

Data Compaction

This first, rather elementary chapter deals with *non-redundant* representation of information; in other words, we shall treat data compaction codes (i.e. algorithms for lossless data compression). More common, eventually lossy, data compression needs arguments and methods from signal theory, and will be considered in the last chapter of this book.

1.1 Entropy Coding

All coding methods that we shall encounter in this section are based on a preliminary statistical evaluation of our set of data. In a certain sense, the coding algorithms will treat the statistical profile of the data set rather than the data itself. Since we are only interested in coding methods, we shall always feel free to assume that the statistics we need are plainly at our disposal – so that our algorithms will run correctly.

Note that our probabilistic language is part of the tradition of information theory – which has always been considered as a peripheral discipline of probability theory. But you are perfectly allowed to think and argue in a purely deterministic way: the statistical evaluation of the data for compaction can be thought of as a specification of parameters – in the same way as the choice of the right number of nodes (or of the correct sampling frequency) in interpolation theory.

A historical remark: do not forget that almost all good ideas and clever constructions in this section have come to light between 1948 and 1952.

1.1.1 Discrete Sources and Their Entropy

We shall consider *memoryless* discrete sources, producing words (strings of letters, of symbols, of characters) in an alphabet $\{a_0, a_1, \dots, a_{N-1}\}$ of N symbols.

We shall call

$p_j = p(a_j) \equiv$ the probability of (the production of) the letter a_j , $0 \leq j \leq N - 1$.

Notation $\mathbf{p} = (p_0, p_1, \dots, p_{N-1}) \equiv$ the probability distribution which describes the production of our source.

- *Regarding the alphabet:* think of $\{0, 1\}$ (a binary source: for example, a binary facsimile image) or of $\{00000000, 00000001, \dots, 11111111\}$ (a source of 256 symbols, in 8-bit byte representation: for example, the ASCII character code).
- *Regarding the memoryless production:* this is a condition of probabilistic modelling which is *very strong*. Namely:

For a word $\mathbf{w} = a_{j_1} a_{j_2} \cdots a_{j_n}$ of length n , the *statistically independent* production of its letters at any moment is expressed by the identity

$$p(\mathbf{w}) = p(a_{j_1})p(a_{j_2}) \cdots p(a_{j_n}).$$

This identity (the probability of a word is the product of the probabilities of its letters) models the production of our source by the iterated roll of a loaded dice, the faces of which are the letters of our alphabet – with the probability distribution \mathbf{p} describing the outcome of our experience.

Note that this rather simple modelling has its virtues beyond simplicity: it describes the ugliest situation for data compression (which should improve according to the degree of correlation in the production of our data), thus meeting the demands of an austere and cautious design.

At any rate, we now dispose of an easy control for modelling – having a sort of “commutation rule” that permits us to decide what should be a *letter*, i.e. an atom of our alphabet. For a given binary source, for example, the words 01 and 10 may have sensibly different frequencies. It is evident that this source cannot be considered as a memoryless source for the alphabet $\{0, 1\}$; but a deeper statistical evaluation may show that we are permitted to consider it as a memoryless source over the alphabet of 8-bit bytes.

The Entropy of a Source

The entropy of a (discrete) source will be the *average information content* of a “generic” symbol produced by the source (measured in bits per symbol).

Let us insist on the practical philosophy behind this notion: you should think of entropy as a *scaling factor towards (minimal) bit-representation*: 1,000 symbols produced by the source (according to the statistics) “are worth” $1,000 \times \text{entropy bits}$.

Prelude *The information content of a message.*

Let $I(\mathbf{w})$ be the quantity of information contained in a word \mathbf{w} that is produced by our source. We search a definition for $I(\mathbf{w})$ which satisfies the following two conditions:

- (1) $I(\mathbf{w})$ is inversely proportional to the probability $p(\mathbf{w})$ of the production of \mathbf{w} (“the less it is frequent, the more it is interesting”).
Moreover, we want the information content of a *sure* event to be *zero*.
- (2) $I(a_{j_1} a_{j_2} \cdots a_{j_n}) = I(a_{j_1}) + I(a_{j_2}) + \cdots + I(a_{j_n})$ (the information content of a word is the sum of the information contents of its letters – this stems from our hypothesis on the statistical independence in the production of the letters).

Passing to the synthesis of (1) and (2), we arrive at the following condition:

$I(\mathbf{w}) = F\left(\frac{1}{p(\mathbf{w})}\right)$ where the real function F has to be strictly monotone (increasing) and must satisfy the identity $F(x \cdot y) = F(x) + F(y)$ as well as $F(1) = 0$.

Now, there is essentially only one (continuous) function F which satisfies our conditions: the *logarithm*.

Thus the following definition comes up naturally.

Definition $I(\mathbf{w}) = \text{Log}_2\left(\frac{1}{p(\mathbf{w})}\right) = -\text{Log}_2 p(\mathbf{w})$.

$$\left[\text{Recall: } y = \text{Log}_2 x \iff x = 2^y \iff x = e^{y \text{Ln } 2} \iff y = \frac{\text{Ln } x}{\text{Ln } 2} \right]$$

But why the logarithm to the base 2?

Answer We want the unity of the information content to be the bit.

Let us make things clearer with two examples.

- (a) Consider a source which produces a_0 = heads and a_1 = tails with the same probability ($\mathbf{p} = (p_0, p_1) = (\frac{1}{2}, \frac{1}{2})$). We get $I(a_0) = I(a_1) = -\text{Log}_2 2^{-1} = 1$.
That is logical: when tossing coins with equal chance, heads is naturally coded by 0 and tails is naturally coded by 1.
- (b) Let us pursue this line of thought: now, our outcome will be the 256 integers between 0 and 255 (rolling a very, very big dice with 256 faces), all of equal chance: $\mathbf{p} = (p_0, p_1, \dots, p_{255}) = (\frac{1}{256}, \frac{1}{256}, \dots, \frac{1}{256})$. $I(a_0) = I(a_1) = \dots = I(a_{255}) = -\text{Log}_2 2^{-8} = 8$. Once more: no surprise; assuming equal chance, the information content of any of the integers 0, 1, ..., 255 has to be 8 bits: they are 8-bit bytes!

But back to our source:

Let $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$ be the probability distribution which describes the (memoryless) production of the letters of our alphabet.

Definition of the *entropy* of the source:

$H(\mathbf{p}) \equiv$ the average quantity of information per symbol (in bits per symbol)

$$\begin{aligned} &= p_0 I_0 + p_1 I_1 + \cdots + p_{N-1} I_{N-1} = -p_0 \text{Log}_2 p_0 - p_1 \text{Log}_2 p_1 \\ &\quad - \cdots - p_{N-1} \text{Log}_2 p_{N-1}. \end{aligned}$$

Exercises

- (1) Compute the entropy of the source which produces the eight letters a_0, a_1, \dots, a_7 , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_7)$ with $p_0 = \frac{1}{2}, p_1 = \frac{1}{4}, p_2 = p_3 = \frac{1}{16}, p_4 = p_5 = p_6 = p_7 = \frac{1}{32}$.
- (2) Let us consider a memoryless source which produces four letters a_0, a_1, a_2, a_3 , according to the probability distribution $\mathbf{p} = (p_0, p_1, p_2, p_3)$. *Let us change our viewpoint.* Consider the source as a producer of the 16 symbols $a_0a_0, a_0a_1, \dots, a_3a_2, a_3a_3$, according to the product distribution $\mathbf{p}^{(2)} = (p_{00}, p_{01}, \dots, p_{23}, p_{33})$ with $p_{ij} = p_i p_j$, $0 \leq i, j \leq 3$. Show that $H(\mathbf{p}^{(2)}) = 2H(\mathbf{p})$. Generalize.

Remarks

Our situation: the alphabet $\{a_0, a_1, \dots, a_{N-1}\}$ will remain fixed; we shall vary the probability distributions...

- (1) $H(\mathbf{p}) = 0 \iff$ The source produces effectively only one letter
(for example the letter a_0), with $p(a_0) = 1$.
Recall: a sure event has information content zero.
Hence: the entropy will be *minimal* (will be zero) as a characteristic of a constant source production.
Thus, we extrapolate (and we are right):
- (2) $H(\mathbf{p})$ is maximal $\iff p_0 = p_1 = \dots = p_{N-1} = \frac{1}{N}$.
In this case, we have $H(\mathbf{p}) = \text{Log}_2 N$.

Exercises

- (1) A binary source produces a_0 = white and a_1 = black according to the probability distribution $\mathbf{p} = (p_0, p_1)$.
Find the condition on the ratio white/black which characterizes $H(\mathbf{p}) < \frac{1}{2}$.
- (2) Gibbs' inequality.
Consider $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$ and $\mathbf{q} = (q_0, q_1, \dots, q_{N-1})$, two strictly positive probability distributions (no probability value is zero).
 - (a) Show that $-\sum_{j=0}^{N-1} p_j \text{Log}_2 p_j \leq -\sum_{j=0}^{N-1} p_j \text{Log}_2 q_j$.
 - (b) Show that the inequality above is an equality $\iff \mathbf{p} = \mathbf{q}$.
 - (c) Deduce from (b): every probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$ satisfies $H(\mathbf{p}) \leq \text{Log}_2 N$ with equality $\iff p_0 = p_1 = \dots = p_{N-1} = \frac{1}{N}$.

[*Hint:*

- (a) *Recall:* $\text{Ln } x \leq x - 1$ for all $x > 0$ with equality $\iff x = 1$.

(b) You should get from (a) the following inequality

$$\sum_{j=0}^{N-1} p_j \left(\ln \frac{q_j}{p_j} - \left(\frac{q_j}{p_j} - 1 \right) \right) \leq 0,$$

where all the terms of the sum are non-positive. This is the clue.]

Entropy Coding, A First Approach

Consider a memoryless source which produces the N symbols a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.

We have seen: every letter a_j “is worth” $I(a_j)$ bits, $0 \leq j \leq N-1$.

This leads to the *natural idea* (Shannon (1948)): associate with the symbols of our alphabet binary code words of variable length in such a way that the length of a code word associated with a letter is precisely the information content of this letter (assume first that all probabilities are powers of 2, so that the information contents will be correctly integers).

More precisely:

Let l_j be the length (the number of bits) of the code word associated to the letter a_j , $0 \leq j \leq N-1$.

Our choice: $l_j = I(a_j)$, $0 \leq j \leq N-1$.

Let us look at the *average length* \bar{l} of the code words:

$$\bar{l} = p_0 l_0 + p_1 l_1 + \dots + p_{N-1} l_{N-1}.$$

Note that \bar{l} is a *scaling factor*: our encoder will transform 1,000 symbols produced by the source (in conformity with the statistics used for the construction of the code) into $1,000 \times \bar{l}$ bits.

But, since we were able to choose $l_j = I(a_j)$, $0 \leq j \leq N-1$, we shall get

$$\bar{l} = H(\mathbf{p}).$$

Example Recall the source which produces the eight letters a_0, a_1, \dots, a_7 , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_7)$ with $p_0 = \frac{1}{2}, p_1 = \frac{1}{4}, p_2 = p_3 = \frac{1}{16}, p_4 = p_5 = p_6 = p_7 = \frac{1}{32}$.

This means: $I(a_0) = 1, I(a_1) = 2, I(a_2) = I(a_3) = 4, I(a_4) = I(a_5) = I(a_6) = I(a_7) = 5$.

We choose the following encoding:

$$\begin{array}{ll} a_0 \mapsto 0 & a_4 \mapsto 11100 \\ a_1 \mapsto 10 & a_5 \mapsto 11101 \\ a_2 \mapsto 1100 & a_6 \mapsto 11110 \\ a_3 \mapsto 1101 & a_7 \mapsto 11111 \end{array}$$

Encoding *without statistics*, i.e. assuming equal chance, will oblige us to reserve three bits for any of the eight letters. On the other hand, with our code, we obtain $\bar{l} = H(\mathbf{p}) = 2.125$.

Let us insist: without statistical evaluation, 10,000 source symbols have to be transformed into 30,000 bits. With our encoder, based on the statistics of the source, we will transform 10,000 letters (produced in conformity with the statistics) into 21,250 bits. Manifestly, we have *compressed*.

Important remark concerning the choice of the code words in the example above.

Inspecting the list of our eight code words, we note that *no code word is the prefix of another code word*. We have constructed what is called a *binary prefix code*. In order to understand the practical importance of this notion, let us look at the following example:

$$A \mapsto 0 \quad B \mapsto 01 \quad C \mapsto 10.$$

Let us try to decode 001010. We realize that there are three possibilities: *AACC*, *ABAC*, *ABBA*. The ambiguity of the decoding comes from the fact that the code word for *A* is the prefix of the code word for *B*. But look at our example: there is no problem to decode

$$0110100111011111110001000 \text{ back to } a_0a_3a_0a_0a_5a_7a_2a_0a_1a_0a_0.$$

McMillan (1956) has shown that every variable length binary code that admits a *unique decoding algorithm* is isomorphic to a *prefix code*. This will be the reason for our loyalty to prefix codes in the sequel.

1.1.2 Towards Huffman Coding

In this section we shall recount the first explosion of ideas in information theory, between 1948 and 1952. Everything will begin with Claude Shannon, the founder of the theory, and will finally attain its “price of elegance” with the algorithm of Huffman, in 1952.

Do not forget that the theory we shall expose is built upon the rather restrictive hypothesis of a *memoryless source*.¹

The Kraft Inequality and its Consequences

Let us consider a memoryless source producing N letters, a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.

Shannon’s coding paradigm. Associate to a_0, a_1, \dots, a_{N-1} words of a binary code, such that the lengths l_0, l_1, \dots, l_{N-1} , of the code words *will correspond* to the information contents of the encoded symbols.

We need to make precise the term “*will correspond* to the information contents of the encoded symbols”.

We aim at

$$l_j \approx I(a_j) = -\text{Log}_2 p_j, \quad 0 \leq j \leq N-1.$$

¹ One can do better – but there are convincing practical arguments for simple modelling.

More precisely, we put

$$l_j = \lceil I(a_j) \rceil = \lceil -\log_2 p_j \rceil, \quad 0 \leq j \leq N-1,$$

where $\lceil \cdot \rceil$ means rounding up to the next integer.

Our first problem will now be the following:

Is Shannon's programme soundly formulated: Suppose that we *impose* N lengths l_0, l_1, \dots, l_{N-1} for the words of a binary code to be constructed. What are the conditions that guarantee the existence of a binary *prefix code* which realizes these lengths? In particular, what about the soundness of the list of lengths derived from a probability distribution, following Shannon's idea? Is this list always realizable by the words of a binary prefix code?

Let us write down most explicitly the Shannon-conditions:

$$\begin{aligned} l_j - 1 < -\log_2 p_j \leq l_j, \quad 0 \leq j \leq N-1, \quad \text{i.e.} \\ 2^{-l_j} \leq p_j < 2 \cdot 2^{-l_j}, \quad 0 \leq j \leq N-1. \end{aligned}$$

Summing over all terms, we get:

$$\sum_{j=0}^{N-1} 2^{-l_j} = \frac{1}{2^{l_0}} + \frac{1}{2^{l_1}} + \dots + \frac{1}{2^{l_{N-1}}} \leq 1.$$

This innocent inequality will finally resolve all our problems.

We begin with a (purely combinatorial) result that has gloriously survived of a dissertation published in 1949:

Proposition (Kraft's Inequality) *Let l_0, l_1, \dots, l_{N-1} be imposed lengths (for N binary code words to construct). Then the following holds:*

There exists a binary prefix code which realizes these lengths $\iff \sum_{j=0}^{N-1} 2^{-l_j} \leq 1$.

Proof Consider the binary tree of all binary words:

$$\begin{array}{ccccccc} & 0 & & & & & 1 \\ & & & & & & \\ 00 & & 01 & & 10 & & 11 \\ & & & & & & \\ 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{array}$$

On level l , there are 2^l binary words of length l , arranged according to their numerical values (every word, considered as the binary notation of an integer, indicates its position). The successors of a word (for the binary tree structure) are precisely its syntactical successors (i.e. the words which admit our word as a prefix).

This will be the convenient framework for the proof of our claim.

\implies : Choose $l > l_j$, $0 \leq j \leq N-1$. Every word of length l_j has 2^{l-l_j} successors on the level l of the binary tree of all binary words. The prefix property implies that these level- l -successor sets are all mutually *disjoint*.

Comparing the cardinality of their union with the number of all words on level l , we get: $\sum_{j=0}^{N-1} 2^{l-l_j} \leq 2^l$, i.e. $\sum_{j=0}^{N-1} 2^{-l_j} \leq 1$.

\Leftarrow : Put $l = \max\{l_j : 0 \leq j \leq N-1\}$, and let n_1, n_2, \dots, n_l be the numbers of code words of length $1, 2, \dots, l$ that we would like to construct. By our hypothesis we have:

$$n_1 \cdot 2^{-1} + n_2 \cdot 2^{-2} + \dots + n_l \cdot 2^{-l} \leq 1,$$

i.e.

$$\begin{array}{ll} n_1 \cdot 2^{-1} \leq 1 & n_1 \leq 2 \\ n_1 \cdot 2^{-1} + n_2 \cdot 2^{-2} \leq 1 & n_2 \leq 2^2 - n_1 \cdot 2 \\ \vdots & \vdots \\ n_1 \cdot 2^{-1} + n_2 \cdot 2^{-2} + \dots + n_l \cdot 2^{-l} \leq 1 & n_l \leq 2^l - n_1 \cdot 2^{l-1} - \dots - n_{l-1} \cdot 2 \end{array}$$

The first inequality shows that we can make our choice on level 1 of the binary tree of all binary words. The second inequality shows that the choice on level 2 is possible, after blockade of the $n_1 \cdot 2$ successors of the choice on level 1. And so on. . .

The last inequality shows that the choice on level l is possible, after blockade of the $n_1 \cdot 2^{l-1}$ successors of the choice on level 1, of the $n_2 \cdot 2^{l-2}$ successors of the choice on level 2, . . . , of the $n_l \cdot 2$ successors of the choice on level $l-1$.

This finishes the proof of our proposition. \square

Exercises

- (1) You would like to construct a binary prefix code with four words of length 3, and six words of length 4. How many words of length 5 can you add?
- (2) Consider an alphabet of four letters: N, E, S, W.
Does there exist a prefix code on this alphabet which consists of two words of length 1, four words of length 2, 10 words of length 3 and 16 words of length 4?
- (3) A memoryless source produces eight letters A, B, C, D, E, F, G, H according to the probability distribution $\mathbf{p} = (p(A), p(B), \dots, p(H))$, with

$$\begin{array}{llll} p(A) = \frac{27}{64}, & p(B) = p(C) = \frac{3}{16}, & p(D) = \frac{1}{16}, \\ p(E) = p(F) = \frac{3}{64}, & p(G) = \frac{1}{32}, & p(H) = \frac{1}{64}. \end{array}$$

- (a) Determine the information content of every letter and compute the entropy $H(\mathbf{p})$ of the source.
- (b) Following Shannon's coding paradigm, find a binary prefix code associated with \mathbf{p} .
- (c) Compute the average length \bar{l} of the code words, and compare it with $H(\mathbf{p})$.

The most important consequence of the characterization of prefix codes via Kraft's inequality is the following theorem. Its small talk version could be: There is no lossless compression below entropy.

Theorem Consider a memoryless source which produces N letters a_0, a_1, \dots, a_{N-1} according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.

Let C be some associated binary prefix code, and

$$\bar{l} = \sum_{j=0}^{N-1} p_j l_j,$$

the average length of the code words (in bits per symbol).

Then: $H(\mathbf{p}) \leq \bar{l}$.

Moreover, the binary prefix codes constructed according to Shannon's idea satisfy the following inequality: $\bar{l} < H(\mathbf{p}) + 1$.

Proof (1) $H(\mathbf{p}) - \bar{l} \leq 0$:

$$H(\mathbf{p}) - \bar{l} = -\sum_{j=0}^{N-1} p_j \log_2 p_j - \sum_{j=0}^{N-1} p_j l_j = \frac{1}{\ln 2} \cdot \sum_{j=0}^{N-1} p_j \ln \left(\frac{2^{-l_j}}{p_j} \right)$$

Now: $\ln x \leq x - 1$ for $x > 0$, hence

$$H(\mathbf{p}) - \bar{l} \leq \frac{1}{\ln 2} \cdot \sum_{j=0}^{N-1} p_j \left(\frac{2^{-l_j}}{p_j} - 1 \right) = \frac{1}{\ln 2} \cdot \sum_{j=0}^{N-1} (2^{-l_j} - p_j).$$

But, due to Kraft's inequality, we have: $\sum_{j=0}^{N-1} (2^{-l_j} - p_j) \leq 0$, and we are done.

(2) *Recall*: following Shannon's idea, one gets for the lengths of the code words associated with our symbols:

$$l_j - 1 < -\log_2 p_j \leq l_j, \quad 0 \leq j \leq N-1.$$

Summing up yields: $\sum_{j=0}^{N-1} (p_j l_j - p_j) < -\sum_{j=0}^{N-1} p_j \log_2 p_j$, i.e.

$$\bar{l} < H(\mathbf{p}) + 1. \quad \square$$

Shannon Codes

Shannon coding is precisely the algorithmic realization of Shannon's coding paradigm:

Encode every source symbol into a binary word – the length of which equals the information content of the source symbol (rounded up to the next integer). We will obtain binary prefix codes. Unfortunately, Shannon codes are not always optimal (in a natural sense, which shall be made precise later) and were soon dethroned by Huffman coding. Why shall we altogether dwell on Shannon coding?

The principal reason is that *arithmetic coding* which is a very interesting “continuous” method of compaction (integrated in certain modes of JPEG

and, more expressly, of JPEG 2000) is nothing but a dynamic version of Shannon coding. With the Shannon codes we are in the antechamber of arithmetic coding.

The idea of Shannon's algorithm is the following:

Consider a memoryless source producing N letters a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.

Assume $p_0 \geq p_1 \geq \dots \geq p_{N-1}$ (in order to guarantee that the following constructions will yield a *prefix* code). Associate with \mathbf{p} a partition of the interval $[0, 1[$ in the following way:

$$\begin{aligned} A_0 &= 0, \\ A_1 &= p_0, \\ A_2 &= p_0 + p_1, \\ A_3 &= p_0 + p_1 + p_2, \\ &\vdots \\ A_N &= p_0 + p_1 + \dots + p_{N-1} = 1. \end{aligned}$$

We note that, the length of every interval $[A_j, A_{j+1}[$ equals the probability of (the production of) the letter a_j :

$$p_j = A_{j+1} - A_j, \quad 0 \leq j \leq N-1.$$

We shall associate with the letter a_j a binary word c_j , which will be *code word of the interval* $[A_j, A_{j+1}[$:

$$c_j = c(A_j, A_{j+1}), \quad 0 \leq j \leq N-1.$$

Let us point out that the realization of Shannon's program demands that the length of the j th code word should be:

$$l_j = \lceil I(a_j) \rceil = \lceil -\log_2 p_j \rceil = \lceil -\log_2 (A_{j+1} - A_j) \rceil, \quad 0 \leq j \leq N-1.$$

These considerations will oblige us to *define* the code word $c(A, B)$ of an interval $[A, B[\subset [0, 1[$ as follows:

$$\begin{aligned} c(A, B) = \alpha_1 \alpha_2 \dots \alpha_l &\iff A = 0 \cdot \alpha_1 \alpha_2 \dots \alpha_l * \text{ (the beginning of the binary} \\ &\text{notation of the real number } A), \text{ with} \\ l &= \lceil -\log_2 (B - A) \rceil. \end{aligned}$$

We insist: the code word $c(A, B)$ of an interval $[A, B[$ is the initial segment of the binary notation of the left boundary A of this interval. One considers as much leading digits as the "information content of the interval" $-\log_2 (B - A)$ demands.

By the way: since the length $B - A$ of an interval $[A, B[\subset [0, 1[$ can actually be considered as the probability “of falling inside” – for certain evident geometric experiences – the value $\text{Log}_2 \frac{1}{B-A}$ has indeed the flavour of an information content.

Exercises

Recall: binary notation of a real number A , $0 \leq A < 1$.

Assume that the development has already been established: $A = 0 \cdot \alpha_1 \alpha_2 \alpha_3 \alpha_4 \dots$. Let us rediscover one by one the digits $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \dots$. Multiply by 2: $2A = \alpha_1 \cdot \alpha_2 \alpha_3 \alpha_4 \dots$ (a comma-shift)

$$\text{If } \begin{cases} 2A \geq 1, \text{ then } \alpha_1 = 1, \\ 2A < 1, \text{ then } \alpha_1 = 0. \end{cases}$$

First case: pass to $A^{(1)} = 2A - 1 = 0 \cdot \alpha_2 \alpha_3 \alpha_4 \dots$,

Second case: pass to $A^{(1)} = 2A = 0 \cdot \alpha_2 \alpha_3 \alpha_4 \dots$,

And so on...

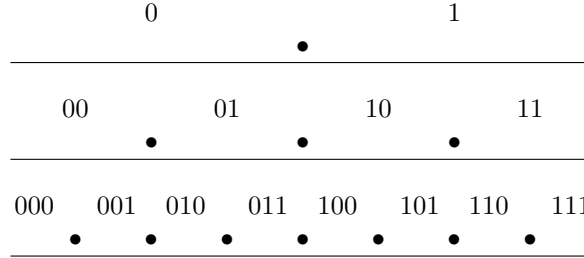
Example Binary notation of $A = \frac{1}{11}$.

$$\begin{aligned} A &= 0 \cdot \alpha_1 \alpha_2 \alpha_3 \alpha_4 \dots, \\ 2A &= \frac{2}{11} < 1 \implies \alpha_1 = 0, \\ 4A &= \frac{4}{11} < 1 \implies \alpha_2 = 0, \\ 8A &= \frac{8}{11} < 1 \implies \alpha_3 = 0, \\ 16A &= \frac{16}{11} = 1 + \frac{5}{11} \implies \alpha_4 = 1, \\ A^{(4)} &= \frac{5}{11} = 0 \cdot \alpha_5 \alpha_6 \alpha_7 \dots, \\ 2A^{(4)} &= \frac{10}{11} < 1 \implies \alpha_5 = 0, \\ 4A &= \frac{20}{11} = 1 + \frac{9}{11} \implies \alpha_6 = 1, \\ A^{(6)} &= \frac{9}{11} = 0 \cdot \alpha_7 \alpha_8 \alpha_9 \dots, \\ 2A^{(6)} &= \frac{18}{11} = 1 + \frac{7}{11} \implies \alpha_7 = 1, \\ A^{(7)} &= \frac{7}{11} = 0 \cdot \alpha_8 \alpha_9 \alpha_{10} \dots, \\ 2A^{(7)} &= \frac{14}{11} = 1 + \frac{3}{11} \implies \alpha_8 = 1, \end{aligned}$$

$$\begin{aligned}
A^{(8)} &= \frac{3}{11} = 0 \cdot \alpha_9 \alpha_{10} \alpha_{11} \cdots, \\
2A^{(8)} &= \frac{6}{11} < 1 \implies \alpha_9 = 0, \\
4A^{(8)} &= \frac{12}{11} = 1 + \frac{1}{11} \implies \alpha_{10} = 1, \\
A^{(10)} &= \frac{1}{11} = 0 \cdot \alpha_{11} \alpha_{12} \alpha_{13} \cdots = A.
\end{aligned}$$

The binary development of $A = \frac{1}{11} = 0 \cdot \overline{0001011101}$ (period length 10).

- (1) Consider the subdivision of the interval $[0, 1[$ by iterated dichotomy:



Let us encode these *standard-intervals* by the *paths* which point at them:

$$\begin{aligned}
010 = \leftarrow \rightarrow \leftarrow & \quad \text{points at} \quad \left[\frac{1}{4}, \frac{3}{8} \right], \\
110 = \rightarrow \rightarrow \leftarrow & \quad \text{points at} \quad \left[\frac{3}{4}, \frac{7}{8} \right].
\end{aligned}$$

Show that the arithmetic code word $c(A, B)$ of a standard-interval $[A, B[$ equals the binary word that points at this interval (in the tree of dichotomy above).

Solution Let us show that the path $\alpha_1 \alpha_2 \cdots \alpha_l$ which points at the interval $[A, B[$ located on level l of our tree of dichotomy is equal to the l first bits (after the comma) of the binary notation of A .

Recursion on l :

$l = 1$: 0 and 1 are, respectively, the first bit of the binary notation of $0 = 0.0000 \dots$ and of $\frac{1}{2} = 0.1000 \dots$

$l \mapsto l + 1$: consider $[A, B[$ on level $l + 1$ of our tree of dichotomy. $[A, B[$ is either the first half or the second half of an interval $[A^*, B^*[$ on level l .

By recursion hypothesis: $A^* = 0 \cdot \alpha_1 \alpha_2 \cdots \alpha_l$, and $\alpha_1 \alpha_2 \cdots \alpha_l$ points at $[A^*, B^*[$. If $[A, B[$ is to the left, then $\alpha_1 \alpha_2 \cdots \alpha_l 0$ points at $[A, B[$ and we have $A = A^* = 0 \cdot \alpha_1 \alpha_2 \cdots \alpha_l 0$.

If $[A, B[$ is to the right, then $\alpha_1 \alpha_2 \cdots \alpha_l 1$ points at $[A, B[$ and we have $A = A^* + \frac{1}{2}(B^* - A^*) = A^* + \frac{1}{2^{l+1}}$, i.e. the binary notation of A is $A = 0 \cdot \alpha_1 \alpha_2 \cdots \alpha_l 1$, as claimed.

- (2) Find the following code words $c(A, B)$
- (a) $c\left(\frac{3}{8}, \frac{1}{2}\right)$
 - (b) $c\left(\frac{1}{12}, \frac{7}{8}\right)$
 - (c) $c\left(\frac{3}{5}, \frac{3}{4}\right)$
 - (d) $c\left(\frac{1}{7}, \frac{1}{6}\right)$
- (3) Determine all intervals $[A, B[$ such that $c(A, B) = 10101$.
(Find first the standard-interval described by 10101, then think about the extent you can “deform” it without changing the code word.)
- (4) Consider a memoryless source which produces N letters a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.
Assume $p_0 \geq p_1 \geq \dots \geq p_{N-1}$.
Show that the associated Shannon code is a *prefix code*.

Solution First, we have necessarily: $l_0 \leq l_1 \leq \dots \leq l_{N-1}$ (where $l_j = \lceil I(a_j) \rceil = \lceil -\log_2 p_j \rceil$ is the length of the j th code word $0 \leq j \leq N-1$). Let us show that the code word $c_j = c(A_j, A_{j+1})$ *cannot* be a prefix of the word $c_{j+1} = c(A_{j+1}, A_{j+2})$ for $0 \leq j \leq N-2$. Otherwise we would have:

$$\begin{aligned} A_j &= 0 \cdot \alpha_1 \alpha_2 \dots \alpha_{l_j} *, \\ A_{j+1} &= 0 \cdot \alpha_1 \alpha_2 \dots \alpha_{l_j} *, \end{aligned}$$

hence $p_j = A_{j+1} - A_j = 0 \cdot 0 \dots 0 *$ (with a block of at least l_j zeros after the comma) $\implies p_j < 2^{-l_j} \implies l_j < I(a_j)$, a contradiction.

Finally, if the code word c_j is a prefix of the code word c_k , $j < k$, then c_j must be necessarily a prefix of the word c_{j+1} (why?), and we can conclude.

- (5) A memoryless source produces four letters A, B, C, D , with

$$p(A) = \frac{1}{2}, \quad p(B) = \frac{1}{4}, \quad p(C) = p(D) = \frac{1}{8}.$$

Write down the Shannon code word of *BADACABA*.

- (6) Consider the source which produces the eight letters a_0, a_1, \dots, a_7 , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_7)$ where $p_0 = \frac{1}{2}, p_1 = \frac{1}{4}, p_2 = p_3 = \frac{1}{16}, p_4 = p_5 = p_6 = p_7 = \frac{1}{32}$.
Find the associated Shannon code.
- (7) Our memoryless source produces the eight letters A, B, C, D, E, F, G, H according to the probability distribution $\mathbf{p} = (p(A), p(B), \dots, p(H))$ with

$$\begin{aligned} p(A) &= \frac{27}{64} & p(B) &= p(C) = \frac{3}{16} & p(D) &= \frac{1}{16}, \\ p(E) &= p(F) = \frac{3}{64} & p(G) &= \frac{1}{32} & p(H) &= \frac{1}{64}. \end{aligned}$$

- (a) Find the associated Shannon code.
- (b) Compute the average word length \bar{l} of the code words.

The Huffman Algorithm

Four years after Shannon's seminal papers, the Huffman algorithm appears, with universal acclaim. Being of utmost mathematical simplicity, it yields nevertheless the best – and thus definitive – algorithmic solution of the prefix coding problem for memoryless discrete sources.

Example Recall our source which produces the eight letters a_0, a_1, \dots, a_7 , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_7)$ with $p_0 = \frac{1}{2}, p_1 = \frac{1}{4}, p_2 = p_3 = \frac{1}{16}, p_4 = p_5 = p_6 = p_7 = \frac{1}{32}$.

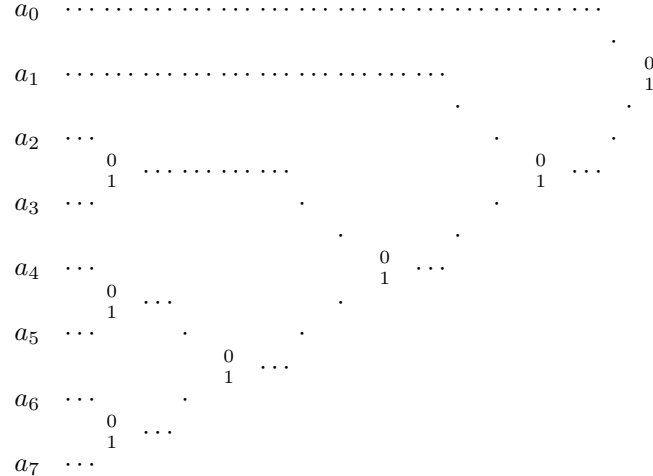
We did encode as follows:

$$\begin{array}{ll} a_0 \mapsto 0 & a_4 \mapsto 11100 \\ a_1 \mapsto 10 & a_5 \mapsto 11101 \\ a_2 \mapsto 1100 & a_6 \mapsto 11110 \\ a_3 \mapsto 1101 & a_7 \mapsto 11111 \end{array}$$

With the standard-interval coding of the preceding section in mind, where the code words are *paths* in a binary tree, one could come up with the following idea.

Let us interpret the code words above as paths in a binary tree which admits the symbols a_0, a_1, \dots, a_7 as leaves (i.e. as terminal nodes).

We will obtain the following structure:



How can we generate, in general, this binary tree in function of the source symbols, more precisely: in function of the probability distribution \mathbf{p} which describes the production of the source?

Let us adopt the following viewpoint.

We shall consider the given symbols as the leaves (as the terminal nodes) of a binary tree *which has to be constructed*. The *code words* associated with the symbols will be the *paths* towards the leaves (the symbols).

We note: the lesser the probability of a letter is, the longer should be the path towards this letter. The algorithm will have to create nodes (antecedents), conducted primarily by the *rare* letters; thus we shall need a numerical control by a *weighting* of the nodes.

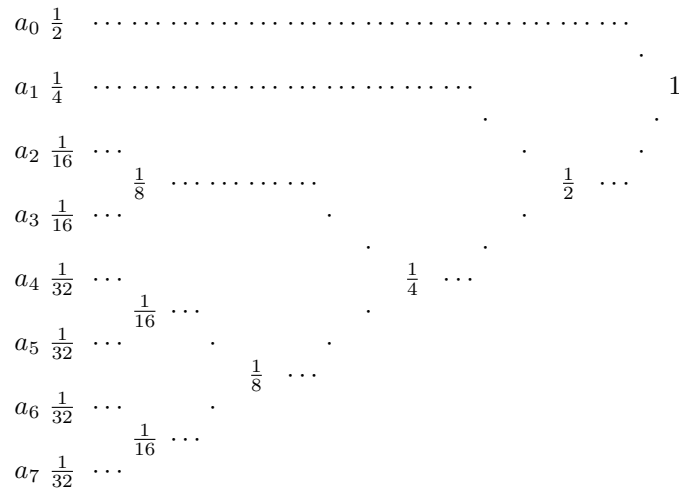
The most primitive algorithm that we can invent – based on these design-patterns – will actually be the best one:

<i>Algorithm of Huffman</i> for the construction of a <i>weighted</i> binary tree:	
Every step will create a new node, antecedent for two nodes taken from a list of candidates.	
<i>Start:</i>	Every source symbol is a weighted node (a candidate), the weight of which is its probability.
<i>Step:</i>	The two nodes of minimal weight (in the actual list of candidates) create an antecedent whose weight will be the sum of the weights of its successors; it replaces them in the list of the candidates.
<i>End:</i>	There remains a single node (of weight equal to 1) in the list of the candidates.

This is a recursive algorithm. Note that with every step the number of (couples of) nodes searching an antecedent becomes smaller and smaller. On the other hand, the sum over all weights remains always constant, i.e. equal to 1.

Attention: “the two nodes of minimal weight” are, in general, *not unique*. You frequently have to make a choice. So, the result of Huffman’s algorithm is, in general, far from unique.

Back to our example:



First, it is a_6 and a_7 which find an antecedent with weight $\frac{1}{16}$ (we make a choice at the end of the list). At the next step, we have no choice: it is a_4 and a_5 which have minimal weight and will thus find their antecedent of weight $\frac{1}{16}$. Now, we have six nodes as candidates, four of which have weight $\frac{1}{16}$. Once more, we shall choose at the end of the list, and we find this way a common antecedent for a_4, a_5, a_6 and a_7 , the weight of which is $\frac{1}{8}$. And so on...

Exercises

- (1) Our memoryless source producing the eight letters A, B, C, D, E, F, G, H according to the probability distribution $\mathbf{p} = (p(A), p(B), \dots, p(H))$ with

$$\begin{aligned} p(A) &= \frac{27}{64}, & p(B) &= p(C) = \frac{3}{16}, & p(D) &= \frac{1}{16}, \\ p(E) &= p(F) = \frac{3}{64}, & p(G) &= \frac{1}{32}, & p(H) &= \frac{1}{64}. \end{aligned}$$

- (a) Find the associated Huffman code.
 (b) Compute the average word length \bar{l} of the code words.
 (2) Consider a source which produces the 12 letters a_0, a_1, \dots, a_{11} according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{11})$ with

$$\begin{aligned} p_0 &= \frac{3}{16}, & p_1 &= \frac{5}{32}, & p_2 &= \frac{1}{8}, \\ p_3 &= p_4 = \frac{3}{32}, & p_5 &= p_6 = p_7 = p_8 = \frac{1}{16}, & p_9 &= p_{10} = p_{11} = \frac{1}{32}. \end{aligned}$$

- (a) Compute $H(\mathbf{p})$.
 (b) Find the associated Huffman code, and compare \bar{l} , the average word length of the code words with $H(\mathbf{p})$.
 (3) A memoryless source producing the three letters A, B, C with the probabilities $p(A) = \frac{3}{4}$, $p(B) = \frac{3}{16}$ and $p(C) = \frac{1}{16}$.
 (a) Compute the entropy of the source, find the associated Huffman code and the average word length for the (three) code words.
 (b) Consider now the same source, but as producer of the nine symbols $AA, AB, AC, BA, BB, BC, CA, CB, CC$, according to the product distribution (i.e. $p(AB) = p(BA) = \frac{9}{64}$).

Generate the associated Huffman code, and compute the average word length of the code words *per initial symbol*. Compare with (a).

Remark The *compressed bit-rate* ρ of an encoder is defined as follows:

$$\rho = \frac{\text{average length of the code words}}{\text{average length of the source symbols}}.$$

It is clear that this definition makes sense only when complemented by an evaluation of the stream of source symbols: what is the average length of the source symbols – in *bits* per symbol?

- (4) A binary source, which we consider as memoryless on words of length 4. We shall adopt the hexadecimal notation (example: $d = 1101$). We observe the following probability distribution:
 $p(0) = 0.40, p(4) = 0.01, p(8) = 0.01, p(c) = 0.05,$
 $p(1) = 0.01, p(5) = 0.03, p(9) = 0.04, p(d) = 0.01,$
 $p(2) = 0.01, p(6) = 0.04, p(a) = 0.03, p(e) = 0.01,$
 $p(3) = 0.05, p(7) = 0.01, p(b) = 0.01, p(f) = 0.28.$
 Generate the associated Huffman code, and compute the compressed bit-rate.
- (5) A facsimile system for transmitting line-scanned documents uses black runlengths and white runlengths as the source symbols. We observe the following probability distribution:
 $p(B1) = 0.05, p(B5) = 0.01, p(W1) = 0.02, p(W5) = 0.01,$
 $p(B2) = 0.02, p(B6) = 0.02, p(W2) = 0.02, p(W6) = 0.01,$
 $p(B3) = 0.01, p(B7) = 0.01, p(W3) = 0.01, p(W7) = 0.01,$
 $p(B4) = 0.10, p(B8) = 0.25, p(W4) = 0.05, p(W8) = 0.40.$
As to the notation: $B3 \equiv 000, W5 \equiv 1111$.
 (a) Find the Huffman code for this system.
 (b) Compute the compressed bit-rate.
- (6) A Huffman code associated with an alphabet of eight letters; we have the following eight code words (where two of them are masked):
 00, 10, 010, 1100, 1101, 1111, \mathbf{w}_1 , \mathbf{w}_2 . Find \mathbf{w}_1 and \mathbf{w}_2 .
- (7) An alphabet of eight letters $a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7$.
 A Huffman encoder has associated the following eight code words:
 $c_0 = 00, c_1 = 01, c_2 = 100, c_3 = 101, c_4 = 1100, c_5 = 1101, c_6 = 1110,$
 $c_7 = 1111.$
 Find a probability distribution $\mathbf{p} = (p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7)$ such that the Shannon encoder yields the same list of code words.
- (8) A binary source which is memoryless on the eight binary triples.
 A Huffman encoder associates the eight code words 00, 01, 100, 101, 1100, 1101, 1110, 1111.
 (a) Find a probability distribution which fits with this code.
 (b) Is it possible to choose a probability distribution which gives rise to a compressed bit-rate of 70%?
- (9) Let a_0 be the symbol of highest probability p_0 of an alphabet which has N symbols ($N \geq 3$). A Huffman encoder associates a binary code word of length l_0 . Show the following assertions:
 (a) If $p_0 > \frac{4}{5}$, then $l_0 = 1$.
 (b) If $p_0 < \frac{1}{3}$, then $l_0 \geq 2$.
- (10) The optimal questionnaire.
 You would like to participate in a TV game: you will have to find the profession of a person ("chosen at random") by three yes or no questions. You look at the statistics: there are 16 main professions $P1, P2, \dots, P16$, occurring with the following frequencies:

$p(P1) = 0.40$, $p(P5) = 0.05$, $p(P9) = 0.02$, $p(P13) = 0.01$,
 $p(P2) = 0.18$, $p(P6) = 0.04$, $p(P10) = 0.02$, $p(P14) = 0.01$,
 $p(P3) = 0.10$, $p(P7) = 0.03$, $p(P11) = 0.02$, $p(P15) = 0.01$,
 $p(P4) = 0.06$, $p(P8) = 0.03$, $p(P12) = 0.01$, $p(P16) = 0.01$.

- (a) Find the strategy for the optimal questionnaire.
 (b) Will you have a good chance (with only three questions)?

Huffman Coding in JPEG

Situation *JPEG treats a digital image as a sequence of blocks of 8×8 pixels. More precisely, a data unit will be a triple of 8×8 matrices. The first one for the pixel-values of luminance (Y), the two others for the pixel-values of chrominance (Cb , Cr).*

A linear invertible transformation (the 2D Discrete Cosine Transform) will transform each of these three matrices in a 8×8 matrix of the following type:

Significant values	Less
Significant	Values

In lossy compression mode, an appropriate quantization procedure will finally set to zero most of the less significant values.

We ultimately come up with quantized schemes (of 64 *integers*) of the following type:

Significant quantized values	Frequently
Zero quantized	Values

The value of the DC coefficient (*direct current*) in the left upper corner of the matrix will not be interesting – at least in the present context.

The Huffman coding deals with the 63 AC coefficients (*alternating current*), the absolute values of which are – in general – sensibly smaller than (the absolute value of) the dominant DC coefficient.

We shall make use of a sequential zigzag reading according to the scheme on the top of the next page.

The encoding concerns *the sequence of the non-zero coefficients* in the zigzag reading of the quantized scheme. It is clear that we also have to take into account the zero runlengths between the non-zero coefficients.

DC	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Zigzag ordering of the quantized coefficients.

In order to prepare Huffman coding conveniently, we begin with a hierarchy of 10 categories for the non-zero coefficients:

1	-1	1
2	-3, -2	2, 3
3	-7, -6, -5, -4	4, 5, 6, 7
4	-15, -14, ..., -9, -8	8, 9, ..., 14, 15
5	-31, -30, ..., -17, -16	16, 17, ..., 30, 31
6	-63, -62, ..., -33, -32	32, 33, ..., 62, 63
7	-127, -126, ..., -65, -64	64, 65, ..., 126, 127
8	-255, -254, ..., -129, -128	128, 129, ..., 254, 255
9	-511, 510, ..., -257, -256	256, 257, ..., 510, 511
10	-1,023, -1,022, ..., -513, -512	512, 513, ..., 1,022, 1,023

Attention *There is a tacit convention concerning the encoding of all these integers; in category 4 for example, the code words will have four bits:*

$-15 \mapsto 0000$, $-14 \mapsto 0001$, ..., $-9 \mapsto 0110$, $-8 \mapsto 0111$, $8 \mapsto 1000$, $9 \mapsto 1001$, ..., $14 \mapsto 1110$, $15 \mapsto 1111$.

We observe that a non-zero coefficient occurring in the sequential reading of a quantized scheme can be characterized by three parameters:

- (1) The number of zeros which separate it from its non-zero predecessor.
- (2) Its category.
- (3) Its number within the category.

Example Consider the sequence 0800−2040001...

This means for

	Runlength/category	Value within the cat.
8	1/4	1000
−2	2/2	01
4	1/3	100
1	3/1	1

In order to be able to encode the sequential reading of the quantized coefficients, we need only a coding table for the symbols of the type run-length/category.

We shall give the table for the *luminance* AC coefficients.

The table has been developed by JPEG (*Joint Photographic Experts Group*) from the average statistics of a large set of images with 8 bit precision. It was not meant to be a default table, but actually it is.

Remark On two particular symbols.

- (1) (EOB) \equiv *end of block* indicates the end of the non-zero coefficients in the sequence of the 63 AC coefficients to be encoded. The code word for this happy event will be 1010.
- (2) (ZRL) \equiv *zero run list* indicates the outcome of the integer 0 preceded by a block of 15 zeros.²

² Attention, our zeros are zeros as integers – and not as bits!

Runlength/cat.	Code word	Runlength/cat.	Code word
0/0 (EOB)	1010	3/9	111111110010100
0/1	00	3/ <i>a</i>	111111110010101
0/2	01	4/1	111011
0/3	100	4/2	1111111000
0/4	1011	4/3	111111110010110
0/5	11010	4/4	111111110010111
0/6	1111000	4/5	111111110011000
0/7	11111000	4/6	111111110011001
0/8	1111110110	4/7	111111110011010
0/9	111111110000010	4/8	111111110011011
0/ <i>a</i>	111111110000011	4/9	111111110011100
1/1	1100	4/ <i>a</i>	111111110011101
1/2	11011	5/1	1111010
1/3	1111001	5/2	11111110111
1/4	111110110	5/3	111111110011110
1/5	11111110110	5/4	111111110011111
1/6	111111110000100	5/5	111111110100000
1/7	111111110000101	5/6	111111110100001
1/8	111111110000110	5/7	111111110100010
1/9	111111110000111	5/8	111111110100011
1/ <i>a</i>	111111110001000	5/9	111111110100100
2/1	11100	5/ <i>a</i>	111111110100101
2/2	11111001	6/1	1111011
2/3	1111110111	6/2	111111110110
2/4	111111110100	6/3	111111110100110
2/5	111111110001001	6/4	111111110100111
2/6	111111110001010	6/5	111111110101000
2/7	111111110001011	6/6	111111110101001
2/8	111111110001100	6/7	111111110101010
2/9	111111110001101	6/8	111111110101011
2/ <i>a</i>	111111110001110	6/9	111111110101100
3/1	111010	6/ <i>a</i>	111111110101101
3/2	111110111	7/1	11111010
3/3	111111110101	7/2	111111110111
3/4	111111110001111	7/3	111111110101110
3/5	111111110010000	7/4	111111110101111
3/6	111111110010001	7/5	111111110110000
3/7	111111110010010	7/6	111111110110001
3/8	111111110010011	7/7	111111110110010

Runlength/cat.	Code word	Runlength/cat.	Code word
7/8	111111110110011	<i>b</i> /7	111111111010101
7/9	111111110110100	<i>b</i> /8	111111111010110
7/ <i>a</i>	111111110110101	<i>b</i> /9	111111111010111
8/1	111111000	<i>b</i> / <i>a</i>	111111111011000
8/2	11111111000000	<i>c</i> /1	1111111010
8/3	111111110110110	<i>c</i> /2	111111111011001
8/4	111111110110111	<i>c</i> /3	111111111011010
8/5	111111110111000	<i>c</i> /4	111111111011011
8/6	111111110111001	<i>c</i> /5	111111111011100
8/7	111111110111010	<i>c</i> /6	111111111011101
8/8	111111110111011	<i>c</i> /7	111111111011110
8/9	111111110111100	<i>c</i> /8	111111111011111
8/ <i>a</i>	111111110111101	<i>c</i> /9	111111111100000
9/1	111111001	<i>c</i> / <i>a</i>	111111111100001
9/2	111111110111110	<i>d</i> /1	11111111000
9/3	111111110111111	<i>d</i> /2	111111111100010
9/4	111111111000000	<i>d</i> /3	111111111100011
9/5	111111111000001	<i>d</i> /4	111111111100100
9/6	111111111000010	<i>d</i> /5	111111111100101
9/7	111111111000011	<i>d</i> /6	111111111100110
9/8	111111111000100	<i>d</i> /7	111111111100111
9/9	111111111000101	<i>d</i> /8	111111111101000
9/ <i>a</i>	111111111000110	<i>d</i> /9	111111111101001
<i>a</i> /1	111111010	<i>d</i> / <i>a</i>	111111111101010
<i>a</i> /2	111111111000111	<i>e</i> /1	111111111101011
<i>a</i> /3	111111111001000	<i>e</i> /2	111111111101100
<i>a</i> /4	111111111001001	<i>e</i> /3	111111111101101
<i>a</i> /5	111111111001010	<i>e</i> /4	111111111101110
<i>a</i> /6	111111111001011	<i>e</i> /5	111111111101111
<i>a</i> /7	111111111001100	<i>e</i> /6	111111111110000
<i>a</i> /8	111111111001101	<i>e</i> /7	111111111110001
<i>a</i> /9	111111111001110	<i>e</i> /8	111111111110010
<i>a</i> / <i>a</i>	111111111001111	<i>e</i> /9	111111111110011
<i>b</i> /1	1111111001	<i>e</i> / <i>a</i>	111111111110100
<i>b</i> /2	111111111010000	<i>f</i> /0 (ZRL)	11111111001
<i>b</i> /3	111111111010001	<i>f</i> /1	111111111110101
<i>b</i> /4	111111111010010	<i>f</i> /2	111111111110110
<i>b</i> /5	111111111010011	<i>f</i> /3	111111111110111
<i>b</i> /6	111111111010100	<i>f</i> /4	111111111111000
<i>f</i> /5	111111111111001	<i>f</i> /8	111111111111100
<i>f</i> /6	111111111111010	<i>f</i> /9	111111111111101
<i>f</i> /7	111111111111011	<i>f</i> / <i>a</i>	111111111111110

then the quantized scheme³

$$\begin{array}{cccccccc} 120 & 120 & 120 & 120 & 120 & 120 & 120 & 120 \\ 150 & 150 & 150 & 150 & 150 & 150 & 150 & 150 \end{array} \xrightarrow{\quad} \begin{array}{cccccccc} 57.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \xrightarrow{\quad}$$

68 is the round of $\frac{1,080}{16}$

-46 is the round of $\frac{-546.6}{12}$

-4 is the round of $\frac{-57.1}{14}$

-1 is the round of $\frac{-17}{24}$

0 is the round of $\frac{-4.3}{72}$

$$\begin{array}{cccccccccccccccc}0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\0 & 0 & 0 & 0 & 0 & 0 & .\end{array}$$

⁴ *Attention:* we have got a prefix code – the separating blanks in our notation are redundant and exist only for the convenience of the reader. . .

$$\begin{array}{cccccccc}
68 \cdot 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-46 \cdot 12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-4 \cdot 14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 \cdot 24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0.72 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
=
\begin{array}{cccccccc}
1,088 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-552 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-56 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

This is the moment where the loss of information, due to quantization, becomes apparent. The decompression will be finished after transforming the matrix of dequantized values back, and making the necessary rounds (do not forget: we did start with 8-bit bytes). We obtain:

30	30	30	30	30	30	30	30
61	61	61	61	61	61	61	61
91	91	91	91	91	91	91	91
119	119	119	119	119	119	119	119
153	153	153	153	153	153	153	153
181	181	181	181	181	181	181	181
211	211	211	211	211	211	211	211
242	242	242	242	242	242	242	242

Exercises

- (1) The Huffman coding described above has a two-level structure: there are code words for the type of the integer to be encoded, and there are the code words for its numerical value. Writing down the total Huffman tree for this coding, what will be the number of leaves (of code words)?
- (2) We shall treat five 8×8 matrices of luminance values, together with their quantized 2D DCT matrices. Establish in every case the associated Huffman coding, and compute the compressed bit-rate.

$$\begin{array}{l}
\begin{array}{cccccccc}
159 & 152 & 142 & 134 & 133 & 140 & 149 & 155 \\
176 & 170 & 162 & 156 & 157 & 163 & 171 & 177 \\
132 & 129 & 123 & 120 & 121 & 126 & 132 & 136 \\
72 & 71 & 69 & 68 & 69 & 70 & 72 & 74 \\
69 & 70 & 72 & 73 & 73 & 71 & 69 & 67 \\
123 & 126 & 131 & 134 & 133 & 129 & 123 & 119 \\
157 & 163 & 171 & 177 & 176 & 170 & 162 & 156 \\
132 & 139 & 149 & 157 & 158 & 151 & 142 & 135
\end{array}
\mapsto
\begin{array}{cccccccc}
64 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\
17 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \\
\\
\begin{array}{l}
\begin{array}{cccccccc}
83 & 89 & 91 & 84 & 73 & 68 & 75 & 83 \\
96 & 98 & 96 & 86 & 78 & 82 & 99 & 114 \\
82 & 85 & 84 & 75 & 67 & 69 & 83 & 97 \\
88 & 99 & 108 & 107 & 99 & 94 & 97 & 104 \\
88 & 101 & 115 & 118 & 112 & 108 & 111 & 116 \\
95 & 103 & 110 & 107 & 99 & 97 & 105 & 114 \\
122 & 130 & 136 & 131 & 120 & 114 & 117 & 124 \\
96 & 111 & 127 & 131 & 121 & 109 & 105 & 105
\end{array}
\mapsto
\begin{array}{cccccccc}
50 & 0 & 0 & -3 & 0 & 0 & 0 & 0 \\
-9 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\end{array}
\end{array}$$

(c)	25	32	41	46	47	48	51	53		41	-5	1	-3	0	0	0	0
	45	57	75	90	94	89	80	73		-15	2	-4	1	0	0	0	0
	26	34	43	48	46	41	37	34		0	-3	0	1	0	0	0	0
	82	89	92	84	72	69	78	90	\mapsto	1	0	0	-1	0	0	0	0
	87	101	109	98	80	79	102	126		0	0	1	0	0	0	0	0
	61	78	91	84	67	65	87	110		0	0	0	0	0	0	0	0
	112	122	131	129	123	126	141	156		-3	0	0	0	0	0	0	0
	94	88	80	77	85	106	131	149		0	0	0	0	0	0	0	0
(d)	133	139	89	198	114	91	151	114		63	0	0	0	1	0	-1	0
	126	127	95	207	136	110	148	124		0	1	0	-1	0	1	0	-1
	134	128	186	73	32	158	93	127		2	0	-2	0	1	0	-1	0
	104	128	75	40	246	129	143	141	\mapsto	0	1	0	-1	0	1	0	-1
	141	143	129	246	40	75	128	104		0	0	-1	0	1	0	-1	0
	127	93	158	32	73	186	128	134		0	-1	0	1	0	-1	0	1
	124	148	110	136	207	95	127	126		-1	0	1	0	-1	0	1	0
	114	151	91	114	198	89	139	133		0	1	0	-1	0	1	0	-1
(e)	65	61	68	59	69	60	67	63		32	0	0	0	0	0	0	0
	61	72	52	78	50	76	56	67		0	0	0	0	0	0	0	0
	68	52	81	44	84	47	76	60		0	0	0	0	0	0	0	0
	59	78	44	88	40	84	50	69	\mapsto	0	0	0	0	0	0	0	0
	69	50	84	40	88	44	78	59		0	0	0	0	0	0	0	0
	60	76	47	84	44	81	52	68		0	0	0	0	0	0	0	0
	67	56	76	50	78	52	72	61		0	0	0	0	0	0	0	0
	63	67	60	69	59	68	61	65		0	0	0	0	0	0	0	1

We observe a stubborn propagation of non-zero values in the quantized scheme (d). This comes from a (pictorially) perverse distribution of the luminance values in the initial scheme. Note that in general the zigzag ordering has the property that the probability of coefficients being zero is an approximately monotonic increasing function of the index.

Huffman Coding is Optimal

Situation A memoryless source, producing N letters a_0, a_1, \dots, a_{N-1} according to the fixed probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.

Now consider *all* associated binary prefix codes.

Such a binary prefix code C is *optimal* : \Leftrightarrow

The average word length $\bar{l} = p_0 l_0 + p_1 l_1 + \dots + p_{N-1} l_{N-1}$ of its words is *minimal*. (Note that $\bar{l} = \bar{l}(l_0, l_1, \dots, l_{N-1})$ is a function of the lengths of the N code words associated with a_0, a_1, \dots, a_{N-1} ; the probabilities p_0, p_1, \dots, p_{N-1} are the *constants* of the problem.)

Our goal: we shall show that the *Huffman algorithm necessarily produces an optimal binary prefix code*.

But first several characteristic properties of optimal codes:

Proposition *Let C be an optimal binary prefix code, associated with $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.*

Then we necessarily have:

1. $p_j > p_k \implies l_j \leq l_k$.
2. The code will have an even number of words of maximal length.
3. Whenever several code words have the same length, two of them will be equal except for the last bit.

Proof 1. Assume $p_j > p_k$ and $l_j > l_k$; then

$$(p_j - p_k)(l_j - l_k) > 0,$$

i.e.

$$p_j l_j + p_k l_k > p_j l_k + p_k l_j.$$

This shows: if we exchange the code words of a_j and of a_k , then we get a better code.

2. Otherwise there would exist a code word of maximal length without a partner which differs only in the last bit. We can suppress its last bit, and will obtain a word which is not a code word (prefix property!). This permits us to change to a better code.
3. Consider all code words of length l (arbitrary, but fixed). Assume that all these words remain distinct when skipping everywhere the last bit. Due to the prefix property, we would thus get a better code. \square

The Huffman codes are optimal: this is an immediate consequence of the following proposition.

Proposition *Consider a source \mathbb{S} of N states, controlled by the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.*

Replace the two symbols a_{j_1} and a_{j_2} of smallest probabilities by a single symbol $a_{(j_1, j_2)}$ with probability $p_{(j_1, j_2)} = p_{j_1} + p_{j_2}$. Let \mathbb{S}' be the source of $N - 1$ states we get this way.

Let C' be an optimal binary prefix code for \mathbb{S}' , and let \mathbf{x} be the code word of $a_{(j_1, j_2)}$.

Let C be the ensuing binary prefix code for \mathbb{S} :

$$\begin{aligned} a_{j_1} &\longmapsto \mathbf{x}0, \\ a_{j_2} &\longmapsto \mathbf{x}1. \end{aligned}$$

Then C is optimal for \mathbb{S} .

Proof The lengths of the code words (L for C' , l for C):

$$l_j = \begin{cases} L_{(j_1, j_2)} + 1, & \text{if } j = j_1, j_2, \\ L_j, & \text{else.} \end{cases}$$

One gets for the *average lengths* (\bar{L} for C' , \bar{l} for C):

$$\begin{aligned} \bar{l} &= \sum_{j \neq j_1, j_2} p_j l_j + p_{j_1} l_{j_1} + p_{j_2} l_{j_2} = \sum_{j \neq j_1, j_2} p_j L_j + p_{(j_1, j_2)} L_{(j_1, j_2)} + p_{j_1} + p_{j_2} \\ &= \bar{L} + p_{j_1} + p_{j_2}. \end{aligned}$$

But $p_{j_1} + p_{j_2}$ is a *constant* for our optimality arguments (we make the word lengths vary), and it is the constant of the smallest possible difference between \bar{l} and \bar{L} (due to the choice of p_{j_1} and p_{j_2} !). Thus, \bar{L} minimal $\implies \bar{l}$ minimal. \square

Corollary *The Huffman algorithm produces optimal binary prefix codes.*

In particular, the average word length \bar{l} of the code words is constant for all Huffman codes associated with a fixed probability distribution \mathbf{p} (note that you will be frequently obliged to make choices when constructing Huffman trees).

Observation *For every Huffman code associated with a fixed probability distribution \mathbf{p} we have the estimation $H(\mathbf{p}) \leq \bar{l} < H(\mathbf{p}) + 1$ (this is true for Shannon coding, hence a fortiori for Huffman coding).*

Exercises

- (1) Does there exist a binary prefix code which consists of three words of length 3, of five words of length 4 and of nine words of length 5?
Does there exist a Huffman code which consists of three words of length 3, of five words of length 4 and of nine words of length 5?
- (2) Show: every Huffman code is a Shannon code.
More precisely: let C be a Huffman code (given by its binary tree); then there exists a probability distribution \mathbf{p} such that the set of code words of C is the associated Shannon code.
- (3) Let $\{A, B, C, D\}$ be an alphabet of four letters, with $p(A) \geq p(B) \geq p(C) \geq p(D)$.
(a) Find all associated Huffman codes.
(b) Give an example of a Shannon code (in choosing appropriate probabilities) which is not a Huffman code.
- (4) Is an *optimal* binary prefix code necessarily a Huffman code?

Approximation of the Entropy via Block Encoding

Consider a memoryless source, producing the N letters a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.

Let us change our outlook:

We shall take the *words of length n* ($n \geq 1$) as our new production units:

$\mathbf{x} = a_{j_1} a_{j_2} \cdots a_{j_n}$ will be the notation for these “big letters”.

The product probability distribution:

$$\mathbf{p}^{(n)} = \left(\prod p_{j_1} p_{j_2} \cdots p_{j_n} \right)_{0 \leq j_1, j_2, \dots, j_n \leq N-1}$$

- (there are N^n words of length n on an alphabet of N elements).

Recall

$$H(\mathbf{p}^{(n)}) = n \cdot H(\mathbf{p})$$

(this was a previous exercise).

Proposition *A memoryless source, producing the N letters a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.*

Let us pass to an encoding of blocks in n letters.

Let C be an associated Huffman code, and let \bar{l}_i be the average word length of the code words per initial symbol.

Then we have:

$$H(\mathbf{p}) \leq \bar{l}_i < H(\mathbf{p}) + \frac{1}{n}.$$

Proof Let $\bar{l}_n = \sum p(\mathbf{x})l(\mathbf{x})$ be the average length of the code words of C . We have the following estimation:

$$H(\mathbf{p}^{(n)}) \leq \bar{l}_n < H(\mathbf{p}^{(n)}) + 1.$$

But $H(\mathbf{p}^{(n)}) = n \cdot H(\mathbf{p})$, $\bar{l}_n = n \cdot \bar{l}_i$; whence the final result. \square

Exercises

- (1) Consider a binary source which produces one bit per unit of time (for example, every μs). We know: $p_0 = \frac{3}{4}$, $p_1 = \frac{1}{4}$. Suppose that our bitstream has to pass through a channel which accepts only 0.82 bits per unit of time. Construct an adapter by means of Huffman block encoding.
(N.B.: $H(\mathbf{p}) = 0.81$).
- (2) A memoryless source, producing the N letters a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$. Let $C = C_1$ be an associated binary prefix code, and let \bar{l} be the average word length of its code words.
For $n \geq 1$ let us encode as follows:

$$\mathbf{x} = a_{j_1} a_{j_2} \cdots a_{j_n} \longmapsto c(\mathbf{x}) = c(a_{j_1})c(a_{j_2}) \cdots c(a_{j_n}).$$

Let C_n be the ensuing code; $n \geq 1$.

- (a) Show that C_n is a binary prefix code associated with $\mathbf{p}^{(n)}$, $n \geq 1$.
- (b) Show that we have, for each C_n : $\bar{l}_i = \bar{l}$.
- (c) Give a necessary and sufficient condition for all the C_n to be optimal (with respect to $\mathbf{p}^{(n)}$).

1.1.3 Arithmetic Coding

The arithmetic coding is a dynamic version (presented by a recursive algorithm) of Shannon block encoding (with continually increasing block lengths). Actually, all of these block codes will be visited very shortly: for every block length $n \geq 1$, we shall encode only a single word of length n : the initial segment of length n of a given stream of source symbols.

Arithmetic coding is much easier to explain than to understand. That is why we shall adopt – at least at the beginning – a very pedantic viewpoint. Towards the end, we shall be concerned with more practical aspects of arithmetic coding.

The Elias Encoder

The Elias encoder was initially meant to be a purely academic construction. Its first (and rather discreet) presentation dates from 1968. It was between 1976 and 1979 (Pasco, Jones, Rubin) that arithmetic coding began to be considered as a practically interesting method for lossless compression.

The situation A memoryless source, producing the N letters a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.

We shall always suppose $p_0 \geq p_1 \geq \dots \geq p_{N-1}$.

The arithmetic encoder will associate with a stream of source symbols $a_{j_1} a_{j_2} \dots a_{j_n} \dots$ (which could be theoretically unlimited), a bitstream $\alpha_1 \alpha_2 \alpha_3 \dots \alpha_l \dots$ (which would then also be unlimited).

But let us stop after n encoding steps:

The code word $\alpha_1 \alpha_2 \alpha_3 \dots \alpha_l$ of l bits associated with the n first source symbols $a_{j_1} a_{j_2} \dots a_{j_n}$ will be the code word $c(a_{j_1} a_{j_2} \dots a_{j_n})$ of a Shannon block encoding formally adapted to recursiveness according to the device: “every step yields a tree-antecedent to the next step”.

We shall, in particular, inherit from the preceding section: if the actual production of the source is statistically correct, then l will be the *average length* of the code words for a Shannon block encoding, and consequently

$$H(\mathbf{p}) \leq \frac{l}{n} < H(\mathbf{p}) + \frac{1}{n}.$$

In this sense, the arithmetic encoder will work optimally, i.e. very close to the entropy of the source. The number of bits produced by the arithmetic encoder, counted per source symbol, will be asymptotically equal to the entropy of the source. As a first initiation in arithmetic coding, let us consider a “superfluous” version.

Example A (memoryless) source, producing the four letters a, b, c, d according to the probability distribution given by $p(a) = \frac{1}{2}$, $p(b) = \frac{1}{4}$, $p(c) = p(d) = \frac{1}{8}$.

The Shannon code:

$$\begin{aligned} a &\longmapsto 0 & 0 &= 0.\underline{0}000\dots \\ b &\longmapsto 10 & \frac{1}{2} &= 0.\underline{1}000\dots \\ c &\longmapsto 110 & \frac{3}{4} &= 0.\underline{11}000\dots \\ d &\longmapsto 111 & \frac{7}{8} &= 0.\underline{111}000\dots \end{aligned}$$

Now, look at the source word *badacaba*. Its code word is 10011101100100. Let us write down this concatenation in “temporal progression”:

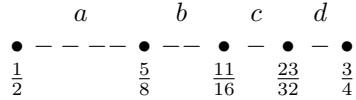
b	10
ba	100
bad	100111
$bada$	1001110
$badac$	1001110110
$badaca$	10011101100
$badacab$	1001110110010
$badacaba$	10011101100100

Recall our binary tree of all *standard-intervals* obtained by iterated dichotomy.

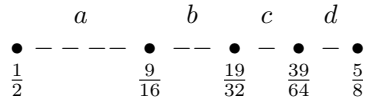
10	is the code word of the interval	$\left[\frac{1}{2}, \frac{3}{4}\right[$,
100	is the code word of the interval	$\left[\frac{1}{2}, \frac{5}{8}\right[$,
100111	is the code word of the interval	$\left[\frac{39}{64}, \frac{5}{8}\right[$,
1001110	is the code word of the interval	$\left[\frac{39}{64}, \frac{79}{128}\right[$,
1001110110	is the code word of the interval	$\left[\frac{315}{512}, \frac{631}{1024}\right[$,
10011101100	is the code word of the interval	$\left[\frac{315}{512}, \frac{1,261}{2,048}\right[$,
1001110110010	is the code word of the interval	$\left[\frac{2,521}{4,096}, \frac{5,043}{8,192}\right[$,
10011101100100	is the code word of the interval	$\left[\frac{2,521}{4,096}, \frac{10,085}{16,384}\right[$.

We have obtained a *chain* of eight intervals: $\left[\frac{1}{2}, \frac{3}{4}\right[$ is the interval of b for the Shannon partition of the interval $[0, 1[$.

The interval $\left[\frac{1}{2}, \frac{5}{8}\right[$ is the interval of a for the Shannon partition of the interval $\left[\frac{1}{2}, \frac{3}{4}\right[$:



The interval $\left[\frac{39}{64}, \frac{5}{8}\right[$ is the interval of d for the Shannon partition of the interval $\left[\frac{1}{2}, \frac{5}{8}\right[$:



and so on...

Let us insist:

b points at its interval $\left[\frac{1}{2}, \frac{3}{4}\right[$.

ba points at the interval of a within the interval of b .

bad points at the interval of d within the interval of a within the interval of b , and so on...

The recursive algorithm for the construction of the chain of intervals corresponding to the successive arrivals of the source symbols:

Start: $A_0 = 0$ $B_0 = 1$.

Step: The initial segment $s_1 s_2 \cdots s_m$ of the source stream points at $[A_m, B_m[$.

Compute three division points D_1 , D_2 and D_3 for the interval $[A_m, B_m[$:

$$D_1 = A_m + p(a)(B_m - A_m),$$

$$D_2 = A_m + (p(a) + p(b))(B_m - A_m),$$

$$D_3 = A_m + (p(a) + p(b) + p(c))(B_m - A_m).$$

Let s_{m+1} be the $(m+1)$ st source symbol.

$$\text{If } s_{m+1} = \begin{cases} a \\ b \\ c \\ d \end{cases} \text{ then } \begin{cases} A_{m+1} = A_m, B_{m+1} = D_1, \\ A_{m+1} = D_1, B_{m+1} = D_2, \\ A_{m+1} = D_2, B_{m+1} = D_3, \\ A_{m+1} = D_3, B_{m+1} = B_m. \end{cases}$$

End: $m = n$. The code word of $s_1 s_2 \cdots s_n$ is the word $c(A_n, B_n) \equiv$ the code word of the interval $[A_n, B_n[$.

Note that the least upper bounds B_n of our intervals are not very important (and are often completely neglected): the code word is a prefix of the binary notation of A_n , and its length l is determined by the information content $I(s_1 s_2 \cdots s_n)$.

Let us sum up our observations:

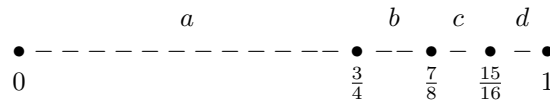
The simple Shannon coding $c(badacaba) = c(b)c(a)c(d)c(a)c(c)c(a)c(b)c(a)$ becomes considerably more complicated from our viewpoint of “dynamic block coding” and by the arithmetic generating the correct strings of intervals. Our first impression is that there is no advantage in adopting this new vision of computing code words.

But now let us change the data.

Our source will still produce the four letters a, b, c, d , but now according to a new probability distribution given by $p(a) = \frac{3}{4}$, $p(b) = \frac{1}{8}$, $p(c) = p(d) = \frac{1}{16}$.

We compute: $I(a) = 0.42$, $I(b) = 3$, $I(c) = I(d) = 4$.

The Shannon code:



$a \mapsto 0$

$b \mapsto 110$

$c \mapsto 1110$

$d \mapsto 1111$

Let us choose a source word in conformity with the statistics: *daaabaaa-caaabaaa*.

The associated Shannon code word is 11110001100001110000110000 and has 26 bits.

But if we adopt an arithmetic encoding, making use of the recursive algorithm explained earlier, we will end with a code word of l bits, where $l = \lceil I(daaabaaacaaabaaa) \rceil = \lceil 12I(a) + 2I(b) + I(c) + I(d) \rceil = \lceil 5.04 + 6 + 8 \rceil = 20$.

So, the arithmetic code word of *daaabaaacaaabaaa* is *shorter* than the (concatenated) Shannon code word. This is a general fact: whenever the probabilities are not powers of $\frac{1}{2}$, arithmetic coding is better than any block coding (of fixed block length).

Exercises

- (1) The memoryless source which produces the four letters a, b, c, d , according to the probability distribution given by $p(a) = \frac{3}{4}$, $p(b) = \frac{1}{8}$, $p(c) = p(d) = \frac{1}{16}$.
Compute the arithmetic code word of *daaabaaacaaabaaa* (thus completing the example above).
- (2) A memoryless binary source such that $p_0 = \frac{3}{4}$, $p_1 = \frac{1}{4}$.
Compute the arithmetic code word of 00101000.
- (3) A memoryless source producing the three letters a, b, c according to the probability distribution given by $p(a) = \frac{3}{4}$, $p(b) = p(c) = \frac{1}{8}$.
Find the arithmetic code word of *aabaacaa*.
- (4) Write down the general version of the recursive algorithm for arithmetic coding. *Recall*: we have to do with a memoryless source producing N letters a_0, a_1, \dots, a_{N-1} , according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$, and $p_0 \geq p_1 \geq \dots \geq p_{N-1} > 0$.
- (5) The situation as in exercise (4). Suppose that all probabilities are powers of $\frac{1}{2}$: $p_j = 2^{-l_j}$, $0 \leq j \leq N-1$.
Show that in this case the arithmetic code word of a source word $s_1 s_2 \dots s_n$ is *equal* to the Shannon code word (obtained by simple concatenation of the code words for s_1, s_2, \dots, s_n).
- (6) A memoryless source, producing the N letters a_0, a_1, \dots, a_{N-1} according to the (“decreasing”) probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$.
Let \mathbf{s}_1 and \mathbf{s}_2 be two source words such that $c(\mathbf{s}_1) = c(\mathbf{s}_2)$ (they have the *same* arithmetic code word).
Show that either \mathbf{s}_1 is a prefix of \mathbf{s}_2 or \mathbf{s}_2 is a prefix of \mathbf{s}_1 .

[**Solution** \mathbf{s}_1 points at $[A_m, B_m[$.
 \mathbf{s}_2 points at $[C_n, D_n[$.

Our hypothesis: $A_m = 0 \cdot \alpha_1 \alpha_2 \dots \alpha_l^*$, $C_n = 0 \cdot \alpha_1 \alpha_2 \dots \alpha_l^*$, where $l = \lceil -\text{Log}_2(B_m - A_m) \rceil = \lceil -\text{Log}_2(D_n - C_n) \rceil$.

This means:

$$2^{-l} \leq B_m - A_m < 2 \cdot 2^{-l},$$

$$2^{-l} \leq D_n - C_n < 2 \cdot 2^{-l}.$$

Suppose $m \leq n$.

We have only to show that $[C_n, D_n] \subseteq [A_m, B_m[$ (this inclusion implies that s_1 is a prefix of s_2).

Otherwise, we would have $[C_n, D_n] \cap [A_m, B_m[= \emptyset$

Hence, $|C_n - A_m| \geq 2^{-l}$, i.e. the binary notations of A_m and of C_n would already differ (at least) at the l th position after the comma. But this is a contradiction to our hypothesis above. \square

The Arithmetic Decoder

The arithmetic decoder will reconstruct, step by step, the source stream from the code stream. It has to discern the encoding decisions by the means of the information which arrive with the code stream. Every decision of the encoder is dedicated to the identification of a source symbol – a process which has to be imitated by the decoder.

This is the occasion where our modelling in terms of (chains of) intervals will be helpful.

First, consider the situation from a *static* viewpoint. The decoder can take into account the *complete* code stream $\alpha_1\alpha_2\cdots\alpha_L$.

This means that we face the code word $c(A_n, B_n)$ of an interval

$A_n = 0 \cdot \alpha_1\alpha_2\cdots\alpha_L$ (* \equiv masked part),

$L = \lceil -\log_2(B_n - A_n) \rceil$.

We search: the source stream $s_1s_2\cdots s_n$, which points at this interval (in the Shannon partition tree whose structure comes from the lexicographic ordering of the pointers).

Let us look at the following.

Example A (memoryless) source producing the three letters a, b, c according to the probability distribution \mathbf{p} given by $p(a) = \frac{3}{4}, p(b) = p(c) = \frac{1}{8}$.

The code word: 100110111.

Our terminal interval $[A_n, B_n[$ is described by

$A_n = 0.100110111$,

$\frac{1}{2^9} \leq B_n - A_n < \frac{1}{2^8}$.

The decoding will consist of keeping track of the *encoder's* decisions.

The logical *main argument* for decoding is the following: the hierarchy of the Shannon partition tree accepts only *chains of intervals* or *empty intersections*.

This means: whenever a *single* point of such an interval is to the left or to the right of a division point, then the *entire* interval will be to the left or to the right of this point (recall the solution of exercise (6) at the end of the preceding section).

Let us begin with the decoding.

First step:

Compute $D_1 = \frac{3}{4} = 0.11$,

$D_2 = \frac{7}{8} = 0.111$,

$A_n < D_1 \implies [A_n, B_n[\subset [0, D_1[\equiv$ the interval of a .

$\Rightarrow s_1 = a$.

Thus, $A_1 = 0$, $B_1 = \frac{3}{4}$.

Second step:

Compute $D_1 = \frac{9}{16} = 0.1001$,

$$D_2 = \frac{21}{32} = 0.10101,$$

$D_1 < A_n < D_2 \Rightarrow [A_n, B_n[\subset [D_1, D_2[\equiv \text{the interval of } ab$.

$\Rightarrow s_1 s_2 = ab$.

Thus, $A_2 = \frac{9}{16}$, $B_2 = \frac{21}{32}$.

Third step:

Compute $D_1 = \frac{81}{128} = 0.1010001$,

$$D_2 = \frac{93}{128} = 0.1011101,$$

$A_n < D_1 \Rightarrow [A_n, B_n[\subset [A_2, D_1[\equiv \text{the interval of } aba$.

$\Rightarrow s_1 s_2 s_3 = aba$.

Thus, $A_3 = \frac{9}{16}$, $B_3 = \frac{81}{128}$.

Fourth step:

Compute $D_1 = \frac{315}{512} = 0.100111011$,

$$D_2 = \frac{639}{1024} = 0.1001111111,$$

$A_n < D_1 \Rightarrow [A_n, B_n[\subset [A_3, D_1[\equiv \text{the interval of } abaa$.

$\Rightarrow s_1 s_2 s_3 s_4 = abaa$.

Thus, $A_4 = \frac{9}{16}$, $B_4 = \frac{315}{512}$.

Fifth step:

Compute $D_1 = \frac{1233}{2048} = 0.10011010001$,

$$D_2 = \frac{2493}{4096} = 0.100110111101,$$

A_n and D_2 have the same binary notation – until the masked part of A_n .

Question How shall we continue?

Answer $A_n = D_2$ and $s_5 = c$ (i.e. $[A_n, B_n[\subset [D_2, B_4[$).

Justification If we suppose that $A_n = D_2$ and $s_5 = c$, then we have found a source word $\mathbf{s} = s_1 s_2 s_3 s_4 s_5 = abaac$ with the code word 10011011 (note that $I(abaac) = 3 \times 0.42 + 3 + 3 = 7.26$).

Suppose, furthermore, that only the letter a was produced in the sequel ($\frac{2493}{4096}$ remains then the left end point $A_5 = A_6 = A_7$, etc. of the code interval), so we will have $s_1 s_2 s_3 s_4 s_5 s_6 s_7 = abaacaa$ with $I(abaacaa) = 5 \times 0.42 + 3 + 3 = 8.1$.

This source word will produce our code word, but also

$$s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 = abaacaaa \quad (I(s_1 \cdots s_8) = 8.52),$$

$$s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 = abaacaaaa \quad (I(s_1 \cdots s_9) = 8.94).$$

We obtain this way three source words

$$s_1 s_2 s_3 s_4 s_5 s_6 s_7,$$

$$s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8,$$

$$s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9,$$

which produce the same code word 10011011.

Back to the question: why $A_n = \frac{2493}{4096}$?

The answer: Under this hypothesis, we obtain a coherent and logical decoding. But, according to exercise (6) of the preceding section, *every* string of source symbols with the code word 100110111 must be a *prefix* of *abaacaaaa*.

Attention *The argument concerning the end of our decoding algorithm depends on a complete message: the string of the received code bits has to be the code word of an interval.*

Now consider an *incomplete message*.

In the situation of our example, assume that the first three bits of the code stream are 101. This is not a code word of an interval in our partition tree associated with the given probability distribution. Let us try to determine altogether the first two source symbols. 101 is the prefix of the binary notation of the greatest lower bound A of an interval $[A, B[$ coming up at some step of the encoding:

$$A = 0.101 * .$$

We note: $A < \frac{3}{4} = D_1 \equiv$ the inferior division point of the first step of the encoder,

$$\implies s_1 = a.$$

The division points of the second step (inside the interval $[A_1, B_1[= [0, \frac{3}{4}[$):

$$D_1 = \frac{9}{16} = 0.1001, D_2 = \frac{21}{32} = 0.10101,$$

$$A > D_1 \implies s_2 = b \text{ or } s_2 = c.$$

The case $s_1 s_2 = ac$ gives $A_2 = 0.10101$, and this is ok.

The case $s_1 s_2 = ab$ needs a closer inspection of the possible continuations.

And indeed: the code word of $s_1 s_2 s_3 = abc$ is 1010010, since the second division point D_2 inside $[A_2, B_2[= [\frac{9}{16}, \frac{21}{32}[$ is $D_2 = \frac{165}{256} = 0.10100101$.

We see: our rules about the decoding at the end of the code stream can only be applied in a *terminal* situation. We have to face the code word of an interval.

Exercises

- (1) Write down the decoding algorithm in general, i.e. for a memoryless source, producing the N letters a_0, a_1, \dots, a_{N-1} according to the probability distribution $\mathbf{p} = (p_0, p_1, \dots, p_{N-1})$ with $p_0 \geq p_1 \geq \dots \geq p_{N-1} > 0$.
- (2) A (memoryless) binary source $\mathbf{p} = (p_0, p_1)$ with $p_0 = \frac{3}{4}$, $p_1 = \frac{1}{4}$. Decode 1101010.
- (3) The binary source of exercise (2). The decoder receives a code stream beginning with 1001. Find the first three bits of the source stream.
- (4) A memoryless source which produces the four letters a, b, c, d according to the probability distribution \mathbf{p} given by $p(a) = \frac{3}{4}$, $p(b) = \frac{1}{8}$, $p(c) = p(d) = \frac{1}{16}$. Decode 11101111000101010000.
- (5) A (binary) arithmetic encoder produces zeros and ones; hence, it is nothing but a binary source. What then is its entropy?

Practical Considerations

It is clear that the spirit of arithmetic coding demands a continual bitstream output of the encoder. On the other hand, the constraints of a bounded arithmetic call for perpetual renormalization in order to avoid an asphyxiation in more and more complex computations. So, we need a practicable version of the Elias encoder and decoder, i.e. a sort of prototype for effective implementation.

Some (almost) trivial observations

Let $[A, B[$ be a (small) interval taking place in our computations. We are interested in the division point

$$D = A + p \cdot (B - A) \quad (p \equiv \text{a cumulative probability}).$$

If the difference $B - A$ is *small*, then we will have (generically – there are the well-known exceptions)

$$\begin{aligned} A &= a + r, \\ B &= a + s, \end{aligned}$$

where a describes the *identical* prefix of the binary notation⁵ of A and of B .

Compute $d = r + p \cdot (s - r)$ (subdivision of the remainder parts ...).

Then $D = a + d$.

On the other hand (stability of our computations under “arithmetic zoom”): consider, for $m \geq 1$,

$$R = 2^m r,$$

$$S = 2^m s.$$

$$\text{Then } R + p \cdot (S - R) = 2^m (r + p \cdot (s - r)).$$

Consequence *Suppose that the binary notations of A and of B are identical up to a certain position t :*

$$\begin{aligned} A &= \alpha_1 \cdots \alpha_t \alpha_{t+1} \cdots, \\ B &= \alpha_1 \cdots \alpha_t \beta_{t+1} \cdots. \end{aligned}$$

Then the computation of $D = A + p \cdot (B - A)$ is equivalent to the computation of $T = R + p \cdot (S - R)$

$$\text{with } R = 0 \cdot \alpha_{t+1} \cdots$$

$$S = 0 \cdot \beta_{t+1} \cdots$$

More precisely, if $T = 0 \cdot \tau_1 \tau_2 \cdots$, then $D = 0 \cdot \alpha_1 \cdots \alpha_t \tau_1 \tau_2 \cdots$

This allows a renormalizable arithmetic: our computations remain in the higher levels of appropriate partition trees of simple intervals inside $[0, 1[$.

The criterion for sending off a string of code bits will come from a simple syntactical comparison.

Let $[A_n, B_n[$ be the interval of the source word $s_1 s_2 \cdots s_n$.

Write $A_n = 0 \cdot \alpha_1 \alpha_2 \cdots \alpha_t \alpha_{t+1} \cdots$,

$$B_n = 0 \cdot \beta_1 \beta_2 \cdots \beta_t \beta_{t+1} \cdots.$$

⁵ We shall never consider “false” infinite notations with period “1”.

If $\alpha_1 = \beta_1$ $\alpha_2 = \beta_2 \cdots \alpha_t = \beta_t$, then our encoder will produce (send off) the string of code bits $\alpha_1 \alpha_2 \cdots \alpha_t$ and will continue the encoding with

$$\begin{aligned} A'_n &= 0 \cdot \alpha_{t+1} \cdots, \\ B'_n &= 0 \cdot \beta_{t+1} \cdots. \end{aligned}$$

Example The (memoryless) source which produces the three letters a, b, c according to the probability distribution \mathbf{p} given by $p(a) = \frac{3}{4}$, $p(b) = p(c) = \frac{1}{8}$.

Consider the source word *abaacaaaa* (an old friend – look at our decoding example in the last section!).

First step of the encoder:

$$\begin{aligned} D_1 &= \frac{3}{4} = 0.11, \\ D_2 &= \frac{7}{8} = 0.111, \\ s_1 = a &\implies A_1 = 0 = 0.000 \dots, \\ &\quad B_1 = \frac{3}{4} = 0.11. \end{aligned}$$

No common prefix can be discarded.

Second step of the encoder:

$$\begin{aligned} D_1 &= \frac{9}{16} = 0.1001, \\ D_2 &= \frac{21}{32} = 0.10101, \\ s_2 = b &\implies A_2 = \frac{9}{16} = 0.\underline{1}001, \\ &\quad B_2 = \frac{21}{32} = 0.1\underline{0}101. \end{aligned}$$

We send off: $\alpha_1 \alpha_2 = 10$.

We renormalise: $A'_2 = 0.01 = \frac{1}{4}$,
 $B'_2 = 0.101 = \frac{5}{8}$.

Third step of the encoder:

$$\begin{aligned} D_1 &= \frac{1}{4} + \frac{3}{4} \cdot \frac{3}{8} = \frac{17}{32} = 0.10001, \\ D_2 &= \frac{1}{4} + \frac{7}{8} \cdot \frac{3}{8} = \frac{37}{64} = 0.100101, \\ s_3 = a &\implies A_3 = \frac{1}{4} = 0.01, \\ &\quad B_3 = \frac{17}{32} = 0.10001. \end{aligned}$$

No common prefix can be discarded.

Fourth step of the encoder:

$$\begin{aligned} D_1 &= \frac{1}{4} + \frac{3}{4} \cdot \frac{9}{32} = \frac{59}{128} = 0.0111011, \\ D_2 &= \frac{1}{4} + \frac{7}{8} \cdot \frac{9}{32} = \frac{127}{256} = 0.01111111, \\ s_4 = a &\implies A_4 = \frac{1}{4} = 0.01, \\ &\quad B_4 = \frac{59}{128} = 0.\underline{0}111011. \end{aligned}$$

We send off: $\alpha_3 \alpha_4 = 01$.

We renormalise: $A'_4 = 0$,
 $B'_4 = 0.11011 = \frac{27}{32}$.

Fifth step of the encoder:

$$\begin{aligned} D_1 &= \frac{3}{4} \cdot \frac{27}{32} = \frac{81}{128} = 0.1010001, \\ D_2 &= \frac{7}{8} \cdot \frac{27}{32} = \frac{189}{256} = 0.10111101, \\ s_5 = c &\implies A_5 = \frac{189}{256} = 0.10111101 \\ &\quad B_5 = \frac{27}{32} = 0.\underline{1}1011. \end{aligned}$$

We send off: $\alpha_5 = 1$.

We renormalise: $A'_5 = 0.0111101 = \frac{61}{128}$,
 $B'_5 = 0.1011 = \frac{11}{16}$.

Sixth step of the encoder:

$$D_1 = \frac{61}{128} + \frac{3}{4} \cdot \frac{27}{128} = \frac{325}{512} = 0.101000101,$$

$$s_6 = a \implies A_6 = A_5 = 0.0111101,$$

$$B_6 = D_1 = 0.101000101.$$

No common prefix can be discarded.

Seventh step of the encoder:

$$D_1 = \frac{61}{128} + \frac{3}{4} \cdot \frac{81}{512} = \frac{1,219}{2,048} = 0.10011000011,$$

$$s_7 = a \implies A_7 = A_6 = 0.0111101,$$

$$B_7 = D_1 = 0.10011000011.$$

No common prefix can be discarded.

Eighth step of the encoder:

$$D_1 = \frac{61}{128} + \frac{3}{4} \cdot \frac{243}{2,048} = \frac{4,633}{8,192} = 0.1001000011001,$$

$$s_8 = a \implies A_8 = A_7 = 0.0111101,$$

$$B_8 = D_1 = 0.1001000011001.$$

No common prefix can be discarded.

Ninth step of the encoder:

$$D_1 = \frac{61}{128} + \frac{3}{4} \cdot \frac{729}{8,192} = \frac{17,803}{32,768} = 0.100010110001011,$$

$$s_9 = a \implies A_9 = A_8 = 0.0111101,$$

$$B_9 = D_1 = 0.100010110001011.$$

No common prefix can be discarded.

Exercises

- (1) Continue the previous example: find the shortest source word $s_1 s_2 \dots s_9 s_{10} \dots$ such that the encoder will effectively send off (after the convenient syntactical tests) $\alpha_6 \alpha_7 \alpha_8 \alpha_9 = 0111$.
- (2) True or false: if $s_1 s_2 \dots s_n$ is the beginning of the source stream, then its code word $c(s_1 s_2 \dots s_n)$ is the beginning of the code stream?
- (3) Our example cited earlier seems to indicate the danger of a *blocking* due to inefficiency of the syntactical test: it is the constellation $[A_n, B_n[= [0.011 \dots 1*, 0.100 \dots 0* [$ which looks dangerous for renormalization. Try to control the situation numerically. Will we need an algorithmic solution (exceptional case)?

Remark In arithmetic coding, the sequence of source symbols $s_1 s_2 \dots s_n$ will be encoded and decoded progressively; this allows us to implement an *adaptive version* of arithmetic coding which will learn progressively the actual probability distribution $\mathbf{p}^{(n)}$ (after the production of the first n source symbols).

Let us make this more precise: at the beginning, there is no statistical information concerning the production of the N symbols a_0, a_1, \dots, a_{N-1} . We are obliged to put $\mathbf{p}^{(0)} = (\frac{1}{N}, \frac{1}{N}, \dots, \frac{1}{N})$ (uniform distribution).

Now let $\mathbf{p}^{(n)} = (p_0^{(n)}, \dots, p_{N-1}^{(n)})$ be the actual distribution at the n th pulse.

$\mathbf{p}^{(n)}$ was computed at the last node n_k of a control pulsation ($n_k \leq n$) according to the histogram of $s_0 s_1 \dots s_{n_k}$.

Let $s_{n+1} = a_j$ ($0 \leq j \leq N-1$).

Then we shall put: $A_{n+1} = A_n + \left(\sum_{k=0}^{j-1} p_k^{(n)} \right) (B_n - A_n),$

$$B_{n+1} - A_{n+1} = p_j^{(n)} \cdot (B_n - A_n).$$

The decoder will recover s_{n+1} , since it knows $\mathbf{p}^{(n)}$ – the actual probability distribution after the production of the n th character – due to the histogram established with the information of anterior decoding.

1.2 Universal Codes: The Example LZW

The algorithms for data compaction which we shall treat now are “universal” in the following sense: the idea of a memoryless source – which is a nice but very rare object – will be sacrificed. We shall joyfully accept source productions of a high statistical complexity; this means that explicit statistical evaluations of the source production will disappear. It is a new way of coding (by means of a *dictionary*) that creates an *implicit* histogram.

The birth of the central algorithm Lempel–Ziv–Welch (LZW) dates to around 1977. There exist a lot of clever variants. We shall restrict ourselves to the presentation of a single version which is seductively clear and simple.

1.2.1 LZW Coding

Situation *A source produces a stream of letters, taken from a finite alphabet.*

The encoder will establish a *dictionary* of strings of characters (of motives) which are *characteristic* for the source stream (and it will create this way *implicit statistics* of the source production). The compressed message is given by the stream of *pointers* (\equiv the numbers attributed to the strings in the dictionary). Note that we meet here a method of coding where the code words have *fixed* length.

The Encoding Algorithm

The *principal aim* of the encoding algorithm is the writing of a *dictionary*:

Strings of characters \longleftrightarrow Numbers for these strings

The production of the code stream (i.e. of the sequence of pointers) is (logically) secondary.

The encoding algorithm works in *three steps*:

- (1) Read the next character x (arriving from the source).
- (2) Complete a *current string* s (which is waiting in a buffer for admission to the dictionary) by concatenation: $s \mapsto sx$.
- (3) Write the current string into the dictionary as soon as it will be admissible (i.e. unknown to the dictionary) – otherwise go back to (1). Produce (i.e. send off) the number of s at the moment where you write sx into the dictionary; initialize $s = x$.

Remark What is the reason for this enigmatic delay at the output of the encoder? Let us insist: the encoder produces the code word of s at the moment when it writes sx into the dictionary.

The reason is the following. The decoder will have to *reconstruct* the dictionary by means of the code stream which it receives (the dictionary of the encoder will not survive the end of the compaction). Without this delay between writing and producing, the reconstruction of the dictionary by the decoder would be impossible.

Let us recapitulate the encoding algorithm in a more formalized version.

STEP: read the next character x of the source stream.

- If no character x (end of message),
then produce the code word (the pointer) of the current string s ;
end.
- If the string sx exists already in the table,
then replace s by the new current string sx ;
 repeat STEP.
- If the string sx is not yet in the table,
then write sx in the table, produce the code of s and put $s = x$;
 repeat STEP.

Example Initialization of the dictionary: (1) a
 (2) b
 (3) c

Encode *baca caba baba caba*.

Read	Produce	Write	Current string
		(1) a	
		(2) b	
		(3) c	
b			b
a	(2)	(4) ba	a
c	(1)	(5) ac	c
a	(3)	(6) ca	a
c			ac
a	(5)	(7) aca	a
b	(1)	(8) ab	b
a			ba
b	(4)	(9) bab	b
a			ba
b			bab
a	(9)	(10) $baba$	a
c			ac
a			aca
b	(7)	(11) $acab$	b
a			ba
	(4)		

Look, once more, how the source stream is transformed into the code stream:

b a c ac a ba bab aca ba

(2) (1) (3) (5) (1) (4) (9) (7) (4)

Exercises

- (1) The situation of the preceding example.
Encode now *baba caba baca caba baba caba*.
- (2) A binary source. We initialize: (0) 0
(1) 1
Encode 010101010101010101010101.
- (3) (a) Give an example of an LZW encoding which produces a sequence of pointers of the type ... (#)(#) ... (i.e. with a pointer doubled).
(b) Does there exist sequences of LZW code words of the type ... (#)(#)(#) ... (i.e. with a pointer tripled)?

1.2.2 The LZW Decoder

A First Approach

The principal goal of the decoder is the *reconstruction of the dictionary* of the encoder. It has to correctly interpret the stream of code words (pointers) that it receives. The down-to-earth decoding (the identification of the code words) is a part of this task.

The *current string* s , candidate for admission to the dictionary, will still remain in the centre of the algorithm. Let us begin immediately with an example.

Example Initialized dictionary: (1) *a*
(2) *b*
(3) *c*

Decode (3)(1)(2)(5)(1)(4)(6)(6).

Read	Produce	Write	Current string
		(1) <i>a</i>	
		(2) <i>b</i>	
		(3) <i>c</i>	
(3)	<i>c</i>		<i>c</i>
(1)	<i>a</i>	(4) <i>ca</i>	<i>a</i>
(2)	<i>b</i>	(5) <i>ab</i>	<i>b</i>
(5)	<i>ab</i>	(6) <i>ba</i>	<i>ab</i>
(1)	<i>a</i>	(7) <i>aba</i>	<i>a</i>
(4)	<i>ca</i>	(8) <i>ac</i>	<i>ca</i>
(6)	<i>ba</i>	(9) <i>cab</i>	<i>ba</i>
(6)	<i>ba</i>	(10) <i>bab</i>	<i>ba</i>

Attention We note that the two first columns are the result of a mechanical identification according to the input information of the decoder, whereas the two last columns come from a backward projection. What did the encoder do? Recall that the encoder can only write into the dictionary when appending a single character to the current string. Every word in the dictionary is preceded by the “dispersed pyramid” of all its prefixes. The steps of non-writing of the encoder disappear during the decoding. We write at every step.

We observe:

- (1) *At the beginning of every decoding step, the current string will be the prefix of the next writing into the dictionary. We append the first character of the decoded string.*
- (2) *At the end of every decoding step, the current string will be equal to the string that we just decoded (the column “produce” and the column “current string” of our decoding model are identical: at the moment when we identify a code word, the demasked string appears in the “journal” of the encoder – a consequence of the small delay for the output during the encoding).*

The Exceptional Case

Example The situation is as in the first example. Decode (2)(1)(4)(6).

Read	Produce	Write	Current string
		(1) <i>a</i>	
		(2) <i>b</i>	
		(3) <i>c</i>	
(2)	<i>b</i>		<i>b</i>
(1)	<i>a</i>	(4) <i>ba</i>	<i>a</i>
(4)	<i>ba</i>	(5) <i>ab</i>	<i>ba</i>
(6)	<i>bax</i>	(6) <i>bax</i>	<i>bax</i>

Look a little bit closer at the last line of our decoding scheme. We have to identify a pointer which points at nothing in the dictionary, but which shall get its meaning precisely at this step. But recall our previous observations. We need to write a string of the form *bax* (current string plus a new character); but this must be, at the same time, the decoded string. On the other hand, the character *x* must be the *first* character of the decoded string. Thus, $x = b$ (= the first character of the current string).

All these considerations generalize:

We have to identify a code word (a pointer) the source word of which is not yet in the dictionary. We proceed precisely as in the foregoing example.

Let *s* be the current string at the end of the last step. Arrives the fatal pointer (‡). If we take our decoding observations (1) and (2) as rules, then we are obliged to produce: *sx* write: (‡) : *sx* put: $s = sx$, *x* is an (a priori)

unknown character, which will be identified according to the rule: x = the first character of the current string s .

The question remains: what is the encoding constellation which provokes the exceptional case of the decoder?

Exercise Show that the exceptional decoding case comes from the following encoding situation: the encoder sends off the last pointer available in the dictionary.

The Decoding Algorithm

Let us sum up, in pseudo-program version, the functioning of our decoder:

```

STEP: read the next pointer ( $n$ ) of the code stream.
  if    no such ( $n$ ), end.
  if    the string of ( $n$ ) is not in the dictionary,
    then produce  $sx$ , where  $x$  is the first character of the
          current string  $s$ , write  $sx$  into the dictionary,
          replace  $s$  by the new current string  $sx$ 
          repeat STEP
  else  the string  $u$  of ( $n$ ) exists in the dictionary,
    then produce  $u$ , write  $sx$  into the dictionary, where  $s$  is the
          current string, and  $x$  is the first character of  $u$ , replace
          the current string by  $u$ ,
          repeat STEP.

```

Exercises

- (1) Initialized dictionary: (1) a (2) b (3) c (4) d
 Decode (1)(3)(4)(3)(5)(7)(2)(11)(2)(1)(2)(6)(1)(17)(1).
- (2) Initialized dictionary: (1) a (2) b (3) c (4) d
 Decode (3)(1)(2)(6)(8)(4)(8)(7)(10)(5)(10)(7)(12)(11)(9)(1)(5)(14)(18)
 (5)(16)(19)(25)(13)(18)(28)(10)(15)(18)(7)(21)(20)(1)(32)(25)
 (39)(33)(18).
- (3) A LZW decoder, initialized: (1) a (2) $b \dots$ (26) z .
 At its 76th decoding step the decoder encounters an exceptional case.
 What is the number of this pointer?
- (4) Initialized dictionary: (0) 0 (1) 1
 - (a) What is the bitstream which is encoded in (0)(1)(2)(3)(4)(5)(6)(7)(8)
 (9)(10)(11)?
 - (b) Let us continue: we shall produce in this way all the integers, in natural
 order, from 0 to 65,535.

Let us write the pointers in 16 bits. What will be the compressed bit-rate

$$\rho = \frac{\text{number of code bits}}{\text{number of source bits}}?$$

- (5) The situation of exercise (4). The binary source producing a bitstream such that the stream of LZW-codes will be $(0)(1)(2) \dots (65,534)(65,535)$. Model this source as a memoryless (!) source for the alphabet of the 256 8-bit bytes $00000000, 00000001, \dots, 11111111$. What will be (roughly) the source statistics? What will be its entropy?
- (6) Initialized dictionary: (1) *a* (2) *b* (3) *c* (4) *d*
 Decode (4)(1)(3)(6)(1)(5)(7)(7)(5)(5)(2)(1)(15)(14)(9)(19)(17)(1)(19)(12)
 (16)(19)(13)(22)(10)(20)(26)(25)(15)(33)(32)(35)(15)(24)(38)(22)
 (7)(1).



<http://www.springer.com/978-3-540-33218-3>

Algorithmic Information Theory
Mathematics of Digital Information Processing
Seibt, P.
2006, VI, 443 p., Hardcover
ISBN: 978-3-540-33218-3