successor has not occurred before in this context?

A tailored *trie* structure is used to store the contexts. The substrings are stored in backward order, so that the last symbol of the context is stored on the first level, and the first symbol of the context on the leaf level. Every node stores the frequencies of observed successor symbols for the context represented by the path from the current node to the root. At a certain position of coding, we can just scan the message from the 'cursor' backwards, and simultaneously traverse down the tree, directed by the scanned symbols. An example trie, with maximal context length 3 is presented below. It uses a restricted alphabet of {A, D, J, P}, and is based on the string "JAPADAPADAA". Attached to each node there is an array of four numbers, representing the frequencies of successor symbols A, D, J, P, in this order.

---

[1] Ross N. Williams: "Dynamic-History Predictive Compression", *Information Systems*, Vol. 13, No. 1, 1988, pp. 129–140.



The context lengths are increased, and the trie extended, using a controlling parameter, called *extensibility threshold*, *et*. It is the minimum number of occurrences that a node (i.e. the related context) must have before it can get children (i.e. the context extended backwards). The value of *et* is chosen between $[2, \infty)$, depending on the desired model size.
There are two other parameters to control the model size:

- Depth limit m of the trie sets also a limit to the context buffer and the time consumed per symbol, namely $O(m)$.
- Node limit z gives the available number of nodes in memory; the model can be 'frozen' after reaching the limit.

The prediction itself can be tuned by a parameter called *credibility threshold*, *ct*. It is the minimum number of occurrences that a context must have before it can be used to make a prediction. This is based on the plausible belief that a shorter context with more precise statistics can often make a better prediction than a longer context with very rough statistics. The value of *ct* is chosen from $[1, \infty)$.

The coding of symbols can be done e.g. by arithmetic coding, using the frequencies stored in the predictor node. Now we are faced with a so-called *zero-frequency problem*. If the actual successor has not occurred before, then straightforward computation would give it probability zero, which is illegal. We should assign a small positive probability for each non-occurred symbol, but it is not clear what that value should be. Williams suggests the following formula for the probability of a symbol that has followed the given context x times out of y; parameter q is the alphabet size:

$$\xi(x, y) = \frac{qx + 1}{q(y + 1)}$$

The formula is sensible, because

- $\xi(x, y) \approx x/y$, for large x and y.
- if $x = y = 0$, then every symbol is given probability $1/q$ (case with no statistics),
- if $x = y/q$ then the symbol is given probability $1/q$.

We have now solved all principal problems, and can present the encoding algorithm.

Algorithm 5.4. Dynamic-context predictive compression.
Input: Message $X = x_1x_2 \ldots x_n$, parameters *et*, *m*, *z*, and *ct*.
Output: Encoded message.

```
begin
    Create(root);  nodes := 1;
    Initialize arithmetic coder
    for i := 1 to q do root.freq[i] := 0
    for i := 1 to n do
    begin
        current := root;  depth := 0
        next := current.child[x_{i−1}]          /* Assume a fictitious symbol x_0 */
        while depth < m and next ≠ NIL cand next.freq ≥ ct do
        begin
            current := next
            depth := depth + 1
            next := current.child[x_{i−depth−1}]
        end
        arith_encode(ξ(current.cumfreq[x_i−1], current.freqsum),
                     ξ(current.cumfreq[x_i], current.freqsum))
        {Start to update the trie }
        next := root; depth := 0
        while next ≠ NIL do
        begin
            current := next
            current.freq[x_i] := current.freq[x_i] + 1
            depth := depth + 1
            next := current.child[x_{i−depth}]
        end
        /* Continues … */

        /* Study the possibility of extending the trie */
        if depth < m and nodes < z and current.freqsum ≥ et
        then  begin
            new(newnode)
            for j := 1 to q do
            begin
                newnode.freq[j] := 0
                newcode.child[j] := NIL
            end
            current.child[x_{i-depth}] := newnode
            newnode.freq[x_i] := 1
            nodes := nodes + 1
        end
    end
    Finalize arithmetic coder
end
```

The decompression program is (again) analogous to compression: At each position, we use the current context to determine the node which the encoder has used in prediction. The related frequency vector is fed to an arithmetic decoder, which solves the symbol. The maintenance of the trie is performed identically with the encoder.

Williams has performed empirical tests to determine the compression efficiency of his method with different types of source data. Some of his results are as follows:

Text type	Source size	Bits per symbol
English text (Latex)	39 836	3.164
Dictionary	201 039	4.081
Pascal program	20 933	2.212

While these results are already quite promising (well below the order-0 entropy of the studied sources), we can pinpoint a certain defect in the usage of available prediction information: If prediction fails, i.e. the actual successor of the context has not occurred before, the symbol is given a small slice of probability (determined by function ξ). The *same* probability is given to *any* symbol that has not occurred before with this context. However, some *smaller* context might provide us valuable information about the relative proportions of symbols, which have not appeared with the larger context. This observation leads us to the method to be described next.

5.3. Prediction by partial match[1]

Now we try to use more of the contextual information collected from the front part of the message, for predicting the distribution of the next symbol. The initial idea is to apply *blending of contexts* of different lengths, as follows: The probability of symbol x is computed as the weighted sum

$$P(x) = \sum_{o=-1}^{k} w_o P_o(x)$$

where o goes through all possible orders of contexts, and $P_o(x)$ is the statistically determined probability for order o. The heuristically determined weights w_o should sum up to 1. Clearly, $P(x) > 0$ if $w_{-1} > 0$, which is sufficient for correct coding.

In practice, blending of different contexts is done in a different manner, by finding the longest ($\leq k$) *matching* context, and then *escaping* to a shorter context, if the longest gives a zero probability to the successor symbol. More precisely, we maintain the observed frequencies of all contexts of orders 0 to k, as well as the frequencies of their successors. The first estimate for the conditional probability of symbol x_i in message x is

$$P(x_i|x_{i-k}...x_{i-1}) = \frac{Freq(x_{i-k}...x_i)}{Freq(x_{i-k}...x_{i-1})}$$

If this is zero, we encode a fictitious *escape* symbol (*esc*), with some small but non-zero probability $P(esc)$[2], repeat the attempt to encode x_i, but now using context length k–1. This is repeated until the above formula gives a non-zero probability, or the order becomes −1, which gives the probability $1/q$. Note especially that the frequency of an empty context
(x_i ...x_{i-1}) is the number of processed symbols (= i–1). In principle, the encoding of each source symbol consists of encoding zero or more escapes plus the actual symbol.

The procedure can be still improved by applying the so called *exclusion principle*: When computing the probability of symbol s in some context with order $< k$, we can exclude all the symbols which have occurred as successors of longer contexts. We know that the current symbol is not one of them; otherwise we would not have had to escape to a shorter context. This idea increases the probabilities and hence improves the compression efficiency. On the other hand, it has a negative effect on the processing speed. The following notation will be used for the adjusted probability:

$$P_{ex}(x_i \mid x_{i-j}...x_{i-1}) = \frac{Freq(x_{i-j}...x_{i-1}x_i)}{Freq_{ex}(x_{i-j}...x_{i-1})} = \frac{Freq(x_{i-j}...x_{i-1}x_i)}{\sum_{s\in S} Freq(x_{i-j}...x_{i-1}s \mid Freq(x_{i-j-1}...x_{i-1}s) = 0)}$$

The arithmetic coder is used as the final encoder, therefore we also have to calculate cumulative probabilities $Cum_{ex}(x_i \mid x_{i-j}...x_{i-1})$. The whole encoding algorithm is as follows:

---

[1] J.G. Cleary and I.H. Witten: “Data compression using adaptive coding and partial string matching”, *IEEE Trans. Communications*, Vol. 32, No. 4, 1984, pp. 396-402.

[2] Actually, the probability $P(x_i|\ldots)$ must be slightly decreased, in order to make room for the escape.

Algorithm 5.5. Compression using prediction by partial match.
Input: Message $X = x_1x_2 \dots x_n$, parameter k = longest context length.
Output: Encoded message.

```
begin
    Initialize arithmetic coder
    contexts := { }
    for i := 1 to n do
    begin
        coded := False
        exclusion_set := { }
        j := Max{m ≤ k | (x_{i−m} ... x_{i−1}) ∈ Contexts}
        while not coded do
        begin
            if Prob_ex(x_i | x_{i−j}...x_{i−1}) > 0
            then begin
                arith_encode(Cum_ex(x_i | x_{i−j}...x_{i−1}), Cum_ex(x_i−1 | x_{i−j}...x_{i−1}))
                coded := True
            end
            else begin
                arith_encode(1.0−Prob_ex(esc | x_{i−j}...x_{i−1}), 1.0)
                exclusion_set := successors(x_{i−j}...x_{i−1})
            end
            if not coded
            then begin
                j := j − 1;
                if j = −1
                then begin
                    m := |{z < x_i | z ∉ exclusion_set}|
                    q' := q − |exclusion_set|
                    arith_encode(m/q', (m+1)/q')
                    coded := True
                end
            end
        end
        for j := 0 to Min(i, k) do
        begin
            contexts := contexts ∪ {x_{i−j} ... x_i}
            freq(x_{i−j} ... x_i) := freq(x_{i−j} ... x_i) + 1
        end
    end
    Finalize arithmetic coder
end
```

Above we have assumed that functions $Prob_{ex}$ and Cum_{ex} consult the variables *contexts* and *exclusion_set*. Note also that substrings in *contexts* include also the predicted symbols, i.e. their length is maximally k+1. There are several implementation details to be solved, e.g. the escape probability, strategy for maintaining frequencies, and the data structure used to store the contexts. We shall talk about each of these in turn.

There is no unique (theoretically optimal) solution to the *zero frequency problem*, i.e. to the determination of escape probability. The first idea is to assign the *esc* symbol always the smallest possible frequency, i.e. 1, and

$$Prob_A(esc \mid x_{i-j}...x_{i-1}) = \frac{1}{Freq_{ex}(x_{i-j}...x_{i-1}) + 1}$$

$$Prob_A(x_i \mid x_{i-j}...x_{i-1}) = \frac{Freq(x_{i-j}...x_i)}{Freq_{ex}(x_{i-j}...x_{i-1}) + 1}$$

This is reasonable in the sense that the escape probability decreases when the context frequency increases, i.e. it becomes more and more probable that the successor has already been met before. PPM equipped with this strategy is called PPMA. Another, not so obvious deduction is that the escape probability is proportional to the escapes already taken with the present context. In other words, if some context has had many different successors, there will probably be many more.

$$Prob_B(esc \mid x_{i-j}...x_{i-1}) = \frac{dif}{Freq_{ex}(x_{i-j}...x_{i-1})}$$

where

$$dif = \#\{s \mid s \in Succ(x_{i-k}...x_{i-1})\}$$

This technique, named PPMB, also includes a feature that a symbol is not predicted until it has occurred twice. Therefore, the probability of successor x_i is obtained as follows:

$$Prob_B(x_i \mid x_{i-j}...x_{i-1}) = \frac{Freq(x_{i-j}...x_{i-1}x_i) - 1}{Freq_{ex}(x_{i-j}...x_{i-1})}$$

The '-1' in the numerator stands for the first occurrence, handled as an escape. Still another formula was suggested by Alistair Moffat[1]. It is similar to PPMB, but begins predicting symbols as soon as they have occurred once as successors. This approach, called PPMC, has turned out more successful than PPMA and PPMB in practice. The formulas for *esc* and the real successor are as follows:

$$Prob_C(esc) = \frac{dif}{Freq_{ex}(x_{i-j}...x_{i-1}) + dif}$$

$$Prob_C(x_i \mid x_{i-j}...x_{i-1}) = \frac{Freq(x_{i-j}...x_{i-1}x_i)}{Freq_{ex}(x_{i-j}...x_{i-1}) + dif}$$

Algorithm 5.5 used the rule that, after prediction, the frequency of the predicted symbol was increased by one for all orders of contexts from 0 to k. The PPMC version, instead, applies so called *update exclusion*: The symbol frequency is increased only for the context used to predict it. This can be rationalized by thinking that the frequencies for lower order contexts represent only the cases where the symbol cannot be predicted by higher-order contexts. This modification to the algorithm has been observed to improve the compression efficiency by a couple of per cent. Moreover, it reduces the time required for updating the frequencies.

---

[1] A. Moffat: "A note on the PPM data compression algorithm", Research Report 88/7, Dept. of Computer Science, Univ. of Melbourne, Australia, 1988.

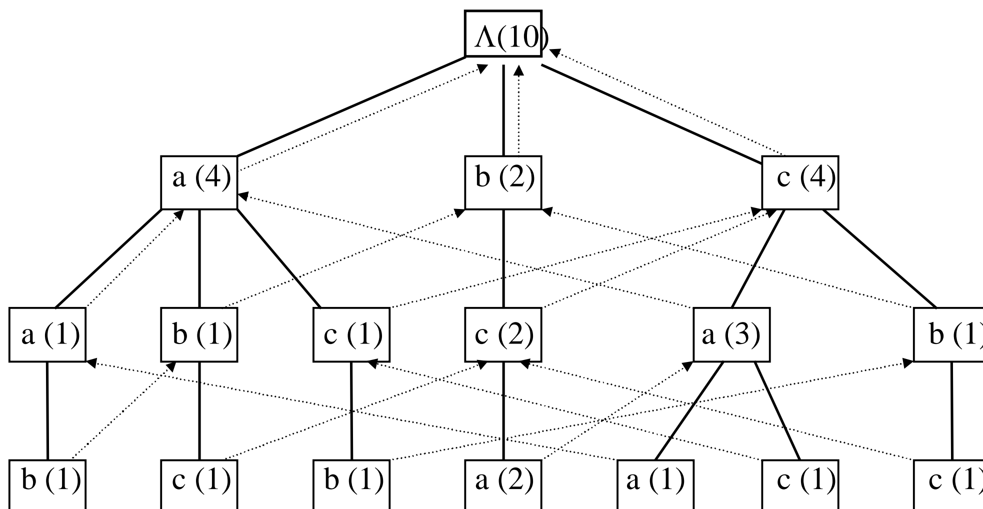
PPMC uses only 8 bits for the frequencies; periodic scaling is needed, but the method becomes more adaptive to local changes in the statistical properties of the message.

Another modification, called *lazy exclusion*, does not reduce the alphabet when taking an escape during prediction. This gives somewhat worse compression (about 5%), but the compression process becomes significantly faster (even to the half), because there is no need to keep track of excluded symbols. This, and some other simplifications are implemented in a version called PPMC', which is tuned for high efficiency. There are at least two simple alternatives for maintaining the contexts (and their successors):

1. *Backward tries* were described in Section 5.2. Their advantage is that all contexts of different orders can be found along a single path from the root down to the longest matching context. Finding the next current context must be started from the root.

2. *Forward tries* store the contexts in forward order. This is useful in the sense that the successors can be stored within the same structure, i.e. the height of the trie is k+1. Counts are maintained for each node (partial context): The count represents the number of occurrences of the context (represented by the path from the root), and the counts of its children record how often the corresponding symbols have occurred in the context. Thus, the parent count should be equal to the sum of child counts (+1 for the last processed context, whose successor has not been counted yet).

Forward tries are not very efficient as such, because every escape to a shorter context requires walking a new path from the root. However, they can be made much faster by equipping the nodes with so called *vine* (*suffix*) *pointers*. Suppose that the path of symbols to a given node on level m is $[s_1, s_2, ..., s_m]$. The vine pointer from this node points to the node on level m−1, representing path $[s_2, ..., s_m]$. Vine pointers have two advantages. First, after finding the longest matching context, the possible escapes can be realized by following the vine pointers to shorter contexts. Second, the next current context can be found by following the vine pointer from the previous predicted node. Thus, it is never necessary to search for contexts, but only to follow pointers. Updating the tree may involve more time, unless update exclusion is applied. New contexts must be added, anyway. An example tree, derived from the message "cacbcaabca" is given below, for k=2. Vine pointers are shown with dashed arrows.



The compression performance of PPMC is excellent, producing about 2 bits per symbol for long messages in English language. Therefore it is generally used as a reference method. Several improved versions of PPM have been published. One of them, presented by P. Howard[1], suggested a new variant (D) of estimating the escape probabilities: When processing a new successor after some context, PPMC adds 1 to both escape frequency (variable *dif*) and the new symbol frequency. PPMD, instead, adds only ½ to both, so that altogether 1 is added, no matter if the symbol is new or met before in this context. The formula for the escape probability is simply

$$Prob(escape) = \frac{u/2}{n}$$

where u is the number of different symbols in the context and n is the total frequency of the context. This, together with some other special tunings, improves the compression efficiency about 5−10%, compared to PPMC (see table of comparison below).

The inventors of PPM have later published[2] a version called *PPM**, the idea of which is to relax from setting a fixed maximum length for the contexts. Instead, the method chooses always the longest *deterministic* context currently in the model. This means that the context has earlier occurred with only one successor. If there is no fitting deterministic context in the current model, the longest stored context is used, instead. This approach represents extreme prediction: choose as skew a distribution as possible, based on the most uncertain statistics. Practical experiments have shown that, as for compression efficiency, this is a relatively profitable strategy, although improvements are perhaps smaller than expected.

PPM* uses a generalized context trie, where each path from the root to a leaf represents a context (substring) that has had only a single successor in the message, up to the current position. Each leaf node has a pointer (= index) to the successor symbol of the context *in the source message*. This makes it simple to extend the path if the same context appears again: add one node for the pointed symbol, and add one to the pointer value. To update the trie, a linked list of pointers to the currently active contexts (= suffixes of the processed part of the message) can be maintained. This list in a way replaces the vine pointers suggested in the earlier implementation of PPM.

The implementation of PPM* is obviously more challenging than PPM. The amount of model data can be much larger, the trie can be much higher, and processing much slower. The authors suggest a so called PATRICIA-style trie, where space is saved by collapsing non-branching sub-paths into single nodes. This will make the size of the trie linear in the length of the input message. However, the maintenance of frequency counts for the nodes becomes more complicated.

PPM* uses a similar technique for handling escapes as PPMC. The choice of escape probability is here even more crucial, because the method uses longest deterministic contexts. These contexts tend to grow and grow, keeping the prediction insecure (= escapes occur) throughout the message, while for PPMC the proportion of escapes reduces when coding proceeds. PPM* also applies the technique of excluding predictions made by higher-order contexts, after escaping to lower order. Another tailored improvement in the PPM family,

---

[1] P.G. Howard: "The Design and Analysis of Efficient Lossless Data Compression Systems", Technical Report CS-93-28, Brown Univ., 1993.

[2] J.G. Cleary, W.J. Teahan, and I.H. Witten: "Unbounded Length Contexts for PPM", *Proc. of the Data Compression Conference*, Snowbird, Utah, 1995, pp. 52-61.

called PPMZ, has been presented by Charles Bloom[1]. He attacked three features that caused PPM* to be less efficient than expected:

1. The unbounded context may produce too imprecise prediction – the order should be decided more locally. In PPMZ, only deterministic (one successor) unbounded contexts are used, otherwise a fixed limit is set on the context length (8 in the experiments).

2. Different message types may have different optimal local orders. Bloom has noticed that the best confidence measure is given by the probability of the most probable symbol (*MPS*). The context with the highest *Prob*(*MPS*) is attempted first in prediction and, in case of failure, followed by an escape to the next smaller context.

3. The escape probability estimations in PPMA ... PPMD have all been based on heuristic formulas. Although no theoretically best formula has been presented, it is possible to base the escape probability on statistics and prediction. Bloom built a 4-component synthetic context for escapes, including PPM order, number of escapes, number of successful matches, and the previous symbols. This information is squeezed into 16 bits and used as an index to a table (2^{16} entries) storing the actual number of escapes and matches. In coding, the escape probability is just looked up from this table.

The following table gives a comparison of PPMC, PPMD+ (a tuned version of PPMD), PPM* and PPMZ for *Calgary Corpus*, a popular benchmark for compression programs. The results denote bits per source symbol (character or byte).

File	Size	PPMC	PPMD+	PPM*	PPMZ
bib	111 261	2.11	1.86	1.91	1.74
book1	768 771	2.48	2.30	2.40	2.21
book2	610 856	2.26	1.96	2.02	1.87
geo	102 400	4.78	4.31	4.83	4.03
news	377 109	2.65	2.36	2.42	2.24
obj1	21 504	3.76	3.73	4.00	3.67
obj2	246 814	2.69	2.38	2.43	2.23
paper1	53 161	2.48	2.33	2.37	2.22
paper2	82 199	2.45	2.32	2.36	2.21
pic	513 216	1.09	0.80	0.85	0.79
progc	39 611	2.49	2.36	2.40	2.26
progl	71 646	1.90	1.68	1.67	1.47
progp	49 379	1.84	1.70	1.62	1.48
trans	93 695	1.77	1.47	1.45	1.24
average	**224 402**	**2.48**	**2.25**	**2.34**	**2.12**

The excellent results of PPMZ have been obtained at the cost of extremely high processing time and large working storage. Thus, the methods cannot be considered candidates for practical compression software. However, they may have other interesting applications where powerful predictive capability is needed. Such models have even been used to predict the stock market.

[1] C. Bloom: “Solving the Problems of Context Modeling”, California Inst. of Technology, http://www.cbloom.com/src/index.html/, 1998.

5.4. Burrows-Wheeler Transform

In 1994, M. Burrows and D.J. Wheeler[1] presented a compression method, which at first sight seems to be totally different from all known techniques. It first performs a *transformation* of the original message into a form which contains all the original symbols, but ordered quite differently, so that symbols are highly clustered. The transformed message is then compressed by some simple technique. The compression performance is about the same order as with the best predictive methods, but the speed is much better.

A precondition for any transformation is that it must be reversible, i.e. an *inverse* transformation must exist to recover the original message. The Burrows-Wheeler transform has this property, and is done as follows. Given a sequence $X[0..N-1]$ of symbols from alphabet S, form a *matrix* having all *rotations* (cyclic shifts) of X as rows. The rows of the matrix are then *sorted* into lexicographically increasing order. The *last column L* of the sorted matrix M is the result of transformation, plus the index (I) of the row containing the original message. An example transformation of the message 'MISSISSIPPI' is as follows:

```
                                      Matrix M
 0  MISSISSIPPI                      IMISSISSIPP                  P
 1  IMISSISSIPP                      IPPIMISSISS                  S
 2  PIMISSISSIP                      ISSIPPIMISS                  S
 3  PPIMISSISSI                      ISSISSIPPIM      LAST        M
 4  IPPIMISSISS      SORT            MISSISSIPPI     COLUMN       I
 5  SIPPIMISSIS    -------->         PIMISSISSIP    -------->     P
 6  SSIPPIMISSI                      PPIMISSISSI                  I
 7  ISSIPPIMISS                      SIPPIMISSIS                  S
 8  SISSIPPIMIS                      SISSIPPIMIS                  S
 9  SSISSIPPIMI                      SSIPPIMISSI                  I
10  ISSISSIPPIM                      SSISSIPPIMI                  I
```

The result of the transformation consists of $I = 4$ and $L = [$ P, S, S, M, I, P, I, S, S, I, I $]$.
Let us now derive the reverse transformation for recovering the original message. The first observation is that column L contains all symbols of the original message, but in different order. Therefore, by *sorting L*, we can recover the *first* column F of the sorted matrix M. Since the rows of M are all rotations, we can construct a matrix M', with L as the first column and F as the second, and the rest of the columns unknown, but sorted lexicographically on the *second* column. Thus, M' is a cyclic rotation of M to the right by one position.

```
           L F
           P I ...
           S I ...
           S I ...
           M I ...
M' =       I M ...
           P P ...
           I P ...
           S S ...
           S S ...
           I S ...
           I S ...
```

[1] M. Burrows & D.J. Wheeler: "A Block-sorting Lossless Data Compression Algorithm", Research Rep. 124, Digital Systems Research Center, Palo Alto, California, 1994.

More precisely, $M'[i, j] = M[i, (j-1) \bmod N]$. All rotations of the original message appear once in both M and M'. The original message could be recovered if we could order the symbol pairs in the first two columns of M' appropriately (cf. domino pieces):

MI, IS, SS, SI, IS, SS, SI, IP, PP, PI

To conclude the correct order of 'chaining', we make the following observation: If we consider the rows of M' starting with a certain symbol x, we know that these rows are mutually in sorted order, because the first symbol is x in all of them, and the rest are sorted (lexicographically from the second column on). Therefore, these rows appear in M and M' *in the same order*. The corresponding rows of M appear in 1-rotated form in rows of M' with x as the *second* symbol. This makes up the connection: The rows of M' with x as the first symbol are (in the same order) the 1-rotations of rows of M' with x as the second symbol. For example, the rows of M' starting with IM, IP, IS, IS correspond in this order to the rows starting with PI, SI, SI, MI, so that the symbol 'I' is the connecting link.

Formally, we can represent the chain by an index vector T storing the connections: If $L[j]$ is the k'th instance of x in L, then $T[j] = i$, where $F[i]$ is the k'th instance of x in F. Thus, it holds that $L[j] = F[T[j]]$. For the above example, the vector T is as follows:

Index	*L*	*F*	*T*
0	P	I	5
1	S	I	7
2	S	I	8
3	M	I	4
4	I	M	0
5	P	P	6
6	I	P	1
7	S	S	9
8	S	S	10
9	I	S	2
10	I	S	3

The reverse transformation recovers the original message from back to front, starting from the L-symbol from row 4, which was the order number of the original message in the sorted matrix M. Since M' is its 1-rotation (to the right), the *last* symbol of the message is 'I'. We follow the T-link to row 0, which gives us the symbol 'P', preceding the last 'I'. Then the T-link leads us to row 5, giving the previous symbol 'P', and so on.

The construction of array T can be done in linear time: For each symbol, determine its first occurrence in column F. Then scan through column L, and assign each row the next index value of the related symbol.

There is a symmetric alternative to set the links, so that the original message is recovered in forward order, but this feature is not important; by now it should be clear to the reader that it is question of an *off-line* compression method. In an on-line situation, the method is still applicable by dividing the message into blocks, and handling one block at a time. However, the block size should be rather large to obtain good compression gain.

By now, we have just transformed the original message into another, equally long form. To understand the properties of the transformed string, as for compression, we should realize that subsequent symbols of L are predecessors of *very similar contexts*, because these contexts are sorted. We could express it so that *we are using the whole message as a context* for each symbol. A rotated message has somewhere an adjacent pair, representing the last symbol and the first symbol of the original message. Extending the contex over this point probably does not give any advantage, so in practice we can interpret that the context of a symbol is the related *suffix* of the original message. Furthermore, we can mirror the original message, if wanted, to obtain a context that is the *prefix* of the message up to the current symbol. This is in fact quite close to the idea of deterministic contexts applied in PPM*.

From the above observation we conclude that the symbols in L have a high probability of repeating themselves in succession. This can be taken advantage of in the final compression of the transformed message. Burrows and Wheeler have suggested a simple *move-to-front* coding, which proceeds as follows. We first initialize a list Z to contain all symbols of the alphabet S. The symbols of L are encoded according to their position number $[0..N-1]$ in this list. Moreover, after each symbol, we move the related list item in Z to the front of the list. Because equal symbols tend to cluster, there is a high probability of getting mostly small order numbers as a result, even sequences of 0's, when the same symbol is repeated. After this second transformation, the resulting sequence R of numbers is encoded using e.g. Huffman or arithmetic coding.

Assuming that Z is initially [I, M, P, S], then move-to-front coding of L = [P, S, S, M, I, P, I, S, S, I, I] produces the sequence R = [2, 3, 0, 3, 3, 3, 1, 3, 0, 1, 0]. This example is too small to see the considerable clustering of symbols in L, and sequences of 0's in the result.

Now that we are convinced of the correctness of the method, and its potential in compression, we should study how it can be effectively implemented. The matrix representation was purely for illustrative purpose, and its actual construction is out of the question. So, the main problem is, how do we perform the sorting. Burrows and Wheeler describe a sophisticated variant of Quicksort for suffixes. We take here a simpler approach: Create a vector $P[0..N-1]$, containing indexes of the symbols in X. Initially $P[i] = i$ for all i. Then we can sort P with any *comparison-based* method, using a sophisticated comparison operator:

$P[i] \leq P[j]$ if the rotation starting at location i is lexicographically $\leq$ rotation starting at j.

This means that we compare symbols pairwise from indexes i and j on, until they do not match. After sorting, P points to the symbols of column F. Column L is obtained from the previous positions as follows:

$$L[i] = X[(P[i]-1) \bmod N]$$

In practice, the performance can be improved by making word-based comparisons of 4 ASCII symbols at a time, but word alignments must be adjusted. Another alternative would be to apply *radix* sorting in forward order, so that we first partition the rotations according to their first symbols, then to their second symbols, and so on. Again, rotations would be represented by pointers to their starting positions. After a few partitioning levels, the subsets are so small that they can be sorted with some simple method. Still another (linear-time) alternative is to build a *suffix-tree*, but its memory consumption is rather high.

One may wonder if there are other alternatives to move-to-front coding that would perform better. The first idea is to apply any normal text compression method, such as PPM, to the transformed message. However, in spite of repetitions of symbols, the properties of the transformed message are quite different. The neighbouring symbols do not give *contextual* information; they only 'increase' the probability of symbols to reappear.

One clear improvement, suggested also by Burrows and Wheeler, is to encode *runs* of zeroes separately, because they are so common; roughly 2/3 of all numbers to be coded are normally zeroes.

Peter Fenwick[1] has tried to improve on the results of Burrows and Wheeler, and experimented several variations of final coding. One of his observations was that *unary coding* of numbers from move-to-front compression performs surprisingly well. This means that the distribution of number probabilities is close to negative exponential. The following table shows the compression performance of the original Burrows-Wheeler (BW) method and Fenwick's best version (called BZIP), in comparison to the earlier PPMC and PPMZ programs. It can be seen that the performance of BW and BZIP is between PPMC and PPMZ.

File	Size	PPMC	PPMZ	BW	BZIP	GZIP
bib	111 261	2.11	1.74	2.07	1.95	2.51
book1	768 771	2.48	2.21	2.49	2.40	3.26
book2	610 856	2.26	1.87	2.13	2.04	2.70
geo	102 400	4.78	4.03	4.45	4.48	5.34
news	377 109	2.65	2.24	2.59	2.51	3.06
obj1	21 504	3.76	3.67	3.98	3.87	3.83
obj2	246 814	2.69	2.23	2.64	2.46	2.63
paper1	53 161	2.48	2.22	2.55	2.46	2.79
paper2	82 199	2.45	2.21	2.51	2.42	2.89
pic	513 216	1.09	0.79	0.83	0.77	0.82
progc	39 611	2.49	2.26	2.58	2.50	2.67
progl	71 646	1.90	1.47	1.80	1.72	1.81
progp	49 379	1.84	1.48	1.79	1.71	1.81
trans	93 695	1.77	1.24	1.57	1.50	1.61
average	**224 402**	**2.48**	**2.12**	**2.43**	**2.34**	**2.70**

A variant of BW, called bzip2, is nowadays a standard codec in Unix/Linux. What is especially interesting is that the speed of BW is amazingly good, thinking of all the phases needed. Compared to GZIP, the other popular compression program in Unix, the tuned version of BW is not much slower, but gives clearly better compression, especially for long texts in natural language (see above). PPMZ, on the other hand, is an order of magnitude slower. Similar to GZIP, also in BW the decompression is much faster than compression. This is an important property e.g. in information retrieval. Thus the transform-based techniques seem to give a good solution to the speed/compression trade-off.

[1] P.M. Fenwick: "The Burrows-Wheeler Transform for Block Sorting Text Compression: Principles and Improvements", *The Computer Journal*, Vol. 39, No. 9, 1996, pp. 731–740.

6. Dictionary models for text compression

Until now we have mostly studied compression methods that encode one symbol at a time. Next we move to the class of methods that encode longer substrings as units. This is a very natural approach, considering the parsing of text into words, prefixes, suffixes, phrases, etc. In general, *dictionary-based coding* means replacing substrings of the source message by some kind of pointers or indexes to an implicit or explicit dictionary.

One might assume that by generalizing the coding unit from a single symbol to multiple symbols, we should get advantage to the compression efficiency. However, it can be proved that *each dictionary scheme has an equivalent statistical scheme achieving at least the same compression*. In other words, statistical methods are at least as good, and this is also the observation in practice: statistical methods outperform the dictionary methods. The advantages of the latter are simplicity and speed, which have made them generally more popular. The situation might change in the future, along with the increasing speed of computer processors.

Quite many dictionary methods have been suggested, because there are so many alternative ways to implement the basic idea. Some of the design decisions are:

- Selection of substrings to the dictionary
- Restricting the length of substrings
- Restricting the *window* of the dictionary in adaptive methods
- Encoding of dictionary references

One classification is the same as presented before, based on the generation time of the dictionary:

1. *Static dictionaries* are fixed in advance, before seeing the message to be compressed. This is useful only when the type of messages is known precisely in advance. For example, we could use a normal English dictionary for coding messages in English. However, in most cases these kinds of dictionaries are too large and too non-specific.
2. *Semi-adaptive dictionaries* create a specific dictionary for each message before compression. Since the dictionary must be transmitted to the decoder, the dictionary size is usually restricted. Finding an *optimal* dictionary is an NP-complete problem, so a heuristic is needed. Typically, one tries to find approximately equi-frequent substrings, so that encoding can be performed using fixed-length codes.
3. *Adaptive dictionaries* are based on the processed part of the message. As with statistical techniques, this category is by far the most popular, and both of the large method 'families' to be studied (LZ77 and LZ78) are adaptive.

For static and semi-adaptive dictionaries, we can distinguish different kinds of *parsing strategies* of dividing the message into substrings:

- *Greedy*: The message is parsed from left to right, and at each step the longest matching substring is chosen. (In adaptive methods, this is the only reasonable choice.)
- *Longest-Fragment-First* (*LFF*): Choose the longest substring somewhere in the unparsed part of the message, matching with a dictionary substring.
- *Optimal*: Create a *directed graph* of all matching substrings and determine its *shortest path*.

Jacob Ziv and Abraham Lempel were the pioneers of adaptive dictionary compression. In years 1977 and 1978 they published two seminal papers[1], which have been the basis for most text compression methods currently in practical use. In the literature, methods are usually classified according to their origin into LZ77 and LZ78 families of methods. The former applies an *implicit* dictionary, whereas the latter builds dynamically an *explicit* dictionary. We shall study each of them in turn, as well as some representative methods.

6.1. LZ77 family of adaptive dictionary methods

In the LZ77 approach, the dictionary is simply part of the already encoded sequence. 'Part' means that we maintain a *sliding window* of N symbols, divided into two parts: the *search buffer* contains a suffix of $N–F$ symbols of the processed sequence, and the *lookahead buffer* contains the rest F symbols. The sliding can be implemented by using a circular buffer, so that the oldest symbols are always overwritten by the newest ones, as the encoding proceeds. A typical search buffer size $N–F$ is 4–16Kbytes. The reasons for restricting the domain of substrings into a fixed-size search buffer are twofold. First, the references to substrings can be made shorter, and second, scanning the search buffer can be made faster. Moreover, due to locality of occurrences, the best matches are often found rather close. The reasons for restricting the size of the lookahead buffer are similar: representation of the length can be restricted and finding the match can be made faster. A typical value for F is 16. Longer matches are so rare that degradation of compression is negligible.

The encoder searches the window for the longest match. After finding it, the encoder encodes it with the triple <*offset, length, char*> where *offset* is the distance of the match start from the current position, *length* is the length of the match, and *char* is the next symbol in the lookahead buffer after the match (i.e. the first non-matching symbol). The reason for sending the third component is to handle the cases where no match can be found. In this case both *offset* and *length* are 0, and the third component gives the symbol to proceed with. If fixed-length codes are used, coding the triple takes $\lceil \log_2(N-F) \rceil + \lceil \log_2 F \rceil + \lceil \log_2 q \rceil$ bits.

Suppose that our message is as follows:

...cabracadabrarrarrad...

Suppose further that the length of the window $N = 13$, the look-ahead buffer size $F = 6$, and the current window contents is as follows:

cabraca	*dabrar*

The lookahead buffer contains the string "*dabrar*", so we are looking for "*d...*" from the search buffer. Since there is no '*d*', we transmit the triple <0, 0, *d*>. This seems wasteful, but in practice the situation is rare. Moreover, practical implementations use somewhat different techniques (see later). After encoding '*d*', we move the window one step forward, and obtain

[1] J. Ziv and A. Lempel: "A Universal Algorithm for Data Compression", *IEEE Trans. on Information Theory*, Vol. 23, No. 3, 1977, pp. 337-343.
J. Ziv and A. Lempel: "Compression of Individual Sequences via Variable-Rate Coding", *IEEE Trans. on Information Theory*, Vol. 24, No. 5, 1978, pp. 530-536.

abracad	*abrarr*

Searching backwards, we first find "*ad...*", having match length 1, then we find "*ac...*", also with match length 1, and then finally "*abra...*", having the longest match length 4. The position of the match is encoded as the distance from the current position, i.e. 7. The whole triple is thus <7, 4, *r*>. The window is moved 5 symbols forward, which produces

adabrar	*rarrad*

The longest match in the search buffer is "*rar*", but in this case we can use an even longer match, namely "*rarra*", overlapping the lookahead buffer. It is easy to see that the decoding succeeds without problems: symbols are copied from the referenced substring one by one; at the time when the copying passes the end of the search buffer, the next symbols ("*ra*") have already been solved and can be 'reused'. The following observation holds: *The encoder looks for the longest match with its first symbol in the search buffer*. The triple to be sent is thus <3, 5, *d*>. An extreme situation occurs when encoding *runs* of equal symbols. Assume the following coding situation (with '|' separating the search and lookahead buffers):

... *xa*	*aaaaaaaab* ...

The next triplet will be <1, 8, *b*>, because "*a*|*aaaaaaa*" matches with "|*aaaaaaaa*". Thus, the coding is very effective.

The authors of the LZ77 scheme have shown that asymptotically the performance of this algorithm approaches the best possible that could be obtained by any (semi-adaptive) scheme having full knowledge of the statistics of the source. However, this is true only asymptotically, and in practice it is possible to make different kinds of improvements for finite messages. As for compression efficiency, the composition of the triple and its coding have been varied, as well as the window size.

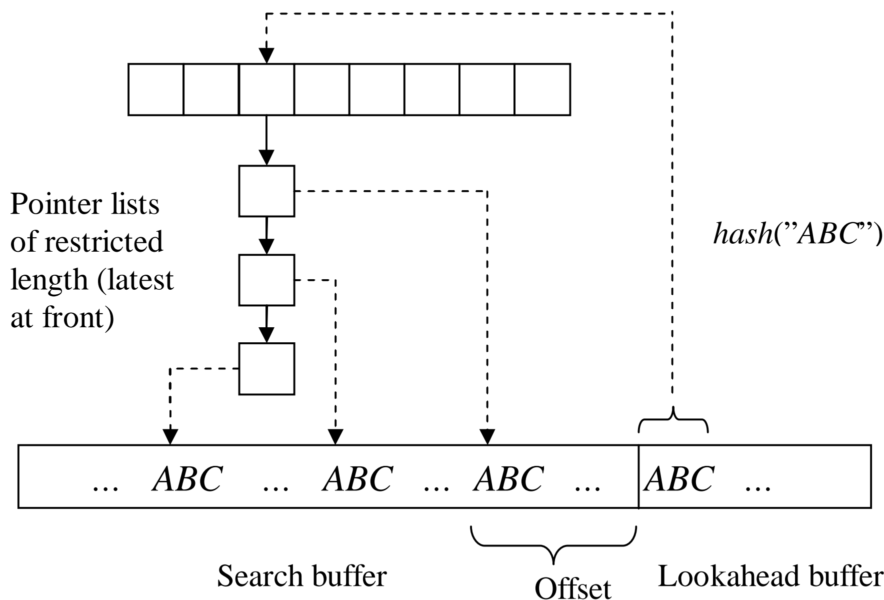
- **LZR** (Rodeh et al., 1981) uses the whole processed part of the message as the search buffer. It applies universal, variable-length coding of integers, representing the arbitrarily large offset.
- **LZSS** (Storer & Szymanski, 1982) avoids the third component of the triple: It sends a flag bit to tell, whether the next code represents an offset&length pair, or a single symbol.
- **LZB** (Bell, 1987) uses γ-coding for the match length, so parameter F is no more relevant. Further, LZB includes some other tunings, concerning e.g. the front part of the message.
- **LZH** (Brent, 1987) combines LZ77 with Huffman coding.

As for speed, the basic LZ77 encoding is quite slow, due to string searching. On the other hand, decoding is extremely fast – a typical feature of many dictionary methods. Encoding can be speeded up by auxiliary index structures (e.g. trie or hash).

Popular compression packages, such as PKZip, Zip, LHarc, ARJ and Zoo, all use algorithms based on LZ77, but are equipped with variable-length coders. We study here a popular package used especially in Unix environment (part of the GNU software):

GZip

GZip uses a hash table to locate previous occurrences of substrings. The next three symbols of the lookahead buffer are hashed to a number, used as an index to a hash table. The table contains heads for lists containing references to the locations where the three symbols have occurred in the search buffer. To keep searching fast, the lists have limited length, i.e. some occurrences are forgotten. Keeping the latest occurrences at front slightly improves the compression efficiency, because the pointer (offset) refers to the closest instance of the longest match. If no suitable match is found, raw symbols are encoded. The used data structure is illustrated below. The task is to find the longest match among the substrings starting with “*ABC* ...”. Note that generally the match is more than three characters.



Two Huffman codes are used for representing the substrings:

1. Lengths of matches and raw symbols are considered a single set of values (e.g. symbols as 0..255 and lengths as $l + 253$, where $l = 3, 4, ...$ These values are Huffman-coded.

2. Offsets are Huffman-coded, so that recent matches, being usually more frequent, are encoded with a smaller number of bits. If the first code represents a raw symbol, not a length, then the offset can be omitted.

Notice that the reversed order of the codes is extremely important for the decoder. It may seem odd to combine match lengths and raw symbols into a single code, but in this way the flag bit (cf. LZSS) is avoided, and in the long run this will pay off. The Huffman codes are generated *semi-adaptively*, by compressing blocks of up to 64 Kbytes at a time. A code table of the *canonical* Huffman code is placed at the beginning of the compressed block. This means that GZip is not really a single-pass method, but since 64Kbytes can easily be kept in the main memory, the program needs to read the input only once (as was done in the block-wise Burrows-Wheeler technique).

The basic parsing technique in GZip is *greedy*. However, it offers a slower option to check, whether parsing a raw character + following match would result in better compression than the direct longest match.

GZip outperforms most other Ziv-Lempel methods in terms of compression effectiveness and decoding speed. One faster variant, called **LZRW1**, has been presented by Ross Williams. It maintains a hash table with only a single pointer per each entry, not a list. It thus remembers only the last occurrence of each trigram of symbols (disregarding collisions). Moreover, it updates the table only by the first three symbols of the real match. The compression performance is, of course, significantly worse than with gzip.

6.2. LZ78 family of adaptive dictionary methods

The drawback of the LZ77 approach is as follows: Choosing a small window does not yield the longest possible matches, whereas choosing a larger window gives longer matches, but also larger offset values and thereby longer codes. Another source of inefficiency is the fact that LZ77 actually reserves many codes for the same substring; one for each instance within the window. LZ78 tries to overcome these problems by avoiding the restricted window, and by building an *explicit* dictionary. It must be built by both encoder and decoder in an identical manner. The code sequence consists of <*index*, *char*> pairs, where *index* refers to the dictionary entry corresponding to the longest match, and *char* is the symbol to follow. This pair will also be the newest entry in the dictionary, i.e. the longest match extended with the next symbol. Thus the 'words' in the dictionary grow one symbol at a time. Index 0 is reserved for the no-match case, so that the coded substring consists of only *char*, which is added to the dictionary. We study the encoding using the following input message:

wabba-wabba-wabba-wabba-woo-woo-woo

Initially, the dictionary is empty, so the first three symbols must be encoded with index 0. Thereafter, the dictionary will contain these symbols, so that the second *b* is already found from there. It is extended with symbol *a*, and *ba* will be the next dictionary entry. Then we again meet a fresh symbol '-', and add it to the dictionary as such. Symbol *w* is found next, matching the dictionary entry 1, and extended with *a*, and so on. The whole sequence is developed in the table on the next page.

Although LZ78 is able to build the dictionary adaptively, it has the drawback that the size of the dictionary grows without bound. When the size reaches the reserved memory, then we either have to continue with the obtained fixed dictionary or prune it, or discard it and restart from an empty dictionary.

LZ78, in addition to being practical, has the important theoretical property that if the source message is generated by a *stationary*, *ergodic* source, then the compression performance is asymptotically *optimal* (i.e. the code size approaches entropy in the limit). An ergodic model means that any message produced by it becomes entirely representative of the source as the message length grows longer. Unfortunately, the convergence to the optimum is quite slow.

Lookahead buffer	Encoder output	Dictionary index	Dictionary entry
wabba-wabba-...	<0, w>	1	w
abba-wabba-w...	<0, a>	2	a
bba-wabba-wa...	<0, b>	3	b
ba-wabba-wab...	<3, a>	4	ba
-wabba-wabba...	<0, ->	5	-
wabba-wabba-...	<1, a>	6	wa
bba-wabba-wa...	<3, b>	7	bb
a-wabba-wabb...	<2, ->	8	a-
wabba-wabba-...	<6, b>	9	wab
ba-wabba-woo...	<4, ->	10	ba-
wabba-woo-wo...	<9, b>	11	wabb
a-woo-woo-wo..	<8, w>	12	a-w
oo-woo-woo	<0, o>	13	o
o-woo-woo	<13, ->	14	o-
woo-woo	<1, o>	15	wo
o-woo	<14, w>	16	o-w
oo	<13, o>	17	oo

LZW

There are many ways to modify the basic LZ78 technique. One of the most famous, called LZW, was suggested by Terry Welch[1]. It has a similar idea as LZSS had in modifying LZ77, namely discarding the character component from the code tuples. The encoded sequence in LZW thus consists of only indices to the dictionary. The dictionary must be initialized with all symbols of the alphabet. After finding a matching substring *x*, the next dictionary entry will be *x*|*a* where '*a*' is the first symbol of the next matching substring. In decoding, building of the dictionary proceeds 'one step behind', and may cause a situation where a substring is needed before it is stored in the dictionary. The problem and its solution are exemplified by encoding the sequence

aabababaaa

The dictionary is initialized with symbols *a* and *b*. The encoder first notices the longest match for '*aaba...*' to be '*a*', and adds '*aa*' to the dictionary. Then it finds the longest match for '*abab...*' which is again '*a*', and '*ab*' is added to the dictionary. The longest match for '*baba...*' is '*b*', and '*ba*' is added to the dictionary. The longest match for '*ababaaa*' is '*ab*', and '*aba*' is added to the dictionary, and so on. Thus, the first symbol of each parsed substring is the last symbol of the previously added entry to the dictionary. The whole dictionary will be as follows:

[1] T.A. Welch: "A Technique for High-Performance Data Compression", *IEEE Computer*, June 1984, pp. 8-19.

Index	Substring	Derived from
0	a	
1	b	
2	aa	0+a
3	ab	0+b
4	ba	1+a
5	aba	3+a
6	abaa	5+a

The decoder gets the sequence [0, 0, 1, 3, 5, 2]. Let us now try to follow the decoder's steps. After receiving a certain index, the decoder outputs the related substring, and adds an entry to the dictionary with the last symbol still unsolved. It will be completed only after seeing the next substring, the first symbol of which is the correct last symbol of the previously added dictionary entry. The first three indexes are trivial to decode, because they refer to singular symbols. Thereafter, the decoder's dictionary is {a, *b*, *aa*, *ab*, *b?*}. The next index is 3, which represents the substring '*ab*'. We can thus replace the '*b?*' by '*ba*', and add substring '*ab?*' as the 5th entry to the dictionary. Now we have the tricky situation where the index (5) obtained by the decoder should be used before the last symbol of the related entry is solved. The solution is, however, simple: We know that the unsolved symbol of the previously added dictionary entry starts the next substring. The next substring is '*ab?*', so its first symbol is '*a*', and this is the correct value for '?'. Now the decoder adds the entry '*aba?*' to the dictionary (entry 6). The last code index = 2 is simply decoded to '*aa*', and the job is finished.

LZW was originally proposed for compressing disk files with the aid of special channel hardware. In this situation speed is of utmost importance, and thus many implementations use a fixed maximum size for the dictionary, typically 4096 entries (= 12 bits for indexes). The dictionary remains static, after having grown to this size. There are three important practical variants of the LZW scheme, explained below.

Unix Compress (LZC)

One of the most popular file compression methods in Unix has been the *compress* program (*uncompress* is the decoder), called also LZC, where compressed files are denoted with '.Z' suffix. The indexes to the dictionary of size M are encoded using $\lceil \log_2 M \rceil$ bits, so that the code sizes grow with one bit always when the dictionary size exceeds some power of two. The maximum number of dictionary entries can be selected between 2^9 and 2^{16}. After reaching the maximum, the dictionary keeps static, but the system monitors the compression ratio. If it falls below a given threshold, the dictionary is discarded, and the building process is restarted. The method is thus also locally adaptive to some extent.

GIF

The Graphics Interchange Format (GIF) was developed by Compuserve Information Service as a method for encoding graphical images. It is very similar to the Unix compress program. The first byte of the compressed message contains the number b of bits per pixel in the original image (typically 8). The initial size of the dictionary is 2^{b+1} entries, of which $0..2^b-1$ are the original pixel values, 2^b is the so called *clear code*, which is used to reset the dictionary to the initial state. Another special codeword value is 2^b+1, which denotes end-of-information. When the dictionary is filled up, its size is doubled, up to the maximum of 4096

entries, after which it remains static. The coded data are represented in *blocks* of maximally 255 bytes. Each block is surrounded by a header (containing the length) and a terminator. The end-of-information code should precede the last terminator.

The performance of GIF is not especially good, except for drawings and other graphic images. The efficiency is close to arithmetic coding with no context information. The reason for inefficiency for photographic images is that image data is fundamentally different from text: it does not consist of regular phrases, but rather digitized representations of more or less continuous 2-dimensional surfaces, with some noise included. GIF images use a so called *indexed colour model*, i.e. a *palette* of colours, where visually close tones may have quite different indexes. Therefore, continuity of tone values is lost. The best image compression methods (such as JPEG, JPEG 2000) are based on colour models (such as RGB) supporting (sampled and quantified) continuity, and therefore methods applicable to digital signals can be used. However, these methods are typically *lossy*, i.e. part of the image information is lost. GIF compression is in principle lossless, but the step of converting the original image into indexed representation is lossy.

V.42 bis

The CCITT (now called ITU-T) Recommendation *V.42* is an error-correcting procedure for data transmission in telephone networks. *V.42 bis* is the related data compression standard. Actually, V.42 can operate in two modes, in compressed and in uncompressed (transparent) modes, because there may be types of data which cannot be compressed at all (and which would in fact extend in size, if compressed). The compressed mode uses a modified LZW algorithm. Again, the minimum dictionary size is 512 entries, and the recommendation is 2048, but the actual size is ‘negotiated’ between the sender and receiver. Three special codeword values are distinguished: 0 = enter transparent mode, 1 = flush the data, 2 = increment codeword size.

V.42 bis contains some modifications to the original LZW scheme: After the dictionary has grown to its maximum size, the method reuses those entries which are not prefixes to other dictionary entries. This means that these entries have not been used in coding, and seem to be of little value. Furthermore, it is wise to reuse first the oldest such entries. Due to creation order, they are found first when scanning from start to end.

To reduce the effect of errors, there is a certain maximum length for encoded substrings. CCITT recommends a range of 6 to 250, with default = 6 symbols.

A third modification concerns the special situation in LZW where the latest entry of the dictionary is used immediately, before its last symbol has been solved. V.42 bis simply forbids this kind of reference, but instead suggests to send the constituents of the last entry.

Some other versions of LZ78, suggested in the literature, are the following:

LZT (by P. Tischer, 1987) applies the *LRU* scheme to dynamically maintain a full dictionary, replacing the least recently used substrings by new ones.

LZMW (by V.S. Miller and M.N. Wegman, 1984) builds long substrings more quickly: It creates a new substring always by concatenating the last two matching substrings, not just adding a single character.

LZJ (by Matti Jakobsson, 1985) stores in the dictionary *all distinct* previously occurred substrings of the input, up to a certain maximum length h ($h = 6$ is sufficient). The maximum size M of the dictionary is also fixed, e.g. $M = 8192$ is suitable. After having reached this maximum size, the dictionary is pruned by replacing substrings that have occurred only once. It is peculiar to LZJ that the encoding is faster than decoding, making it suitable for archiving purposes.

LZFG (by E.R. Fiala and D.H. Greene, 1989) is one of the most practical LZ variants, involving many fine-tuned details. It can actually be considered a combination of the ideas of LZ77 and LZ78. It maintains a similar sliding window as LZ77, but stores in the dictionary only substrings that are prefixed by some previously matched substring. When the window is moved, substrings falling out are pruned from the dictionary. The data structure used for representing the dictionary is a *Patricia trie*, where a parent-child arc can represent a sequence of several symbols. The leaf nodes represent the whole remaining part of the window, so there is basically no upper bound for the match lengths. The encoded sequence consists of references to the trie nodes. If the match ends to a node whose parent arc represents only a single symbol, only the node number is needed. If the parent-child arc represents a longer path, another number must be used to specify the exact endpoint of the match.

There are many details in LZFG, which are out of the scope of the current discussion. However, the coding of numbers is interesting, and differs from our earlier 'universal' variable-length representations (α-, γ-, and δ-codes). Indexes of internal nodes (and possible match lengths) are encoded using a technique called *phasing-in*, where we know the range of integers $[0, m-1]$, but m need not be a power of 2. Two different codeword lengths are used, so that the numbers from 0 to $2^{\lceil \log m \rceil}-m-1$ are encoded with $\lfloor \log_2 m \rfloor$ bits, and numbers from $2^{\lceil \log m \rceil}-m$ to $m-1$ are encoded with $\lfloor \log_2 m \rfloor+1$ bits,

Another number coding system, namely the family of *Start-Step-Stop* codes, is used in LZFG. It is a kind of parameterized version of γ-code, and meant for a limited range of integers. It employs n different lengths of simple binary codes. Parameter *start* gives the number of bits used to represent the first 2^{start} numbers. The next set of codewords contains *start*+*step* bits each, with $2^{start+step}$ different values. The next contains *start*+2·*step* bits, and so on, until the final length *stop*. Each codeword is prefixed by the unary representation of the length; the final length (*stop*) can be represented without the ending '1' of the unary code. The number of different codes available is

$$\frac{2^{stop+step} - 2^{start}}{2^{step} - 1}$$

Since this grows exponentially with *stop*, relatively high numbers can be represented with small parameter values. For example, start-step-stop (1, 2, 5) code would have codewords 10, 11, 01000, 01001, 01010, 01011, 01100, 01101, 01110, 01111, 0000000, 0000001, ..., 0011111 for integers 0 to 41. The normal k-length binary code is a start-step-stop code with parameters $(k, 1, k)$. The γ-code, on the other hand, has parameters $(0, 1, \infty)$.

By tuning the three parameters, the start-step-stop code can be made to approximate the Huffman code of some observed distribution. This is just what has been done in LZFG. The leaf nodes are encoded with (10, 2, 14) code, and the related length components with (1, 1, 10) code. In addition, phasing-in can be applied to shorten the longest codes.

6.3. Performance comparison

We conclude this section by making a comparison between different methods in the LZ77 and LZ78 families, as well as the extreme version of statistical coding, namely PPMZ. The test data is again the Calgary Corpus. The results, in bits per source symbol, are as follows.

File	Size	LZ77	LZSS	LZH	GZIP	LZ78	LZW	LZJ’	LZFG	PPMZ
bib	111 261	3.75	3.35	3.24	2.51	3.95	3.84	3.63	2.90	1.74
book1	768 771	4.57	4.08	3.73	3.26	3.92	4.03	3.67	3.62	2.21
book2	610 856	3.93	3.41	3.34	2.70	3.81	4.52	3.94	3.05	1.87
geo	102 400	6.34	6.43	6.52	5.34	5.59	6.15	6.05	5.70	4.03
news	377 109	4.37	3.79	3.84	3.06	4.33	4.92	4.59	3.44	2.24
obj1	21 504	5.41	4.57	4.58	3.83	5.58	6.30	5.19	4.03	3.67
obj2	246 814	3.81	3.30	3.19	2.63	4.68	9.81	5.95	2.96	2.23
paper1	53 161	3.94	3.38	3.38	2.79	4.50	4.58	3.66	3.03	2.22
paper2	82 199	4.10	3.58	3.57	2.89	4.24	4.02	3.48	3.16	2.21
pic	513 216	2.22	1.67	1.04	0.82	1.13	1.09	2.40	0.87	0.79
progc	39 611	3.84	3.24	3.25	2.67	4.60	4.88	3.72	2.89	2.26
progl	71 646	2.90	2.37	2.20	1.81	3.77	3.89	3.09	1.97	1.47
progp	49 379	2.93	2.36	2.17	1.81	3.84	3.73	3.14	1.90	1.48
trans	93 695	2.98	2.44	2.12	1.61	3.92	4.24	3.52	1.76	1.24
average	**224 402**	**3.94**	**3.43**	**3.30**	**2.70**	**4.13**	**4.71**	**4.00**	**2.95**	**2.12**

The general observations are:

- The LZ77 family seems to be slightly better that LZ78 family.
- GZip represents a compression-effective implementation of LZ77.
- LZFG represents a compression-effective implementation of LZ78.
- Both LZ77 and LZ78 families are far behind the best predictive methods.

The trade-off between compression performance and processing time explains the popularity of LZ-methods in practice. Generally, encoding in the LZ78 family seems to be somewhat faster, but decoding slower than in the LZ77 family. The primary reasons are that an LZ78 encoder finds the matches faster than an LZ77 encoder, but an LZ78 decoder must rebuild the dictionary, whereas an LZ77 decoder just copies the substrings referred to. Both, in turn, are an order of magnitude faster than, say, PPMZ, used for comparison. For example, GZip has been observed to be about 23 times faster than PPMZ for the Calgary Corpus, even excluding the ‘pic’ file (which took several hours for PPMZ).

7. Introduction to Image Compression

The compression methods described in earlier sections can in principle be applied to image data, as well, by transforming the 2-dimensional collection of image elements (most often *pixels*) into a 1-dimensional sequence of symbols. However, image data has so special characteristics that the compression performance would not be too good. An example is the GIF format for image representation. The problem is with the *modelling* part of compression. Huffman and arithmetic coders are, of course, still of utmost importance.

Photographic images are digitized representations of more or less continuous color / brightness 'surfaces', a kind of 2-dimensional signals, where the numeric value of a pixel is highly correlated with the numerical values of its neighbouring pixels. Among different images, it is much harder to find unifying characteristics than, say, among English documents. Image modelling is concerned with geometric properties and local mathematical dependencies in the source, rather than symbol patterns. In simple terms, text and images represent different scales of measurement: *nominal* vs. *interval* / *ratio* scale.

Digitized (raster) images can be classified into the following categories:

- *Bi-level* images contain only two pixel values; 0=white and 1=black. They are often called also black-and-white images.
- *Grey-scale* images have typically 256 different levels of greyness from white to black. Thus, a pixel is represented by an 8-bit byte.
- *Color* images have three color components (e.g. red, green, blue), and each is represented as a grey-scale image. Thus, each pixel takes 24 bits.

Image compression can be divided into two categories: *lossless* an *lossy* compression. The latter means that only an approximation of the original image can be recovered. In what follows, we discuss some basic methods in both categories, concentrating on the former.

7.1. Lossless compression of bi-level images

There are many applications where black-and-white images are quite sufficient. Any printed document can be interpreted as a bi-level image, e.g. in telefax transmission. From compression point of view, symbolic representation of the text on the page would yield a much higher efficiency. On the other hand, the image form is much more flexible; handwritten text is as easy to process as typewritten.

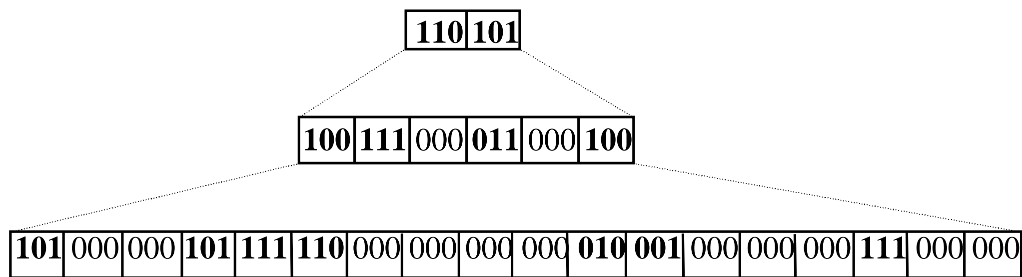
The basis of compressing bi-level images is twofold:

- The proportions of black and white pixels are usually quite different. Typically, there are more white than black pixels.
- The pixels of the same color are highly *clustered*. We can distinguish subareas of the image that are totally white or totally black.

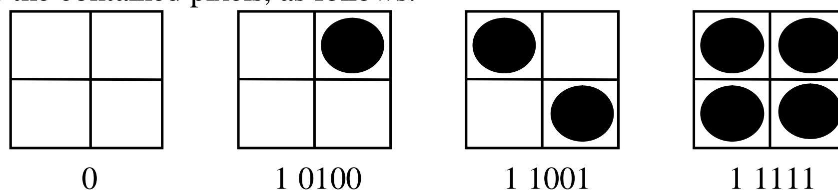
Any methods developed for *bit-vector* compression are useful here, because they can be applied to rows (or columns) of the image. One of the simplest such methods is *run-length coding*, where the length of each run of zeroes / ones is encoded. Since white and black runs alternate, only the lengths are needed, assuming some starting color. Actually, the method should be called 'run-length modelling'; the coding of lengths can be done using any of the

aforementioned techniques: Huffman, arithmetic, universal integer encoding, etc. A considerable improvement to compression efficiency can be obtained by *predictive run-length coding*. Each pixel value (0/1) is predicted on the basis of a few of its neighbours above and left (west, northwest, north, northeast). The original bi-level image is transformed to a so called *error image*, containing a bit for each pixel: 0, if prediction was successful, and 1, if not. According to experiments, the proportion of ones reduces to one third, and compression improves by about 30%. The decoder performs the reverse transformation (while the same predictive neighbours are known to it, as well).

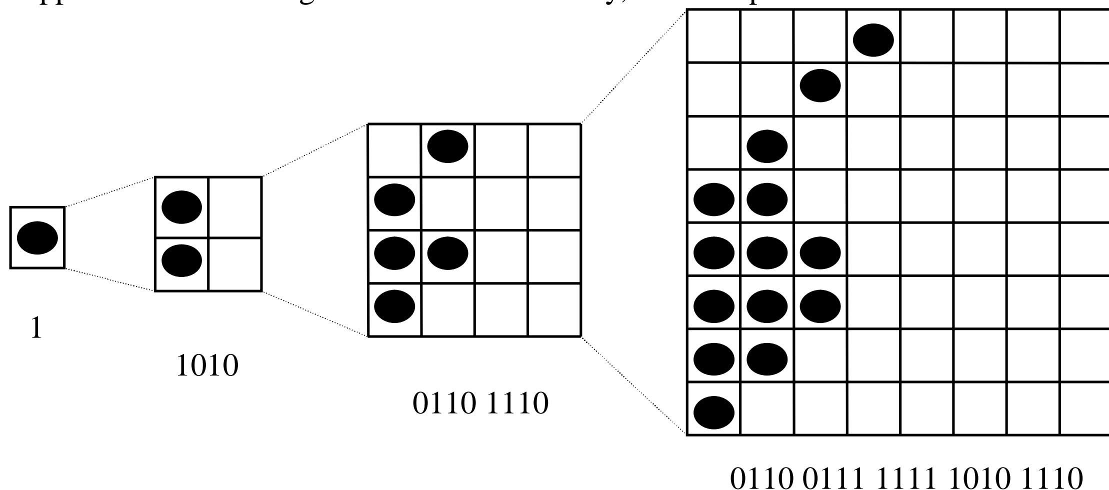
Another simple method for bit-vector compression is *block coding*, where the vector is divided into fixed-length blocks. A completely white block is encoded as a single zero, whereas other blocks are encoded as 1 plus the block contents. This is efficient enough for almost white images. It can be generalized to *hierarchical block coding*, where the flag bits are compressed by the same technique, recursively. The following example with block size = 3 illustrates the two-level case. Bits in bold are transmitted to the decoder.



The same idea can still be generalized into two dimensions, so that the image area is divided into rectangular boxes, and completely white boxes are encoded with 0, and other blocks with 1 plus the contained pixels, as follows:



This approach can also be generalized hierarchically, for example



If the proportion of black areas is relatively large, it might be more efficient to reserve a short code also for completely black blocks. Further compression (of the order 10%) is achieved by e.g. Huffman coding of the 2×2 blocks. A static codebook is sufficient in practical

applications. More improvement can be obtained by applying a similar prediction as was suggested in predictive run-length coding.

One of the earliest application areas of bi-level image compression has been the *facsimile* or *fax*. The number of pixels per sheet is either 1728×1188 (low resolution), or 1827×2376 (high resolution). CCITT (now ITU-T) has classified the fax equipment into four groups:

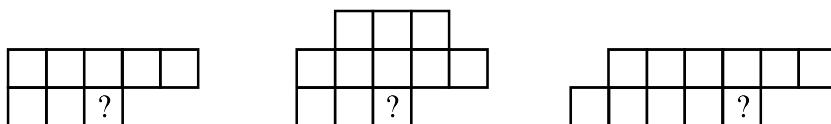
- Group 1: Analog scheme, speed ≤ 6 min / A4 sheet.
- Group 2: Analog scheme, speed ≤ 3 min / A4 sheet
- Group 3: Digital scheme, applies 1- or 2-dim. compression, speed ≤ 1 min /A4 sheet.
- Group 4: Digital scheme, applies 2-dim. compression, speed ≤ 1 min / A4 sheet.

The 1-dimensional scheme of group 3 performs coding independently for each line, using run-length coding. The lengths are Huffman coded, but not as such; they are expressed as two base-64 numbers:

$$length = 64 \times m + t$$

where $t = 0, ..., 63$ and $m = 0, ..., 27$. There are separate Huffman codes for black and white run lengths. The 2-dimensional option uses a relatively complicated scheme of coding a line on the basis of the previous line. The method is a modification of so called *Relative Element Address Designate* (*READ*); the details are skipped here. To prevent error propagation, the group 3 compression restricts the number of lines compressed with the 2-dimensional scheme. In normal resolution, every other line is compressed by the 1-dimensional scheme, and in high resolution every fourth. Group 4 uses only the 2-dimensional scheme.

The standard for bi-level image compression has been developed by **JBIG** (Joint Bi-Level Image Compression Group, 1993), which was a joint experts group of ISO (International Standards Organization), CCITT (Consultative Committee on International Telephone and Telegraph) and IEC (International Electrotechnical Commission). In JBIG, the encoding of a pixel is based on a *context* of either 7 or 10 neighbouring pixels, which have already been coded (and known to the decoder). In addition, for 10-pixel contexts, there are two-line and three-line options:



There are 128 different 7-pixel contexts and 1024 different 10-pixel contexts. JBIG method gathers statistics from all of them, and uses 128 or 1024 different *QM-coders* for final encoding. The larger context is in principle more accurate, but converges more slowly.

In addition to the normal *sequential mode* (row by row), JBIG offers also a so called *progressive mode* of operation, where first a low-resolution image is sent. Then the resolution is doubled repeatedly until the highest resolution. The next level takes a few bits from the previous level to its context. The details of progressive transmission are skipped.

There is a newer version of JBIG, namely JBIG-2 (from 2000) which is becoming quite popular. It is meant, in addition to telefax, also for storage of document images, wireless transmission, printer spooling, etc. It compresses even more effectively than the old JBIG by dividing the image into three kinds of regions:

- *Symbol regions* contain mainly text, i.e. regular shapes, coded using a dictionary method. The dictionary entries consist of binary patterns to be matched with the bi-level image fragment. The method allows approximate matching and thereby *lossy* compression.
- *Halftone regions* contain halftone (raster) images, and compressed also with a dictionary that offers bit patterns for halftone elements.
- *Generic regions* represent the rest, compressed using the traditional predictive method.

7.2. Lossless compression of grey-scale images

The compression of grey-scale (and color) images is in most applications allowed to be *lossy*, i.e. the decoder does not recover the image into exactly the original form, but into an approximation, which is considered to be similar enough for the human eye. Some critical applications, such as x-rays in medicine, demand lossless compression.

The above methods for bi-level images could in principle be used also for grey-scale images. If each pixel is represented by 8 bits, we can distinguish 8 *bit planes* of the original image, each of which is a bi-level image. In fact, by sending the planes in order from the most to least significant, we obtain a *progressive* compression method for grey-scale images. Still better results are obtained if the pixel codes are transformed into *Gray codes* before partitioning the image into bit planes. Gray code has the property that codewords of two successive integers differ at only one position. Another improvement is obtained by including some pixels from the previous bit plane in the contexts of the current plane.

Most of the lossless compression methods for grey-scale images apply *prediction*. The pixels are processed line by line from left to right, in so called *row-major order* (*raster scan order*). Since the grey-levels tend to change smoothly, a few neighbouring pixels (among the already processed ones) are sufficient to make a good prediction of the current pixel. The difference between real and predicted values (called *prediction error* or *residual*) is encoded and transmitted to the decoder, which does the same prediction and recovers the real pixel value on the basis of the coded difference. If the prediction function is a linear combination of the neighbour values, the method is said to use a *linear prediction model*.

The standard for grey-scale (and color) image compression has been defined by **JPEG** (Joint Photographic Experts Group, 1991). It offers both lossy and lossless modes of compression. The latter applies the linear prediction model, by offering seven alternative prediction functions (actually eight, of which the first makes no prediction). By denoting the predicted pixel by x and its three neighbours in the north, west and northwest by N, W and NW, the prediction formulas are:

NW	N
W	X

1. $X = N$
2. $X = W$
3. $X = NW$
4. $x = N + W - NW$
5. $x = W + (N - NW) / 2$
6. $x = N + (W - NW) / 2$
7. $x = (N + W) / 2$

The prediction errors are coded either by Huffman or arithmetic coding. Typical compression performance in lossless mode for photographic images is around 50%, i.e. 4 bits per pixel. This seems to be a relatively low gain, compared e.g. to text compression. One reason is that ‘natural’ images usually contain so called *texture*, some kind of noise, which makes compression difficult, but which gives the image its natural character. On the other hand,

lossy JPEG can in most cases achieve compression of less than 1 bit per pixel, while the decoded result is still practically indistinguishable from the original.

The speed of compression can be improved, if we replace the final coder by some non-optimal method, which assumes a static (skew) distribution of prediction errors.

By using a larger context (above and left of the predicted pixel, so that the decoder can repeat the same prediction), one can get a more precise prediction, at the cost of processing time. A good example of this kind of extension is **CALIC** ("Context Adaptive Lossless Image Compression", 1994). It uses a seven-pixel context and tries to deduce, which direction would be best to use in prediction. In other words, it tries to find horizontal/vertical similarities among the context pixels. If it seems that there is a horizontal line, then the W neighbour might be best, while for a vertical line N would be a better choice.

		NN	NNE
	NW	N	NE
WW	W	X	

A later variant of lossless grey-scale image compression is **JPEG-LS** (from late 1990's), which can be regarded as a simplification of CALIC. It was originally developed by HP under the name LOCO-I. It applies a two-step prediction with a 4-pixel context (W, NW, N, NE). The first estimate for the predicted symbol is the median of W, N and NW pixels. This is then refined by the average error made earlier for the same 4-pixel context. The compression results are better than with the old lossless JPEG, but slightly worse than those obtained by (the more complicated) CALIC.

As a final note about lossless image compression, the modelling differs essential from lossy image compression (to be discussed next). As we saw, lossless models resemble predictive methods in text compression, even though closeness of the numerical values of neighbors was utilized also. The extremely popular image formats GIF (Graphics Interchange Format) and PNG (Portable Network Graphics) use methods borrowed from text compression: GIF is based on LZ78 and PNG uses a modification of LZ77, combined with prediction.

7.3. Lossy image compression: JPEG

Using lossless techniques, it is hard to compress photographic images much below 50%. This is due the small variations (noise) among neighboring pixel values, even in a very smooth area in the picture. On the other hand, in most applications it is not absolutely necessary to recover the original image *precisely*, as long as a human observer cannot tell the difference – or at least it is not disturbing. By relaxing somewhat from the strict preservation requirement, it is possible to get much higher compression gains. Because of the large size of images, most applications use lossy image compression. An example where this is not allowed is medical imaging, for example x-rays.

The best-known and most-used standard for lossy image compression is JPEG. It is a result of committee work (Joint Photographic Experts Group, 1986-92, ISO standard 1994). It uses quite a different approach to modeling the picture, compared to lossless JPEG, namely conversion to frequency domain by a discrete, two-dimensional *cosine transform* (*DCT*). It resembles the better-known *Fourier transform*, which is the standard tool in signal processing. Since images can be considered 2-dimensional signals, it is not surprising that similar techniques apply. The cosine transformation converts the image to a linear combination of cosine functions of different wavelengths (frequencies). The reason why this helps in compression is that the smooth or smoothly changing areas can be represented with only a few non-zero coefficients for the low-frequency components. The areas with rapid changes, on the other hand, can be modeled more roughly (with higher error), because a human observer is not so sensitive to those. Conclusively, the transform converts the image pixel matrix into a coefficient matrix of the same size, but with quite skewed element values. The transform is of course reversible, enabling decoding.

Let us study the technique a bit more detailed, and start with *grey-scale* images, the pixel values of which are typically 8 bits, with the range 0..255. The image is partitioned into blocks of 8x8 pixels, which are the units of transform. The reasons for not applying the transform to the whole image are twofold. First, the transform is a relatively heavy operation, and second, the regularities in the image are usually local, so that using larger units would not pay. Nevertheless, the neighbors across block boundaries correlate, which is not taken advantage of in compression (except for the lowest-frequency component, which is encoded on the basis of the corresponding component of the previous neighbor.

The 2-D discrete cosine transform of NxN block $f(x, y)$ is obtained by the following formula:

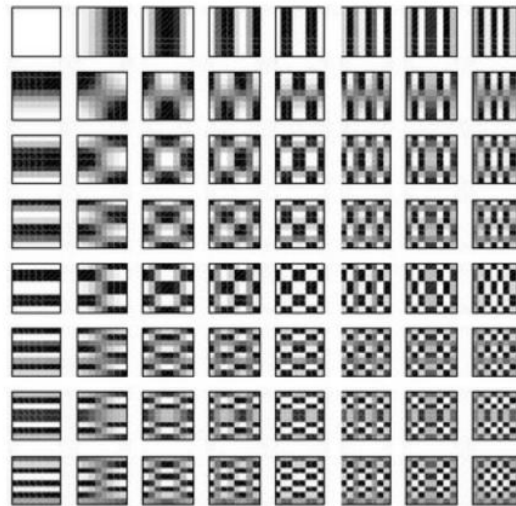
$$C(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\left\{ f(x,y)\cdot\cos\frac{(2x+1)u\pi}{2N}\cdot\cos\frac{(2y+1)v\pi}{2N}\right\}$$

and the reverse transformation by a very similar formula:

$$f(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1}\left\{ \alpha(u)\alpha(v)C(u,v)\cdot\cos\frac{(2x+1)u\pi}{2N}\cdot\cos\frac{(2y+1)v\pi}{2N}\right\}$$

Above the constants were: N=8, $\alpha(u) = \sqrt{1/N}$ for u=0, and $\sqrt{2/N}$ for u>0.

Note that the transformation is in principle *lossless*, if the calculations are done with infinite precision. The coefficients $C(u, v)$ represent the weights of the 64 2-D basis functions, so that their linear combination recovers the original block $f(x, y)$. The basis functions are illustrated below.



A numeric example of a block transform is shown below (example borrowed from http://en.wikipedia.org/wiki/JPEG):

Original:

$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix}$$

Transform result:

$$\begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix}$$

The coefficients in the top-left corner represent the low-frequency components, and are more important to the visual quality of the image than the coefficients in the bottom-right corner, representing the highest frequencies. This difference is taken into consideration in the next step, namely *quantization* of coefficients. It is the only lossy step in the process, and means reducing the precision of coefficients by dividing them with suitable integers. The JPEG standard defines quantization matrices for different image qualities. The common feature is that the divisors are smallest for the top-left coefficients and largest for the bottom-right coefficients. An example quantization matrix, and related quantization result of the above block are as follows (example borrowed again from http://en.wikipedia.org/wiki/JPEG):

Quantization matrix

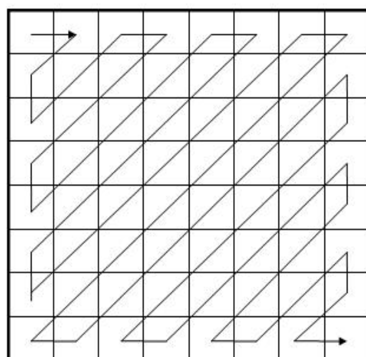
$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Quantization result

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The fact that the largest values are concentrated on the top-left corner is typical. It is also typical that the majority of the other values are zero, especially with higher compression ratios. This illustrates concretely the importance of low-frequency components in the cosine transform, and the reason why compression really works: the distribution is *skew*, and zeroes dominate (cf. the result of Burrows-Wheeler transform, followed by move-to-front transform).

The quantized numbers could be coded with arithmetic of Huffman coding, of which the latter is more common. Before coding, the obtained matrix is *linearized* into a 1-dimensional vector of 64 values. Due to the diagonally-oriented distribution, linearization is not perform row-by-row or column-by-column, but diagonally, in a so-called *zig-zag* order:



The previous coefficients constitute the following linearized vector:

[−26, −3, 0, −3, −2, −6, 2, −4, 1, −4, 1, 1, 5, 1, 2, −1, 1, −1, 2, 0, 0, 0, 0, 0, −1, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

The first (top-left) coefficient (so-called *DC-coefficient*) is encoded separately from others, because it has different statistics. It represents the average grey-value of the block, and is correlated with the DC-coefficients of the neighboring blocks. Therefore, *differences* from neighbors are encoded, instead, using Huffman method. The 63 other values are not encoded as such, but *run-length coding* is applied, in the following way: The alphabet for Huffman coding consists of *pairs* of the following components:

- Number of zeroes before the next non-zero element
- Number of *significant bits* in the non-zero element

The former is restricted to values 0..15, and the latter to 1..10, resulting in 160 combinations. Two special pairs are:

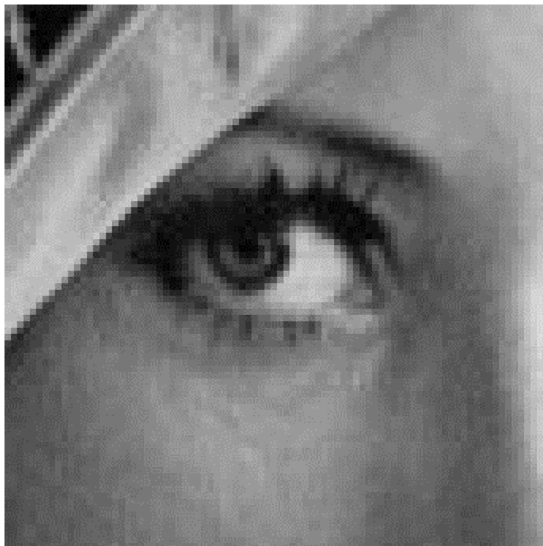
- <0, 0> that represents EOB = end-of-block
- <15, 0> that represents a sequence of zeroes without an ending non-zero element

The reason for considering the pairs as symbols of the alphabet is that the components of the pair are highly correlated: A long sequence of zeroes is probably followed by a small (rather than large) non-zero value. Default Huffman tables are defined in the JPEG standard.

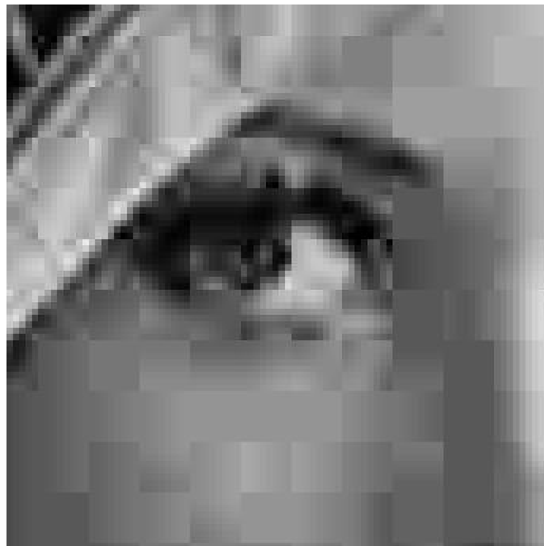
In addition to coding the pairs, also the non-zero values must be represented, but for that no special code is needed, because the number of bits for them is known from the second component of the related pair. Simple signed base-2 representation is sufficient (values can be positive or negative).

JPEG-decoding proceeds in the opposite direction. After Huffman decoding, the quantized coefficient matrices can be recovered by reversing the zig-zag scanning of elements. *Dequantization* means multiplying the matrix elements by the corresponding elements of the quantization matrix (see the matrices at the top of page 77). The resulting values represent approximations of the original coefficients. Finally, the inverse of DCT-transform returns the block of pixel values, which are close to the original. An example shows the effect (so-called blocking artefacts) of too large quantization values.

Compression ratio 7:1



Compression ratio 30:1



Until now, we have assumed a monochrome, grey-scale image. *Color images* are based on some color model, which most often consists of three components. The best-known color model is Red-Green-Blue (RGB), used in monitors. It is not advantageous to compress the three component images separately, because there are high correlations between them. Therefore, JPEG first applies a transform to a so-called *YUV* model (*YIQ* in North-America), which distinguishes the *luminance* component from two *chrominance* components. These three images are less correlated. An example (Lena) below shows how the ‘energy’ in the image is concentrated on the luminance component image.

Moreover, because the human visual system is less sensitive to changes of color than changes of brightness, the two chrominance components are usually *subsampled* by 2, i.e. they constitute only one quarter of the original image size, each. Subsampling is already a *lossy* compression step. The three component images are compressed with the above described method, with certain tunings.

As a conclusion from the studied image compression techniques, we can make the following general observations and recommendations:

- Digitized representations of continuous physical phenomena need totally different modeling techniques from text-type data.
- The *correlations* within the source datasets should be carefully analyzed and localized. One should try to use the correlations in *transforming* the dataset into another form where the distribution of values is as skew as possible, and where the components are decorrelated.
- When developing a lossy method, the loss should be positioned to a place where it is not easily noticed.
- In entropy coding, there are lots of choices for the alphabet. For Huffman coding, one should choose symbols, possibly artificial groups of symbols/values, which are not mutually correlated.
- For Huffman coding, a large (extended) alphabet works often better than small.
- Predefined (static) Huffman tables should be used, if possible, for speed reasons, and to avoid sending the tables to the decoder.

Image compression is a very large subject area, which would deserve a course of its own. The same holds for digital video, which would be quite impractical without compression.