

Information theory notes

Riccardo Lo Iacono

Unipa

Contents.

1	Basics of information theory	1
1.1	Shannon's Entropy	1
1.2	Encoding of a source	4
1.3	Average code length and Kraft inequality	7
2	Huffman encoding	9
2.1	The algorithm	9
2.2	Canonical Huffman	12

- 1 - Basics of information theory.

Information theory, alongside data compression, have a key role in modern CS. The former defines a set of theoretical tools, usefull 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 can we define it formally? And also, is there a way to measure information?

The first to answer these questions was *Claude Shannon*, by many considered the father of modern information theory. In [2] Shannon analyzes various communication systems: from the discrete noiseless one to the continous noisy one. A general structure of these systems is shown in *Figure 1*.

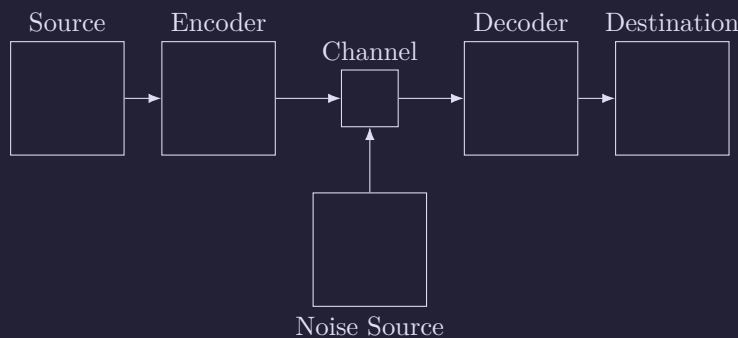


Figure 1: Diagram of a general communication system.

- The *Source* or, more precisely, the *Information Source* refers 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.

Section 1 - Basics of information theory.

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:

$$H = -K \sum_{i=1}^{n=|\Sigma|} p_i \log_b p_i. \quad (1)$$

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.

- 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 $H(S) = H(p_1, \dots, p_n)$, then H is continuos 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 $H(\tau) = H_n(p_1, \dots, p_n)$. Meaning that, H must not care of the order of the probabilities.
4. If $A(n) = H_n\left(\frac{1}{n}, \dots, \frac{1}{n}\right)$, 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

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

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

Section 1 - Basics of information theory.

Theorem (Theorem 2, [2]) *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:

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

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 $A(2^k) \leq A(n^r) < A(2^{k+1})$, 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

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

Section 1 - Basics of information theory.

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 .

■

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

- 1.2 - Encoding of a source 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 receiving 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 $C : S \rightarrow X$ 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 specific type of *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 not imposed any condition on C , one may define it in such a way that two, or more, source symbols share the same codeword. It is 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 the context is given, there's no way to know which one is correct.

Section 1 - Basics of information theory.

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 context 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, we’ll then answer the first.

- 1.2.2 - Sardinas-Patterson algorithm.

The *Sardinas-Patterson* algorithm provides a way to check whether a code C , is UD or not. Conceptually the algorithm, and the theorem from which it derives, are 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, \dots such that:

- $S_0 = C$

Section 1 - Basics of information theory.

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

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

Therefore, the algorithm has three halt conditions:

- $\forall i > 0, S_0 \cap S_i \neq \emptyset$ then the code is not UD.
- $\forall i > 0, S_i = \emptyset$ 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, $S_5 = \{ad\}$.

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

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 .

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

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

Section 1 - Basics of information theory.

- 1.2.3 - Prefix codes.

Let's look back at the example of *Section 1.2.1*. Given the codewords $\{0, 1, 01, 11\}$ for the source symbol s_1, s_2, s_3, s_4 respectively, how should we decode the string 000111? As said before, unless some context is given, we can't be sure.

Then what's the issue? Essentially, even though to each source symbol is assigned a distinct codeword, these are not prefix-free, meaning that some of the codewords are prefix of others (eg. the codeword for s_1 is prefix of the codeword for s_3). From what just stated, a natural solution to the problem is to define codes that are prefix-free. One can also prove that, prefix-free (or simply prefix) codes are also instantaneous (each source symbol can be decoded without reading further in the string).

Theorem. Let C be a prefix-free code, then C is UD.

Proof It follows from *Theorem 1.1*. ■

- 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

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

Section 1 - Basics of information theory.

- 1.3.1 - The Kraft-McMillan inequality.

We have discussed 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 us consider a source alphabet $S = \{s_1, \dots, s_n\}$ and a code alphabet $X = \{x_1, \dots, x_d\}$. 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 lengths 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. A proof of this theorem is beyond the scope of this [notes](#).

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

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

A first attempt to achieve 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 following.

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 [1] provides a method to create compact codes.

Before we actually analyze such method, it's of interest to point out some properties that optimal codes must satisfy. Let S be a source with some probability distribution, let's also assume that $p_1 \geq p_2 \geq \dots \geq p_n$. 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 shortest codewords are associated to the symbols with the highest 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 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 source*, obtained by replacing the two least common symbols from S , with one whose probability 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.

Section 2 - Huffman encoding.

- 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 $\{\frac{1}{3}, \frac{1}{6}, \frac{1}{3}, \frac{1}{12}, \frac{1}{12}\}$, 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 \blacksquare one in the figures below). The next node y (the \blacksquare 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 2.b*. Lastly, we merge the last two symbols, leading to the tree in *Figure 2.a*.

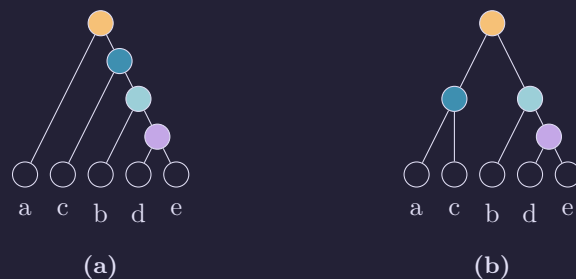


Figure 2: Huffman trees for the source S .

Section 2 - Huffman encoding.

Step 3. Traversing the tree we get $C = \{0, 10, 110, 1110, 1111\}$ for tree in *Figure 2.a* or $C = \{00, 01, 10, 110, 111\}$ for the one in *Figure 2.b*.

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

- 2.1.2 - The decoding.

Let's assume we receive a text encoded using Huffman, how do we decode it? We 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 known, the decoding is immediate. In fact, we read the received text and traverse the tree accordingly until reaching a leaf. Repeating the process for the whole text.

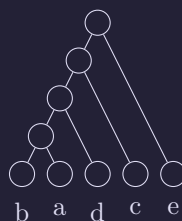
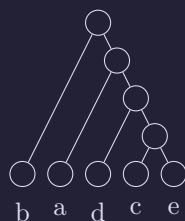
Example. Let $\omega = 00101111100001$ be a text encoded using Huffman. Let's assume that the tree used is the one in *Figure 2.b*. Starting from the root, we read a zero as first symbol of the encoded text, thus we consider the left subtree. Since we are not at a leaf we keep reading the encoded text. 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 $\omega = abedac$.

- 2.1.3 - Exercises.

Exercise 2.1 Apply Huffman encoding to the following strings.

- $\omega_0 = abbabccadeabf$
- $\omega_1 = 010010001001010$
- $\omega_2 = baabbacabbdefea$

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: $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{16}, \frac{1}{8}, \frac{1}{16}\}$.



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

Section 2 - Huffman encoding.

- $\omega_0 = 111011000100010$
- $\omega_1 = 111100110110110011110$

- 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 advance, 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 Transferred** node.
- For each symbol in the text:
 - If the symbol was already encoded, encode 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 maintain the Huffman properties.

One can prove that the adaptive approach has a better locality than 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 just transmit the length of the codewords.

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

Section 2 - Huffman encoding.

- 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;  
}  
└
```

┐

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

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

Example. Let C be the Huffman code obtained by the tree in *Figure 2.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.

Section 2 - Huffman encoding.

- 2.2.2 - The decoding.

Let's assume ω to be 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` using a forward approach.

Section 2 - REFERENCES.

References.

- [1] David A. Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceeding of I.R.E.* (1952), pp. 1092–1102.
- [2] Claude Shannon. “A mathematical theory of communication”. In: *Bell System Technical Journal* 27 (1948), pp. 397–423, 623–656.

Section 2 - Acronyms.

Acronyms.

ACL average code length. 7–9

UD uniquely decodable codes. 5–8