Notes on Information Theory

Riccardo Lo Iacono

September 26, 2025

Contents.

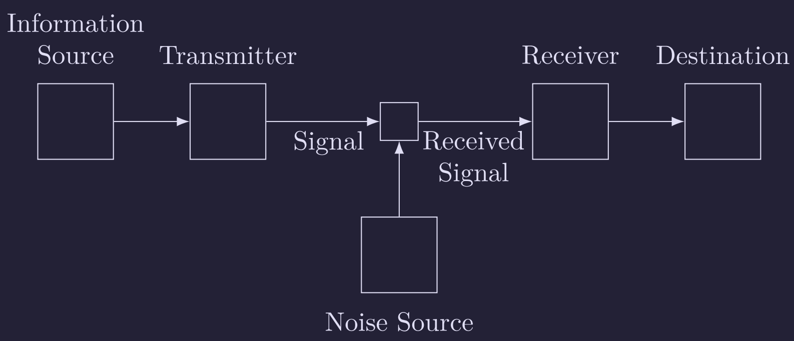
- 1 - Basics of information theory.

Information theory, alongside data compression, plays a key role in modern computer science. The former provides theoretical tools useful, among other things, for understanding the limits of computability; the latter enables the reduction of space requirements (in terms of bits) without loss of information.

- 1.1 - Shannon's Entropy.

When discussing information, we all have a general notion of what it represents, but how is it defined formally? Is there a way to quantify information?

The first to answer these questions was *Claude Shannon*, widely considered the father of information theory. In [10], Shannon analyzes various communication systems: from the discrete noiseless one to the continuous noisy one. A general structure of such systems is shown below.



Here:

- The *Source*, or more precisely, the *Information Source*, refers to some entity (a human, a computer, etc.) that produces messages.
- The *Transmitter* encodes the messages coming from the source and transmits them through the channel.
- The *Receiver* has the opposite role of the transmitter.
- The *Destination* is the entity to which the messages are intended.

Remark. We note that, henceforth, we refer to the noiseless source. We also assume that the source is memoryless; that is, each symbol produced is independent of the previous one.

Let S be a source of information. Let Σ be the alphabet of symbols used by the source, and for each symbol let $p_i = \Pr(S = \sigma_i)$ at any given time. We seek a function H, if it exists, that quantifies the uncertainty associated with S.

One can prove (see [10, Appendix 2]) that the only form that H can take is the following:

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

Here, K is a positive constant, and b is usually 2.

We define $H(S)$ to be the *entropy* of S. Formally, entropy measures the average amount of information contained in each symbol of the source output. Thus, the information associated with an event[1]can be defined as the reduction in uncertainty once the outcome is observed.

- 1.2 - Encoding of a source.

Let's now focus our attention onto the source itself. In general, the alphabets they use may not be suitable for transmission for various reasons. For this reason, an *encoder* is used to convert the source alphabet into 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 the *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 C be a generic mapping from a source S to some new alphabet X. Since we have imposed no conditions 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

Consider the following alphabet: $\Sigma = \{s_1, s_2, s_3, s_4\}$. Assume the code $C = \{0, 11, 01, 11\}$ was defined for Σ. How should we decode the text 000111? There are two possible interpretations: either as $s_1 s_1 s_3 s_2$ or as $s_1 s_1 s_3 s_4$. As noted above, the decoded message may not be unique. Moreover, without additional context, there is no way to determine which interpretation is correct.

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

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

- 1.2.1 - Uniquely decodable codes (UD).

One might think that non-singular codes are sufficient, but in most they don't. In fact, it can happen that a codeword is a prefix of another, making decoding ambiguous or tedious.

Example

Consider the following code: $C = \{0, 1, 01, 11\}$. Assume the alphabet of the previous example. How should we decode the string 000111? Again, multiple decodings are possible: for example $s_1 s_1 s_3 s_2 s_2$ or $s_1 s_1 s_3 s_4$. The correct decoding cannot be determined unless context is given.

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

- How do we construct such codes?
- How can we check whether a given code is UD?

The next section focuses on the second question, while *Sections 2-4* focus 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 its underlying theorem are based on the following remark: consider a string that is the concatenation of codewords. If we try to construct two distinct factorizations, each word in one of the factorizations is either part of a word in the other factorization, or it starts with a prefix that is a suffix of a word in the other factorization. Hence, a code is non-UD if 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:*

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

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

Example

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:

Iteration 1 Let $\beta = a$. If we let $\omega = d$, we have $\alpha = \beta\omega = ad \in S_0$; hence, $\omega = d \in S_1$. By the same logic we $bb \in S_1$.

Iteration 2 Let $\beta = d$. If we let $\omega = eb$, we get $\alpha = deb \in S_0$; thus, $eb \in S_2$. With a similar reasoning we get $cde \in S_2$.

Iteration 3 Let $\alpha = c$. If we let $\omega = de$, it derives that $\beta = cde \in S_2$; therefore, $de \in S_3$.

Iteration 4 Let $\beta = de$. If we let $\omega = b$, we get $\alpha = deb \in S_0$; hence, $b \in S_4$.

Iteration 5 Let $\beta = b$. If we let $\omega = bcde$, we have $\alpha = bbcde \in S_0$; thus, $bcde \in S_5$. By the same logic $ad \in S_5$

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

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

Table 1: Steps of Sardinas-Patterson for the code C. Note that each ω has been added to the row of the value use for α.

- 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 symbols s_1, s_2, s_3, s_4 respectively, how should we decode the string 000111? As noted before, unless some context is given, we cannot be certain.

Then, what is the issue? Essentially, even though each source symbol is assigned a distinct codeword, these codewords are not prefix-free, meaning that some are prefixes of others (e.g., the codeword for s_1 is a prefix of the codeword for s_3). From this observation, a natural solution is to define codes that are prefix-free. One can also prove that prefix-free (or simply prefix) codes are also instantaneous; that is, each codeword can be immidiately decoded without ambiguity.

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

Proof It follows directly from *Theorem 1.1.* □

- 1.3 - Average code length and Kraft inequality.

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

Definition (Average code length (ACL)) Let C be a code with source alphabet $S = \{s_1, \ldots, s_n\}$ and code alphabet $X = \{x_1, \ldots, x_m\}$. Let $\{c_1, \ldots, c_n\}$ be the codewords with lengths $l_1, \ldots, l_n$, respectively. Let $\{p_1, \ldots, p_n\}$ be the probabilities of 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 the above, it makes sense to look for the UD code with the lowest average code length. That is, among all 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 $\{p_1, \ldots, p_n\}$, and code alphabet X of size d. Then*

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

- 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 $S = \{s_1, \ldots, s_n\}$ be a source alphabet and $X = \{x_1, \ldots, x_d\}$ a code alphabet. Let $l_1, \ldots, l_n$ be a set of lengths. Then, a necessary and sufficient condition for the existence of a prefix-free code C over the alphabet X with codeword lengths $l_1, \ldots, l_n$ is that*

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

where d is the size of the code alphabet.

In other words, if a set of lengths satisfies the inequality, then there exists at least one 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 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 proposed by Shannon and *Robert Mario Fano*, who developed the so-called Shannon-Fano encoding.

Shannon-Fano encoding.

We will consider the binary case. The encoding itself is very simple: we order the symbols in decreasing order, divide the symbols into 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, and finally repeat the procedure recursively for each set.

Example

Consider the alphabet $\Sigma = \{a, b, c, d\}$ and let $p_a = 1/2, p_b = 1/8, p_c = 1/4$ and $p_d = 1/8$ Applying the steps above, we get the codes shown in the table below. Here, each color is used to show a different iteraction.

Symbol	Codeword
a	0
c	10
b	110
d	111

One can prove that Shannon-Fano encoding is not optimal. We present it just for hystorical reasons.

- 2 - Huffman encoding.

The first to solve the problem of creating optimal codes was *D. A. Huffman*, who in [5] provided a method to construct compact codes.

Before analyzing this method, it is of interest to point out some properties that optimal codes must satisfy. Let S be a source with probability distribution $\{p_1, p_2, \dots, p_n\}$, ordered such that $p_1 \geq p_2 \geq \ldots \geq p_n$. Let C be a compact prefix code for S, and let c_i denote the codeword associated with symbol of probability p_i. Then:

1. To reduce the expected code length, the shortest codewords are associated with the most probable symbols. That is,

   $$p_i \geq p_j \implies |c_i| \leq |c_j|.$$

   If this were not the case, swapping the codewords would result in a code with a lower average code length ACL.

2. The least probable symbols have codewords of equal length.

3. The longest codewords differ only in their final symbol.

- 2.1 - The algorithm.

We consider the binary case; the extension to the general case is straightforward. Let

$$S = \begin{pmatrix} s_1 & s_2 & \cdots & s_n \\ p_1 & p_2 & \cdots & p_n \end{pmatrix}$$

denote a source with probabilities arranged in non-increasing order. Define $R(S)$ as the *reduced source*, obtained by replacing the two least probable symbols in S with a single symbol whose probability is the sum of the merged symbols:

$$R(S) = \begin{pmatrix} s_1 & s_2 & \cdots & (s_{n-1}, s_n) \\ p_1 & p_2 & \cdots & p_{n-1} + p_n \end{pmatrix}$$

Let C_R denote a binary prefix code for $R(S)$, and let z be the codeword assigned to the merged symbol (s_{n-1}, s_n). A prefix code C for S can then be obtained from C_R by assigning the i-th symbol of S the i-th codeword in C_R for $i \leq n-2$. The codewords for s_{n-1} and s_n are $z0$ and $z1$, respectively.

- 2.1.1 - The encoding.

Let ω be a string to encode, produced by some source S. To encode such string, one must perform the following steps:

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 this process until a root is formed.
4. Build the code by traversing the tree from the root to each leaf, assigning 0 to left paths and 1 to right paths.

Example

Let $S = \{1, 2, 3, 4, 5\}$ whose probabilities are $\{1/3, 1/6, 1/3, 1/12, 1/12\}$, respectively. Applying the steps above yields:

Step 1. Sorting the symbols according to the probabilities gives the order: $1, 3, 2, 4, 5$.

Step 2. Consider the two least probable symbols, d and e. Their sum defines a new node x in the tree (the one in the figures). The next node y (the one) is formed by x and the third least probable symbol b. At this point, node z (the one) can be obtained by either merging a and c or merging c and y, both resulting in an optimal code.

In this example, c and y are merged; the alternative is shown in (b). Finally, the last two symbols are merged to form the tree (a).

Figure 1: Huffman trees for the source S. (a) (on the left) is obtained by merging c and y to form z; (b) (on the right) merges a an c instead.

Step 3. Traversing the tree produces $C = \{0, 10, 110, 1110, 1111\}$ for tree (a) and $C = \{00, 01, 10, 110, 111\}$ for tree (b).

From the steps above, it is evident that the slowest part of the encoding is the necessity to scan the text twice.

- 2.1.2 - The decoding.

Let's assume we receive a text encoded using Huffman. How can we decode it? We remark that decoding requires knowledge of either the Huffman tree or the source probabilities, which in both cases constitute an overhead relative to the bare encoding. Assuming this overhead is known, decoding is straightforward. Specifically, one reads the encoded text and traverses the tree accordingly until reaching a leaf. This process is repeated until the entire text is decoded.

Example

Let $\omega = 00101111100001$ be a text encoded using Huffman. Assume the tree used is the **(b)** one of *Figure 1*. Starting from the root, the first symbol in the encoded text is zero, so the left subtree is considered. Since the current node is not a leaf, reading continues. Reading another zero, we again follow the left subtree. Upon reaching a leaf, we decode 00 as a. The traversal then returns to the root, and the process is repeated. After a few iterations, the decoded text is $\omega = abedac$.

- 2.1.3 - Adaptive Huffman.

As previously noted, the main limitation of the classic Huffman approach is the necessity to scan the text twice, which slows down encoding. Additionally, proper operation requires knowledge of the symbol probabilities in advance, which is not always feasible. To address these issues, a new method has been developed, known as *Adaptive Huffman*.

The key feature of this approach is that the tree is built and updated dynamically. The procedure can be summarized as follows:

- Start with an initially empty tree, or one containing only a special **Not Yet Transferred** (**NYT**) node.
- For each symbol in the text:
  - If the symbol has already been encoded, encode it using the existing code.
  - If it is a new symbol: encode the **NYT** node first, then encode the symbol itself and add it to the tree as a leaf.
  - After encoding a symbol, update the tree: increment the frequency of the symbol and its ancestors, and reorganize the tree to maintain the Huffman properties.

It can be shown that the adaptive approach exhibits 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 issues is that the encoder must transmit, alongside the text, the tree used for encoding, which constitutes a non-trivial overhead. Additionally, the decoding phase is slow due to the necessity of traversing the tree for each symbol.

It should be noted that there exist compact codes that are not produced by the Huffman algorithm. We now introduce the so-called *Canonical Huffman encoding*, a variant of Huffman that addresses the issues above by requiring the encoder to transmit only the lengths of the codewords.

This encoding is particularly useful when the source alphabet is large and fast decoding is required.

- 2.2.1 - The encoding.

Let C be the standard Huffman encoding for a source. To obtain the canonical version, proceed as follows:

1. Compute the length of each codeword in C with respect to each symbol in the source alphabet.

2. Construct the num array, where each entry num[l] stores the number of symbols having length l.

3. Construct the symb array, where each entry symb[l] contains the symbols of length l.

4. Construct the fc array, where each entry fc[1] stores the first codeword of all symbols with codeword length l. This construction is implemented using the shown in *Figure 2.a*

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

- 2.2.2 - The decoding.

Let ω be a text encoded using the Canonical Huffman encoding. Let $l = 1$ be a counter and v the first bit read from ω. To retrieve ω's content we check whether $v <$ fc[1]; if so, we shift v by one to the left and scan the next bit and increment l by one, if not we return the symbol symb[l, v − fc[l]].

Figure 2.b shows a C implementation of the procedure just described: here we assume next_bit to be some existing primitive. We note that the procedure shown in (a) builds fc using a reverse approach; this, in general, produces a fc array that differs from one built using the forward approach.

```
fc[MAX] = 0;
for (int l = MAX - 1; l >=
        1; l--) {
    fc[l] = (fc[l + 1] +
          num[l + 1] >> 1);
}
```

```
char v = next_bit();
int l = 1;
while (v < fc[l]) {
    v = v << 1 + next_bit
          ();
    l++
}
return symb[l, v - fc[l]];
```

Figure 2: (a) (on the left) implements the fc construction; (b) (on the right) implementats the Canonical Huffman decoding.

Example

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

1. Let l_i denote the length of the i-th codeword: $l_0 = l_1 = l_2 = 2$, $l_3 = l_4 = 3$.
2. Computing num yields: num[1] = 0, num[2] = 3, and num[3] = 2.
3. Computing symb gives: symb[1] = null, symb[2, 0] = a, symb[2, 1] = c, symb[2, 2] = b, symb[3, 0] = d, and symb[3, 1] = e. For convenience, a second index is used to navigate within a list when needed.
4. Computing fc yields: fc[1] = 2 (used in the decoding process), fc[2] = 1, and fc[3] = 0.
5. Observe that symb[1] = null, indicating that there are no codewords to assign. For symb[2], assign 01 to a, 10 to c, and 11 to b. For symb[3], assign 000 to d and 001 to e.

Now let $\omega = 01100000011011$ be a text encoded using the Canonical Huffman encoding, and let symb and fc be those just computed. Applying the procedure in (b), one can verify that $\omega = acdecb$.

Exercises

1. Given the following strings, compute both the standard and the canonical Huffman encodings, and compare the results in terms of code length.
   - $\omega_0 = abbaca$
   - $\omega_1 = cabbab$
2. Decode the following strings assuming they have been encoded with standard Huffman coding, and the alphabet is $\Sigma = \{a, b, c\}$. Also reconstruct the Huffman tree for each string.
   - $s_0 = 1010110$, with symbol frequencies inferred from the string

- $s_1 = 1100101$, with symbol frequencies inferred from the string

3. Compute the adaptive Huffman encoding for the following strings, updating the tree after each symbol. Compare the final code length with the standard Huffman encoding.

   - $\omega_0 = abacb$
   - $\omega_1 = bbaca$

4. Decode the following strings assuming adaptive Huffman coding was used, and reconstruct the tree step by step.

   - $s_0 = 0110101$, with initial alphabet $\Sigma = \{a, b, c\}$
   - $s_1 = 1101010$, with initial alphabet $\Sigma = \{a, b, c\}$

5. Decode the following canonical Huffman encoded strings, given the symbol lengths for each alphabet. Ensure that the decoded text matches the original string.

   - $s_0 = 000110101$, with symbol lengths $\{a : 2, b : 3, c : 3\}$
   - $s_1 = 0110010$, with symbol lengths $\{a : 2, b : 3, c : 3\}$

- 3 - Aritmethic coding.

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

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

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

which, according to Shannon's theorem, takes values in the range $[0, 1]$. The redundancy of the code is defined as

$$\rho(C) = 1 - \vartheta(C),$$

where, when considering Huffman coding, one can show that the redundancy approaches 1 as the probability of the most frequent symbol approaches 1.

This naturally leads to the question: can we do better than Huffman? The answer is affirmative, but it requires a completely different approach: instead of assigning a codeword to each symbol, we assign one to the entire text. The method we are about to present is known as *aritmethic coding (AC)*.

- 3.1 - The encoding.

The AC encoding is relatively straightforward, as summarized in the steps below. Let ω be the text to encode, and assume that the alphabet and corresponding probabilities are known. Then:

1. Initialize the range $[0, 1]$.
2. For each symbol in ω:
   (a) Divide the current range into n intervals (n being the size of the alphabet), each proportional to the probabilities of the symbols.
   (b) Select as the current subinterval the one corresponding to the symbol being analyzed.
3. Output the binary representation[2]of the midpoint of the last subinterval.

Algorithmically, this procedure can be implemented via the procedure shown in *Figure 3*.

[2]We assume the reader is familiar with representing fractional numbers in binary.

```
procedure AC_encode(S, n, P[σ], f_σ){
    s_0 = 1
    l_0 = 0
    i = 1
    while (i <= n)  {
        s_i = s_{i-1} * P[S[i]];
        l_i = l_{i-1} + s_{i-1} * f_{S[i]}
        i = i + 1
    }
    return (x ∈ (l_n, l_n + s_n))
}
```

Figure 3: Pseudo code for encoding strings using AC.

Example

Let $\omega = abac$ and $S = \{a, b, c\}$ with probabilities $\{1/2, 1/4, 1/4\}$.

By applying the above steps, we obtain:

Step 1. Initialize the range $[0, 1]$.

Step 2. Consider each iteration; each step is illustrated in *Figure 4*. We read "a", selecting the range $[0, 1/2]$. We then update the ranges and repeat the process until the end of the text.

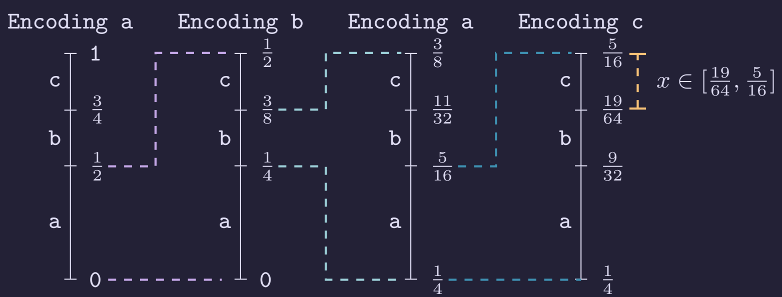


Figure 4: Step by step encoding of $\omega = abac$ via the AC.

Step 3. Return the midpoint of the last range.

How many bits are required to represent the encoding? From the code in *Figure 3*, the size of the final range is given by

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

from which it follows that the output size is independent of the permutation of the string symbols. It can be shown that $l_n + s_n/2$ bits suffice our needs.

- 3.2 - The decoding.

Decoding a text compressed via AC, is as straightforward as its encoding. In fact, the decoding proceeds similarly to the encoding. The procedure, shown in *Figure 5*, is as follows:

1. Initialize the range $[0, 1]$.
2. Divide the subrange according to the symbol probabilities, as done during encoding.
3. At each step, select as the current interval the one in which the encoded text falls.

```
procedure AC_decode (binaryS, n, P[σ], f_σ){
    s_0 = 1
    l_0 = 0
    i = 0
    while (i <= n) {
        subdivide the current interval into subranges
            according to the probabilities
        return the symbol x associated to the current
            range

        S = S::σ
        s_i = s_{i-1} * P[σ]
        l_i = l_{i-1} + s_{i-1} * f_σ
        i = i + 1
    }
    return S
}
```

Figure 5: Pseudo code for decoding AC encoded strings.

Example

Let $x = 0.3047$. Let the alphabet and the probabilities be those of the previous example.

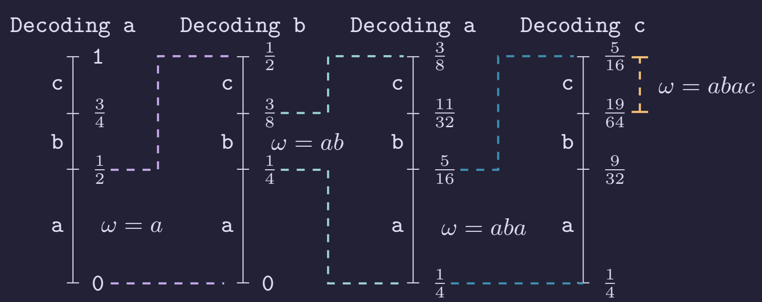


Figure 6: Step by step decoding of $x = 0.3047$ via the AC.

Applying the decoding steps to x yields what shown in *Figure 6*.

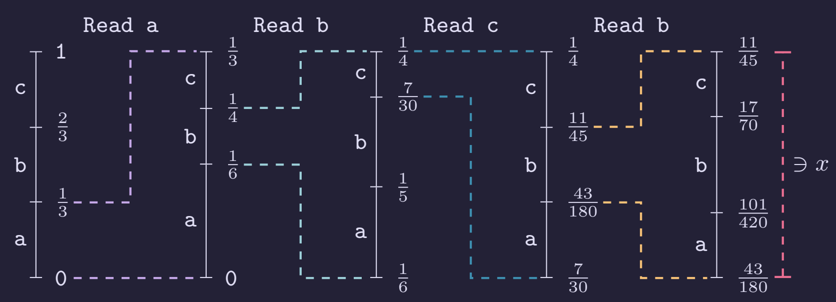
- 3.3 - The adaptive version.

It is easy to observe from *Figure 3* that the necessity of knowing the probabilities a priori slows down the encoding. The question then arises: can we improve arithmetic coding? The answer is affirmative, by assuming initially equal probabilities. Briefly, before any symbol is read, we assume that each symbol in the alphabet has the same probability of appearing in the text. Then, at each step, we apply the procedure in *Figure 3* and update the probabilities accordingly.

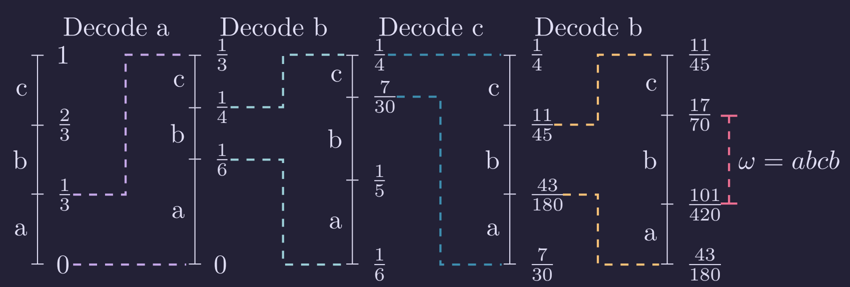
What about decoding? We can apply the same logic, with a slight difference: instead of updating the probabilities after a symbol is read, we update them each time a symbol is decoded using the procedure in *Figure 4*.

Example

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



Now, let us decode 0.2416 with $n = 4$. Again, we assume equal initial probabilities, decode the first symbol (a), and update the probabilities.



Repeating the process, we successfully decode the text $abcb$.

Exercises

1. Given the following strings, compute both the standard and the adaptive AC encoding, and compare the results.

   (a) $\omega_0 = acabba$

   (b) $\omega_1 = aabbca$

2. The following strings have been encoded via adaptive AC encoding, try to decode them assuming the alphabet is $\Sigma = a, b, c, d$.

   (a) $s_0 = 0.456731$, with $n = 4$.

   (b) $s_1 = 0.123$, with $n = 10$.

   (c) $s_2 = 0.549201$, with $n = 5$.

- 4 - Integers encoding.

What occurs when the source to be encoded employs positive integers as symbols? How can one determine a universal representation? More precisely, how can we construct a code that is prefix-free and whose ACL is $\mathcal{O}(\ln x)$ for any x?

Is it possible to employ the previously discussed codes, or is it necessary to develop new ones? The answer depends on the context. For example, if the distribution of the integers is known and the range is relatively small, Huffman coding remains applicable. Otherwise, alternative coding schemes, which will be discussed in the subsequent sections, are required.

Remark. The restriction to positive integers can be relaxed by mapping any positive integer x to $2x + 1$ and any negative integer y to $-2y$.

- 4.1 - Simple approaches.

One of the simplest approaches is to employ the binary representation of the given integers. However, there is a limitation: such a representation is not prefix-free.

A more formally valid, though still not efficient on its own, method is the so-called *unary encoding*. Specifically, given an integer x, we encode it as a sequence of $x - 1$ zeros followed by a one. It is evident that this representation is prefix-free; however, it is not practical, as the number of bits required grows linearly with x.

Clearly, an alternative is required. Although numerous codes exist for encoding integers, the following sections concentrate on the *Elias codes* and the *Fibonacci codes*.

- 4.2 - Elias codes.

When referring to the Elias code, we generally mean two codes: the *gamma* code and the *delta* code, both proposed by *Peter Elias* in [3].

Before introducing the codes, it is useful to define some notation. Let x be an integer, and let $B(x)$ denote its binary representation. We define $|B(x)|$ as the number of bits required to represent x.

- 4.2.1 - Gamma code.

Let x be a positive integer. Its gamma code, denoted $\gamma(x)$, is a binary sequence composed of the following elements:

- the unary encoding of $|B(x)|$, and
- the binary representation of x with the most significant bit removed.

Decoding is straightforward: count the nomber of zeros up to the first 1, say they are k, treat the next $k + 1$ bits (1 included) as the integer x.

Example

For $x = 11$, the gamma encoding is

$$\gamma(x) = 0001011,$$

where 0001 corresponds to the unary encoding of $|B(x)|$ and 011 represents the binary representation of x with the most significant bit removed.

It can be shown that this encoding requires at most $2\lfloor \log x \rfloor + 1$ bits. Most importantly, this code is particularly efficient when the probability distribution is $p(x) = 2x^{-2}$.

- 4.2.2 - Delta code.

Let x be a positive integer. Its delta code, denoted $\delta(x)$, is a binary sequence composed of the following elements:

- the gamma encoding of $|B(x)|$, and
- the binary representation of x with the most significant bit excluded.

Decoding is straightforward: count the number of zeros up to the first 1, say they are k; treat the next $k + 1$ bits as the integer h; interpret the following h bits as the integer x.

Example

For $x = 11$, the delta encoding is

$$\delta(x) = 00100011,$$

where 00100 corresponds to the gamma encoding of $|B(x)|$ and 011 represents the binary encoding of x with the most significant bit excluded.

It can be shown that the delta code requires at most

$$1 + \log x + 2 \log \log x$$

bits. Importantly, delta codes are efficient when the probability distribution is

$$p(x) = \frac{1}{2x(\log x)^2}.$$

- 4.3 - Fibonacci code.

Before presenting Fibonacci codes, recall that Fibonacci numbers are defined by the recurrence

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

We also introduce a fundamental result essential for Fibonacci codes.

Theorem (Zeckendorf) *Any positive integer can be uniquely expressed as the sum of non-consecutive Fibonacci numbers.*

Using this theorem, we can construct a code that efficiently encodes integers. Let n be a positive integer. The encoding procedure is as follows:

1. Find the largest Fibonacci number less than or equal to n.
2. Suppose it is the i-th Fibonacci number. Subtract it from n and record the remainder. Set the $(i-1)$-th bit to 1 (the leftmost bit has index 0).
3. Repeat the above steps, replacing n with the remainder, until it becomes 0.
4. Append an additional 1 to the right end of the codeword.

Example

For $n = 73$, it can be expressed as $F_{10} + F_7 + F_5$. Applying the above procedure, the Fibonacci encoding of n is

$$0001010011.$$

Decoding is straightforward. Remove the additional 1 at the right end of the codeword. Then, from left to right, replace the i-th 1 with the $(i+1)$-th Fibonacci number, and sum all the corresponding Fibonacci numbers.

Example

To decode 0001010011, first remove the extra bit, obtaining 000101001. Then, replace the i-th 1 with the $(i+1)$-th Fibonacci number, resulting in

$$F_5 + F_7 + F_{10} = 73.$$

Exercises

1. Encode $5, 9, 20$ using gamma codes. Write the unary prefix and binary suffix for each.
2. Decode the gamma codewords $00101, 00111, 0001010$. Show the intermediate steps.
3. Encode $15, 30, 50$ using delta codes. Include the gamma encoding of the binary length and the binary part without the most significant bit.
4. Decode the delta codewords $001010111, 0011011110, 01000111010$. Reconstruct the integers from the gamma length and binary parts.
5. Encode $23, 37, 60$ using Fibonacci codes. List the Fibonacci numbers used, positions of 1s, and the final codeword.
6. Decode the Fibonacci codewords $0010101, 0001010101, 01001101$. Remove the extra 1, map to Fibonacci numbers, and sum to obtain the integer.
7. Encode $8, 18, 35$ using gamma, delta, and Fibonacci codes. Compare the code lengths and discuss the most efficient code for each integer.

- 5 - Compression optimization.

So far, we have discussed several compression techniques under the assumption that the input to such compressors is "optimal"[3]. However, what if this is not the case? Can we preprocess the string in such a way that, once it reaches the compressor, it becomes optimal? The answer is affirmative, and this section introduces one of such optimizations.

- 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 the string to be compressed and thus preprocessed using the BWT. The first step of the transform is to compute all lexicographically[4], and finally, the last column[5]is returned together with the index (starting from 0) of the original string.

Example

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

$$aleph\ , ephal\ , halep\ , lepha\ , phale$$

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

We now highlight some important properties of the BWT.

1. $\forall i \neq I$, where I is the index of the original string, $F[i]$ follows $L[i]$ in ω.

2. $F[I]$ is the first symbol of ω.

3. For any character x, the i-th occurrence of x in F corresponds to the i-th occurrence of x in L.

, cyclic rotations of ω. Next, these rotations are sorted

- 5.1.1 - Inverse of the BWT.

Since the BWT is a transform, it is desirable for it to be reversible. Indeed, the BWT is reversible and admits two distinct but equivalent inverse transforms: the FL-mapping and the LF-mapping.

We illustrate these inverse transforms using the BWT of the previous example as input.

[3]In this context, "optimal" refers to a string with the lowest possible entropy.

[5]For the sake of this discussion, assume that the lexicographic order is the usual alphabetical one. This is, in general, not true.

[5]By last column we mean the concatenation of the last symbol of each rotation.

FL-mapping.

Let $L = BWT(aleph)$ and $I = 0$ be the index of the original string. We construct F by lexicographically sorting L. 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 $i = 0, 1, \ldots, 4$. Concretely, $\omega[0] = F[I] = a$, $\omega[1] = F[\tau[0]] = F[3] = l$, and so forth.

LF-mapping.

Let $L = BWT(aleph)$ and $I = 0$ be the index of the original string. We construct F by lexicographically sorting L. 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 $i = 0, 1, \ldots, 4$. Concretely, $\omega[4] = L[I] = h$, $\omega[3] = L[\sigma[0]] = L[2] = p$, and so forth.

- 5.2 - Analysis of the BWT.

From the discussion of the BWT, a natural question arises: how much does the compression improve by preprocessing the string using the BWT? This question is addressed in [9] by Manzini. Here, Manzini analyzes the compression of BWT-based algorithms in terms of the *empirical entropy* of the input string. The use of empirical entropy instead of Shannon's entropy is to search in the fact that the former is defined in terms of occurrences of a symbol or a group of symbols. Therefore, it is defined for any string without requiring any assumption on the underlying probability distribution, which makes it suitable for a worst case analysis.

Let S be a string of lenght n over the alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$, and let n_i be the number of occurrences of σ_i in S. The zeroth order empirical entropy of the string S is defined as

$$H_0(S) = -\sum_{i=1}^{m} \frac{n_i}{n} \log \frac{n_i}{n}.$$

We use the convention that $0 \log 0 = 0$. As can be easily understood, $|S|H_0(S)$ represent the output size of an ideal compressor which uses $-\log \frac{n_i}{n}$ bits for coding symbols σ_i. If we consider the k preciding symbols in defining the codewords, a better compression ratio can be achived. For any string $\omega \in \Sigma^*$ define ω_S to be the concatenation of the single symbol following ω in S. Then, the quantity

$$H_k(S) = \frac{1}{|S|} \sum_{\omega \in \Sigma^*} |\omega_s| H_0(S)$$

represents the k-th order empirical entropy.

Manzini also introduces the concept of *modified* empirical entropy $H_k^*(S)$, (see [9, Definition 2.1] for the mathematical formulation), which is defined by imposing the additional requirement that the encoding of S takes at least the number of bits needed to represent S in binary.

Among all the other things discussed throughout the paper, of interest are the following results for the proves of which, once again, we remind to the paper.

Theorem (Manzini) *For any string S over an 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 $\mu = 1$.

Here, $BW_0 = Order_0 + MTF + BWT$, where $Order_0$ denotes a 0-th order compressor.

Theorem. *For any string S over an alphabet Σ and for any $k \geq 0$, there exists a constant g_k such that*

$$BW_0 + RL \leq (5 + 3\mu)|S|H_k^*(S) + g_k,$$

where RL denotes the run-length encoding of the string.

To conclude this section, observe that the primary bottleneck of the BWT is the sorting of cyclic rotations. In the following section, we show how this problem can be reduced to the sorting of suffixes.

Remark. Both MFT and RLE will be briefly discussed in *Section 7.1*

- 5.2.1 - Efficient computation of the BWT.

Let ω be the string for which we wish to compute the BWT, and let $\tilde{\omega} = \omega\$$, where $\$ \notin \Sigma$. If we compute the BWT of $\tilde{\omega}$, it suffices to sort the suffixes rather than all cyclic rotations, as would be required for ω.

A natural question arises: how are these suffixes computed and sorted? Over the years, many algorithms have been proposed (see [1, 6, 7]), all of which share a common component: a specialized data structure, the SA. The following section summarizes the algorithm proposed in [6]; for a complete understanding, the original paper is recommended.

Formally, let $T = T[1, n]$ be a text, and define T_i as the suffix of T starting at position i for all $i = 1, \ldots, n$. Denote by $Suff(T)$ the set of all suffixes of T:

$$Suff(T) = \{T_i \mid i = 1, \ldots, n\}.$$

The suffix array, SA, is defined as a sorted array of $Suff(T)$.

Remark. Using the SA, the complexity of computing the BWT is reduced to that of the algorithm used to sort the suffixes.

The DC3 algorithm.

Let S be the text for which we wish to compute the suffixes. Our goal is to construct SA_S. The first step of the algorithm is to compute the following position sets:

$$B_k = \{i \in [0, n-1] \mid i \bmod 3 = k\}.$$

Let $C = B_1 \cup B_2$. For each $i \in C$, define the triplet $r_i = [s_i, s_{i+1}, s_{i+2}]$, padding with \$ if necessary, and define R as the concatenation of all triplets:

$$R = r_1 r_2 \ldots$$

There exists a correspondence between the suffixes of R and those of S. Next, radix sort the triplets and replace each with its rank to define R'. If all symbols are distinct, the order is determined by the ranks; otherwise, the suffixes of R' are recursively sorted using the DC3 algorithm to compute $SA_{R'}$.

Once the suffixes are computed, for all $i \in C$ define $\mathrm{rank}(S_i)$ as the rank of S_i. For all $i \in B_0$, define the pair $(s_i, \mathrm{Rank}(S_{i+1}))$ and sort them as follows: for all $i, j \in B_0$,

$$S_i \leq S_j \iff (s_i, \mathrm{Rank}(S_{i+1})) \leq (s_j, \mathrm{Rank}(S_{j+1})),$$

using radix sort on the pairs.

Finally, merge the two sets of suffixes as follows: for all $i \in C$ and $j \in B_0$, distinguish two cases:

$$\begin{aligned} i \in B_1 : \quad & S_i \leq S_j \iff (s_i, \mathrm{Rank}(S_{i+1})) \leq (s_j, \mathrm{Rank}(S_{j+1})) \\ i \in B_2 : \quad & S_i \leq S_j \iff (s_i, s_{i+1}, \mathrm{Rank}(S_{i+2})) \leq (s_j, s_{j+1}, \mathrm{Rank}(S_{j+2})). \end{aligned}$$

Theorem. *The complexity of the DC3 algorithm is $\mathcal{O}(n)$.*

Proof Aside from the recursive call, all operations can be performed in linear time. The recursion is applied to a string of length $2n/3$, giving the recurrence

$$T(n) = T(2n/3) + \mathcal{O}(n),$$

whose solution is $T(n) = \mathcal{O}(n)$. □

Example

Let $S = mathisawesome$. Computing B_k for $k = 0, 1, 2$, we have

$$B_0 = \{0, 3, 6, 9, 12\},$$
$$B_1 = \{1, 4, 7, 10\},$$
$$B_2 = \{2, 5, 8, 11\},$$

so that $C = \{1, 2, 4, 5, 7, 8, 10, 11\}$.

By considering all $i \in C$ and computing all r_i, we obtain

$$R = [ath][thi][isa][saw][wes][eso][ome][me\$].$$

After radix sorting the triplets, we get

$$R' = (1, 7, 3, 6, 8, 2, 5, 4),$$

from which $SA_{R'} = (8, 0, 5, 2, 7, 6, 3, 1, 4)$. Note that index 8 corresponds to the empty suffix, the lowest in order.

Before computing $\text{Rank}(S_i)$ for all $i \in C$, reduce $SA_{R'}$ to SA_R by excluding index 8. Assign ranks to entries in SA_R in order. Then, assign the j-th rank to the entry of R at index $k = SA_R[j-1]$ and trace it back to the corresponding S_i. For illustration, see:

j	1	2	3	4	5	6	7	8
$SA_R[j-1]$	0	5	2	7	6	3	1	4
$R[k]$	[ath]	[eso]	[isa]	[$me\$$]	[ome]	[saw]	[thi]	[wes]
S_i	S_1	S_8	S_4	S_{11}	S_{10}	S_5	S_2	S_7

The remaining steps are left as an exercise to the reader.

- 6 - Dictionary based compressors.

The compression schemas we've seen so far use a statistical approach to compress the text. This is, in some case an issue: the source distribution may be unknown, it may change with time, etc. If that's the case, compressors based on Huffman or the arithmetic encoding lose their strenght.

To solve this issue a new class of compressors was developed: the class of *dictionary based* compressor. Formally, a dictionary D is defined as the set of pairs $(f, c), f \in F, c \in c$ where F and C are, respectively, the set of factor and the set of associated codewords.

$$D = \{(f, c) \mid f \in F, c \in C\}.$$

But how does the use of this dictionaries solve our issue? Basically, these type of compressors use a dictionary as some sorta of look-up table.

Though we refer to these schemas as dictionary based compressors, more often then not, we actually refer to a sub-class: the Lempel-Ziv (LZ) based compressors.

- 6.1 - Lempel-Ziv based compressors.

Initially posposed by *A. Lempel* and *J. Ziv* in [13–15], these are a type of dictionary based compressors in which the dictionary is built dinamically; that is, instead of having a pre-defined codebook, we create it while the text is being read. The reason this works is simple: with an high degree of probability, the text previously encoded is a great soure of patterns (it shares the same language, style and structure of the upcoming text).

One of the most important aspects of LZ-based compressors is the fact that there's no need to transmit the dictionary, as we shall see shortly.

Remark. Though other variants exists, in the next few sections we'll focus on LZ-77, LZss, LZ-78 and LZW.

- 6.2 - LZ77.

The idea is to divide the text in two parts: a *search buffer* – a fixed size portion of the text already processed – and a *lookahead buffer* – the portion of text not yet seen; which are identified by two pointers i, j as shown in *Figure 7*.

$$S = \underbrace{s_1\, s_2\, s_3\, s_4\, s_5 \cdots}_{\texttt{Search Buffer}}\underbrace{\cdots\cdots\cdots\cdots s_n}_{\texttt{Lookahaed Buffer}}$$

Figure 7: LZ buffers.

Basically, we look for the longest existing pattern in the search buffer, that is prefix of the lookahead buffer. The compression produces a set of triplets $< f, l, c >$ where f is the distance between i and j, l is the length of the found prefix and c is the symbol after the prefix.

Example

Let $S = aabbaaabaca$. Let's assume we are in the situation below.

```
        1 2 3 4 5 6 7 8 9 10 11

S =     a a b b a a a b a c a
        |---------|---------|
          Search    Lookahead
          Buffer    Buffer
```

Since the search buffer contains a patter that is prefix of the lookahead, we return the triplet $< 5, 3, a >$.

The decompression is easy, and, as stated previously, doesn't require the dictionary to be transmitted. In fact, after reciveing a triplet, we look f characters deep into the search buffer (that is initially empty), copy l symbols into the lookahead buffer (which at this point represent the text being decoded at each step) and concatenate c to it. If needed, we move the search buffer.

Example

Let us consider the triplet $< 5, 3, a >$ of the previous example. Again, we assume we are in a situation like the one below.

```
        1 2 3 4 5

S =     a a b b a a a b a
        |---------|---------|
          Search    Lookahead
          Buffer    Buffer
```

We look 5 characters deep into the search buffer, copy 3 symbols (abb) into the lookahead buffer and add an a at the end.

- 6.2.1 - LZss: a variation to LZ77.

In 1982, *Storer* and *Szymanski* observed that, while compressing with LZ77 two situation may occur: a match is found, or it has not. In either case the use of a third component makes no sense. In [11] they propose a variant of LZ77, the LZss, that instead of the triples, make use of pairs of the type: $< d, |\alpha| >$ when a match is found, or $< 0, c >$ for when it has not.

Besides this change, the rest of the algorithm is analogous to LZ77.

Comparison of LZ77-type parsings.

Let us denote by z the number of phrases in LZ77 encoding, and by z' those of LZss. Define LZ77-/LZss-type parsings, respectively, as follow.

LZss-type parsing Let $S = t_1 t_2 \ldots t_r$ be a decomposition of S into non-empty strings $t_1, t_2, \ldots, t_r$. We say that $t_1, t_2, \ldots, t_r$ is an LZss-type parsing if of each $i \in [1, r]$, the string t_i is either a letter or has an occurence in the string $s[1, |t_1, t_2, \ldots, t_r| - 1]$.

LZ77-type parsing Let $S = t_1 t_2 \ldots t_r$ be a decomposition of S into non-empty strings $t_1, t_2, \ldots, t_r$. We say that $t_1, t_2, \ldots, t_r$ is an LZ77-type parsing if of each $i \in [1, r]$, the string $t_i[1, |t_i| - 1]$ has an occurence in the string $s[1, |t_1, t_2, \ldots, t_r| - 2]$.

We get the following results.

Lemma (see [15]) *For any string S:*

1. *$z' = |LZss(S)|$ is smaller or equal to the size of any LZss-type factorizations.*

2. *$z = |LZ77(S)|$ is smaller or equal to the size of any LZ77-type factorizations.*

Lemma ([8, Lemma 3]) *For any string S, it holds*

$$\frac{1}{2} z' < z \leq z'$$

Proof Let us consider the following: given S a string, let $f_1 f_2 \ldots f_z$ be its LZ77 parsing. Consider $f_1 t_2 t_2' \ldots t_z t_z'$ where $t_i = f_i[1, |f_i| - 1]$ and $t_i' = f_i[|f_i|]$. One can easily observe that the latter is a LZss-type parsing of size at most $2z - 1$. Hence $z' < 2z$. Moreover, the LZss parsing of S is a LZ77-type factorizations, therefore $z \leq z'$. □

- 6.3 - LZ78.

Mainly differs from LZ77 by the fact that the dictionary is explicitly built. Formally speaking, LZ78 outputs a sequence of pairs of the form $< index, nextchar >$, where

- *index* is the position of the prefix in the dictionary, and
- *nextchar* is the character following the known prefix.

Each phrase formed this way is then added to the dictionary.

A question arise naturally: how do we store such phrases? One of the main features that we look for in such a data structure is a fast look up. For this reason, in general, we use trie[6].

[6] We assume the reader to be familiar with such data structure.

Example

Let $S = aabbaaabaca$ and let $D = \{\}$ be the current dictionary. When trying to encode the first a we observe that $D = \varnothing$ meaning there is no phrase to encode a; thus, the pair $< 0, a >$ is formed and added to D. When encoding the second a a phrase is found, thus the pair $< 1, a >$ is formed and added to D. At this point D contains the phrases a, aa at position 1 and 2, respectively. We proceed similarly to encode the whole string.

Remark. There exist a version of LZ78, knwon as LZW proposed by *Terry A. Welch* in [12]. This variant differs from LZ78 by the absence of *nextchar*. Precisely, each new phrase is obtained by appending the first character of the following phrase to the current one.

- 7 - Pattern matching on compressed indexes.

Let us consider the following problem: given a document, can we search for a pattern in it efficiently? It is well known that several algorithms exist to solve this problem, the most notable of which is the *Knuth-Morris-Pratt (KMP)* algorithm.

A natural question then is whether similar efficiency can be achieved when the document is compressed; that is, whether both space and time efficiency are attainable in pattern matching on compressed texts. An affirmative answer was provided by Ferragina and Manzini through a novel data structure they defined: the *FM-index*.

The remainder of this section will describe the algorithm to build the FM-index for a given text, though we refer the reader to [4] for further details. We first discuss the *Backward-search* algorithm applyied to text compressed using BWT-based compressors, we then discuss the results of Ferragina and Manzini.

We recall that for any given string S, the output of BWT(S) is the last colunm L and the index I correspondig to the position S in the list of permutions. To work properly the Backward-search requires, alongside L, two auxiliaries data structure:

- C[1, ..., n] (n begin the size of the alphabet) that at C[c] stores the number of characters lexicographically smaller the c in T, and
- Occ(c, q) stroring the occurencies of c in the prefix L[1, q].

Let P be the pattern we are interested in. The algorithm proceeds as follow: we first read the right-most character in P, say it's c, we then we consider First = C[c] + 1 and Last = C[c + 1]. We continue updating the value of c, First and Last accordingly until either First > Last or we have reached the end of the pattern. The full algorithm is shown in *Figure 8*.

```
Backward_search(P[1, p], C) {
    i = p
    c = P[p]
    First = C[c] + 1
    Last = C[c + 1]
    while ((First <= Last) and (i >= 2)) {
        c = P[i − 1]
        First = C[c] + Occ(c, First − 1) + 1
        Last = C[c] + Occ(c, Last)
        i = i + 1
    }
    if (Last < First) then "No patter found"
        else <First, Last>
}
```

Figure 8: Pseudo code for the implementation of Backward-search.

Example

Consider $L = BTW(\omega) = ipssm\$pissii$ for some string $\omega \in \Sigma^*$, and let $P = pssi$. Since L is the output of the BWT of some string, we know how to compute the F column[7]; thus we get $F = \$iiiimppssss$. We summarize the example in *Figure 9*.

L	i	p	s	s	m	$	p	i	s	s	i	i
F	$	i	i	i	i	m	p	p	s	s	s	s
	1	2	3	4	5	6	7	8	9	10	11	12
		First ↑			↑ Last							

L	i	p	s	s	m	$	p	i	s	s	i	i
F	$	i	i	i	i	m	p	p	s	s	s	s
	1	2	3	4	5	6	7	8	9	10	11	12
									First ↑	↑ Last		

L	i	p	s	s	m	$	p	i	s	s	i	i
F	$	i	i	i	i	m	p	p	s	s	s	s
	1	2	3	4	5	6	7	8	9	10	11	12
											First ↑	↑ Last

L	i	p	s	s	m	$	p	i	s	s	i	i
F	$	i	i	i	i	m	p	p	s	s	s	s
	1	2	3	4	5	6	7	8	9	10	11	12
								Last ↑	↑ First			

Figure 9: Backward-search on $L = ipssm\$pissii$. . We use the colors ■ , ■, ■, ■ to show each value of c, *First* and *Last*.

By the algorithm in *Figure 8*, we begin by considering $c = P[4] = i$ and, since C is known (we can easily compute it from F), we compute $First = 2$ and $Last = 5$. Since no halt condition is met, we proceed reading the next symbol in the pattern and update both *First* and *Last*. Hence, we have $c = P[3] = s$ and $First = 9$ and $Last = 10$. Again, the looping conditions are met, thus we update c, *First* and *Last* once again. Proceeding analogously for the reminder of P, we find the P does not appear in ω.

[7] We use F to visualize what C actually represents.

Let us observe that the number of iteration as strictly depending on the lenght of the pattern; additionally it is strongly affected from the computation of OCC. Thus, if we can construct a OCC array such that OCC [c][q] $= Occ(c,q)$, then the backward-search would take $\mathcal{O}(|P|)$. This idea allowed Ferragina and Manzini to create a compressed index – the FM-index – by using an implementation of the backward-search that works in $\mathcal{O}(|P|)$ and requires just $5n\ H_k(T) + o(n)$ bits.

- 7.1 - FM-index.

The FM-index has several applications in computational biology and many other scientific fields. The reason behind its success is due the fact that it allows to search and index all the occurences of a given pattern P efficiently. In fact, as we should show, it takes time $\mathcal{O}(|P|)$ to count all the occurences of the pattern, time $\mathcal{O}\big(|P|\log^{1+\varepsilon} n\big)$, where ε is an arbitrary positive constant chosen during the construction of the index, to locate them and at most $5n\ H_k(T) + \mathcal{O}(n/\log^{\varepsilon} n)$ bits of space.

The algorithm operates under the assumption that the input text has been compressed through a specific sequence of preprocessing steps. First, the BWT is applied to the original text. The resulting sequence is then processed using the MTF transform, followed by RLE. Finally, a variable-length prefix coding scheme is employed to produce the compressed representation. We refer to such a compressor as BW_RLX, keeping the same notation used by Ferragina and Manzini.

We briefly recall how MTF and RLE work.

MTF: Consider the array $MTF[0, |\Sigma| - 1]$ lexicographically sorted. Replace character c in the text with the number of distinct characters seen since the previous occurence of c. Move c to the front of the MTF array. For instance, let $MTF[0, 3]$, let $S = abbcdcdb$ and assume the lexicographic order to be the alphabetical one ($a < b < c < d$). Applying the MTF to S, we denote it by S_{MTF}, we have $S_{MTF} = 01023112$.

RLE: Consider a string S with some runs[8] in it. Replace each run with its length and the character. For instance, let $S = \$iiiimppssss$, its RLE is $S_{RLE} = \$4im2p4s$.

Let us consider a variant. We use such variant for the rest of this section. Consider a string with runs of zeros. Any such sequence can be wrote as $0^k, k \in \mathbb{Z}^+$. Consider the representation of $k+1$ in binary. Swap the most and the least significant bits, after that drop the new least significant bit. Replace the run with this representation of $k+1$.

[8] In this context, by "run" we refer to a sequence of the same symbol.

References.

[1] Uwe Baier. “Linear-time Suffix Sorting - A New Approach for Suffix Array Construction”. In: *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*. Ed. by Roberto Grossi and Moshe Lewenstein. Vol. 54. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 23:1–23:12. DOI: 10.4230/LIPICS.CPM.2016.23.

[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. DOI: 10.1109/DCC.1997.582009.

[3] Peter Elias. “Universal Codeword Sets and Representations of the Integers”. In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 194–203. DOI: 10.1109/tit.1975.1055349.

[4] Paolo Ferragina and Giovanni Manzini. “Opportunistic Data Structures with Applications”. In: *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, Redondo Beach, California, USA, November 12-14, 2000*. IEEE Computer Society, 2000, pp. 390–398. DOI: 10.1109/SFCS.2000.892127.

[5] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1007/bf02837279.

[6] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. “Linear work suffix array construction”. In: *J. ACM* 53.6 (2006), pp. 918–936. DOI: 10.1145/1217856.1217858.

[7] Pang Ko and Srinivas Aluru. “Space efficient linear time construction of suffix arrays”. In: *J. Discrete Algorithms* 3.2-4 (2005), pp. 143–156. DOI: 10.1016/J.JDA.2004.08.002.

[8] Dmitry Kosolobov and Arseny M. Shur. “Comparison of LZ77-type parsings”. In: *Inf. Process. Lett.* 141 (2019), pp. 25–29. DOI: 10.1016/J.IPL.2018.09.005.

[9] Giovanni Manzini. “An analysis of the Burrows-Wheeler transform”. In: *J. ACM* 48.3 (2001), pp. 407–430. DOI: 10.1145/382780.382782.

[10] Claude E. Shannon. “A mathematical theory of communication”. In: *Bell Syst. Tech. J.* 27.3 (1948), pp. 379–423. DOI: 10.1002/J.1538-7305.1948.TB01338.X.

[11] James A. Storer and Thomas G. Szymanski. “Data compression via textual substitution”. In: *J. ACM* 29.4 (1982), pp. 928–951. DOI: 10.1145/322344.322346.

[12] Terry A. Welch. “A Technique for High-Performance Data Compression”. In: *Computer* 17.6 (1984), pp. 8–19. DOI: 10.1109/MC.1984.1659158.

[13] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Trans. Inf. Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.

[14] Jacob Ziv and Abraham Lempel. “Compression of individual sequences via variable-rate coding”. In: *IEEE Trans. Inf. Theory* 24.5 (1978), pp. 530–536. DOI: 10.1109/TIT.1978.1055934.

[15] Jacob Ziv and Abraham Lempel. “On the Complexity of Finite Sequences”. In: *IEEE Transactions on Information Theory* 22.1 (1976), pp. 75–81. DOI: 10.1109/TIT.1976.1055501.

Acronyms.

AC aritmethic coding. 13

ACL average code length. 5

BWT Burrows-Wheeler Transform. 22

LZ Lempel-Ziv. 27

UD uniquely decodable codes. 3