*I*nformation Theory notes

*R*iccardo Lo Iacono

Unipa

Contents.

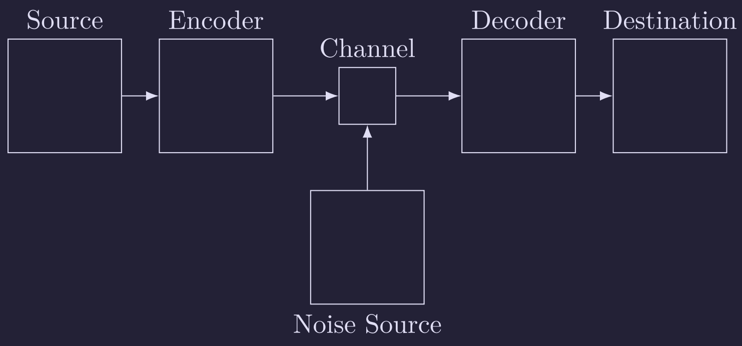
- 1 - Basics of information theory.

Information theory, alongside data compression, has a key role in modern computer science. The former defines a set of theoretical tools useful, among other things, to understand the limitations of what is computable; the latter allows to reduce the amount of space required (in terms of bits) without losing information.

- 1.1 - Shannon's Entropy.

When talking about information we all have a general notion of what it represents, but how is it defined formally? Also, is there a way to measure information?

The first to answer these questions was *Claude Shannon*, by many condsidered the father of information theory. In [8] Shannon analizes various communication systems: from the discrete noiceless one to the continous noisy one. A general structure of these systems is shown below.



Here:

- The *Source* or, more precisely, the *Information Source* referes to some entity (a human, a computer, etc) that produces messages.
- The *Encoder* encodes the messages coming from the source, and transmits them trough the channel. .
- The *Channel* is the physical mean trough which the messages are transmitted.
- The *Decoder* has the opposite role of the *Encoder*.
- The *Destination* is the entity to which the messages are meant for.

Remark. What follows in this section, and the those following, refers just to the discrete noiceless case. We also assume the source to be memoryless, meaning that each symbol produced is indipendent from the previous one.

Let S be a source of information. Let Σ be the alphabet of symbols used by the source, and for each symbols let $p_i = \Pr(S = \sigma_i)$ at any given time. What we are looking for is some function H, if it exists, that measures the uncertainty we have about S.

As we will show in *Section 1.1.1*, the only form H can assume is the following:

$$H = -K \sum_{i=1}^{n=|\,\Sigma\,|} p_i \log_b p_i. \tag{1}$$

where K is a positive constant and b is usually 2.

We define $H(S)$ to be the *entropy* of S. Formally, the entropy measures the average amount of information stored in each symbol of the source output. Thus, we can define the information as the quantity of uncertainty lost once the outcome of a given event is known.

Note. In this context, by "event" we refer to the symbol produced by the source at a given time.

- 1.1.1 - Entropy: an axiomatic definition.

In *Section 1.1* we stated that the only form H can assume is the one of *Equation* (1). The reason behind this, though one may use different approaches (eg. the connection with the entropy in quantum mechanics), is the way entropy is defined axiomatically.

Axiom. Let H be a measure of uncertainty of a source S, then H must satisfy the following properties.

1. H must be a continuos function of the probabilities. Meaning that if $H(S) = H(p_1, \ldots, p_n)$, then H is continuous in $[0, 1]$.
2. $H_2\left(\frac{1}{2}, \frac{1}{2}\right) = 1$.
3. If τ is a permutation of the probabilities, then $H(\tau) = H_n(p_1, \ldots, p_n)$. Meaning that, H must not care of the order of the probabilities.
4. If $A(n) = H_n\left(\frac{1}{n}, \ldots, \frac{1}{n}\right)$, then H is monotonically increasing. Equally probable events imply greater uncertainty.
5. If a choice can be divided into k consecutive ones, then the original value of H must be the weighted sum of the new H. Meaning that

$$\begin{aligned} H_n(p_1, \ldots, p_n) = {} & H_{n-k+1}(p_1 + \ldots + p_k, p_{k+1}, \ldots, p_n) \\ & + (p_1 + \ldots + p_k) H_k\left(\frac{p_1}{p_1 + \ldots + p_k}, \ldots, \frac{p_k}{p_1 + \ldots + p_k}\right). \end{aligned}$$

Theorem (Theorem 2, [8]) *The only function H that satisfies the above axioms, has the form of* Equation (1)*.*

Proof Let's show how *Equation* (1) satisfies all the axioms.

1. From axiom 5 we have $A(nm) = A(n) + A(m)$. Let's note in fact that:

$$\begin{aligned} A(nm) &= H_{nm}\left(\frac{1}{nm}, \dots, \frac{1}{nm}\right) \\ &= H_{nm-n+1}\left(\frac{1}{m}, \frac{1}{nm}, \dots, \frac{1}{nm}\right) + \frac{1}{m} H_n\left(\frac{1}{n}, \dots, \frac{1}{n}\right) \\ &\underbrace{= \dots =}_{m \text{ times}} H_m\left(\frac{1}{m}, \dots, \frac{1}{m}\right) + H_n\left(\frac{1}{n}, \dots, \frac{1}{n}\right) \\ &= A(m) + A(n). \end{aligned}$$

2. From the previous point, we also have that $A(n^k) = kA(n)$, for any n, k.

3. Let us consider some r big enough such that $\exists k : 2^k \leq n^r < 2^{k+1}$. It follows that

$$k \log_b 2 \leq r \log_b n < (k+1) \log_b 2.$$

Let's divide everything by $r \log_b 2$, we now have

$$\frac{k}{r} \leq \log_2 n < \frac{k+1}{r} \implies \left| \log_2 n - \frac{k}{r} \right| < \frac{1}{r}.$$

From axiom 4 follows $A(2^k) \leq A(n^r) < A(n^{k+1})$, which means that $kA(2) \leq rA(n) < (k+1)A(2)$. From a similar reasoning as before, we get

$$\left| \frac{A(n)}{A(2)} - \frac{k}{r} \right| < \frac{1}{r} \implies A(n) \approx \frac{A(2)}{\log_b 2} \log_b n$$

since ${}^{A(n)}/_{A(2)} \approx \log_b n$. Thus, $A(n) = c \log_b n$.

4. Suppose $p \in \mathbb{Q} \iff p = \frac{r}{s}, r \leq s$. Then we have

$$\begin{aligned} A(s) &= H_s\left(\frac{1}{s}, \dots, \frac{1}{s}\right) \\ &= H_{s-r+1}\left(\frac{r}{s}, \frac{1}{s}, \dots, \frac{1}{s}\right) + \frac{r}{s} H_r\left(\frac{1}{r}, \dots, \frac{1}{r}\right) \\ &= \dots = H_2\left(\frac{r}{s}, 1 - \frac{r}{s}\right) + \frac{s-r}{s} A(s-r) + \frac{r}{s} A(s). \end{aligned}$$

From the previous point, we can conclude that

$$A(n) = -c[(1-p)\log_b(1-p) + p\log_b p].$$

Also, since $\mathbb{Q}$ is dense in $\mathbb{R}$ and H is continuous, we can extend what above to any $p \in \mathbb{R}$.

5. Lastly, *Equation* (1) can be easily proved to hold, by induction on n.

$\blacksquare$

The proof we provide is berely a sketch. See [8] (*Appendix 2*) for a detailed proof.

- 1.2 - Source encoding and Sardinas-Patterson algorithm.

Let's now focus our attention onto the source itself. In general, the alphabet these use may not be suitable for transmission, for some reason or another. For this particular reason, an *encoder* is used to convert the source alphabet, to a new one, which is more suitable for transmission. On the recieving side, a *decoder* is used to convert back to the original alphabet. This process is called *source encoding/decoding*.

Formally, given a source S defined on some alphabet, and X a new alphabet, called *input alphabet*, we define a function $C : S \to X$ that maps sequences of symbols of S to sequences of symbols in X. A sequence of symbols in X is called a *codeword*.

Before we consider any particular *code*, let us consider the general case. Let's C be a generic mapping from a source S to some new alphabet X. Since we have imposed no condition on C, one may define it in such a way that two, or more, source symbols share the same codeword. It easy to observe that, in doing so, the decoded message may not be correct or even unique.

Example. Let's consider the code shown below. How should we de-

Source	**Code**
s_1	0
s_2	11
s_3	01
s_4	11

code the text 000111? We have two possibilities: either as $s_1s_1s_3s_2$ or as $s_1s_1s_3s_4$. As said before, the decoded message might not be unique. Also, unless context is given, there's no way to know which one is correct. $\square$

From the above example, we can conclude that a "good" code must encode each source symbols with a unique codeword. We call such codes *non-singular codes*.

- 1.2.1 - Uniquely decodable codes (UD).

One may think that non-singular codes are enough, but in most cases they are not. In fact, it might happen that a codeword is prefix of another, making the decode process tedious.

Example. Let's consider the following codes. How should we decode

Source	**Code**
s_1	0
s_2	1
s_3	01
s_4	11

the string 000111? Once again, we have many possibilities: for example $s_1s_1s_3s_2s_2$ or $s_1s_1s_3s_4$. Additionally, unless contex is given, we don't know which one is correct. □

We say that a code is UD if and only each sequence of codewords corresponds to at most one sequence of source symbols. From this definition, two questions follow:

- How we construct such codes?
- How can we check whether a given code is UD or not?

The next section focus on the second question, *Section 2-4* will be focused on the construction of UD codes.

- 1.2.2 - Sardinas-Patterson algorithm.

The *Sardinas-Patterson* algorithm provides a way to check whether a code is UD or not. Conceptually the algorithm, and the theorem it's built on, is based on the following remark: consider a string that is the concatenation of codewords. If we try to construct two distinct factorization, each word in of the factorization is either part of a word in the other factorization, or it starts with a prefix that is suffix of a word in the other factorization. Hence, a code is non-UD if it happens that a suffix is itself a codeword.

As stated before the algorithm is based on a theorem, given below.

Theorem 1.1 *Given C a code on an alphabet Σ, consider the sets $S_0, S_1, \ldots$ such that:*

- $S_0 = C$
- $S_i = \{\omega \in \Sigma^* \mid \exists\alpha \in S_0, \exists\beta \in S_{i-1} : \alpha = \beta\omega \vee \beta = \alpha\omega\}$

then necessary and sufficient condition for C to be a UD code is that, $\forall n > 0, S_0 \cap S_n = \varnothing$.

Therefore, the algorithm has three halt conditions:

- $\forall i > 0, S_0 \cap S_i \neq \varnothing$ the the code is not UD.
- $\forall i > 0, S_i = \varnothing$ then the code is UD.
- $\forall i, j > 0, S_i = S_j$ then the code is UD.

Example. Let's consider the following code: $C = \{a, c, ad, abb, bad, deb, bbcde\}$. Is it UD? Applying Sardinas-Patterson, step by step we get the following:

Iter 1. Let $\alpha_0 = ab, \alpha_1 = abb$ and $\beta_0 = \beta_1 = a$ then, $S_1 = \{d, bb\}$.

Iter 2. Let $\alpha_0 = deb, \alpha_1 = bbcde$ and $\beta_0 = d, \beta_1 = bb$ then, $S_2 = \{eb, cde\}$.

Iter 3. Let $\alpha_0 = c$ and $\beta = cde$ then, $S_3 = \{de\}$.

Iter 4. Let $\alpha_0 = deb$ and $\beta = de$ then, $S_4 = \{b\}$.

Iter 5. Let $\alpha_0 = bad$ and $\beta = b$ then, $S_5 = \{ad\}$.

S_0	S_1	S_2	S_3	S_4	S_5
a	d	eb			
c					
ad					
abb	bb	cde	de	b	ab
bad					
deb					
bbcde					

Table 1: Steps of Sardinas-Patterson for the code C.

From the above steps (summarised in *Table 1*), follows that the code is not UD since $S_0 \cap S_5 = \{ab\} \neq \varnothing$. □

Exercise 1.1 Apply the Sardinas-Patterson algorithm to the following codes:

- $C_0 = \{a, b, cd, abb, abcd\}$
- $C_1 = \{0, 01, 100, 11001, 01011\}$
- $C_2 = \{010, 0001, 0110, 11000, 00011, 00110, 11110, 101011\}$

- 1.3 - Average code length and Kraft inequality.

In the previous section, we've introduced prefix codes. The question now is, given two distinct prefix-free codes, which one is more convenient? A good choice would be the one that, on average, has the shortest length.

Definition (Average code length (ACL)) Let C be a code with a source alphabet $S = \{s_1, \dots, s_n\}$ and a code alphabet $X = \{x_1, \dots, x_m\}$. Let $\{c_1, \dots, c_n\}$ be the codewords with lengths $l_1, \dots, l_n$ respectively. And let $\{p_1, \dots, p_n\}$ be the probabilities for the source symbols. Then, we define the quantity

$$L_S(C) = \sum_{i=1}^{n} p_i l_i$$

as the average code length of the code C.

From what above, it makes sense to look for the UD code with the lowest average code length. That is, among all the UD codes for the same source and with the same code alphabet, the one with the lowest ACL. But how do we find such codes? A good starting point is the entropy of the source. Once again, thanks to Shannon, we have the following result.

Theorem (Shannon) *Let C be a UD code for a memoryless source S, whose probabilities are $\{p_1, \dots, p_n\}$, and alphabet code X with a size d. Then*

$$L_S(C) \geq \frac{H(S)}{\log_b d}$$

- 1.3.1 - The Kraft-McMillan inequality.

We have disscussed what prefix-free codes are, and what the average code length represents. We now provide a necessary and sufficient condition for the existence of a prefix code: the *Kraft-McMillan inequality.*

Theorem 1.2 (Kraft-McMillan) *Let $S = \{s_1, \dots, s_n\}$ be a source alphabet and $X = \{x_1, \dots, x_d\}$ a code alphabet. Let $l_1, \dots, l_n$ be a set of lengths. Then, necessary and sufficient condition for the existence of a prefix-free code C over the alphabet X with codewords lenghts $l_1, \dots, l_n$ is that*

$$\sum_{i=1}^{n} d^{-l_i} \leq 1$$

with d size of the code alphabet.

In other terms, if a set of lengths satisfies the inequality then, there exist at least a way to arrange the codewords into a prefix code.

- 1.3.2 - Optimal codes and the Shannon-Fano encoding.

The concept of average code length allows us to define a intresting class of codes: the *compact* (or optimal) codes. These are UD codes that have the lowest ACL.

A first attempt to achive such codes was firstly proposed by Shannon and *Robert Mario Fano*, who developed the so called Shannon-Fano encoding (1949).

Shannon-Fano encoding.

We will be considering the binary case. The encoding itself is very simple: we order the symbols in decreasing order, divide the symbols in two sets such that the sum of probabilities in each set is almost equal, encode the symbols in the first set with 0 and the other with 1. Lastly, repeat the procedure for each set recursively.

Example. Let's assume the alphabet $\{a, b, c, d, e\}$ whose probabilities are $\{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}\}$. Applying the steps above we get the codes shown in the table below. $\square$

Symbol	Encoding
a	0
b	10
c	110
d	1110
e	1111

One can prove that Shannon-Fano encoding is not optimal.

- 2 - Huffman encoding.

The first to solve the problem of creating optimal code was *D.A. Huffman*, who in [4] provides a method to create compact codes.

Before we actually analyze such method, it's of intrest to point out some properties that optimal codes must satisfy. Let S be a source with some probability distribution, let's also assume that $p_1 \geq p_2 \geq \ldots \geq p_n$. Let C be a compact prefix code for S such that c_i is the codeword associated to p_i. Then:

1. To reduce the expected code length, the shortes codewords are associated to the symbols with the higtest probability. This means that
$$p_i \geq p_j \implies |\, c_i \,| \leq |\, c_j \,|$$
If this was not the case, swapping the codewords would result in a code with a lower ACL.

2. The least probable symbols have codewords with the same length.

3. The longest codewords differ only by the last symbol.

- 2.1 - The algorithm.

We treat the binary case; the extension to the general case is immediate. Let
$$S = \begin{pmatrix} s_1 & s_2 & \cdots & s_n \\ p_1 & p_2 & \cdots & p_n \end{pmatrix}$$
be a source, such that the probabilities are sorted non-increasingly. Denote by $R(S)$ the *reduced soource*, obtained by replacing the two least common symbols from S, with one whose probabily is the sum of those of the replaced symbols. This means
$$R(S) = \begin{pmatrix} s_1 & s_2 & \cdots & (s_{n-1}, s_n) \\ p_1 & p_2 & \cdots & p_{n-1} + p_n \end{pmatrix}$$
Assume C_R to be a binary prefix code for $R(S)$, and let z be the codeword associated to the merged symbols (s_{n-1}, s_n). Then, a prefix code C for S can be obtained by C_R by assigning to the i-th symbol of S the i-th codeword in C_R, for $i \leq n-2$. The codeword for s_{n-1} and s_n are simply $z0$ and $z1$ respectively.

- 2.1.1 - The encoding.

Let ω be a string to encode, produced by some source S. To encode such string we have to:

1. Scan ω to get the probabilities of each symbol. Unless these are known a priori.
2. Sort the symbols non-increasingly according to their probabilities.
3. Build a (binary) tree where the leaves are the source symbols, and each node is the sum of the (two) lowest probabilities. Repeat the process until a root is formed.
4. Build the code by traversing the tree from the root to a leaf, and assigning a 0 to the left paths, a 1 to the right one.

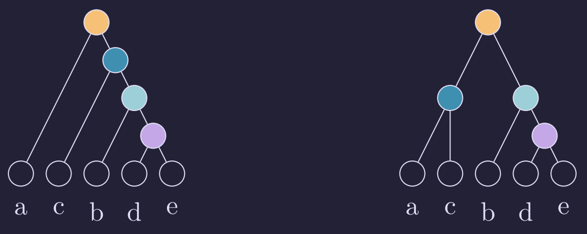
Example. To ease things out, the example skips the scanning step.

Let $S = \{a, b, c, d, e\}$ whose probabilities are $\left\{\frac{1}{3}, \frac{1}{6}, \frac{1}{3}, \frac{1}{12}, \frac{1}{12}\right\}$, respectively. Applying the steps above we have the following.

Step 1. Sorting the symbols according to the probabilities, we get the following order: a, c, b, d, e.

Step 2. To build the tree let's consider the two least probable symbols, in this case d and e. The sum of their probabilities will define a new node x in the tree (the ■ one in the figures below). The next node y (the ■ one) is given by x and the third least probable symbol b. At this point we can make two choices to define the node z: we can either merge a and c or merge c and y. Both will build a optimal code.

For the sake of this example, let's merge c and y, the other choice is shown in *Figure 1.b*. Lastly, we merge the last two symbols, leading to the tree in *Figure 1.a*.



(a) **(b)**

Figure 1: Huffman trees for the source S.

Step 3. Traversing the tree we get $C = \{0, 10, 110, 1110, 1111\}$ for tree in *Figure 1.a* or $C = \{00, 01, 10, 110, 111\}$ for the one in *Figure 1.b*.

$\square$

From the steps above, it's obvious that the slowest part of the encoding is given by the neccessity to scan the text twice.

- 2.1.2 - The decoding.

Let's assume we recieve a text encoded using Huffman, how do we decode it? We shall remark that to decode the text we need to know either the Huffman tree, or the source probabilities, which represent in both cases an overhead to the bare encoding. Assuming that such overhead is know, the decoding is immediate. In fact, we read the recieved text and traverse the tree accordingly until reaching a leaf. Repeating the process for the whole text.

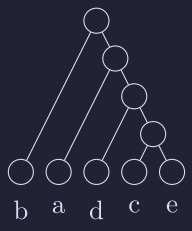
Example. Let $\omega = 00101111100001$ be a text encoded using Huffman. Let's assume that the tree used is the one in *Figure 1.b*. Starting from the root, we read a zero as first symbol of the encoded text, thus we consider the left subtree. Since we are not at a leaf we keep reading the encoded text. We read another zero, so once again we consider the left subtree. Since we reached a leaf we stop and decode 00 as a. We move back to the root and repeat the process. After few steps we decode $\omega = abedac$. $\square$

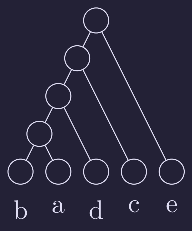
- 2.1.3 - Exercises.

Exercise 2.1 Apply Huffman encoding to the following strings.

- $\omega_0 = abbabccadeabf$
- $\omega_1 = \mathbf{010010001001010}$
- $\omega_2 = baabbbacabbdeffea$

Exercise 2.2 Determine which of the following is the Huffman tree for the source $S = \{a, b, c, d, e\}$ whose probabilities are, respectively, the following: $\left\{\frac{1}{4}, \frac{1}{2}, \frac{1}{16}, \frac{1}{8}, \frac{1}{16}\right\}$.





Exercise 2.3 Using the tree from *Exercise 2.2*, decode the following strings:

- $\omega_0 = 111011000100010$
- $\omega_1 = 1111001101101100111110$

- 2.1.4 - Adaptive Huffman.

As it was pointed out previously, the main issue with the classic Huffman approach, is the necessity to scan the text twice, which slows down the encoding. Also, to work properly we need to know the symbols probabilities in advace, which is not always the case. To solve both these issues, a new approach has been developed known as *Adaptive Huffman*.

The key point of this new approach is the fact that the tree is built and updated dynamically. Essentially we:

- Start with an initially empty tree, or one which only contains a special **Not Yet Transfered** node.
- For each symbol in the text:
  - If the symbol was already encoded, endode it with the existing code.
  - If it's a new symbol: encode the **NYT** node first, then encode the symbol itself and add the symbol to the tree as a leaf.
  - After encoding a symbol, update the tree: increment the frequency of the symbol and its ancestors. Reorganize the tree to mantain the Huffman properties.

One can prove that the adaptive approach has a better locality the the classic one.

- 2.2 - Canonical Huffman.

Let us consider some limitations of the Huffman encoding discussed so far. One of the main problems, is the fact that the encoder has to transmit alongside the text, the tree used for it: a not so trivial overhead. Additionally, the decoding phase is slow due to the necessity to traverse the tree for each symbol.

Let us observe that there exist codes, that are compact, that are not produced by the Huffman algorithm. We now introduce the so called *Canonical Huffman encoding*, which is a variant of Huffman, that solves the issues above by requiring the encoder to transmit the lengthts of the codewords.

Such new encoding, comes in handy when the source alphabet is large, and a fast decoding is mandatory.

- 2.2.1 - The encoding.

Let C be the usual Huffman encoding for some source. To obtain the canonical version from it, we need to:

1. Compute the length of the codewords in C, with respect to each symbol in the source alphabet.
2. Construct the num array which stores at each entry num[l], how many symbol have length l.
3. Construct the symb array which stores at each entry symb[l], the symbols having length l.
4. Construct the fc array that stores at each entry fc[l], the first codeword off all symbols having codeword length l. Such construction is done via the following snippet.

   ```
   fc[MAX] = 0 // MAX = 3 in the example
   for(l = MAX - 1; l >= 1; l--){
         fc[l] = (fc[l + 1] + num[l + 1]) >> 1;
   }
   ```

5. Assign consecutive codewords to the symbols in symb[l] starting from symb[l].

Note. The snippet shown uses a reverse approach, which usually produces codewords that differ from those built via a forward approach.

Example. Let C be the Huffman code obtained by the tree in *Figure 1.b*.

1. Let l_i be the length of the i-th codeword, we have: $l_0 = l_1 = l_2 = 2, l_3 = l_4 = 3$.
2. Computing num we have: num[1] = 0, num[2] = 3 and num[3] = 2.
3. Computing symb we get: symb[1] = null, symb[2, 0] = a, symb[2, 1] = c, symb [2, 2] = b, symb[3, 0] = d and symb[3, 1] = e. To ease things out, we use a second index to move within a list when needed.
4. Computing fc we get: fc[1] = 2 (this value will be used in the decoding process), fc[2] = 1, fc[3] = 0.
5. Let's observe that symb[1] = null, meaning there're no codewords to assign. Considering symb[2] we assign 01 to a, 10 to c and 11 to b. For what regards symb[3] we have that 000 is assigned to d and 001 to e.

□

We assume a 1-based indexing for the arrays and a 0-based one for the lists.

- 2.2.2 - The decoding.

Let's assume ω to be a a encoded text via the Canonical Huffman encoding. To decode such text, we simply apply the following procedure.

```
char v = next_bit();
// Assume next_bit() as a
      primitive
int l = 1;
while(v < fc[l]){
     v = 2*v + next_bit();
     l++;
}
return symb[l, v - fc[l]];
```

Example. Let $\omega = 01100000011011$ be a text encoded using the Canonical Huffman encoding, and let symb and fc be those computed in the previous example. Applying the snippet above, one can verify that $\omega = acdecb$. □

- 2.2.3 - Exercises.

Exercise 2.4 Apply the canonical Huffman encoding to the codes obtained in *Exercise 2.1*.

Exercise 2.5 Rewrite the procedure to construct the fc array using a forward approach.

- 3 - Aritmethic coding.

Let us recall that, given C a code, $L_S(C)$ represents the average code length of the codewords in C.

From it, we can define two other important quantities: the efficiency and the redundancy of the code. Formally, we define the efficiency as

$$\vartheta(C) = \frac{H_n(S)}{L_S(C)}$$

which, from Shannon's theorem, is a value in the range $[0, 1]$. We define the redundancy of the code as

$$\rho(S) = 1 - \vartheta(C).$$

When considering Huffman, one can prove that its redundancy approaches 1 when the most frequent symbol probability approaches 1.

The question then is: can we do better than Huffman? The answer is yes, but we have to use a completely different approach: instead of assigning a codeword to each symbol, we assign one to the whole text. The method we are about to present is known as *aritmethic coding (AC)*.

- 3.1 - The encoding.

The AC encoding is pretty straightforward, as summarized in the steps below. Let ω be the text to encode and let's assume the alphabet, alongside the probabities, is know. Then:

1. We start by initializing the range $[0, 1]$.
2. For each symbol in ω:
   (a) Divide the current range into n intervals (n the size of the alphabet), each proportional to the probabities of the symbols.
   (b) Select as current subinteval the one corresponding to the symbol that is begin analyzed.
3. Output the binary representation[1] of the lowerbound of the last subinteval.

Example. Let $\omega = abac$ and let $S = \{a, b, c\}$ with probabilities $\left\{\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right\}$.
By applying the above steps, we get:

Step 1. We initialize the range $[0, 1]$.

Step 2. Let us consider each iteraction, each step can be observed in *Figure 2*. We read "a" thus, we select the range $[0, 1/2]$. We proceed to updated the ranges, and repeat the process until the end of the text.

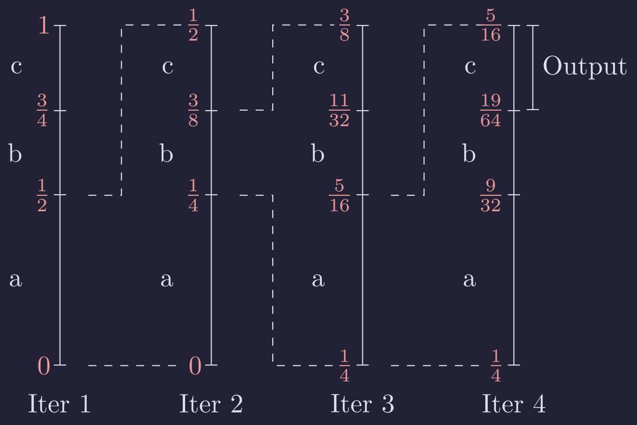


Figure 2: Step by step encoding of $\omega = abac$ via the AC.

Step 3. We return the mid value of the last range.

□

[1] We assume the reader knows how fractional numbers are represented in binary.

Algorithmically speaking, we simply apply the following code.

```c
double AC_encode(char* S, double* cumFun, double* Prob){
    double high = 1;
    double low = 0;
    int i = 1;
    double range;
    while (i <= n)  {
        range = high - low;
        low = low + range * cumFun[S[i]];
        high = low + range * (cumFun[S[i]] + Prob[S[i]]);
    }
    return high - low;
}
```

Figure 3: C implementation of the AC encoding.

How many bits do we need to represent the encoding? From the code in *Figure 3*, we can deduce that the size of the final range is given by

$$range_n = \prod_{i=1}^{n} Prob[S[i]]$$

from which we can derive that the outpus size is indipendent of the string symbols permutation. Then by the above remark, and the following results, we can deduce that we need $l_n + \frac{s_n}{2}$ bits, where l_n and s_n are, respectively, the last two values of low and high in the code above.

Lemma. *Let* $x = 0.b_1 b_2 \dots$ *be the binary representation of a real number. Then, it's truncation to the first* d *bits is also a real number,*

Proof Let's observe that

$$x - trunc_d(x) = \sum_{i=1}^{\infty} b_{d+i} 2^{-(d+i)} \leq \sum_{i=1}^{\infty} 1 * 2^{-(d+i)} = 2^{-d} \sum_{i=1}^{\infty} 2^{-i} = 2^{-d}$$

Therefore,

$$x - trunc_d(x) \leq 2^{-d} \iff x - d^{-d} \leq trunc_d(x)$$

■

Corollary. Given $x \in [low, high]$ its truncation to the first $\lceil \log \frac{2}{high-low} \rceil$ falls within the range.

Proof It follows by the above *Lemma* putting $d = \lceil \log \frac{2}{high-low} \rceil$. ■

- 3.2 - The decoding.

Deconding a text compressed via the AC encoding, is straightforward as its encoding. In fact, the decoding proceeds similarly to the encoding. We just

1. Initialize the range $[0, 1]$.
2. We divide the subrange according to the probabilities, as done for the encoding.
3. At each step we select as current interval the one where the encoded text fall in to.

Algorithmically speaking we proceed applying the procedure in *Figure 4*.

```
char* AC_decode (int encodedText, int textLength, double*
        Prob, double* cumFun){
    double high = 1;
    double low = 0;
    int  i = 1;
    double range;
    char* decoded = malloc(sizeof(char * textLength));
    while (i <= n) {
        // divide the current range into subranges
             according to the probabilities
        // return the symbol x associated to the current
             range
        range = high - low;
        decoded[i -1] = x;
        low = low + range * cumFun[x];
        high = low + range * (cumFun[x] + Prob[x]);
    }
    return decoded;
}
```

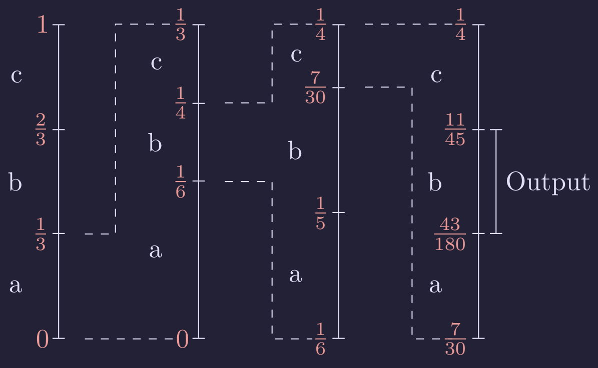
Figure 4: C like implementation of the AC decoding.

- 3.3 - The adaptive version.

It's easy to observe from *Figure 3*, that the necessity to know the probabilities a priori, slows down the encoding. The question then is: can we improve the AC encoding? The answer is yes, by assuming an initial equality in the probabilities. Briefly, before any symbol is read, we assume that each of the alphabet symbol has the same probability to appear within the text. Then, at each step, we simply apply the same procedure in *Figure 3* and then update the probabilities.

What about the decoding? We can apply the same logic, with a little differce: instead of updating the probabilities after a symbols is read, we update them, each time a symbol is decoded by applying the procedure in *Figure 4*.

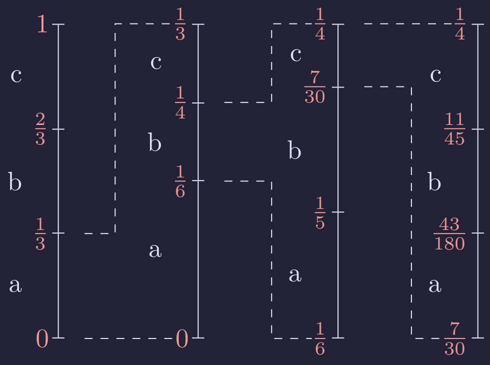
Example. Let $\omega = abcb$ be the text to encode. Before any character is read, we have the following probabilities: $p_a = p_b = p_c = \frac{1}{3}$. Once the first character (a) is read we select as current range the $[0, \frac{1}{3}]$ then we update the probabilities as $p_a = \frac{1}{2}, p_b = p_c = \frac{1}{4}$. We repeat the same process until the end of the text, and encode ω as $[\frac{43}{180}, \frac{11}{45}]$.



Read "a" Read "b" Read "c" Read "b"

Figure 5: Step by step encoding of $\omega = abcb$ via the adaptive AC.

Let's now decode 0.2416, with $n = 4$. Again we assume equality in the probabilities, and thus, we decode (a) and update the probabilities. Repeating the process, we decode the text *abcb*. □



Decode "a" Decode "b" Decode "c" Decode "b"

Figure 6: Step by step decoding of 0.2416 via the adaptive AC.

- 3.4 - Exercises.

Exercise 3.1 Apply the AC encoding to the following strings, then compare the outputs with the one obtained using the adaptive version.

1. $\omega_0 = acabba$
2. $\omega_1 = aabbca$

Exercise 3.2 Decode the following texts, encoded using the adaptive AC encoding. Assume the alphabet to be $\{a, b, c, d\}$.

1. $s_0 = 0.456731$, with $n = 4$.
2. $s_1 = 0.123$, with $n = 10$.
3. $s_2 = 0.549201$, with $n = 5$.

- 4 - Integers encoding.

What happens when the source to encode uses positive integers as symbols? How can we find a universal representaion? Meaning, how can we find a code that is prefix free, and whose ACL is $\mathcal{O}(\ln x)$, for any x.

Can we still use the codes seen so far, or do we need new ones? The answer is we can do both depending on the case. For instance, if we know the distribution of such integers and the range is relatively small, we can still use Huffman. If not, we need to use some new codes: those we are about to discuss.

Remark. We can relax the constrain on the positiveness, by mapping any positive integers x to $2x + 1$ and any negative integers y to $-2y$.

- 4.1 - Simple approaches.

One of the simplest approch is to use the binary reprentation of the given integers, but there's an issue: such reprentation is not prefix free.

A more valid, but still not reasonable on its own, is the so called *unary encoding*. Briefly, given x an integer we encode it as a sequence of $x - 1$ 0s followed by a 1. It obvious that such reprentation is indeed prefix free, but it's not reasonable – the higher the integer, the higher is the number of bits needed for its reprentation.

Clearly we needs something else. Even though there's plenty of codes to encode integers, the following sections focus on the *Elias codes* and the *Fibonaci codes*.

- 4.2 - Elias codes.

When we talk about Elias code we refer, generally, to two codes (the *gamma* code and the *delta* code), proposed by *Peter Elias* in [3].

Before we start talking about the codes, let us define some notation useful to understand what follows. Let x be an Integer, and let $B(x)$ be its binary representation. We define $|B(x)|$ as the number of bits needed to represent x.

Note. One can also think of $|B(x)|$ as the length of $B(x)$

- 4.2.1 - Gamma code.

Let x be a positve Integer. Its gamma code ($\gamma(x)$) is a binary sequence composed by:

- the unary encoding of $\mid B(x) \mid$
- the binary representaion of x minus the most significant bit.

Example. If $x = 11$, then its gamma encoding is:

$$\gamma(x) = 0001011$$

where 0001 is the unary encoding of $\mid B(x) \mid$ and 011 the binary representaion of x minus the most significant bit. □

The decoding is trivial, so we skip it. It's easy to prove that with this kind of code, we need at most $2\lfloor \log x \rfloor + 1$ bits. Also, most importantly, this encoding is reasonable when $p(x) = 2x^{-2}$.

- 4.2.2 - Delta code.

Let x be a positive Integer, its delta code ($\delta(x)$) is a binary sequence composed by:

- the delta code of $| \cdot | B(x)$
- the binary representation of x excluding the most significant bit

Example. Let $x = 11$, then its delta code is:

$$\delta(x) = 00100011$$

where 00100 is the gamma encoding of $| \cdot | B(x)$ and 011 the binary encoding of x, most significant bit excluded. □

As before, the decoding is trivial, so we skip it. Once can prove that the code uses at most $1 + \log x + 2 \log \log x$ bits. Also, and most importantly, gamma codes are reasonable when $p(x) = \frac{1}{2x(\log x)^2}$.

- 4.3 - Fibonacci code.

Before we present Fibonacci codes, let us recall that Fibonacci numbers are defined by the recurrence

$$F_n = \begin{cases} 0, & n = 0; \\ 1, & n = 1; \\ F_{n-1} + F_{n-2}, n \geq 2 \end{cases}$$

Let us also introduce a foundamental result for the existence of Fibonacci codes.

Theorem (Zackendorf) *Any given positive integer can be uniquely wrote as the sum of non consecutive Fibonacci numbers.*

Using the above theorem, we can constuct a code that efficiently encodes integers. Let n be a positive integer, we encode it as follow:

1. Find the greatest Fibonacci number lower or equal to n.
2. Say it was the i-th Fibonacci number, subtract it from n and keep trace of the remainder. Set the (i-1)-th bit to 1. (The left most bit has index 0).
3. Reapet the above steps by substituting n with the remainder, until it's 0.
4. Append an addiotional 1 to the right of the codeword.

Example. Let $n = 73$, it's easy to se that it can be wrote as $F_{10}+F_7+F_5$. From which by the above steps, we get that n is encode as 0001010011.□

For what regards the decoding, we simply remove the additional 1 at the right of the codeword, and starting from the left side we replace the i-th 1 with the (i + 1)-th Fibonacci number, then sum everything.

Example. Let us decode 0001010011. As said, we remove the extra bit at the right, from which we get 000101001. Then, we replace the i-th 1 with the (i + 1)-th Fibonacci number. Thus, we get $F_{4+1} + F_{6+1} + F_{9+1}$, which basically is $F_5 + F_7 + F_{10} = 73$. □

- 4.4 - Exercises.

Exercise 4.1 Apply the delta encoding and Fibonacci encoding to the following integers, then compare the codewords obtained for each.

1. 1230
2. 15
3. 901

- 5 - Compression optimization.

So far, we've discussed of a few compression techniques assuming that the input of such compressors was “optimal”[2]. But what if, this was not the case? Can we preprocess the string in such a way that, once it reaches the compressor, this is optimal? The answer is yes and this section introduced a few of such optimization.

- 5.1 - Burrows-Wheeler Transform.

The Burrows-Wheeler Transform (BWT), introduced by *M. Burrows* and *D. Wheeler* in [2], is a simple, yet efficient tool to enhance the compressibility of a given string.

Let ω be a string we want to compress, thus preprocess using BWT. First step of the transform, is to compute all the cyclic rotations of ω. Then, we sort them lexicographically[3] and lastly, we return the last column. and the index (starting from 0) of the inital string.

Note. By last column we mean the concatenation of the last symbol of each rotation.

Example. Let $\omega = aleph$. Its cyclic rotations, already sorted, are

$$\begin{gathered} aleph \\ ephal \\ halep \\ lepha \\ phale \end{gathered}$$

Thus, we return the pair $(hlpae, 0)$. □

Let us point out some interesting properties of the BWT.

1. $\forall i \neq I$, where I is the index of the inital string, we have that $F[i]$ follows $L[i]$ in ω.
2. $F[I]$ is the first symbol of ω.
3. For any character x, the i-th occurence of x in F, corresponds to the i-th occurence of x in L.

- 5.1.1 - Inverse of the BWT.

Since the BWT is a transform, we want it to be reversible. As obvious, the BWT is indeed reversible, and actually it has two distinct, yet equivalent, inverse transform: the FL-mapping and the FL-mapping.

[2]In this context by “optimal”, we mean that the string has the lowest possible entropy.

[3]For the sake of our disscussion, assume that the lexicographic order is the usual alphabetical one. This is, in general, not true.

Let us use the BWT of the previous example as input to the inverse transforms.

FL-mapping.

Let $L = BWT(aleph)$ and $I = 0$ the index of the original string. We construct F by lexicographically sorting L. Let's define the permutation that maps the symbols in F to those in L as

$$\tau = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 3 & 4 & 0 & 1 & 2 \end{pmatrix}.$$

We recover ω by computing $\omega[i] = F[\tau^i[I]]$, for $i = 0, 1, \dots, 4$. In fact, we have: $\omega[0] = F[I] = a, \omega[1] = F[\tau[0]] = F[3] = l$, and so on.

LF-mapping.

Let $L = BWT(aleph)$ and $I =$ the index of the original string. We construct F by lexicographically sorting L. Let's define the permutation that maps the symbols in L to those in F as

$$\sigma = \begin{pmatrix} 3 & 4 & 0 & 1 & 2 \\ 0 & 1 & 2 & 3 & 4 \end{pmatrix}.$$

We recover ω by computing $\omega[n-1-i] = L[\sigma^i[I]]$, for $i = 0, 1, \dots, 4$. In fact, we have: $\omega[4] = L[I] = h, \omega[3] = L[\sigma[0]] = L[2] = p$ and so on.

- 5.2 - Analysis of the BWT.

From the discussion of the BWT, a question might arise naturally: how much does the compression improve by preprocessing the string using the BWT? The question to this question can be found in [7]. Here Manzini analyzes the compression of BWT-based algorithm, in terms of their *empirical entropy*[4]. Even though the paper referes just to the 0-th order empirical entropy, defined as follow,

$$H_0(s) = -\sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n_c}{n}$$

where n_c is the number of occurences of c in s; one could generalize the idea to a k-th order empirical entropy as follow

$$H_k(s) = \frac{1}{|\, s \,|} \sum_{w \in \Sigma^k} |\, w_s \,| H_0(w_s) \tag{2}$$

where w_s is the concatenation of symbols that follow w in s.

In the paper Manzini provides also the following results.

Theorem (Manzini) *For any string s over some alphabet Σ and for any $k \geq 0$, it holds that*

$$BW_0 = \leq 8|\, s\,|H_k(s) + \left(\mu + \frac{2}{25}\right)|\, s\,| + h^k(2h\log h + 9)$$

where $h = |\, \Sigma\,|$ and $\mu = 1$.

Let us point out that $BW_0 = Order_0 + MTF + BWT$, where $Order_0$ is some 0-order compressor.

Theorem. *For any string s over some alphabet Σ and for any $k \geq 0$, there exists some constant g_k, such that it holds*

$$BW_0 + RL \leq (5 + 3\mu)|\, s\,|H^*_k(s) + g_k$$

where RL is the run length encoding of the string.

To conclude this section, let us observe that the bottleneck of the BWT is given by the sorting of the cyclic rotations. We should show, in the following section, that the problem can be reduced to the sorting of suffixes.

- 5.2.1 - Efficient computation of the BWT.

Let ω be the string of which we want to compute the BWT, and let $\tilde{\omega} = \omega\$$, where $\$ \notin \Sigma$. If we compute the BWT of $\tilde{\omega}$, we can just focus on sorting the suffixes rather then the whole cyclic rotations, as the BWT of ω would require.

A question arise naturally: how do we compute and sort these suffixes? Through the years many algorithms have been proposed (see [1, 5, 6]), all of them with a thing in common – a special data structure – the suffix array (SA). If $T = T[1, n]$ is a text, we define T_i to be the suffix $T[i, n], \forall i \in [1, n]$. We denote by $Suff(T) = \{T_i \mid i \in [1, n]\}$. The SA is a sorted array for $Suff(T)$. Using this structure, the complexity of the BWT is reduced to the complexity of algorithm used to build the SA.

The DC3 algorithm.

ToDo. Wait for the lecturer to explain it.

[4]Due to the way it's defined, see (2), empirical entropy is perfect for a worst-case analysis.

- 6 - Dictionary based compressors.

ToDo. Start it.

References.

[1] Uwe Baier. “Linear–time Suffix Sorting – A New Approach for Suffix Array Construction”. In: *CPM* 23 (2016).

[2] M. Burrows and D. J. Wheeler. *A Block-sorting Lossless Data Compression Algorithm*. Tech. rep. Research Report 124. Palo Alto, California: Digital Equipment Corporation, Systems Research Center, May 1994.

[3] Peter Elias. “Universal Codeword Sets and Representations of the Integers”. In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 194–203.

[4] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[5] Sanders Kärkkäinen and Burkhardt. “Linear work suffix array construction”. In: *Journal of the ACM* 53.6 (2006), pp. 918–936.

[6] Ko and Aluru. “Space efficient linear time construction of suffix array”. In: *IEEE Transactions on Computers* 3.2 (2011), pp. 1471–1484.

[7] G Manzini. “An analysis of the Burrows-Wheeler Transform”. In: *Journal of the ACM* 48.3 (2001), pp. 407–430.

[8] Claude E. Shannon. “A Mathematical Theory of Communication”. In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423.

Acronyms.

AC aritmethic coding. 14–19

ACL average code length. 7–9, 19

BWT Burrows-Wheeler Transform. 22

SA suffix array. 24

UD uniquely decodable codes. 5–8