Some notes for the course

ALGORITHMIC INFORMATION THEORY

Armando J. Pinho

Important note*: This document contains a summary of most of the topics addressed in the course "Algorithmic Information Theory". As such, it should be regarded as a guide and a complement, and by no means the only source of information nor a substitute of the classes.*

(Draft version of February 1, 2023)

Contents

1 Introduction

What is "Algorithmic information theory"? According to the wikipedia as of February 2023:

> *"**Algorithmic information theory (AIT)** is a branch of theoretical computer science that concerns itself with the relationship between **computation** and **information** of computably generated objects (as opposed to stochastically generated), such as strings or any other data structure."*

Asking the GPT chat "What is algorithmic information theory?" on February 1st 2023, produced the following answer (Be careful! Do not believe everything it says...):

> *Algorithmic information theory is a branch of computer science and mathematics that studies the relationship between computation, information, and randomness. It deals with the question of how much information can be generated by an algorithm, how much randomness can be generated by an algorithm, and how information and randomness can be compressed and decompressed. The theory provides a foundation for the study of data compression, cryptography, and the limits of computation and artificial intelligence.*

In the remainder of this section, there are some questions that, in some way, have to do with the area of algorithmic information theory. Some of them are related to the area of computation, whereas some others involve the question of what is information and how should we measure it. Although it is not expected that you have answers for all of them right now, you should nevertheless start thinking about them.

Discussion topic 1.1
Consider the following questions and try to sketch answers for them:

- *What is data?*
- *How to measure the quantity of data?*
- *What is information?*
- *How to measure the quantity of information?*
- *What is a computation?*
- *Are there problems that computers cannot solve?*

Discussion topic 1.2
Consider the following statement:

> *One bit of **data** contains, at most, one bit of **information**.*

Do you think this is true? Why?

Discussion topic 1.3
Consider the following three binary strings:

1. $0101010101010101010101010101010101010101010101010101010101010101$
2. $01101010000001001111001100110011111110011101111001100100100001000$
3. $1101111001110101111101101111101110101101111000101110010100111011$

What would you say about each one of them?

Discussion topic 1.4
*Alice told Bob that she found an image compression software that is able to reduce the size of **every** image in, at least, 50% of its original size—for example, if originally the image occupies* $1\,000\,000$ *bytes, after compression it will require at most* $500\,000$ *bytes.*

*Bob was not very impressed by Alice's statement. However, when Alice added that she would also be able to **always** recover the original image from its compressed version, Bob immediately replied:*

> *"THAT IS IMPOSSIBLE! Every lossless (i.e., reversible) compression method is limited, i.e., it cannot compress all messages!"*

Is Bob correct? Why?

Discussion topic 1.5
Consider the following game, which is a case of a Post canonical system[1]*. The objective of the game is to discover the sequence of rules (among a previously specified set) that allows transforming a certain word into another word.*

Consider, for example, the case where the words can be formed using only the letters M, I and U, and that the allowed transformation rules are:

1. $xI \rightarrow xIU$
2. $Mx \rightarrow Mxx$
3. $xIIIy \rightarrow xUy$

[1]This problem can be reformulated as a string rewriting system, also known as a semi-Thue system, which is related to the notion of unrestricted grammar.

4. $xUUy \rightarrow xy$

Let us try some transformations:

1. *Give the steps to transform MI into MUI.*
2. *Can you transform UIM into MIU? Why?*
3. *And MIU into UIM?*
4. *What are the steps to transform MI into MU?*

Can you think of an algorithm that answers questions of the type "Is it possible to transform the word w into the word z, using a certain set of rules"?

Discussion topic 1.6
Consider the set of all functions $f : \mathbb{N} \rightarrow \{0, 1\}$. Alice told Bob that there are functions in that set that cannot be calculated by any finite program, regardless of the programming language used to implement it. Bob thinks Alice is wrong. Is she wrong? Why?

Discussion topic 1.7
Alice wants to play a game with Bob, where a coin has to be tossed. To show that she is using a fair coin, she tosses it twenty times. However, in all trials, heads comes up. Can Bob trust this coin? Why?

Would it be different if the sequence of outcomes had been "00101001110101010110"? Why?

Discussion topic 1.8
Consider the following procedure, suggested by John von Neumann (1903–1957):

1. *Toss a coin twice*
2. *If the results match, start over, forgetting both results*
3. *If the results differ, use the first result, forgetting the second,*

with which he claimed to be possible to obtain the equivalent to a sequence generated by a fair coin, even if the coin is unfair.

Do you agree with him? Why?

Discussion topic 1.9
You work at a company that sells large sequences of random numbers. Your boss thinks he needs to have some way of testing the "randomness" of the sequences before they are sent to

the clients (let us consider this as a kind of quality control of the production. . .). Because you are the best informatics engineer of the company, he asks you to develop a program for testing the randomness of arbitrary sequences.

What should you answer him? Have you any ideas of how to do it? Can you define "randomness"?

Discussion topic 1.10
John von Neumann once said:

> *"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."*

What do you think he meant to say with this statement?

Discussion topic 1.11
Consider the following definition of a certain number:

"The smallest positive integer not definable in fewer than twelve words".

Does this number exit? Why?

(This is known as Berry's paradox.)

Discussion topic 1.12
You are at a party with friends. After some drinks, one of them says:

> *"I've had a great idea! I'll write a program that will be capable of analysing other programs and tell us if they are bug-free or not! I wonder why no one have done this before!"*

What would you say to your friend?

Discussion topic 1.13
Consider the following segment of pseudo-code:

```
F(uint x) {
   while(x > 1) {
      if(x % 2 == 0)
         x = x / 2
      else
         x = 3 * x + 1
   }

   return
}
```

Does function F return for all x?
(This is known as Collatz's conjecture.)

2 A combinatorial measure of information

Problem 2.1
Alice is tossing a coin repeatedly. She wants to send the outcomes to Bob. To send the ***message****, Alice is only able to use a flashlight, by turning it on and off once per second.*

1. *Suppose that Alice and Bob agreed on the size of the message, n, beforehand, i.e., on how many coin tossing outcomes will be sent in each message. Propose a procedure for this transmission scheme.*
2. *Suppose now that Alice intends to send several messages of lengths, n_1, n_2, $\dots$, not known in advance by Bob. Can you suggest a transmission procedure that works in this case?*

Problem 2.2
Alice wants to send a letter to Bob, using the same transmission scheme. Consider an ***alphabet*** *of 27* ***symbols*** *(26 letters plus the space). How can this be done?*

Problem 2.3
Now Alice wants to play a game with Bob (they are finally face to face!). She thinks of a number between 1 and 100. Bob is allowed to ask every question he wants, but only gets a yes or no answer.

1. *What is the minimum number of questions that Bob needs to sequentially pose to Alice (the next question is posed only after knowing the answer to the previous question), in order to discover the number Alice thought of? What should those questions be?*
2. *Now Bob is only allowed to pose all the questions at once. Is the minimum number of required questions the same? In this case, what should be the questions?*

The minimum number of sequential questions is attained, for example, by successively halving the set of possibilities (a kind of binary search...). Taking as example the game of guessing the number, we have

$$2^b = 100,$$

where b is the number of questions. Hence,

$$b = \log_2 100 \approx 6.644,$$

meaning that, on average, about 6.644 questions are needed to find the number—in some cases six questions are enough, whereas in other cases seven questions are needed.

Example 2.1
Guess a number between 1 *and* 100*:*

$$\begin{gathered}\textit{Is } x \geq 50? \to 1\\ \textit{Is } x \geq 75? \to 0\\ \textit{Is } x \geq 62? \to 1\\ \textit{Is } x \geq 68? \to 0\\ \textit{Is } x \geq 65? \to 0\\ \textit{Is } x \geq 63? \to 1\\ \textit{Is } x = 63? \to 0\end{gathered}$$

Hence, x *has to be* 64 *(required seven yes/no questions).*

Example 2.2
Guess a number between 1 *and* 100*:*

$$\begin{gathered}\textit{Is } x \geq 50? \to 1\\ \textit{Is } x \geq 75? \to 0\\ \textit{Is } x \geq 62? \to 1\\ \textit{Is } x \geq 68? \to 0\\ \textit{Is } x \geq 65? \to 1\\ \textit{Is } x \geq 66? \to 0\end{gathered}$$

Hence, x *has to be* 65 *(in this case, six yes/no questions were enough).*

Problem 2.4
Using this searching scheme, the average number of questions needed is 6.72*. Verify this value (by calculating how many of the 100 numbers require the seven questions to be defined). Why is this value larger than the theoretical bound of* $\log_2 100 \approx 6.644$ *bits?*

Now imagine that x *is represented in binary. What should be the "natural" questions to pose in this case? Using this approach, can you attain an average number of questions smaller than* 6.72*?*

Notice that, if all the questions had to be provided at once, then in the example of guessing a number between 1 and 100 we had to ask always a minimum of $\lceil \log_2 100 \rceil = 7$ questions.

Problem 2.5
Suppose now that the number to be guessed is between 1 *and* 3*. Compare and discuss the impact on the average number of questions that have to be posed if:*

(a) Questions are posed sequentially, after knowing the previous answers;

(b) The questions are provided in batch.

Discuss also aspects such as the case where the numbers are not equally probable and also the difference between $\log_2 3 \approx 1.585$ *and the average number of questions estimated for the several scenarios addressed.*

Programming 2.1
Write a program that generates the minimal sequence of yes/no answers that allows discovering a certain number between 1 *and* n*.*

Problem 2.6
Using the ideas above, show that, in general, sorting has an average case lower bound of $\Omega(n \log n)$ *comparison operations.*

Problem 2.7
Because the game was getting boring, Alice says to Bob that she might start lying in, at most, one of the yes/no questions she is answering. Can you help Bob finding out the minimum number of additional questions he needs to pose in order to compensate for a possible wrong answer of Alice?

In order to be able to correct a possible error, we may use the simple scheme of posing the same question three times and then decide by majority (in this case, we should use a total of 21 questions). This approach is known as a **repeat code**. If the questions are posed sequentially, it is easy to see that this number can be reduced, in the worst case, to 15 (How?). Moreover, even if all questions are posed at once, it is possible to solve the problem with only 11 questions!

To see how this can be done, we have first to be convinced that one-bit error correction is possible if each **codeword** differs at least in three bits from all other possible codewords (i.e., the **Hamming distance** between codewords is at least three). So, each codeword needs to have a "forbiden region" around it of radius one (known as a radius-one ball):

- A n-bit string has $n+1$ strings at Hamming distance less or equal than one.
- For k bits of data, we need 2^k codewords and, hence, 2^k disjoint balls.
- This requires $2^k(n+1)$ distinct n-bit strings.
- Therefore,

$$2^n \geq 2^k(n+1), \quad \text{and} \quad k \leq n - \log_2(n+1).$$

In the case of our problem ($k = 7$), this condition is satisfied for $n = 11$. An effective code can be constructed using ideas initially proposed by Richard Hamming (1915–1998) (Hamming, 1950). Although without entering into much detail, we can see the **Hamming codes** as build by introducing parity bits (computed according to specific rules) in the positions of the codeword that are powers of two. In our case, the codewords would have the form

$$\boxed{p_1|p_2|d_1|p_3|d_2|d_3|d_4|p_4|d_5|d_6|d_7}$$

where the p_i are the parity bits and the d_i the data bits[2], originating a code with a rate of $7/11 \approx 0.64$.

Problem 2.8

Three players wearing hats enter a room. The hats can be black or white and are assigned by flipping a fair coin. The rules of the game are:

- *No player can see his own hat, but can see the hat of every other player.*
- *When in the room, the players cannot communicate. However, before having the hats, they are allowed to decide on a game strategy.*
- *Each player can either announce his guess regarding his hat color or pass.*
- *All players will do the announcement or pass simultaneously.*

The group wins the game if at least one person guesses correctly and no one guesses incorrectly.

What is the best strategy to give the group the highest possible probability of winning the game? Can you relate this problem with a Hamming code of length three?

With $n = 3$, we can form $2^3 = 8$ different 3-bit strings. Suppose that we pick the strings 000 and 111 as the possible codewords. Then, the set of 8 3-bit strings form a Hamming$(3, 1)$ code (verify this). If each of the 3-bit strings occurs with equal probability, then there is a $1/4$ probability of generating a correct codeword (2 out of 8) and a $3/4$ probability of generating a erroneous codeword (6 out of 8). Therefore, the strategy for the game is the following:

1. If the player sees the other two players with the same hat color, then he assumes that he has a different hat color, because this would be the most probable case. Hence, he announces that color.

[2]Of course, the position of the parity bits is arbitrary, provided that both the encoder and decoder know their positions. For example, the Hamming$(7, 4)$ can be obtained using

$$d_1, d_2, d_3, d_4, d_2 \oplus d_3 \oplus d_4, d_1 \oplus d_3 \oplus d_4, d_1 \oplus d_2 \oplus d_4.$$

2. If the player sees the other two players with different hat colors, then, since in that case the probability of his hat having any of the two colors is the same, he passes.

Notice that, using this strategy, the probability of choosing the wrong color is still 50% each time someone announces a color. The gain is that the wrong guesses are not evenly distributed—from the total of twelve announcements associated to the eight combinations, the six that are wrong are concentrated in just two cases; all other six cases are error free.

Homework 2.1
Try to find out what is the highest probability of winning the game when there are seven players. Hint: use the Hamming$(7,4)$ code. What should be the game strategy in this case?

The problems that we have been addressing consider a measure of information that is usually called **combinatorial**, because it is only concerned with the number of possible objects involved, assuming that they occur with equal probability. Hence, for m distinct objects (a size-m alphabet), the amount of **combinatorial information** needed to specify each one is

$$\log_2 m \quad \text{bits.}$$

This approach for measuring information can be traced back at least to the works of Nyquist and Hartley:

- Harry Nyquist (1889–1976), *Certain Factors Affecting Telegraph Speed*, Bell System Technical Journal (Nyquist, 1924):

  > *"This paper considers two fundamental factors entering into the maximum speed of* ***transmission of intelligence*** *by telegraph. These factors are signal shaping and choice of codes."*

- Ralph Hartley (1888–1970), *Transmission of Information*, Bell System Technical Journal (Hartley, 1928):

  > *"A quantitative measure of "information" is developed which is based on physical as contrasted with psychological considerations."*

In his paper, Hartley proposes measuring the information content, H, of a message as

$$H = n \log m = \log m^n,$$

where m denotes the number of possible symbols (the size of the alphabet) and n the number of symbols in the message. So, Hartley defined the amount of information in a message as the logarithm of the number of possible messages (message space).

The base of the logarithm determines the unit of information used: **hartley** for base 10, **nat** for base e, **bit** for base 2.

Problem 2.9
Alice has a deck of playing cards (52 cards). After shuffling the deck, Alice wants to send enough information to Bob in order for him to put his deck in the same order. How many bits does Alice need to send to Bob to communicate this information? Can you propose a method for doing it?

3 A probabilistic measure of information

Let us return to Alice and Bob and consider the following new problem:

Problem 3.1
Using the flashlight communication system, Alice wants to communicate to Bob the results of throwing a pair of dice (i.e., a number from 2 to 12). However, Alice is running out of batteries for the flashlight and, therefore, she wants to preserve them as much as possible. So, she wonders if, in this case, it is possible to use, on average, less than $\log_2 11 \approx 3.46$ *bits for sending to Bob the outcome of each trial... Can you help her solving this problem?*

The first aspect to take into consideration is that the possible outcomes of rolling two dice do not occur with the same frequency. Let us denote by Σ the alphabet representing those outcomes, i.e., $\Sigma = \{2, 3, \ldots, 12\}$. For example, there is only one combination for getting a twelve, whereas for getting a seven there are six possible combinations. Hence, Alice will have to transmit symbol "7" about six times more often than the symbol "12" of the alphabet.

The key idea is to use shorter representations, i.e., less bits, for the most frequent symbols.

The Morse code (19th century) is a good example of this principle, recognized well before the foundations of the theory of information were established by Claude Shannon (1916–2001) in the mid of the 20th century:

- Claude Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal (Shannon, 1948):

  > *"The recent development of various methods of modulation such as PCM and PPM which exchange bandwidth for signal-to-noise ratio has intensified the interest in a general theory of communication. A basis for such a theory is contained in the important papers of Nyquist and Hartley on this subject. In the present paper we will extend the theory to include a number of new factors, in particular the effect of noise in the channel, and the savings possible due to the* ***statistical structure of the original message*** *and due to the nature of the final destination of the information."*

Discussion topic 3.1
The Morse code is composed of sequences of dots and dashes of various lengths, according to the convention shown in Fig. 3.1. In this code, the characters of a word need to be separated by additional space (three units of time, where one unit of time is equivalente to the length of a dot). For example, the letters "ET" originate the sequence "·   —", to distinguish from the letter "A" that is encoded as "· —". Do you think this explicit separation could be avoided? If you think so, how should the code be constructed?

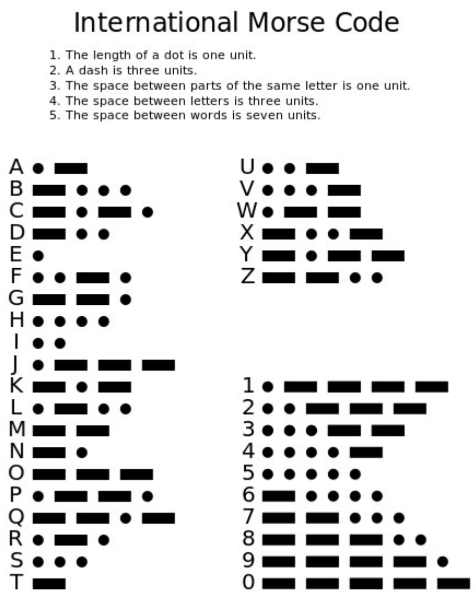

International Morse Code
1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.
A • ▬
B ▬ • • •
C ▬ • ▬ •
D ▬ • •
E •
F • • ▬ •
G ▬ ▬ •
H • • • •
I • •
J • ▬ ▬ ▬
K ▬ • ▬
L • ▬ • •
M ▬ ▬
N ▬ •
O ▬ ▬ ▬
P • ▬ ▬ •
Q ▬ ▬ • ▬
R • ▬ •
S • • •
T ▬
U • • ▬
V • • • ▬
W • ▬ ▬
X ▬ • • ▬
Y ▬ • ▬ ▬
Z ▬ ▬ • •
1 • ▬ ▬ ▬ ▬
2 • • ▬ ▬ ▬
3 • • • ▬ ▬
4 • • • • ▬
5 • • • • •
6 ▬ • • • •
7 ▬ ▬ • • •
8 ▬ ▬ ▬ • •
9 ▬ ▬ ▬ ▬ •
0 ▬ ▬ ▬ ▬ ▬

Figure 3.1: Morse code table.

3.1 About codes

Consider an alphabet with m symbols, $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$, and the corresponding **estimated frequencies of occurrence** of each symbol, p_i. Of course,

$$\sum_{i=1}^{m} p_i = 1.$$

If symbol $\sigma_i \in \Sigma$ is represented by a codeword w_i, then the objective is to **minimize**

$$\sum_{i=1}^{m} p_i l_i, \tag{3.1}$$

where $l_i \in \mathbb{N}^+$ is the length of the codeword w_i, i.e., its number of bits. Often, we will represent the length of a binary string w_i by $|w_i|$, i.e., $l_i = |w_i|$.

Obviously, we need to impose additional constraints to the minimization of (3.1), otherwise we could arrive at completely useless solutions, such as $l_i = 1, \forall_i$...

Discussion topic 3.2
Can you suggest restrictions that should be imposed in the minimization of (3.1)?

Of course, one of the restrictions should be that $w_i \neq w_j, \forall_{i \neq j}$, otherwise the code is not reversible, i.e., it is not possible to recover the original message. But this condition is not enough for reversibility (can you see why?). In fact, we need a stronger condition: The code should be **uniquely decodable**. This implies that two different messages will always have two different encodings. Recall the encoding of "ET" and "A" using the Morse code of Fig. 3.1 and the need for additional space to separate the letters. Without it, two different messages would originate the same encoding, i.e., "· —".

Note that, for our purposes, an **encoding** is the assignment of binary strings to elements of the alphabet. Some encodings are of **fixed length** (the ASCII encoding is an obvious example), whereas other encodings are of **variable length**. To attain compression, we need the latter ones.

Problem 3.2
Find codes for which $w_i \neq w_j, \forall_{i \neq j}$*, but that are not uniquely decodable.*

A sufficient (but not necessary) condition for unique decodability is **immediate decodability**. This means that symbol σ_i is immediately determined as soon as the last (i.e., the rightmost) bit of w_i is read during decoding. For this to happen, the codewords need to be **prefix-free**, i.e., a shorter codeword cannot be a prefix of a longer codeword (why do we need this requirement?).

Problem 3.3
Show that the Morse code is not a ***prefix-free code*** *(Suggestion: try to build a decoding tree). What are the practical implications of this limitation of the Morse code?*

Symbol	Probability	Code 1	Code 2	Code 3	Code 4
σ_1	0.5	0	0	0	0
σ_2	0.25	0	1	10	01
σ_3	0.125	1	00	110	011
σ_4	0.125	10	11	111	0111
Average length		1.125	1.25	1.75	1.875

Table 1: Example of several codes, with different properties.

Table 1 shows four different codes for a 4-symbol alphabet (although not all of them are useful...). They have different properties, namely:

- Code 1 is not uniquely decodable.
- Code 2 is also not uniquely decodable.
- Code 3 is uniquely decodable and instantaneous.

- Code 4 is uniquely decodable, but not instantaneous.

Problem 3.4
Consider now the following uniquely decodable, but not instantaneous, code:

Symbol	*Codeword*
σ_1	0
σ_2	01
σ_3	11

How to decode the string "011111111111111111*"?*
(Note that the first symbol can be either σ_1 or σ_2…)

Problem 3.5
Consider now the following code:

Symbol	*Codeword*
σ_1	0
σ_2	01
σ_3	10

How to decode the string "$010101010101010101 0$*"?*
(This code is neither instantaneous nor uniquely decodable)

Algorithm 3.1 (Unique decodability)
Suppose that we have two binary codewords a and b, with $k = |a|$, $n = |b|$, with $k < n$, and where a is a prefix of b. We refer to the last $n - k$ bits of b as the dangling suffix.

The algorithm for finding if a code is uniquely decodable is the following:

- *Construct a set with all the codewords.*
- *For every pair of codewords, generate the dangling suffix (if it exists) and add it to the set.*
- *Repeat until:*
  (a) You get a dangling suffix that is a codeword;
  (b) There are no more unique dangling suffixes.

If the algorithm terminates with condition (a), i.e., if it finds a dangling suffix that is a codeword, then the code is not uniquely decodable.

Example: $\{0, 01, 10\} \rightarrow \{0, 01, 10, 1\} \rightarrow \{0, 01, 10, 1, 0\}$.

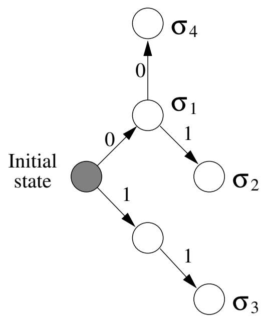


Figure 3.2: Example of a decoding tree of a non instantaneous (and also non uniquely decodable) code.

If the algorithm terminates with condition (b), i.e., if there are no more unique dangling suffixes, then the code is uniquely decodable.

Example: $\{0, 01, 11\} \rightarrow \{0, 01, 11, 1\}$.

As mentioned, if, in a certain encoding, none of the codewords is a prefix of another codeword, then the code is necessarily uniquely decodable. An easy way of checking if a code is prefix-free is to build the corresponding rooted binary tree. In a prefix-free code, the codewords are only associated with the external nodes of the tree.

Example 3.1
Consider an alphabet with symbols σ_1, σ_2, σ_3 and σ_4, and the following codeword assignment: $\sigma_1 \rightarrow 0, \quad \sigma_2 \rightarrow 01, \quad \sigma_3 \rightarrow 11, \quad \sigma_4 \rightarrow 00.$

If the decoder receives "0001", then it cannot determine if the encoder has sent the string "$\sigma_1\sigma_1\sigma_2$" or "$\sigma_4\sigma_2$". As can be seen using the corresponding decoding tree (Fig. 3.2), this code has several problems:

- *The path to node σ_3 is done by an intermediate node that does not contain a bifurcation: the code is inefficient.*
- *Node σ_1 is not a terminal node. The paths to nodes σ_2 and σ_4 include node σ_1: the code is not instantaneous (in fact, it is also not uniquely decodable, as can be seen using Algorithm 3.1).*

Example 3.2
Let us now consider that each branch of the tree connects to a terminal node or to a decision node that leads to a terminal node. Based on this restriction, we arrive at the following codeword assignment: $\sigma_1 \rightarrow 0, \quad \sigma_2 \rightarrow 10, \quad \sigma_3 \rightarrow 110, \quad \sigma_4 \rightarrow 111.$ *As can be confirmed (see Fig. 3.3), this code is uniquely decodable (and also instantaneous).*

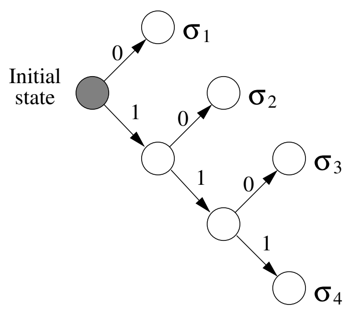


Figure 3.3: Example of a decoding tree of a uniquely decodable (an instantaneous) code. This code is prefix-free.

Another way of verifying if a code is prefix-free is to map the codewords in the unit interval, where each codeword occupies a non-overlapping fraction of length 2^{-l_i} of the unit interval (note that when a codeword of length k is used, a 2^{-k} fraction of the associated binary tree is "taken", because of the prefix-free condition).

Problem 3.6
Suppose that Alice wants to build a prefix-free code such that $l_1 = l_4 = 2$, $l_2 = l_3 = 3$ and $l_5 = 1$. Can you show to Alice that this is not possible? (Try both analogies: the binary tree and the unit interval covering)

Discussion topic 3.3
Given a set of lengths, l_i, in increasing order, discuss strategies for covering the unit interval as efficiently as possible. Can the covering be done as efficiently as previously even if the lengths are given in an arbitrary order?

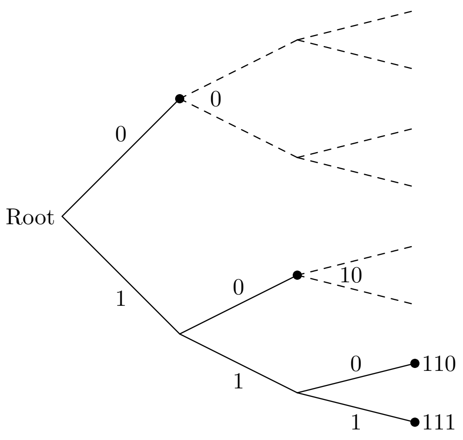


Figure 3.4: Tree associated to a prefix-free code.

Theorem 3.1 (Kraft inequality)
For any instantaneous (i.e., prefix-free) binary code, the codeword lengths $l_1, l_2, \ldots, l_m$ must satisfy the inequality

$$\sum_{i=1}^{m} 2^{-l_i} \leq 1.$$

Proof. Referring to the code tree of Fig. 3.4, consider all nodes at level $l_{\max}$ (the length of the longest codeword). A codeword at level l_i has $2^{l_{\max}-l_i}$ descendants at level $l_{\max}$. Because of the prefix-free condition, each of these descendant sets must be disjoint. Also, the total number of nodes in these sets must be less than or equal to $2^{l_{\max}}$. Therefore, summing over all the codewords, we have $\sum 2^{l_{\max}-l_i} \leq 2^{l_{\max}}$ or $\sum 2^{-l_i} \leq 1$. □

Theorem 3.2 (Counterpart of the Kraft inequality)
If a set of lengths $l_1, l_2, \ldots, l_m$ satisfy the Kraft inequality, then there exists an instantaneous code with these word lengths.

Proof. Construct a tree of depth $l_{\max}$. Label the first node (lexicographically) of depth l_1 as codeword 1 and remove its descendants from the tree. Then label the first remaining node of depth l_2 as codeword 2, and so on. Proceeding this way, we construct a prefix code with the specified $l_1, l_2, \ldots, l_m$. □

3.2 Definition of information according to Shannon

The definition of **information** according to Shannon is associated with the probability of certain events (the symbols). Consider E an event, i.e., a set of outcomes of some random experiment. If $P(E)$ is the probability of event E to occur, then the information (also called by Shannon "self-information") associated with the occurrence of E is given by

$$i(E) = \log_b \frac{1}{P(E)} = -\log_b P(E).$$

Recall that $\log(1) = 0$ and that $-\log(x)$ increases as x decreases from one to zero. Hence, an event with probability one does not carry any information. On the other hand, as the probability of the event approaches zero, the information that can be associated with that event approaches infinity.

Moreover, the information associated with the occurrence of two **independent** events, E_1 and

E_2,[3] is $i(E_1, E_2) = i(E_1) + i(E_2)$, because

$$\begin{aligned} i(E_1, E_2) &= -\log_b P(E_1, E_2) = -\log_b P(E_1)P(E_2) = \\ &= -\log_b P(E_1) - \log_b P(E_2) = i(E_1) + i(E_2). \end{aligned}$$

As mentioned before, the unit of information depends on the base of the logarithm, b: if $b = 2$, we measure information in bits; if $b = e$, in nats; if $b = 10$, in hartleys. From now on, when the base of the logarithm is not specified, we assume $b = 2$.

For a certain partition of the sample space, S, in E_j sets, i.e., if

$$\bigcup_j E_j = S \quad \text{and} \quad E_j \cap E_k = \emptyset, \forall j \neq k,$$

the associated **average self-information** is given by

$$H = \sum_j P(E_j) i(E_j) = -\sum_j P(E_j) \log P(E_j),$$

more usually called **entropy**. Note that the entropy is completely defined by a probability distribution—hence the name "**probabilistic**" in this definition of information.

Example 3.3
Consider tossing one fair coin. In this case, we have:

- *Two symbols (i.e.,* $\Sigma = \{heads, tails\}$*),* $P_i = 0.5$
- $H = -(0.5 \log 0.5 + 0.5 \log 0.5) = 1$
  $\Longrightarrow$ *One bit for each trial*
- $R = \log |\Sigma| - H = \log 2 - 1 = 0$
  This is called ***redundancy*** *and measures the* ***difference between the combinatorial and probabilistic information measures****.*

Example 3.4
The case of two fair and independent coins:

- *Four symbols,* $P_i = 0.25$
- $H = -\sum_{i=1}^{4} P_i \log P_i = 2$ *bits*

[3]Recall that if E_1 and E_2 are independent, then $P(E_1, E_2) = P(E_1)P(E_2)$.

- $R = \log|\Sigma| - H = \log 4 - 2 = 0$

Example 3.5
Two independent, but biased, coins:

- *Four symbols,* $P_1 = 0.5625$*,* $P_2 = P_3 = 0.1875$ *and* $P_4 = 0.0625$
- $H \approx 1.62$ *bits*
- $R = \log|\Sigma| - H = \log 4 - 1.62 = 0.38$
- *In this case, representing each symbol, on average requires only* 1.62 *bits.*
- *This means that, for example,* 1000 *outcomes of the process can be represented by approximately* 1620 *bits, instead of* 2000 *bits, as suggested by the combinatorial definition of information.*
- *As already mentioned, this data reduction can be obtained using* ***variable-length codes****.*

Shannon demonstrated that, if an **information source** produces events with probabilities $P(E_i)$, where $\{E_i\}$ is a partition of S, then, **on average**, it is not possible to use less than H bits to represent each event.

Usually, and for simplicity, we label the events E_i with symbols from an alphabet Σ which is often a subset of the integers, for example, $\Sigma = \{1, 2, \ldots, m\}$. However, for the sake of generality, we will use the more generic $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ alphabet.

Let us assume that an information source produces sequences of symbols $\mathbf{X} = \{X_1, X_2, \ldots\}$,[4] with $X_i \in \Sigma$. The nth-order entropy, H_n, is defined as

$$H_n = -\frac{1}{n} \sum_{x_1=\sigma_1}^{\sigma_m} \cdots \sum_{x_n=\sigma_1}^{\sigma_m} P(X_{k+1} = x_1, \ldots, X_{k+n} = x_n) \log P(X_{k+1} = x_1, \ldots, X_{k+n} = x_n),$$

where $\{X_{k+1}, X_{k+2}, \ldots, X_{k+n}\}$ are sub-sequences of length n generated by the information source. Shannon showed that, for a **stationary source**,

$$H = \lim_{n\to\infty} H_n,$$

where H is the **entropy of the source**. Notice that, if the events are **independent and identically distributed** (i.i.d.), then

$$H = H_1 = H_n = -\sum_{x=\sigma_1}^{\sigma_m} P(X_k = x) \log P(X_k = x), \quad \forall_k, \tag{3.2}$$

i.e., the entropy and the 1st-order entropy of the source are the same.

[4]$\mathbf{X}$ is a discrete-time random process and, therefore, the X_i are random variables.

Problem 3.7
Show that (3.2) holds $\forall_n$*, if the events are independent and identically distributed.*

Generally, the entropy of an information source is not known. Therefore, we have to compute **estimates** for it.

3.3 Source modeling

Consider the following sequence of symbols, produced by a certain information source:

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

If we take it as a "good" representation of the statistics of the information source, then we can estimate the probabilities of the symbols as

$$P(1) = P(6) = P(7) = P(10) = 1/16$$
$$P(2) = P(3) = P(4) = P(5) = P(8) = P(9) = 2/16.$$

Assuming i.i.d., the entropy of this source is 3.25 bits.

However, if we transform the original sequence,

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

into

```
1 1 1 -1 1 1 1 -1 1 1 1 1 1 -1 1 1
```

we now have only two different symbols (-1 and 1), for which $P(1) = 13/16$ and $P(-1) = 3/16$. Now, the entropy is only 0.7 bits...

Discussion topic 3.4
We have seen that the entropy is a measure of the average of information produced by a source. According to the example above, apparently it is possible to change the amount of information produced by a source, using a reversible transformation (show that the transformation used above is, in fact, reversible).

Do you think this is indeed possible? [5]

Let us denote by $x_1^n = x_1 x_2 \ldots x_n,\ x_i \in \Sigma$, the sequence of outputs (symbols from the source alphabet Σ) that an information source has generated until instant n. Often, we refer to x_1^n (or

[5]In fact, it is not! So, what is wrong in this reasoning?

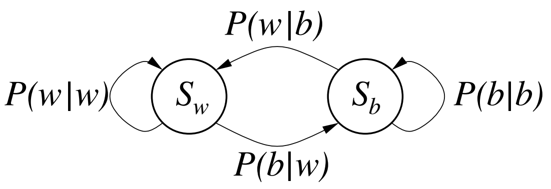


Figure 3.5: Example of a transition diagram of a Markov model for a binary image.

simply x, if size can be omitted) as a string over the alphabet Σ. The transformation that was performed in the example above corresponds to the observation that the relation

$$x_n = x_{n-1} + r_n$$

provides a "satisfactory" **model** for this information source. In that case, only the values of r_n need to be encoded. This model is called **static**, because it does not change with n. Otherwise, the model is called **adaptive**.

Discussion topic 3.5
How do we know that a certain model is "satisfactory"? For example, why is it resonable to say that the model above is "satisfactory"?

Correctly modeling an information source (and, therefore, **finding the underlying structure of the data**) is one of the most important aspects in data compression.

One of the most used aproaches for representing data dependencies relies on the use of **Markov models**. In lossless data compression, we use a specific type, called discrete time Markov chain or **finite-context model**.

A k-order Markov model verifies

$$P(x_n|x_{n-1}\dots x_{n-k}) = P(x_n|x_{n-1}\dots x_{n-k}\dots),$$

where the string $c = x_{n-1}\dots x_{n-k}$ is called the state or **context** of the process.

A 1st-order Markov model reduces to

$$P(x_n|x_{n-1}) = P(x_n|x_{n-1}x_{n-2}\dots).$$

As an example, consider that we want to model a binary image. We have two states, S_w (white pixel) and S_b (black pixel), and four possible transitions, namely $S_w \to S_w$, $S_w \to S_b$, $S_b \to S_w$, $S_b \to S_b$. The state transition diagram of this 1st-order model is shown in Fig. 3.5.

The entropy of a process with N states S_i is simply the average value of the entropy of each state, i.e.,

$$H = \sum_{i=1}^{N} P(S_i)H(S_i),$$

where $H(S_i)$ denotes the entropy of state S_i and P_i is the probability of occurrence of state S_i. Therefore, in our example,

$$H(S_w) = -P(b|w)\log P(b|w) - P(w|w)\log P(w|w).$$

Example 3.6
Consider a 1st-order model with

$$P(w|b) = 0.3 \quad \text{and} \quad P(b|w) = 0.01.$$

The state probabilities can be calculated according to

$$P(S_w) = \frac{P(w|b)}{P(w|b) + P(b|w)} \approx 0.968$$

and

$$P(S_b) = \frac{P(b|w)}{P(w|b) + P(b|w)} \approx 0.032.$$

Therefore, considering independence in the occurrence of symbols, we have,

$$H = -0.968\log_2 0.968 - 0.032\log_2 0.032 \approx 0.204 \textit{ bps}$$

However, considering the 1st-order Markov model, we obtain an estimate of the entropy of the source that is

$$H(S_b) = -0.3\log_2 0.3 - 0.7\log_2 0.7 \approx 0.881 \textit{ bps}$$
$$H(S_w) = -0.01\log_2 0.01 - 0.99\log_2 0.99 \approx 0.081 \textit{ bps}$$
$$H = 0.968 \times 0.081 + 0.032 \times 0.881 \approx 0.107 \textit{ bps}$$

Markov models are particularly useful in text compression, because the next letter in a word is generally heavily influenced by the preceding letters. In fact, the use of Markov models for written English appeared in the original work of Shannon. In 1951, he estimated the entropy of English to be in between about 0.6 and 1.3 bits per letter.

For simplicity, consider only 26 letters and the space character. The frequency of letters in English is far from uniform: the most common, "E", occurs about 13%, whereas the least common, "Q" and "Z", occur about 0.1% of the time. The frequency of pairs of letters is also nonuniform. For example, the letter "Q" is always followed by a "U". The most frequent pair is "TH" (occurs about 3.7%).

We can use the frequency of the pairs to estimate the probability that a letter follows any other letter. This reasoning can also be used for constructing higher-order models. However, the size of the model grows exponentially. For example, to build a third-order Markov model we need to estimate the values of $p(x_n|x_{n-1}x_{n-2}x_{n-3})$. This requires a table with $27^4 = 531\,441$ entries and enough text to correctly estimate the probabilities.

The following examples have been constructed using empirical distributions collected from samples of text (Shannon, 1948, 1951).

Example 3.7 (Zero-order approximation (Independent and equiprobable symbols))

XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGXYD QPAAMKBZAACIBZLHJQD

Entropy: $\log 27 = 4.76$ *bits per letter.*

Example 3.8 (First-order approximation (Independent, but with the correct $p(x)$))

OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA TH EEI ALHENHTTPA OOBTTVA NAH BRL

Estimated entropy: ≈ 4.03 *bits per letter.*

Example 3.9 (Second-order approximation (1st-order Markov model, i.e., $p(x_n|x_{n-1})$))

ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TUCOOWE AT TEASONARE FUSO TIZIN ANDY TOBE SEACE CTISBE

Estimated entropy: ≈ 3.32 *bits per letter.*

Example 3.10 (Third-order approximation (2nd-order Markov model, i.e., $p(x_n|x_{n-1}x_{n-2})$))

IN NO IST LAT WHEY CRATICT FROURE BERS GROCID PONDENOME OF DEMONSTURES OF THE REPTAGIN IS REGOACTIONA OF CRE

Estimated entropy: ≈ 3.1 *bits per letter (excluding spaces).*

Example 3.11 (First-order word approximation)

REPRESENTING AND SPEEDILY IS AN GOOD APT OR COME CAN DIFFERENT NATURAL HERE HE THE A IN CAME THE TO OF TO EXPERT GRAY COME TO FURNISHES THE LINE MESSAGE HAD BE THESE

Example 3.12 (Second-order word approximation)

THE HEAD AND IN FRONTAL ATTACK ON AN ENGLISH WRITER THAT THE CHARACTER OF THIS POINT IS THEREFORE ANOTHER METHOD FOR THE LETTERS THAT THE TIME OF WHO EVER TOLD THE PROBLEM FOR AN UNEXPECTED

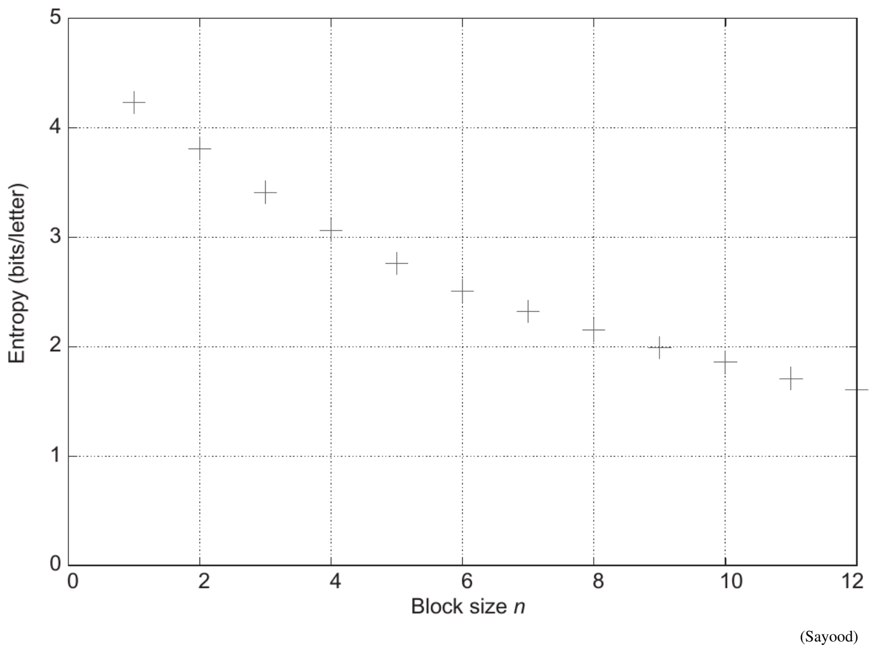


Figure 3.6: Estimated value of the entropy of a text using serveral orders.

Figure 3.6 shows the evolution of the estimate of the entropy of a book when the order of the model increaces.

The number of different contexts (or states of the Markov process) is given by $|\Sigma|^k$, where k is the order of the model, which clearly grows exponentially with the size of the alphabet. For example, in a 2-symbol alphabet, a finite-context model of depth 4 originates only $2^4 = 16$ states or different contexts. However, if the alphabet is of size 256 (for example, if we consider a gray-level image), then the number of contexts jumps to $256^4 = 4\,294\,967\,296!$

Often, the finite-context models are "learned" online, i.e., as the sequence of symbols is processed (note that in Example 3.6 the model is static, i.e., the probabilities are considered fixed during the operation of the model). In the case of online operation, since the probabilistic models are continuously adapting, after processing the first n symbols of x, the average number of bits associated with an order-k finite-context model is given by

$$H_n = -\frac{1}{n}\sum_{i=1}^{n} \log P(x_i|x_{i-k}^{i-1}). \tag{3.3}$$

Problem 3.8
Verify (3.3) and discuss the practical implications of x_{n-k}^{n-1} when $n < k$ (note that the convention is that $x_n^n = x_n$) and propose solutions to handle it.

In the case of online learning of the finite-context model, a table (or some other more sophisticated data structure, such as an hash-table) is used to collect counts that represent the number of times that each symbol occurs in each context. For example, suppose that, in a certain moment, a binary ($|\Sigma| = 2$) source is modeled by an order-3 finite-context model represented by the table

x_{i-3}	x_{i-2}	x_{i-1}	n_0	n_1
0	0	0	10	25
0	0	1	4	12
0	1	0	15	2
0	1	1	3	4
1	0	0	34	78
1	0	1	21	5
1	1	0	17	9
1	1	1	0	22

where n_i indicates how many times symbol "i" occurred following that context. Therefore, in this case, we may estimate the probability that a symbol "0" follows the string "100" as being $34/(34 + 78) \approx 0.30$.

This way of estimating probabilities, based only on the relative frequencies of previously occurred events, suffers from the problem of assigning probability zero to events that were not (yet) seen. That would be the case, in this example, if we used the same approach to estimate the probability of having a "0" following a string of three or more "1s".

Problem 3.9
Discuss the potential implications of the "zero-probability" problem and give suggestions to avoid it.

In 1774, Laplace (1749–1827) proposed a rule for estimating the probability that a given event succeeds, after occurring n_1 successes and n_2 failures in $n = n_1 + n_2$ trials, as

$$\frac{n_1 + 1}{n + 2}.$$

This formula is in fact correct if the trials are independent and if an uniform prior over the estimated probability is considered.

For the more general case of a beta prior, $B(\alpha_1, \alpha_2)$, the estimator adopts the form (G. F. Hardy, 1889)

$$\frac{n_1 + \alpha_1}{n + \alpha_1 + \alpha_2}.$$

The multinomial generalization of the formula leads to

$$\frac{n_i + \alpha}{n + \alpha|\Sigma|},$$

where, for simplicity, we consider $\alpha = \alpha_1 = \cdots = \alpha_{|\Sigma|}$, and the α_i are the parameters of the Dirichlet family of distributions.

A worth noting aspect of this estimator is that, defining

$$\mu = \frac{n}{n + \alpha|\Sigma|},$$

it can be written as

$$\mu\frac{n_i}{n} + (1-\mu)\frac{1}{|\Sigma|},$$

showing how it evolves from a uniform estimator to a frequency estimator, as n increases.

3.4 Some more on Shannon entropy

3.4.1 Entropy, joint entropy and conditional entropy

As already shown, given a probability distribution, we may define a quantity called **entropy**, that has many properties that agree with our intuitive notion of what a measure of information should be. We have seen that the entropy expresses the **amount of uncertainty** about a certain random variable. It measures the **amount of information** required **on average** to describe the random variable.

This notion can be extended to define **mutual information**, which is a measure of the amount of information that one random variable contains about another. In fact, the mutual information is a special case of a more general quantity called **relative entropy**, which gives a measure of the "distance" between two probability distributions.

Consider X a discrete random variable that takes values in a finite alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ and has a probability mass function $p(x) = P(X = x)$, $x \in \Sigma$.

The entropy of X can be interpreted as the **expected value** of the random variable $\log\dfrac{1}{p(X)}$, where X is drawn according to $p(x)$. Therefore,

$$H(X) = E\left[\log\frac{1}{p(X)}\right] = E[-\log p(X)],$$

where $E[g(X)]$ denotes the expected value of $g(X)$, defined as

$$E[g(X)] = \sum_{x\in\Sigma} g(x)p(x).$$

Example 3.13
Let X be a binary random variable with alphabet $\Sigma = \{0, 1\}$ and probability mass function $P(X = 1) = p$ and $P(X = 0) = 1 - p$. Then, the entropy of X is $H(X) = -p\log p - (1 - p)\log(1 - p)$ (see Fig. 3.7).

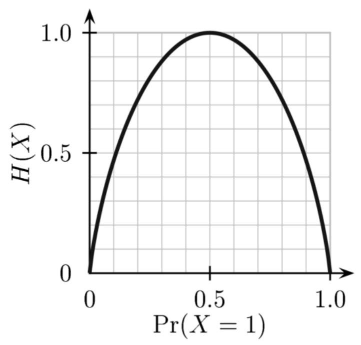


Figure 3.7: Entropy curve for a binary alphabet.

Theorem 3.3
$0 \leq H(X) \leq \log|\Sigma|$, *where* $|\Sigma|$ *denotes the cardinality of* Σ. *Also,* $H(X)=0$ *iff* $p(x)=1$ *for some* $x \in \Sigma$, *and* $H(X)=\log|\Sigma|$ *iff* $p(x)=1/|\Sigma|$.

Proof of $H(X) \geq 0$. Since each $p(x) \leq 1$, then each term $p(x)\log p(x) \leq 0$, implying $H(X) \geq 0$. Also, $H(X)=0$ requires that $p(x)\log p(x)=0, \forall x \in \Sigma$, implying $p(x)=1$ or $p(x)=0$.

Therefore, $H(X)=0$ iff $p(x)=1$ for some $x \in \Sigma$ (and, obviously, $p(x)=0$ for the other x). □

Proof of $H(X) \leq \log|\Sigma|$. Consider the inequality $\ln x \leq x-1$ and the subset $\Sigma' \subset \Sigma$ such that $p(x)>0, \forall x \in \Sigma'$. Then

$$H(X)-\log|\Sigma'| = \sum_{x\in\Sigma'} p(x)\log\frac{1}{p(x)} - \sum_{x\in\Sigma'} p(x)\log|\Sigma'| =$$

$$= \frac{1}{\ln 2}\sum_{x\in\Sigma'} p(x)\ln\frac{1}{|\Sigma'|p(x)} \leq \frac{1}{\ln 2}\sum_{x\in\Sigma'} p(x)\left[\frac{1}{|\Sigma'|p(x)}-1\right] =$$

$$= \frac{1}{\ln 2}\left[\sum_{x\in\Sigma'}\frac{1}{|\Sigma'|} - \sum_{x\in\Sigma'} p(x)\right] = 0.$$

The rest of the proof is left as exercise. □

We can extend the definition of entropy to a pair of random variables (X,Y) with joint distribution $p(x,y)=P(X=x,Y=y)$. For simplicity, we will assume that both random variables are drawn from the same alphabet Σ.

Definition 3.1 (Joint entropy)
The joint entropy $H(X,Y)$ of a pair of discrete random variables (X,Y), with a joint distribution $p(x,y)$, is defined as

$$H(X,Y) = -\sum_{(x,y)\in\Sigma^2} p(x,y)\log p(x,y) = E[-\log p(X,Y)].$$

Note that if $p(x,y) = p(x)p(y)$ (i.e., if X and Y are independent random variables), then $H(X,Y) = H(X) + H(Y)$. On the other hand, if X and Y are dependent, then $H(X,Y) < H(X) + H(Y)$.

Definition 3.2 (Conditional entropy)
For a pair of discrete random variables (X,Y), having a joint distribution $p(x,y)$, we define conditional entropy $H(Y|X)$ as

$$H(Y|X) = \sum_{x\in\Sigma} p(x)H(Y|X=x) = -\sum_{x\in\Sigma} p(x)\sum_{y\in\Sigma} p(y|x)\log p(y|x) =$$

$$= -\sum_{(x,y)\in\Sigma^2} p(x,y)\log p(y|x) = E[-\log p(Y|X)].$$

Theorem 3.4 (Chain rule for the entropy of a pair of random variables)

$$H(X,Y) = H(X) + H(Y|X).$$

Proof.

$$H(X,Y) = -\sum_{(x,y)\in\Sigma^2} p(x,y)\log p(x,y) = -\sum_{(x,y)\in\Sigma^2} p(x,y)\log p(x)p(y|x) =$$

$$= -\sum_{(x,y)\in\Sigma^2} p(x,y)\log p(x) - \sum_{(x,y)\in\Sigma^2} p(x,y)\log p(y|x) =$$

$$= -\sum_{x\in\Sigma} p(x)\log p(x) + H(Y|X) = H(X) + H(Y|X).$$

□

Example 3.14

Consider the following joint distribution of (X, Y):

$Y\backslash X$	1	2	3	4
1	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$
2	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{32}$
3	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$
4	$\frac{1}{4}$	0	0	0

The marginal distributions of X and Y are, respectively, $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ and $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$. Hence, $H(X) = 1.75$ bits and $H(Y) = 2$ bits. Also,

$$H(X|Y) = \sum_{i=1}^{4} P(Y = i)H(X|Y = i) = \frac{1}{4}H\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right) +$$

$$\frac{1}{4}H\left(\frac{1}{4}, \frac{1}{2}, \frac{1}{8}, \frac{1}{8}\right) + \frac{1}{4}H\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right) + \frac{1}{4}H(1, 0, 0, 0) = 1.375 \text{ bits.}$$

Similarly, $H(Y|X) = 1.625$ bits and $H(X, Y) = 3.375$ bits.

Note that, generally, $H(Y|X) \neq H(X|Y)$, but that $H(X) - H(X|Y) = H(Y) - H(Y|X)$.

3.4.2 Relative entropy

The relative entropy is a measure of the "distance" between two distributions. It expresses the inefficiency of assuming that the distribution is $q(x)$ when the true distribution is $p(x)$.

Definition 3.3 (Relative entropy)
The relative entropy or Kullback-Leibler distance between two probability mass functions, $p(x)$ and $q(x)$, is defined as

$$D(p\|q) = \sum_{x\in\Sigma} p(x) \log \frac{p(x)}{q(x)} = E_p \left[\log \frac{p(X)}{q(X)}\right].$$

Note that we consider the conventions: $0 \log \frac{0}{0} = 0$, $0 \log \frac{0}{q} = 0$ and $p \log \frac{p}{0} = \infty$.

Example 3.15
Let $\Sigma = \{0, 1\}$ and consider two distributions, $p(x)$ and $q(x)$, on the elements of Σ. Let $p(0) = 1 - r$, $p(1) = r$, $q(0) = 1 - s$ and $q(1) = s$. Then

$$D(p\|q) = (1 - r) \log \frac{1 - r}{1 - s} + r \log \frac{r}{s}$$

and

$$D(q\|p) = (1-s)\log\frac{1-s}{1-r} + s\log\frac{s}{r}$$

If $r = 1/2$ *and* $s = 1/4$*, then* $D(p\|q) = 0.208$ *bits,* $D(q\|p) = 0.189$ *bits.*

Note that, in general, $D(p\|q) \neq D(q\|p)$, and, if $p(x) = q(x)$, then $D(p\|q) = D(q\|p) = 0$.

3.4.3 Mutual information

The mutual information expresses the amount of information that one random variable contains about another rv.

Definition 3.4 (Mutual information)
Consider two random variables X and Y, with joint probability mass function $p(x,y)$ and marginal functions $p(x)$ and $p(y)$.

The mutual information, $I(X;Y)$, can be defined as the relative entropy between the joint distribution and the product $p(x)p(y)$, i.e.,

$$I(X;Y) = D(p(x,y)\|p(x)p(y)) =$$

$$= \sum_{(x,y)\in\Sigma^2} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} = E_{p(x,y)}\left[\log\frac{p(X,Y)}{p(X)p(Y)}\right].$$

Also, the mutual information indicates the reduction in the uncertainty of one random variable due to the knowledge of the other. In fact, we can write

$$I(X;Y) = \sum_{(x,y)\in\Sigma^2} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} = \sum_{(x,y)\in\Sigma^2} p(x,y)\log\frac{p(x|y)}{p(x)} =$$

$$= -\sum_{(x,y)\in\Sigma^2} p(x,y)\log p(x) + \sum_{(x,y)\in\Sigma^2} p(x,y)\log p(x|y) =$$

$$= -\sum_{x\in\Sigma} p(x)\log p(x) + \sum_{(x,y)\in\Sigma^2} p(x,y)\log p(x|y) =$$

$$= H(X) - H(X|Y) = H(Y) - H(Y|X) = H(X) + H(Y) - H(X,Y).$$

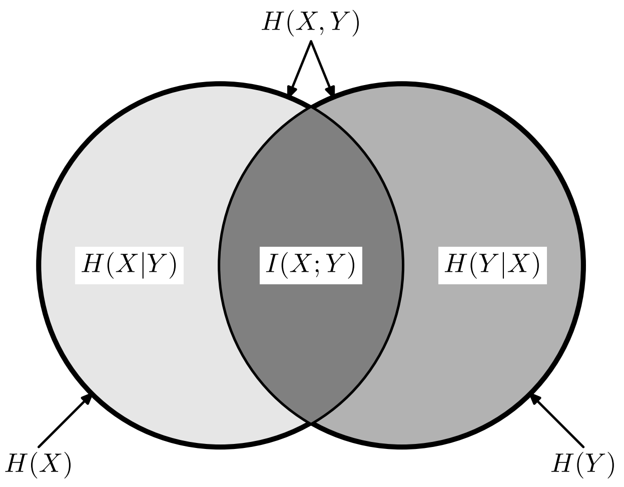


Figure 3.8: Some relations between entropy and mutual information.

Example 3.16
Considering again the joint distribution of (X, Y),

$Y\backslash X$	1	2	3	4
1	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$
2	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{32}$
3	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$
4	$\frac{1}{4}$	0	0	0

where $H(X) = 1.75$ bits, $H(Y) = 2$ bits, $H(X|Y) = 1.375$ bits and $H(Y|X) = 1.625$, we obtain a mutual information between X and Y of

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) = 0.375 \quad \text{bits.}$$

Theorem 3.5 (Relations between the mutual information and the entropy)
The following relations between the mutual information of a pair of random variables, X and Y, and the corresponding entropies hold (see also Fig. 3.8):

$$\begin{aligned}
I(X;Y) &= H(X) - H(X|Y) \\
I(X;Y) &= H(Y) - H(Y|X) \\
I(X;Y) &= H(X) + H(Y) - H(X,Y) \\
I(X;Y) &= I(Y;X) \\
I(X;X) &= H(X) - H(X|X) = H(X).
\end{aligned}$$

The mutual information of a random variable with itself is the entropy of the random variable (verify this). This is why the entropy is sometimes referred to as **self-information**.

Theorem 3.6 (Chain rule for the entropy)
Let $X_1, X_2, \dots, X_n$ be a set of random variables drawn according to $p(x_1, x_2, \dots, x_n)$. Then

$$H(X_1, X_2, \dots, X_n) = \sum_{i=1}^{n} H(X_i | X_{i-1}, \dots, X_1).$$

Proof.

$$H(X_1, X_2) = H(X_1) + H(X_2|X_1),$$
$$H(X_1, X_2, X_3) = H(X_1) + H(X_2, X_3|X_1) = H(X_1) + H(X_2|X_1) + H(X_3|X_2, X_1),$$
$$\vdots$$
$$H(X_1, X_2, \dots, X_n) = H(X_1) + H(X_2|X_1) + \cdots + H(X_n|X_{n-1}, \dots, X_1).$$

□

Definition 3.5 (Conditional mutual information)
The conditional mutual information of the random variables X and Y given Z is defined as

$$I(X;Y|Z) = H(X|Z) - H(X|Y,Z) = E_{p(x,y,z)}\left[\log \frac{p(X,Y|Z)}{p(X|Z)p(Y|Z)}\right].$$

(Remember that $I(X;Y) = H(X) - H(X|Y)$)

Theorem 3.7 (Chain rule for the information)

$$I(X_1, X_2, \dots, X_n; Y) = \sum_{i=1}^{n} I(X_i; Y | X_{i-1}, X_{i-2}, \dots, X_1).$$

Proof.

$$\begin{aligned}
I(X_1, X_2, \dots, X_n; Y) &= H(X_1, X_2, \dots, X_n) - H(X_1, X_2, \dots, X_n|Y) = \\
&= \sum_{i=1}^{n} H(X_i|X_{i-1}, \dots, X_1) - \sum_{i=1}^{n} H(X_i|X_{i-1}, \dots, X_1, Y) = \\
&= \sum_{i=1}^{n} I(X_i; Y|X_{i-1}, X_{i-2}, \dots, X_1).
\end{aligned}$$

□

Theorem 3.8 (Jensen's inequality)
If $f(x)$ is a convex function and X is a random variable,

$$E[f(X)] \geq f(E[X]).$$

Moreover, if $f(x)$ is strictly convex, the equality implies that $X = E[X]$ with probability 1, i.e., X is constant.

Theorem 3.9 (Information inequality)
Let $p(x)$ and $q(x)$ be two probability mass functions, with $x \in \Sigma$. Then, with equality iff $p(x) = q(x), \forall x$,

$$D(p\|q) \geq 0.$$

Proof of the non-negativity. Let $\Sigma' = \{x|p(x) > 0\}$ be the support set of $p(x)$. Then

$$-D(p\|q) = \sum_{x\in\Sigma'} p(x) \log \frac{q(x)}{p(x)} \leq \quad \text{(by Jensen inequality)}$$

$$\leq \log \sum_{x\in\Sigma'} p(x) \frac{q(x)}{p(x)} = \log \sum_{x\in\Sigma'} q(x) \leq \log \sum_{x\in\Sigma} q(x) = \log 1 = 0,$$

i.e., $D(p\|q) > 0$. □

Proof that $D(p\|q) = 0$ iff $p = q$. Since $\log x$ is a strictly concave function of x, we have

$$\sum_{x\in\Sigma'} p(x) \log \frac{q(x)}{p(x)} = \log \sum_{x\in\Sigma'} p(x) \frac{q(x)}{p(x)}$$

iff $q(x)/p(x)$ is constant everywhere (i.e., if $q(x) = cp(x), \forall x$). Therefore,

$$\sum_{x\in\Sigma'} q(x) = c \sum_{x\in\Sigma'} p(x) = c.$$

Moreover, it is required that $\sum_{x\in\Sigma'} q(x) = 1$ for having

$$\log \sum_{x\in\Sigma'} q(x) = \log \sum_{x\in\Sigma} q(x),$$

implying $c = 1$. Hence, we have $D(p\|q) = 0$ iff $p(x) = q(x), \forall x$. □

Corollary 3.9.1 (Nonnegativity of mutual information)
For any two random variables, X and Y,

$$I(X;Y) \geq 0,$$

with equality iff X and Y are independent.

Proof.

$$I(X;Y) = D(p(x,y)\|p(x)p(y)) \geq 0,$$

with equality iff $p(x,y) = p(x)p(y)$, i.e., when X and Y are independent. □

Theorem 3.10 (Upper bound on the entropy)

$$H(X) \leq \log |\mathcal{X}|,$$

with equality iff X has uniform distribution over $\mathcal{X}$.

Proof. Let $u(x) = 1/|\Sigma|$ be the uniform probability mass function over Σ, and let $p(x)$ be the probability mass function for X. Then,

$$D(p\|u) = \sum_{x\in\Sigma} p(x) \log \frac{p(x)}{u(x)} = \log|\Sigma| - H(X) \geq 0.$$

Therefore, $H(X) \leq \log|\Sigma|$, with equality iff $p(x) = u(x)$. □

Theorem 3.11 (Conditioning reduces entropy)

$$H(X|Y) \leq H(X)$$

with equality iff X and Y are independent.

Proof.

$$0 \leq I(X;Y) = H(X) - H(X|Y).$$

□

This theorem states that knowing another random variables, Y, can only reduce the uncertainty in X. However, this is only true **on average**. Specifically, $H(X|Y=y)$ may be greater than or less than or equal to $H(X)$, but, on average, $H(X|Y) = \sum_y p(y)H(X|Y=y) \leq H(X)$.

Example 3.17
Let (X,Y) have the following joint distribution,

$Y\backslash X$	1	2
1	0	$\frac{3}{4}$
2	$\frac{1}{8}$	$\frac{1}{8}$

Then, $H(X) = H(\frac{1}{8}, \frac{7}{8}) \approx 0.544$ *bits,* $H(X|Y = 1) = 0$ *and* $H(X|Y = 2) = 1$ *bit. However,*

$$H(X|Y) = \frac{3}{4}H(X|Y = 1) + \frac{1}{4}H(X|Y = 2) = 0.25 \quad \text{bits.}$$

Thus, the uncertainty in X *increases if* $Y = 2$ *is observed and decreases if* $Y = 1$*. Nevertheless, on average, the uncertainty decreases.*

Theorem 3.12 (Independence bound on the entropy)
Let $X_1, X_2, \ldots, X_n$ *be drawn according to* $p(x_1, x_2, \ldots, x_n)$*. Then,*

$$H(X_1, X_2, \ldots, X_n) \leq \sum_{i=1}^{n} H(X_i),$$

with equality iff the X_i *random variables are independent.*

Proof. By the chain rule for entropies and using the fact that conditioning reduces the entropy,

$$H(X_1, X_2, \ldots, X_n) = \sum_{i=1}^{n} H(X_i|X_{i-1}, \ldots, X_1) \leq \sum_{i=1}^{n} H(X_i),$$

with equality iff the X_i are independent. □

---

4 Data compression

4.1 Variable-length coding

4.1.1 Optimal codes

For a given source distribution p_i, we want to find the prefix-free code with the minimum expected length, i.e., we want to minimize

$$L = \sum p_i l_i,$$

over all integers $l_1, l_2, \ldots, l_m$ satisfying the Kraft inequality, i.e.,

$$\sum 2^{-l_i} \leq 1.$$

By relaxing the integer constraint on l_i, we obtain the optimal code lengths, l_i^*, using an optimization method[6], that gives

$$l_i^* = -\log p_i.$$

This noninteger choice of codeword lengths yields an average codeword length

$$L^* = \sum p_i l_i^* = -\sum p_i \log p_i = H.$$

Theorem 4.1
The average length L of any instantaneous binary code for a source distribution p_i is greater than or equal to the entropy H, i.e.,

$$L \geq H,$$

with equality iff $2^{-l_i} = p_i$.

Proof.

$$H - L = -\sum p_i \log p_i - \sum p_i l_i = \sum p_i \log 2^{-l_i} - \sum p_i \log p_i =$$

$$= \sum p_i \log \frac{2^{-l_i}}{p_i} \leq \log \left(\sum p_i \frac{2^{-l_i}}{p_i} \right) = 0,$$

where the inequality results from the use of Jensen's inequality. □

As shown, the choice of word lengths $l_i = -\log p_i$ yields $L = H$. However, in practice, these values may not be integers. Therefore, we might want to construct a code by rounding up the $-\log p_i$ lengths, i.e.,

$$l_i = \lceil -\log p_i \rceil.$$

[6]Consider, for example, that $l_i = -\log q_i$. Using Lagrange multipliers to solve this contrained optimization problem, we have to find the gradient of $-\sum p_i \log q_i + \lambda(\sum q_i - 1)$.

Note that these lengths satisfy the Kraft inequality, because

$$\sum 2^{-\lceil -\log p_i \rceil} \leq \sum 2^{-(-\log p_i)} = \sum p_i = 1.$$

Moreover, we can write

$$-\log p_i \leq l_i < -\log p_i + 1 \;\Rightarrow\; H \leq L < H + 1.$$

Note that the optimal code can only be better than this code.

Theorem 4.2
Let $l_1^*, l_2^*, \ldots, l_m^*$ be the optimal codeword lengths for a source distribution p_i and let L^* be the associated expected length of an optimal code, i.e., $L^* = \sum p_i l_i^*$. Then

$$H \leq L^* < H + 1.$$

Proof. We have seen that if $l_i = \lceil -\log p_i \rceil$ then

$$H \leq L = \sum p_i l_i < H + 1.$$

Since L^* is less than L and since $L^* \geq H$, we have the theorem. $\square$

In this last theorem, we have an overhead of at most one bit, due to the fact that $-\log p_i$ is not always an integer. We can reduce the overhead per symbol by spreading it out over many symbols. Therefore, let us consider a system in which we send strings of n symbols at once. Consider that L_n is the expected codeword length per input symbol, i.e., if $l(x_1 x_2 \cdots x_n)$ is the length of the codeword associated with string $x_1 x_2 \cdots x_n$, where $x_i \in \Sigma$, then

$$L_n = \frac{1}{n} \sum p(x_1 x_2 \cdots x_n) l(x_1 x_2 \cdots x_n) = \frac{1}{n} E[l(X_1, X_2, \ldots, X_n)].$$

Theorem 4.3
The minimum expected codeword length per symbol satisfies

$$\frac{H(X_1, X_2, \ldots, X_n)}{n} \leq L_n^* < \frac{H(X_1, X_2, \ldots, X_n)}{n} + \frac{1}{n}.$$

Moreover, if $X_1, X_2, \ldots, X_n$ is a stationary stochastic process,

$$L_n^* \to H(\mathcal{X}),$$

where $H(\mathcal{X})$ is the entropy rate of the process.

Particularly, if the process is i.i.d., then

$$H \leq L_n^* < H + \frac{1}{n}.$$

4.1.2 A counting argument

Let us consider that there exists an algorithm, $\mathcal{C}$, that is able to compress all messages of n bits (or more). There are 2^n messages of n bits. By hypothesis, $\mathcal{C}$ is able to reduce the size of each of the 2^n messages to less than n bits.

How many codewords can be formed with less than n bits? There are 2^{n-1} with $n-1$ bits, 2^{n-2} with $n-2$ bits, $2^{n-(n-1)} = 1$ bit, $\ldots$, $2^{n-n} = 1$ with $n-n = 0$(!) bits.

Therefore, the total number of codewords with less than n bits is

$$\sum_{i=0}^{n-1} 2^i = \frac{1-2^n}{1-2} = 2^n - 1.$$

Hence, at least, two of the messages should share the same codeword, meaning that it is impossible to have a one-to-one codeword assignment. This implies that it is impossible to have a (reversible) compression algorithm that is able to compress all messages. For some of them, it will not compress or even it will expand its length.

4.1.3 Huffman codes

Huffman codes are variable-length codes developed by David A. Huffman (1925–1999) (Huffman, 1952). These codes are optimum, in the sense that they minimize the average length of the encoded messages, among all possible variable-length codes.

They are prefix-free codes, i.e., none of the codewords has a prefix made of a shorter codeword. Recall that, for example, if a given codeword is represented by the binary string "100", then the binary string "10001" cannot belong to that code. As shown before, this property ensures unique and instantaneous decoding.

The Huffman procedure for building the codes is based on two observations regarding optimum prefix codes:

- In an optimum code, symbols that occur more frequently should have shorter codewords than symbols that occur less frequently.
- In and optimum code, the two symbols that occur less frequently should have the same length.

The Huffman procedure contains the additional restriction that:

- The codewords of the two lowest probability symbols differ only in the last bit.

Therefore, for constructing the Huffman codes, the symbols (tree nodes) with the smallest probabilities are combined first. The new node gets the sum of the probabilities of the two combined nodes. This procedure is repeated until having a single node with a probability equal to one. Finally, the codewords are obtained by associating 0's and 1's to the branches of the tree.

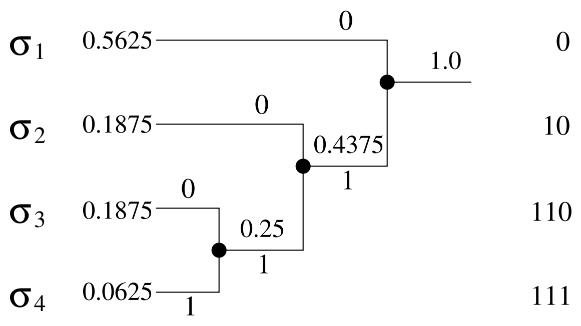


Figure 4.1: Example of tree associated with a 4-symbol Huffman code.

Example 4.1

In the example of Fig. 4.1, the ***mean number of bits/symbol*** *is*

$$1 \times 0.5625 + 2 \times 0.1875 + 3 \times 0.1875 + 3 \times 0.0625 \approx 1.69$$

The entropy of this information source is 1.62 *bits. Therefore, the* ***redundancy*** *of this code is* ≈ 0.07 *bits/symbol.*

The decoding process is straightforward. For example, suppose that the decoder received the binary string:

```
0100111101100
```

Then, the symbol sequence is

0	10	0	111	10	110	0
σ_1	σ_2	σ_1	σ_4	σ_2	σ_3	σ_1

When there are several nodes available for combining (i.e., with equal probabilities), it is possible to design different codes. In this case, which one shall we choose? For example, consider the following probability table, associated to a 5-symbol alphabet,

	σ_1	σ_2	σ_3	σ_4	σ_5
p_i	0.4	0.2	0.2	0.1	0.1

The first order entropy of this information source is 2.12 bits, and Fig. 4.2 shows two possible codes for this probability distribution. As can be verified, both codes have the same average number of bits/symbol:

$$2 \times 0.4 + 2 \times 0.2 + 2 \times 0.2 + 3 \times 0.1 + 3 \times 0.1 = 2.2$$

$$1 \times 0.4 + 3 \times 0.2 + 3 \times 0.2 + 3 \times 0.1 + 3 \times 0.1 = 2.2$$

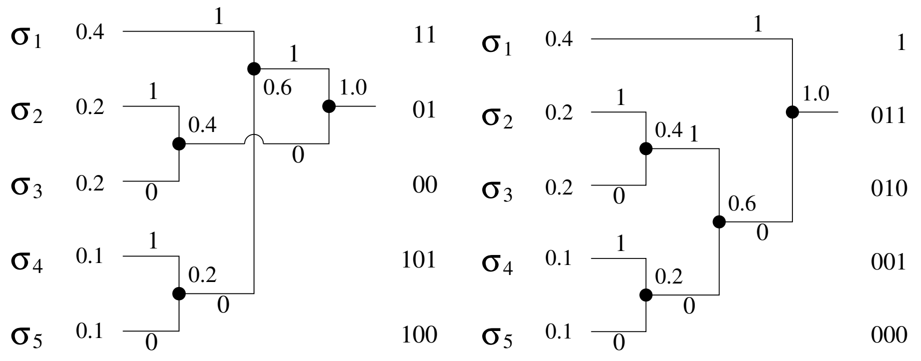


Figure 4.2: Two alternative Huffman codes for the same probability distribution.

After a sufficiently long transmission time, both codes are equivalent. However, those having higher variances might be less convenient, because they might require instantaneous higher bitrates and hence larger buffers. The first code has variance

$$0.4(2-2.2)^2 + 2 \times 0.2(2-2.2)^2 + 2 \times 0.1(3-2.2)^2 = 0.16,$$

whereas the second has variance 0.96.

Programming 4.1
Write a program that simulates the encoding process for a given probability distribution and corresponding Huffman code (for example, those shown in Fig. 4.2). Using that program, analyze the statistics of bitstream produced by the encoder with the aim of finding out if it is still compressible.

Programming 4.2
Using simulation, estimate the average time to overflow/underflow of the buffer, as a function of buffer size. As a possible example, verify that the leftmost code of Fig. 4.2 is less demanding than the code shown on the right, regarding buffer size.

4.1.4 Shannon-Fano codes

Shannon-Fano coding was proposed before Huffman coding and, although not optimum as Huffman coding, it nevertheless provides good variable-size codes. Shannon-Fano codes are like Huffman codes, in the sense that they also are variable-length and immediately decodable codes.

For constructing these codes, the symbols are first ordered by increasing or decreasing probability. Then, the symbols are split into two sets having as much as possible similar total probabilities. One of the sets gets a "0" label, whereas the other gets a "1" label. This is repeated until having sets with only one or two symbols.

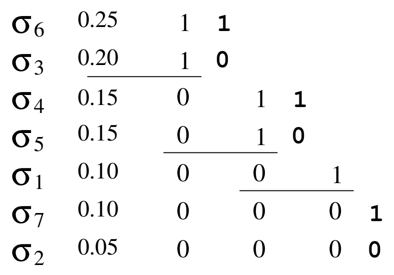


Figure 4.3: Example of construction of a Shannon-Fano code.

Example 4.2
Consider the following probability table:

σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7
0.10	0.05	0.20	0.15	0.15	0.25	0.10

The Shannon-Fano code for this probability distribution is shown in Fig. 4.3.

Note that whereas the Shannon-Fano tree is created from the root to the leaves, the Huffman algorithm works from leaves to the root.

4.1.5 Alphabet extension

When

$$p_i \to 2^{-k}, \quad k \in \mathbb{N}, \quad \forall i,$$

the efficiency of the variable-length codes gets closer to the entropy of the information source. Particularly, when the probabilities are powers of $1/2$, the average length of the codes equals the entropy of the source. In this case,

$$-\log p_i \in \mathbb{N}, \quad \forall i,$$

i.e., the optimum lengths of the codewords are integers. However, for small alphabets and for highly skewed probability distributions, the corresponding variable-length codes can be far from optimal!

Example 4.3
Let us consider two symbols, σ_1 and σ_2. Varying the values of p_1 and p_2, we get the following entropy values:

p_1	0.9	0.8	0.7	0.6	0.5
p_2	0.1	0.2	0.3	0.4	0.5
H	0.47	0.72	0.88	0.97	1.00

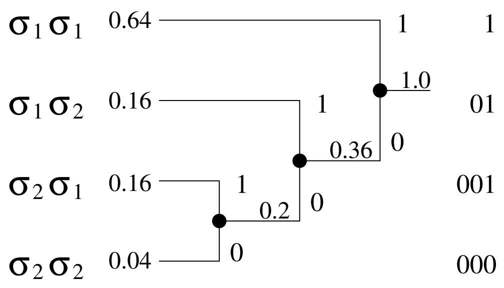


Figure 4.4: Huffman code resulting from the aggregation of pairs of symbols.

As expected, the larger the difference between these two probabilities, less bits, on average, are in theory needed for representing a message.

Consider, for example, the case where $p_1 = 0.8$ *and* $p_2 = 0.2$*:*

- *Using Huffman coding or Shannon-Fano coding, we get an average code length of one bit per symbol.*
- *This is 28% longer than what is theoretically necessary.*

Let us now consider groupings of pairs of symbols and assume ***independence****. The probabilities of the four combinations are*

Pair	Probability
$\sigma_1\sigma_1$	$0.8 \times 0.8 = 0.64$
$\sigma_1\sigma_2$	$0.8 \times 0.2 = 0.16$
$\sigma_2\sigma_1$	$0.2 \times 0.8 = 0.16$
$\sigma_2\sigma_2$	$0.2 \times 0.2 = 0.04$

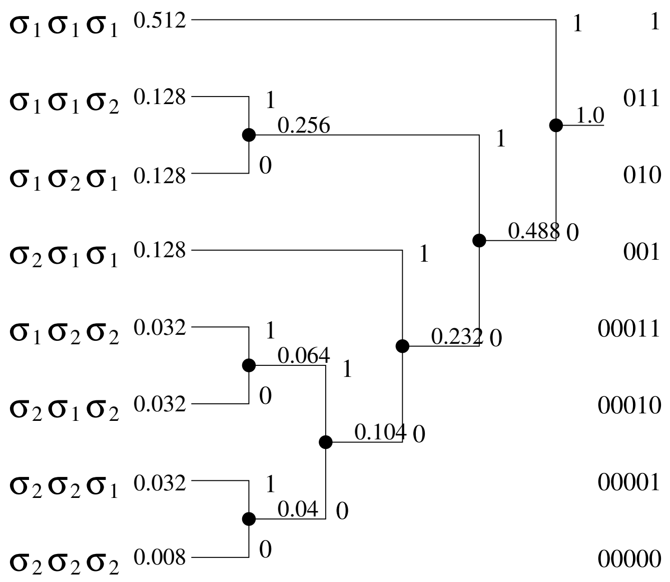
Constructing a Huffman code for these new symbols (see Fig. 4.4), we get an average length of

$$1 \times 0.64 + 2 \times 0.16 + 3 \times 0.16 + 3 \times 0.04 = 1.56 \textit{ bits},$$

i.e., an average length of 0.78 *bits/original symbol.*

Consider now groupings of three symbols, with probabilities

Group	Probability
$\sigma_1\sigma_1\sigma_1$	$0.8 \times 0.8 \times 0.8 = 0.512$
$\sigma_1\sigma_1\sigma_2$	$0.8 \times 0.8 \times 0.2 = 0.128$
$\sigma_1\sigma_2\sigma_1$	$0.8 \times 0.2 \times 0.8 = 0.128$
$\sigma_1\sigma_2\sigma_2$	$0.8 \times 0.2 \times 0.2 = 0.032$
$\sigma_2\sigma_1\sigma_1$	$0.2 \times 0.8 \times 0.8 = 0.128$
$\sigma_2\sigma_1\sigma_2$	$0.2 \times 0.8 \times 0.2 = 0.032$
$\sigma_2\sigma_2\sigma_1$	$0.2 \times 0.2 \times 0.8 = 0.032$
$\sigma_2\sigma_2\sigma_2$	$0.2 \times 0.2 \times 0.2 = 0.008$



$\sigma_1\sigma_1\sigma_1$ 0.512
$\sigma_1\sigma_1\sigma_2$ 0.128
$\sigma_1\sigma_2\sigma_1$ 0.128
$\sigma_2\sigma_1\sigma_1$ 0.128
$\sigma_1\sigma_2\sigma_2$ 0.032
$\sigma_2\sigma_1\sigma_2$ 0.032
$\sigma_2\sigma_2\sigma_1$ 0.032
$\sigma_2\sigma_2\sigma_2$ 0.008
0.256
0.064
0.04
0.104
0.232
0.488
1.0
1
011
010
001
00011
00010
00001
00000

Figure 4.5: Huffman code for groups of three symbols.

The average length of the corresponding Huffman code is 2.184 *bits, i.e.,* 0.728 *bits/original symbol (see Fig. 4.5).*

4.1.6 Some other variable-length codes

Sometimes, it is necessary to design codes for representing, efficiently, integer numbers whose probabilities decrease with their values. In this case, it is possible to construct codes that are simpler than the Huffman codes, although sometimes with some loss of efficiency. On the other hand, these are "open" codes, i.e., they are able to accommodate rare, but possible, (large) numbers.

The **comma code** (also known as the **unary code**) is the simplest variable-length code. Each codeword is built using a string of 0's (1's), terminated by one 1 (0):

$$\begin{array}{lcl} 0 & - & 0 \\ 1 & - & 10 \\ 2 & - & 110 \\ 3 & - & 1110 \\ 4 & - & 11110 \\ \dots & & \end{array}$$

Note that the decoder is just a counter.

Problem 4.1
Show that the unary code is optimum if the integers, i, occur with probabilities

$$p_i = \frac{1}{2^{(i+1)}}, \quad i = 0, 1, 2, \ldots$$

Shift-codes are generalizations of the comma code.

$B = 1$	$B = 2$	$B = 3$
0	00	000
10	01	001
110	10	010
1110	1100	011
11110	1101	100
111110	1110	101
1111110	111100	110
11111110	111101	111000
...	...	...

- These codes are organized by levels, based on a set of 2^B codewords of length B.
- One of those codewords is used to indicate the shift to the next level.
- For $B = 1$ we have the comma code.

Another type of shift code uses, besides the base, B, an increment, I. The first level is formed by all codewords of length B, except for the shift codeword. The next level (level 2) is formed by all codewords of $B + I$ bits, except for the shift codeword, concatenated with the (prefix) shift codeword of the previous level.

Generalizing, level L consists of all $B + I(L - 1)$ bit codewords, except for the shift sequence, concatenated with the prefix (shift codeword) of level $L - 1$.

Example 4.4

$B = 1, I = 1$		$B = 2, I = 1$		$B = 1, I = 2$	
L	*Codeword*	L	*Codeword*	L	*Codeword*
1	0	1	00	1	0
2	100	1	01	2	1000
2	101	1	10	2	1001
2	110	2	11000	2	1010
3	111000	2	11001	2	1011
3	111001	2	11010	2	1100
3	111010	2	11011	2	1101
3	111011	2	11100	2	1110
3	111100	2	11101	3	111100000
3	111101	2	11110	3	111100001
3	111110	3	111110000	3	111100010
	...		...		...

4.1.7 Golomb code

The **Golomb code** was proposed by Solomon W. Golomb (1932–2016) (Golomb, 1966). It relies on separating an integer into two parts:

- One of those parts is represented by a **unary code**.
- The other part is represented using a **binary code**.

In fact, the Golomb code is a family of codes that depend on an integer parameter, $m > 0$. The Golomb code is optimum for an information source following a distribution

$$p_i = \alpha^i(1-\alpha), \quad i = 0, 1, 2, \ldots$$

where

$$m = \left\lceil -\frac{1}{\log \alpha} \right\rceil .$$

An integer $i \geq 0$ is represented by two numbers, q and r, where

$$q = \left\lfloor \frac{i}{m} \right\rfloor \qquad \text{and} \qquad r = i - qm.$$

The quotient, q, can have the values $0, 1, 2, \ldots$, and is represented by the corresponding unary code. The remainder of the division, r, can have the values $0, 1, 2, \ldots, m-1$, and is represented by the corresponding binary code.

Example 4.5
Consider $i = 11$ and $m = 4$. Then,

$$q = \left\lfloor \frac{11}{4} \right\rfloor = 2, \quad r = 11 - 2 \times 4 = 3 \quad \rightarrow \quad 001\ 11$$

If m is not a power of 2, then the binary code (that represents the remainder of the division), is not efficient. In that case, the number of bits used can be reduced using a **truncated binary code**:

- First, define $b = \lceil \log m \rceil$.
- Encode the first $2^b - m$ values of r using the first $2^b - m$ binary codewords of $b - 1$ bits.
- Encode the remainder values of r by coding the number $r + 2^b - m$ in binary codewords of b bits.

Example 4.6
Consider $m = 5$. In this case, we have $b = \lceil \log 5 \rceil = 3$. Therefore, values $r = 0, 1, 2$ are represented using 2 bits (respectively "00", "01" and "10"), whereas the values $r = 3, 4$ will be encoded using 3 bits (respectively "110" and "111").

Example 4.7
Golomb code for $m = 5$ and $i = 0, 1, \ldots, 15$:

i	q	r	*Codeword*	i	q	r	*Codeword*
0	0	0	000	8	1	3	10110
1	0	1	001	9	1	4	10111
2	0	2	010	10	2	0	11000
3	0	3	0110	11	2	1	11001
4	0	4	0111	12	2	2	11010
5	1	0	1000	13	2	3	110110
6	1	1	1001	14	2	4	110111
7	1	2	1010	15	3	0	111000

4.2 Dictionary based compression

In many cases, the information source produces recurring patterns. A possible approach to take advantage of this behaviour is to keep a **dictionary** with the (most frequent) patterns. When one of those patterns occurs, it is encoded using a reference to the dictionary. If it does not appear in the dictionary, it can be encoded using some other (usually less efficient) method. Therefore, the idea is to split the patterns into two classes: frequent and infrequent.

Example 4.8
Suppose we have a source alphabet with 32 symbols (for example, 26 letters plus some punctuation marks). For an i.i.d. source, we would need 5 bits/symbol. Treating all 32^4 *(*$1\,048\,576$*) 4-symbol patterns as equally likely, it would imply 20 bits for each 4-symbol pattern.*

Now, suppose we put the 256 most common patterns in a dictionary. If the pattern is in the dictionary, send '0' followed by the (8-bit) index of the pattern in the dictionary. Otherwise, send '1' followed by the 20 bits encoding the pattern.

Notice that, if p *is the probability of finding the pattern in the dictionary, then the average code-length is*

$$L = 9p + 21(1-p) = 21 - 12p.$$

To be useful, this scheme should attain $L < 20$*. This happens for* $p \geq 0.084$*. However, for an i.i.d. source, the probability of a 4-symbol pattern over a 32-symbol alphabet is only* $0.00000095\ldots$

4.2.1 Tunstall codes

As we have seen, variable-length codes assign variable-length bit strings to the symbols of the alphabet. One of the problems with this approach is that errors in the encoded data propagate easily.

In the **Tunstall code**, all codewords are of equal length. Instead, each codeword represents variable-length groups of alphabet symbols.

Example 4.9
Consider the following 2-bit Tunstall code for the alphabet $\Sigma = \{A, B\}$*:*

Sequence	*Codeword*
AAA	*00*
AAB	*01*
AB	*10*
B	*11*

Then, the sequence "AAABAABAABAABAAA" is encoded as "001101010100".

The design of a code that has a fixed codeword length, but a variable number of symbols per codeword, needs to meet the following two conditions:

- We should be able to parse all source strings into substrings of symbols that appear in the codebook (dictionary). This property is usually known as "**generality**".
- We should maximize the average number of source symbols represented by each codeword.

Example 4.10
To understand the meaning of the first condition, consider now the following code:

Sequence	*Codeword*
AAA	*00*
ABA	*01*
AB	*10*
B	*11*

Let us try to encode the same sequence, "AAABAABAABAABAAA", but now using this new code:

- *First, we encode "AAA" with code "00".*
- *Then, we encode "B" with code "11".*
- *Next, we run into trouble, because there is no way to proceed...*

Lemma 4.1 (Dictionary generality)
A dictionary consisting of strings from an alphabet Σ satisfies the generality condition iff for every possible string of symbols x from Σ there exists at least one prefix of x in the dictionary.

Discussion topic 4.1
Discuss conditions that a dictionary must meet in order to obey the generality restriction.

A n-bit Tunstall code for an i.i.d. source over a size-m alphabet can be built using the following algorithm:

Algorithm 4.1 (Tunstall code)

- *Start with the m symbols of the alphabet in the codebook.*
- *Then, remove the entry with highest probability and add the m strings obtained by concatenating this symbol with every symbol of the alphabet. The new probabilities are the product (because independence is assumed) of the probabilities of the concatenated symbols—notice that this operation increases the size of the codebook from m to $m + (m - 1)$.*
- *Repeat this procedure k times, subject to the constraint $m + k(m - 1) \leq 2^n$, where k is the largest possible integer that verifies the condition.*

Example 4.11
Consider the design of a 3-bit Tunstall code for the alphabet

Symbol	*Probability*
A	*0.60*
B	*0.30*
C	*0.10*

Because "A" is the most probable symbol, we first obtain

Sequence	*Probability*
B	*0.30*
C	*0.10*
AA	*0.36*
AB	*0.18*
AC	*0.06*

Finally, we have

Sequence	*Probability*	*Code*
B	*0.300*	*000*
C	*0.100*	*001*
AB	*0.180*	*010*
AC	*0.060*	*011*
AAA	*0.216*	*100*
AAB	*0.108*	*101*
AAC	*0.036*	*110*

Discussion topic 4.2
Are the Tunstall codes uniquely decodable? Why?

4.2.2 String parsing

The ultimate objective of dictionary-based data compression is to parse the source string into substrings (also known as phrases or factors) whose corresponding codewords have the smallest possible total length. Parsing, i.e., creating a partition of the string, plays an important role in this process.

For an arbitrary dictionary, complying with the generality constraint, there are optimal (in the sense of producing the least number of phrases) ways of parsing the string to be compressed. Typically, they involve more than one pass over the string of data, using dynamic programming. More direct approaches, although possibly sub-optimal, rely on greedy strategies. In this case, assuming that processing is done left-to-right, the greedy algorithm chooses the longest possible prefix of the yet uncompressed string that exists in the dictionary.

For some special cases, the greedy strategy can be, in fact, optimal. This happens when the dictionaries are prefix-complete or suffix-complete. A dictionary is **prefix-complete** if for every string in the dictionary, all of its prefixes are also in the dictionary. A dictionary is **suffix-complete** if for every string in the dictionary, all of its suffixes are also in the dictionary.

Theorem 4.4 (Cohn and Khazan 1996)
With respect to a suffix-complete dictionary, greedy parsing left-to-right is optimal; dually, with respect to a prefix-complete dictionary, greedy parsing right-to-left is optimal.

4.2.3 LZ77 compression

In the 1970's, Jacob Ziv (1931–) and Abraham Lempel (1936–) proposed two different approaches for data compression, using dictionary-based principles. Contrarily to the Tunstall code that we have just seen, where the dictionary is built beforehand (i.e., it is **static**), these methods are **adaptive**—the dictionary is "learned" at the same time compression is performed.

The first algorithm, usually known as LZ77 (or LZ1), was presented in

- J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory (Ziv and Lempel, 1977).

Essentially, it relies on a separation of the data in two parts:

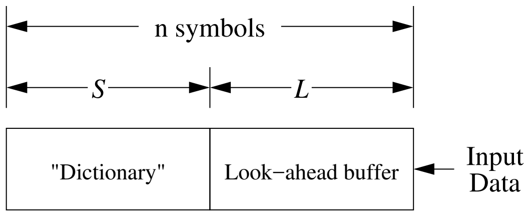


Figure 4.6: Separation between the “dictionary” and data to be encoded in LZ77.

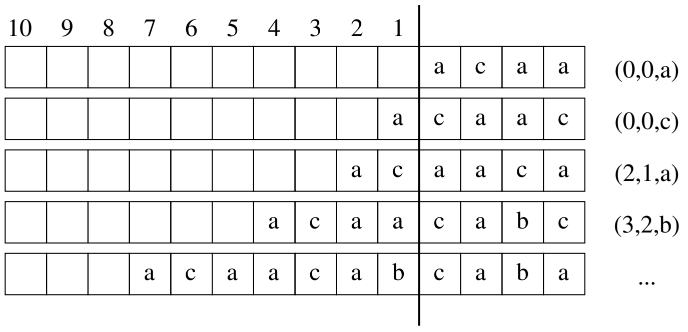


Figure 4.7: Simple example of LZ77 encoding.

- Data already encoded;
- Data that are to be encoded,

as shown in Fig. 4.6.

During operation, the data go first through a **look-ahead buffer** and then through a **search buffer** (the “dictionary”). The algorithm searches, in the “dictionary”, for the largest possible prefix of the string in the look-ahead buffer. Obviously, the larger the prefix, the higher will be the coding efficiency.

The codeword that is generated in each coding step is composed of **three components**:

1. A pointer, that indicates the position of the matching string in the “dictionary”.
2. The length of the string matched.
3. The first symbol that in the look-ahead buffer follows the matching string.

Figure 4.7 shows a simple example of encoding. The look-ahead buffer can also be used as a “dictionary” extension, as exemplified in Fig. 4.8.

In LZ77 encoding, a codeword requires

$$\lceil \log_2 S \rceil + \lceil \log_2 (L-1) \rceil + \lceil \log_2 m \rceil$$

bits, where S is the size of the “dictionary”, L is the size of the look-ahead buffer and m is the size of the alphabet. Taking as example $S = 2048$, $L = 33$ and $m = 256$, a codeword would

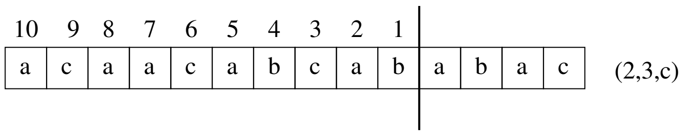


Figure 4.8: Example of the use of the look-ahead buffer for "dictionary" extension in LZ77 encoding.

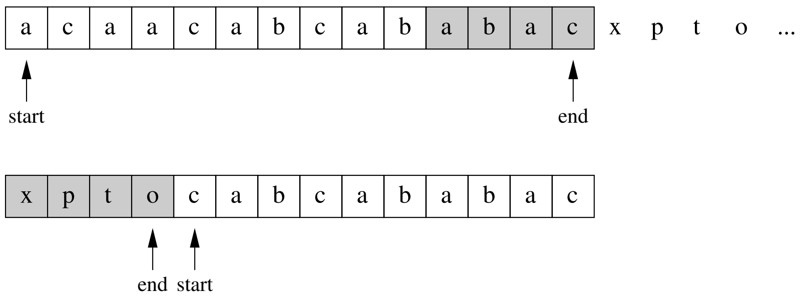

a c a a c a b c a b a b a c x p t o ...
start
end
x p t o c a b c a b a b a c
end start

Figure 4.9: Look-ahead buffer implemented using a circular queue.

require $11 + 5 + 8 = 24$ bits, which is three times the number of bits required to represent a single symbol (8 bits).

Discussion topic 4.3
There is a number of practical questions that need to be addressed when implementing this compression algorithm, such as

- *The search time required by each coding step as a function of the dictionary size.*
- *The probability of finding matching sequences as a function of the dictionary size.*
- *The processing time required by each coding step as a function of the look-ahead buffer size.*
- *The probability of finding matching sequences as a function of the look-ahead size.*
- *The impact in codeword length when the sizes of the dictionary and/or look-ahead buffer are changed.*
- *The potential inefficiency of using three component codewords when having to encode isolated symbols.*

Discuss them.

The implementation of LZ77-type algorithms needs also to take into consideration the efficient use of data structures. For example, the use of circular buffers (see Fig. 4.9) and tree structures (see Fig. 4.10) may dramatically improve processing speed.

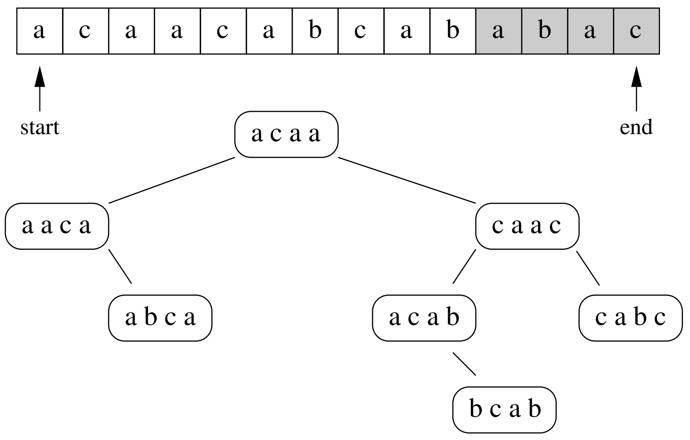


Figure 4.10: Dictionary implemented using a binary tree.

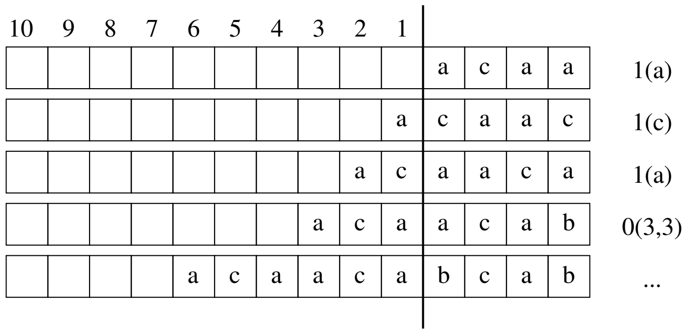


Figure 4.11: Example of LZSS encoding.

4.2.4 LZSS compression

LZSS is a variant of LZ77 proposed by James Storer (1953–) and Thomas Szymanski (Storer and Szymanski, 1982). As shown in the previous section, with LZ77 sometimes it is possible to produce codewords that use more bits than the original bits of the string it represents. With LZSS, this is avoided by enforcing codewords with only **two components**—one bit is used to indicate if the data are encoded or sent as uncoded, literal form. Therefore, when a sequence is not found in the dictionary, the symbol code is sent, instead of $(0, 0, \dots)$ as in LZ77. To distinguishing both cases, one bit is used. Figure 4.11 gives an example of an encoding using LZSS.

4.2.5 LZ78 compression

The second algorithm proposed by Jacob Ziv and Abraham Lempel, known as LZ78 (or LZ2), was published in

- J. Ziv and A. Lempel, *Compression of individual sequences via variable-rate coding*,

Dictionary:	a	aa	b	aaa	d	aab	aad	o
Index:	1	2	3	4	5	6	7	8
Codeword:	0,a	1,a	0,b	2,a	0,d	2,b	2,d	0,o

Figure 4.12: LZ78 encoding of string “aaabaaadaabaado”.

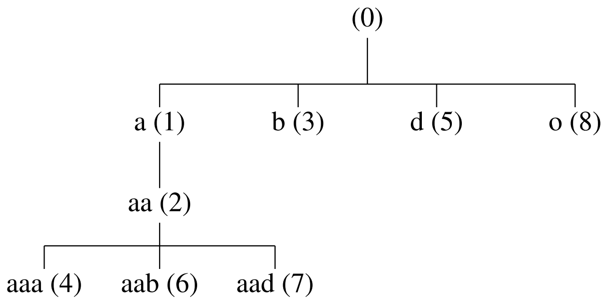


Figure 4.13: Tree data structure for fast access to the LZ78 dictionary.

IEEE Trans. on Information Theory (Ziv and Lempel, 1978).

LZ78 relies on a distinct approach from that of LZ77, where the use of an explicit dictionary is probability the most obvious. In LZ78 coding:

- The data are parsed into strings.
- Each string is built from the largest matching string found so far, plus the first non-matching symbol.
- The new string is represented using the index of the matching string (which is in the dictionary), followed by the code of the unmatched symbol.
- Finally, the new string is added to the dictionary.

Figure 4.12 shows how the string “aaabaaadaabaado” is represented by the LZ78 encoding algorithm. It is worth noting that, apparently, this new strategy seems not to put restrictions to the distance at which repeating patterns occur, whereas with LZ77, if this distance exceeds the size of the searching area (the “dictionary”), they are not taken into account. Nevertheless, it was shown that both techniques are asymptotically optimal for ergodic sources, i.e., asymptotically, its encoding efficiency approaches the entropy of the source.

As with LZ77-type algorithms, it is fundamental to use appropriate data structures to speed up processing. In the case of LZ78, a possibility is to represent the dictionary using tree structures, such as in the example of Fig. 4.13.

Discussion topic 4.4

In practice, the dictionary cannot grow without bound, because, for example,

- *The size of the codewords is related to the size of the dictionary.*
- *The amount of memory for storing the dictionary (both by the encoder and by the decoder) might be too large.*

Therefore, in practice, it is necessary to implement mechanisms for limiting the growth of the dictionary. Discuss advantages and disadvantages of possible mechanisms for doing so, such as,

- *Dictionary "freezing", when it reaches some predefined size.*
- *When the dictionary reaches some predefined size, it is deleted and started again (note that this is identical to block coding).*
- *When the dictionary is full, some entries are deleted, for example, those not used for a longer time or less used.*

Discussion topic 4.5
LZ77 relies on a suffix-complete dictionary, whereas LZ78 relies on a prefix-complete dictionary (explain why). Based on Theorem 4.4, discuss possible implications of this fact.

4.2.6 LZW compression

In 1984, Terry Welch (1939–1988) published a paper where solutions for some of the problems associated to LZ78 are proposed, namely (Welch, 1984):

- Symbols seen for the first time are encoded more efficiently (in LZ78, two-component codewords are generated in this case).
- The number of components of the codewords is reduced to just **one**.

Initially, all symbols of the alphabet are inserted in the dictionary. Then, the algorithm operates as follows:

- Symbols are read, one by one, from the string to encode.
- These symbols (σ) are concatenated to a string, x, while $x\sigma$ can be found in the dictionary.
- When it is not possible to add one more symbol, the index of x in the dictionary is sent, $x\sigma$ is inserted in the dictionary, and x is initialized with symbol σ.

Example 4.12

Consider the encoding of string "aaabaaadaabaado". Initially, the dictionary contains all symbols of the alphabet:

0	…	*97*	*98*	*99*	*100*	…	*255*
nul	…	*a*	*b*	*c*	*d*	…	

String: $\boxed{aa}$*abaaadaabaado; Codeword sent: 97*

…	*97*	*98*	*99*	*100*	…	*256*
…	*a*	*b*	*c*	*d*	…	*aa*

String: a$\boxed{aab}$*aaadaabaado; Codeword sent: 256*

…	*97*	*98*	*99*	*100*	…	*256*	*257*
…	*a*	*b*	*c*	*d*	…	*aa*	*aab*

String: aaa$\boxed{ba}$*aadaabaado; Codeword sent: 98*

…	*97*	*98*	*99*	*100*	…	*256*	*257*	*258*
…	*a*	*b*	*c*	*d*	…	*aa*	*aab*	*ba*

String: aaab$\boxed{aaa}$*daabaado; Codeword sent: 256*

…	*97*	*98*	*99*	*100*	…	*256*	*257*	*258*	*259*
…	*a*	*b*	*c*	*d*	…	*aa*	*aab*	*ba*	*aaa*

String: aaabaa$\boxed{ad}$*aabaado; Codeword sent: 97*

…	*97*	*98*	*99*	*100*	…	*256*	*257*	*258*	*259*	*260*
…	*a*	*b*	*c*	*d*	…	*aa*	*aab*	*ba*	*aaa*	*ad*

String: aaabaaa$\boxed{da}$*abaado; Codeword sent: 100*

…	*97*	*98*	*99*	*100*	…	*256*	*257*	*258*	*259*	*260*	*261*
…	*a*	*b*	*c*	*d*	…	*aa*	*aab*	*ba*	*aaa*	*ad*	*da*

String: aaabaaad$\boxed{aaba}$*ado; Codeword sent: 257*

…	*97*	*98*	*99*	*100*	…	*256*	*257*	*258*	*259*	*260*	*261*	*262*
…	*a*	*b*	*c*	*d*	…	*aa*	*aab*	*ba*	*aaa*	*ad*	*da*	*aaba*

String: aaabaaadaab$\boxed{aad}$*o; Codeword sent: 256*

…	*97*	*98*	*99*	*100*	…	*256*	*257*	*258*	*259*	*260*	*261*	*262*	*263*
…	*a*	*b*	*c*	*d*	…	*aa*	*aab*	*ba*	*aaa*	*ad*	*da*	*aaba*	*aad*

...

Example 4.13

Decoding is performed as follows. Initially, the dictionary contains all symbols of the alphabet.

0	...	*97*	*98*	*99*	*100*	...	*255*
nul	...	*a*	*b*	*c*	*d*	...	

Codeword received: 97 ; String: $\boxed{a?}$

...	*97*	*98*	*99*	*100*	...	*256*
...	*a*	*b*	*c*	*d*	...	*a?*

Codeword received: 256 ; String: a$\boxed{aa?}$

...	*97*	*98*	*99*	*100*	...	*256*	*257*
...	*a*	*b*	*c*	*d*	...	*aa*	*aa?*

Codeword received: 98 ; String: aaa$\boxed{b?}$

...	*97*	*98*	*99*	*100*	...	*256*	*257*	*258*
...	*a*	*b*	*c*	*d*	...	*aa*	*aab*	*b?*

Codeword received: 256 ; String: aaab$\boxed{aa?}$

...	*97*	*98*	*99*	*100*	...	*256*	*257*	*258*	*259*
...	*a*	*b*	*c*	*d*	...	*aa*	*aab*	*ba*	*aa?*

Codeword received: 97 ; String: aaabaa$\boxed{a?}$

...	*97*	*98*	*99*	*100*	...	*256*	*257*	*258*	*259*	*260*
...	*a*	*b*	*c*	*d*	...	*aa*	*aab*	*ba*	*aaa*	*a?*

Codeword received: 100 ; String: aaabaaa$\boxed{d?}$

...	*97*	*98*	*99*	*100*	...	*256*	*257*	*258*	*259*	*260*	*261*
...	*a*	*b*	*c*	*d*	...	*aa*	*aab*	*ba*	*aaa*	*ad*	*d?*

Codeword received: 257 ; String: aaabaaad$\boxed{aab?}$

...	*97*	*98*	*99*	*100*	...	*256*	*257*	*258*	*259*	*260*	*261*	*262*
...	*a*	*b*	*c*	*d*	...	*aa*	*aab*	*ba*	*aaa*	*ad*	*da*	*aab?*

Codeword received: 256 ; String: aaabaaadaab$\boxed{aa?}$

...	*97*	*98*	*99*	*100*	...	*256*	*257*	*258*	*259*	*260*	*261*	*262*	*263*
...	*a*	*b*	*c*	*d*	...	*aa*	*aab*	*ba*	*aaa*	*ad*	*da*	*aaba*	*aa?*

...

4.3 Arithmetic coding

4.3.1 Motivation

We have seen that one of the main limitations of variable length codes is that they are optimum only if the probabilities of the symbols are powers of $1/2$.

Example 4.14
Suppose we have

$$p_1 = \frac{1}{2}, \quad p_2 = \frac{1}{4}, \quad p_3 = \frac{1}{4}$$

The first-order entropy of this information source is 1.5 *bits/symbol. Using a Huffman code, we have, for example,*

$$\sigma_1 \to 0 \quad \sigma_2 \to 10 \quad \sigma_3 \to 11$$

The average length of this code is 1.5 *bits/symbol and, therefore, its redundancy is zero (i.e., this code is optimal for this source).*

Example 4.15
Consider now that the probability distribution of the source symbols is

$$p_1 = \frac{2}{3}, \quad p_2 = \frac{1}{6}, \quad p_3 = \frac{1}{6}.$$

In this case, the entropy of the source is ≈ 1.26*. However, as in the previous case, the Huffman encoding algorithm assigns 1 bit to* σ_1 *and 2 bits to the other two symbols. The average length of this code is* ≈ 1.33*, i.e.,* 0.07 *bits per symbol more than the source entropy. The difference between the average code length and the entropy increases when* $\max(p_i) > 0.5$ *and* $\max(p_i) \to 1$.

As with Huffman coding, arithmetic coding needs an estimate of the probabilities of the symbols of the alphabet. Arithmetic coding represents the entire message by a single codeword: a number in the $[0, 1)$ interval. This is why arithmetic coding is not affected by the skewness of the probability distribution, as variable length codes might be. Besides, arithmetic coding has the advantage of allowing a clear separation between the **coding** process and source **modeling**.

4.3.2 Encoding

For encoding, the $[0, 1)$ interval is partitioned according to the probability distribution of the symbols. The only restriction is that the decoder has to use the same partitioning.

Example 4.16
Consider the following alphabet and corresponding probability distribution:

Symbol	*Prob.*	*Interval*
d	0.1	$[0, 0.1)$
e	0.3	$[0.1, 0.4)$
f	0.1	$[0.4, 0.5)$
g	0.05	$[0.5, 0.55)$
n	0.1	$[0.55, 0.65)$
r	0.2	$[0.65, 0.85)$
u	0.05	$[0.85, 0.9)$
space	0.1	$[0.9, 1)$

Let us consider the encoding of the message "fred".

*Initially, the **coding interval** is* $[0, 1)$. *The first symbol to encode, "f", has the* $[0.4, 0.5)$ *interval assigned to it, according to the probability table. After encoding this symbol, the coding interval will be* $[0.4, 0.5)$. *The next symbol, "r", is associated with the* $[0.65, 0.85)$ *interval. Then, the **lower limit** of the new coding interval will be*

$$0.4 + 0.65(0.5 - 0.4) = 0.465$$

*and the **upper limit***

$$0.4 + 0.85(0.5 - 0.4) = 0.485.$$

The next symbol, "e", transforms the current coding interval, $[0.465, 0.485)$, *into* $[0.467, 0.473)$, *according to*

$$0.467 = 0.465 + 0.1(0.485 - 0.465)$$

and

$$0.473 = 0.465 + 0.4(0.485 - 0.465).$$

After encoding symbol "d", we get the interval $[0.467, 0.4676)$.

Therefore, if the current coding interval is $[L^n, H^n)$, and if the next symbol to encode is represented by the interval $[l, h)$, then

$$L^{n+1} = L^n + l(H^n - L^n)$$

and

$$H^{n+1} = L^n + h(H^n - L^n).$$

Example 4.17
Consider the encoding of the binary string `0110`, *using* $P(0) = 0.6$ *and* $P(1) = 0.4$. *The encoded message is a number in the interval* $[0.504, 0.5616)$ *(see Fig. 4.14). Another possible view of the encoding process can be seen in Fig. 4.15, in this case with renormalized intervals.*

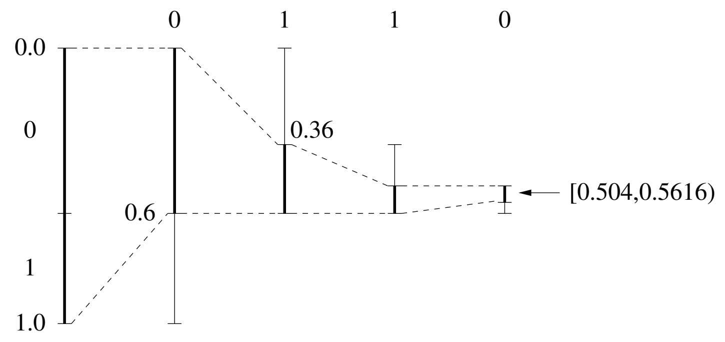


Figure 4.14: Example of the arithmetic encoding of a simple string.

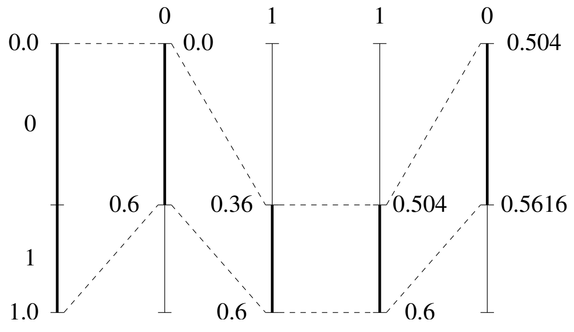


Figure 4.15: The same example as of Fig. 4.14, but with a different visualization of the intervals.

4.3.3 Decoding

For decoding the "fred" message, we proceed as follows. Let x be the value received by the decoder (i.e., the encoded message), such that $x \in [0.467, 0.4676)$. For example, consider $x = 0.467$. Since $x \in [0.4, 0.5)$, then the first symbol of the message is "f". The "effect" of this symbol is then eliminated from the encoded message, using the following procedure:

1. Subtract the lower limit of the "f" interval from x, i.e., $x - 0.4 = 0.467 - 0.4 = 0.067$.
2. Divide this new value by the range of the "f" interval (0.1, in this case): $0.067/0.1 = 0.67$.

The new number, $x = 0.67$, is in the $[0.65, 0.85)$ interval, meaning that "r" is the next symbol. Re-normalizing, we obtain $x = (0.67 - 0.65)/(0.85 - 0.65) = 0.1$. Because $x \in [0.1, 0.4)$, then the next symbol is "e". Re-normalizing again, $x = (0.1 - 0.1)/(0.4 - 0.1) = 0$. Since $x \in [0, 0.1)$, then the symbol is "d".

Notice that, there is no way to detect, automatically, the end of a message encoded using arithmetic coding. Therefore, it is necessary to indicate the length of the message (i.e., the number of symbols of the message) or to use a special "end-of-message" symbol.

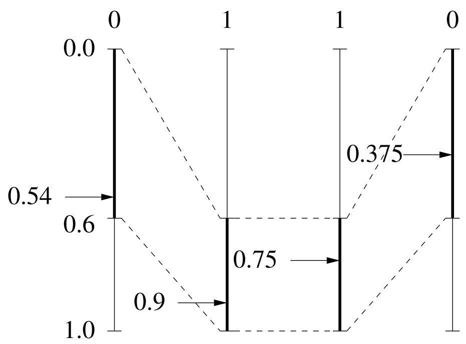


Figure 4.16: Decoding of the example depicted in Figs. 4.14 and 4.15.

Example 4.18
The decoding process of the previous example can be seen in Fig. 4.16, where $x = 0.54$ *was considered.*

4.3.4 Implementation issues

Obviously, it is not possible to obtain an useful encoder by direct implementation of the described process. In fact, the precision of the floating point representation of the computer would rapidly be exhausted. Nevertheless, even if we had access to a computer with infinite precision, it certainly would not be efficient to perform arithmetic operations with several thousands or millions of decimal places. Moreover, it is generally not acceptable to have to wait for the end of the message in order to start receiving the first bits of the encoded message.

As we saw earlier, as new symbols arrive at the encoder, the limits of the coding interval get closer to each other, implying the need of an increasing precision in order to distinguish them apart. However, it is reasonable to admit that, at a certain point, both limits will lay above 0.5 or below 0.5. When this happens, the most significant bit of L and H will stay permanently equal to 0, if $L < 0.5$ and $H < 0.5$, or equal to 1, if $L \geq 0.5$ and $H \geq 0.5$. Notice that

$$\begin{aligned} x \geq 0.5_{(10)} &\longrightarrow x = 0.1\ldots_{(2)} \\ x < 0.5_{(10)} &\longrightarrow x = 0.0\ldots_{(2)} \end{aligned}$$

Therefore, this bit can be sent out by the encoder.

On the other hand, we can expand the interval in the following way,

$$\begin{array}{lll} \text{Sent bit}: \ 0 & [0, 0.5) \longrightarrow [0, 1) & x \to 2x \\ \text{Sent bit}: \ 1 & [0.5, 1) \longrightarrow [0, 1) & x \to 2(x - 0.5), \end{array}$$

which, in principle, allow us to keep the precision under control.

But if $L \to 0.0111\ldots_{(2)}$ and $H \to 0.1000\ldots_{(2)}$?

To take into account this situation, whenever the coding interval belong to $[0.25, 0.75)$, we perform the expansion

$$[0.25, 0.75) \longrightarrow [0, 1) \qquad x \rightarrow 2(x - 0.25),$$

and a counter takes note of how many times these re-scalings occurred since the last bit has been sent by the encoder.

Notice that this operation is performed when $0.25 \le L < 0.5$ ($L = 0.01\ldots_{(2)}$) and $0.5 \le H < 0.75$ ($H = 0.10\ldots_{(2)}$). After this change of scale, we get $L = 0.0\ldots_{(2)}$ and $H = 0.1\ldots_{(2)}$. The result of this operation is that the second most significant bit is "deleted".

If, after incorporating the next symbol, $L \ge 0.5$ and $H \ge 0.5$, then the corresponding "1" is sent, followed by as many "0" bits as the value of the counter. On the other hand, if $L < 0.5$ and $H < 0.5$, it is sent the corresponding "0" bit, followed by as many "1" bits as the value of the counter.

5 A brief introduction to computability theory

One question that may arise when thinking about computers is: **Can a computer solve any type of problem?** And, if the answer to this question is "no": Which are the problems that cannot be solved by computers?

In fact, according to Nigel J. Cutland (1944–), in his book "*Computability: An introduction to recursive function theory*", Cambridge University Press (Cutland, 1980),

> *"We could describe computability theory, from the viewpoint of computer science, as beginning with the question What can computers do in principle (without restrictions of space, time or money)?—and, by implication—What are their inherent theoretical limitations?"*

Hence, one of the objectives of the field of computability theory is precisely to study which problems are solvable by computers and which are not. However, to achieve this goal, we need first to have a precise definition of what do we mean by a **digital computer**. In this section, we give a brief overview of some important computation models.

5.1 Automata

5.1.1 Some notation

We define an **alphabet** as a nonempty finite set. We refer to the members of the alphabet as **symbols**. Usually, and without loss of generality, we consider the alphabet $\Sigma = \{0, 1\}$. A **string** over an alphabet is a finite sequence of symbols from that alphabet. For example, if $\Sigma = \{0, 1\}$, then 011101 is a string over the alphabet Σ. If s is a string, then we represent by $|s|$ the length (i.e., number of symbols) of s. We denote by Σ^* the set of all finite-length strings, i.e.,

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\},$$

where ϵ denotes the empty string, i.e., $|\epsilon| = 0$.

For convenience, we assume an ordering of the strings, such that they appear ordered by size and, for the same size, ordered lexicographically (this is the ordering used to display the set above). Note that one such ordering allows to put all finite-length strings into a one-to-one correspondence with the integers and, hence, to treat strings as integers.

A **language**, A, is a subset of Σ^*, i.e., $A \subseteq \Sigma^*$. A language is **prefix-free** if no string of the language is a proper prefix of another string of the language.

5.1.2 Finite-state machines

A finite-state machine (or finite automaton) is a computation model with a very limited amount of memory. However, it is very useful for recognizing patterns in data.

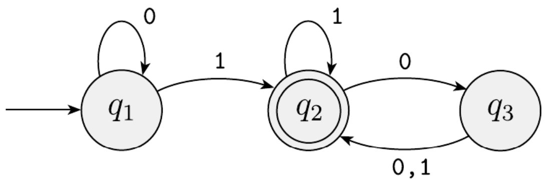


Figure 5.1: Example of a three-state finite-state machine (Sipser, 2012).

Example 5.1
*Consider the finite-state machine displayed in Fig. 5.1, M_1, with three states (q_1, q_2 and q_3), where state q_2 is an **accepting state**.*

*For example, this machine **accepts** string 1101, but **rejects** string 101000. In fact, it accepts all strings that end with a 1 or that end with an even number of 0s following the last 1.*

Definition 5.1 (Finite-state machine)
A finite-state machine (or finite automaton) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. *Q is a finite set called the states,*
2. *Σ is a finite set called the alphabet,*
3. *$\delta : Q \times \Sigma \to Q$ is the transition function,*
4. *$q_0 \in Q$ is the start state, and*
5. *$F \subseteq Q$ is the set of accept states.*

Example 5.2
Referring to Fig. 5.1, we have

1. *$Q = \{q_1, q_2, q_3\}$,*
2. *$\Sigma = \{0, 1\}$,*
3. *δ is described as*

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. *q_1 is the start state, and*

5. $F = \{q_2\}$.

If A is the set of strings accepted by M_1, then we say that M_1 **recognizes** or **accepts** the language A and we write $A = L(M_1)$. The language recognized by the finite automaton of Fig. 5.1 is

$$A = \{w|w \text{ contains at least one 1 and an even number of 0s follow the last } 1\}.$$

Definition 5.2 (Regular language)
A language is called a ***regular language*** *if some finite automaton recognizes it.*

Problem 5.1
Design a finite-state machine that recognizes the language

$$A = \{w|w \textit{ contains the string } 001\}.$$

Definition 5.3
Let A and B be languages. The regular operations ***union***, ***concatenation*** *and* ***star*** *are defined according to:*

Union: $A \cup B = \{x|x \in A \textit{ or } x \in B\}$;

Concatenation: $A \circ B = \{xy|x \in A \textit{ and } y \in B\}$;

Star: $A^* = \{x_1x_2 \dots x_k|k \geq 0 \textit{ and each } x_i \in A\}$.

Theorem 5.1
The collection of regular languages is closed under all three regular operations.

Problem 5.2
The complement of a language A, $\bar{A}$, is the language containing all the strings not in A, i.e.,

$$\bar{A} = \{x|x \notin A\}$$

Show that if A is regular, then $\bar{A}$ is also regular.

Problem 5.3
Using the result of the previous problem, show that the regular languages are closed under intersection.

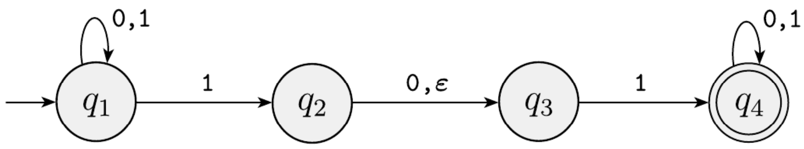


Figure 5.2: Example of a nondeterministic finite automata (Sipser, 2012).

5.1.3 Nondeterministic finite-state machines

In the case of nondeterministic finite automata (see Fig. 5.2), it is possible to have more than one exit from a certain state, for the same input symbol. When this happens, the machine splits into multiple copies of itself and follows *all* possibilities in parallel. Also, a special transition symbol ϵ indicates that the transition is made without reading any input symbol.

If the next input symbol does not appear on any of the arrows exiting from the current state, then that copy of the machine dies (terminates). If any one of the copies is in an accept state at the end of the input, then the machine accepts the input string.

Example 5.3
The automaton displayed in Fig. 5.2 accepts strings that contain either 101 or 11 as a substring. It is defined according to:

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$, ($\Sigma_\epsilon = \{0, 1, \epsilon\}$)
3. δ *is described as*

	0	1	ϵ
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

4. q_1 *is the start state, and*
5. $F = \{q_4\}$.

Theorem 5.2
Deterministic and nondeterministic finite automata recognize the same class of languages, i.e., they are equivalent.

Discussion topic 5.1
The language $\{0^n 1^n | n \geq 0\}$ *is not regular. Can you find why?*

And what about language $\{ww | w \in \Sigma^*\}$*? Is it regular? Does your answer depend on* Σ*?*

Theorem 5.3
A language is regular if and only if some regular expression describes it. In other words, regular expressions and finite automata are equivalent in their descriptive power.

Problem 5.4
Find the languages described by the following regular expressions, assuming $\Sigma = \{0, 1\}$:

1. 0^*10^*
2. $\Sigma^*1\Sigma^*$
3. $(\Sigma\Sigma)^*$
4. $1^*(01^+)^*$ — *Note: a^+ is sometimes used as a shorthand for aa^**

Problem 5.5
Find the regular expression equivalent to the regular language

$$A = \{w|w \textit{ has an equal number of occurrences of } 01 \textit{ and } 10 \textit{ as substrings}\}.$$

5.1.4 Pushdown automata

Example 5.4
Consider the following list of rules,

1. $A \to 0A1$
2. $A \to B$
3. $B \to \#$

which is an example of a ***context-free grammar****.*

A grammar is a collection of substitution rules, comprising variables on the left-hand side (the A and B in the example), and sequences of variables and alphabet symbols (in the example, the alphabet is $\{0, 1, \#\}$) on the right-hand side. The derivation of a string starts at the first rule and continues until not having more variables to substitute. For example, using the grammar of the previous example, we may obtain

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000\#111.$$

All strings obtained using this procedure form the **language of the grammar**. Any language generated by a context-free grammar is called a **context-free language**.

Problem 5.6
Can you find why the grammar described in the previous example generates the language $\{0^n\#1^n | n \geq 0\}$*?*

Discussion topic 5.2
Consider the grammar

$$S \rightarrow aSb \mid SS \mid \epsilon$$

and discuss the meaning of the corresponding context-free language if a *means the left parenthesis symbol, i.e., "(", and* b *means the right parenthesis symbol, i.e., ")". Note that the symbol "|" means "or", allowing a compact way of describing several rules starting with the same variable. Also, recall that* ϵ *represents the empty string.*

A **pushdown automaton** is like a nondeterministic finite automaton, but with a **stack** (see Fig. 5.3). Pushdown automata are equivalent in power to context-free grammars and more powerful than finite automata. The **stack** is a valuable resource, because it can hold an unlimited amount of information. For example, a finite automaton is unable to recognize the language $\{0^n1^n | n \geq 0\}$, because it cannot store very large numbers in its finite memory.

Problem 5.7
Can you see why the stack of the pushdown automaton allows the recognition of language $\{0^n1^n | n \geq 0\}$*?*

Pushdown automata can also be deterministic. However, contrarily to finite automata, they are not equivalent in power to the nondeterministic counterpart. Moreover, only the nondeterministic version is equivalent to the whole class of context-free grammars.

Theorem 5.4
A language is context free if and only if some pushdown automaton recognizes it.

Definition 5.4
A (nondeterministic) pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$*, where*

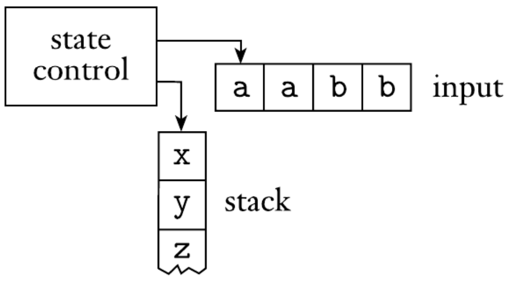


Figure 5.3: Schematic representation of a pushdown automaton (Sipser, 2012).

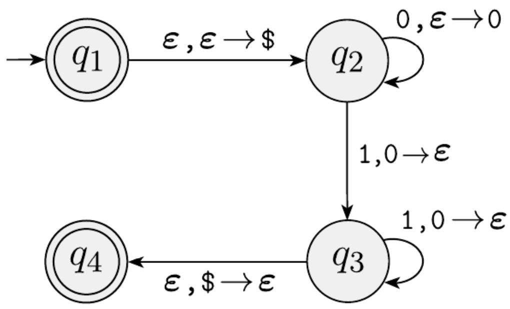


Figure 5.4: Example of a pushdown automaton that is able to recognize the language $\{0^n1^n|n \geq 0\}$ (Sipser, 2012).

1. *Q is a finite set of states,*
2. *Σ is the finite input alphabet,*
3. *Γ is the finite stack alphabet,*
4. *$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function, $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$, $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$*
5. *$q_0 \in Q$ is the start state, and*
6. *$F \subseteq Q$ is the set of accept states.*

Example 5.5
Figure 5.4 shows a pushdown automaton for recognizing language $\{0^n1^n|n \geq 0\}$.

Problem 5.8
Find a pushdown automaton for the language $\{ww^r|w \in \{0,1\}^*\}$.

Note 1: w^r denotes the string w reversed, i.e., written from right to left.

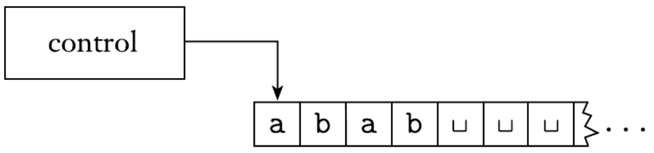


Figure 5.5: Schematic representation of a Turing machine.

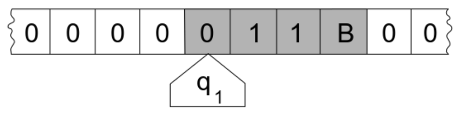


Figure 5.6: Example of a tape and head of a Turing machine. The q_1 written inside the head indicates the current internal state of the machine.

Note 2: Only a nondeterministic pushdown automaton is able to represent this language, because the machine has to "guess" the middle of the input string, in order to decide if the second half is equal to the first half reversed.

Discussion topic 5.3
Consider the language $\{w\#w | w \in \{0,1\}^*\}$. Is it a context-free language? Why?

Do you think that if, instead of having only one stack, the machine was equipped with two stacks, its computing power would be changed? Why?

5.2 Turing machines

As in the case of finite automata and pushdown automata, a Turing machine is also composed of a control based on a finite, non-empty, set of states. However, instead of having a very limited amount of memory present as in a finite automata, or an unlimited amount of memory, but of restricted access (last-in, first-out), a Turing machine has access to an unlimited and unrestricted amount of memory (Fig. 5.5).

A Turing machine is a mathematical model (or abstraction) of a hypothetical computing device where:

1. At each step, the machine is in one of a predefined finite (non-empty) set of states.
2. Symbols from a predefined finite (non-empty) set can be both read from and written on a tape (see Fig. 5.6, for an example).
3. The read-write head can move both to the left and to the right.
4. The tape is unlimited.

5. At start, the tape is blank, except for some finite number of squares.

6. The machine stops when a special state (the halting state) is encountered or when it fails to find a continuation state.

7. The special states for rejecting and accepting strings of a language, that we have seen when the automata were addressed, in the case of Turing machines take effect immediately after halting, even without having exhausting the input. There might be a specific halting state for accepting and another one for rejecting, or just a single halting state and the interpretation of acceptance or rejection left written on the tape.

According to Marvin Minsky (1927-2016), in his book "*Computation: Finite and Infinite Machines*", Prentice-Hall, Inc. (Minsky, 1967):

> *"A Turing machine is a finite-state machine associated with an external storage or memory medium. This medium has the form of a sequence of squares, marked off on a linear tape. The machine is coupled to the tape through a head, which is situated, at each moment, on some square of the tape (Fig. 5.6). The head has three functions, all of which are exercised in each operation cycle of the finite-state machine. These functions are: reading the square of the tape being "scanned," writing on the scanned square, and moving the machine to an adjacent square (which becomes the scanned square in the next operation cycle)."*

Discussion topic 5.4
Consider again the language $\{w\#w|w \in \{0,1\}^*\}$, which cannot be generated by a context-free grammar. Using its unlimited and unrestricted memory capability, can you sketch a way of deciding if a certain string is in this language or not, using a Turing machine?

More formally, a Turing machine can be described as follows.

Definition 5.5
A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, where

1. *Q is a finite set of states,*

2. *Σ is the finite input alphabet, not containing the blank symbol, $\sqcup$,*

3. *Γ is the finite tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma \backslash_{\{\sqcup\}}$,*

4. *$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,*

5. *$q_0 \in Q$ is the start state,*

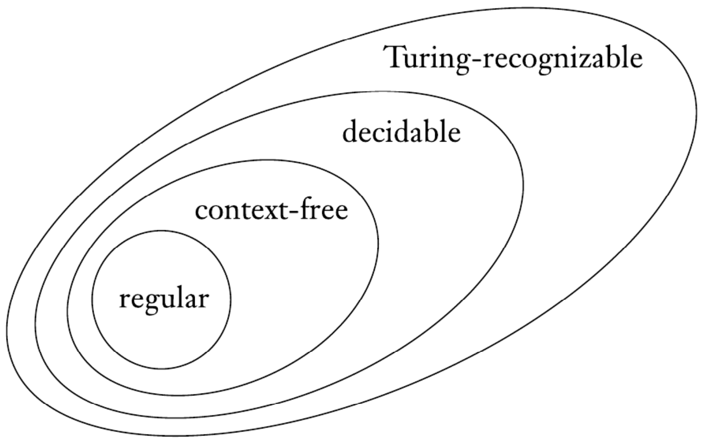


Figure 5.7: Hierarchy of the classes of languages (Sipser, 2012).

6. *$q_a \in Q$ is the accept state, and*
7. *$q_r \in Q$ is the reject state, with $q_r \neq q_a$.*

A Turing machine equipped with accepting and rejecting states, started with a certain string written on the tape, may have one of three possible behaviours: it stops at an accepting state; it stops at a rejecting state; it loops forever. Note that running forever is not possible in a finite automata or in a pushdown automata, because, by construction, these computation models stop after reading the input string. Contrarily, Turing machines may run forever, never halting.

Definition 5.6
*A language is **Turing-recognizable / recursively enumerable** if some Turing machine recognizes it.*

This means that a Turing machine recognizes a language if it accepts every string in the language and rejects or does not halt on the strings not in the language.

Definition 5.7
*A language is **Turing-decidable / decidable / recursive**, if some Turing machine decides it.*

This means that a Turing machine decides a language if it accepts every string in the language and rejects every string not in the language. Figure 5.7 shows the hierarchical relation among languages.

There are many variants of Turing machines, for example, using more than one tape/head, using one-way or two-way unlimited tapes, using alphabets of different sizes and with different symbols, and even using nondeterminism. However, all of them share the same computational power.

In fact, there are a number of remarkable facts about Turing machines, some of which are the following:

- A Turing machine can be restricted to a two-symbol alphabet, without lossing any of its computational power. For example, suppose that a certain Turing machine T uses k different symbols and that $n = \lceil \log k \rceil$. Then, we may construct another Turing machine, T', replacing each symbol of T by its n-bit binary representation and regarding the tape of T' as a sequence of n-bit blocks. Of course, to attain equivalence between T and T' it is also necessary to change appropriately the finite-state machine of T' (which will necessarily have more states than machine T).

- It can also be shown that Turing machines which have tapes that are unlimited in both directions have the same computing power as machines having an unlimited tape in only one direction.

- It is neither an advantage (nor a disadvantage) to have multiple tapes in a Turing machine—it can always be converted into an equivalent single tape machine.

- Probably more surprising is the fact that any Turing machine is equivalent to a Turing machine with only two internal states. Of course, this can only be achieved by enlarging the alphabet of the tape symbols.

The idea behind a Turing machine was first proposed by Alan Turing (1912-1954), in his famous paper:

> Alan Turing, "*On Computable Numbers, with an Application to the Entsgheidungsproblem*", Proceedings of the London Mathematical Society (Turing, 1936).

Almost simultaneously, Alonzo Church (1903-1995) proposed a different, but equivalent, notion of computability, using what became known as λ-calculus:

> Alonzo Church, "*An Unsolvable Problem of Elementary Number Theory*", American Journal of Mathematics (Church, 1936).

In their attempt to solve some problems related to the foundations of mathematics, both had the need to rigorously defining the concept of "computation". Note that, at that time, a "computer" was always seen as a human computer, i.e., someone that performs pencil-and-paper calculations.

Basically, we say that something is **computable** if it can be produced, in finite time, by a process equivalent to our usual intuitive notion of **algorithm** or **effective procedure**. This definition was proposed by Alan Turing in his 1936 paper, where he also refers as equivalent the idea of “effective calculability” of Alonzo Church.

Again, from Minsky (1967):

> *“The idea of an algorithm or effective procedure arises whenever we are presented with a set of instructions about how to behave. This happens when, in the course of working on a problem, we discover that a certain procedure, if properly carried out, will end up giving us the answer. Once we make such a discovery, the task of finding the solution is reduced from a matter of intellectual discovery to a mere matter of effort; of carrying out the discovered procedure—obeying the specified instructions.”*

This notion of algorithm motivated a conjecture, known as the **Church-Turing thesis**, stating that something is computable by a human being following an algorithm (of course, ignoring resources, such as time, number of sheets of paper and pencils), if and only if it is computable by a Turing machine. There is no proof for this claim. However, 80 years have passed since the papers of Church and Turing and no counterexample has yet been found!

Example 5.6
Consider the following example of a Turing machine, taken from the original paper of Turing (Turing, 1936). The set of instructions for the machine is:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
b	*None*	*P0, R*	*c*
c	*None*	*R*	*e*
e	*None*	*P1, R*	*f*
f	*None*	*R*	*b*

If started on an empty tape, the machine writes alternating 0’s and 1’s on the tape, leaving a blank square between each pair of digits.

Problem 5.9
What happens if, in the previous example, the tape is not totally blank?

Example 5.7
This example is from Minsky (1967). A certain Turing machine is described by the following

six quintuples:

$$\begin{array}{l}(q_0,\ \ 0,\ \ q_0,\ \ 0,\ \ R)\\(q_0,\ \ 1,\ \ q_1,\ \ 0,\ \ R)\\(q_0,\ \ B,\ \ H,\ \ 0,\ \ -)\\(q_1,\ \ 0,\ \ q_1,\ \ 0,\ \ R)\\(q_1,\ \ 1,\ \ q_0,\ \ 0,\ \ R)\\(q_1,\ \ B,\ \ H,\ \ 1,\ \ -)\end{array},$$

where H indicates a halting state. Moreover, it is assumed that the machine starts in state q_0.

This machine outputs a 1 or a 0 depending on whether the number of 1's in a string of 1's and 0's is odd or even. The machine should be started with the head positioned over the leftmost 1 written on the tape. Also, a B should be inserted in the tape beyond the rightmost 1.

Suggestions: Simulate (by hand) a few cases and see how the machine operates. Eventualy, you will find out that the input sequence is erased during the computation. Modify the operation rules in order to not erase the original sequence.

In this example (and also in the previous one), the machine always moves to the right. Hence, it cannot use the tape as auxiliary memory. In fact, with this restriction, it does not have more power than a simple **finite-state automaton**—are you able to design a finite-state automaton that implements this same operation, i.e., parity checking?

Example 5.8
One more example from Minsky (1967). In this case, the machine, intended to do parenthesis checking, is described in a slightly different way:

Q	S	Q'	S'	D
0	)	1	X	0
0	(	0	(	1
0	A	2	A	0
0	X	0	X	1

Q	S	Q'	S'	D
1	)	1	)	0
1	(	0	X	1
1	A	H	0	−
1	X	1	X	0

Q	S	Q'	S'	D
2	)	*impossible*		
2	(	H	0	−
2	A	H	1	−
2	X	2	X	0

The beginning and end of the expression that is written on the tape are marked by A's—for example, "$A(()())A$".

As in the previous example, try a couple of examples and find out how the machine operates, including how the output of the computation is provided. For start, discover where should the tape head be positioned when the computation begins (as before, in state 0)...

Recall that this problem (parenthesis consistency checking) cannot be solved by a finite-state automaton, but can be solved by a **push-down automaton**.

One of the most important results included in Turing's paper (Turing, 1936) shows that it is possible to build **universal Turing machines**. A universal Turing machine is a machine that can simulate any other Turing machine. A direct consequence of this discovery is the notion of stored-program computer. All but the most trivial computers are universal, in the sense that they can mimic the actions of other computers. Nevertheless, it is possible to construct a universal standard Turing machine using 4 states and 6 symbols. Moreover, it was recently suggested (although still debated) that a universal Turing machine without a halting state is possible using 2 states and 3 symbols!

5.3 Computable and non-computable functions

Let us consider the following general definition of **function** (Minsky, 1967):

> *"... we will think of a function as an association between arguments and values— as a set of ordered pairs $\langle x, y\rangle$ such that there is just one pair for each x. For each function there may be many definitions or rules that tell how to find the value y, given the argument x. Two definitions or rules are equivalent if they define the same function. It may be very difficult, given two different rules, to tell if they are in fact equivalent. In fact it may, in a sense, be impossible... "*

It should now be evident that it is possible to define functions in terms of the behaviour of Turing machines, for example, considering as argument of the function the content of the tape when the machine is started and as value of the function the content of the tape when the machine halts.

Notice that, without loss of generality, we may interpret the content of the tape as a binary string and, by means of an appropriate one-to-one mapping between non-negative integers and strings, interpret them also as non-negative integers. In fact, as mentioned before, it is always possible to convert a certain Turing machine into another one equivalent, but with tape alphabet $\Sigma = \{0, 1\}$.

We recall that we denote by Σ^* the set of all finite-length binary strings and that we assume an ordering of these strings according to

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, \ldots,$$

where ϵ denotes the empty string. One such ordering allows to put all finite-length strings into a one-to-one correspondence with the non-negative integers and, hence, to treat strings as numbers. Hence, it is possible to view the function defined by a Turing machine, T, as a $\mathbb{N} \to \mathbb{N}$ numerical function. A numerical function for which there exists an algorithm that allows the calculation of its values is said to be a **computable function**.

Definition 5.8 (Turing-computable functions (Minsky, 1967))
A function $f(x)$ will be said to be Turing-computable if its values can be computed by some Turing machine T_f whose tape is initially blank except for some standard representation of the argument x. The value of $f(x)$ is what remains on the tape when the machine stops.

Definition 5.9 (Computable function (Sipser, 2012))
A function $f : \Sigma^* \to \Sigma^*$ is a computable function if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

It is also worth mentioning that, from a computational point of view, it is straighforward to relate functions with languages, by considering a language

$$L_f = \{(x, y) | y = f(x)\},$$

and then study the decidability of L_f instead of the computability of f.

Discussion topic 5.5
Consider the function

$$g(n) = \begin{cases} 1 & \textit{if there is a run of exactly } n \textit{ consecutive 7s in the decimal expansion of } \pi \\ 0 & \textit{otherwise.} \end{cases}$$

Do you think that it is possible to find and algorithm able to compute function g? Why?

5.3.1 The diagonalization argument

Problem 5.10
Consider a $n \times n$ matrix of numbers, M. What is the smallest set of values from M that allow us to build a vector of n numbers that does not coincide with any of the rows of M?

Consider all possible Turing machines. Because a Turing machine is defined by a finite structure (the input alphabet, the tape alphabet, the set of possible states and the transition table), then it is possible to encode each Turing machine into a different finite-length binary string. In other words, the Turing machines are countable (as the set Σ^* is) and we can index them. So, let us denote by T_i the Turing machine that appears in position i in the string ordering. Now, consider Table 2, in which each row corresponds to a Turing machine and each column indicates if the machine accepts the string (a "1" is shown in the table) or it rejects or does not halt (in that case, a "0" is shown).

Let us now build a Turing machine, T', that results from complementing the diagonal of Table 2. According to the example of Table 2, that machine would have the following behaviour:

	ϵ	0	1	00	01	10	$\cdots$	001	$\cdots$
T'	1	1	1	0	1	0	$\cdots$	1	$\cdots$

	ϵ	0	1	00	01	10	$\cdots$	001	$\cdots$
T_1	**0**	0	1	1	0	0	$\cdots$	1	$\cdots$
T_2	1	**0**	0	1	0	0	$\cdots$	0	$\cdots$
T_3	1	1	**0**	1	1	1	$\cdots$	0	$\cdots$
T_4	0	1	1	**1**	0	1	$\cdots$	1	$\cdots$
T_5	1	0	1	1	**0**	1	$\cdots$	0	$\cdots$
T_6	0	0	0	1	0	**1**	$\cdots$	1	$\cdots$
$\vdots$							$\ddots$		
T_9	0	1	1	1	0	1	$\cdots$	**0**	$\cdots$
$\vdots$									

Table 2: The diagonalization argument, showing that there are more functions than possible Turing machines. The Turing machine that behaves according to the complement of the diagonal is not in this table. However, we assumed that **all** Turing machines are in the table!

This machine T' cannot be in the listing of Table 2 and, therefore, there must be non-computable functions or, equivalently, undecidable languages. In fact, there are much more non-computable than computable functions, because the set of functions $f : \mathbb{N} \to \mathbb{N}$ (or, equivalently, $f : \{0,1\}^* \to \{0,1\}^*$) is uncountable (notice that, for each argument of f, there are infinitely many possibilities!).

The diagonalization argument was introduced by Georg Cantor (1845–1918) to prove that the cardinality of the set of real numbers, $\mathbb{R}$, denoted as $\mathfrak{c} = |\mathbb{R}|$, is strictly greater than the cardinality of the natural numbers, $\mathbb{N}$, denoted as $\aleph_0 = |\mathbb{N}|$. This argument was also used by Cantor to show that the **power set** of a set S, i.e., the set of all subsets of S, denoted by $\mathcal{P}(S)$ has cardinality strictly greater than S, i.e., $|\mathcal{P}(S)| > |S|$. This result is known as Cantor's theorem. Note that for a finite set S, the number of possible subsets (including the empty set and the set itself) is equal to $2^{|S|}$. Cantor also showed that $2^{|\mathbb{N}|} = 2^{\aleph_0} = \mathfrak{c}$, i.e., the power set of the natural numbers has the same cardinality of the reals. Finally, note that Cantor's theorem implies the existence of and infinite number of different infinities!

Discussion topic 5.6

Consider functions that, for each integer argument, answer with "0" or "1", i.e., belong to the set $\mathcal{F} = \{f|\mathbb{N} \to \{0,1\}\}$. *Relate* $\mathcal{F}$ *with the interval* $[0,1]$ *of the real line and, hence, show that* $\mathcal{F}$ *is uncountable.*

Turing machines permit "only" the implementation of a special class of functions, known as **partial recursive functions**. A function f is partial if it is not defined for some elements of its domain. For example, the computer pseudo-code

```
read(x)
if(x == 1)
    print(x)
else
    while(true)
```

computes the partial function

$$f(x) = \begin{cases} 1 & \text{if } x = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

5.3.2 Primitive recursive functions

A **primitive recursive function** is a (computable and total) function that can be implemented using only loops of the form

```
for i=1 to n
    ...
end
```

where “n” is fixed in advance (i.e., before the loop starts) and “i” can only be modified by the control of the loop. In other words, the number of times the loop executes is determined and fixed in advance. Therefore, it is not possible to have “while” loops that terminate based on conditions, nor “goto” statements that can jump back to an arbitrary point in the program, nor recursive function calls. These conditions render impossible to end up having infinite loops.

However, even though all programs obeying to these conditions do terminate, there are terminating programs that cannot be written using these restrictions. The Ackermann function (see below) is one of them. It is possible to use the diagonalization argument to show that there should exist computable functions that are not primitive recursive. For simplicity (but without loss of generality), suppose functions $p_i : \mathbb{N} \to \mathbb{N}$, and the list of all primitive recursive functions $p_1, p_2, \ldots$. Consider now the function $f(n) = p_n(n) + 1$. Since, by definition, all p_i halts, then f also always halts. However, $f \neq p_i, \forall_i$. Hence, f is an example of a function that, although not primitive recursive, halts.

5.3.3 Computable functions that are not primitive recursive

The Ackermann function (Wilhelm Ackermann, 1896–1962), is one of the simplest and earliest-discovered examples of a total computable function that is not primitive recursive. This function illustrates that not all total computable functions are primitive recursive. One example in this family of functions is

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

It may not be obvious that the computation of $A(m, n)$ always terminates. However, as can be verified, the recursion is bounded, because in each recursive call either m decreases, or m remains the same and n decreases. Moreover, each time n reaches zero, m decreases, so m eventually reaches zero. However, when m decreases, there is no upper bound on how much n can increase.

5.4 The halting problem

One of the most famous problems in computability theory, that cannot be systematically solved by a computer, is to decide if another program, `prog`, will halt for a certain input `x`. Next we show that such program cannot exist.

Let us consider that such program, `will_halt(prog,x)`, exists. Then, consider a program `prog`, such that

```
prog(x)
    if(will_halt(x,x) == true)
        while(true)
    else
        exit
```

Now, consider what happens if `prog` is called, having as argument a description of itself, i.e., what happens when executing `prog(prog)`. Notice that, inside `prog`, there is a call to `will_halt(prog,prog)`. If this call returns `true`, i.e., if our hypothetical program that decides if a program will halt for a certain input says that it will halt, then `prog` enters an infinite loop and does not halt. On the contrary, if `will_halt(prog,prog)` returns `false`, i.e., if it indicates that `prog` will not stop when called with itself as argument, then it stops. This contradiction shows that it is not possible to build a program that, for all possible cases, verifies if some other program halts or not.

5.5 The busy beaver game

In 1962, the Hungarian mathematician Tibor Radó (1895–1965) published the paper

> Tibor Radó, "*On Non-Computable Functions*", Bell System Technical Journal (Radó, 1962)

where he introduced a game, named **The Busy Beaver**, and used it to give simple examples of non-computable, although well-defined, functions.

In this game, we consider Turing machines with a binary alphabet, i.e., the tape can only have symbols from $\{0, 1\}$. Because the empty space is not allowed, we consider that the tape is initially filled with the symbol "0". The game consists of having n cards (in fact, these cards

C_1	
0	1L2
1	1R3

C_2	
0	1R1
1	1L2

C_3	
0	1R2
1	1L0

Figure 5.8: Example of a 3-card busy beaver game, adapted from Radó (1962).

correspond to states of the Turing machine) of the form represented in Fig. 5.8. The first column of each card contains the symbols of the tape alphabet, in this case "0" or "1". On the right-hand column, we find the actions that the machine should execute when it reads the symbol on the corresponding left-hand column. The first is the symbol that should be written, the second indicates to which side should the head move and the third one to which card (i.e., state) should the machine go ("0" indicates that the machine should enter the "halt" state).

The game runs as follows. The machine is started at state number 1 (card 1) and runs until reaching a possible halting state. However, it may also run forever. If it stops after s shifts (or steps, because in each step it is always required to shift left or right), the number of ones written in the tape is annotated as the score of the machine. The n-state machines achieving the highest scores are called n-state busy beavers.

Problem 5.11
Consider the machine described in Fig. 5.8. Does it stop? If it does, determine its score and the number of steps required to stop.

Problem 5.12
Design a machine that writes an infinite number of ones in the tape.

Problem 5.13
Show that there are $(4(n+1))^{2n}$ different n-state machines (considering the halting state as an additional state).

Problem 5.14
Show that the number of different functions implemented by the different n-state machines is smaller than $(4(n+1))^{2n}$.

Hint: Try to show that there are at least two different machines implementing the same function.

The score of the n-state busy beavers is denoted by the function $\Sigma(n)$. Currently, the values of this function are known only for $n < 5$. They are: $\Sigma(1) = 1$, $\Sigma(2) = 4$, $\Sigma(3) = 6$ and $\Sigma(4) = 13$. Moreover, it is also known that $\Sigma(5) \geq 4098$ and that $\Sigma(6) \geq 3.5 \times 10^{18267}$ (see Fig. 5.9).

C_1	
0	1R2
1	1L5

C_2	
0	1R3
1	1R6

C_3	
0	1L4
1	0R2

C_4	
0	1R5
1	0L3

C_5	
0	1L1
1	0R4

C_6	
0	1L0
1	1R3

Figure 5.9: This 6-state machine writes $\approx 3.515 \times 10^{18267}$ ones on the tape before stopping. To achieve it, it needs to execute $\approx 7.412 \times 10^{36534}$ computation steps...

Besides introducing this game, Radó proved that the function $\Sigma(n)$ is non-computable, by showing that it grows faster than every possible computable function.

Problem 5.15

Show that, in a n-state machine, $\Sigma(n) \geq n$.

Hint: Try to build examples of n-state machines, with n small, that print n ones.

Discussion topic 5.7

A result, known as Rice's theorem, says that all non-trivial semantic properties of programs (i.e., those concerning the behavior of the program) are undecidable (i.e., non-computable). Try to find out more about this theorem and discuss its implications.

6 An algorithmic measure of information

6.1 Motivation

As shown previously, the formulation of a probabilistic measure of information explores statistical regularities of the data, by means of probabilistic modeling of the information sources. However, there are regularities that cannot be captured by probabilistic models alone. For example, several approaches involving the statistical analysis of the initial digits of π have failed to reveal significant deviations from what is usually understood as statistical **randomness**. Nevertheless, the initial sequence of digits of π can be generated by simple programs, showing that the number π is far from being random! In fact, intuitively, we may say that a string is random if it cannot be **compressed**, i.e., if it is not possible to find a **shorter description** for it. This is precisely the core idea behind the algorithmic measure of information.

Discussion topic 6.1
In data compression, what is the role of the decompressor and of the string representing the compressed data? Does it make sense to consider the decompressor as an "interpreter" for a certain language, formed by all possible compressed strings?

Example 6.1
Consider the following three binary strings:

1. $0101010101010101010101010101010101010101010101010101010101010101$
2. $0110101000001001111001100110011111110011101111001100100100001000$
3. $1101111001110101111101101111101110101101111000101110010100111011$

What should be the shortest descriptions for them? The first one seems to be easy to describe. Apparently, the second one looks random, but actually it is just the initial binary representation of $\sqrt{2}-1$. *The third one seems random and probably is (however, in general, it is impossible to prove it!).*

Now, consider the representation of a sequence generated by flipping a fair coin n *times, such as*

$$00100101110101010\ldots010$$

There are 2^n *such sequences and they are equally probable. It is highly likely that such sequence cannot be compressed at all. Therefore, there might not be a better (shorter) program for generating this sequence as "print* $00100101110101010\ldots010$*". Thus, the descriptive complexity of a random binary sequence is as long as the sequence itself.*

Suppose that we want to represent a certain object by a finite string of bits. Generally, an object may have several possible representations (or descriptions), but each representation should describe only one object (we want to unambiguously reconstruct the object from its description). Then, the length of the shortest of all descriptions available for the object can be used as a measure of its **complexity** (and, in fact, of its randomness).

However, the meaning of "**description**" must be defined precisely or otherwise we may run into similar problems as the one known as the **Berry paradox**, where a certain number is defined as "*the smallest positive integer not definable in fewer than twelve words*". If this number exists, then we have just described it in eleven words, which contradicts its definition! On the other hand, if the number does not exist, then we have to conclude that all positive integers can be described in less than twelve words!

The approach that follows relies on algorithms to measure the quantity of information conveyed by a certain string, x. One way to address this problem could involve building a (compact) Turing machine, T_x, that, when started on a blank tape, would write the string x on the tape and halt. Then, the description of x would be a string describing T_x, $\langle T_x \rangle$, using some fixed encoding for the Turing machines and, therefore, the length of the description would be $|\langle T_x \rangle|$.

However, instead of starting the machine on a blank tape, another possible approach is using an auxiliary string written on it. In this case, the string x is described using a Turing machine M and an input w that is used by M, and the description of x is given by some encoding $\langle M, w \rangle$ (note that, in principle, this encoding needs to include information that permits separating $\langle M \rangle$ from w).

6.2 Kolmogorov complexity

Assume that a certain specification method, f, associates at most an object, x, with a description, p. We can think of f as a decompression function (or program), from a set of descriptions, P, into the set of objects, X, i.e., $f : P \to X$. To be useful, descriptions should be finite. This implies that **it is only possible to describe a countable number of different objects**.

For most functions f, the length of the description of object x according to f depends not only on x, but also on f. However, in order to objectively assess the complexity of the objects, we need functions that assign a complexity value to x that depends on x alone. Moreover, if p is the description of x with respect to f, then x has to be generated from f and p through "**an effective procedure**" or, in other words, f has to be a computable function.

Definition 6.1 (Algorithmic complexity)
Given a computable function, f, we define the algorithmic complexity of an object,[7] *x, accord-*

[7]Without loss of generality, from now on we will refer to these objects as strings and also consider that they are binary, i.e., $x, p \in \{0,1\}^*$. Also, as referred before, we assume the usual string ordering, $\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$ and the implicit one-to-one correspondence between binary strings and non-negative integers (for example, the natural number "5" has the same meaning as the string "10").

ing to f, as

$$K_f(x) = \min\{|p| : f(p) = x\},$$

where $x, p \in \{0,1\}^*$, i.e., it corresponds to the minimum description length, over all descriptions that produce x through f. If there is no p able to produce x using f, then we say that $K_f(x) = \infty$ for such x. Note that, although computable, f is not required to be a total function.

Notice that we can think of p as a program and f as a (specialized) computer. If $K_f(x) = \infty$, then it is not possible to produce x with the computer f. Also, notice that, without the computability requirement, f could be any arbitrary partial function from strings to strings, complying only to the condition that $|\{x : K_f(x) = k\}| \leq 2^k$, where k-bit strings are used as the description of strings x for which $K_f(x) = k$. Of course, this would allow us to trivially and arbitrarily find, for a certain x, a f such that $K_f(x) = 0$ (by mapping the empty string to it)!

Consider now r distinct specification methods, $f_1, f_2, \ldots, f_r$. It is easy to construct a new method, f, that assigns to each x a complexity $K_f(x)$ that exceeds only by about $\log r$ bits the minimum of $K_{f_1}(x), K_{f_2}(x), \ldots, K_{f_r}(x)$. The idea is to use the first $\log r$ bits of p to identify the method f_i and then the appropriate program for f_i. Hence, we can say that

$$K_f(x) \leq K_{f_i}(x) + c, \forall i,$$

i.e., that the description method f **minorizes** (additively) every description method f_i.

Definition 6.2
If two methods f and g minorize each other, i.e.,

$$f(x) \leq g(x) + c_1$$

and

$$g(x) \leq f(x) + c_2,$$

then they are called equivalent.

Definition 6.3
Let $\mathcal{C}$ be a subclass of the partial functions over $\mathbb{N}$. A function $f \in \mathcal{C}$ is additively optimal for $\mathcal{C}$ if

$$K_f(x) \leq K_g(x) + c_{f,g}, \quad \forall x, \forall g \in \mathcal{C},$$

where $c_{f,g}$ is a constant depending only on f and g (but not on x).

Clearly, all additively optimal description methods, f, g, are equivalent, according to

$$|K_f(x) - K_g(x)| \leq c_{f,g}.$$

Thus, asymptotically, the complexity of an object x, $K(x)$, when we restrict to optimal methods of specification, does not depend on accidental peculiarities of the chosen optimal method.

The development of the theory of algorithmic information (also known as **Kolmogorov complexity**) is made possible by the remarkable fact that the class of partial recursive (or computable) functions possesses a universal element that is additively optimal. Under this relatively natural restriction on the class of description methods (i.e., computability), we obtain a well-behaved hierarchy of complexities and hence a minimal element. Then, we will consider the class of description methods given by

$$\{\phi : \phi \text{ is a partial recursive function}\}.$$

Lemma 6.1
There is an additively optimal universal partial recursive function.

Proof. Let ϕ_0 be the function computed by a universal Turing machine, U. This machine expects inputs in the form

$$\langle n, p\rangle = \underbrace{11\ldots1}_{|n|}0np.$$

The total program $\langle n, p\rangle$ is a two-part code: the first part, is a self-delimiting encoding of T_n; the second, is the program, p, for T_n. Then, U can simulate T_n with program p, i.e., $\phi_0(\langle n, p\rangle) = \phi_n(p)$, and

$$K_{\phi_0}(x) \leq K_{\phi_n}(x) + c_{\phi_n},$$

where c_{ϕ_n} can be set to $2|n| + 1$. □

Definition 6.4 (Conditional algorithmic complexity)
Let $x, y, p \in \mathbb{N}$ (recall that this is the same as considering $x, y, p \in \{0, 1\}^*$, according to the usual enumeration of strings). Any partial recursive function ϕ, together with p and y, such that $\phi(\langle y, p\rangle) = x$, is a description of x. The complexity K_ϕ of x conditioned to y is defined by

$$K_\phi(x|y) = \min\{|p| : \phi(\langle y, p\rangle) = x\},$$

and $K_\phi(x|y) = \infty$ if there are no such p. Then, p is a program to compute x by ϕ, given y.

Theorem 6.1 (The invariance theorem)
There is an additively optimal universal partial recursive function, ϕ_0, for the class of partial recursive functions, to compute x given y. Therefore, $K_{\phi_0}(x|y) \leq K_\phi(x|y) + c_\phi$, for all partial recursive functions ϕ and all x and y, where c_ϕ is a constant depending on ϕ, but not on x or y.

Proof. Let ϕ_0 be the function computed by a universal Turing machine, U, such that U started on input $\langle y, \langle n, p\rangle\rangle$ simulates T_n on input $\langle y, p\rangle$. Hence, $\forall n$,

$$K_{\phi_0}(x|y) \leq K_{\phi_n}(x|y) + c_{\phi_n},$$

where $c_{\phi_n} = 2|n| + 1$. □

The key point of this theory is not that the universal description method necessarily gives the shortest description in each case, but instead that no other description method can improve on it infinitely often by more than a fixed constant. Also, for every pair of additively optimal functions, u and v,

$$|K_u(x|y) - K_v(x|y)| \leq c_{u,v}.$$

Let us now consider that u and v are universal computers and that p_u and p_v are the shortest programs, such that $u(p_u) = x$ and $v(p_v) = x$, i.e., $K_u(x) = |p_u|$ and $K_v(x) = |p_v|$. Since u is a universal computer, we may precede p_v by a program, e_v, that emulates v on u and obtain x using $u(e_v p_v)$. For identical reasons, we obtain $v(e_u p_u) = x$. Therefore, $\forall x \in \{0,1\}^*$,

$$K_u(x) \leq K_v(x) + |e_v|$$

and

$$K_v(x) \leq K_u(x) + |e_u|,$$

leading to

$$|K_u(x) - K_v(x)| \leq c_{u,v}.$$

Definition 6.5 (Kolmogorov complexity)
Let us fix an additively optimal universal ϕ_0 (the reference function) and define the conditional Kolmogorov complexity $K(x|y) = K_{\phi_0}(x|y)$. Also, fix a particular Turing machine (the reference machine), U, that computes ϕ_0. The unconditional Kolmogorov complexity is defined by

$$K(x) = K(x|\epsilon).$$

Theorem 6.2 (Upper bound of the complexity)
There is a constant, c, such that, $\forall x, y$,

$$K(x) \leq |x| + c, \;\; \textit{and} \;\; K(x|y) \leq K(x) + c.$$

Proof. For the first inequality, define a Turing machine, T, that copies the input to the output. Then, for all x, we have $K_T(x) = |x|$. To prove the second inequality, construct a Turing machine, T, that, for all y, z, computes x on input $\langle z, y\rangle$ iff the universal reference machine U computes x for input $\langle z, \epsilon\rangle$. Then, $K_T(x|y) = K(x)$. By the invariance theorem, there is a constant, c, such that $K(x|y) \leq K_T(x|y) + c = K(x) + c$. □

Theorem 6.3
The number of strings x with complexity $K(x) < k$ satisfies

$$|\{x \in \{0,1\}^* : K(x) < k\}| < 2^k.$$

Proof. There are not many short programs. If we list all the programs of length less than k, we have

$$\underbrace{\epsilon}_{1}, \underbrace{0,1}_{2}, \underbrace{00,01,10,11}_{4}, \ldots, \underbrace{\ldots, \overbrace{11\ldots1}^{k-1}}_{2^{k-1}}$$

and the total number of such programs is

$$1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1 < 2^k.$$

□

Theorem 6.4
There are strings of arbitrarily large Kolmogorov complexity, i.e., $\forall n \in \mathbb{N}$, $\exists x : K(x) > n$.

Proof. If this was not the case, then all of the infinitely many possible finite strings could be generated by the finitely many programs with a complexity below n bits. □

Theorem 6.5
For every computable function, f, there exists a constant, c, such that $K(f(x)) \leq K(x) + c$, for every x such that $f(x)$ is defined.

Definition 6.6
For each constant, c, we say that string x is c-incompressible if $K(x) \geq |x| - c$.

How many strings of length n are c-incompressible? Using a counting argument, we can see that there are, at least, one 0-incompressible string of length n, at least half of all length-n strings are 1-incompressible, at least three-fourths are 2-incompressible, and so on. In general, at least $(1 - 2^{-c})$ of the length-n strings are c-incompressible, showing that even considering a small c, that majority of the strings are c-incompressible.

Theorem 6.6
Let c be a positive integer. For each fixed y, every finite set A of cardinality m has at least $m(1 - 2^{-c}) + 1$ elements x with $K(x|y) \geq \log m - c$.

Proof. The number of programs of length less than $\log m - c$ is

$$\sum_{i=0}^{\log m - c - 1} 2^i = 2^{\log m - c} - 1.$$

Hence, there are at least $m - m2^{-c} + 1$ elements in A that have no program of length less that $\log m - c$. $\square$

Theorem 6.7 (Non-computability of $K(x)$)
The Kolmogorov complexity is not computable.

To show why the Kolmogorov complexity is not a computable function, consider the following pseudo-code, that calls a hypothetical function "`K`", that calculates the Kolmogorov complexity of the string in the nth position of the usual string ordering, $K(n)$:

```
print_complex_string(min_k)
    n = 0
    while(true)
        if(K(n) > min_k)
            print(string(n))
            exit
        else
            n++
```

Obviously, for a value of "`min_k`" greater than the total size of the programs involved, the program would print a string with Kolmogorov complexity greater than the size of the program used to generate it, which is a contradiction! Therefore, it is not possible to have a program that calculates the Kolmogorov complexity of arbitrary strings.

6.3 Information distances

The works of researchers such as Ray Solomonoff (1926–2009), Andrey Kolmogorov (1903–1987), Gregory Chaitin (1947–) and others (Solomonoff, 1964a,b; Kolmogorov, 1965; Chaitin, 1966; Wallace and Boulton, 1968; Rissanen, 1978), related to the definition of complexity measures, has found applications in several areas of knowledge. As shown, the Kolmogorov complexity of x, denoted as $K(x)$, is defined as the size of the smallest program that, on a universal reference machine, produces x and halts. A major drawback of the Kolmogorov complexity is that it is not computable. To overcome this limitation, it is usually approximated by a computable measure, such as Lempel-Ziv based complexity measures (Lempel and Ziv, 1976), linguistic complexity measures (Gordon, 2003) or compression-based complexity measures (Dix *et al.*, 2007). These approximations provide upper bounds on the Kolmogorov complexity.

Bennett et al. proposed an information distance (Bennett *et al.*, 1998), based on the Kolmogorov complexity, that minorizes, in an appropriate sense, every effective metric (Li *et al.*, 2004). It is defined as

$$E(x, y) = \max\{K(x|y), K(y|x)\}, \tag{6.1}$$

and represents the length of the shortest binary program that computes x from y and y from x. Because $E(x, y)$ is an absolute measure, it is not appropriate to assess similarity. Hence, a normalized version was proposed, overcoming this limitation (Li *et al.*, 2004). The normalized information distance (NID) is defined as

$$\mathrm{NID}(x, y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}, \tag{6.2}$$

and is a metric capable of measuring the similarity between two strings. However, because $K(x)$ is non-computable, alternatives have been devised in order to use it in practice.

One such alternative is the normalized compression distance (NCD) (Li *et al.*, 2004; Cilibrasi and Vitányi, 2005), defined as

$$\mathrm{NCD}(x, y) = \frac{C(x, y) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}, \tag{6.3}$$

where $C(x)$ and $C(y)$ represent, respectively, the number of bits of a compressed version of x and y, and $C(x, y)$ the number of bits of the conjoint compression of x and y (usually, x and y are concatenated). Distances near one indicate dissimilarity, while distances near zero indicate similarity.

In the normalized compression distance represented in (6.3), instead of the more obvious direct substitution of $K(x|y)$ by $C(x|y)$, a term corresponding to the conjoint compression of x and y, $C(x, y)$, was preferred. The main reason for adopting this form is that a direct substitution of K by C in (6.2) requires the availability of compressors that are able to produce conditional compression, i.e., $C(x|y)$ and $C(y|x)$. However, most compressors do not have this functionality and, therefore, the NCD avoids it by using suitable manipulations of (6.2) (Cilibrasi and Vitányi, 2005).

Usually, the $C(x, y)$ term is interpreted as the compression of the concatenation of x and y, but, in fact, it could adopt any other form of combination between x and y. Concatenation is often used because it is easy to obtain, but it is known that it may hamper the efficiency of the measure (Cebrián *et al.*, 2005).

To overcome this limitation, a normalized conditional compression distance (NCCD) was proposed, using compressors based on sets of image transformations (Nikvand and Wang, 2010, 2013). The NCCD is therefore computed using a direct substitution of $K(x|y)$ by $C(x|y)$ as follows:

$$\mathrm{NCCD}(x, y) = \frac{\max\{C(x|y), C(y|x)\}}{\max\{C(x), C(y)\}}. \tag{6.4}$$

Compression algorithms provide a natural way of approximating the Kolmogorov complexity, because, together with the appropriate decoder, a bitstream produced by a lossless compression algorithm can be used to reconstruct the original data. The number of bits required for representing these two components (decoder and bitstream) can be viewed as an estimate of the Kolmogorov complexity. Moreover, the search for better compression algorithms is directly related to the problem of improving the complexity bounds.

Successful applications of these principles have been reported in areas such as genomics, virology, languages, literature, music, handwritten digits and astronomy (Cilibrasi and Vitányi, 2005). In order to be suitable to approximate the Kolmogorov complexity, a compression algorithm needs to accumulate knowledge of the data while the compression is performed. It needs to be able to find dependencies, to collect statistics, i.e., it has to create an internal model of the data. The Lempel-Ziv compression algorithms belong to the class of methods that create internal data models. Another class that has been used, are the finite-context models (Pratas *et al.*, 2015; Pinho *et al.*, 2013; Garcia *et al.*, 2013; Pinho *et al.*, 2016; Pinho and Ferreira, 2011b,a; Pratas and Pinho, 2014; Pinho *et al.*, 2014).

References

Bennett, C. H., Gács, P., Vitányi, M. L. P. M. B., and Zurek, W. H. (1998). Information distance. *IEEE Trans. on Information Theory*, **44**(4), 1407–1423.

Cebrián, M., Alfonseca, M., and Ortega, A. (2005). Common pitfalls using the normalized compression distance: what to watch out for in a compressor. *Communications in Information and Systems*, **5**(4), 367–384.

Chaitin, G. J. (1966). On the length of programs for computing finite binary sequences. *Journal of the ACM*, **13**, 547–569.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, **58**(2), 345–363.

Cilibrasi, R. and Vitányi, P. M. B. (2005). Clustering by compression. *IEEE Trans. on Information Theory*, **51**(4), 1523–1545.

Cutland, N. J. (1980). *Computability: An introduction to recursive function theory*. Cambridge University Press.

Dix, T. I., Powell, D. R., Allison, L., Bernal, J., Jaeger, S., and Stern, L. (2007). Comparative analysis of long DNA sequences by per element information content using different contexts. *BMC Bioinformatics*, **8**(Suppl. 2), S10.

Garcia, S. P., Rodrigues, J. M. O. S., Santos, S., Pratas, D., Afreixo, V., Bastos, C. A. C., Ferreira, P. J. S. G., and Pinho, A. J. (2013). A genomic distance for assembly comparison based on compressed maximal exact matches. *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, **10**(3), 793–798.

Golomb, S. (1966). Run-length encodings. *IEEE Trans. on Information Theory*, **12**(3), 399–401.

Gordon, G. (2003). Multi-dimensional linguistic complexity. *Journal of Biomolecular Structure & Dynamics*, **20**(6), 747–750.

Hamming, R. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, **29**(2), 147–160.

Hartley, R. V. L. (1928). Transmission of information. *Bell System Technical Journal*, **7**(3), 535–563.

Huffman, D. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, **40**(9), 1098–1101.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, **1**(1), 1–7.

Lempel, A. and Ziv, J. (1976). On the complexity of finite sequences. *IEEE Trans. on Information Theory*, **22**(1), 75–81.

Li, M., Chen, X., Li, X., Ma, B., and Vitányi, P. M. B. (2004). The similarity metric. *IEEE Trans. on Information Theory*, **50**(12), 3250–3264.

Minsky, M. L. (1967). *Computation: Finite and infinite machines*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Nikvand, N. and Wang, Z. (2010). Generic image similarity based on Kolmogorov complexity. In *Proc. of the IEEE Int. Conf. on Image Processing, ICIP-2010*, pages 309–312, Hong Kong.

Nikvand, N. and Wang, Z. (2013). Image distortion analysis based on normalized perceptual information distance. *Signal, Image and Video Processing*, **7**, 403–410.

Nyquist, H. (1924). Certain factors affecting telegraph speed. *Trans. of the American Institute of Electrical Engineers*, **XLIII**, 412–422.

Pinho, A. J. and Ferreira, P. J. S. G. (2011a). Finding unknown repeated patterns in images. In *Proc. of the 19th European Signal Processing Conf., EUSIPCO-2011*, Barcelona, Spain.

Pinho, A. J. and Ferreira, P. J. S. G. (2011b). Image similarity using the normalized compression distance based on finite context models. In *Proc. of the IEEE Int. Conf. on Image Processing, ICIP-2011*, Brussels, Belgium.

Pinho, A. J., Garcia, S. P., Pratas, D., and Ferreira, P. J. S. G. (2013). DNA sequences at a glance. *PLoS ONE*, **8**(11), e79922.

Pinho, A. J., Pratas, D., and Ferreira, P. J. S. G. (2014). A new compressor for measuring distances among images. In *Image Analysis and Recognition: Proc. of ICIAR-2014*, volume 8814 of *LNCS*, pages 30–37, Vilamoura, Portugal. Springer.

Pinho, A. J., Pratas, D., and Ferreira, P. J. S. G. (2016). Authorship attribution using relative compression. In *Proc. of the Data Compression Conf., DCC-2016*, Snowbird, Utah.

Pratas, D. and Pinho, A. J. (2014). A conditional compression distance that unveils insights of the genomic evolution. In *Proc. of the Data Compression Conf., DCC-2014*, Snowbird, Utah.

Pratas, D., Silva, R. M., Pinho, A. J., and Ferreira, P. J. S. G. (2015). An alignment-free method to find and visualise rearrangements between pairs of DNA sequences. *Scientific Reports*, **5**, 10203.

Radó, T. (1962). On non-computable functions. *Bell System Technical Journal*, **41**(3), 877–884.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, **14**, 465–471.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, **27**, 379–423,623–656.

Shannon, C. E. (1951). Prediction and entropy of printed English. *Bell Labs Technical Journal*, **30**(1), 50–64.

Sipser, M. (2012). *Introduction to the theory of computation*. Cengage Learning, 3rd edition.

Solomonoff, R. J. (1964a). A formal theory of inductive inference. Part I. *Information and Control*, **7**(1), 1–22.

Solomonoff, R. J. (1964b). A formal theory of inductive inference. Part II. *Information and Control*, **7**(2), 224–254.

Storer, J. A. and Szymanski, T. G. (1982). Data compression via textual substitution. *Journal of the ACM*, **29**(4), 928–951.

Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, **42**(2), 230–265.

Wallace, C. S. and Boulton, D. M. (1968). An information measure for classification. *The Computer Journal*, **11**(2), 185–194.

Welch, T. A. (1984). A technique for high-performance data compression. *IEEE Computer*, **17**(6), 8–19.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, **23**, 337–343.

Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, **24**(5), 530–536.