



Information Theory and Data Compression

Monday, March 6, 2023

Information Theory and Data Compression

- Prof: Marinella Sciortino
- ECTS/CFU: 6
- Lectures:
 - Monday: 10-13:00, Laboratorio 1 DMI
 - Friday: 8:30-11:30, Laboratorio 1 DMI
- Final verification: Oral examination + group project assigned during the course
- Reference Textbooks: (For other consultation texts, please refer to the [scheda di trasparenza](#))
 - Cover T.M. and Thomas J.A., Elements of Information Theory, Wiley- Interscience, 2006 (per gli argomenti di teoria dell'Informazione/for topics on Information Theory)
 - Sayood K., Introduction to Data Compression, Morgan Kauffman, 2017 (Per argomenti su compressione dati lossless/for topics on lossless data compression)
 - Salomon D., Motta G. Handbook of Data Compression, Fifth Edition, Springer 2010 (Per argomenti su compressione dati lossless/for topics on lossless data compression)

Main ingredients of the course...

- **Information theory** has as its main object of study the measuring and transmission of **information** emitted by a **source**.
- **Data compression** tries to reduce the number of **bits** used to store or transmit **information**.

This is possible because in the real world, data is very redundant, so data compression aims to **identify and extract redundancy** in the information emitted by certain **sources**.

In the following slides we will first try to give a more formal definition of the terms used above.

What is Information?

- Information - today more than ever - decisively conditions all our human activities.
- Moreover, information plays a fundamental role in the transmission of life itself: in fact, DNA itself contains information.
- It certainly lies at the heart of Computer Science. Although first proposed in 1956, the term "computer science" appears in a 1959 article in Communications of the ACM. In Europe, terms derived from the contracted translation of the expression "automatic information" are used to define the computer science (see INFORMATICA and INFORMATIQUE. Computer Science can be defined as automatic processing of information.

What is Information

- Information is an abstract concept that refers to that which has the power to inform.
- The english term INFORMATION comes from Middle French «Information» and its Latin Etymon: «informatio(-nis)» (from the verb «informare» with the meaning of giving form to something that has no form
- Intuitively, an information is an exchange of knowledge that allows one to overcome an uncertainty, to substitute the known for the unknown
- In Computer Science, what does INFORMATION mean?

Information and information content

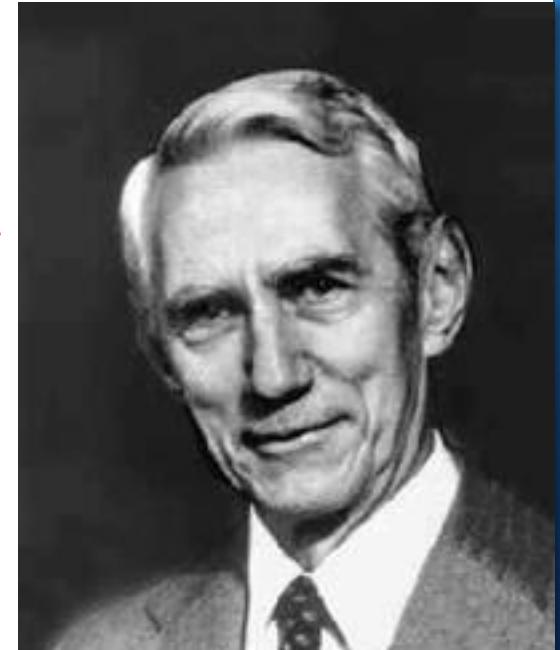
- In computer science, the exchange of information is an exchange of data between a sender and a receiver (who are not necessarily physical persons).
It becomes important to measure the information content of an information.
- Example 1: EXTRACTION OF A NUMBERED BALL. Suppose we have an urn containing 100 numbered balls, for each of which the probability of being taken out is the same. The sender extracts a ball and communicates the number stamped on it to the receiver.
- Example 2: TOSSING A COIN. Suppose the sender has a perfectly balanced coin and, after tossing it, communicates the message 'TESTA' or 'CROCE'.
- **Question:** What is the information with the highest information content? Why?

Some historical information...

- From a historical point of view, the person who first tried to formally define the concept of information was **R.V.L. Hartley** (1888-1970), the scientist best known for his **oscillating circuit**.
- In 1928, in a historical article in the Bell System Technical Journal, Hartley first developed the concept of information based on physical, rather than psychological, considerations for the study of electronic communications.
 - Hartley, R.V.L., "[Transmission of Information](#)", *Bell System Technical Journal*, July 1928, pp.535–563.
- In particular, in the first part of the aforementioned paper, entitled *The Measurement of Information*, Hartley never formally defines this concept, but rather attempts to characterise information in terms of some of its properties. He asserts that information exists in the process of transmission of symbols, in which "each symbol takes on a certain meaning for the communicating parties".
- When someone receives information, each symbol received allows the receiver to 'eliminate alternatives', excluding other possible symbols and their associated meaning. Hartley understands information as the exclusion of events that cannot occur: yes/no, black/white, testa/croce, true/false,
- His ideas awarded him a gold medal from the IEEE (Institute of Electrical and Electronics Engineers, Inc.) in 1946.

Shannon Theory

- Claude E. Shannon (1916-2001), who founded Information Theory, was the first to introduce the distinction between form and meaning in the communication process.
- It is considered that Information Theory originated with the publication of his paper **A Mathematical Theory of Communication**, which appeared in the Bell System Technical Journal in 1948..
- Formalising Hartley's idea, Shannon stated that information is: 'that which reduces uncertainty', anything that can reduce our degree of uncertainty about an event that may occur.
- To this end, he first introduced the term **bit** recognizing its central role in the process of communication. Shannon introduced the bit to represent two possible situations, namely 'element that is not in a set A' and 'element that is in a set A'.
- In more detail, a **BIT** (Binary digit) is the amount of information associated with the communication of an event, between two equiprobables.



Going back to the examples

- Example 2: TOSSING A COIN. Suppose the sender has a perfectly balanced coin and, after tossing it, communicates the message 'TESTA' or 'CROCE'.
- The information content associated with the event described in Example 2 is the minimum possible, since there are only two possible and equiprobable choices. It is precisely 1 bit.
- What is the information content in the case of Example 1? How is it quantified?
- What other issues are involved in the transmission of information?

Elements of a communication process

- According to Shannon, any communicative process essentially consists of five parts:
 - a **sender**, or source of information, which produces a message to be communicated to another entity;
 - an **encoder**, which is used to encode the message so that it can travel on a communication channel;
 - a **communication channel**;
 - a **decoder**, which receives what travels over the channel and decodes it to reconstruct the message;
 - a **receiver**, to whom the message reaches.

Elements of a communication process

- Shannon introduces two other important hypotheses:
 - the transmission of symbols along the channel is a **discrete phenomenon**, i.e., sending each symbol requires a certain amount of time, finite and non-zero;
 - there is a **source of noise**, which acts on the channel by modifying the syntactic content of the information, its form, in an unpredictable manner. In other words, the noise source can transform a symbol travelling along the channel into another symbol, in a totally unpredictable manner.
- Shannon actually developed two theories:
 - with noise - will not be covered in this course
 - without noise (noiseless)

Issues to be considered

- The communication process just described highlights several problems:
 - how to measure the amount of information which travels along the channel. Recall that Shannon defined the bit as the unit of measure of information;
 - how to guarantee secure transmissions. Making transmissions secure means making the sending of information safe from prying eyes by transforming it. Since the 1970s, the theory of how to guarantee secure transmissions has evolved on its own, becoming Cryptography.
 - how to guarantee trusted transmissions. Shannon showed that trusted transmissions cannot be guaranteed, except by introducing redundancy into the message. For example, the greater the number of times the same message is replicated (i.e. sent again), the greater the probability that the message will arrive correct at least once.
 - how much information can be compressed and how to transmit it efficiently. These are some of the main goals of Data Compression.

Summing up...

In Shannon's theory, it is not possible to define the amount of information associated with a message that has already been received, but rather the **amount of potential information** associated with a message that can be sent.

Furthermore, an acceptable measure of information:

- should be linked to the probability that the generic event has of occurring (the higher the information content associated with an event, the lower the probability of the event, i.e. the more unexpected it is) and in particular it should increase as the probability decreases and it should vary with continuity as the latter varies
- in the case of a joint event, it should be equal to the sum of the information associated with the individual events, if these are statistically independent.

How did Shannon formulate this?

Discrete source

- A **source** is characterised by an alphabet A (the elements of this alphabet can be of any nature, an event occurring, a man speaking, a symbol of a text,...).
- The source has a discretized time and at each time unit it produces one element of the alphabet. Each element of the alphabet is produced with a certain probability.
- The probability is a map $\pi: A \rightarrow [0,1]$ such that $\sum_{a \in A} \pi(a) = 1$
- A source S with alphabet $A = \{a_1, a_2, \dots, a_m\}$ and probability $\{p_1, p_2, \dots, p_m\}$, can be represented in the following way:

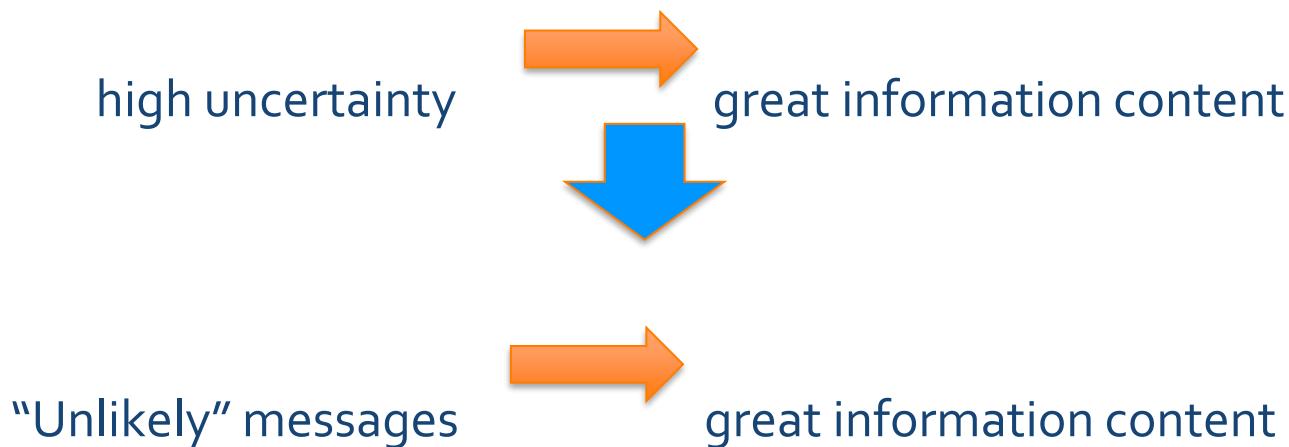
$$\begin{pmatrix} a_1 & a_2 & \dots & a_m \\ p_1 & p_2 & \dots & p_m \end{pmatrix}$$

Source with memory and memoryless

- A **memoryless source** is a system of statistically independent events that are transmitted as messages with assigned probability. The probability of emission remains constant over time whatever symbols have been emitted in the past (the source is stationary).
- A source in which the emission of a symbol depends on a finite number t of previously emitted symbols is called a **source with memory** or **Markov source of order t** . It is specified by assigning the source alphabet and the set of conditional probabilities.
- In this course we will deal with memoryless sources.

Measure of information

- In Shannon Theory, the measure of information is related to the uncertainty associated with the emission of each symbol a_i (i.e., it is related to the uncertainty as to whether the value produced by the source is indeed a_i)



Self-information

- Given a symbol a_i of the alphabet of the source, the **self-information** of a_i is defined as:

$$I(a_i) = \log_b \left(\frac{1}{p_i} \right)$$

- Such a quantity is measured in units of information. The base of the logarithm is not necessarily specified. Its choice determines the unit of measure assigned to the information content: if the base is "e" the unit of measurement will be the NAT, if the base is "10" we will have the HARTLEY, while if the base is "2" we have the BIT (acronym for Binary Digit). The use of BIT is based on the fact that the correct identification of one of two equally probable symbols involves an amount of information equal to :

$$I(a_1) = I(a_2) = \log_2(2) = 1 \text{ bit (si assume questa come unità di misura).}$$

- It is evident from the definition that if the probability of a symbol is "1", then the amount of information is 0 (a result, however, expected), whereas if the probability of the symbol is zero, the amount of information is infinite.
- 1 HARTLEY=3.32 bits 1 NAT=1.44 bits

Properties of $I(a_i) = \log_b \left(\frac{1}{p_i} \right)$

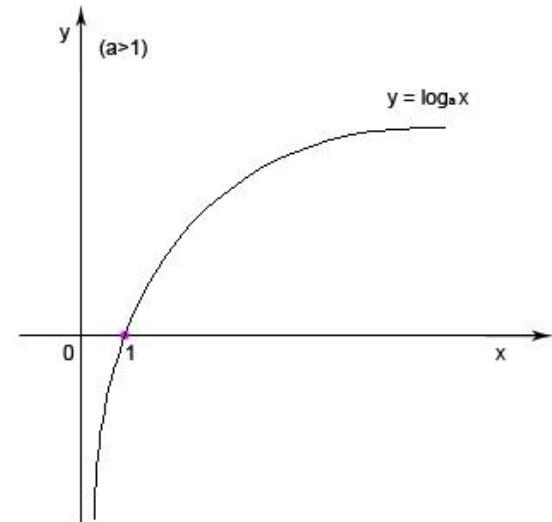
The following properties of self-information are derived from the logarithm function and its properties:

- $I(a_i) \geq 0$ for $0 \leq p_i \leq 1$
- $I(a_i) \rightarrow 0$ for $p_i \rightarrow 1$,
high probability \rightarrow small information
- $I(a_i) > I(a_j)$ per $p_i < p_j$
probability of a_i is less than probability of a_j , then a_i has more information than a_j .
- Given two independent symbols a_i and a_j , we have that

$$\pi(a_i, a_j) = p_i \times p_j,$$

then

$$\begin{aligned} I(a_i, a_j) &= \log_b \frac{1}{\pi(a_i, a_j)} = \\ &= -\log_b \pi(a_i, a_j) = -\log_b p_i - \log_b p_j = I(a_i) + I(a_j) \end{aligned}$$



Self-information

We will consider the binary case($b = 2$):

$$I_i = \log_2 \frac{1}{p_i}$$

For a source with two symbols equally probable:

$$p_1 = p_2 = \frac{1}{2}$$

$$I_1 = I_2 = \log_2 2 = 1 \text{ [bit]}$$

This confirms that 1 bit is the information needed to distinguish between two equiprobable messages.

Entropy of a source

- Given a source without memory S , the ENTROPY of source S is defined as the quantity representing the average information content for each symbol of the entire source, defined as

$$H(S) = \sum_{i=1}^m p_i I(a_i) = \sum_{i=1}^m p_i \log_b \left(\frac{1}{p_i} \right)$$

- This is a weighted average of the information associated with each symbol. The information is weighted with the corresponding probability of the symbol.
- It is measured in bit/symbol (if the base of the logarithm is 2)
- We will from now on assume that the bit is the unit of measurement of information, i.e. that $b=2$.

Axiomatic definition of entropy

- The notion of the entropy of a discrete source was introduced by Shannon in 1948 in an axiomatic way, i.e. a function was searched for to measure the average information produced by a source.
If such a function $H(S) = H_n(p_1, p_2, \dots, p_n)$ exists, then it is reasonable to assume that it has properties described below.
- 1. H_2 is a continuous function of p in the real interval $[0,1]$
- 2. $H_2\left(\frac{1}{2}, \frac{1}{2}\right) = 1$ (we choose the unit of measure)
- 3. (Permutation invariance) $H_n(\sigma(p_1, p_2, \dots, p_n)) = H_n(p_1, p_2, \dots, p_n)$, for each permutation of the source probabilities. That is, the uncertainty does not depend on how the probabilities are assigned to the symbols.
- 4. Let $A(n) = H_n\left(\frac{1}{n}, \dots, \frac{1}{n}\right)$. $A(n)$ must be a function of n that is monotonically increasing. That is, if the events are equiprobable, the higher the number of events the greater the uncertainty is.

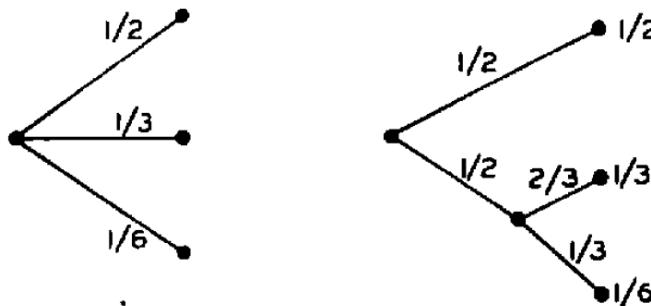
Axiomatic definition of entropy

5. (Grouping axiom) If a choice is split into two successive choices, the original value of H must be the weighted sum of the individual values of H. Then

$$H_n(p_1, p_2, \dots, p_n) = H_{n-1}(p_1 + p_2, p_3, \dots, p_n) + (p_1 + p_2)H_2\left(\frac{p_1}{p_1+p_2}, \frac{p_2}{p_1+p_2}\right)$$

that is, if an experiment is composed, this is expressed in the function.

EXAMPLE $H_3\left(\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\right) = H_2\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{1}{2}H_2\left(\frac{1}{3}, \frac{2}{3}\right)$



Remark

Iterating the reasoning, it is shown that axiom grouping can be extended to the grouping of k symbols, i.e.

$$H_n(p_1, p_2, \dots, p_n) = H_{n-k+1}(p_1 + \dots + p_k, p_{k+1}, \dots, p_n) + (p_1 + \dots + p_k)H_k\left(\frac{p_1}{p_1+\dots+p_k}, \dots, \frac{p_k}{p_1+\dots+p_k}\right)$$

Axiomatic definition of entropy

Theorem: The only function H that satisfies the five previous axioms is

$$H(S) = C \sum_{i=1}^n p_i \log_b \left(\frac{1}{p_i} \right)$$

where C is a positive constant and the base of the logarithm is a value greater than 1.

The constant C depends on the unit of measurement chosen (and the base of the logarithm) and is generally assumed to be equal to 1.

PROOF: It is easily verified that the function satisfies the axioms. Conversely, it is proved that any function that satisfies the axioms must have the form described in the theorem. The demonstration proceeds in steps:

1. From axiom 5 we derive that $A(nm) = A(n) + A(m)$.
2. $A(n^k) = kA(n)$ for all positive integers n and k . The proof is by induction on k using the previous property.
3. $A(n) = C \log_b(n)$ for every value of n , where C is a positive constant depending on the base of the logarithm and the unit of measure we have chosen.
4. $H_2(p, 1-p) = -C[p \log p + (1-p) \log(1-p)]$ for every real number p in the interval $[0,1]$. It is first proved for all rational numbers.
5. $H_n(p_1, p_2, \dots, p_n) = C \sum_{i=1}^n p_i \log \left(\frac{1}{p_i} \right)$, for every n . It is proved by induction on n .

EXAMPLES

Let us assume that the unit of measure is the bit and that the logarithm is expressed in base 2 (unless expressly stated otherwise)

Example 1

Let us consider the source of three symbols with the probabilities described below:

$$S = \begin{pmatrix} a_1 & a_2 & a_3 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \end{pmatrix}$$

The entropy of the source is:

$$H(S) = \frac{1}{2} \log 2 + \frac{1}{4} \log 4 + \frac{1}{4} \log 4 = \frac{3}{2} \text{ bits/symbol}$$

- This can be interpreted either as the average amount of information per symbol provided by the source, or as the average amount of uncertainty the observer has before discovering the output of the source

Example 2

Let us consider the source of three symbols with the probabilities described below:

$$S = \begin{pmatrix} a_1 & a_2 & a_3 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{pmatrix}$$

The entropy of the source is:

$$H(S) = \frac{1}{3} \log 3 + \frac{1}{3} \log 3 + \frac{1}{3} \log 3 = \log 3 \text{ bits/symbol}$$



Kraft-McMillan Inequality

Friday March 10, 2023

Kraft-McMillan inequality

THEOREM: Let us consider a code with alphabet source $S = \{s_1, s_2, \dots, s_m\}$ and alphabet code $X = \{x_1, x_2, \dots, x_d\}$. Let $\{c_1, c_2, \dots, c_m\}$ be the codewords with lengths l_1, l_2, \dots, l_m , respectively.

Necessary and sufficient condition for the existence of an instantaneous code with codeword length l_1, l_2, \dots, l_m is that

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

where d is the number of distinct symbols in the alphabet code.

EXAMPLES

- Given the following binary codes, do they have acceptable lengths for an instantaneous code?

Source	Code A	Code B	Code C	Code D	Code E
s1	00	0	0	0	0
s2	01	100	10	100	10
s3	10	110	110	110	110
s4	11	111	111	11	11

- We recall that the Kraft-McMillan inequality tells us whether the lengths are acceptable, but if this is the case it does not imply that such a code is instantaneous.
- In particular, which codes are instantaneous?

Code A

Code A
00
01
10
11

- We know that the code is instantaneous (or prefix) because the code words are all distinct and have the same length, therefore the lengths are acceptable
- In fact, $\sum_{i=1}^4 2^{-2} = 1$.
- This information alone would not allow us to conclude that code A is prefix, but rather that there exists a prefix code having code words with those lengths.

Code B

Code B
0
100
110
111

- Let us consider the lengths of the code B

- We compute

$$\sum_{i=1}^4 2^{-l_i} = 2^{-1} + 2^{-3} + 2^{-3} + 2^{-3} = \frac{7}{8} \leq 1$$

- The lengths are acceptable.
- Furthermore, by examining the codewords, it is verified that B is indeed a prefix code.

Code C

Code C
0
10
110
111

- C is a prefix code.
- What result do we expect for the lengths of the C code?

$$\sum_{i=1}^4 2^{-l_i} = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = \frac{8}{8} = 1$$

- Kraft-McMillan inequality is satisfied.

Code D

Code D
0
100
110
11

- We compute

$$\sum_{i=1}^4 2^{-l_i} = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = 1$$

- But we know that D is not a prefix code.
- By the Kraft-McMillan inequality we know that it is possible to find a prefix code whose codewords have lengths 1,2,3. In fact, code C is an example of this.

Code E

Code E
0
10
110
11

- We evaluate

$$\sum_{i=1}^4 2^{-l_i} = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-2} = 1 + \frac{1}{8}$$

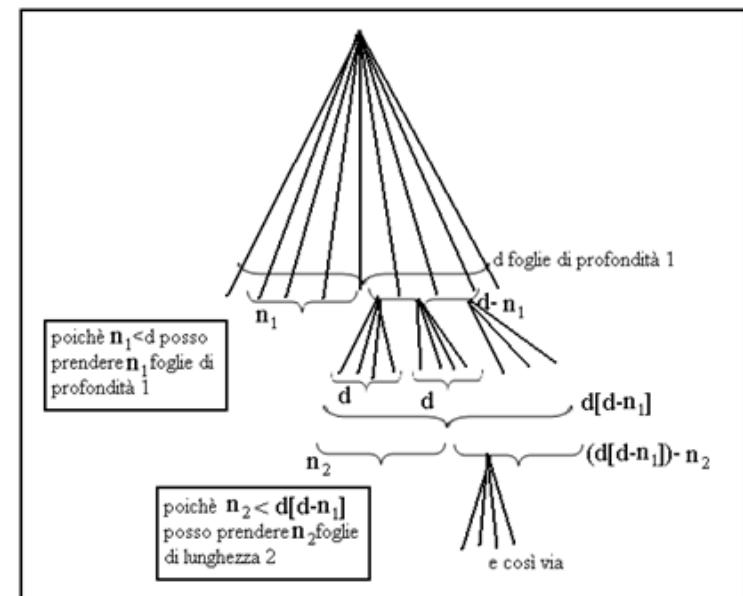
- The lengths do not satisfy the inequality, therefore the code is not instantaneous

Proof (sufficient condition)

- Let us suppose that the lengths l_1, l_2, \dots, l_m satisfy the inequality $\sum_{i=1}^m d^{-l_i} \leq 1$. We want to prove that there exists a prefix code with the above lengths.
- Let us denote by n_1 the number of length values equal to 1, by n_2 the number of length values equal to 2, ..., by n_L the number of length values equal to the maximum value L.
- We have that $\sum_{i=1}^L n_i = m$, since m are the codewords.
 - $\sum_{i=1}^m d^{-l_i} = \sum_{i=1}^L n_i d^{-i} \leq 1$, i.e.
 - $n_1 d^{-1} + n_2 d^{-2} + n_3 d^{-3} + \dots + n_L d^{-L} \leq 1$ we multiply by d
 - $n_1 + n_2 d^{-1} + n_3 d^{-2} + \dots + n_L d^{-L+1} \leq d$ In particular $n_1 \leq d$
 - $n_2 d^{-1} + n_3 d^{-2} + \dots + n_L d^{-L+1} \leq d - n_1$ we multiply by d
 - $n_2 + n_3 d^{-1} + \dots + n_L d^{-L+2} \leq d(d - n_1)$ In particular $n_2 \leq d(d - n_1)$
 - ...

Proof (sufficient condition)

- We obtain a set of inequalities that ensure it is possible to construct a prefix code using a d-ary tree, which has n_1 leaves at depth 1, n_2 leaves at depth 2, ..., n_L leaves at maximum depth L.
- The codewords correspond to the leaves, and they are obtained by concatenating the symbols of the paths needed to reach them. To each outgoing edge from a node, the symbols of the code alphabet are assigned in order.



Exercise

- We want to encode a source with 9 symbols using a prefix code with code alphabet $\{0, 1, 2\}$ and code words of lengths $1, 2, 2, 2, 2, 2, 3, 3, 3$.
- First, we apply the Kraft-McMillan inequality test

$$\sum_{i=1}^9 3^{-l_i} = 1$$

- To find the prefix code, we use the technique suggested in the proof.

Solution

Here is a possible binary prefix code.

s	code
s0	0
s1	10
s2	11
s3	12
s4	20
s5	21
s6	220
s7	221
s8	222

Proof (necessary condition)

In fact, a stronger result can be proved (demonstrated by McMillan in 1956).

If C is a UD code with source alphabet $S = \{s_1, s_2, \dots, s_m\}$ and code alphabet $X = \{x_1, x_2, \dots, x_d\}$. Let $\{c_1, c_2, \dots, c_m\}$ be the codewords with length l_1, l_2, \dots, l_m , respectively.

Then, we have that

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

where d is the number of symbols of the alphabet code.

Proof (necessary condition)

- Let L be the maximum length of the code words in C (which we assume to be uniquely decodable).
- We consider the quantity:

$$\left(\sum_{i=1}^m d^{-l_i} \right)^n = (d^{-l_1} + d^{-l_2} + \cdots + d^{-l_m})^n$$

- If we expand it we obtain m^n terms, i.e.

$$d^{-l_{i_1}-l_{i_2}-\cdots-l_{i_n}} = d^{-k} \text{ where } k = l_{i_1} + l_{i_2} + \cdots + l_{i_n}$$

k can assume values between n and nL .

- Let N_k be the number of terms of the form d^{-k} . Then $\left(\sum_{i=1}^m d^{-l_i} \right)^n = (\sum_{k=n}^{nL} N_k d^{-k})$

Proof (necessary condition)

- It is possible to verify that N_k is also the number of strings consisting of n codewords such that each string has length k . If the code is UD, then $N_k \leq d^k$ (the number of distinct words over d letters of length k).
- Then $\left(\sum_{i=1}^m d^{-l_i}\right)^n \leq \left(\sum_{k=n}^{nL} d^k d^{-k}\right) = nL - n + 1 \leq nL$.
This holds for every n .
- We note that for $x > 1$, $x^n > nL$, if n is large enough.
- We can deduce that $\sum_{i=1}^m d^{-l_i} \leq 1$

Exercise

- We want to encode a source with 10 symbols using a prefix code with code alphabet {0, 1, 2} and code words of lengths 1, 2, 2, 2, 2, 2, 3, 3, 3, 3.
- First, we apply the Kraft-McMillan inequality test:

$$\sum_{i=1}^{10} 3^{-l_i} = \frac{28}{27} > 1$$

- It is impossible to find a prefix (or uniquely decodable) code that meets the required criteria.

Remarks on Kraft-McMillan inequality

- What happens if we consider lengths such that the Kraft-McMillan sum is less than 1?
- For example, consider the lengths {1,3,3}. How to construct a binary C prefix code having code words with such lengths?
- We use the tree technique in which not all leaves will be considered.
- There will certainly be a sequence of symbols in the code alphabet that cannot be decoded by code words.

Solution

- Here is a possible prefix code
- It can be shown that all the leaves of the tree are taken into account if and only if the Kraft-McMillan sum of the code lengths is equal to 1 (Exercise).

s	code
s0	0
s1	100
s2	110
s3	111



Sardinas-Patterson Algorithm

Friday, March 10, 2023

Entropy of a discrete source

Let S be a discrete source with alphabet $A = \{a_1, a_2, \dots, a_m\}$ and probability $\{p_1, p_2, \dots, p_m\}$, i.e.:

$$S = \begin{pmatrix} a_1 & a_2 & \dots & a_m \\ p_1 & p_2 & \dots & p_m \end{pmatrix}$$

The entropy of S is defined as

$$H(S) = \sum_{i=1}^m p_i I(a_i) = \sum_{i=1}^m p_i \log_b \left(\frac{1}{p_i} \right)$$

We assume $b=2$

Upper bound for the entropy

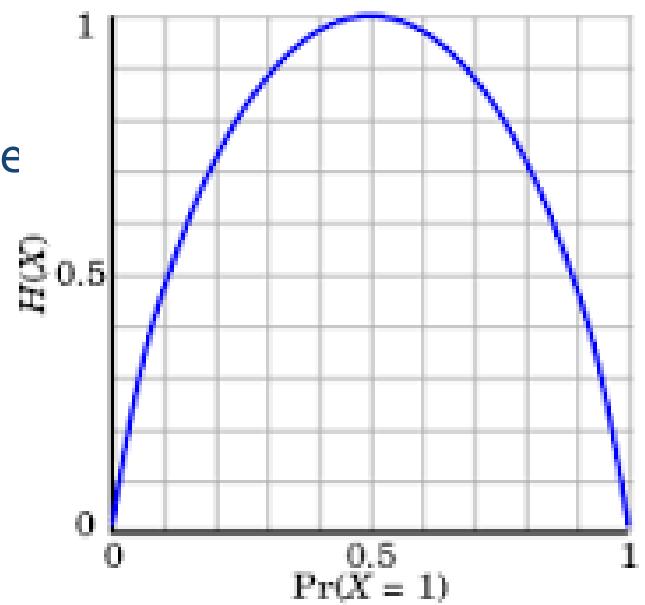
- It is possible to prove that, given a source with m symbols,

$$H(S) \leq \log m$$

- Equality occurs when the source symbols are equiprobable.
- For instance, in case of source with 2 symbols:

$$H(S) = p \log \frac{1}{p} + (1 - p) \log \frac{1}{1-p}$$

we have the maximum value when $p = 1/2$.



Encoding a source

- The source alphabet is generally not suitable for transmission. A **source encoder** is needed to transform the source symbols into symbols of another alphabet, and then transmit the encoded message through the channel.
- On the receiving end, we must of course perform the reverse process, returning the source symbols to the receiver with a **decoder**

Code

- Let S be a source $\{s_1, s_2, \dots, s_m\}$. We define **code** a function that maps any sequence of elements of S to a sequence of symbols of another alphabet $X = \{x_1, x_2, \dots, x_r\}$.
 X is called **input alphabet code**.
- We will study some particular subclasses of codes with specific properties.
- A **block code** is a code that associates each symbol in the source alphabet with a specific sequence of symbols in the code alphabet. These latter sequences of symbols in the code alphabet are called **codewords**.

Non-singular block codes

We are interested in **block codes in which the codewords are all distinct**. If there were identical code words, the decoding could not be unique. For example, if we consider the following block code

Source	Code
s1	0
s2	11
s3	00
s4	11

- We can easily note that 11 represents two distinct messages.
- A block code is called **non-singular** if all codewords are distinct

A non-singular code might not be enough

Let us consider the following non-singular code:

Source	Code
s1	0
s2	11
S ₃	00
S ₄	01

- ❑ For instance, which message is encoded by the sequence 0011?
- ❑ Two possible messages are s_3s_2 e $s_1s_1s_2$
- ❑ Although the code is non-singular, it becomes so when longer strings are considered.

Uniquely decodable code

- A code is said to be **uniquely decodable** (UD) if each sequence of codewords corresponds to at most one sequence of source symbols.
- Questions:
 - How to construct UD codes?
 - How to recognize if a given code is UD?

Example 4

- ❑ Suppose we need a system that communicates the weather situation in the city of Rome at various instants of time. The possible states to be communicated are: Sun, Cloudy, Rain, Fog. Which code should be used?

- ❑ For instance:

Source	Code A	Code B	Code C	Code D
Sun	00	0	0	0
Cloudy	01	10	01	01
Rain	10	110	011	10
Fog	11	1110	0111	1

- ❑ Which code is UD? Why?
- ❑ Using such codes, which messages does the encoding correspond to 011100? And 101100?

Sardinas-Patterson Algorithm



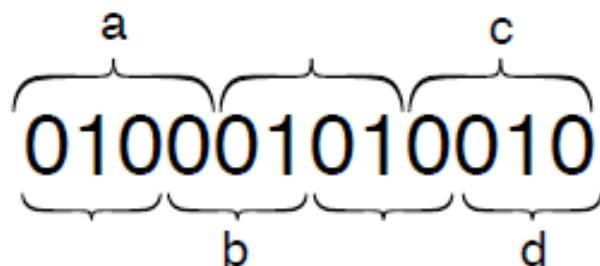
Sardinas-Patterson Algorithm (1953)

- ❑ It is not always straightforward to determine whether a code is uniquely decipherable, as one would have to consider arbitrarily large strings. A criterion is therefore needed to make this operation easier.
- ❑ The Sardinas-Patterson theorem provides a useful algorithm.

Sardinas-Patterson algorithm

It is based on the following consideration: suppose we have a string consisting of a sequence of codewords.

If we try to construct two possible factorizations, each of the words in one factorization is either contained in a word of the other factorization or begins with a prefix that is suffixed to one of the words of the other factorization. The code is non-UD if it happens that a suffix coincides with a code word (see C and D).



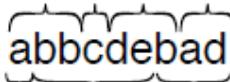
Sardinas-Patterson Algorithm

- It is based on the following theorem (which we do not prove):
- THEOREM: Given a code C on an alphabet A , consider the following sets : S_0, S_1, S_2, \dots such that:
 - $S_0 = C$
 - $S_i = \{w \in A^* \mid \exists a \in S_0, \exists b \in S_{i-1} \text{ such that } a = bw \text{ or } b = aw\}$
(i.e. the set of suffixes),
 - then a necessary and sufficient condition for C to be UD is that
 $\forall n > 0, S_0 \cap S_n = \emptyset$, i.e. none of the generated suffixes must correspond with a codeword.
- The theorem implicitly defines an algorithm. Since the set of generatable suffixes is finite, the process will sooner or later begin to generate already generated sets. The process therefore has 3 halting conditions, namely
 - $\exists i > 0, S_0 \cap S_i \neq \emptyset$ then the code C is not UD
 - $\exists i > 0, S_i = \emptyset$ then the code C is UD
 - $\exists i, j > 0, S_i = S_j$ then the code C is UD
- The algorithm works in time $O(rL)$ where r is the number of codewords and L the sum of the lengths of those words.

Example

Use the Sardinas-Patterson algorithm to verify that the code $\{a, c, ad, abb, bad, deb, bbdce\}$ is UD.

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


abbcdedebad

Example of UD code

Let us consider the code $S=\{abc, abcd, e, dba, bace, ceac, ceab, eabd\}$. Is it UD?

Another example of non-UD code

Let us consider the code $S=\{010, 0001, 0110, 1100, 00011, 00110, 11110, 101011\}$

S_0	S_1	S_2	S_3	S_4	S_5	S_6
010	1	100	11	00	01	0
0001		1110		110	011	10
0110		01011			110	001
1100					0	110
00011						0011
00110						0110
11110						
101011						

S	A	B	C	D
Sun	00	0	0	0
Cloudy	01	10	01	01
Rain	10	110	011	10
Fog	11	1110	0111	1

Example 4 (contd.)

- Using the Sardinas-Patterson algorithm, it can be proven that Code A , Code B and Code C are all uniquely decipherable. Code D is not uniquely decodable (Exercise).
- Which do you prefer among A, B and C?
- Which encoding for the message Sun, Fog , Fog, Rain ?
- If we have different weather stations, for instance

Roma:	Source	Probability
	Sun	1/4
	Cloudy	1/4
	Rain	1/4
	Fog	1/4

Udine:	Source	Probability
	Sun	1/4
	Cloudy	1/8
	Rain	1/8
	Fog	1/2

which code could be the best one? What could be the selection criteria?

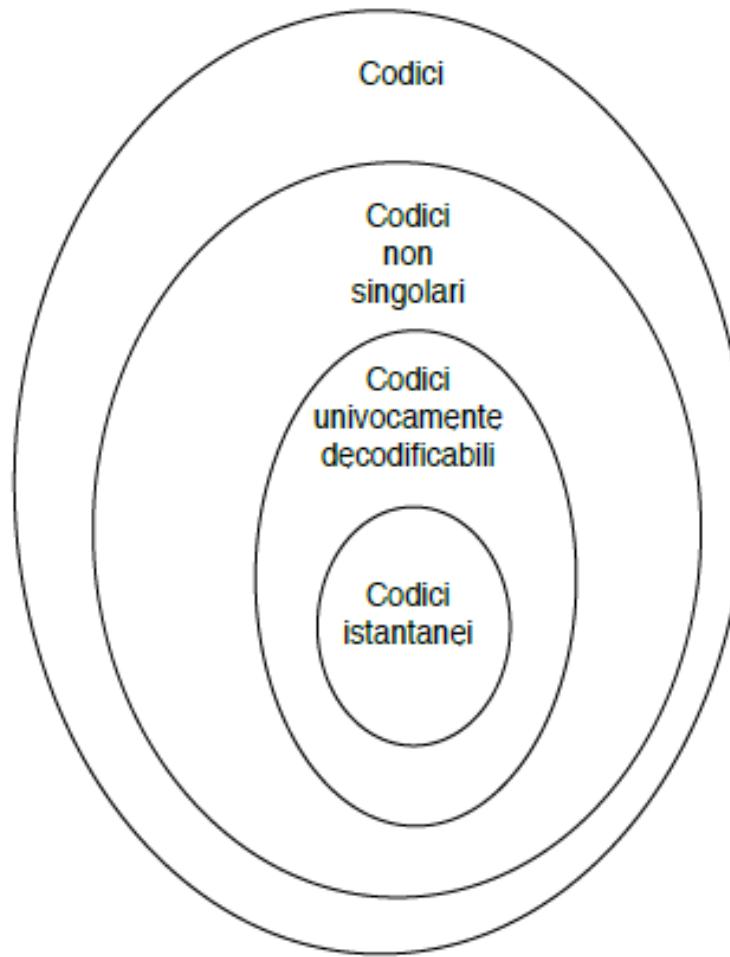
Prefix code



Instantaneous codes and prefix codes

- We can verify that if we receive the message 01110 by using the code $C=\{0,01,011,0111\}$, in order to decode the codewords we need to see the next symbol.
- A code is said to be **instantaneous** if it is possible to decode each word in a sequence of source symbols without waiting for the next code symbol.
- A code is said to be a **prefix code** if no code word is a prefix to another.
- **THEOREM:** A code is instantaneous if and only if it is prefix (proof by exercise).
- **THEOREM:** Prefix codes are uniquely decodable (it follows from the Sardinas-Patterson Theorem)

CODES



Several types of UD-codes

- Let us consider a source with 3 symbols, encoded by using the code $\{0,01,11\}$. Is it UD?
- We can apply Sardinas-Patterson algorithm.
 - $S_0 = \{0,01,11\}, S_1 = \{1\}, S_2 = \{1\}, \dots$
- It is a UD code.
- It can be verified that it is a code with a deciphering delay. The deciphering delay, in this case, is unbounded, since only at the end of the message can be correctly decoded.
Es: 01111111, 01111111
- It is possible to prove that:
 - If $\exists i > 0, S_i = \emptyset$ then C is UD, with bounded deciphering delay (if $i=1$, we have a prefix code) [Levenshtein 1962]
 - $\exists i, j > 0, S_i = S_j$ the code is UD, with unbounded deciphering delay.

Exercise 1

- ❑ Let us consider the following binary code
 $\{a \rightarrow 1, b \rightarrow 011, c \rightarrow 01110, d \rightarrow 1110, e \rightarrow 10011\}$
- ❑ Is it UD?
- ❑ Example: Decoding the string 011101110011

Lab exercises

- Portfolio*: Write a program in a programming language of your choice that applies the Sardinas-Patterson algorithm and returns the type of code received as input.
- Apply the algorithm to test whether $C=\{012, 0123, 4, 310, 1024, 2402, 2401, 4013\}$ is UD.
- Apply the algorithm to verify which of the codes
 $C_1 = \{ 10, 010, 1, 1110 \}, \quad C_2 = \{ 0, 001, 101, 11 \},$
 $C_3 = \{ 0, 2, 03, 011, 104, 341, 11234 \},$
 $C_4 = \{01,10,001,100,000,111\}$ are UD.

* the set of exercises or programs that each working group will discuss during the final examination (i.e., the group project)



Compact codes and Shannon Theorem

Monday March 13, 2023

Remind the Kraft-McMillan inequality

THEOREM: Let us consider a code with alphabet source $S = \{s_1, s_2, \dots, s_m\}$ and alphabet code $X = \{x_1, x_2, \dots, x_d\}$. Let $\{c_1, c_2, \dots, c_m\}$ be the codewords with lengths l_1, l_2, \dots, l_m , respectively.

Necessary and sufficient condition for the existence of an instantaneous code with codeword length l_1, l_2, \dots, l_m is that

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

where d is the number of distinct symbols in the alphabet code.

Remarks

- The Kraft-McMillan inequality tells us whether the lengths of a set C of codewords are acceptable for being a UD (or a prefix) code, but if this is the case it does not imply that such a code is prefix (or UD).
- The proof of the theorem gives a strategy to construct a prefix code, given the set of lengths. A d -ary tree is used if d is the size of the alphabet code. The codewords correspond to the leaves, and they are obtained by concatenating the symbols of the paths needed to reach them. To each outgoing edge from a node, the symbols of the code alphabet are assigned in order.

Some more remarks

- What happens if we consider lengths such that the Kraft-McMillan sum is less than 1?
- In the d-ary tree not all leaves will be considered.
- This means that there will certainly be a sequence of symbols in the code alphabet that cannot be decoded by codewords.

Question: which prefix code to choose?

Given the two prefix binary codes, how can we decide which one is more convenient?

S	Code 1	Code 2
s0	1101	0
s1	110000	10
s2	00100	110
s3	11111	111

Average code length

- Let C be a code with alphabet source $S = \{s_1, s_2, \dots, s_m\}$ and alphabet code $X = \{x_1, x_2, \dots, x_d\}$. Let $\{c_1, c_2, \dots, c_m\}$ be the codewords with length l_1, l_2, \dots, l_m , respectively.

Let $\{p_1, p_2, \dots, p_m\}$ be the probabilities of the source symbols.

- We define the average code length for a source S , as:

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

- We are interested in finding uniquely decodable codes with the lowest possible average length, that is, among all UD codes for the same source and with the same code alphabet, the one with the smallest average length (**compact code**).

Shannon Theorem



Average code length: lower bound

- Theorem (Shannon): If C is a UD code for a memoryless source S with probabilities p_1, p_2, \dots, p_m , and alphabet code X with size d , then:

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

- In the case of codes over binary alphabets, the average code length must be at least equal to the entropy of the source.
- If we consider the definition of entropy in which the base of the log function is d , then we have that $L_S(C) \geq H_d(S)$
- We will study the proof of the theorem because it provides important and useful information for the construction of compact codes.

Proof

- If C is a UD code then $\sum_{i=1}^m d^{-l_i} \leq 1$.
- Let us consider m non-negative numbers q_i such that $\sum_{i=1}^m q_i = 1$.
- It is possible to prove that $\sum_{i=1}^m p_i \log \frac{1}{p_i} \leq \sum_{i=1}^m p_i \log \frac{1}{q_i}$ and the equality holds if and only if $q_i = p_i$ for each i .
- Let us consider, in particular, $q_i = \frac{d^{-l_i}}{\sum_{i=1}^m d^{-l_i}}$
- We can deduce that $H(S) \leq -\sum_{i=1}^m p_i \log q_i = L_S(C) \log d$. Such an inequality uses the fact that C is UD, i.e. $\sum_{i=1}^m d^{-l_i} \leq 1$.

NOTE: By combining the two remarks on the two inequalities in the proof, when UD codes for which $\sum_{i=1}^m d^{-l_i} = 1$ are considered, we have that the equality holds if and only if $p_i = d^{-l_i}$ for each i , i.e. when $l_i = \log_d \frac{1}{p_i}$. In this case the average code length reaches the entropy. This is possible when $\log_d \frac{1}{p_i}$ is an integer value for each i .

When $L_S(C) = H_d(S)$?

- Given a UD code with d-ary alphabet (with d symbols) the average code length reaches the entropy if and only if $\sum_{i=1}^m d^{-l_i} = 1$ e $l_i = \log_d \frac{1}{P_i}$.
- This is possible if $\log_d \frac{1}{P_i}$ is an integer value for each i i.e., $p_i = \left(\frac{1}{d}\right)^{\alpha_i}$, where α_i is an integer value.
- In this case a compact code is obtained. It is enough to choose $l_i = \alpha_i$.
- We note that if $l_i = \alpha_i$:

$$1 = \sum_{i=1}^m p_i = \sum_{i=1}^m \left(\frac{1}{d}\right)^{\alpha_i} = \sum_{i=1}^m d^{-\alpha_i} = \sum_{i=1}^m d^{-l_i}$$

A very special case

- Let us consider a source S such that for each i, $p_i = \left(\frac{1}{d}\right)^{\alpha_i}$, where α_i is a positive integer.
- As we showed before, if we choose $l_i = \alpha_i$ we obtain a compact code.

Exercise

Given the source

Source	Probability
s1	1/4
s2	1/4
s3	1/4
s4	1/4

Is it possible to find a compact code?

- Let us compute the entropy, $H(S)=2$.
- We will never find any compact code with an average code length less than 2
- Since each symbol has probability $\left(\frac{1}{2}\right)^2$, we can construct a compact code having codewords with length 2.

Exercise

Given the source

Source	Probability	Code
s1	1/4	00
s2	1/4	01
s3	1/4	10
s4	1/4	11

Is it possible to find a compact code?

- Let us compute the entropy, $H(S)=2$.
- We will never find any compact code with an average code length less than 2
- Since each symbol has probability $\left(\frac{1}{2}\right)^2$, we can construct a compact code having codewords with length 2.

Exercise

Given the source

Source	Probability	Code1	Code 2
s1	1/2	0	0
s2	1/4	10	10
s3	1/8	110	110
s4	1/8	111	1110

Which code is a compact code?

We can prove that the entropy of the source is $1+3/4$.

Then we can conclude that Code 1 is a compact code.

Code 2 is not a compact code since the average code length is $1+7/8 > 1+3/4$.

Exercise

Given the following source, find a compact code over the alphabet $\{0,1,2\}$.

S	Probability
s0	1/3
s1	1/3
s2	1/9
s3	1/9
s4	1/27
s5	1/27
s6	1/27

Solution

- Let us compute the entropy:
 - $H_3(S) = \frac{13}{9}$
- Let us consider the lengths $\{1, 1, 2, 2, 3, 3, 3\}$
- Let us compute the average code length:
 - $L_S(C) = \frac{13}{9}$
- We obtained a compact code.



S	Code
0	0
1	1
2	20
3	21
4	220
5	221
6	222

Question

- So, which is the procedure when $\log_d \frac{1}{P_i}$ is not an integer value for some i?
- For instance, let us consider the source:

S	Prob.
s0	1/4
s1	3/4

- How to construct a binary compact code?

Shannon encoding



Shannon encoding

Theorem: Given a memoryless source S , if for each $1 \leq i \leq m$ we choose l_i as the integer value such that

$$l_i = \left\lceil \log_d \frac{1}{p_i} \right\rceil,$$

Then we can find a prefix code C , such that

$$H_d(S) \leq L_S(C) < H_d(S)+1$$

- We can verify that for such lengths the Kraft-McMillan inequality holds. In fact, $\log_d \frac{1}{p_i} \leq l_i < \log_d \frac{1}{p_i} + 1$
$$\frac{1}{p_i} \leq d^{l_i} \rightarrow p_i \geq d^{-l_i} \rightarrow \sum_{i=1}^m d^{-l_i} \leq 1.$$
- Then, we can construct a prefix code.
- Multiply by p_i the inequality $\log_d \frac{1}{p_i} \leq l_i < \log_d \frac{1}{p_i} + 1$ and summing for each i , we can obtain the thesis.

NOTE: You don't always get compact codes

Shannon encoding is not optimal

- Let us consider a source with 3 elements. We want a binary compact code.

S	Prob.	$\log(1/\pi)$
s0	2/3	$\log(3/2) \approx 0,58$
s1	2/9	$\log(9/2) \approx 2,17$
s2	1/9	$\log(9) \approx 3,17$

- We can consider the lengths {1,3,4}.
In this way we obtain $L_S(C)=1.78$
- C is not a compact code, since the code C' has average code length $L_S(C') = 1.33$
- Which is the value of entropy?
- $H(S) \cong 1.22$.

S	C'
s0	0
s1	10
s2	11

Exercise

- Which are the codeword lengths of a binary code for the following source?

S	Prob.	$\log(1/p_i)$
s0	1/4	$\log(4)=2$
s1	3/4	$\log(4/3)=0.41$

- We consider the lengths $\{1, 2\}$.

- For instance:

S	Code 1
s0	10
s1	0

- The entropy is $H(S) \cong 0.81$. The average code length is $L_S(C) = 1.25$
- Code 1 is not a compact code since, if I had considered the trivial coding $s_0 \rightarrow 1$ and $s_1 \rightarrow 0$, I would obtain $L_S(C') = 1$. C' is a compact code.



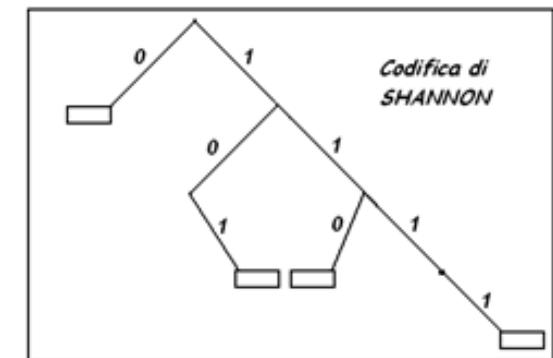
How to construct compact code?

Proposal by Shannon...

S	Prob.	$\log(1/\pi)$
s0	1/4	$\log(4)=2$
s1	3/4	$\log(4/3)=0.41$

- Shannon shows that if one wants to get closer to entropy, one can encode blocks of source elements.
- For instance, we can encode pairs of elements in S , i.e. the source S^2 .

S^2	Prob.	$\log(1/\pi)$	Code C_2
s0s0	1/16	$\log(16)=4$	1111
s0s1	3/16	$\log(16/3)=2.41$	101
s1s0	3/16	$\log(16/3)=2.41$	110
s1s1	9/16	$\log(16/9)=0.83$	0



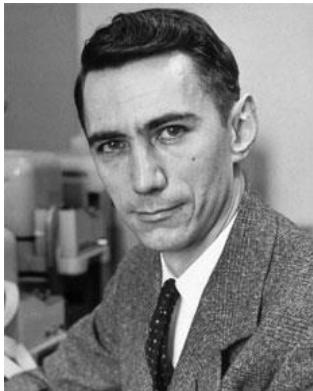
- In this case the codeword lengths are $\{1, 3, 3, 4\}$.
- Recall that $H(S) \cong 0.81$. We can prove that $H(S^2) = 2 \times H(S) \cong 1.62$ and that $L_{S^2}(C_2) \cong 1.93$.
- If we consider the average length for each symbol of S , we obtain $L_2 = \frac{L_{S^2}(C_2)}{2} \cong 0.96$

More in general

- We can prove that $H_d(S^n) = n \times H_d(S)$
- Furthermore, $H_d(S^n) \leq L_{S^n}(C) < H_d(S^n) + 1$
- If we consider the average length per symbol of S , we obtain:
$$H_d(S) = \frac{H_d(S^n)}{n} \leq \frac{L_{S^n}(C)}{n} < \frac{H_d(S^n)}{n} + \frac{1}{n} = H_d(S) + \frac{1}{n}$$
- Denoting by L_n the average length per symbol of S , i.e. $\frac{L_{S^n}(C)}{n}$ you get that if you take large enough blocks of S symbols, you reach entropy, i.e. $\lim_{n \rightarrow \infty} L_n = H_d(S)$.
- **Problem:** The number of source symbols can grow indefinitely. Having chosen a value of n , even large enough, we do not know whether we reach the optimal value.

A bit of history...

- It is important to find compact codes (also called **optimal**) to encode a memoryless source



- Claude Shannon and Robert Mario Fano were searching for uniquely decodable and instantaneous codes that were optimal. In 1949, they invented the so-called Shannon-Fano encoding.
- In 1951, David Huffman was a student of Robert M. Fano in the Electrical Engineering class. Prof. Fano posed his students the problem of finding optimal codes

Shannon-Fano encoding (binary case)

It is a very simple encoding

- ❑ The source symbols are listed in order of non-increasing probability.
- ❑ The list is divided into two groups of symbols whose sums of probability are almost equal.
- ❑ Each symbol in the first group is encoded with 0 and the others with 1.
- ❑ Each of the two groups is divided using the same criterion, adding the new coding symbols.
- ❑ The process continues until each group consists of only one element.

Example

a	$1/2$	0	
b	$1/4$	1 0	
c	$1/8$	1 1 0	
d	$1/16$	1 1 1 0	$H=1.9375$ bits
e	$1/32$	1 1 1 1 0	
f	$1/32$	1 1 1 1 1	$L=1.9375$ bits

Shannon-Fano encoding is not optimal

Entropy of the source $H=2.2328$ bits

<i>a</i>	0.35	00	0
<i>b</i>	0.17	01	100
<i>c</i>	0.17	10	101
<i>d</i>	0.16	110	110
<i>e</i>	0.15	111	111

$$L_1=2.31 \text{ bits} \quad L_2=2.3 \text{ bits}$$

Code Efficiency and Redundancy

- We know that entropy represents, on average, the number of symbols taken from a given code alphabet we need to represent a symbol from a given source.
- Given a code C , $L(C)$ represents the average length of its code words used to represent the symbols from the source.
- The code efficiency is the value $\vartheta(C) = \frac{H_d(S)}{L(C)}$.
From Shannon Theorem it follows that $0 \leq \vartheta(C) \leq 1$.
- The redundancy of a code C is $\rho(C) = 1 - \vartheta(C) = \frac{L(C) - H_d(S)}{L(C)}$.

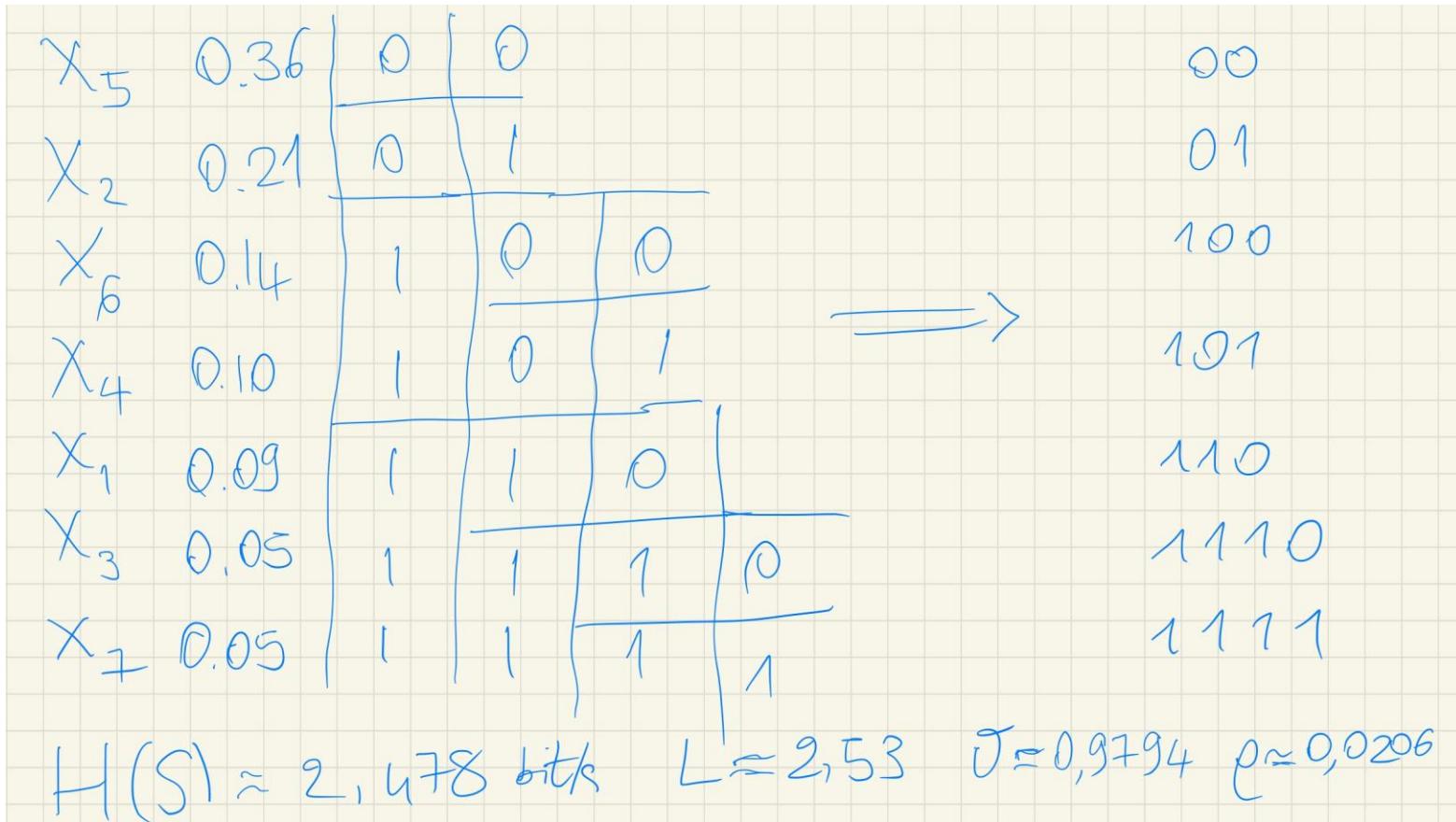
Exercise

- Find the Shannon-Fano encoding for the following source S:

simbolo	x ₁	x ₂	x ₃	x ₄	x ₅	x ₆	x ₇
Probabilità	0,09	0,21	0,05	0,1	0,36	0,14	0,05

- Compute the entropy and the average code length.
- Compute efficiency and redundancy

Solution...



Lab exercises

- Given a source of cardinality m , a set of m lengths and a code alphabet of cardinality d , how to construct a UD code having codewords with the lengths given as input, using the technique used in the proof of Kraft-McMillan theorem?
- Compare the average code length (once the source probabilities have been defined) with the entropy. Is it possible to deduce whether the code is compact?
- More in detail...

Lab exercises

Write a program that, given a memoryless source:

1. Checks the coherence of the probability set
2. Computes the entropy of the source
3. Given a set of lengths l_1, l_2, \dots, l_m and a positive integer d, constructs, if possible, the prefix code on a code alphabet of size d having codewords of length l_1, l_2, \dots, l_m
4. Given a code for the source, computes its average code length
5. Given a code alphabet, computes the Shannon encoding
6. Computes the binary Shannon-Fano encoding



Huffman Encoding

Friday, March 17 2023

Huffman Encoding

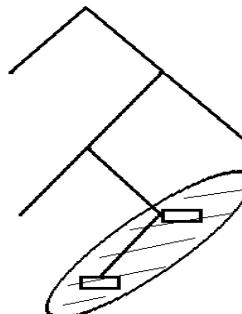
- + Huffman solved the problem posed by Prof. Fano of constructing optimal codes.
- + His result is described in the paper D.A. Huffman, "A method for the construction of minimum-redundancy codes", Proceedings of the I.R.E., settembre 1952, pagg. 1098-1102.



Before describing Huffman's encoding, let us consider a few remarks

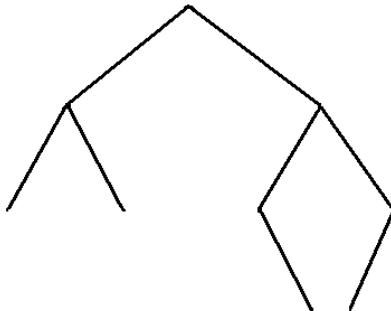
Some properties of optimal prefix codes

- + Let S be a memoryless source with a given probability distribution. Let us suppose that $p_1 \geq p_2 \geq \dots \geq p_m$.
- + Let C be a compact (or optimal) prefix code for S such that c_i is the codeword associated to the probability p_i . Then:
 1. If $p_j > p_k$ then $|c_k| \geq |c_j|$.
In fact, if this were not the case, exchanging c_j for c_k would result in a prefix code with lower average code length.
 2. $|c_m| = |c_{m-1}|$, i.e., the least expected symbols have codewords of the same length.
If it were $|c_m| > |c_{m-1}|$ we could remove the last symbol from c_m and still obtain a prefix code, but with a lower average code length.



Some properties of optimal prefix codes

3. Among the code words of maximum length, there are at least two that differ only by the last symbol. That is, the situation described in the figure cannot arise:



Let M be the maximum length of the codewords. If all codewords of maximum length differed by a symbol that was not the last, they could be replaced by the prefix of length $M-1$, preserving the property of being a prefix code. Such a code, however, would have a lower average code length.

Huffman algorithm (for binary codes)

- + Let S be a source $S = \begin{pmatrix} s_1 & \dots & s_m \\ p_1 & \dots & p_m \end{pmatrix}$ in which the symbols are sorted such that the probabilities are non-increasing: $p_1 \geq p_2 \geq \dots \geq p_m$.
- + We denote by $R(S)$ the reduced source, obtained by replacing the most unlikely symbols with a symbol whose probability is given by the sum of the probabilities of the two symbols it is going to replace:

$$R(S) = \begin{pmatrix} s_1 & \dots & (s_{m-1}, s_m) \\ p_1 & \dots & p_{m-1} + p_m \end{pmatrix}$$

- + Let C_R be a binary prefix code for the source $R(S)$ and let z be the codeword associated to the symbol (s_{m-1}, s_m) . Then, the C code obtained by assigning to each symbol of S the same codeword used for the corresponding symbol of $R(S)$ e by using $z0$ for c_m and $z1$ for c_{m-1} is a prefix code. Moreover, $L_S(C) = L_{R(S)}(C_R) + p_{m-1} + p_m$.
- + If C_R is optimal then C is optimal, as well. In fact, if not, there should be an optimal prefix code C' for S such that $L_S(C') < L_S(C)$.

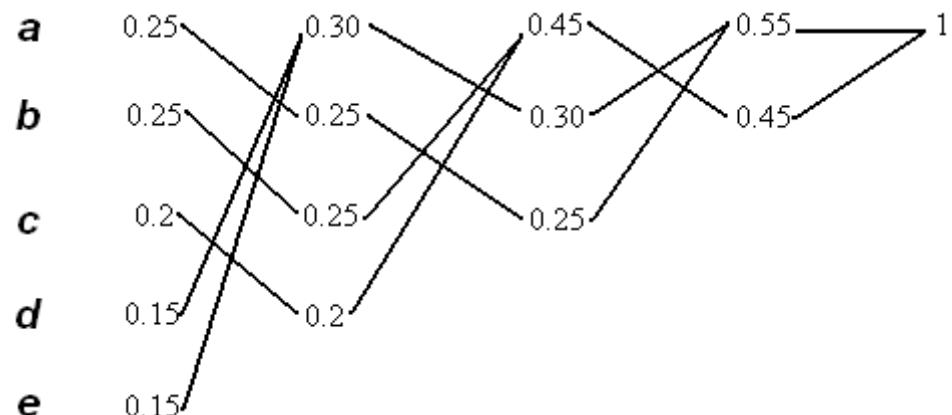
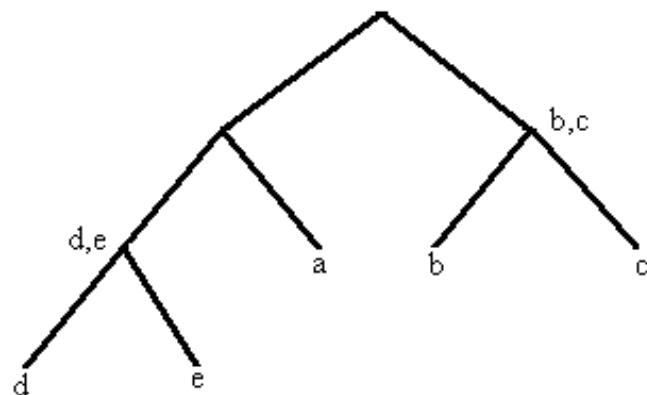
By using Property 3, at least two codewords of maximal length must differ only by the last symbol, let them be $t0$ and $t1$. Consider the code C'_R obtained from C' by replacing $t0$ and $t1$ with the codeword t . In this way we obtain a prefix code for $R(S)$, such that $L_S(C') = L_{R(S)}(C'_R) + p_{m-1} + p_m$. Then, we would have $L_{R(S)}(C'_R) < L_{R(S)}(C_R)$, that is a contradiction.

Huffman algorithm (for binary codes)

- + Given a source S , we construct a sequence of sources S_0, S_1, \dots, S_{m-2} such that $S_0=S$ and $S_{i+1} = R(S_i)$, until a source with only two symbols is reached (note that for a source with two symbols the optimal code has **0** and **1** as codewords).
- + Unlike Shannon-Fano encoding, the code tree is built bottom-up with the source symbols in the leaves.
- + The two leaves with the smallest probabilities become children of the same parent node having as probability the sum of the two probabilities.
- + When you get to the source with two symbols, you find a compact code and from this backwards you find a compact code for all sources until you get to S .
- + The Huffman code so constructed is not unique. Any Huffman code for a given source is optimal.

Example

- + Given the source $S = \begin{pmatrix} a & b & c & d & e \\ 0.25 & 0.25 & 0.2 & 0.15 & 0.15 \end{pmatrix}$
- + Let us construct an optimal binary code by using the Huffman algorithm.



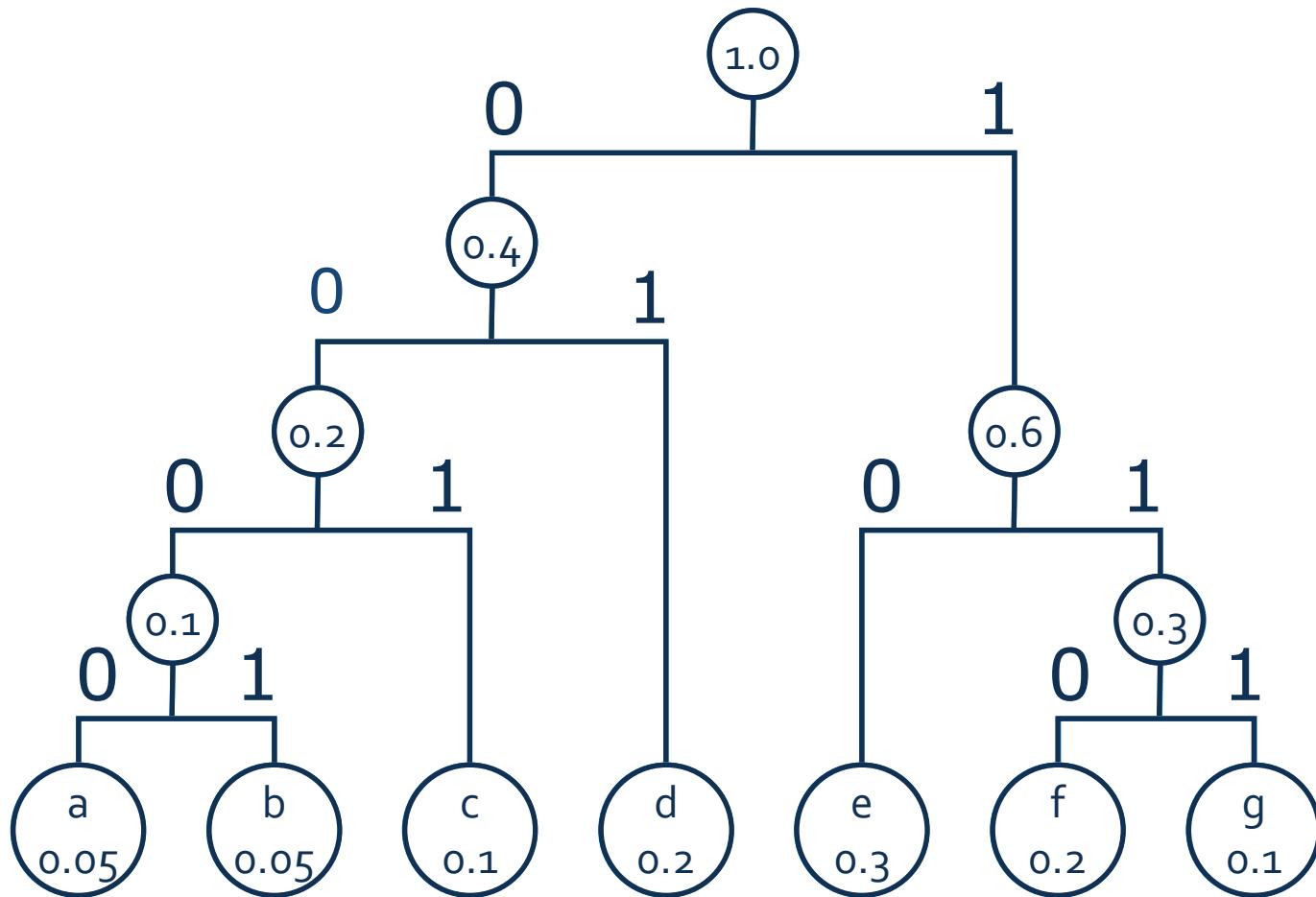
Solution

S	Code
a	01
b	10
c	11
d	000
e	001

- + The source with two symbols will be encoded with 0 and 1, thus obtaining the first compact or optimal code. The other encodings are obtained in a similar way.

Huffman encoding - example

Symbol	Prob.
a	0.05
b	0.05
c	0.1
d	0.2
e	0.3
f	0.2
g	0.1



Huffman encoding - example

Symbol	Prob.	Codeword
--------	-------	----------

a	0.05	0000
-----	------	------

Exercise: Compute $H(S)$ and $L_s(X)$

b	0.05	0001
-----	------	------

$H(S)=2.5464$

c	0.1	001
-----	-----	-----

$L_s(X)=2.6 !!$

d	0.2	01
-----	-----	----

*We are sure that there is no prefix code
with a lower average code length*

e	0.3	10
-----	-----	----

f	0.2	110
-----	-----	-----

g	0.1	111
-----	-----	-----

Optimal d-ary codes

- + The same procedure can be applied when the code alphabet has d symbols.
- + The smallest d probabilities are taken into account, and the corresponding d symbols are then combined to form one symbol. Then, each reduced source has $d-1$ fewer symbols than the previous one.
- + It is required that the last source has d symbols (so that we can construct the trivial compact code). This happens if and only if the initial source has $d+\alpha(d-1)$ symbols, with α being an integer.
- + If this does not happen, 'dummy symbols' are added to the source until this number of symbols is reached. Such added symbols will have zero probability and can therefore be ignored after the code is formed.

Example

- + Consider the following source with 11 symbols. If we want to obtain a compact quaternary prefix code, we must add two dummy symbols, since $13=4+3*3$.

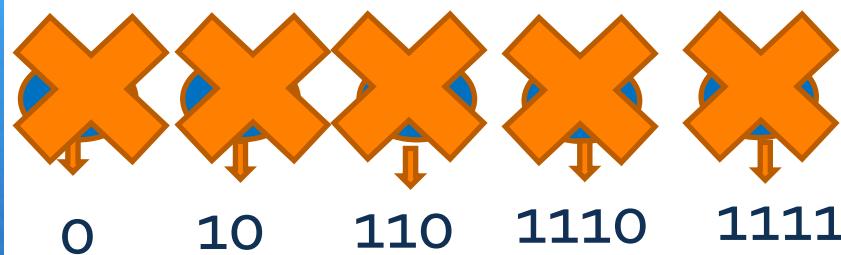
S	Prob
a	0.22
b	0.15
c	0.12
d	0.10
e	0.10
f	0.08
g	0.06
h	0.05
i	0.05
l	0.04
m	0.03
n	0.00
o	0.00

Solution

+ A possible solution is:

S	Prob	Codeword
a	0.22	2
b	0.15	3
c	0.12	00
d	0.10	01
e	0.10	02
f	0.08	03
g	0.06	11
h	0.05	12
i	0.05	13
l	0.04	100
m	0.03	101
n	0.00	102
o	0.00	103

How to apply Huffman algorithm to a given input text

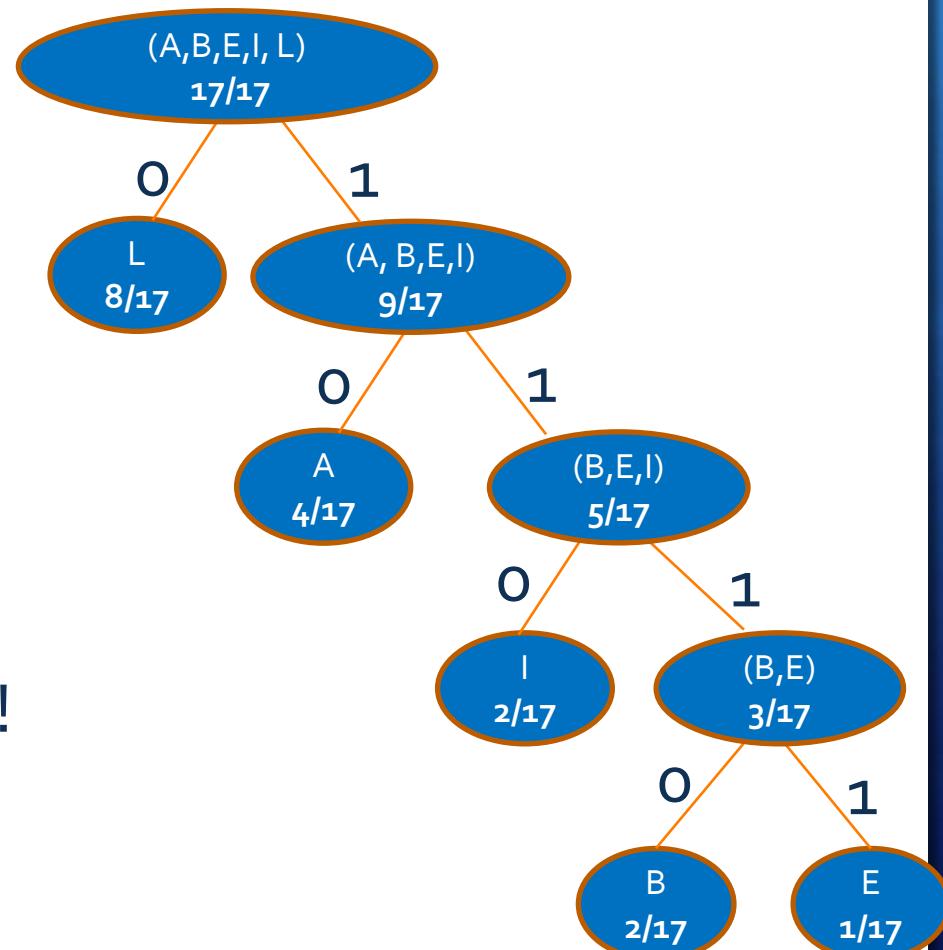


LA BELLA LILLI BALLA



0101110111100100110001101110100010 34 bit!

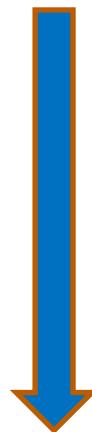
Compression ratio: $34/136 = 1/4 = 25\%$



Decoding algorithm

0101110111100100110001101110100010

010 1110 1111 0010 0110 00110 1110 10 0010



L, 8

A, 4

I, 2

B, 2

E, 1

O

10

110

1110

1111

$34/17 = 2$ bit per symbol!

LA BELLA LILLI BALLA



Huffman compression algorithm with binary codes

1. Scan the whole input T , counting the occurrences of each symbol a
2. Determines the probability of each symbol based on its occurrences, i.e.
$$p(a) = \frac{occ(a)}{|T|}$$
3. Sort the probabilities in a non-increasing order.
4. Generate leaves nodes for each symbol containing the respective probabilities, and add them to the queue.
5. While (Nodes in Queue > 1)
 1. Dequeue the two nodes s and t with smallest probabilities
 2. Add 0 and 1 to the codewords of the left and right node.
 3. Create a new node u with a value equal to the sum of the probabilities of the two nodes
 4. Assign the first node s as the left child and the second node t as the right child.
 5. Add the node u to the queue.
6. The last remaining node in the queue is the root of the Huffman tree.

Huffman encoding - example

Symbol	Prob.	Codeword
a	0.05	0000
b	0.05	0001
c	0.1	001
d	0.2	01
e	0.3	10
f	0.2	110
g	0.1	111

+ Let us encode the text

aeebcddegfced

Sol: **0000 10 10 0001 001 01 01 10 111 110 001 10 01**

Huffman decoding - example

Symbol	Prob.	Codeword
--------	-------	----------

a	0.05	0000
---	------	------

+ Let us decode the message

b	0.05	0001
---	------	------

011100100100000111110

c	0.1	001
---	-----	-----

d	0.2	01
---	-----	----

e	0.3	10
---	-----	----

Sol: dfdcadgf

f	0.2	110
---	-----	-----

g	0.1	111
---	-----	-----

Exercises

1. Let S be a source with the following probability distribution $(1/3, 1/3, 1/4, 1/12)$.
Describe a Huffman encoding for source S .
Questions:
 - A. Can encodings with lengths $(2,2,2,2)$ or $(1,2,3,3)$ be considered?
 - B. If so, which is the average code length for both the encodings?
2. Which is the Huffman encoding if the prob. distrib. is $(1/4, 1/4, 1/4, 1/4)$?
3. Find the Shannon encoding for the source S
4. Finding the Shannon-Fano encoding for the source S
5. Compare the length of the corresponding codeword for each symbol
6. Compare the average code length for such encodings.

Performance of a compressor

Several measures are commonly used to express the performance of a compression method. They are generally evaluated on the texts to which the compression methods are applied.

- + Compression ratio:
$$\frac{\text{size of the compressed text}}{\text{size of the non-compressed text}}$$
- + Compression factor:
$$\frac{\text{size of the non-compressed text}}{\text{size of the compressed text}}$$

The universal compressor does not exist

- + This is proved by a counting argument. Suppose that a universal compressor exists, which compresses all strings of length n . There exist exactly 2^n distinct binary strings of length n .
- + A universal compressor must encode each input by different encodings. Otherwise, the decompressor would not be able to decode correctly. However, there are only $2^n - 1$ strings of length smaller than n .
- + In fact, the most strings are not compressed very much. The fraction of strings that can be compressed from n to m bits is at most $2^{(n-m)}$.
- + Each compressor will therefore compress some inputs, expand others.

Limitations of Huffman encoding

- + Huffman encoding requires creating a coding table based on symbol frequency, which requires preliminary analysis of the input data. This can require significant time and computational resources, especially for large data sets.
- + The coding table must be transmitted since the decoding requires maintaining a mapping between the coding symbols and the original symbols. This requires space (the compressed message is longer)
- + In fact, two steps are needed
 - + **First step:** estimating the probabilities of each symbol (model construction) and building the Huffman tree
 - + **Second step:** encoding the message
- + Risk of data loss: if the coding table is lost or corrupted, it may be impossible to decompress the data

Time complexity for Huffman encoding

- + If a priority queue implemented using min-heap is used, the algorithm has execution time $O(n \log n)$ if n is the number of elements to be encoded.
- + If the elements are ordered according to probability values, the algorithm can become linear using two queues.

How does the decoding work?

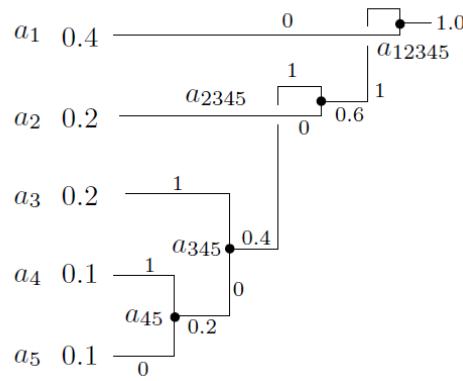
- + Remind that it is necessary to know the Huffman tree or source probabilities in order to reconstruct it. This increases the size of the compressed text but can also increase the decoding execution time.
- + Once the Huffman tree is known, the decoding algorithm is simple. One starts at the root and reads the bits from the compressed text. If the bit is zero, you go to the left, if it is 1 you go to the right of the node; you continue reading the compressed text until you reach a leaf, which determines which character is to be encoded. This character is output by the decoder.

Faster methods exist

- + Decoding a compressed file via Huffman by traversing the code tree for each symbol is conceptually simple, but slow. The compressed file must be read bit by bit and the decoder must scan the code tree node by node for each bit.
- + Choueka et al (1985) proposed a method, later adopted by others, which uses partially decoded tables.
- + These tables depend on the particular Huffman code used, but not on the data to be decoded. The compressed file is read in chunks of k bits each (where k is normally 8 or 16 but can have other values) and the current chunk is used as a pointer to a table. The table entry selected in this way can decode different symbols and point to the table to be used for the next chunk.

Example

- + Suppose we have the following tree, which generates the codewords 0, 10, 111, 1101, and 1100



- + The string $a_1a_1a_2a_4a_3a_1a_5\dots$ is compressed by
 $0|0|10|1101|111|0|1100$
 - + Let us consider $k=3$, then we consider the blocks
 $001|011|011|110|110|0\dots$

Esempio

- + We construct $k+1$ tables, and given the text $001|011|011|110|110|\dots$
We can decode by using the tables:

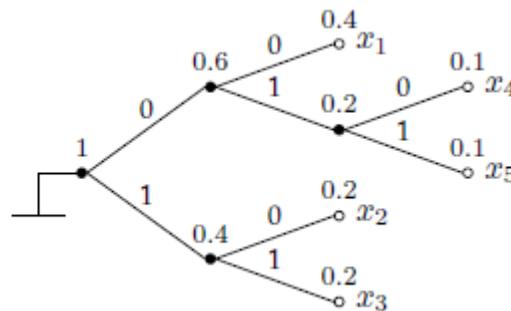
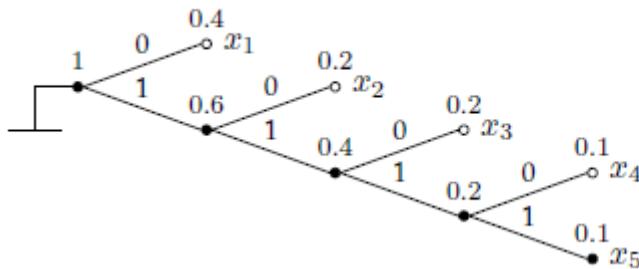
$T_0 = \Lambda$	$T_1 = 1$	$T_2 = 11$	$T_3 = 110$
000 $a_1a_1a_1$ 0	1 000 $a_2a_1a_1$ 0	11 000 a_5a_1 0	110 000 $a_5a_1a_1$ 0
001 a_1a_1 1	1 001 a_2a_1 1	11 001 a_5 1	110 001 a_5a_1 1
010 a_1a_2 0	1 010 a_2a_2 0	11 010 a_4a_1 0	110 010 a_5a_2 0
011 a_1 2	1 011 a_2 2	11 011 a_4 1	110 011 a_5 2
100 a_2a_1 0	1 100 a_5 0	11 100 $a_3a_1a_1$ 0	110 100 $a_4a_1a_1$ 0
101 a_2 1	1 101 a_4 0	11 101 a_3a_1 1	110 101 a_4a_1 1
110 — 3	1 110 a_3a_1 0	11 110 a_3a_2 0	110 110 a_4a_2 0
111 a_3 0	1 111 a_3 1	11 111 a_3 2	110 111 a_4 2

- + Large k values speed up decoding, but require large tables. A large alphabet also requires a large list of tables.
- + In addition, the decoder must set the tables before decoding can begin, and this process can result in decoding being slowed down. There is a trade-off between the value of k and the speed of decoding

Some remarks on Huffman encoding

- + Huffman codes are codes with minimal redundancy.
- + However, although with minimal redundancy, it is better to use codes with **minimal variance** (i.e., where the difference between codeword lengths is minimal). This is because many communication schemes often use buffers of a fixed size.
- + Example:

x	x_1	x_2	x_3	x_4	x_5
$p(x)$	0.4	0.2	0.2	0.1	0.1



- + The right-hand alternative is more convenient.
- + To obtain minimum variance codes, it is better to merge lower subtrees where a choice is possible.

Bounds for redundancy in Huffman codes

- + An upper bound on the redundancy was proved in 2002 in the paper "A Simple Upper Bound on the Redundancy of Huffman Codes" by C. Ye e R.W. Yeung, IEEE transactions on information theory, vol. 48, no. 7, 2002:

$$\cong p_1 + 0.086$$

$$\rho \leq \begin{cases} p_1 + 1 - \log_2 e + \log_2 \log_2 e , & \text{if } p_1 < \frac{1}{2} \\ 2 - p_1 - H_2(p_1), & \text{if } p_1 \geq \frac{1}{2} \end{cases}$$

The bound is tight also when $p_1 \geq \frac{1}{2}$.

- + Moreover, lower bounds on the redundancy also exist
- $$\rho \geq 1 - H_2(p_1), \text{ se } p_1 \leq 0.4$$

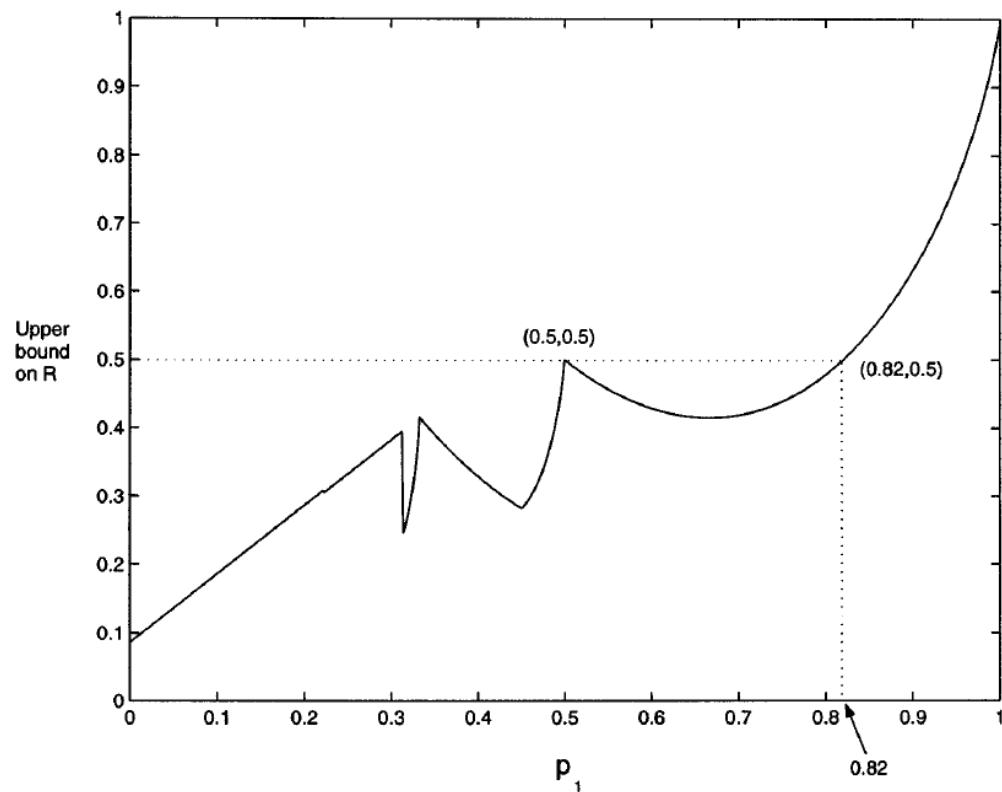
Bounds for redundancy in Huffman codes

- Several studies have obtained more and more refined and specific bounds. For example, it has been shown in several scientific papers that $\rho(C) \leq f(p_1)$, where

$$f(p_1) = \begin{cases} 2 - p_1 - H_b(p_1), & \text{if } 0.5 \leq p_1 < 1 \\ 3 - 5p_1 - H_b(2p_1), & \text{if } 0.4505 \leq p_1 < 0.5 \\ 1 + 0.5(1 - p_1) - H_b(p_1), & \text{if } \frac{1}{3} \leq p_1 < 0.4505 \\ 3 - 7.7548p_1 - H_b(3p_1), & \text{if } 0.3138 \leq p_1 < \frac{1}{3} \\ 2 - 1.25(1 - p_1) - H_b(p_1), & \text{if } 0.2 \leq p_1 < 0.3138 \\ 4 - 18.6096p_1 - H_b(5p_1), & \text{if } 0.1971 \leq p_1 < 0.2 \\ 2 - 1.3219(1 - p_1) \\ \quad - H_b(p_1), & \text{if } \frac{1}{6} \leq p_1 < 0.1971 \\ p_1 + 0.086, & \text{if } p_1 < \frac{1}{6}. \end{cases}$$

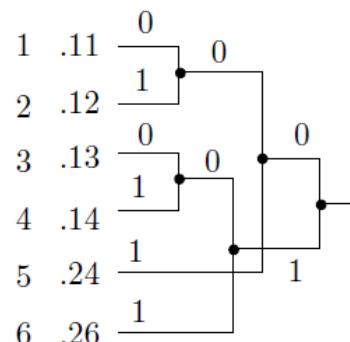
Upper bound on the redundancy

- + Redundancy approaches 1 when the probability of the most frequent symbol tends to 1

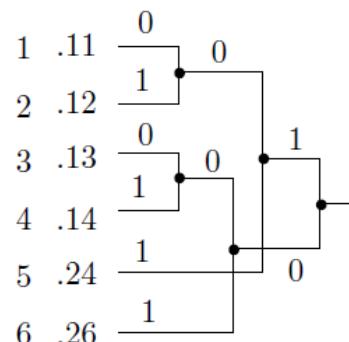


How many Huffman trees?

- + Given a source S with n symbols how many Huffman trees can be generate?



(a)



(b)

000 100 000
001 101 001
100 000 010
101 001 011
01 11 10
11 01 11

(a) (b) (c)

- + There are also encodings that have the same properties as Huffman codes but do not correspond to any Huffman trees

Canonical Huffman encoding

- + Sending the code table or Huffman tree can be expensive, in terms of time and space.
- + There is a variant of Huffman's encoding, called CANONICAL HUFFMAN ENCODING, which does not need to send the tree or table but only the lengths of the codewords.
- + This is sufficient to decode the message. It is useful when the alphabet size is large and decoding must be very fast.

Adaptive version of Huffman encoding

- + Adaptive Huffman encoding was independently designed by Faller (1973) and Gallager (1978)
- + Knuth made improvements to the original algorithm (1985) and the resulting algorithm is known as the **FGK algorithm**.
- + A more recent version was proposed by Vitter (1987) and called the **V algorithm**

The idea

- + The basic idea is to build a Huffman tree that is optimal for the part of the message that has been considered up to a certain moment, and to reorganise it when necessary, in order to maintain its optimality

Pro & Contra - I

- + Adaptive Huffman encoding creates code words using a **variable estimate of the probability of the symbols** emitted by the source
 - + The locality is well exploited

For example, suppose the message begins with a series of characters that are never repeated thereafter. In static Huffman encoding, such characters will be at the bottom of the tree due to the fact that they are very rare in the message as a whole, and will therefore be encoded with many bits. In contrast, in adaptive encoding, characters will be inserted immediately in the leaves closest to the root, only to be pushed further down as more frequent characters appear.

Pro & Contra - II

- + Only a scan of the text is needed
- Sometimes, adaptive encoding requires more bits than static encoding (without overhead, i.e. without taking into account the fact that with Huffman it is necessary to include symbol probabilities or a tree representation in the compressed message, which costs $O(n)$)
- = However, in general, adaptive encoding is competitive with static encoding if overhead is also considered

Lab exercises

- + Portfolio: Write a program in a programming language of your choice given an input text find the Huffman encoding of the text. A decoding procedure to recover the original message by starting from the Huffman encoding is also required.

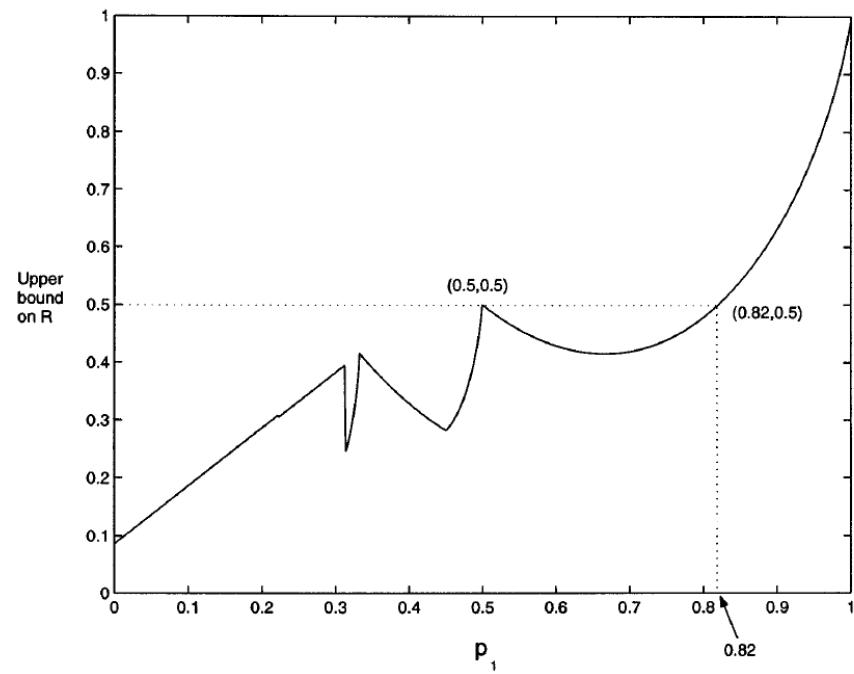


Arithmetic coding

Monday March 20, 2023

Recall that:

- + Given a C code, $L(C)$ represents the average length of its code words to represent source symbols.
- + The efficiency of the code is the value $\vartheta(C) = \frac{H_d(S)}{L(C)}$. From Shannon theorem it follows that $0 \leq \vartheta(C) \leq 1$.
- + The redundancy of the code is $\rho(C) = 1 - \vartheta(C) = \frac{L(C) - H_d(S)}{L(C)}$.
- + In case of Huffman codes, the redundancy approaches to 1 when the most frequent symbol has probability close to 1.



Can we do better than Huffman?

- + It has therefore been proved that one of the main disadvantages of static Huffman encoding concerns the case where there is a symbol that has very high probability.
- + One way to overcome this limitation is to use blocks of characters as symbols. In this way, the per-symbol inefficiency is spread over the entire block.
- + However, the use of blocks is difficult to implement since there has to be a block for every possible combination of characters and thus the number of groups increases greatly with the length of the blocks.

Can we do better than Huffman?

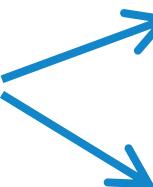
- Huffman Coding is optimal in its framework:

- ~~Static model~~



Adaptive Huffman

- ~~One symbol, one codeword~~



Block
coding

arithmetic
coding

The key idea in arithmetic coding

- + Arithmetic coding was proposed by Peter Elias (another student of Fano together with Huffman) and first presented by Abramson in his book Information Theory and Coding (1963). Early implementations were developed by [Rissanen 76], [Pasco 76], and [Rubin 79]. [Moffat et al. 98] and [Witten et al. 87] discussed many theoretical and practical details of the encoding.
- + Arithmetic coding completely overcomes the idea of replacing an input symbol with a codeword.
- + Instead, it transforms a stream of input symbols into **a single floating-point number in [0,1]**
- + The longer and more complex the message, the more bits are required to represent the output number.

The key-idea

- + The output of the arithmetic coding algorithm is, as usual, a stream of bits
- + However, it can be thought of as if there were an initial bit 0, and the stream represents a fractional number in binary, between 0 and 1

01101010 → 0.01101010

Bit streams and binary representations of a fractional number

- + A bit stream $b_1 b_2 b_3 \dots b_k$ can be interpreted as a real number in the range $[0,1)$ by prepending «0,» to it, i.e.:

$$0, b_1 b_2 \dots b_k = \sum_{i=1}^k b_i 2^{-i}$$

- + Any real number x in the range $[0,1)$ can be converted in a possibly infinite sequence of bits by using the following algorithm. The loop has to end when a level of accuracy is defined:

Converter(x)

Require: A real number $x \in [0, 1)$.

Ensure: The string of bits representing x .

```
1: repeat
2:    $x = 2 * x$ 
3:   if  $x < 1$  then
4:     output = output :: 0
5:   else
6:     output = output :: 1
7:      $x = x - 1$ 
8:   end if
9: until accuracy
```

:: denotes concatenations among bits

Example:

How to encode 0.825

- + $0,825 \times 2 = 1,65 \quad U=1$
- + $0,65 \times 2 = 1,3 \quad U=1$
- + $0,3 \times 2 = 0,6 \quad U=0$
- + $0,6 \times 2 = 1,2 \quad U=1$
- + $0,2 \times 2 = 0,4 \quad U=0$
- + $0,4 \times 2 = 0,8 \quad U=0$
- + $0,8 \times 2 = 1,6 \quad U=1$
- + $0,6 \times 2 = 1,2 \quad U=1$
- +

When we find a value we have seen before,
then we can stop the loop
and the representation is periodic

Ex: $0,11010011001\dots = 0,1\overline{101001}$

Exercise:

- + Finding the binary representation of $1/3$:
 - + $\overline{01}$

- + Finding the binary representation of $3/32$:
 - + $00011\bar{0}$

Encoding algorithm

Arithmetic coding

INPUT: String to be encoded & alphabet with probability distribution (probabilities are calculated using frequencies, or another source)

1. Initialize the range $[0,1]$
2. For each symbol of the input string:
 1. Divide the current interval into n subintervals ($n =$ cardinality of the alphabet). Each subinterval will have a size proportional to the probability associated with each element in the initial alphabet.
 2. Select as the current interval the one corresponding to the symbol being analyzed (input)
3. The lower bound of the last interval (converted to binary) is the result of our encoding (output)

Example

+ Let us encode the string **BILL GATES**

We need the frequencies of the characters in the text

chr	freq.
space	0.1
A	0.1
B	0.1
E	0.1
G	0.1
I	0.1
L	0.2
S	0.1
T	0.1

Note: To explain the algorithm with the example, the numbers we show will be in base 10, but of course in practice they are always in base 2

We partition the interval according to the probabilities

character	probability	range
space	0.1	[0.00, 0.10)
A	0.1	[0.10, 0.20)
B	0.1	[0.20, 0.30)
E	0.1	[0.30, 0.40)
G	0.1	[0.40, 0.50)
I	0.1	[0.50, 0.60)
L	0.2	[0.60, 0.80)
S	0.1	[0.80, 0.90)
T	0.1	[0.90, 1.00)

Encoding the text

chr	low	high
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
<i>Space</i>	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

The final value 0.2572167752 is the encoding of the string **BILL GATES**

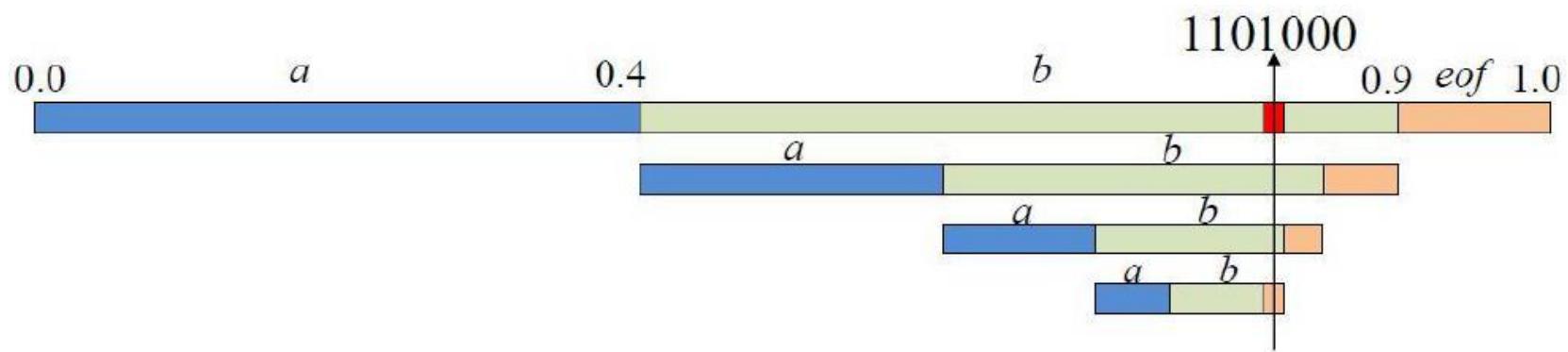
Another example

A	B	C
0.4	0.5	0.1

+ Input: BBBC

Intervallo Corrente	A	B	C	Input	Azione
[0,1)	[0, 0.4)	[0.4, 0.9)	[0.9, 1)	B	Dividi
[0.4, 0.9)	[0.4, 0.6)	[0.6, 0.85)	[0.85, 0.9)	B	Dividi
[0.6, 0.85)	[0.6, 0.7)	[0.7, 0.825)	[0.825, 0.85)	B	Dividi
[0.7, 0.825)	[0.7, 0.75)	[0.75, 0.812)	[0.812, 0.825)	C	Stop
[0.812, 0.825)					

...continued...



- + The final range is $[0.812, 0.825)$ and its binary representation is $[0.1101000, 0.1101001)$
- + The final range is identified by its lower bound. The sequence 1101000 (7 bits) is transmitted

How to proceed?

- + Finding the arithmetic coding of the string SWISS_MISS and the following statistical model (the same model will be used at each step of the algorithm):

Char	Freq	Prob.	Range	CumFreq
		Total CumFreq =		10
S	5	$5/10 = 0.5$	[0.5, 1.0)	5
W	1	$1/10 = 0.1$	[0.4, 0.5)	4
I	2	$2/10 = 0.2$	[0.2, 0.4)	2
M	1	$1/10 = 0.1$	[0.1, 0.2)	1
	1	$1/10 = 0.1$	[0.0, 0.1)	0

- + The cumulative frequency is used to decode the message
- + The symbols and the frequencies must be sent before the compressed text
- + The encoding process needs two variables, Low e High, initialized to 0 and 1 (initial interval)

How to proceed?

- + Low and High are updated at each step as follows:

$\text{NewHigh} := \text{OldLow} + \text{Range} * \text{HighRange}(X);$

$\text{NewLow} := \text{OldLow} + \text{Range} * \text{LowRange}(X);$

where

$\text{Range} = \text{OldHigh} - \text{OldLow}$

$\text{LowRange}(X)$ and $\text{HighRange}(X)$ are the lower bound and the upper bound of the interval corresponding to X , respectively.

The encoding process

Char.		The calculation of low and high
S	L	$0.0 + (1.0 - 0.0) \times 0.5 = 0.5$
	H	$0.0 + (1.0 - 0.0) \times 1.0 = 1.0$
W	L	$0.5 + (1.0 - 0.5) \times 0.4 = 0.70$
	H	$0.5 + (1.0 - 0.5) \times 0.5 = 0.75$
I	L	$0.7 + (0.75 - 0.70) \times 0.2 = 0.71$
	H	$0.7 + (0.75 - 0.70) \times 0.4 = 0.72$
S	L	$0.71 + (0.72 - 0.71) \times 0.5 = 0.715$
	H	$0.71 + (0.72 - 0.71) \times 1.0 = 0.72$
S	L	$0.715 + (0.72 - 0.715) \times 0.5 = 0.7175$
	H	$0.715 + (0.72 - 0.715) \times 1.0 = 0.72$
□	L	$0.7175 + (0.72 - 0.7175) \times 0.0 = 0.7175$
	H	$0.7175 + (0.72 - 0.7175) \times 0.1 = 0.71775$
M	L	$0.7175 + (0.71775 - 0.7175) \times 0.1 = 0.717525$
	H	$0.7175 + (0.71775 - 0.7175) \times 0.2 = 0.717550$
I	L	$0.717525 + (0.71755 - 0.717525) \times 0.2 = 0.717530$
	H	$0.717525 + (0.71755 - 0.717525) \times 0.4 = 0.717535$
S	L	$0.717530 + (0.717535 - 0.717530) \times 0.5 = 0.7175325$
	H	$0.717530 + (0.717535 - 0.717530) \times 1.0 = 0.717535$
S	L	$0.7175325 + (0.717535 - 0.7175325) \times 0.5 = 0.71753375$
	H	$0.7175325 + (0.717535 - 0.7175325) \times 1.0 = 0.717535$

+ We produce as output 71353375



Canonical Huffman Encoding

Monday March 20, 2022

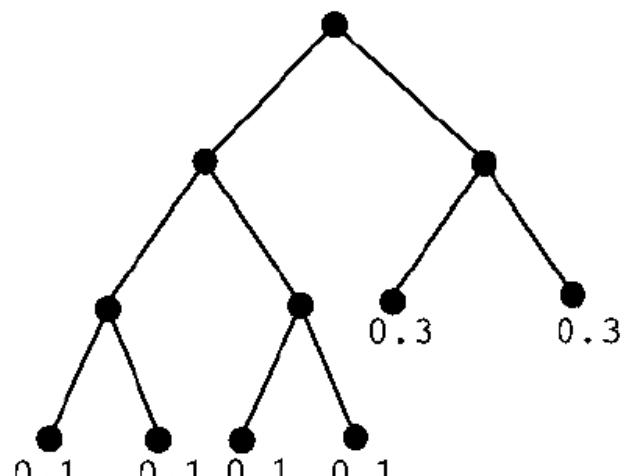
Recall that...

- + The compressed file produced by Huffman algorithm consists of two parts: the table code which contains an encoding of the Huffman tree, and thus has size $\Theta(|A|)$ where A is the alphabet, and the body which contains the codewords of the symbols in the input text T .
- + The size of the preamble is usually dropped from the evaluation of the length of the compressed file; even if this might be a significant size for large alphabets. So, the alphabet size cannot be underestimated, and it must be carefully taken into account.

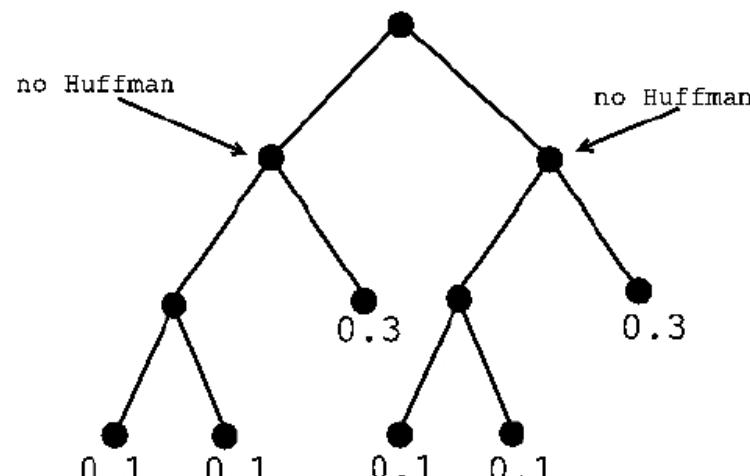
Limitations of the Huffman encoding

- + Huffman encoding has two important limitations:
 - + the tree structure needs to be stored. It can be very consuming when the alphabet is large.
 - + Decoding is slow because you need to traverse the tree for each code word. Moreover, traversing the tree from the root to the leaves involves following many pointers, with limited code locality.
- + There is an elegant variant of Huffman encoding, called canonical, which allows significantly faster decoding using relatively small space.

Example of an optimal code not obtainable by the Huffman algorithm



Huffman

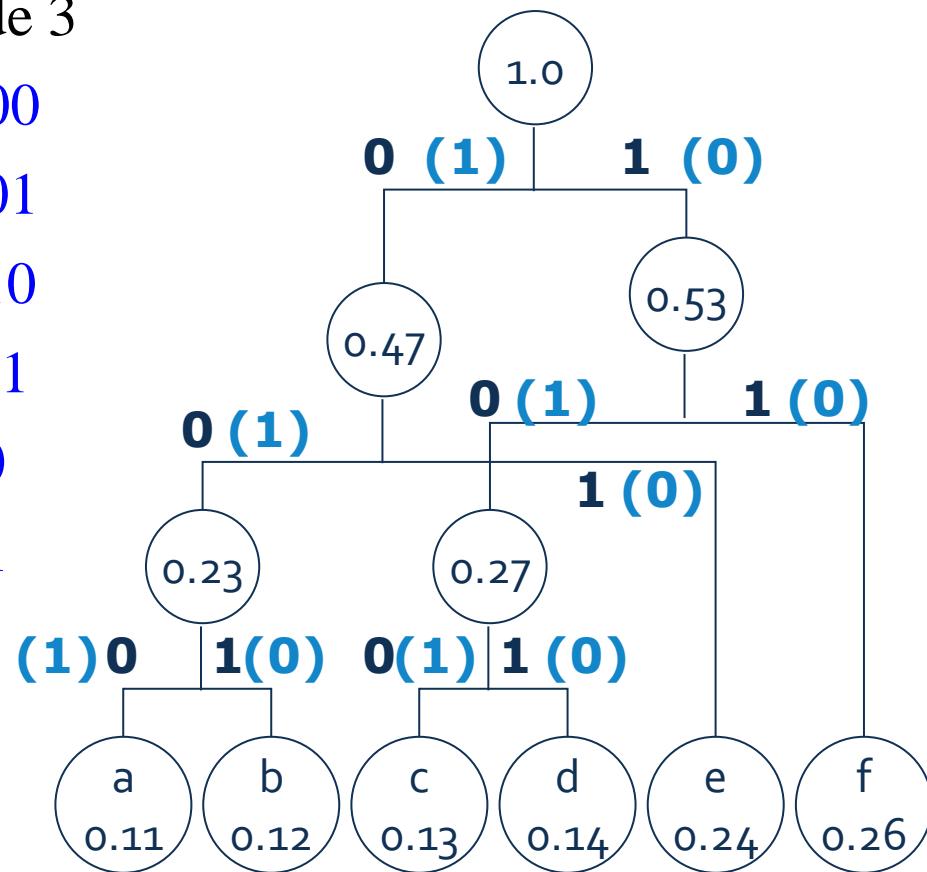


Not Obtainable by Huffman Algorithm

There exist optimal codes not corresponding to any Huffman tree

Symb.	Prob.	Code 1	Code 2	Code 3
a	0.11	000	111	000
b	0.12	001	110	001
c	0.13	100	011	010
d	0.14	101	010	011
e	0.24	01	10	10
f	0.26	11	00	11

?



A new encoding

Symb.	Code 3
a	000
b	001
c	010
d	011
e	10
f	11

- + It is an encoding equivalent to the one produced by Huffman, since the code is prefix as well, and the codeword lengths are the same
- + *Such an encoding has a useful numerical property*
 - + Codewords of the same length are lexicographically sorted
 - + When codewords are lexicographically sorted, they are sorted from longest to shortest, as well.

Benefits of this representation

- + The main advantage is that it is not necessary to store the tree in order to decode it.
- + All that we need are:
 - + A list of symbols sorted according to the lexicographic order of the codewords
 - + An array with the first codeword for each distinct length

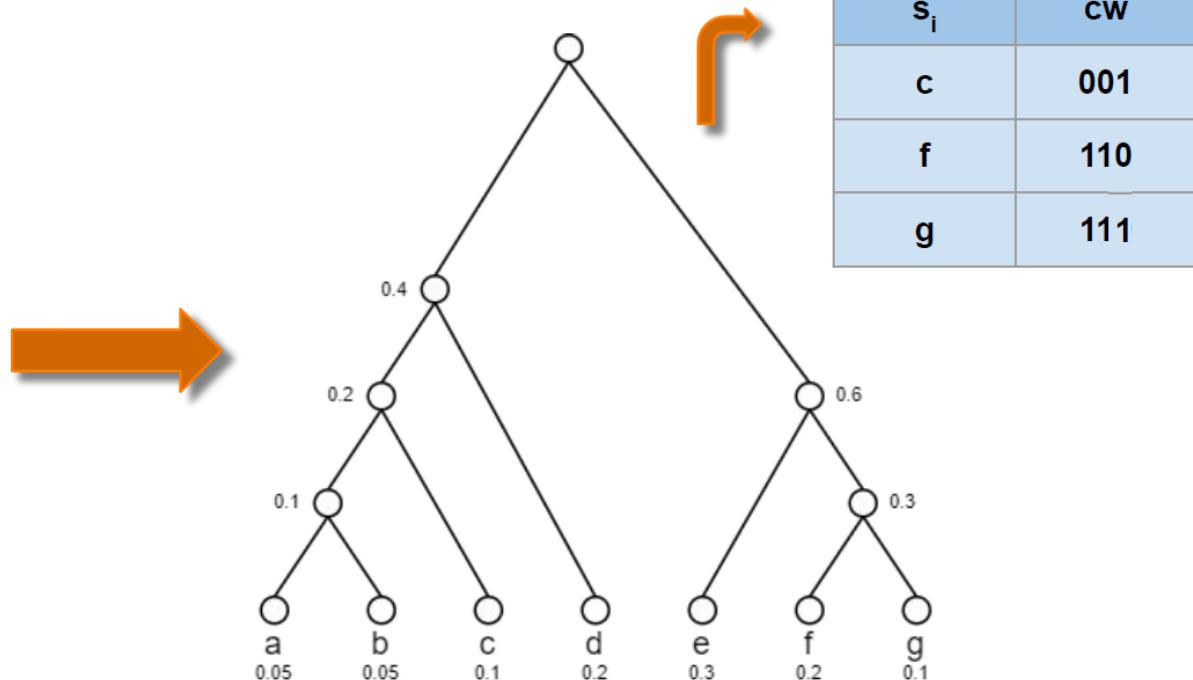
How to compute the canonical Huffman encoding?



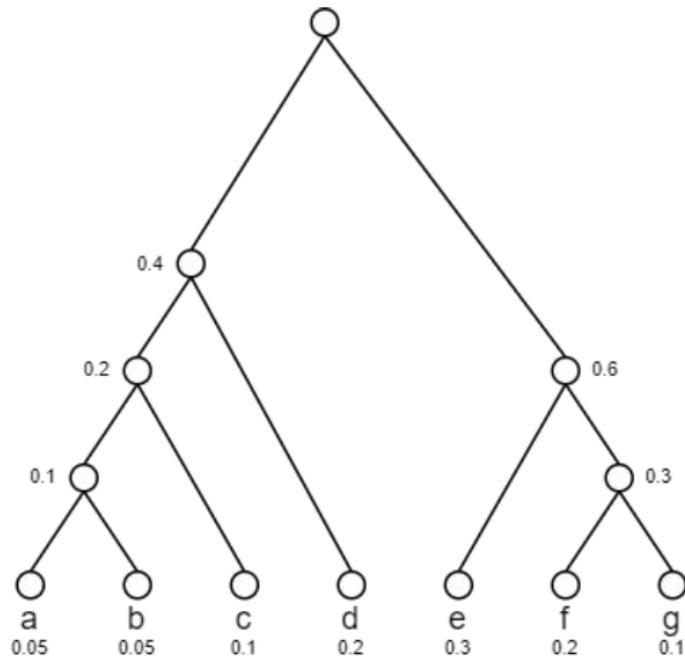
From Huffman to Canonical Huffman

- + Let us compute the Huffman encoding for the following source:

s_i	p_i
a	0.05
b	0.05
c	0.1
d	0.2
e	0.3
f	0.2
g	0.1



From Huffman to Canonical Huffman



We should obtain the new encoding similar to the following(new cw):

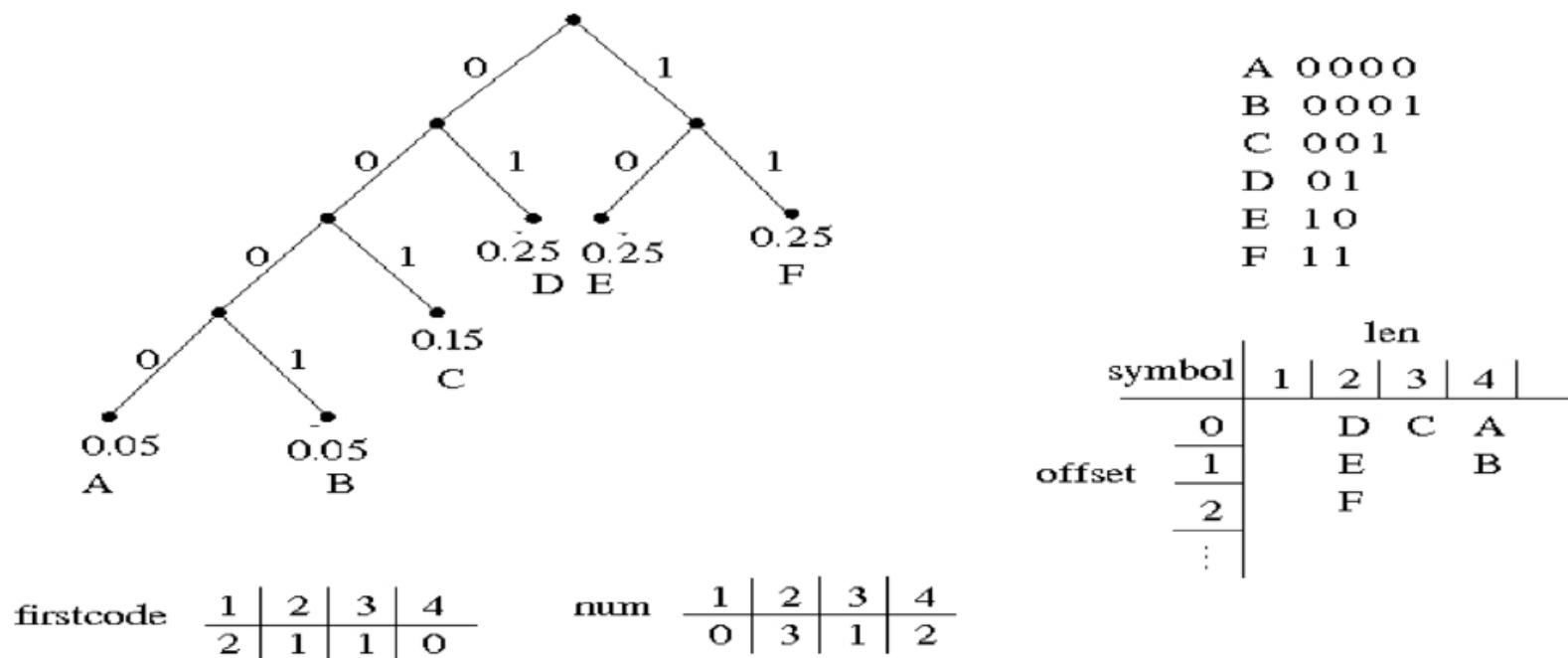
s_i	d	e	c	f	g	a	b
l_i	2	2	3	3	3	4	4
old cw	01	10	001	110	111	0000	0001
new cw	10	11	001	010	011	0000	0001

Canonical Huffman Encoding

The canonical encoding is obtained as follows:

1. Compute the codeword length $L(\delta)$ for each symbol $\delta \in A$
2. Construct the array **num** which stores in the entry $\text{num}[\ell]$ the number of symbols having Huffman codeword of ℓ -bits
3. Construct the array **symb** which stores in the entry $\text{symb}[\ell]$ the list of symbols having Huffman codeword of ℓ -bits
4. Construct the array **firstcode (fc)** which stores in the entry $\text{fc}[\ell]$ the first codeword of all symbols encoded with ℓ bits.
5. Assign consecutive codewords to the symbols in $\text{symb}[\ell]$ starting from the codeword in $\text{fc}[\ell]$.

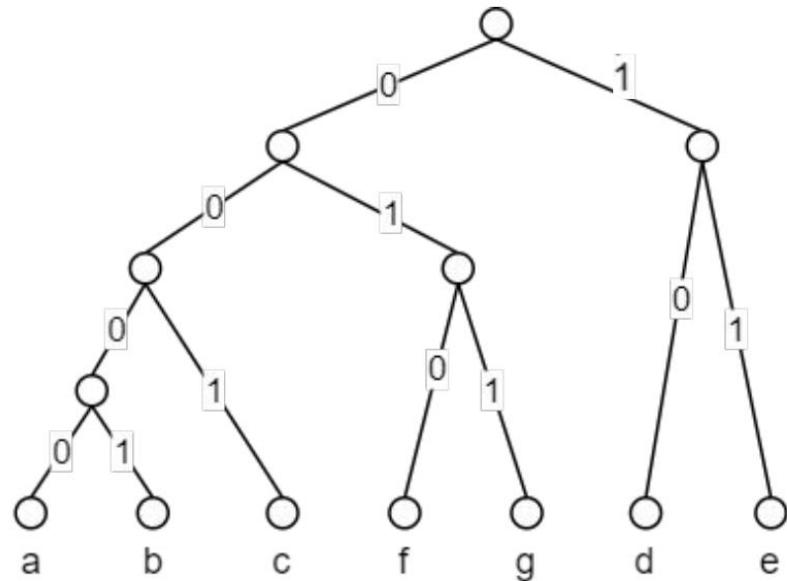
An overview of the data structures



Size of the compressed text

- + The num array is actually useless, so that the Canonical Huffman encoding needs only to store fc and symb arrays, which means
 - + at most \max^2 bits to store fc (i.e. max codewords of length at most max each),
 - + at most $(|A| + \max) \log_2 (|A| + 1)$ bits to encode table symb.
- + Consequently, the key advantage of Canonical Huffman is that we do not need to store the tree-structure via pointers, with a saving of $\Theta(|A| \log_2 (|A| + 1))$ bits.

Canonical Huffman Encoding



s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

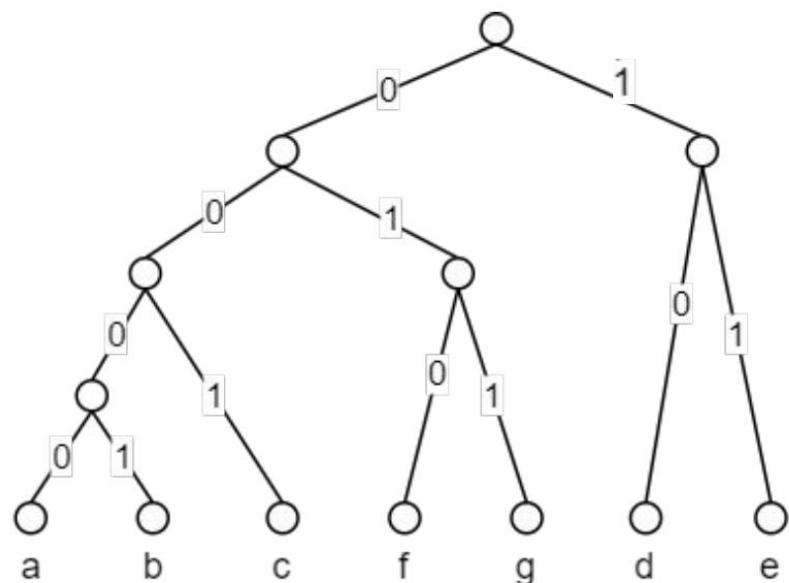
num

1	2	3	4
0			

symbol

1	2	3	4

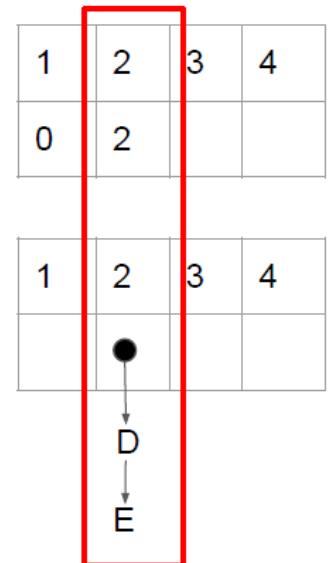
Canonical Huffman Encoding



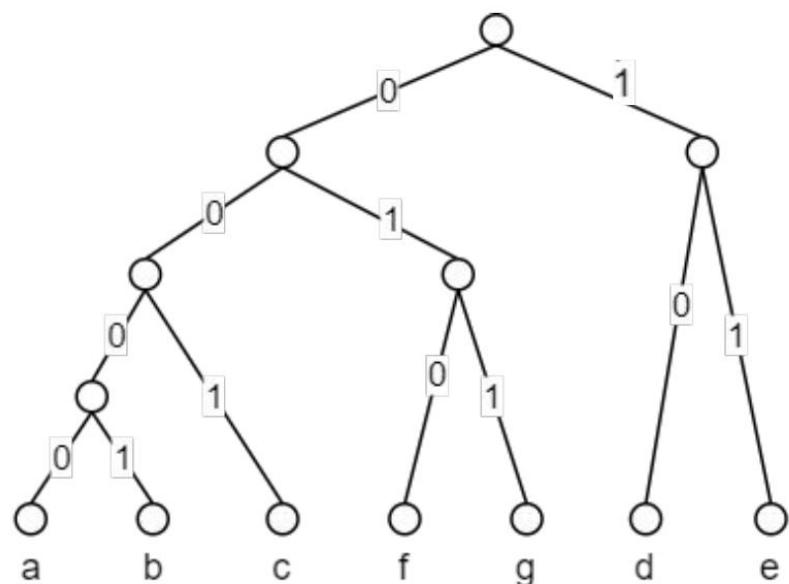
s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

num

symbol



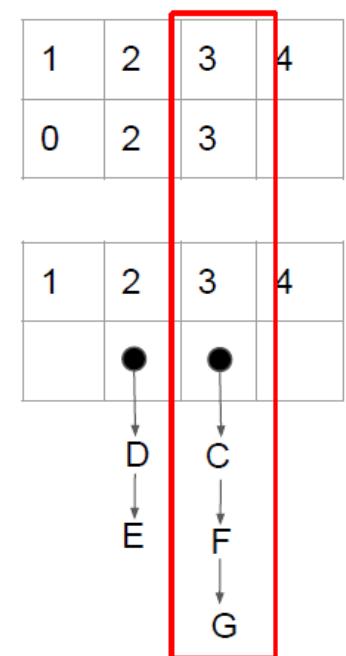
Canonical Huffman Encoding



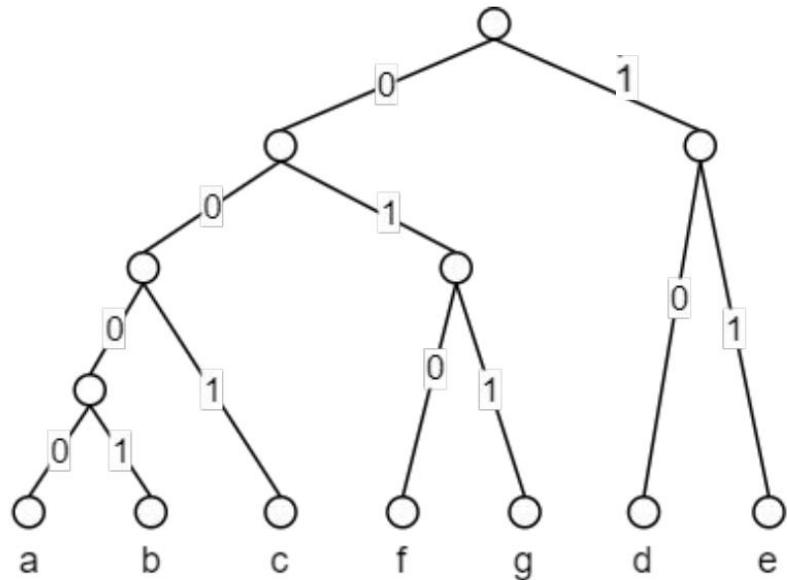
s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

num

symbol



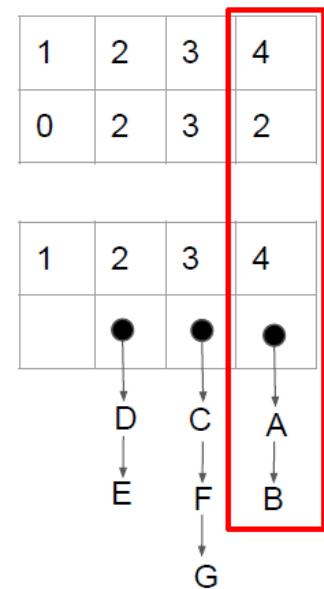
Canonical Huffman Encoding



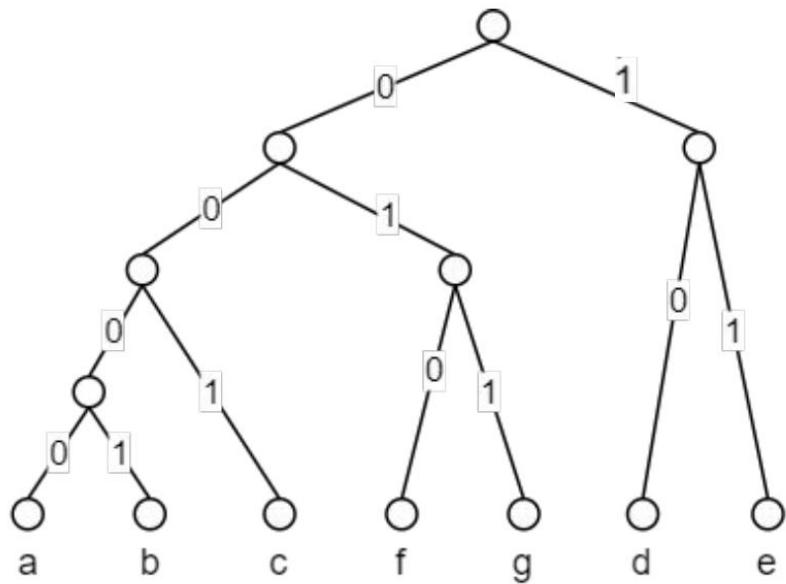
s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

num

symbol



Construction of the array firstcode

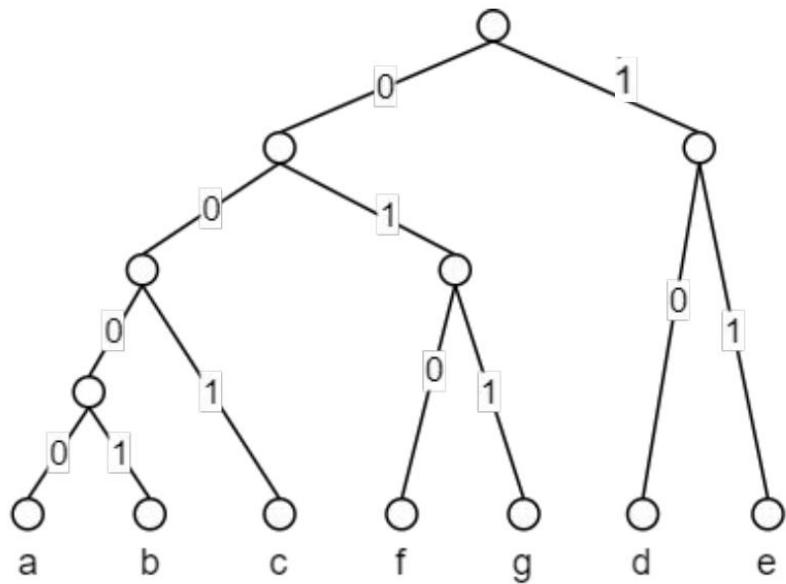


s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

1	2	3	4
			0

fc

Construction of the array firstcode

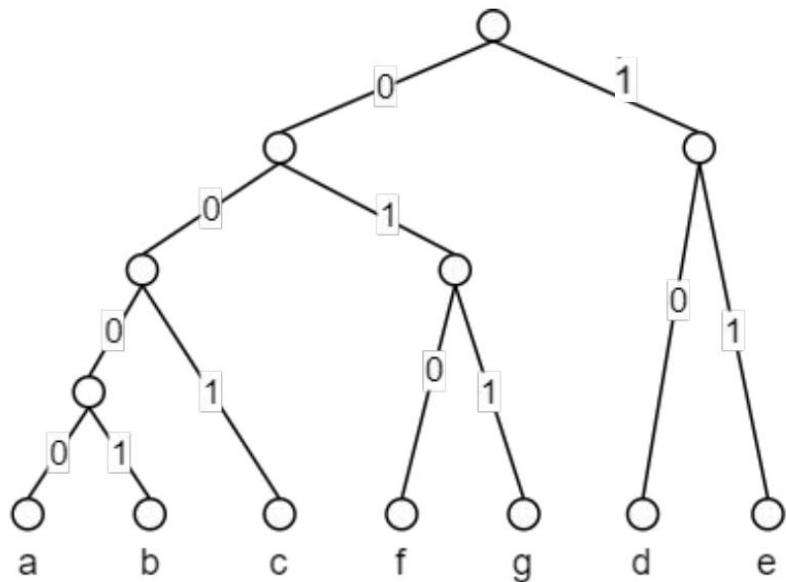


s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

1	2	3	4
		1	0

fc

Construction of the array firstcode

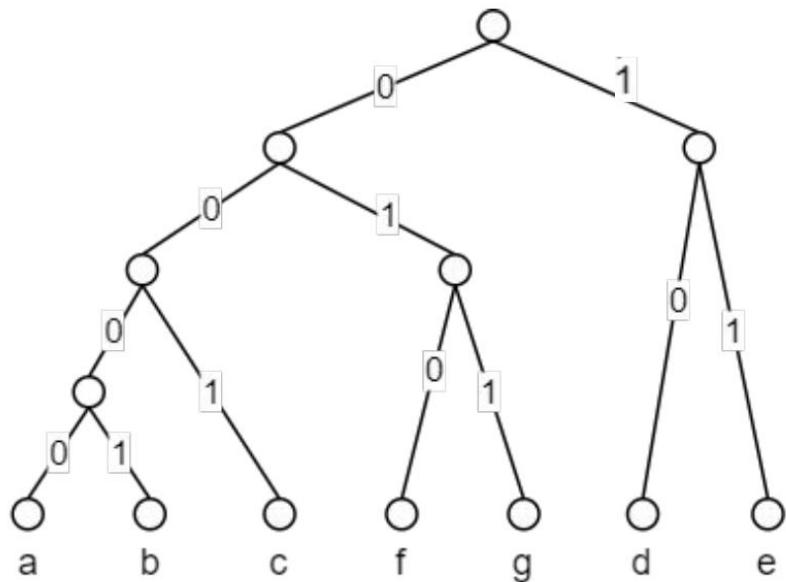


s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

1	2	3	4
2	1	0	

fc

Construction of the array firstcode



s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

fc

1	2	3	4
2	2	1	0

fc[1]=2 is a value used in the decoding step

Construction of the array firstcode

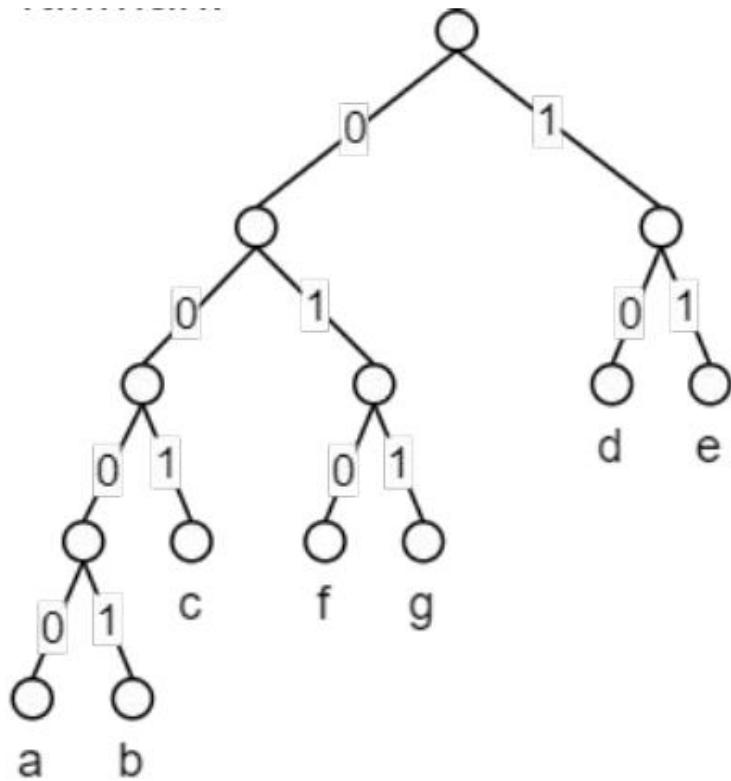
```
fc[max]=0;  
for(l= max-1; l>=1; l--)  
    fc[l]=(fc[l+1] + num[l+1])/2;
```

firstcode	0	0	0	0	0
cw	0	0	0	0	1
firstcode	0	0	0	1	

truncating here

- + *num* is the array counting the occurrences and *max* the maximal codeword length.
- + $fc[\max] = 0$ means that the codewords of maximal length start from “00..0”. The other codewords will be 00001, 00010, 00011 etc. This means that the tree will be unbalanced to the left
- + The pseudocode is reserving $num[\ell + 1]$ words, of length $\ell + 1$, to the symbols in $symb[\ell + 1]$ starting from the value $fc[\ell + 1]$. The first unused codeword of $\ell + 1$ bits is therefore given by the value $fc[\ell + 1] + num[\ell + 1]$.
- + Division by 2 corresponds to a shift to the right, i.e., we eliminate a bit when the word length decreases.
- + It can be proved that the resulting sequence of ℓ bits can be taken as the first codeword $fc[\ell]$ because it does not prefix any other codeword already assigned. The “reason” can be derived graphically by looking at the binary tree which is being built by Canonical Huffman. In fact, the algorithm is taking the parent of the node at depth $\ell + 1$, whose binary-path represents the value $fc[\ell + 1] + num[\ell + 1]$. Since the tree is a fully binary and we are allocating leaves in the tree from left to right, this node is always a left child of its parent, so its parent is not an ancestor of any $(\ell + 1)$ -bit codeword assigned before.
- + $fc[1]=2$ is an impossible codeword because we cannot encode 2 in 1 bit; this is used by the decoder

The tree we obtain is:



s_i	cw_i
a	0000
b	0001
c	001
f	010
g	011
d	10
e	11

Decoding

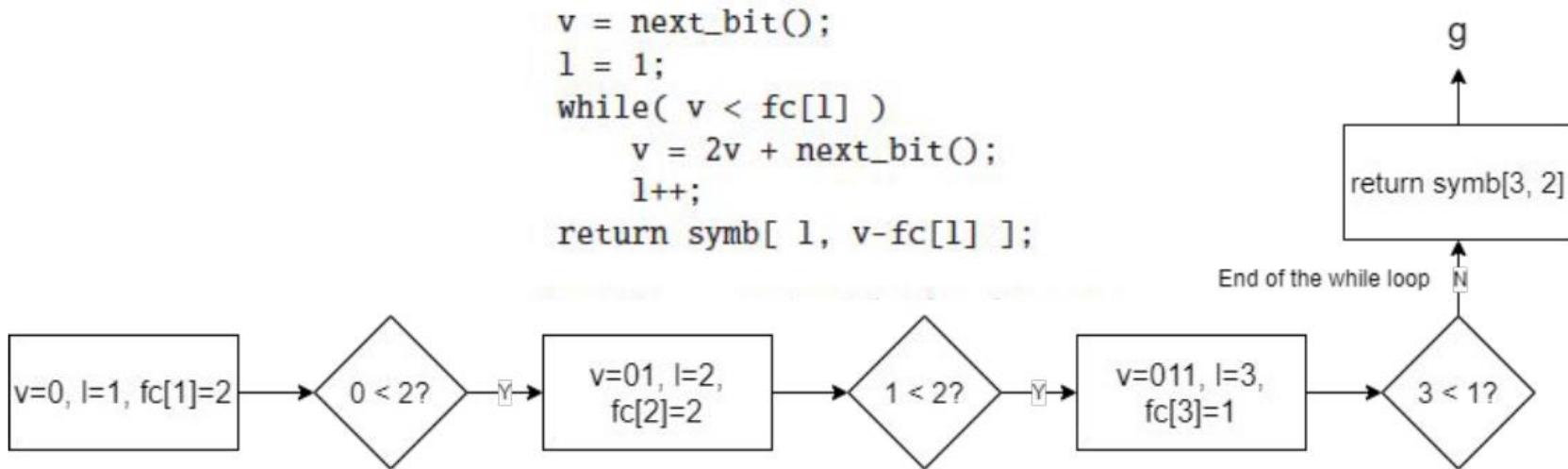
1. The function reads the incoming bits to be decoded
2. The length of the encoded word is computed
3. It returns the symbol corresponding to the codeword of $\ell - bits$

```
v = next_bit();
l = 1;
while( v < fc[1] )
    v = 2v + next_bit();
    l++;
return symb[ l, v-fc[1] ];
```

Note: the value $fc[1] = 2$ seems impossible, because we cannot represent the value 2 with a codeword consisting of one single bit. This is a special value because this way $fc[1]$ will be surely larger than any codeword of one bit, hence the Canonical Huffman algorithm will surely fetch another bit in the while-cycle.

Decoding

- + Example: decoding the message '011' by using $symb[(),(d,e),(c,f,g),(a,b)]$ ed $fc[2,2,1,0]$
- + **NB:** arrays are 1-based, the lists in symb are 0-based



Example

Symb.	length	code	bits	1. Compute array num <i>num</i> : [0 3 1 0 4]
i	l_i	word		2. Compute array <i>firstcode</i> <i>fc</i> : [2 1 1 2 0]
a	2	1	01	3. Compute codewords and symb
b	5	0	00000	
c	5	1	00001	
d	3	1	001	
e	2	2	10	
f	5	2	00010	
g	5	3	00011	
h	2	3	11	

	0	1	2	3
1	-	-	-	-
2	a	e	h	-
3	d	-	-	-
4	-	-	-	-
5	b	c	f	g

Example

```
v = next_bit();
l = 1;
while( v < fc[l] )
    v = 2v + next_bit();
    l++;
return symb[ l, v-fc[l] ];
```

00111100000001001

symbol

	o	1	2	3
1	-	-	-	-
2	a	e	h	-
3	d	-	-	-
4	-	-	-	-
5	b	c	f	g

fc: [2 1 1 2 0]

symbol[3,0] = d
symbol[2,2] = h
symbol[2,1] = e
symbol[5,0] = b
symbol[2,0] = a
symbol[3,0] = d

Decoded: dhebad

Remark

- + It is known that Huffman algorithm provides optimal prefix codes and that the entropy of the source is a lower bound for its average code length.
- + Clearly $H \geq 0$, and it is equal to zero whenever the source emits just one symbol with probability 1 and all the other symbols with probability 0. Moreover, it is also $H \leq \log_2 |A|$, and it is equal to this upper bound for equiprobable symbols.
- + Similarly to the Shannon code (Shannon Theorem), the Huffman code can lose up to 1 bit per compressed symbol with respect to the entropy of the source. It can be proved that if $H \gg 1$, the Huffman code is effective and the extra-bit is negligible;
- + On the other hand, Huffman, as any prefix-free code, cannot encode one symbol in less than 1 bit, so the best compression ratio that Huffman can obtain is $\geq 1/\log_2 |A|$. If A is ASCII, hence $|A|=256$, Huffman cannot achieve a compression ratio for any text which is less than $1/8 = 12,57\%$.

Lab Exercise

- + Compare the Huffman code and the canonical Huffman code for the source:

symb	A	D	E	I	O	R	T
prob	1/5	1/5	1/10	7/100	3/10	1/10	3/100

- + Write a program to compute the canonical Huffman coding.



Arithmetie coding (part 2)

Friday March 24, 2023

The key idea in arithmetic coding

- + Arithmetic coding was proposed by Peter Elias (another student of Fano together with Huffman) and first presented by Abramson in his book Information Theory and Coding (1963). Early implementations were developed by [Rissanen 76], [Pasco 76], and [Rubin 79]. [Moffat et al. 98] and [Witten et al. 87] discussed many theoretical and practical details of the encoding.
- + Arithmetic coding completely overcomes the idea of replacing an input symbol with a codeword.
- + Instead, it transforms a stream of input symbols into **a single floating-point number in [0,1]**
- + The longer and more complex the message, the more bits are required to represent the output number.

The key-idea

- + The output of the arithmetic coding algorithm is, as usual, a stream of bits
- + However, it can be thought of as if there were an initial bit 0, and the stream represents a fractional number in binary, between 0 and 1

01101010 → 0.01101010

Bit streams and binary representations of a fractional number

- + A bit stream $b_1 b_2 b_3 \dots b_k$ can be interpreted as a real number in the range $[0,1)$ by prepending «0,» to it, i.e.:

$$0, b_1 b_2 \dots b_k = \sum_{i=1}^k b_i 2^{-i}$$

- + Any real number x in the range $[0,1)$ can be converted in a possibly infinite sequence of bits by using the following algorithm. The loop has to end when a level of accuracy is defined:

Converter(x)

Require: A real number $x \in [0, 1)$.

Ensure: The string of bits representing x .

```
1: repeat
2:    $x = 2 * x$ 
3:   if  $x < 1$  then
4:     output = output :: 0
5:   else
6:     output = output :: 1
7:      $x = x - 1$ 
8:   end if
9: until accuracy
```

:: denotes concatenations among bits

Example:

How to encode 0.825

- + $0,825 \times 2 = 1,65 \quad U=1$
- + $0,65 \times 2 = 1,3 \quad U=1$
- + $0,3 \times 2 = 0,6 \quad U=0$
- + $0,6 \times 2 = 1,2 \quad U=1$
- + $0,2 \times 2 = 0,4 \quad U=0$
- + $0,4 \times 2 = 0,8 \quad U=0$
- + $0,8 \times 2 = 1,6 \quad U=1$
- + $0,6 \times 2 = 1,2 \quad U=1$
- +

When we find a value we have seen before,
then we can stop the loop
and the representation is periodic

Ex: $0,11010011001\dots = 0,1\overline{101001}$

Exercise:

- + Finding the binary representation of $1/3$:
 - + Solution: $\overline{01}$

- + Finding the binary representation of $3/32$:
 - + Solution: $00011\bar{0}$

Encoding algorithm: how does it work?

INPUT: String to be encoded & alphabet with probability distribution (probabilities are calculated using frequencies, or another source)

1. Initialize the range $[0,1]$
2. For each symbol of the input string:
 1. Divide the current interval into n subintervals ($n =$ cardinality of the alphabet). Each subinterval will have a size proportional to the probability associated with each element in the initial alphabet.
 2. Select as the current interval the one corresponding to the symbol being analyzed (input)
3. The lower bound of the last interval (converted to binary) is the result of our encoding (output)

Example

+ Let us encode the string **BILL GATES**

We need the frequencies of the characters in the text

chr	freq.
space	0.1
A	0.1
B	0.1
E	0.1
G	0.1
I	0.1
L	0.2
S	0.1
T	0.1

Note: To explain the algorithm with the example, the numbers we show will be in base 10, but of course in practice they are always in base 2

We partition the interval according to the probabilities

character	probability	range
space	0.1	[0.00, 0.10)
A	0.1	[0.10, 0.20)
B	0.1	[0.20, 0.30)
E	0.1	[0.30, 0.40)
G	0.1	[0.40, 0.50)
I	0.1	[0.50, 0.60)
L	0.2	[0.60, 0.80)
S	0.1	[0.80, 0.90)
T	0.1	[0.90, 1.00)

Encoding the text

chr	low	high
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
<i>Space</i>	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

The final value 0.2572167752 is the encoding of the string **BILL GATES**

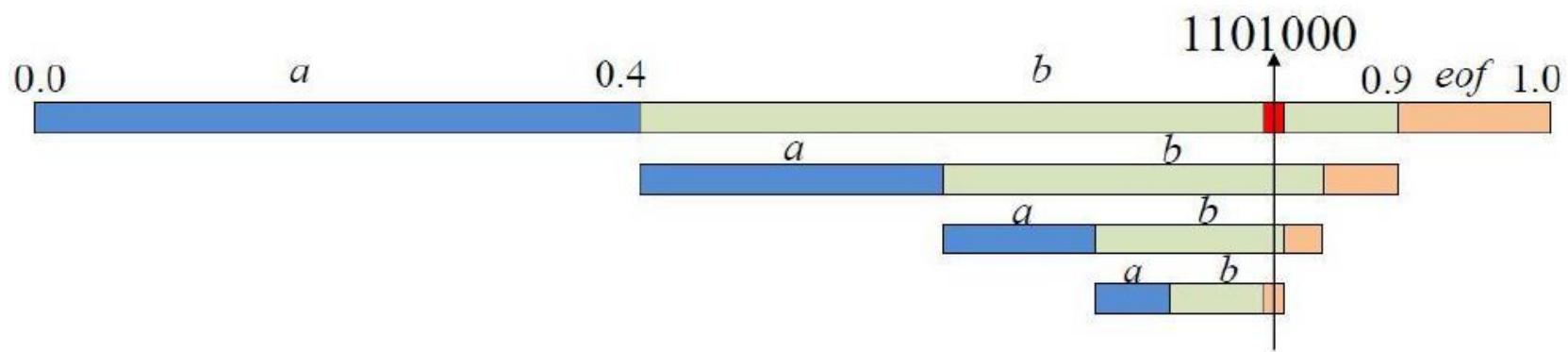
Another example

A	B	C
0.4	0.5	0.1

+ Input: BBBC

Intervallo Corrente	A	B	C	Input	Azione
[0,1)	[0, 0.4)	[0.4, 0.9)	[0.9, 1)	B	Dividi
[0.4, 0.9)	[0.4, 0.6)	[0.6, 0.85)	[0.85, 0.9)	B	Dividi
[0.6, 0.85)	[0.6, 0.7)	[0.7, 0.825)	[0.825, 0.85)	B	Dividi
[0.7, 0.825)	[0.7, 0.75)	[0.75, 0.812)	[0.812, 0.825)	C	Stop
[0.812, 0.825)					

...continued...



- + The final range is $[0.812, 0.825)$ and its binary representation is $[0.1101000, 0.1101001)$
- + The final range is identified by its lower bound. The sequence 1101000 (7 bits) is transmitted

How to proceed?

- + Finding the arithmetic coding of the string SWISS_MISS and the following statistical model (the same model will be used at each step of the algorithm):

Char	Freq	Prob.	Range	CumFreq
		Total CumFreq =		10
S	5	$5/10 = 0.5$	[0.5, 1.0)	5
W	1	$1/10 = 0.1$	[0.4, 0.5)	4
I	2	$2/10 = 0.2$	[0.2, 0.4)	2
M	1	$1/10 = 0.1$	[0.1, 0.2)	1
	1	$1/10 = 0.1$	[0.0, 0.1)	0

- + The cumulative frequency is used to decode the message
- + The symbols and the frequencies must be sent before the compressed text
- + The encoding process needs two variables, Low e High, initialized to 0 and 1 (initial interval)

How to proceed?

- + Low and High are updated at each step as follows:

$\text{NewHigh} := \text{OldLow} + \text{Range} * \text{HighRange}(X);$

$\text{NewLow} := \text{OldLow} + \text{Range} * \text{LowRange}(X);$

where

$\text{Range} = \text{OldHigh} - \text{OldLow}$

$\text{LowRange}(X)$ and $\text{HighRange}(X)$ are the lower bound and the upper bound of the interval corresponding to X , respectively.

The encoding process

Char.		The calculation of low and high
S	L	$0.0 + (1.0 - 0.0) \times 0.5 = 0.5$
	H	$0.0 + (1.0 - 0.0) \times 1.0 = 1.0$
W	L	$0.5 + (1.0 - 0.5) \times 0.4 = 0.70$
	H	$0.5 + (1.0 - 0.5) \times 0.5 = 0.75$
I	L	$0.7 + (0.75 - 0.70) \times 0.2 = 0.71$
	H	$0.7 + (0.75 - 0.70) \times 0.4 = 0.72$
S	L	$0.71 + (0.72 - 0.71) \times 0.5 = 0.715$
	H	$0.71 + (0.72 - 0.71) \times 1.0 = 0.72$
S	L	$0.715 + (0.72 - 0.715) \times 0.5 = 0.7175$
	H	$0.715 + (0.72 - 0.715) \times 1.0 = 0.72$
□	L	$0.7175 + (0.72 - 0.7175) \times 0.0 = 0.7175$
	H	$0.7175 + (0.72 - 0.7175) \times 0.1 = 0.71775$
M	L	$0.7175 + (0.71775 - 0.7175) \times 0.1 = 0.717525$
	H	$0.7175 + (0.71775 - 0.7175) \times 0.2 = 0.717550$
I	L	$0.717525 + (0.71755 - 0.717525) \times 0.2 = 0.717530$
	H	$0.717525 + (0.71755 - 0.717525) \times 0.4 = 0.717535$
S	L	$0.717530 + (0.717535 - 0.717530) \times 0.5 = 0.7175325$
	H	$0.717530 + (0.717535 - 0.717530) \times 1.0 = 0.717535$
S	L	$0.7175325 + (0.717535 - 0.7175325) \times 0.5 = 0.71753375$
	H	$0.7175325 + (0.717535 - 0.7175325) \times 1.0 = 0.717535$

+ We produce as output 71353375

AC-Coding (S)

Require: The input sequence S , of length n , the probabilities $P[\sigma]$ and the cumulative f_σ .

Ensure: A subinterval $[l, l + s)$ of $[0, 1)$.

```
1:  $s_0 = 1$ 
2:  $l_0 = 0$ 
3:  $i = 1$ 
4: while  $i \leq n$  do
5:    $s_i = s_{i-1} * P[S[i]]$ 
6:    $l_i = l_{i-1} + s_{i-1} * f_{S[i]}$ 
7:    $i = i + 1$ 
8: end while
9:  $output = \langle x \in [l_n, l_n + s_n), n \rangle$ 
```

Example

Example

- + Let us consider the sequence $S=abac$ with probabilities $p(a)=1/2$, $p(b)=p(c)=1/4$. The cumulative probabilities are $f_c = \sum_{i < c} p(i)$

$$f_a = 0, f_b = p(a) = \frac{1}{2}, f_c = p(a) + p(b) = \frac{3}{4}$$

- + The loop is repeated 4 times and the intervals are:

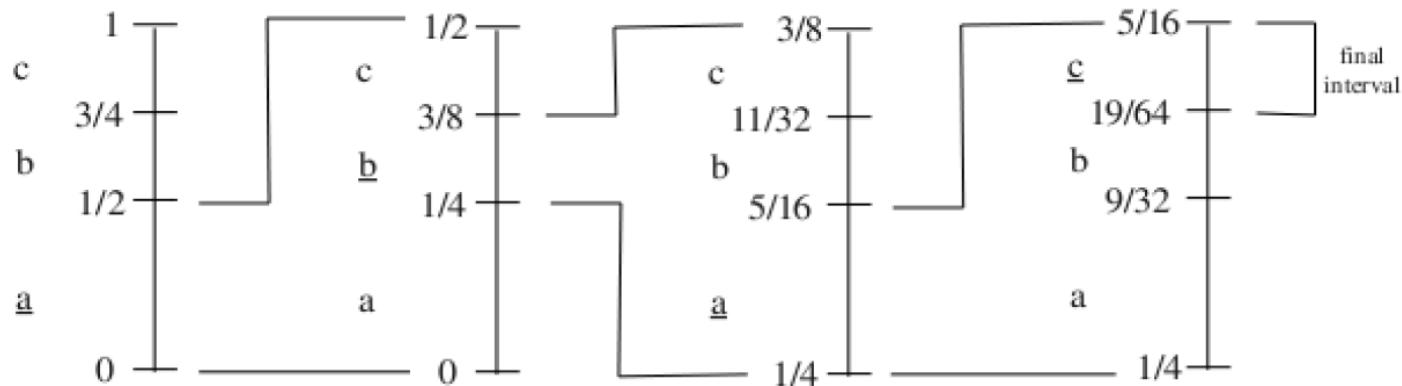
- + $[l_1, l_1 + s_1) = \left[0, \frac{1}{2}\right)$
- + $[l_2, l_2 + s_2) = \left[\frac{1}{4}, \frac{1}{4} + \frac{1}{8}\right)$
- + ...
- + ...

Example

- + Let us consider the sequence $S=abac$ with probabilities $p(a)=1/2$, $p(b)=p(c)=1/4$.
- + The final interval is $[19/64, 5/16)$

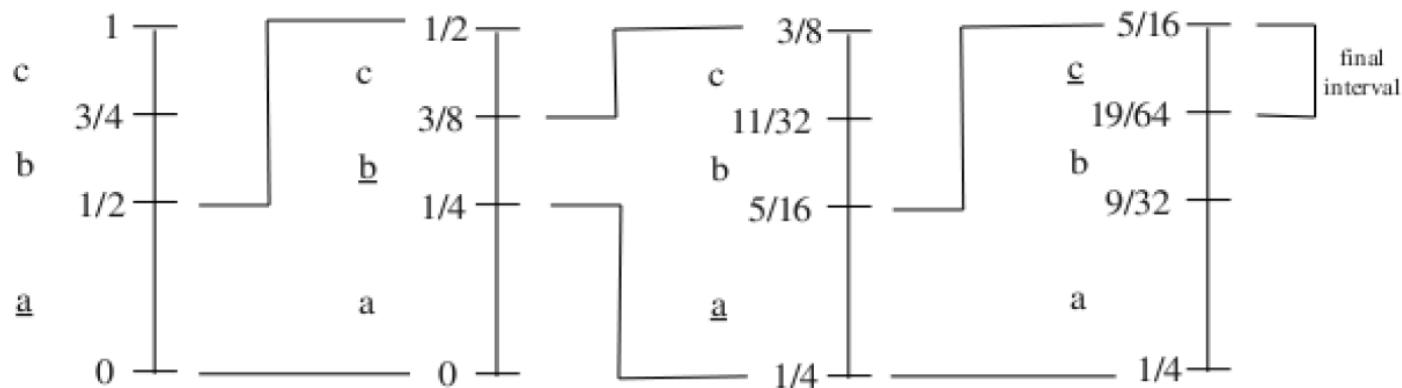
Example

- + Let us consider the sequence $S=abac$ with probabilities $p(a)=1/2$, $p(b)=p(c)=1/4$.
- + The final interval is $[19/64, 5/16)$



Example

- + Let us consider the sequence $S=abac$ with probabilities $p(a)=1/2$, $p(b)=p(c)=1/4$.
- + The final interval is $[19/64, 5/16)$



Decoding algorithm: how does it work?

- + INPUT: String to be decoded & alphabet with probability distribution
 - 1. Initialize the range $[0,1)$
 - 2. For each symbol of the string:
 - 1. Divide the current interval into n subintervals ($n = \text{size of the alphabet}$). Each subinterval will have a size proportional to the probability associated with each element in the initial alphabet.
 - 2. Select as the current interval the symbol corresponding to the subinterval in which the value of the input being analyzed falls.
- + It is a lossless algorithm (without loss of information).

Decoding: example

- + Let us suppose the following alphabet

A	B	C
0.4	0.5	0.1

- + Input: $1101000 = 0.8_{12}$

Intervallo Corrente	A	B	C	Input	Output
[0,1)	[0, 0.4)	[0.4, 0.9)	[0.9, 1)	0.8	B
[0.4, 0.9)	[0.4, 0.6)	[0.6, 0.85)	[0.85, 0.9)	0.81	B
[0.6, 0.85)	[0.6, 0.7)	[0.7, 0.825)	[0.825, 0.85)	0.812	B
[0.7, 0.825)	[0.7, 0.75)	[0.75, 0.812)	[0.812, 0.825)	0.812	C

- + Output: BBCB

How does it work?

- + We start from a static model and a string, for instance
71353375

Char	Freq	Prob.	Range	CumFreq
		Total CumFreq =		10
S	5	$5/10 = 0.5$	[0.5, 1.0)	5
W	1	$1/10 = 0.1$	[0.4, 0.5)	4
I	2	$2/10 = 0.2$	[0.2, 0.4)	2
M	1	$1/10 = 0.1$	[0.1, 0.2)	1
□	1	$1/10 = 0.1$	[0.0, 0.1)	0

Char	Freq	Prob.	Range	CumFreq
		Total	CumFreq=	10
S	5	$5/10 = 0.5$	[0.5, 1.0)	5
W	1	$1/10 = 0.1$	[0.4, 0.5)	4
I	2	$2/10 = 0.2$	[0.2, 0.4)	2
M	1	$1/10 = 0.1$	[0.1, 0.2)	1
□	1	$1/10 = 0.1$	[0.0, 0.1)	0

The decoding process

- + We compute Low and High of each interval in which lies the value $\text{Code} := (\text{Code}-\text{LowRange}(X))/\text{Range}$ where Range is the length of the interval in which X lies

Char.	Code–low	Range
S	$0.71753375 - 0.5 = 0.21753375/0.5 = 0.4350675$	
W	$0.4350675 - 0.4 = 0.0350675 /0.1 = 0.350675$	
I	$0.350675 - 0.2 = 0.150675 /0.2 = 0.753375$	
S	$0.753375 - 0.5 = 0.253375 /0.5 = 0.50675$	
S	$0.50675 - 0.5 = 0.00675 /0.5 = 0.0135$	
□	$0.0135 - 0 = 0.0135 /0.1 = 0.135$	
M	$0.135 - 0.1 = 0.035 /0.1 = 0.35$	
I	$0.35 - 0.2 = 0.15 /0.2 = 0.75$	
S	$0.75 - 0.5 = 0.25 /0.5 = 0.5$	
S	$0.5 - 0.5 = 0 /0.5 = 0$	

- + It ends when o is reached.

Some remarks

- + Arithmetic coding is better than Huffman when we have asymmetrical distributions, since it can be closer to the entropy
- + Example: Let us consider the source S with the alphabet A={a,b} and $p(a)=99/100$ and $p(b)=1/100$. In this case,

$$H(S) = p(a)\log_2 \frac{100}{99} + p(b) \log_2 \frac{100}{1} \cong 0,08056$$

- + Any prefix code (and an Huffman code, as well) must at least use one bit per symbol, so the average length of the code must be at least 1, which is much greater than the entropy!
- + In arithmetic coding, binary encoding of a fractional number is used.

If the probabilities are very unbalanced...

- + Let us encode $a_2 a_2 a_1 a_3 a_3$

Char	Prob.	Range
a_1	0.001838	[0.998162, 1.0)
a_2	0.975	[0.023162, 0.998162)
a_3	0.023162	[0.0, 0.023162)

(a)

- + The encoding process

a_2	$0.0 + (1.0 - 0.0) \times 0.023162 = 0.023162$
a_2	$0.0 + (1.0 - 0.0) \times 0.998162 = 0.998162$
a_2	$0.023162 + .975 \times 0.023162 = 0.04574495$
a_1	$0.023162 + .975 \times 0.998162 = 0.99636995$
a_1	$0.04574495 + 0.950625 \times 0.998162 = 0.99462270125$
a_1	$0.04574495 + 0.950625 \times 1.0 = 0.99636995$
a_3	$0.99462270125 + 0.00174724875 \times 0.0 = 0.99462270125$
a_3	$0.99462270125 + 0.00174724875 \times 0.023162 = 0.994663171025547$
a_3	$0.99462270125 + 0.00004046977554749998 \times 0.0 = 0.99462270125$
a_3	$0.99462270125 + 0.00004046977554749998 \times 0.023162 = 0.994623638610941$

- + The decoding process (the end of decoding is not recognized)

Char.	Code-low	Range
a_2	$0.99462270125 - 0.023162 = 0.97146170125 / 0.975 = 0.99636995$	
a_2	$0.99636995 - 0.023162 = 0.97320795 / 0.975 = 0.998162$	
a_1	$0.998162 - 0.998162 = 0.0 / 0.00138 = 0.0$	
a_3	$0.0 - 0.0 = 0.0 / 0.023162 = 0.0$	
a_3	$0.0 - 0.0 = 0.0 / 0.023162 = 0.0$	

If the probabilities are very unbalanced...

- + A symbol eof is appended.

Char	Prob.	Range
a_1	0.001838	[0.998162, 1.0)
a_2	0.975	[0.023162, 0.998162)
a_3	0.023162	[0.0, 0.023162)

(a)

Char	Prob.	Range
eof	0.000001	[0.999999, 1.0)
a_1	0.001837	[0.998162, 0.999999)
a_2	0.975	[0.023162, 0.998162)
a_3	0.023162	[0.0, 0.023162)

(b)

- + Encoding of $a_3 a_3 a_3 a_3$ eof.

$$\begin{aligned}
 a_3 & \quad 0.0 + (1.0 - 0.0) \times 0.0 = 0.0 \\
 a_3 & \quad 0.0 + (1.0 - 0.0) \times 0.023162 = 0.023162 \\
 a_3 & \quad 0.0 + .023162 \times 0.0 = 0.0 \\
 a_3 & \quad 0.0 + .023162 \times 0.023162 = 0.000536478244 \\
 a_3 & \quad 0.0 + 0.000536478244 \times 0.0 = 0.0 \\
 a_3 & \quad 0.0 + 0.000536478244 \times 0.023162 = 0.000012425909087528 \\
 a_3 & \quad 0.0 + 0.000012425909087528 \times 0.0 = 0.0 \\
 \text{eof} & \quad 0.0 + 0.0000002878089062853235 \times 0.999999 = 0.0000002878086184764172 \\
 & \quad 0.0 + 0.0000002878089062853235 \times 1.0 = 0.0000002878089062853235
 \end{aligned}$$

- + Decoding

Char.	Code—low	Range
a_3	0.0000002878086184764172-0	=0.0000002878086184764172 /0.023162=0.00001242589666161891247
a_3	0.00001242589666161891247-0	=0.00001242589666161891247/0.023162=0.000536477707521756
a_3	0.000536477707521756-0	=0.000536477707521756 /0.023162=0.023161976838
a_3	0.023161976838-0.0	=0.023161976838 /0.023162=0.999999
eof	0.999999-0.999999	=0.0 /0.000001=0.0

Remarks

- + If the input size is known, it is not necessary to encode eof.
- + In that case, the size of the input must be transmitted to the decoder.

Arithmetic Coding: optimality

- + Lemma: Take a real number $x = 0.b_1b_2\dots$. If we truncate it to the first d bits, we obtain a real number
$$trunc_d(x) \in [x - 2^{-d}, x].$$
- + Proof: The real number differs from its truncation, possibly, on the bits that follow the position d . Therefore
$$x - trunc_d(x) = \sum_{i=1}^{\infty} b_{d+i} 2^{-(d+i)} \leq \sum_{i=1}^{\infty} 1 \cdot 2^{-(d+i)} = 2^{-d} \sum_{i=1}^{\infty} 2^{-i} = 2^{-d}$$
- + Corollary: The truncation of an element of the interval $[l, l+s)$, for instance $l+s/2$ to the first $\lceil \log_s^2 \rceil$ bits falls in the interval $[l, l+s)$
- + Proof: It is enough to set $d = \lceil \log_s^2 \rceil$ and apply the lemma

Arithmetic coding: optimality

- + Theorem: The number of bits emitted by AC for a sequence of length n is at most $2 + nH(S)$, where $H(S)$ is the entropy of the source.
- + Proof: By using the previous corollary, we know that the number of bits in output is

$$\left\lceil \log\left(\frac{2}{s_n}\right) \right\rceil < 2 - \log s_n = 2 - \log \prod_{i=1}^n p(S[i]) = 2 - \sum_{i=1}^n \log p(S[i])$$

- + If we assume n sufficiently large, we can estimate $p(x) = n_x/n$ where n_x is the number of occurrence of x in S . So, we obtain

$$2 - \sum_{\sigma \in A} n_\sigma \log p(\sigma) = 2 - n \left(\sum_{\sigma \in A} p(\sigma) \log p(\sigma) \right) = 2 - nH(S)$$

Some remarks

- + there is a waste of only two bits on an entire input sequence S , hence $2/n$ bits per symbol. This is a vanishing lost as the input sequence becomes longer and longer.
- + The size of the output is a function of the set of symbols constituting S with their multiplicities, but not of their order.
- + Huffman coding requires $n+nH(S)$ bits for compressing a sequence of n symbols, so Arithmetic Coding is much better.
- + As we are going to see, it calculates the representation on the fly thus it can easily accommodate the use of dynamic modeling.
- + On the other hand, it must be said that (Canonical) Huffman is faster and can decompress any portion of the compressed file provided that we known its first codeword. This is however impossible for Arithmetic Coding which allows only the whole decompression of the compressed file.

We have seen that

- + A number in a smaller interval needs more bits to be encoded
- + More likely symbols do not greatly decrease the size of the interval, while less likely symbols lead to much smaller intervals
- + It is shown that the number of bits required for encoding an interval of size s is proportional to $-\log_2 s$
- + The length of the final interval is equal to the product of the probabilities of the individual input symbols

Almost optimal encoding

- + We have that $-\log s \sim -\sum_{i=1}^m p_i \log p_i$
- + The number of bits generated by arithmetic encoding (which is an approximation of $-\log s$) is almost equal to entropy.
- + For this reason, arithmetic encoding is almost optimal as the number of output bits, and is able to encode very probable events in just a fraction of a bit!
- + In practice, the algorithm is not exactly optimal due to the use of limited-precision arithmetic (overflow and underflow problems), and because the final encoding must still be done by an integer number of bits which is an integer.

Dynamic model. It is possible to re-compute the probability distribution at each step.

Input: String **bccb** with the alphabet source $\{a,b,c\}$

- + Let us suppose that all the probabilities are $1/3$.
- + When we encode **b** all the symbols have the same probability.
- + The arithmetic encoding maintains two numbers, *low* and *high*, representing the interval $[low, high)$ subset of $[0, 1)$
- + Firstly *low*=0 and *high*=1

Example

- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.

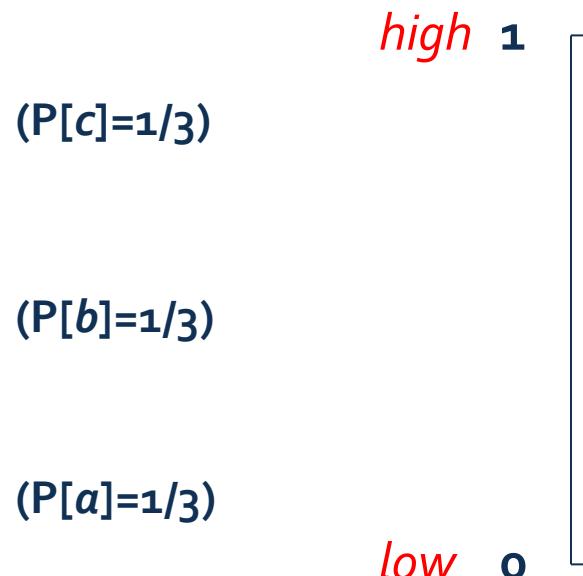
Example

- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.



Example

- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.



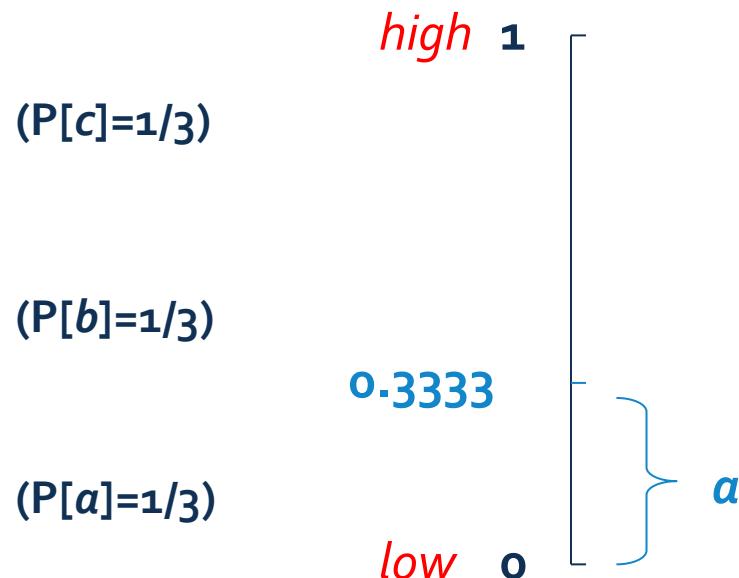
Example

- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.



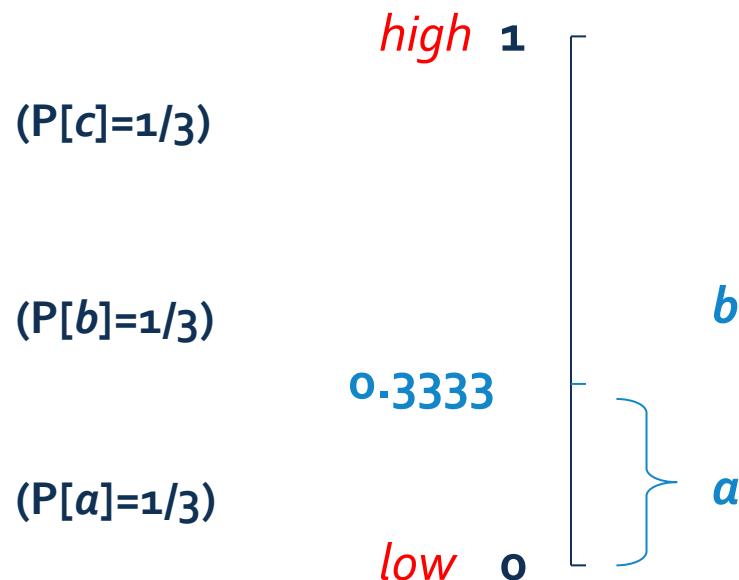
Example

- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.



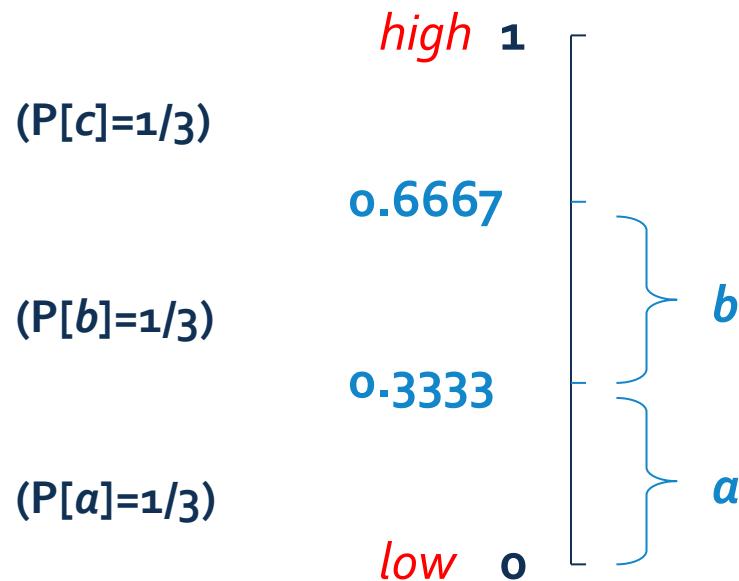
Example

- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.



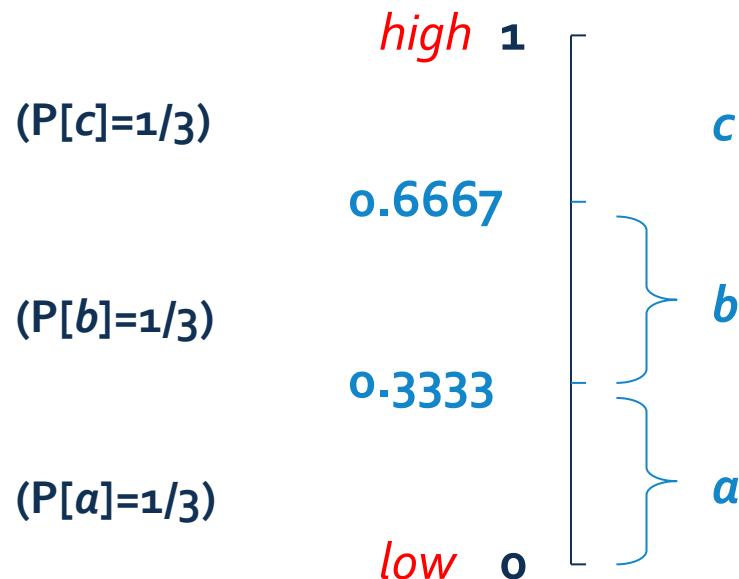
Example

- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.



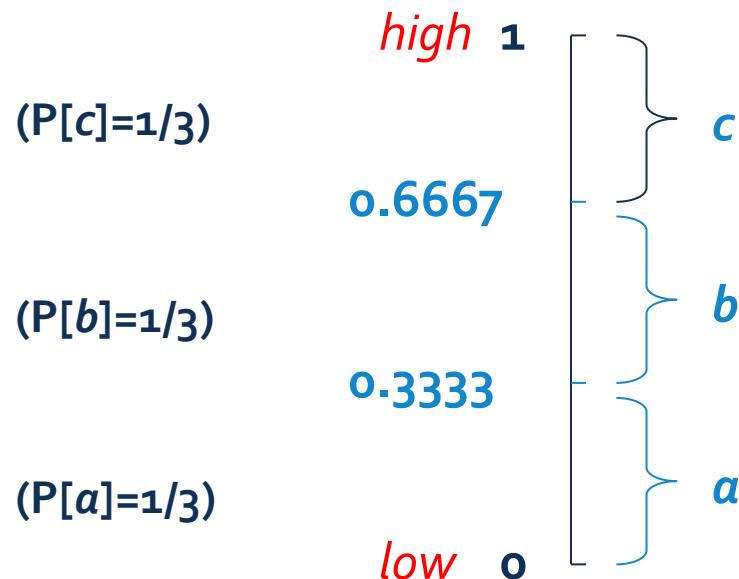
Example

- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.

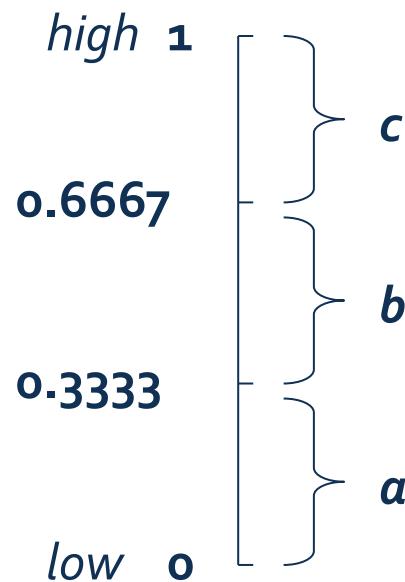


Example

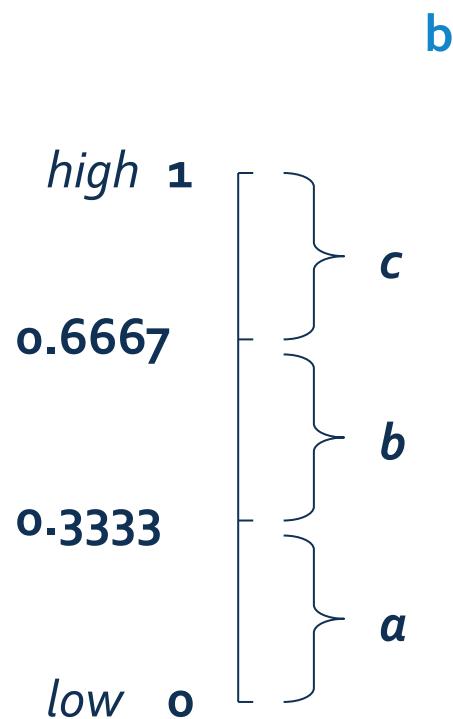
- + The range from *low* to *high* is divided among the symbols of the alphabet according to their probability.



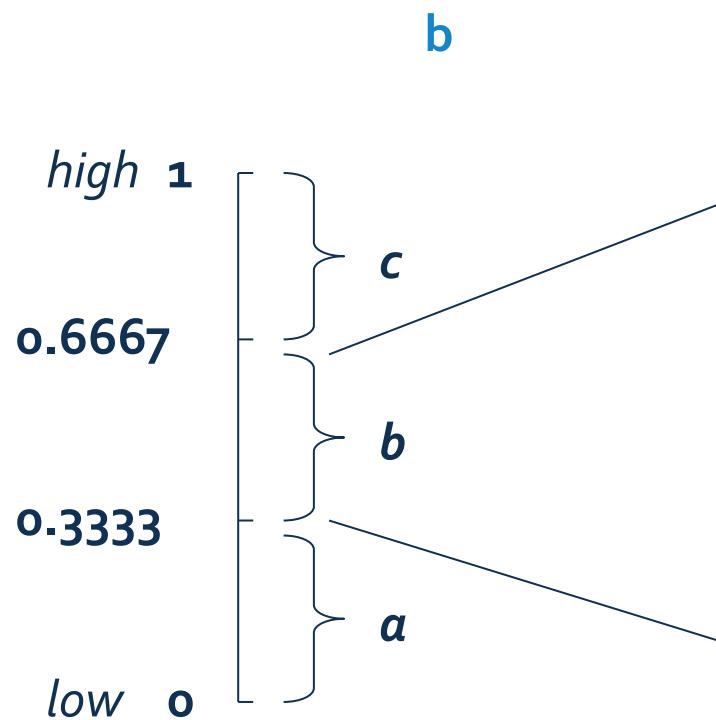
...continued... (the probabilities can be re-computed at each step)



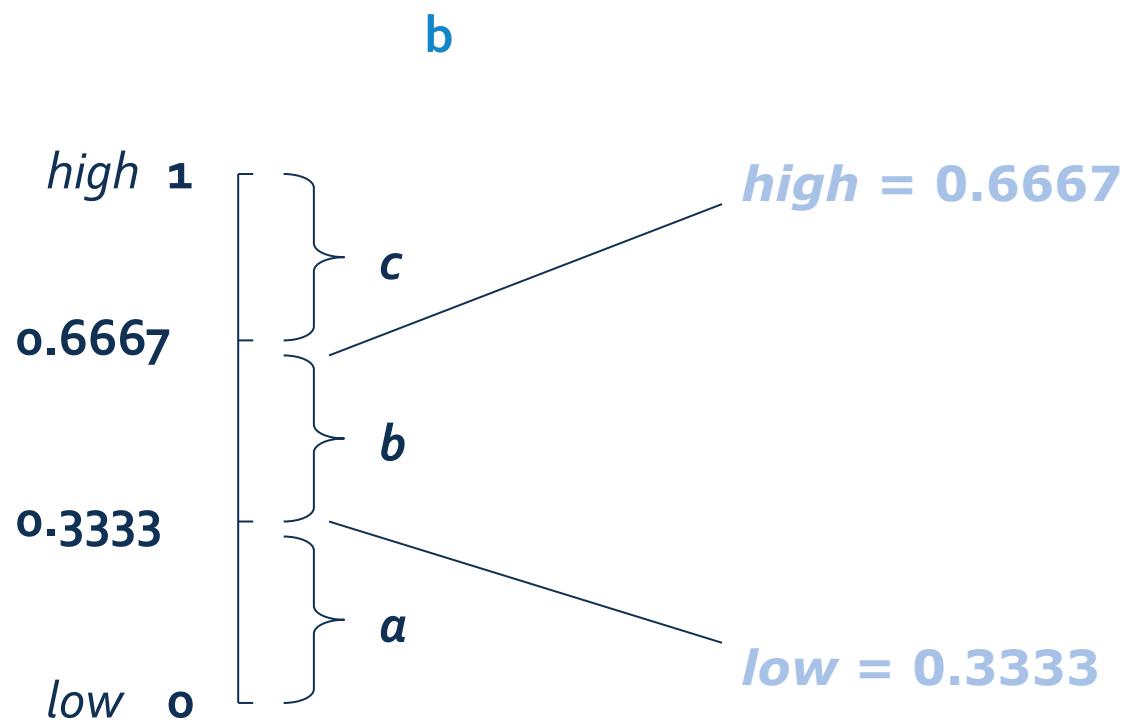
...continued... (the probabilities can be re-computed at each step)



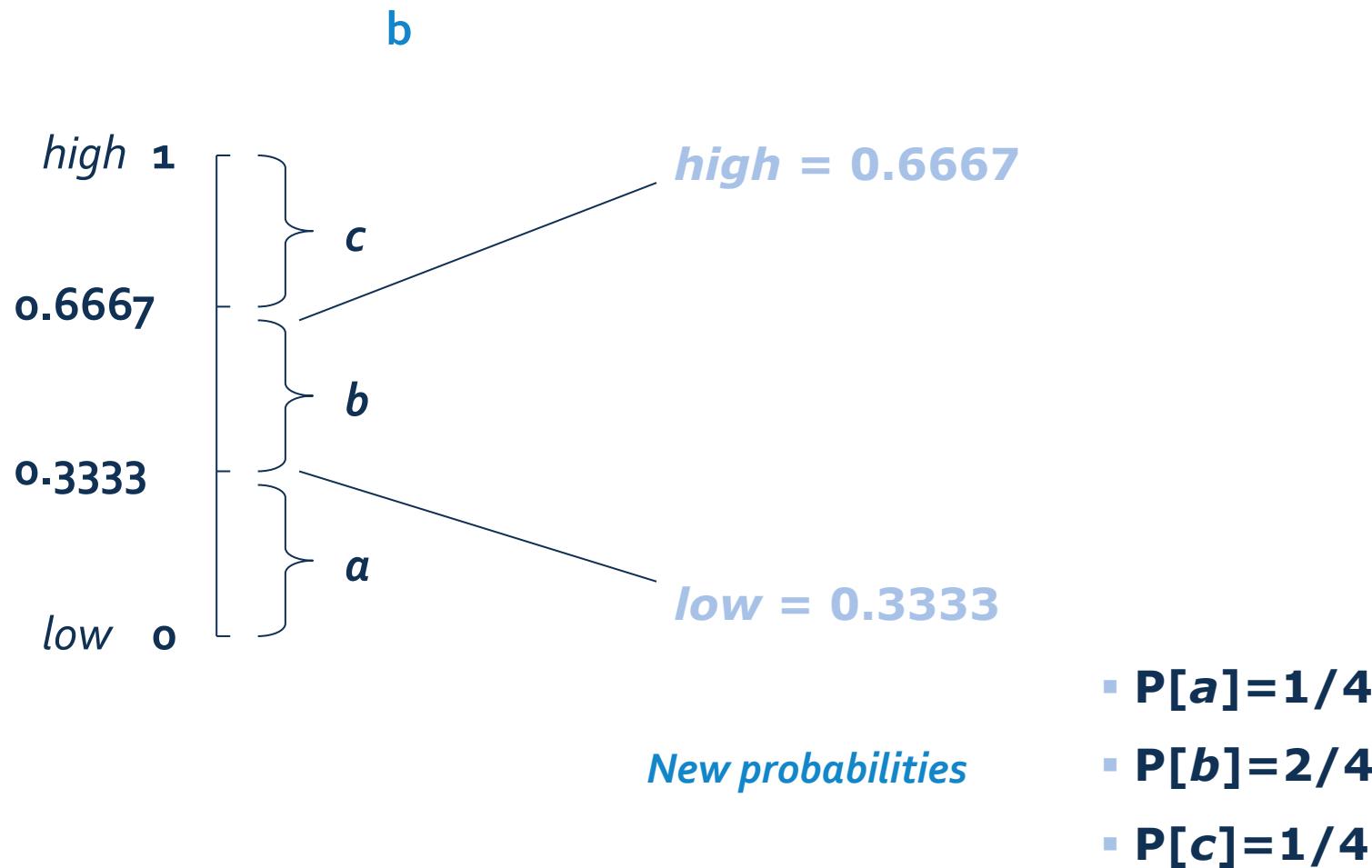
...continued... (the probabilities can be re-computed at each step)



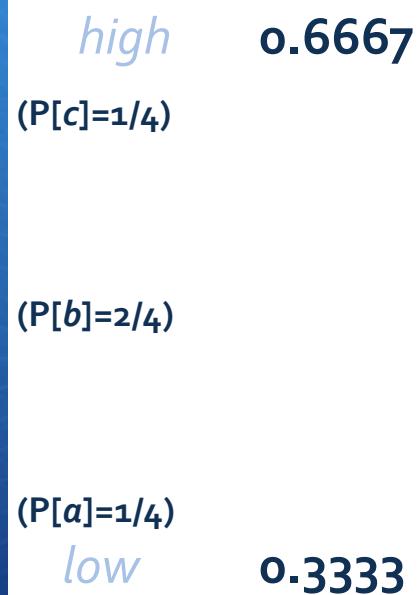
...continued... (the probabilities can be re-computed at each step)



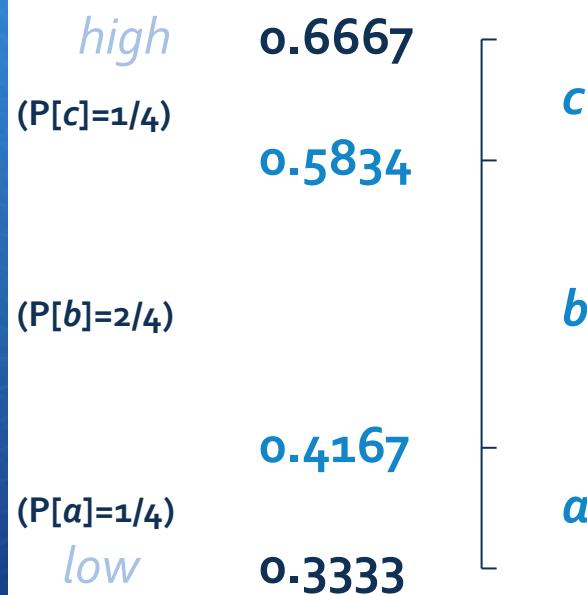
...continued... (the probabilities can be re-computed at each step)



...continued...



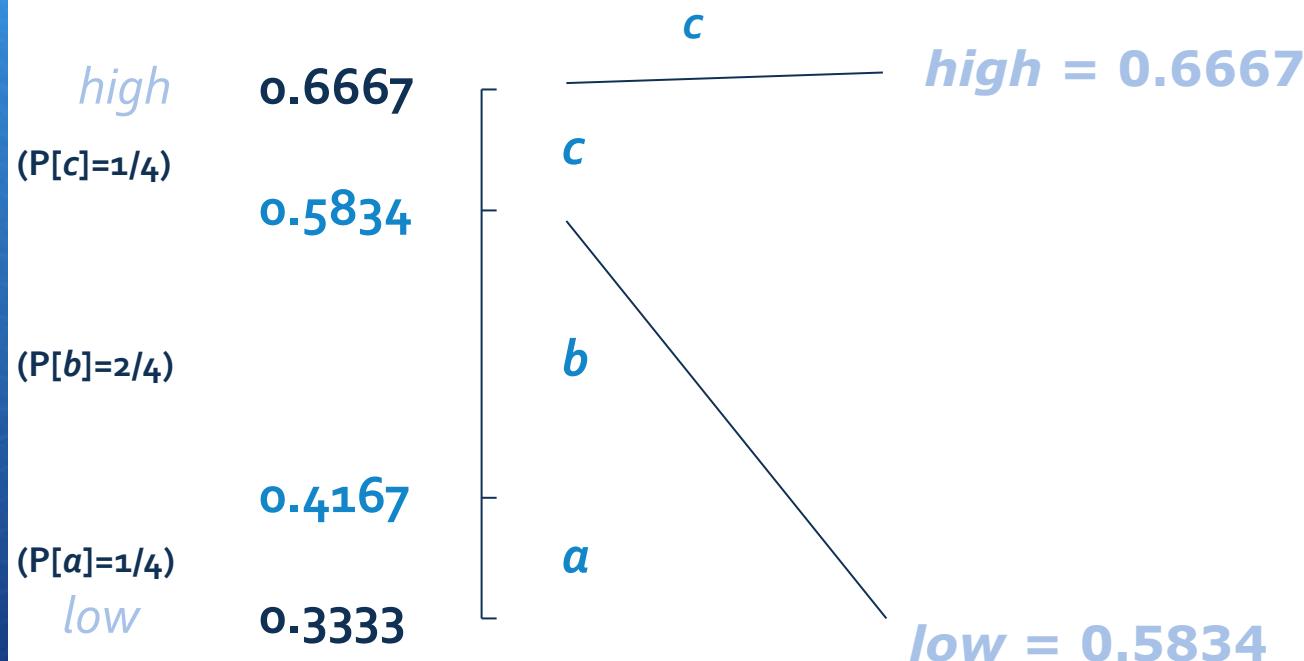
...continued...



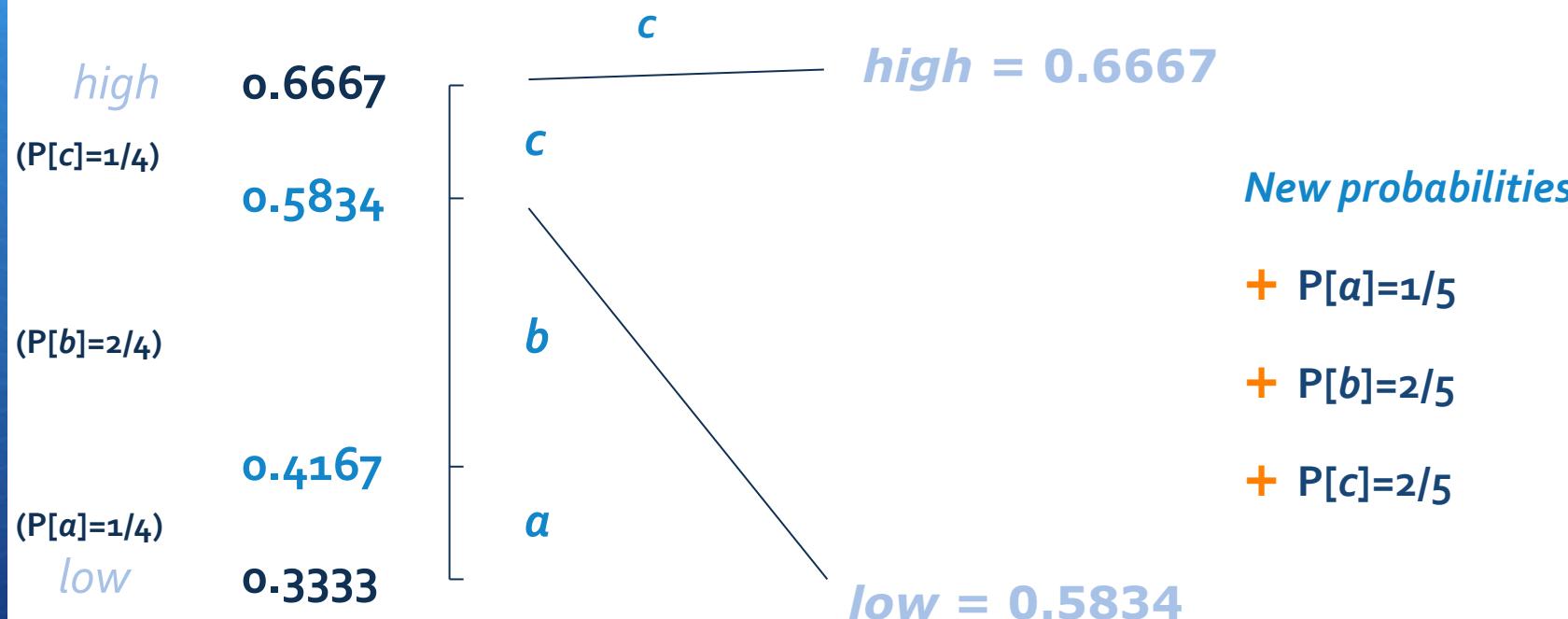
...continued...



...continued...



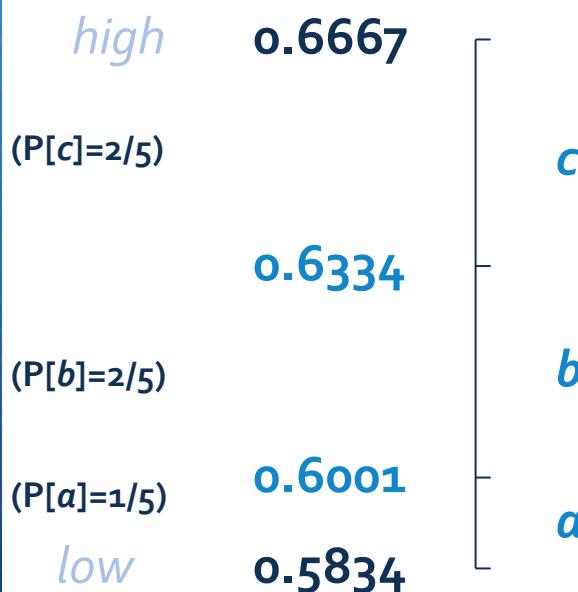
...continued...



...continued...

<i>high</i>	0.6667
($P[c]=2/5$)	
$(P[b]=2/5)$	
$(P[a]=1/5)$	
<i>low</i>	0.5834

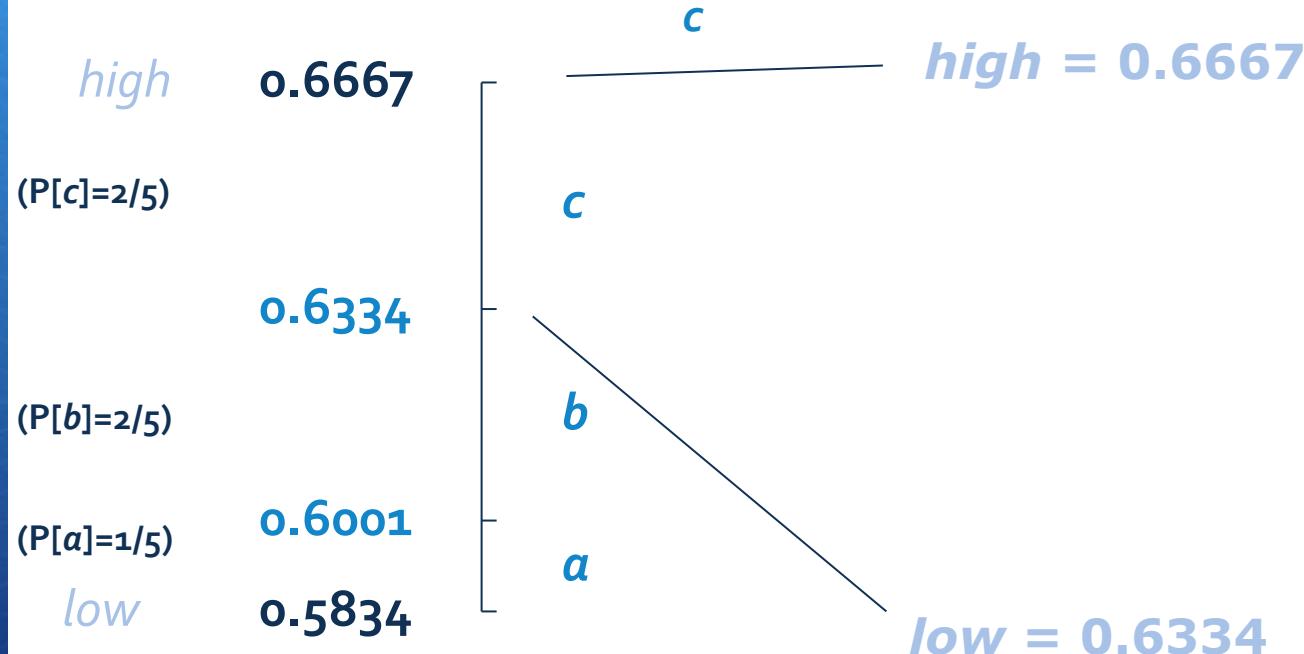
...continued...



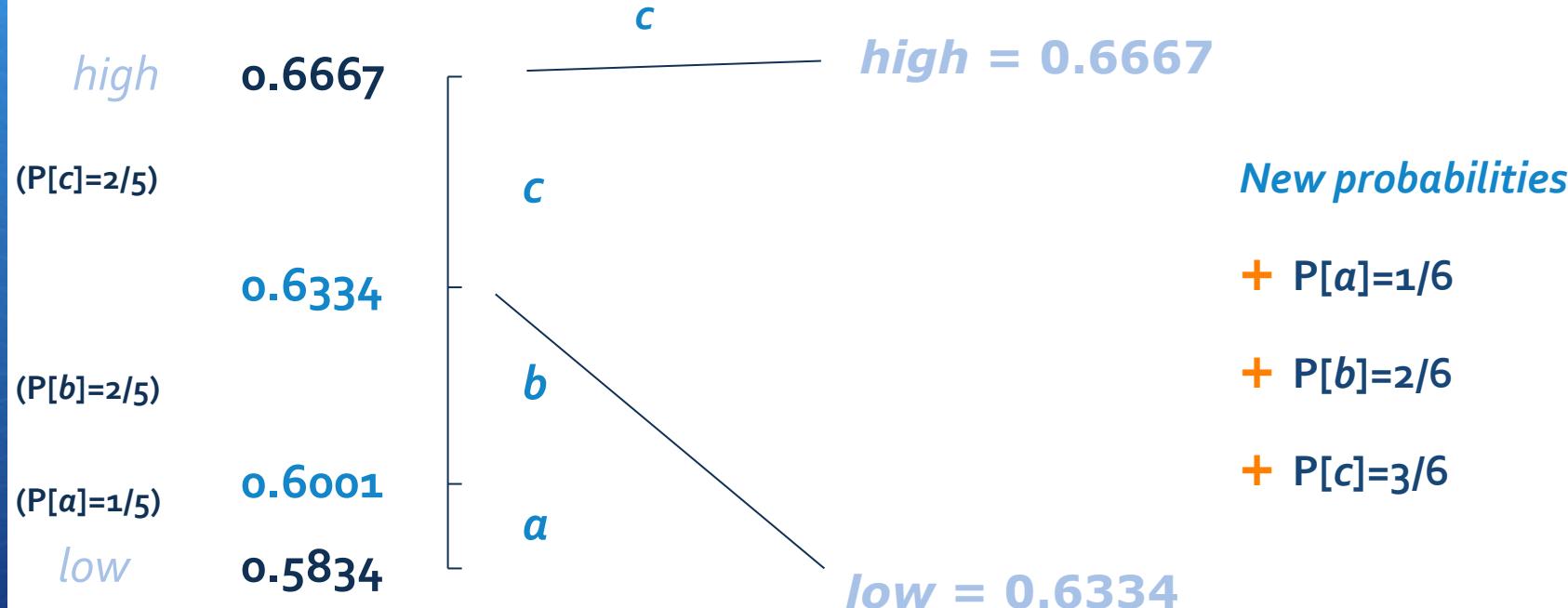
...continued...



...continued...



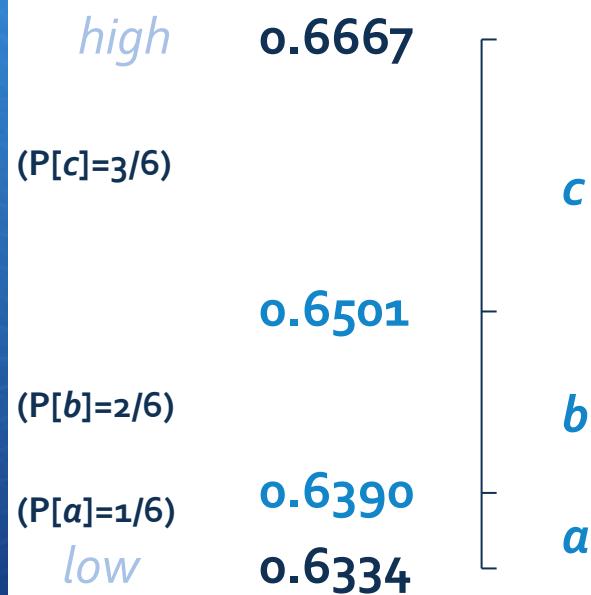
...continued...



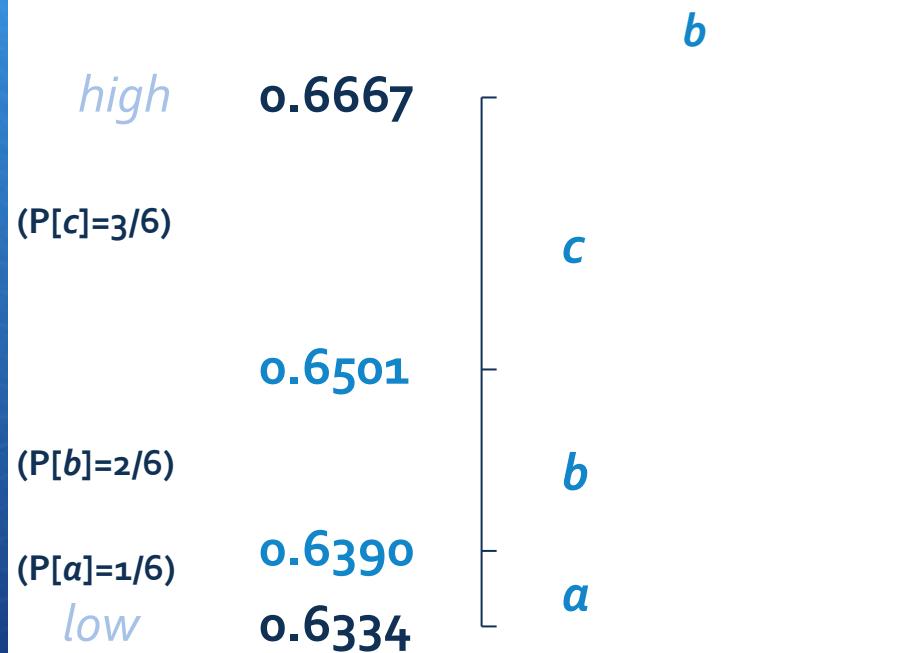
...continued...

<i>high</i>	0.6667
$(P[c]=3/6)$	
$(P[b]=2/6)$	
$(P[a]=1/6)$	
<i>low</i>	0.6334

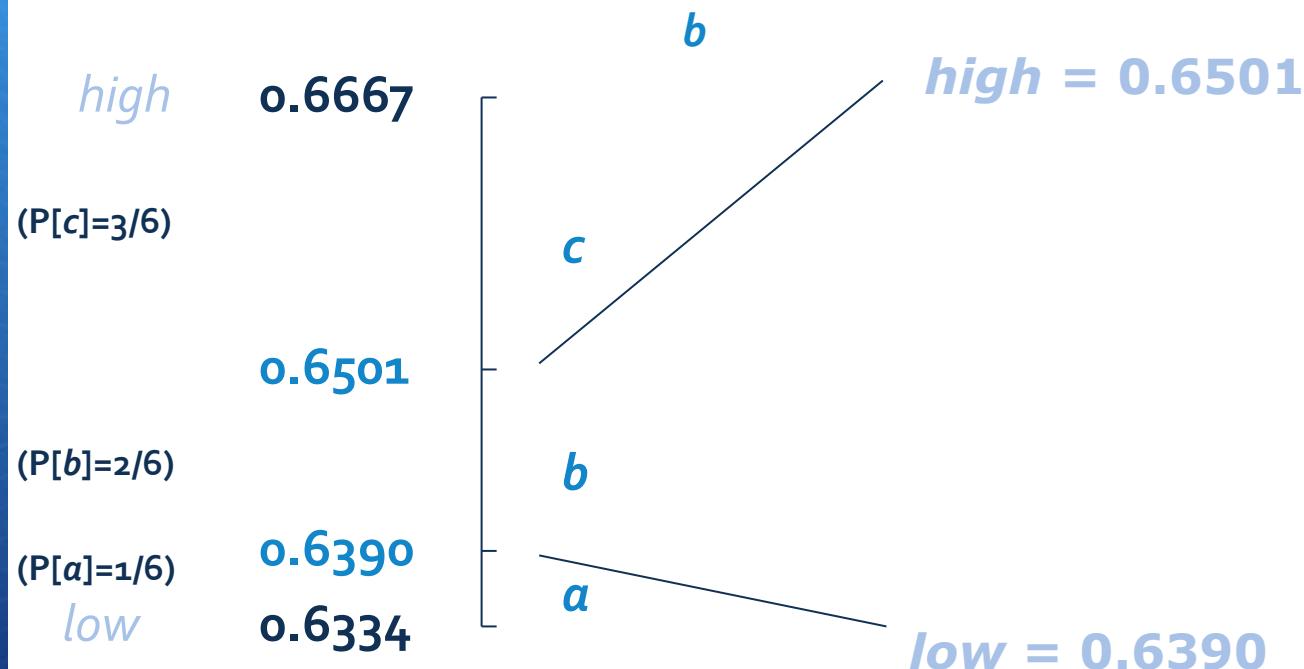
...continued...



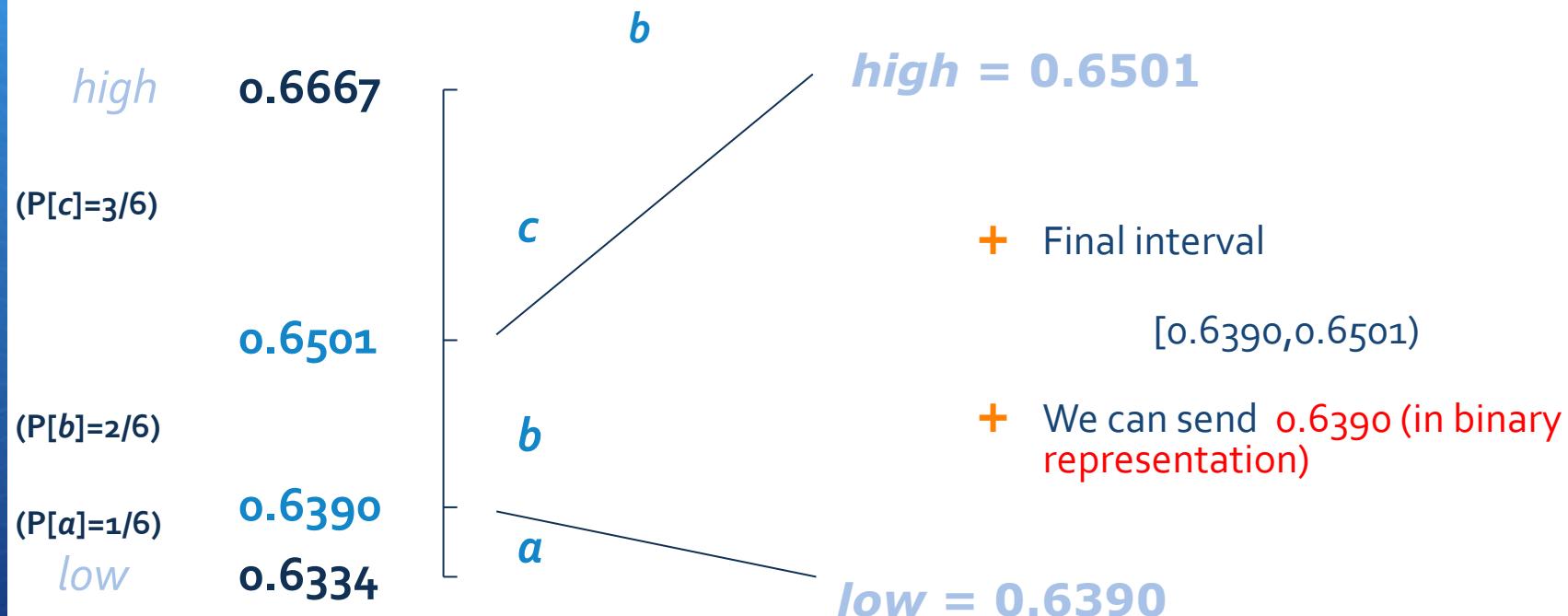
...continued...



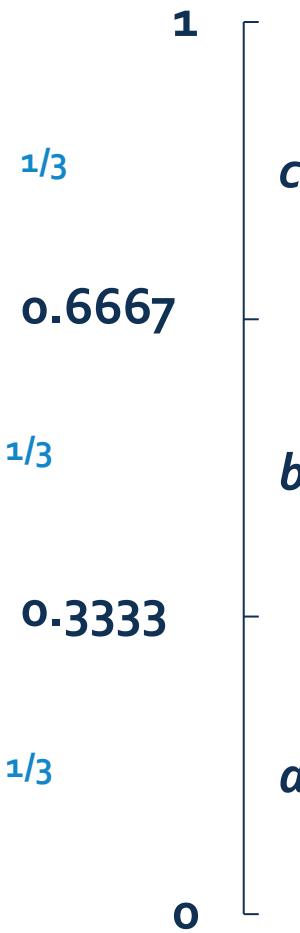
...continued...



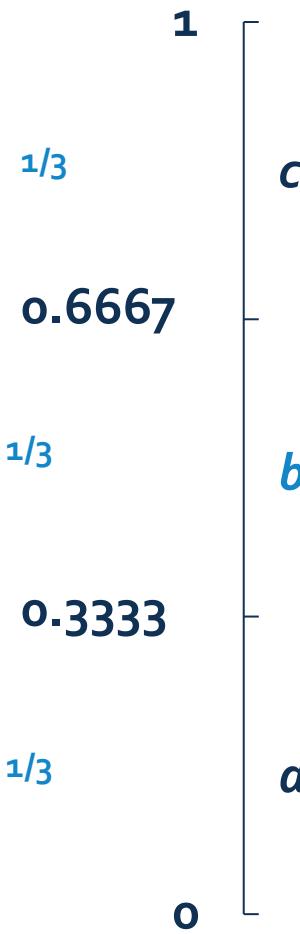
...continued...



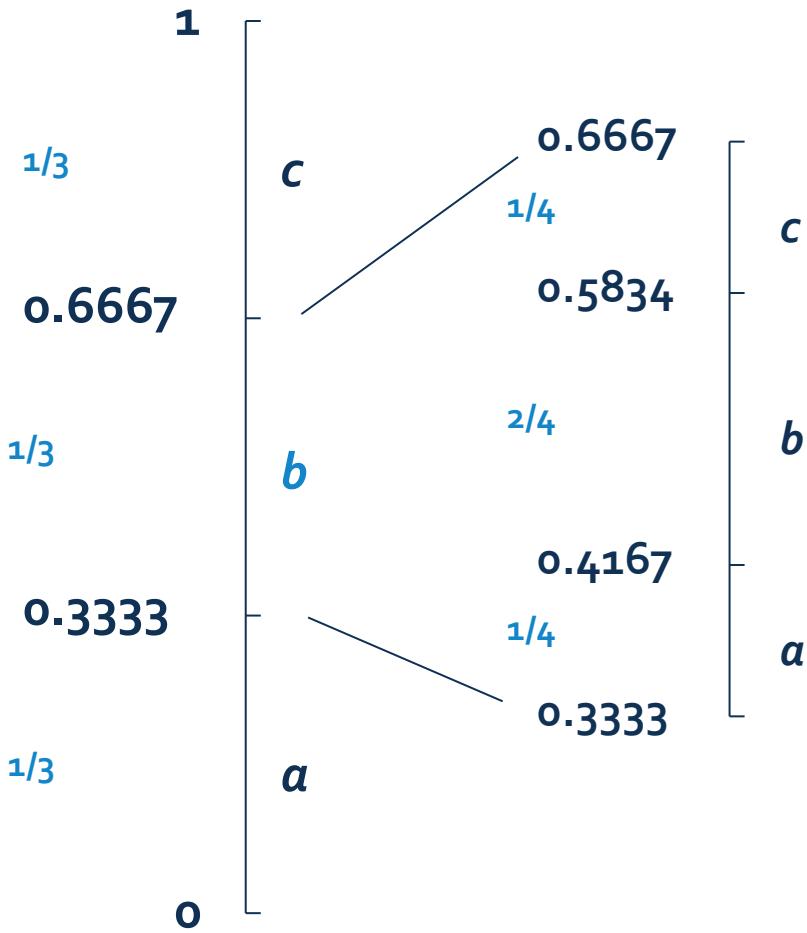
In summary



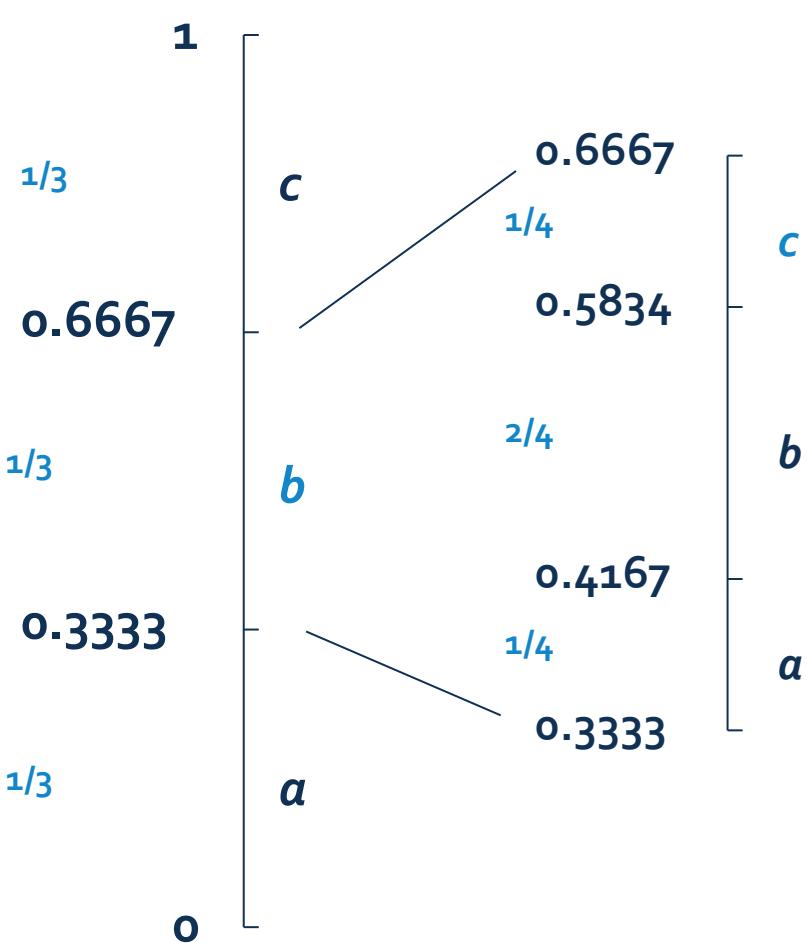
In summary



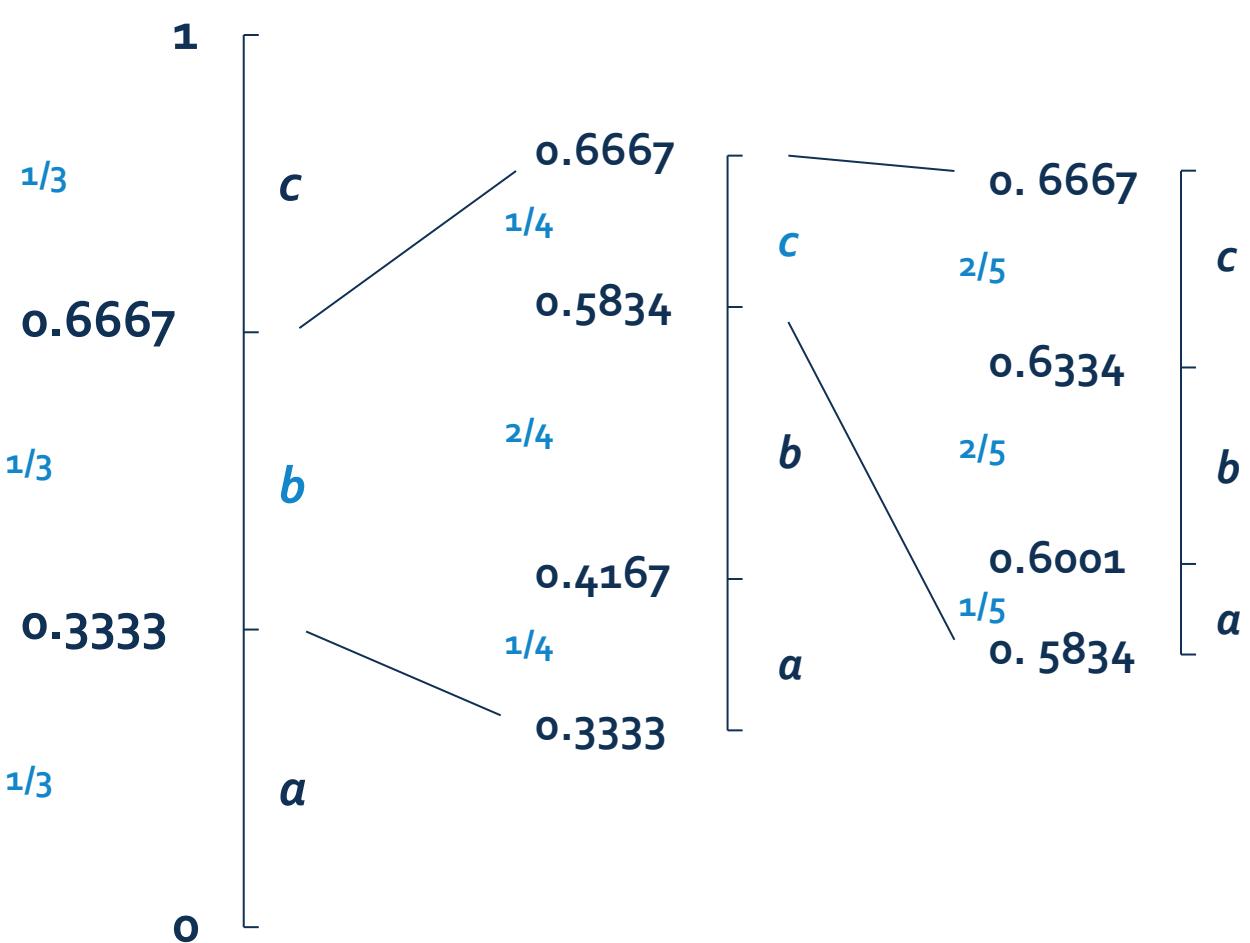
In summary



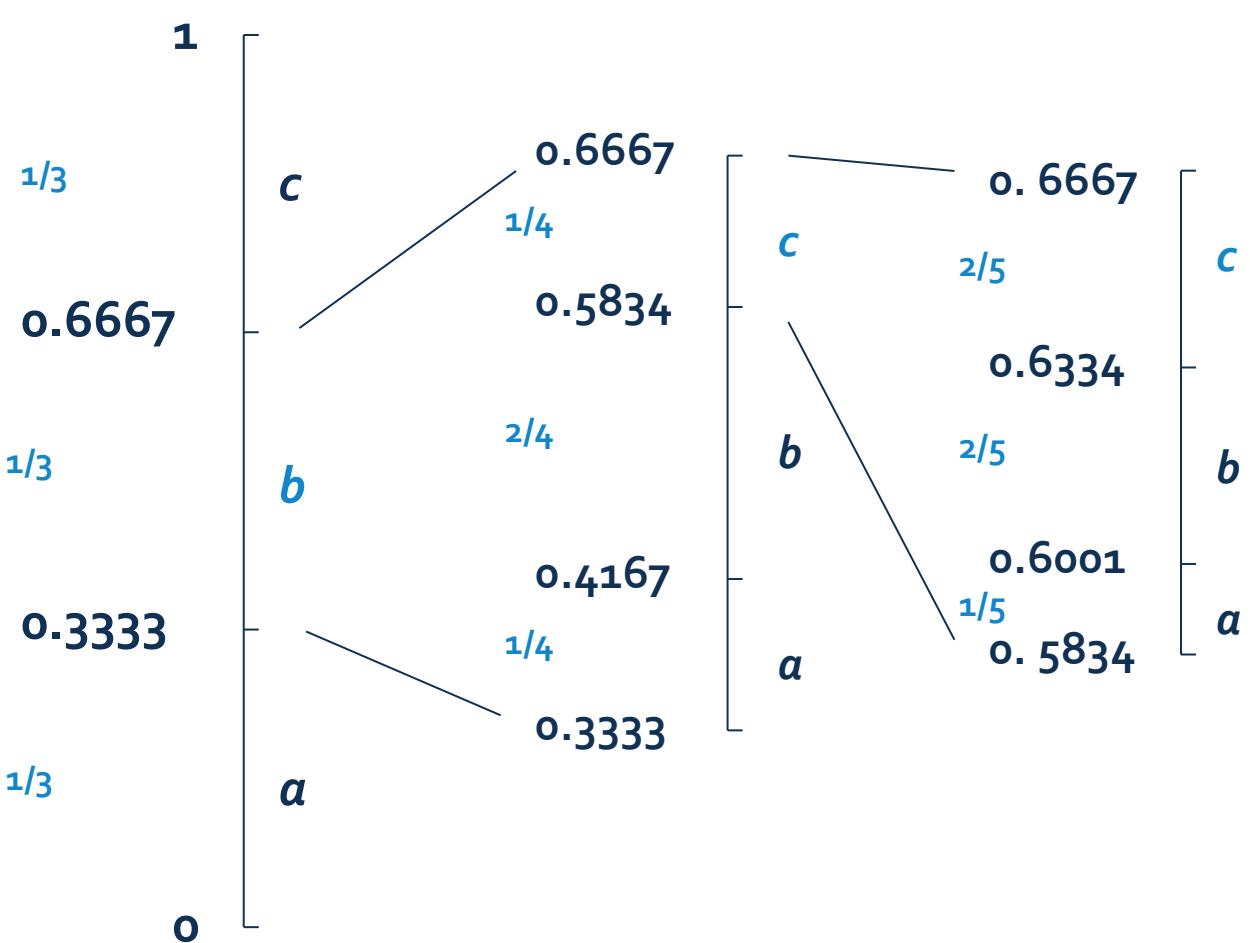
In summary



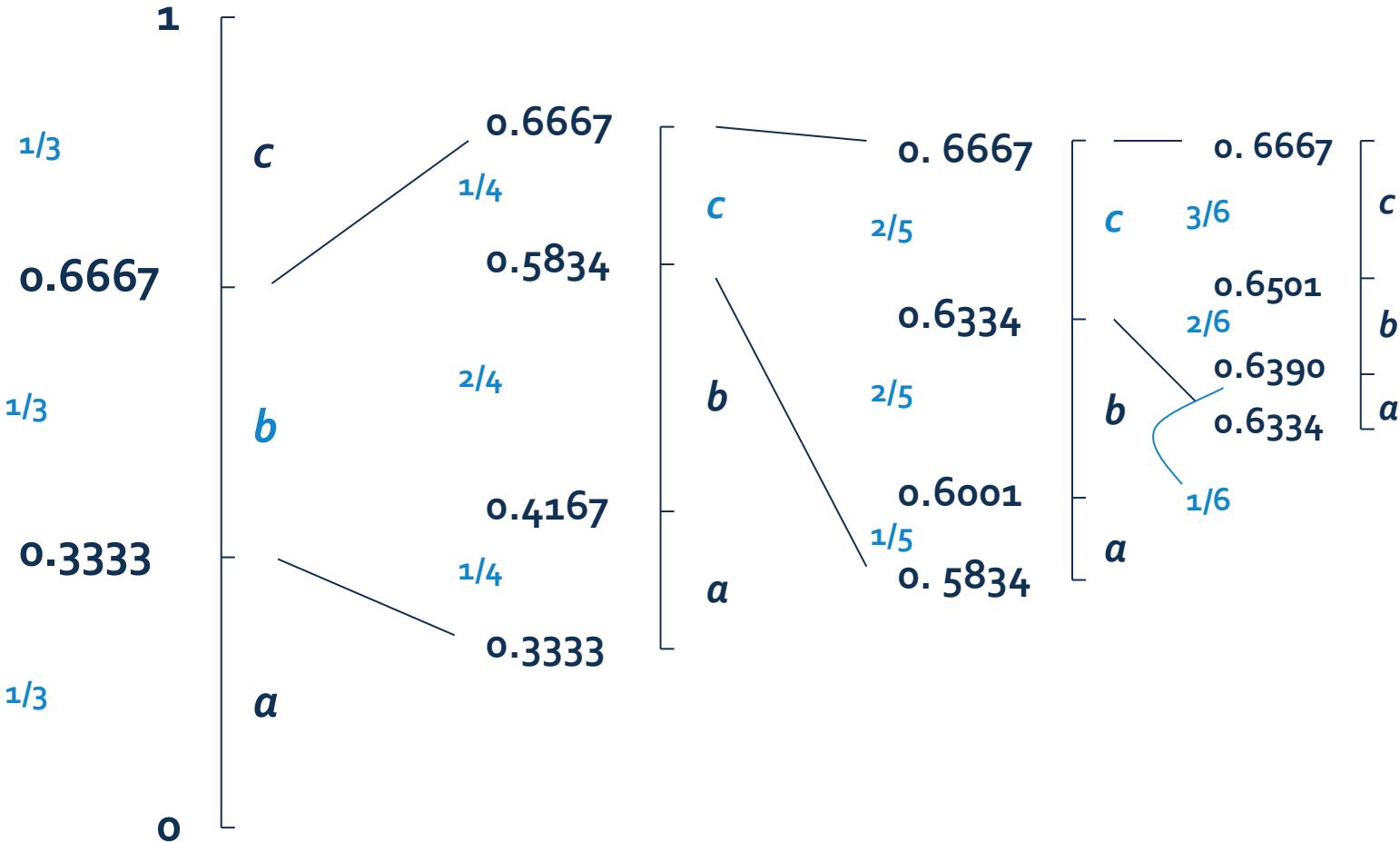
In summary



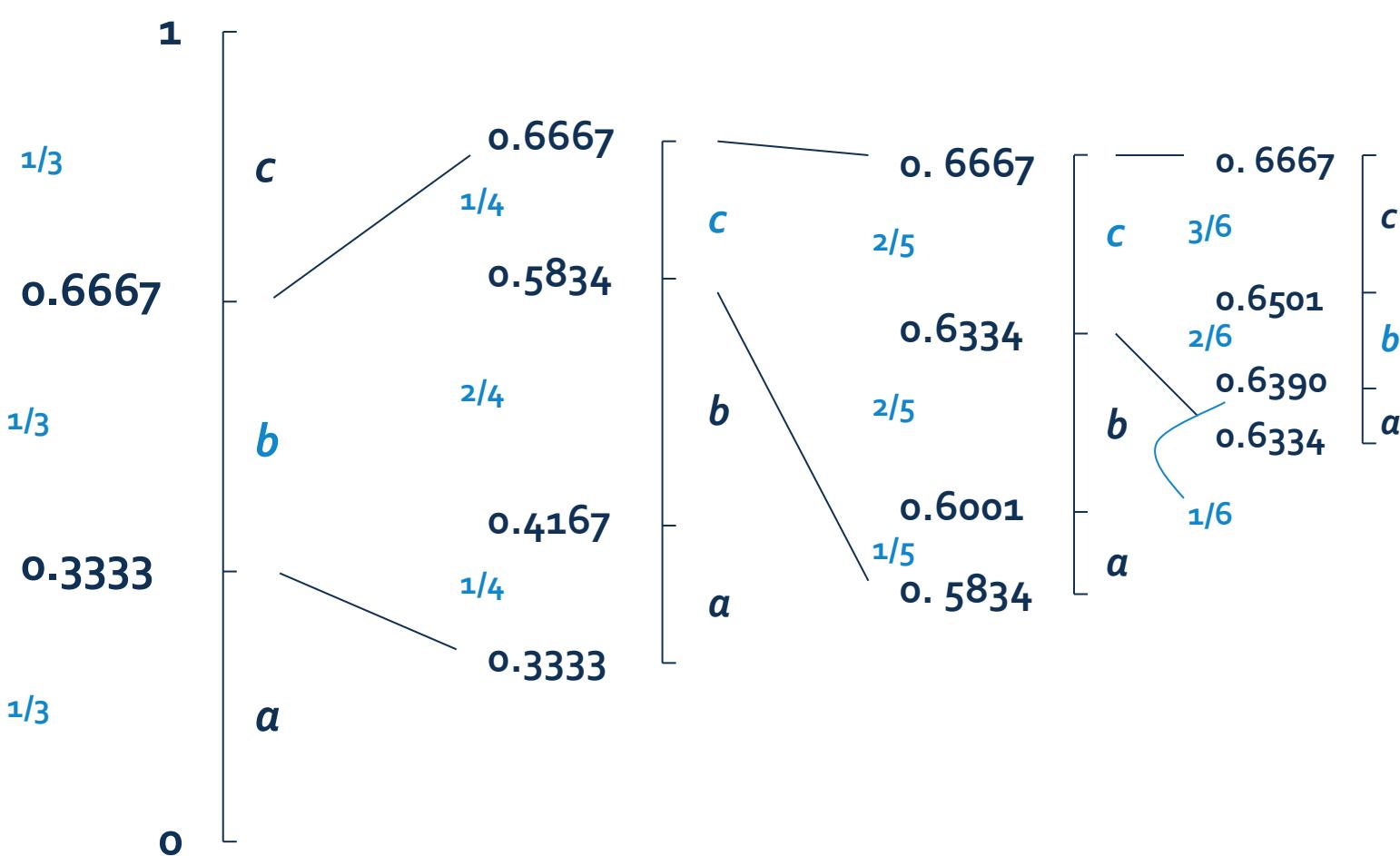
In summary



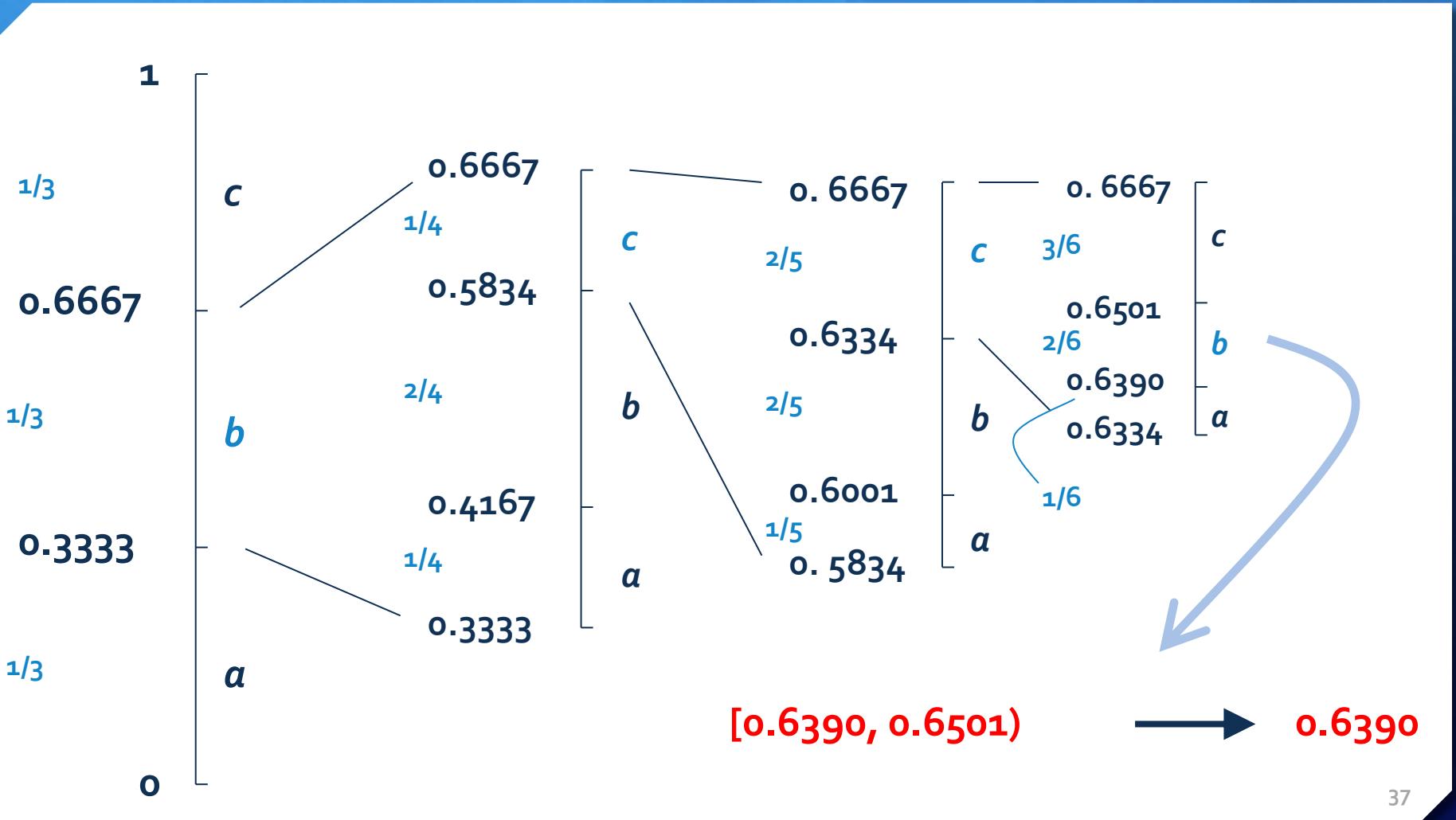
In summary



In summary



In summary



Decoding

- + Suppose we have to decode 0.6390
- + The decoder must know the probabilities of the symbols, as it simulates the behaviour of the encoder.
- + It starts with low=0 and high=1 and divides the interval in exactly the same way as the encoder:
a in [0, 1/3), b in [1/3, 2/3), c in [2/3, 1)

Decoding - II

- + The transmitted number falls in the interval corresponding to b, so b will be the first symbol of the decoding.
- + The decoder evaluates the new values for low (0.3333) and high (0.6667), updates the symbol probabilities and divides the range according to these new probabilities
- + Decoding proceeds until the entire string is reconstructed.

Decoding - III

- + 0.6390 in [0.3333, 0.6667) b 
- + 0.6390 in [0.5834, 0.6667) $c\dots$ 

And so on...

AC-Decoding (b,n)

Require: The binary representation b of the compressed output, the length n of S , the probabilities $P[\sigma]$ and the cumulative f_σ .

Ensure: The original sequence S .

- 1: $s_0 = 1$
 - 2: $l_0 = 0$
 - 3: $i = 1$
 - 4: **while** $i \leq n$ **do**
 - 5: subdivide the interval $[l_{i-1}, l_{i-1} + s_{i-1})$ into subintervals of length proportional to the probabilities of the symbols in Σ (in the predefined order)
 - 6: take the symbol σ corresponding to the subinterval in which $0.b$ lies
 - 7: $S = S :: \sigma$
 - 8: $s_i = s_{i-1} * P[\sigma]$
 - 9: $l_i = l_{i-1} + s_{i-1} * f_\sigma$
 - 10: $i = i + 1$
 - 11: **end while**
 - 12: $output = S$
-

Limitations for the implementation

- + The number x produced by the coding phase is known only when the entire input is processed: this is a disadvantage in situations like digital communications, in which for the sake of speed, we desire to start encoding/decoding before the source/compressed string is completely scanned; some possible solutions are:
 - + 1. the text to be compressed is subdivided into blocks, which are compressed individually; this way, even the problem of specifying the length of the text is relieved: only the length of the last block must be sent to permit its decompression, or the original file can be padded to an integral number of blocks, if the real 'end of file' is not important.
 - + 2. the two extremes of the intervals produced at each compression step are compared and the binary prefix on which their binary representation coincides is emitted. This option does not solve the problem completely, in fact it can happen that they don't have any common prefix for a long time, nonetheless this is effective because it happens frequently in practice.

Limitations for the implementation

- + More significantly, the encoding and decoding algorithms presented above require arithmetic with infinite precision which is costly to be approximated
- + There are several proposals about using finite precision arithmetic. There is a practical implementation , proposed by Witten, Neal and Clearly; sometimes called Range Coding. It is mathematically equivalent to Arithmetic Coding, which works with finite precision arithmetic so that subintervals have integral extremes.
- + The decoding process involving subtraction and division is simple in principle, but also impractical. The code, which is a single number, is normally long and can also be very long. A 1 Mbyte file can be encoded, for example, into a 500 Kbyte file consisting of a single number. Dividing a 500 Kbyte number can be complex and slow.

Range Coding

- + Specific implementation measures aim to avoid operations on real numbers with arbitrary precision
- + Arithmetic between integers is used (Witten, Neal and Clearly 1987)
- + The idea is to keep integers in the range $[0, R]$ where $R=2^k$. Whenever the current interval falls into the subinterval $[0, R/2]$, $[R/2, R]$ or $[R/4, 3R/4]$ the interval expands by a factor of 2. This results in the output of special bits that distinguish such cases.

Arithmetic coding vs. Huffman coding

- + A detailed comparison is provided in Bookstein, A., & Klein, S. T. (1993). *Is Huffman coding dead?* *Computing*, 50(4), 279–296.
- + In a typical English-language text, space is the most common symbol, with a probability of about 18%, so Huffman's upper limit of redundancy is quite small.
- + On the contrary, in black and white images, arithmetic encoding is much better than Huffman's unless a block (group) technique is used
 - + Arithmetic encoding requires less memory as the symbol representation is calculated on the fly
 - + Arithmetic encoding is more suitable for high performance models, whose predictions have a very high estimated probability

Arithmetic coding vs. Huffman coding

- + Huffman decoding is generally faster
- + In arithmetic encoding, it is not easy to start decoding in the middle of the bit stream.
- + In Huffman encoding, on the other hand, appropriate "entry points" can be used
- + In a collection of texts and images, Huffman is likely to be used for texts and arithmetic encoding for images.
- + If the given probabilities are correct, Arithmetic encoding may perform better. If, on the other hand, the statistical model is incorrect, then Huffman encoding is more robust to errors.

Exercise

- + Which is the binary representation of $19/64$?
- + Let us consider the statistical static model in which the probabilities are $p(a)=1/2$, $p(b)=p(c)=1/4$. How to recover the original message starting from the value $19/64$?

Lab Exercise

- + Let us consider the input text T=MISSISSIPPI
 - 1. Find the arithmetic coding by using the dynamic approach
 - 2. Find the Huffman code for T
 - 3. Find the canonical Huffman coding



Integer coding

Monday, March 27 2023

Definition of the problem

- + Let $S = s_1, s_2, \dots, s_n$ be a sequence of positive integers s_i , possibly repeated. The goal is to represent the integers of S as binary sequences which are self-delimiting (i.e. by using prefix codes) and use few bits.
- + Remark: the request about s_i of being positive integers can be relaxed by mapping a non-negative integer x to $2x+1$ and a negative integer x to $-2x$, thus turning again the set S to a set of just positive integers.

0	-1	1	-2	2	-3	3	-4	4	-5	5	...
1		2		3		4		5		6	

- + Such a problem can be applied in several real contexts:
 - + Search engines store for each term t the list of documents (i.e. Web pages, blog posts, tweets, etc. etc.) where t occurs. Answering a user query, formulated as sequence of keywords t_1, t_2, \dots, t_k , then consists of finding the documents where all t_i 's occur. This is implemented by intersecting the document lists for these k terms. Documents are usually represented via integer IDs, which are assigned during the crawling of those documents from the Web.
 - + Storing efficiently such lists of integers is very important
 - + Using a fixed-length binary encoding (i.e. 4 or 8 bytes) may require considerable space, and thus time for their retrieval, since the modern search engines index up to millions of documents.
 - + Two kinds of compression tricks are used: 1. sorting the document IDs in each list, and then encode each of them with the difference between it and its preceding ID in the list; 2. encoding each difference with a variable-length sequence of bits which is short for small integers.
 - + Several compression algorithms (i.e., LZ77, LZ78) produce encodings containing integers. In other compression schemes (for instance based on BWT) encoding of integers are used in the intermediate steps, with smaller values most probable and larger values increasingly less probable.

Main question

- + How to design a variable-length binary representation for (unbounded) integers which takes as few bits as possible and is prefix-free?

Simple binary codings

- + The simplest idea consists of taking $m = \max_j s_j$ and then encode each integer s_i by using $1 + \lceil \log_2 m \rceil$ bits.
- + This fixed-size encoding is useful when the integers to be represented are all close to each other and concentrated around the value 0, otherwise it becomes ineffective.
- + We could store each integer s_i with $1 + \lceil \log_2 s_i \rceil$ bits! The problem with this approach is that such a code is not a prefix code (the first codewords could be 0, 1, 10, 11, 100). How to decode 11011?

Universal encoding of integers

- + In the original definition, a **universal integer code** is a prefix code that associates each positive integer with a binary encoding, with the additional property that whatever the distribution over the integers (provided that this distribution is monotone, i.e. $p(i) \geq p(i + 1)$ for each i), the lengths of the code words are proportional (up to a constant) with the expected lengths assigned by an optimal code with an assigned probability distribution.
- + More formally, if $l(x)$ is the length of the codeword for x and $H(p)$ denotes the entropy of a source with distribution p , there exist constants c and d greater than 1 such that for any non-increasing distribution p we have

$$\sum_{x \in N} l(x)p(x) \leq cH(p) + d$$

- + It is shown that the universality property of an integer coding is related to the fact that the length of the code for the integer x is $O(\log(x))$ for each x . This means that they are as optimal as binary encoding with the additional property of being a prefix code. Binary encoding is in fact not a prefix code ($2=10$ is a prefix of $4=100$)

Huffman codes as universal integer codes

- + In general, most prefix codes assign longer codewords to greater integers.
- + Universal codes are used when the exact probabilities are not known, but only their ranking.
- + Huffman codes are often more convenient than universal integer codes. However, universal codes are useful when Huffman cannot be used.
 - + For example, the probability distribution is not known precisely.
 - + Or, if the sender but not the receiver knows the probability distribution, Huffman needs to transmit the probabilities to the receiver. A universal integer encoder does not need this.

Unary codes

- + One of the simplest instantaneous integer codes is the unary code.
- + The unary code encodes the positive integer x by $x-1$ bits set to 0 followed by a bit set to 1.
- + The length of the code word for the integer x is therefore x .
- + The code is trivially instantaneous. The first code words are 1, 01, 001, 0001.
- + Clearly the length of this encoding is exponentially longer than the length $\Theta(\log x)$ binary representation of the number.
It is not a universal coding.

Optimal codes with respect to a probability distribution

- + A code is optimal with respect to a given probability distribution p , if $I(x) = \log \frac{1}{p(x)}$, for each integer x
- + By solving the equation $|U(x)| = \log \frac{1}{p(x)}$ with respect to $p(x)$, we derive the distribution of the integers s_i for which the unary code is optimal.
- + Unary codes are optimal when the distribution is $p(x) = 2^{-x}$
- + Using this same argument we can also deduce that the fixed-length binary encoding, which uses $1 + \lfloor \log_2 m \rfloor$ bits for each integer, is optimal if $p(x) = \frac{1}{m}$, i.e, when the integers in S are distributed uniformly within the range $\{1, 2, \dots, m\}$.

Elias codes

- + There are two very simple universal codes for integers.
- + They have been introduced in the paper
Elias, Peter (March 1975). "Universal codeword sets and representations of the integers". IEEE Transactions on Information Theory. 21 (2): 194–203

Gamma code

- + Given an integer $x \geq 1$, the code $\gamma(x)$ is a binary sequence composed of two parts:
 - + a sequence of $|B(x)| - 1$ zeroes
 - + the binary representation $B(x)$ of x .

$$\gamma(9) = \boxed{0001001}$$

U(4) Bin(9)

How to compute the Gamma code of x

- + Let $N = \lfloor \log_2 x \rfloor$ be the exponent of the greatest power of 2 smaller than or equal to x , i.e. $2^N \leq x < 2^{N+1}$.
- + Write the bit '0' N times and, then, write the binary representation of x using $N+1$ bits.

To encode the integer x , $2\lfloor \log_2 x \rfloor + 1$ bits are used.
It is a universal code for integers

First Gamma codewords

x	Binary encoding	Gamma code
$1 = 2^0 + 0$	1	1
$2 = 2^1 + 0$	1 0	0 10
$3 = 2^1 + 1$	1 1	0 11
$4 = 2^2 + 0$	1 00	00 100
$5 = 2^2 + 1$	1 01	00 101
$6 = 2^2 + 2$	1 10	00 110
$7 = 2^2 + 3$	1 11	00 111
$8 = 2^3 + 0$	1 000	000 1000
$9 = 2^3 + 1$	1 001	000 1001
$10 = 2^3 + 2$	1 010	000 1010
$11 = 2^3 + 3$	1 011	000 1011
$12 = 2^3 + 4$	1 100	000 1100
$13 = 2^3 + 5$	1 101	000 1101
$14 = 2^3 + 6$	1 110	000 1110
$15 = 2^3 + 7$	1 111	000 1111
$16 = 2^4 + 0$	1 0000	0000 10000
$17 = 2^4 + 1$	1 0001	0000 10001

Decoding Gamma code

- + In order to decode a gamma code:
 - + Count the consecutive number of zeroes up to the first 1, say they are N.
 - + Fetch the following N+1 bits (included the 1) and interpret the sequence as the integer x

For instance: 0001001 -> $\gamma(9)$

Remarks

- + One can prove (by exercise) that the γ code is optimal when the distribution is

$$p(x) \approx \frac{1}{2x^2}$$

- + The inefficiency in the γ -code resides in the unary coding of the length $|B(x)|$ which is really costly as x becomes larger and larger.

Delta codes

- + Given an integer $x \geq 1$, the code $\delta(x)$ is a binary sequence composed of two parts:
 - + $\gamma(|B(x)|)$
 - + the binary representation $B(x)$ of x , the most significant bit excluded.

$$\delta(14) = \textcolor{green}{00} \textcolor{teal}{1} \textcolor{blue}{00} \textcolor{orange}{X} \textcolor{red}{110}$$

A diagram illustrating the construction of the delta code for 14. The code is shown as a sequence of binary digits: 00100X110. Three arrows point to different parts of this sequence: a green arrow points to the first three digits (001) and is labeled "U(3)"; a blue arrow points to the next three digits (00X) and is labeled "Bin(4)"; and a red arrow points to the last three digits (X110) and is labeled "Bin(14)".

U(3) Bin(4) Bin(14)

How to compute the Delta code of x

- + Let $N = \lfloor \log_2 x \rfloor$ be the exponent of the greatest power of 2 smaller than or equal to x , i.e. $2^N \leq x < 2^{N+1}$.
- + Write the gamma code for $N+1$ followed by the binary representation of x (the most significant bit is excluded)
 - + Example: $\delta(14) = 00100\ 110$

To represent an integer x we use

$$(1 + 2\lfloor \log_2(\lfloor \log_2 x \rfloor + 1) \rfloor) + \lfloor \log_2 x \rfloor \\ \approx 1 + \log x + 2 \log \log x \text{ bit}$$

- + It is a universal code for integers
- + It has been proved that δ code is optimal when the distribution is

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

First delta codes

x	N	N+1	Gamma code	Delta code
$1 = 2^0$	0	1	1	1
$2 = 2^1 + 0$	1	2	0 10	0 1 0 0
$3 = 2^1 + 1$	1	2	0 11	0 1 0 1
$4 = 2^2 + 0$	2	3	00 100	0 1 1 00
$5 = 2^2 + 1$	2	3	00 101	0 1 1 01
$6 = 2^2 + 2$	2	3	00 110	0 1 1 10
$7 = 2^2 + 3$	2	3	00 111	0 1 1 11
$8 = 2^3 + 0$	3	4	000 1000	00 1 00 000
$9 = 2^3 + 1$	3	4	000 1001	00 1 00 001
$10 = 2^3 + 2$	3	4	000 1010	00 1 00 010
$11 = 2^3 + 3$	3	4	000 1011	00 1 00 011
$12 = 2^3 + 4$	3	4	000 1100	00 1 00 100
$13 = 2^3 + 5$	3	4	000 1101	00 1 00 101
$14 = 2^3 + 6$	3	4	000 1110	00 1 00 110
$15 = 2^3 + 7$	3	4	000 1111	00 1 00 111
$16 = 2^4 + 0$	4	5	0000 10000	00 1 01 0000
$17 = 2^4 + 1$	4	5	0000 10001	00 1 01 0001

Decoding Delta codes

- + Let L be the number of zeroes until the first bit 1 is reached.
- + Fetch the following L+1 bits (included the 1) and interpret the sequence as the integer N+1
- + Consider the integer obtaining by decoding 1 followed by the next N bits
 - + Example: 00 101 0011
 - + L = 2
 - + consider 1 followed by 01 -> 5
 - + consider the next 4 bits prepended by 1, i.e.
 $10011 = 2^4 + 3 = 19$

Lab exercise

- + Write a program that returns the gamma and delta codes for the first n positive integers. Compare these lengths.
- + Compare the average length A of the gamma codes for the first n positive integers and compare it with the average length B of the binary encoding.
- + What is the value of A/B ? How does this value vary as n varies from 1 to 1000?

Fibonacci code

- + Fibonacci numbers: $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, F_7 = 13, F_8 = 21, F_9 = 34, F_{10} = 55, F_{11} = 89, \dots$
- + Such a code was introduced by R. Capocelli in 1989.
- + It is based on the Zeckendorf theorem: every positive integer can be represented uniquely as the sum of one or more distinct Fibonacci numbers (of order greater than or equal to 2) in such a way that the sum does not include any two consecutive Fibonacci numbers.
- + Example $110=89+21$
 - + Other representations can be found, for instance $110=89+13+8$ but they contain consecutive Fibonacci numbers
- + Example $73=55+13+5$

Fibonacci code

+ More formally:

Given a number $N > 0$, the correspondent Fibonacci codeword is defined as

$$d_0 d_1 d_2 \dots d_{k-1} d_k$$

if $N = \sum_{i=0}^{k-1} d_i F_{i+2}$ e $d_{k-1} = d_k = 1$.

For instance, the Fibonacci code for 73 is 0001010011

Fibonacci code

- + The Zeckendorf representation has the property that no two consecutive bits are equal to 1.
- + It is different from the polynomial binary representation in base ϕ , and it adds a bit '1' so that it ends with '11', resulting in a prefix code.
- + 11 appears only at the end.

First Fibonacci codewords

Symbol	Fibonacci representation	Fibonacci code word
1	$F(2)$	11
2	$F(3)$	011
3	$F(4)$	0011
4	$F(2)+F(4)$	1011
5	$F(5)$	00011
6	$F(2)+F(5)$	10011
7	$F(3)+F(5)$	01011
8	$F(6)$	000011
9	$F(2)+F(6)$	100011
10	$F(3)+F(6)$	010011
11	$F(4)+F(6)$	001011
12	$F(2)+F(4)+F(6)$	101011
13	$F(7)$	0000011
14	$F(2)+F(7)$	1000011

Enconding algorithm

To encode a positive integer N :

1. Find the largest Fibonacci number equal to or less than N ; subtract this number from N , keeping track of the remainder.
2. If the number subtracted was the i th Fibonacci number $F(i)$, put a 1 in place $i-2$ in the codeword (counting the left most digit as place 0).
3. Repeat the previous steps, substituting the remainder for N , until a remainder of 0 is reached.
4. Place an additional 1 after the rightmost digit in the codeword.

Decoding Fibonacci code

- + remove the final "1",
- + assign the remaining the values 1,2,3,5,8,13... (the Fibonacci numbers) to the bits in the code word, and sum the values of the "1" bits.

Remarks

- + It is a universal code for integers. It is possible to prove that the integer x is represented by using at most $1 + \log_{\varphi} \sqrt{5}x$ bits
- + It is also a synchronizing code, i.e. a UD code that contains a codeword (or substring) that allows the next code word to be identified.
- + This is useful in the case of encoding errors, to remedy the error.

Levenshtein code

The code of zero is "o";

To encode a positive number:

1. Initialize the step count variable C to 1.
2. Write the binary representation of the number without the leading "1" to the beginning of the code.
3. Let M be the number of bits written in step 2.
4. If M is not 0, increment C, repeat from step 2 with M as the new number.
5. Write C "1" bits and a "o" to the beginning of the code.

First Levenshtein codewords

0	0
1	10
2	110 0
3	110 1
4	1110 0 00
5	1110 0 01
6	1110 0 10
7	1110 0 11
8	1110 1 000
9	1110 1 001
10	1110 1 010
11	1110 1 011
12	1110 1 100
13	1110 1 101
14	1110 1 110
15	1110 1 111
16	11110 0 00 0000
17	11110 0 00 0001

Decoding Levenshtein code

To decode a Levenstein-coded integer:

1. Count the number of "1" bits until a "0" is encountered.
2. If the count is zero, the value is zero, otherwise
3. Start with a variable N, set it to a value of 1 and repeat count-1 times:
 4. Read N bits, prepend "1", assign the resulting value to N

Exercise: Which is the length of the Levenshtein code for a positive integer x?

Rice codes

- + Gamma and delta codes are universal and pretty efficient whenever the set S is concentrated around zero; however, it must be noted that these two codes need a lot of bit shifts to be decoded and this may be slow if numbers are larger and thus encoded in many bits.
- + There are situations in which integers are concentrated around some value, different from zero; here, Rice coding becomes advantageous both in compression ratio and decoding speed. They are used in several image and audio data compression methods.
- + It is a parametric code, i.e., it depends on a positive integer k .
- + The Rice code $R_k(x)$ of an integer x , given a parameter k , consists of two parts: the quotient $q = \lfloor (x-1)/2^k \rfloor$ and the remainder $r = x - 2^k q - 1$.
- + The quotient is stored in unary using $q + 1$ bits (the $+1$ is needed because q may be 0), the remainder r is in the range $[0; 2^k)$ and thus it is stored in binary using k bits. So, the quotient is encoded in variable length, whereas the remainder is encoded in fixed length.

EXAMPLE: Let us consider $R_6(83)$

$$R(83) = 0000010010$$

The diagram shows the binary number 0000010010. A green arrow points down to the first six digits, 000001, labeled U(6). A blue arrow points down to the last four digits, 0010, labeled Bin(2).

Rice codes

- + The closer 2^k is to the value of x , the shorter is the representation of q , and thus the faster is its decoding. For this reason, k is chosen in such a way that 2^k is concentrated around the mean of S 's elements.
- + The bit length of $R_k(x)$ is $q + k + 1$. This code is a particular case of the Golomb Code (used in some JPEG compression schemes), it is optimal when the values to be encoded follow a geometric distribution with parameter p , namely $p(x) = (1 - p)^{x-1}p$. In this case, if $2^k = \ln \frac{2}{p} \approx 0,69 \text{ mean}(S)$, the Rice code generate an optimal prefix-code

Lab exercises

- + Portfolio: Write a program in a programming language of your choice that computes all the universal codes of integers we have studied (both encoding and decoding).
- + Plot for each $n=1,..1000$ the lengths of the binary, gamma, delta, Fibonacci codes. Also consider the Rice encoding for $k=5$ and $k=7$
- + Report the statistics on the following experiments:
 - + Number of bits required to encode 100 integers between 1 and 100,000 (Consider integers 1, 1011, 2021, ...)
 - + Number of bits required to compress 100 random integers between 1 and 1000.
 - + Number of bits required to encode a sequence of 1000 integers with a distribution chosen in advance



Burrows-Wheeler Transform

Monday, April 3rd 2023

Burrows-Wheeler Transform

- + It was introduced in 1994 by M. Burrows and D. Wheeler in the paper:
A block sorting lossless data compression algorithm
Technical report 124, Digital Equipment Corporation, 1994.
Google Scholar reports **3560 citations from scientific papers**
- + It is a very powerful but also simple tool that transforms a string by making it more compressible
- + Bzip2 (Seward, 1996) e Szip (Schindler, 1997) are text-compressors based on BWT and are among the best compressors available.
- + BWT has been used to construct an indexing compressed data structure (Ferragina and Manzini 2000). Such a structure is called FM-index.
Google Scholar reports **1455 citations from scientific papers**
- + Several applications also in Bioinformatics (BWA and Bowtie2 are very well known tools to align reads to a reference genome. They are based on BWT)
- + Thanks to BWT and FM-index, M. Burrows, P. Ferragina and G. Manzini have just received the ACM Paris Kanellakis Theory and Practice Award for theoretical accomplishments that have had a significant and demonstrable effect on the practice of computing.

Burrows-Wheeler Transform (BWT)

- Transform

abraca  *caraab, 1*

- Reverse

caraab, 1  *abraca*

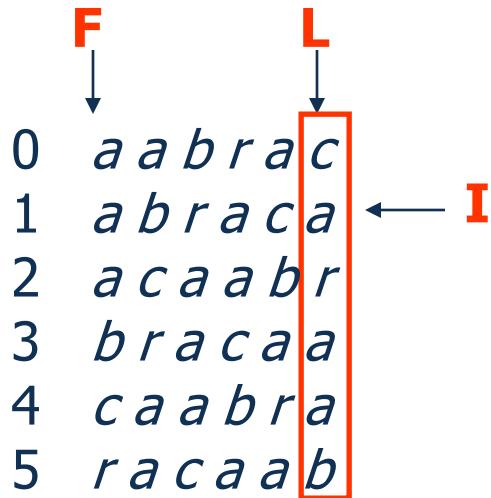
BWT

- INPUT: $w = \text{abraca}$
- Lexicographically sorting of all cyclic rotations of w

	F	L	I
0	$aabrac$		
1	$abraca$		← 1
2	$acaab$		
3	$braca$		
4	$caab$		
5	$racaab$		

- OUTPUT: $\text{BWT}(w) = \text{caraab}$ and the index $I=1$

Properties



- $\forall i \neq I$ the character $L[i]$ is followed in w by $F[i]$;
- The first letter of w is $F[I]$;
- For every character x , the i -th occurrence of x in F corresponds to the i -th occurrence of x in L .

The transformation is reversible

Given $L = \text{BWT}(w) = \text{caraab}$ and $I = 1$:

- Construct F by lexicographically sorting the elements of L

F	L	$\tau = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 4 & 5 & 0 & 2 \end{pmatrix}$
I →	$a \ 0$	$0 \ c$
	$a \ 1$	$1 \ a$
	$a \ 2$	$2 \ r$
	$b \ 3$	$3 \ a$
	$c \ 4$	$4 \ a$
	$r \ 5$	$5 \ b$

$$w = a \ b \ r \ a \ c \ a$$

Recovering w

For each $i = 0, \dots, n-1$,

$$w[i] = F[\tau^i[/]]$$

where:

$$\tau^0[x] = x$$

$$\tau^{i+1}[x] = \tau[\tau^i[x]]$$

Example:

- + Compute the BWT of the text $w=abaababa$

Evaluating the performance of BWT



Why does BWT improve compression?

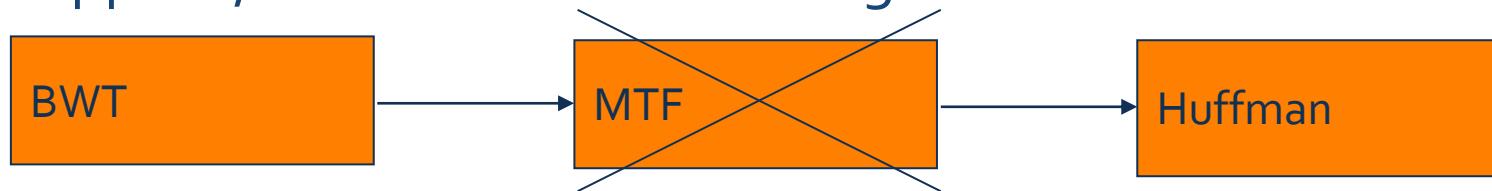
In general, the output $bwt(s) = L$ is locally homogeneous: it consists in concatenating several substrings containing only a small number of distinct symbols (long equal-letter runs).

w="...She...the...The... He...the...that...the...she...the...those..."

ha	t
he	t
he	S
he	s
he	t
ho	t

Approach proposed by Burrows and Wheeler

- + Burrows and Wheeler suggested to process $bwt(s)$ by using Move-to-Front coding.
- + MTF coding is a *symbol ranking* technique, that produces a globally homogeneous text, and in general more compressible.
- + After MTF coding, a zero-order compression algorithm is applied, such as the Huffman algorithm or arithmetic coding.



PROBLEM: Is the MTF coding required?

ANSWER: No, Giancarlo e S. in 2003. Two years later, Ferragina et al. in 2005 provided a partition technique, but the implementation is not efficient.

Why is MTF useful?

- “**Clustering**” of symbols and MTF

- MoveToFront Coding (MFT)

Encodes a character x with an integer that counts the number of distinct symbols seen since the previous occurrence of x .

EXAMPLE abaaaabbbbbccccccaaaaaa → 01100010000200020000

- BWT+MTF =many runs of zeroes are guaranteed to be effective for compressors such as Huffman and AC

Move-to-front coding

- + Given the list X containing the characters of the alphabet in lexicographic order, MTF acts on the characters in L and produces a vector of integers $R[0], \dots, R[n-1]$.
- + For each character, MTF stores its position in the list and move the symbol in front of the list.
- + If $bwt(s)$ contains many substrings containing a few different symbols, MFT coding generates only a small number of integers (mainly 0 and 1)

Example:

$bw(s) = L = "caraab"; X = \{"a", "b", "c", "r"\}$

<u>caraab</u>	<u>abcr</u>	2
<u>caraab</u>	<u>cabr</u>	1
<u>caraab</u>	<u>acbr</u>	3
<u>caraab</u>	<u>racb</u>	1
<u>caraab</u>	<u>arcb</u>	0
<u>caraab</u>	<u>arcb</u>	3
	<i>bacr</i>	

$$R=[2 \ 1 \ 3 \ 1 \ 0 \ 3]$$

Move-to-front decoding

- + Let X be the list containing the symbols of the alphabet in lexicographic order.
- + For each $i = 0, \dots, n-1$, assign to $L[i]$ the character at position $R[i]$ in the list X (0-based numeration) and move the character to the front of X .
- + After n steps the text L , the last column of the matrix M , is recovered.

Important issues

- + What is the compression ratio performance of BWT-based algorithms?
- + How is BWT efficiently computed, so that it can be used effectively as preprocessing of a compressor?
- + How can BWT be useful for the pattern matching problem?

Analysis of a BWT-based compressor

- + Since 1994, numerous BWT-based compression techniques have been introduced. The first analysis of the performance of such compressors was carried out in

G. Manzini. An analysis of the burrows-wheeler transform. Journal of the ACM, 48(3):407–430, 2001.

- + G. Manzini compares the compression ratio of BWT-based algorithms and the **empirical entropy** of the input.
The empirical entropy is defined in terms of the number of occurrences of each symbol or group of symbols. It is defined for each string without any probabilistic assumptions about the source, so it can be used in worst-case analysis.

0-order empirical entropy

- + It is based on the probability distribution implicitly defined by the input, i.e. if s is the input of size n on an alphabet with h symbols

$$H_0(s) = - \sum_{i=1}^h \frac{n_i}{n} \log \frac{n_i}{n}$$

- + We assume that $0 \log 0 = 0$
- + The value $|s|H_0(s)$ represents the size of the output of an ideal compressor that uses $-\log \frac{n_i}{n}$ bits to encode the i -th symbol of the alphabet.

Examples

- $n_1 = n_2 = \dots = n_h :$

$$H_0(s) = \log(h)$$

Esempi

- $n_1 = n_2 = \dots = n_h :$
$$H_0(s) = \log(h)$$
- $n_1 \gg n_2, n_3 \dots, n_h :$
$$H_0(s) \approx 0$$

Esempi

- $n_1 = n_2 = \dots = n_h :$

$$H_0(s) = \log(h)$$

- $n_1 \gg n_2, n_3 \dots, n_h :$

$$H_0(s) \approx 0$$

- $s = \text{mississippi}$

$$H_0(s) = -\frac{1}{11} \log\left(\frac{1}{11}\right) - \frac{4}{11} \log\left(\frac{4}{11}\right) - \frac{4}{11} \log\left(\frac{4}{11}\right) - \frac{2}{11} \log\left(\frac{2}{11}\right) = 1.82$$

o-order compressors

- + When a text is compressed by using **Huffman Coding o Arithmetic Coding** the output has a size close to $|s|H_o(s)$ bits.

o-order compressors

- + When a text is compressed by using **Huffman Coding o Arithmetic Coding** the output has a size close to $|s|H_o(s)$ bits.
- + In particular, one can prove that

$$|Huff(s)| \leq |s|H_0(s) + |s|$$

$$|Arit(s)| \leq |s|H_0(s) + \mu|s| \quad (\mu \approx 0.01)$$



It is the bound achieved by implementations using finite-precision arithmetic

k-th order empirical entropy

- + Zero-order entropy is often not accurate enough to measure the performance of a compressor.
- + For example, if we want to compress an English text, we would benefit both from knowledge of the frequencies but also from the fact that these frequencies may depend on the context.
- + If we consider a context of length k:

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

where w_s is a string containing all the symbols that follows w in s.

- + We can give an analogous definition by considering the symbols preceding w.
- + This means that $|s|H_k(s)$ represents a lower bound for a compressor that uses a memory of length k
- + $H_k(s)$ is a function that decreases as k increases, i.e., $H_{k+1}(s) \leq H_k(s)$
- + There exists also a notion of modified empirical entropy H_k^* that uses more accurate statistics (for instance, all the occurrence of the factors up to length k)

Examples

$s = \text{mississippi}$ ($k=1$)

- $m_s=i \rightarrow H_0(i)=0$
- $i_s=ssp \rightarrow H_0(ssp)=0.92$
- $s_s=sisi \rightarrow H_0(sisi)=1$
- $p_s=pi \rightarrow H_0(pi)=1$

Examples

s = mississippi (k=1)

- m_s=i → H₀(i)=0
- i_s=ssp → H₀(ssp)=0.92
- s_s=sisi → H₀(sisi)=1
- p_s=pi → H₀(pi)=1

$$H_1(s) = \frac{1}{11} (0 + 3 \cdot 0.92 + 4 \cdot 1 + 2 \cdot 1) = 0.79$$

Examples

s = mississippi (k=1)

- m_s=i → H₀(i)=0
- i_s=ssp → H₀(ssp)=0.92
- s_s=sisi → H₀(sisi)=1
- p_s=pi → H₀(pi)=1

$$H_1(s) = \frac{1}{11} (0 + 3 \cdot 0.92 + 4 \cdot 1 + 2 \cdot 1) = 0.79$$

$$H_0(s) = 1.82$$

Why is BWT important?

- + Note that $bwt(s^R) = \bigcup_{w \in A^k} \pi(w_s)$
- + Moreover $H_0(w_s) = H_0(\pi(w_s))$

Theorem (Manzini) : Let $u = bwt(s^R)$, where s^R is the reverse of the text s . For each positive integer k , there exists a factorization u_1, u_2, \dots, u_m of u such that

$$H_k(s) = \frac{1}{|u|} \sum_{i=1}^m |u_i| H_0(u_i)$$



Burrows-Wheeler Transform (part 2)

Monday, April 7th 2023

Burrows-Wheeler Transform (BWT)

- + It is a reversible transformation introduced in 1994 by Michael Burrows and David Wheeler in the field of Data Compression.
 - + M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL System Research Center, 1994.
- + It represents an interesting combinatorial tool (from a theoretical and applicative point of view)
- + Why is it useful?
 - + The main and simple motivation is related to the **clustering effect** that the BWT produces. More in particular, BWT permutes the input and it is likely to reduce the number of runs.
- + EXAMPLE: When $v = \text{mathematics}$, we have:
 - + $\text{bwt}(v) = \text{mmihttsecaa}$.

BWT: Definition

- + The BWT takes as input a word v and produces:
 - + a permutation $bwt(v)$ of the symbols of v , obtained as concatenation of the last symbols of the lexicographically sorted list M of its conjugates;
 - + the index I is the row of M containing the original word v .
- + EXAMPLE: $v = mathematics$

m a t h e m a t i c s
a t h e m a t i c s m
t h e m a t i c s m a
h e m a t i c s m a t
e m a t i c s m a t h
m a t i c s m a t h e
a t i c s m a t h e m
t i c s m a t h e m a
i c s m a t h e m a t
c s m a t h e m a t i
s m a t h e m a t i c

Lex sorting
→

	<i>M</i>
0	<i>a t h e m a t i c s m</i>
1	<i>a t i c s m a t h e m</i>
2	<i>c s m a t h e m a t i</i>
3	<i>e m a t i c s m a t h</i>
4	<i>h e m a t i c s m a t</i>
5	<i>i c s m a t h e m a t</i>
6	<i>m a t h e m a t i c s</i> ←
7	<i>m a t i c s m a t h e</i>
8	<i>s m a t h e m a t i c</i>
9	<i>t h e m a t i c s m a</i>
10	<i>t i c s m a t h e m a</i>

- + Output: $bwt(v) = L = mmihttsecaa$ and $I = 6$.

Why does BWT improve compression?

In general, the output $bwt(s) = L$ is locally homogeneous: it consists in concatenating several substrings containing only a small number of distinct symbols (long equal-letter runs).

w="...She...the...The... He...the...that...the...she...the...those..."

ha	t
he	t
he	S
he	s
he	t
ho	t

BWT is reversible

- + Starting from $\text{bwt}(v)$ and I , it is possible to reconstruct the word v .
- + The reconstruction procedure is based on the following properties:

1. The last symbol of v is $L[I]$
2. For each symbol z , the i -th occurrence of z in L corresponds to the i -th occurrence of z in F (LF mapping);

$$LF = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 6 & 7 & 5 & 4 & 9 & 10 & 8 & 3 & 2 & 0 & 1 \end{pmatrix}$$

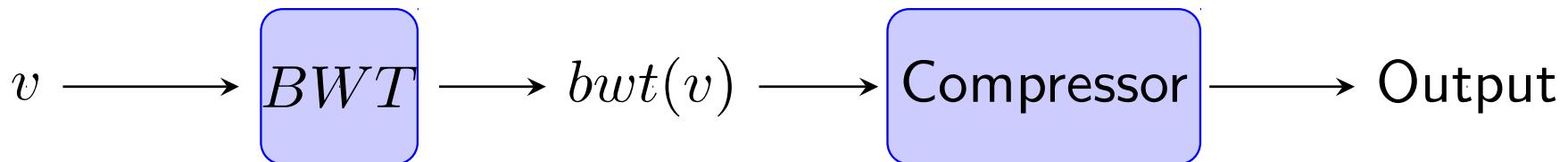
3. For all $i = 0, \dots, n-1$ and $i \neq I$, the symbol $F[i]$ follows $L[i]$ in the original word.
- + Example: $\text{bwt}(v) = L = \text{mmihttsecaa}$ and $I = 6$
 - + v can be reconstructed from right to left.
 - + More in general, for $i=0$ to $n-1$, $v[n-1-i] = L[\text{LF}^i[I]]$, where $\text{LF}^0[x]=x$ and $\text{LF}^i[x]=\text{LF}[\text{LF}^{i-1}[x]]$

F	L
0 a	m 0
1 a	m 1
2 c	i 2
3 e	h 3
4 h	t 4
5 i	t 5
6 m	s 6
7 m	e 7
8 s	c 8
9 t	a 9
10 t	a 10

Evaluating the performance of BWT



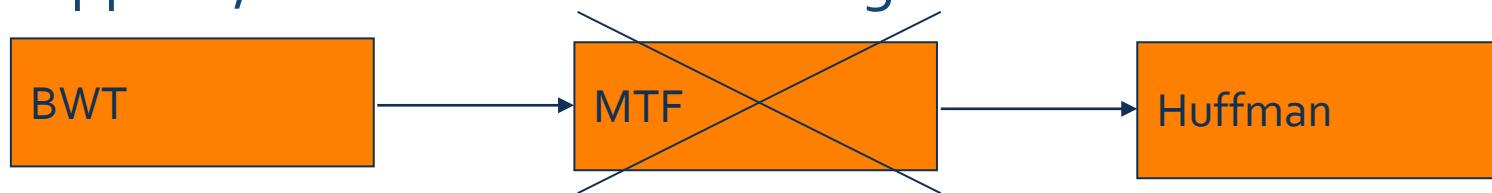
A BWT-based compression scheme



- + **Compressor** is, in general, an o-th order compressor, such as Huffman or Arithmetic Coding, that uses as preprocessing:
 - + Move-to-Front Encoding (it is a technique for symbol ranking)
 - + Run-Length Encoding

Approach proposed by Burrows and Wheeler

- + Burrows and Wheeler suggested to process $bwt(s)$ by using Move-to-Front coding.
- + MTF coding is a *symbol ranking* technique, that produces a globally homogeneous text, and in general more compressible.
- + After MTF coding, a zero-order compression algorithm is applied, such as the Huffman algorithm or arithmetic coding.



PROBLEM: Is the MTF coding required?

ANSWER: No, Giancarlo e S. in 2003. Two years later, Ferragina et al. in 2005 provided a partition technique, but the implementation is not efficient.

Important issues

- + What is the compression ratio performance of BWT-based algorithms?
- + How is BWT efficiently computed, so that it can be used effectively as preprocessing of a compressor?
- + How can BWT be useful for the pattern matching problem?

Analysis of a BWT-based compressor

- + Since 1994, numerous BWT-based compression techniques have been introduced. The first analysis of the performance of such compressors was carried out in

G. Manzini. An analysis of the burrows-wheeler transform. Journal of the ACM, 48(3):407–430, 2001.

- + G. Manzini compares the compression ratio of BWT-based algorithms and the **empirical entropy** of the input.
The empirical entropy is defined in terms of the number of occurrences of each symbol or group of symbols. It is defined for each string without any probabilistic assumptions about the source, so it can be used in worst-case analysis.

0-order empirical entropy

- + It is based on the probability distribution implicitly defined by the input, i.e. if s is the input of size n on an alphabet with h symbols

$$H_0(s) = - \sum_{i=1}^h \frac{n_i}{n} \log \frac{n_i}{n}$$

- + We assume that $0 \log 0 = 0$
- + The value $|s|H_0(s)$ represents the size of the output of an ideal compressor that uses $-\log \frac{n_i}{n}$ bits to encode the i -th symbol of the alphabet.

o-order compressors

- + When a text is compressed by using **Huffman Coding o Arithmetic Coding** the output has a size close to $|s|H_o(s)$ bits.

o-order compressors

- + When a text is compressed by using **Huffman Coding o Arithmetic Coding** the output has a size close to $|s|H_o(s)$ bits.
- + In particular, one can prove that

$$|Huff(s)| \leq |s|H_0(s) + |s|$$

$$|Arit(s)| \leq |s|H_0(s) + \mu|s| \quad (\mu \approx 0.01)$$



It is the bound achieved by implementations using finite-precision arithmetic

k-th order empirical entropy

- + Zero-order entropy is often not accurate enough to measure the performance of a compressor.
- + For example, if we want to compress an English text, we would benefit both from knowledge of the frequencies but also from the fact that these frequencies may depend on the context.
- + If we consider a context of length k:

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

where w_s is a string containing all the symbols that follows w in s.

- + We can give an analogous definition by considering the symbols preceding w.
- + This means that $|s|H_k(s)$ represents a lower bound for a compressor that uses a memory of length k
- + $H_k(s)$ is a function that decreases as k increases, i.e., $H_{k+1}(s) \leq H_k(s)$
- + There exists also a notion of modified empirical entropy H_k^* that uses more accurate statistics (for instance, all the occurrence of the factors up to length k)

Why is BWT important?

- + Note that $bwt(s^R) = \bigcup_{w \in A^k} \pi(w_s)$
- + Moreover $H_0(w_s) = H_0(\pi(w_s))$

Theorem (Manzini) : Let $u = bwt(s^R)$, where s^R is the reverse of the text s . For each positive integer k , there exists a factorization u_1, u_2, \dots, u_m of u such that

$$H_k(s) = \frac{1}{|u|} \sum_{i=1}^m |u_i| H_0(u_i)$$

bwt Compression (Manzini, 2001)

Let **Order- α** a generic α -order compressor, i.e. we assume that there exists a constant μ such that for each string s we have that

$$|Order_0(s)| \leq |s|H_0(s) + \mu|s|.$$

Let $bwt_\alpha(s)$ be the output of the compressor $Order_0(mtf(bw(s)))$

Theorem: For each k :

$$|bwt_0(s)| \leq 8|s|H_k(s) + \left(\mu + \frac{2}{25}\right)|s| + h^k(2h \log h + 9)$$

(h =size of the alphabet)

Run length encoding

- + RLE is a technique frequently used in image compression, i.e. when the input contains long runs of the same symbols.
- + A run of equal d symbols of length m is transmitted by coding the length m of the run, i.e., by encoding m in binary, where d represents 1.
- + For instance, dddd->dod
- + Some version of RLE use the encoding of the pair $(d,5)$, i.e., $(d,101)$
- + Other versions encode the number in binary with an alphabet $\{0,1\}$ different from the other symbols and, when encoding in binary, discard the most significant bit.

bwt Compression (Manzini, 2001)

Let **Order- α** a generic α -order compressor, i.e. we assume that there exists a constant μ such that for each string s we have that

$$|Order_0(s)| \leq |s|H_0(s) + \mu|s|.$$

Let $bwt_{RL}(s)$ be the output of the compressor

$$Order_0(rle(mtf(bw(s))))$$

Theorem: For each k :

$$|bwt_{RL}(s)| \leq (5 + 3\mu)|s|H_k^*(s) + g_k$$

where g_k is a constant only dependent on k and h
(h =taglia alfabeto)

Runtime of a BWT-based compressor

- + From the point of view of the time complexity of the algorithm, the bottleneck for BWT is the sorting of cyclic rotations.
- + How long does this step take?
- + How do you efficiently sort n strings of length n ?

How to compute the Burrows-Wheeler Transform by sorting suffixes

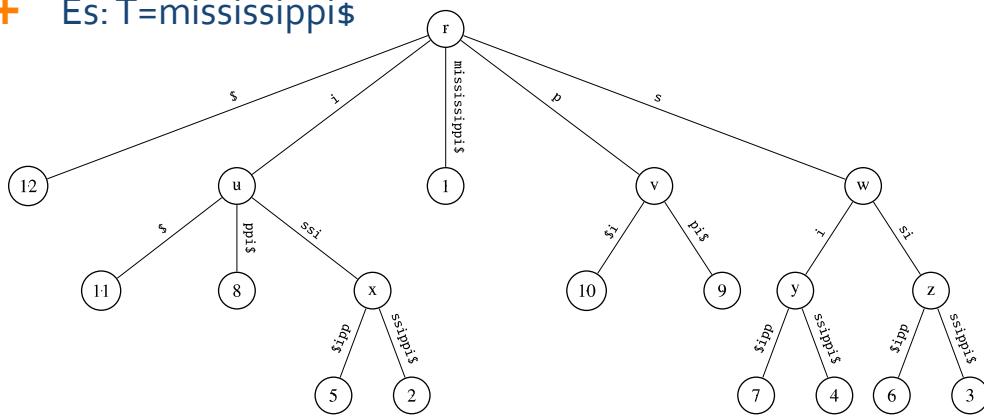
- + If we append an end-of-string to the text we reduce the problem of sorting cyclic rotations to the sorting of suffixes
- + For instance, let $T=abaaba\$$

\$ a b a a b a
a \$ a b a a b
a a b a \$ a b
a b a \$ a b a
a b a a b a \$
b a \$ a b a a
b a a b a \$ a

6	\$
5	a \$
2	a a b a \$
3	a b a \$
0	a b a a b a \$
4	b a \$
1	b a a b a \$

Suffix tree and suffix array

- + Let $T = T[1..n]$ a text. For $i \in [1..n]$, let T_i be the suffix $T[i..n]$
- + We denote by $\text{Suff}(T)$ the set $\{T_i \mid i \in [1..n]\}$ of all suffixes of T .
- + Suffix tree (ST) and suffix array (SA) are data structures to store $\text{Suff}(T)$.
 - + Suffix tree is a compact tree to represent $\text{Suff}(T)$
 - + Suffix array is a sorted array for $\text{Suff}(T)$.
- + Es: $T = \text{mississippi\$}$



Text suffixes	Indexes	Sorted Suffixes	SA
mississippi\$	1	\$	12
ississippi\$	2	i\$	11
ssissippi\$	3	ippi\$	8
sissippi\$	4	issippi\$	5
issippi\$	5	ississippi\$	2
ssippi\$	6	mississippi\$	1
sippi\$	7	pi\$	10
ippi\$	8	ppi\$	9
ppi\$	9	sippi\$	7
pi\$	10	sissippi\$	4
i\$	11	ssippi\$	6
\$	12	ssissippi\$	3

Suffix tree and suffix array

- + Both suffix tree and suffix array have linear size, since the compact tree has $O(n)$ nodes and an array has n entries.
- + However, the suffix array is more convenient as it occupies $|int| n$ bytes, usually $4n$ bytes! The suffix tree typically occupies about $16n$ bytes.
- + We will see how to construct efficiently the suffix array of a text.

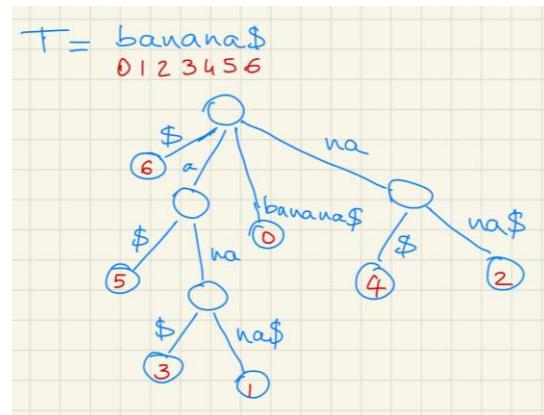
Suffix Tree and SUFFIX ARRAY: Compared space complexity

- + If you have a text that is a 10Gb genome, the suffix tree would take up 200 Gb
- + Suffix arrays are a more convenient structure

Suffix array

- + It is a data structure introduced in
 - + Manber and Myers: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing, 22(5), 1993, 935–948.
- + More formally, the suffix array SA of a text T[0,..n) is an array of integers such that $SA[i]=j$ means that the suffix $T[j,n)$ of T is the i-th suffix (has rank i) in the list of lexicographically sorted suffixes of T.
- + Example: T=banana\$

i	$SA[i]$	$T_{SA[i]}$
0	6	\$
1	5	a\$
2	3	ana\$
3	1	anana\$
4	0	banana\$
5	4	na\$
6	2	nana\$



The leaves of a subtree represent a range of consecutive positions in the suffix array, for instance $T(a)=SA[1,3]$

How to compute BWT efficiently?

- + If the input word has length n , we have to sort n strings of length n (the cyclic rotations).
- + A common strategy consists in adding a \$-symbol at the end of the input word. Sorting the conjugates is equivalent to sorting the suffixes of $v\$$.

Actually, we compute the BWT of a different word.

EXAMPLE:
 $v=caabra$

L						
↓						
\$	c	a	a	b	r	a
a	\$	c	a	a	b	r
a	a	b	r	a	\$	c
a	b	r	a	\$	c	a
b	r	a	\$	c	a	a
I → c	a	a	b	r	a	\$
	r	a	\$	c	a	a

$M_{lex}(caabra\$)$

It is different from

L						
↓						
a	a	b	r	a	c	r
a	b	r	a	c	a	a
a	c	a	a	b	r	
b	r	a	c	a	a	
c	a	a	b	r	a	
r	a	c	a	a	b	

$M_{lex}(caabra)$

...but it is easier to compute!

If we come back to our example...

$$BWT[i] = \begin{cases} T[SA[i] - 1] & \text{if } SA[i] > 0 \\ \$ & \text{if } SA[i] = 0 \end{cases}$$

- + T=abaaba\$ BWT(T)=abba\$aa

\$	a	b	a	a	b	a
a	\$	a	b	a	a	b
a	a	b	a	\$	a	b
a	b	a	\$	a	b	a
a	b	a	a	b	a	\$
b	a	\$	a	b	a	a
b	a	a	b	a	\$	a

6	\$
5	a \$
2	a a b a \$
3	a b a \$
0	a b a a b a \$
4	b a \$
1	b a a b a \$

Running time of BWT computation

- + The execution time depends on the running time of the suffix array construction.
- + It is linear if we use linear algorithms to construct the suffix array of a text.
- + The first linear algorithm for constructing a suffix array was provided in 2003, only about 10 years after the introduction of BWT (this lecture!).

SUFFIX ARRAY CONSTRUCTION

- + Suffix array construction: survey
 - + Puglisi, Smyth, Turpin: A taxonomy of suffix array construction algorithms. ACM Computing Surveys, 39(2), Article 4, 2007.
- + The suffix array of a text can be constructed in linear time (first papers appeared in 2003)
 - + difference cover sampling (DC3):
 - Kärkkäinen, Sanders and Burkhardt: Linear work suffix array construction. Journal of the ACM, 53(6), 2006, 918–936.
 - + induced sorting (SA-IS):
 - Ko and Aluru: Space efficient linear time construction of suffix arrays. Journal of Discrete Algorithms, 3(2), 2005, 143–156.
 - Nong, Zhang and Chan: Two efficient algorithms for linear time suffix array construction. IEEE Transactions on Computers, 60(10), 2011, 1471–1484.
- + Not recursive approach
 - Uwe Baier: Linear-time Suffix Sorting - A New Approach for Suffix Array Construction. CPM 2016: 23:1-23:12

Recursive Suffix array construction

- + DC₃ and SAIS algorithm construct the suffix array by using sampling strategies.
- + We can assume that the alphabet of the text T [0..n) is [1..n] (or characters corresponding to integers) and that T [n] = o (or \$)
- + The outline of the algorithms is:
 1. Choose a subset C ⊂ [0..n].
 2. Sort the set Suff_C. This is done as follows:
 - + (a) Construct a reduced string R of length |C|, whose characters are order preserving names of text factors starting at the positions in C.
 - + (b) Construct the suffix array of R recursively.
 3. Sort the set Suff(T) using the order of Suff_C.
- + Assume that
 - + |C| ≤ αn for a constant α < 1, and
 - + excluding the recursive call, all steps in the algorithm take linear time.
- + Then the total time complexity can be expressed as the recurrence $t(n) = O(n) + t(\alpha n)$, whose solution is $t(n) = O(n)$.

DC3 algorithm

- + The idea was presented in Juha Karkkainen and Peter Sanders. Simple linear work suffix array construction. In Procs of the International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science vol. 2791, Springer, 943–955, 2003.

The three steps

Step 1. Construct the suffix array $SA^{1,2}$ limited to the suffixes starting at positions

$$P_{1,2} = \{i : i \bmod 3 = 1, \text{ or } i \bmod 3 = 2\}:$$

- + Build a special string $T^{1,2}$ of length $(2/3)n$ which compactly encodes all suffixes of T starting at positions $P_{1,2}$
- + Build recursively the suffix-array SA' of $T^{1,2}$
- + Derive the suffix-array $SA^{1,2}$ from SA'

Step 2. Construct the suffix array SA^0 of the remaining suffixes starting at positions

$$P_0 = \{i : i \bmod 3 = 0\}:$$

- + For every $i \in P_0$, represent suffix $T[i,n]$ with a pair $\langle T[i], \text{pos}(i+1) \rangle$, where it is $i+1 \in P_{1,2}$.
- + Assume to have pre-computed the array $\text{pos}[i+1]$ which provides the position of the $(i+1)$ -th text suffix $T[i+1, n]$ in $SA^{1,2}$
- + Radix-sort the above $O(n)$ pairs

Step 3. Merge the two suffix arrays $SA^{1,2}$ and SA^0 into one

- + This is done by deploying the decomposition which ensures a constant-time lexicographic comparison between any pair of suffixes

Example and more details on STEP 1

- + Let us consider the text $T[0,11]=\text{mississippi\$}$.
- + The suffix array of T is $SA = [11 \ 10 \ 7 \ 4 \ 1 \ 0 \ 9 \ 8 \ 6 \ 3 \ 5 \ 2]$ (we will construct it!)
- + Let us consider $P_{1,2}=\{1,2,4,5,7,8,10,11\}$ and $P_0=\{0,3,6,9\}$
- + STEP 1. To efficiently obtain $SA^{1,2}$, we reduce the problem to the construction of the suffix array for a string $T^{1,2}$ of length about $2/3n$. An elegant solution consists of considering the two text suffixes $T[1, n]$ and $T[2, n]$, pad them with the special symbol $\$$ in order to have multiple-of-three length, and then decompose the resulting strings into triplets of characters
 $T[1, \cdot] = [t_1, t_2, t_3][t_4, t_5, t_6][t_7, t_8, t_9] \dots$ and $T[2, \cdot] = [t_2, t_3, t_4][t_5, t_6, t_7][t_8, t_9, t_{10}] \dots$
 $T[1, \cdot] = [\text{i s s}][\text{i s s}][\text{i p p}][\text{i \$ \$}] \quad T[2, \cdot] = [\text{s s i}][\text{s s i}][\text{p p i}][\text{\$ \$ \$}]$
- + Construct $R = [\text{i s s}][\text{i s s}][\text{i p p}][\text{i \$ \$}][\text{s s i}][\text{s s i}][\text{p p i}][\text{\$ \$ \$}]$
- + PROPERTY: *Every suffix $T[i, n]$ starting at a position $i \in P_{1,2}$, can be put in correspondence with a suffix of R consisting of an integral sequence of triplets. In particular, if $i \bmod 3 = 2$ then the text suffix coincides exactly with a suffix of R ; if $i \bmod 3 = 1$, then the text suffix is prefix of a suffix of R which terminates with special symbol $\$$.*
- + We encode such triplets via integers (lexicographic naming) by radix sorting the triplets in R and associating to each distinct triplet its rank in the lexicographic order.

Example and more details on STEP 1

- + Sorted triples and sorted ranks by radix sort in $O(n)$ time

$$[\$ \$ \$] [i \$ \$] [i p p] [i s s] [i s s] [p p i] [s s i] [s s i]$$
$$\begin{matrix} 0 & 1 & 2 & 3 & 3 & 4 & 5 & 5 \end{matrix}$$

- + $T^{1,2}$ string of ranks

$$R = [i s s] [i s s] [i p p] [i \$ \$] [[s s i] [s s i] [p p i] [\$ \$ \$]]$$
$$\begin{matrix} 3 & 3 & 2 & 1 & 5 & 5 & 4 & 0 \end{matrix}$$

- + The lexicographic comparison between suffixes of R (aligned to the triplets) equals the lexicographic comparison between suffixes of $T^{1,2}$
- + If all symbols in $T^{1,2}$ are different, then we do not need to recurse because suffixes can be sorted by looking just at their first characters. Otherwise, we apply recursively Step 1 to the string $T^{1,2}$
- + We derive the suffix-array $(7, 3, 2, 1, 0, 6, 5, 4)$.
- + We can turn this suffix array into $SA^{1,2}$ by turning the positions in $T^{1,2}$ into positions in T . This can be done via simple arithmetic operations, given the layout of the triplets in $T^{1,2}$, and obtains in our running example the suffix array $SA^{1,2} = (11, 10, 7, 4, 1, 8, 5, 2)$.

EXAMPLE and MORE details on Step 2

- + We decompose a suffix $T[i, n]$ as composed by its first character $T[i]$ and its remaining suffix $T[i + 1, n]$.
- + Since $i \in P_o$, the next position $i + 1 \in P_{1,2}$, and thus the suffix $T[i + 1, n]$ occurs in $SA^{1,2}$.
- + We can then encode the suffix $T[i, n]$ with a pair of integers $\langle T[i], pos(i + 1) \rangle$, where $pos(i+1)$ denotes the lexicographic rank in $SA^{1,2}$ of the suffix $T[i+1, n]$. We set $pos(n + 1) = 0$

0 1 2 3 4 5 6 7 8 9 10 11

- + Since $T = mississippi \$$ and $SA^{1,2} = (11, 10, 7, 4, 1, 8, 5, 2)$ then we have the following pairs corresponding to the positions of $P_o = \{0, 3, 6, 9\}$:

0 = <m, 4> 3 = <s, 3> 6 = <s, 2> 9 = <p, 1>

- + We sort the pair by applying radix sort in $O(n)$ time:

0 < 9 < 6 < 3

EXAMPLE and MORE details on Step 3

Recall that $T = mississippi \$$
0 1 2 3 4 5 6 7 8 9 10 11

We start from and $SA^{1,2} = (11, 10, 7, 4, 1, 8, 5, 2)$ and $SA^0 = (0, 9, 6, 3)$ constructed in Steps 1 and 2.

- + The final step merges the two sorted arrays SA^0 and $SA^{1,2}$ in $O(n)$ time by resorting an interesting observation which motivates the split in 2/3 and 1/3 the original array.
- + If we consider two suffixes $T[i, n] \in SA^0$ and $T[j, n] \in SA^{1,2}$, they belong to two different suffix arrays so we have no *lexicographic relation* known for them, and we cannot compare them character-by-character because this would incur in a very high cost.
- + Actually, we need to compare the characters *one or two characters* plus the lexicographic rank of its remaining suffix, similarly in Step 2.
 - + if $j \bmod 3 = 1$ then we compare $T[j, n] = T[j]T[j+1, n]$ against $T[i, n] = T[i]T[i+1, n]$. Both suffixes $T[j+1, n]$ and $T[i+1, n]$ occur in $SA^{1,2}$, so we can compare the pairs $\langle T[i], pos(i+1) \rangle$ and $\langle T[j], pos(j+1) \rangle$ in constant time.
 - + if $j \bmod 3 = 2$ then we compare $T[j, n] = T[j]T[j+1, n]T[j+2, n]$ against $T[i, n] = T[i]T[i+1, n]T[i+2, n]$. Both the suffixes $T[j+2, n]$ and $T[i+2, n]$ occur in $SA^{1,2}$. So we can compare the triples $\langle T[i], T[i+1], pos(i+2) \rangle$ and $\langle T[j], T[j+1], pos(j+2) \rangle$ in constant time.

EXAMPLE and MORE details on Step 3

Recall that $T=mississippi \$$
0 1 2 3 4 5 6 7 8 9 10 11

We start from and $SA^{1,2} = (11, 10, 7, 4, 1, 8, 5, 2)$ and $SA^0 = (0, 9, 6, 3)$ constructed in Steps 1 and 2.

- + In order to merge the arrays we need to store the pairs or the triples or both for each element of the arrays.

SA ⁰				SA ^{1,2}								
0	9	6	3		11	10	7	4	1	8	5	2
$\langle M, 4 \rangle$	$\langle P, 1 \rangle$	$\langle S, 2 \rangle$	$\langle S, 3 \rangle$		$\langle i, 0 \rangle$	$\langle i, 5 \rangle$	$\langle i, 6 \rangle$	$\langle i, 7 \rangle$				
$\langle M, i, 7 \rangle$	$\langle P, i, 0 \rangle$	$\langle S, i, 5 \rangle$	$\langle S, i, 6 \rangle$		$\langle \$, \$, -1 \rangle$				$\langle P, P, 1 \rangle$	$\langle S, S, 2 \rangle$	$\langle S, S, 3 \rangle$	

- + We obtain the $SA = [11 10 7 4 1 0 9 8 6 3 5 2]$

Running time of DC3

- + The time complexity can be modeled by the recurrence relation
$$T(n) = T(\frac{2}{3}n) + O(n)$$
- + The overall time complexity is $O(n)$
- + Note that we can use a different sampling, for instance $P_{2,0}$ and P_1 or with tuples having different length q (DIFFERENCE COVER)

Master Theorem

- + Sia $T(n)$ una funzione non decrescente che soddisfa la seguente relazione:

$$T(n) = a T(n/b) + f(n)$$

$$T(1) = c$$

dove $a \geq 1$, $b > 1$, $c > 0$. Se $f(n)$ è $\Theta(n^d)$ dove $d \geq 0$ allora

$$T(n) = \begin{cases} \Theta(n^d) & \text{Se } a < b^d \\ \Theta(n^d \log n) & \text{Se } a = b^d \\ \Theta(n^{\log_b a}) & \text{Se } a > b^d \end{cases}$$

EXERCISE

- + Applying the DC3 algorithm to the string $T = \text{yabbadabbado\$}$ in order to obtain the suffix array $SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)$

Solution

- + Apply the DC3 algorithm (Karkkainen and Sanders) to sort the suffixes of the string

$T = yabbadabbado\$\$$

We would like to find the suffix array $SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)$

STEP1: Positions $i \bmod 3 = 1$ and $i \bmod 3 = 2$

$\{1, 4, 7, 10, 2, 5, 8, 11\}$

We construct the string

$u = [abb][ada][bba][do\$][bba][dab][bad][o\$\$]$

We sort the triplets that are named by using their rank

$u' = 1 2 4 6 4 5 3 7 \rightarrow$ we append an eof symbol

$SA(u') = (0, 1, 6, 4, 2, 5, 3, 7)$

$T \quad y \ a \ b \ b \ a \ d \ a \ b \ b \ a \ d \ o \ \$ \ \$$

$i \quad 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13$

$Rank(S_i) \quad - \ 1 \ 4 \ - \ 2 \ 6 \ - \ 5 \ 3 \ - \ 7 \ 8 \ - \ 0$

$Rank(S_i) \quad 4 \quad 3 \quad 1 \quad 2 \quad 0$

Poiché $(\$, 0) < (a, 5) < (a, 7) < (b, 2) < (y, 1)$

$SA_{12} = (1, 4, 8, 2, 7, 5, 10, 11)$

$SA_0 = (12, 6, 9, 3, 0)$



Dictionary-based Compression

Friday April 21, 2023

Compression based on Dictionaries

- + The compression methods we have studied so far (Huffman Encoding, Arithmetic Encoding) are based on statistics of the source symbols.
- + When we don't have much a priori knowledge of the source characteristics and/or if statistical tests are impossible or not realisable at all, the problem of compression becomes considerably more complicated.
- + The class of compressors we consider here is based on dictionaries of codewords.
- + Dictionary-based (LZ-based) compressors represent a very wide class of compressors.
- + Large Text Compression Benchmark: One of the largest collections and benchmarks of lossless compressors, including approximately 200 programs.

• Statistical:	82 (42.0%)	} 93.7%
• LZ:	77 (39.4%)	
• BWT:	24 (12.3%)	
• Other:	12 (6.3%)	

<http://mattmahoney.net/dc/text.html>

A different strategy

- + Dictionary-based compression methods replace substrings of the message with a codeword that identifies that substring in a dictionary (**codebook**)
- + The dictionary contains a list of substrings and the associated codewords
- + Unlike methods based on symbol statistics, dictionary-based methods often use predetermined codewords rather than explicit probability distributions

Dictionary

- + More formally, a DICTIONARY for a text T is a set

$D = \{(f, c) \mid f \in F, c \in C\}$, where F is a subset of factors (or substrings) of the text T and C is the set of the correspondent codewords.

Example of static dictionary: digrams

- + One of the most common forms of static dictionary is the encoding of digrams. The dictionary consists of all the letters of the alphabet followed by some pairs of letters, called digrams.
- + For example, suppose we were to construct a dictionary of size 256 for digram coding of all printable ASCII characters. The first 95 entries of the dictionary would be the 95 printable ASCII characters. The remaining 161 entries would be the most frequently used pairs of characters.
- + The digram encoder reads a two-character input and searches the dictionary to see if this input exists in the dictionary.
 - + If it does, the corresponding index is encoded and transmitted.
 - + If it does not, the first character of the pair is encoded. The second character in the pair then becomes the first character of the next digram. The encoder reads another character to complete the digram, and the search procedure is repeated.

Example

- + Suppose we have the alphabet $A=\{a,b,c,d,r\}$. Based on the knowledge about the source we build the dictionary:

Code	Entry	Code	Entry
000	a	100	r
001	b	101	ab
010	c	110	ac
011	d	111	ad

- + How to encode the sequence **abracadabra** ?
- + The encoder reads the first two characters ab and checks to see if this pair of letters exists in the dictionary. It does and is encoded using the codeword 101. The encoder then reads the next two characters ra and checks to see if this pair occurs in the dictionary. It does not, so the encoder sends out the code for r, which is 100, then reads in one more character, c, to make the two-character pattern ac. This does exist in the dictionary and is encoded as 110. Continuing in this fashion, the remainder of the sequence is coded.
- + The output is: **101100110111101100000**

Static dictionaries

- + Another possibility is to use longer words in the dictionary, e.g. common words such as articles or prepositions or recurring words. These strings are the **phrases** of the dictionary
- + A dictionary with a predefined set of words does not normally allow for high compression
- + Performance is better if the dictionary is adapted to the source, i.e. if we lose independence from the source

Text-dependent dictionaries

- + For example, common phrases in a sports newspaper are rare in a book dealing with economics
- + To avoid the dictionary being inappropriate for the source we can build one for each message to be compressed....
- + but there is a considerable overhead to transmit and store it
 - + Deciding on the size of the dictionary in order to maximise compression is a very hard problem

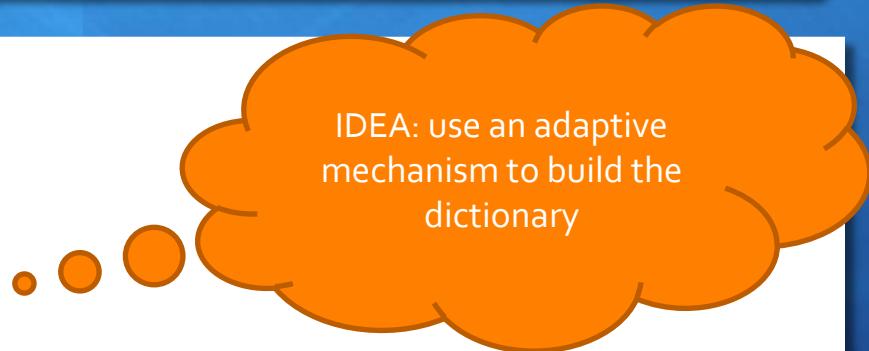
Dictionary-based compression



Abraham Lempel



Jacob Ziv



- ▶ They are the founders of a new compression paradigm.
 - ▶ Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.* **23**, 337–343 (1977)
 - ▶ Ziv, J., Lempel, A.: *Compression of individual sequences via variable-length coding*. *IEEE Trans. Inf. Theor.* **24**, 530–536 (1978)

The key-idea

- + It is possible to automatically construct a dictionary that includes the strings that have been viewed, up to the current step in the message to be compressed
- + Moreover, the preceding text constitutes an excellent dictionary since it is very likely to share the same language and style as the text yet to be seen
- + Basically all dictionary-based adaptive methods are based on one of the methods introduced by Lempel and Ziv.

The key-idea

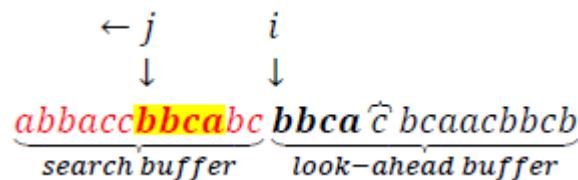
- + It is not necessary to transmit the dictionary since the decoder can construct it in the same way as the encoder did
- + The many variants of Lempel-Ziv methods differ in the way pointers are represented and in the limitations on the text strings to which these pointers can refer
- + The large amount of variants is also due to certain patents and disputes over them

The LZ77 family

- + Quite easy to implement
- + Fast decoding with little use of memory

LZ77 encoding

- + The encoding is done using a sliding window consisting of two parts: a **search buffer**, containing the portion already encoded, and a **look-ahead buffer**, containing the segment yet to be encoded.
- + The separation of the two buffers is achieved by using a pointer i which identifies the beginning of the look-ahead buffer. A second pointer j , starting with i , will examine the search buffer for **the longest sequence (if any) that is a prefix for the look-ahead buffer**.



- + The compressed message consists of a sequence of triplets (o, l, s)
 - o is the distance between i and j (offset)
 - l is the length of the prefix (phrase)
 - s is the symbol appearing on the look-ahead buffer after the prefix

The encoding process

INPUT	OUTPUT
<u>babbababbaabbaabaabaaa</u> <i>look-ahead buffer</i>	(0,0, b)
<u>b</u> <u>abbababbaabbaabaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a)
<u>ba</u> <u>bbbababbaabbaabaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b)
<u>babb</u> <u>ababbaabbaabaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b) (3,2, a)
<u>babbaba</u> <u>bbaabbaabaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b) (3,2, a) (5,3, a)
<u>babbababbaa</u> <u>bbaabbaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b) (3,2, a) (5,3, a) (4,4, b)
<u>babbababbaabbaab</u> <u>aa baaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b) (3,2, a) (5,3, a) (4,4, b) (3,5, a)

An example - encoding

- + alphabet {a,b}

ab a a b a b aabb

An example - encoding

- + alfabeto {a,b}

a b a a b a b aabb

<0,0,a>

An example - encoding

+ alfabeto {a,b}

a b a a b a b aabb

<0,0,a> <0,0,b>

An example - encoding

+ alfabeto {a,b}

ab a a b a b aabb

<0,0,a> <0,0,b> <2,1,a>

An example - encoding

+ alfabeto {a,b}

ab a a b a b aabb

<0,0,a> <0,0,b> <2,1,a> <3,2,b>

An example - encoding

+ alfabeto {a,b}

ab a a b a b aabb

<0,0,a> <0,0,b> <2,1,a> <3,2,b> <5,4,b>

The decoding process

INPUT	OUTPUT
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$ b$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$b a$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$\widehat{b} \, a \, _{\widehat{b}} \, b$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$b \, \widehat{ab} \, b \, _{\widehat{ab}} \, a$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$ba \, \widehat{bba} \, ba \, _{\widehat{bba}} \, a$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$babbaba \, \widehat{bbaa} \, _{\widehat{bbaa}} \, b$

The last step

(0,0,b) (0,0,a) (2,1,b) (3,2,a) (5,3,a) (4,4,b) (3,5,a)

babbababbaabb aab | aa baa a

- + The algorithm does not copy the prefix in a whole block but symbol by symbol

aab?? → a ab a? → a a b a a → aa b aa b → aab a a b a → aab a a ba a

- + If the identified prefix terminates at the look-ahead buffer boundary and, within the look-ahead buffer, this prefix repeats from the beginning (even if only in part), then encoding can be performed that takes this repetition into account by means of an offset less than the length of the prefix itself.

An example - decoding

+ <0,0,x><0,0,y><2,1,z><2,1,x><5,3,z><6,3,z><5,2,z>

An example - decoding

+ <0,0,x><0,0,y><2,1,z><2,1,x><5,3,z> <6,3,z><5,2,z>

SOL. x y xz xx yxzz xxxyz zxz

Another example

+ <0,0,a><0,0,c><2,1,a><4,2,b><1,10,a>

+ Every character is available when needed

Another example

+ <0,0,a><0,0,c><2,1,a><4,2,b><1,10,a>

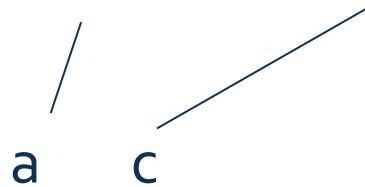
/

a

+ Every character is available when needed

Another example

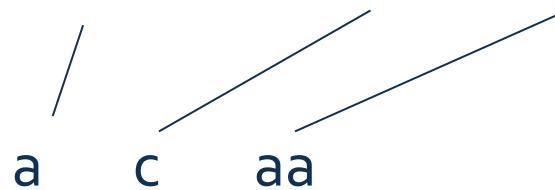
+ <0,0,a><0,0,c><2,1,a><4,2,b><1,10,a>



+ Every character is available when needed

Another example

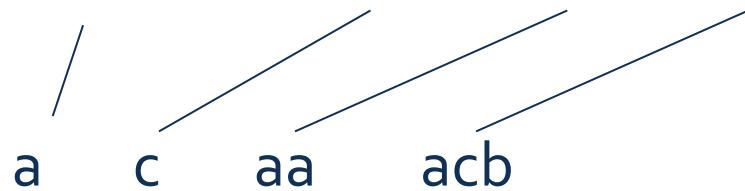
+ <0,0,a><0,0,c><2,1,a><4,2,b><1,10,a>



+ Every character is available when needed

Another example

+ $\langle 0,0,a \rangle \langle 0,0,c \rangle \langle 2,1,a \rangle \langle 4,2,b \rangle \langle 1,10,a \rangle$



+ Every character is available when needed

Another example

+ $\langle 0,0,a \rangle \langle 0,0,c \rangle \langle 2,1,a \rangle \langle 4,2,b \rangle \langle 1,10,a \rangle$

a c aa acb ??

+ Every character is available when needed

Another example

+ $\langle 0,0,a \rangle \langle 0,0,c \rangle \langle 2,1,a \rangle \langle 4,2,b \rangle \langle 1,10,a \rangle$

a c aa acb bbbbbbbba

+ Every character is available when needed

Further details on LZ77

- + LZ77 algorithm places limitations on how far back a pointer can refer (i.e. on the length of the first component of the triple) and on the maximum size of the string referred to (i.e. on the length of the second component)
- + For example, in English text there is no gain in using a sliding windows of more than a few thousand characters

Further details on LZ77

- + At the same time, the length of the match is rarely over 16 characters, so the extra cost to allow longer match usually is not justified
- + Exercise: encode the sequence

010020\$0110\$\$0111

with a sliding window of 7 symbols and a maximal match length of 3.
Calculate the compression ratio

SOL. <0,0,0><0,0,1><2,1,0><0,0,2><2,1,\$><7,2,1><5,2,\$>,<6,3,1>
0000000 0000001 0100100 0000010 0100111 1111001 1011011 1101101
 $C=(17*2)/7*8=0.607 < 1!!!$

LZ77 - encoding

- + Encode the text $S[1..N]$ using LZ77, with a sliding window of W characters

$p=1$

WHILE $p < N$ {

 search for the longest match for $S[p\dots]$ in $S[p-W \dots p-1]$.

 Suppose the match occurs at position $p-m$, with length l

 output the triple $< p-m, l, S[p+l] >$

$p=p + l + 1$

}

LZ77 - decoding

- + Decode the text $S[1..N]$ using LZ77, with a sliding window of W characters

$p=1$

FOREACH triple $\langle f, l, c \rangle \{$

$S[p \dots p + l - 1] = S[p - f \dots p - f + l - 1]$

Suppose the match occurs at position $p-f$, with length l

$S[p+l] = c$

$p=p+l+1$

}

LZ algorithms

- + Ziv and Lempel, in their 1977 paper, described their contribution as follows :
“[...] universal coding scheme which can be applied to any discrete source and whose performance is comparable to certain optimal fixed code book scheme designed for completely specified sources [...]”
- + That is, unlike Huffman encoding and arithmetic encoding, information on the characteristics of the source was implicitly deduced from the repetitiveness of the text.

LZ77: pro and contra

- + LZ77's compressor is based on a sliding window (search buffer) $W[1:w]$ which contains a portion of the input sequence that has been processed so far, typically consisting of the last w characters, and a look-ahead buffer B which contains the suffix of the text still to be processed.
 - + For instance, in the figure $W = \text{aabbababb}$ of size 9, and $B = \text{baababaabbaa\$}$.
- $\leftarrow \dots \boxed{\text{aabbababb}} \text{ baababaabbaa\$} \rightarrow$
- + The compressor operates in two main stages: parsing and encoding.
 - + Parsing consists of transforming the input sequence S into a sequence of triples of integers (called phrases). Encoding turns these triples into a (compressed) bit stream by applying either a statistical compressor (i.e. Huffman or Arithmetic) to each triplet-component individually, or an integer encoding scheme.
 - + LZ77 searches for the longest prefix of B which occurs as a substring of WB . The size of the dictionary is quadratic in W 's length.
 - + Let us consider the following parsing steps

$\boxed{\text{aabbabab}}$	\Rightarrow	$\langle 0, 0, a \rangle$
$\boxed{a\text{abbabab}}$	\Rightarrow	$\langle 1, 1, b \rangle$
$\boxed{\text{aab}\mid\text{babab}}$	\Rightarrow	$\langle 1, 1, a \rangle$
$\boxed{\text{aabba}\mid\text{bab}}$	\Rightarrow	$\langle 2, 3, EOF \rangle$

LZ77: pro and contra

- + The decoding process is very fast.
- + The longer is the window W , the longer are possibly the phrases, the fewer is their number and thus possibly the shorter is the compressed output; but of course, in terms of compression time, the longer is the time to search for the longest copied substring
Vice versa, the shorter is W , the worse is the compression ratio but the faster is the compression time.

Storer and Szymanski

- + Storer and Szymanski in 1982 observed that in the parsing process two situations may occur: a longest match has been found, or it has not.
- + In the former case it is not reasonable to add the character following (third component in the triple), given that we anyway advance in the input sequence. In the latter case it is not reasonable to emit two o's (first two components in the triple) and thus waste one integer encoding. The simplest solution to these two inefficiencies is to always output a pair, rather than a triple, with the form: $(d, |\alpha|)$ or $\langle o, c \rangle$.
- + This variant of LZ77 is named LZss.

LZss: example

- + Lzss encoding for the sequence aabbabab.

aabbabab	\Rightarrow	$\langle 0, a \rangle$
a abbabab	\Rightarrow	$\langle 1, 1 \rangle$
aa bbabab	\Rightarrow	$\langle 0, b \rangle$
aab babab	\Rightarrow	$\langle 1, 1 \rangle$
aabb abab	\Rightarrow	$\langle 3, 2 \rangle$
aabbab ab	\Rightarrow	$\langle 2, 2 \rangle$

Variants of LZ77

- + The LZ77 has been gradually refined
 - + **First component of the triple:** it is useful to use variable length, assigning shorter codewords to recent matches (that are more common)
 - + **second component of the triple:** variable **length codes** that uses less bits to represent smaller numbers
 - + **third component of the triple second component of the triple** in some variants it is added only when needed (when?), with a 1-bit flag to indicate the presence or the absence of this third component
 - + Storer e Szymanski in 1982: variant LZss.

Gzip

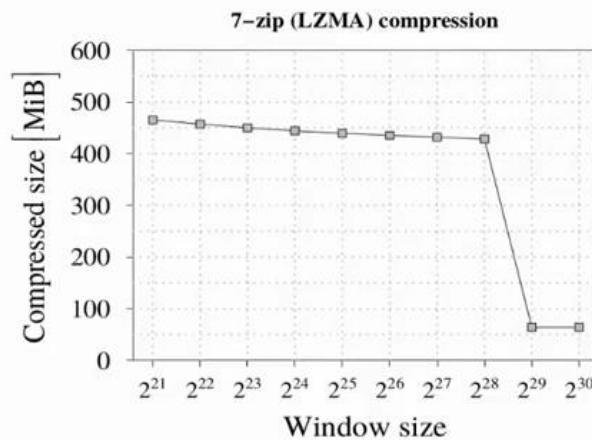
- + Gzip is a very effective variant of Lz77. It is distributed by the GNU project (gzip = GNU zip)
home page of the project gzip: www.gzip.org

Gzip

- + The key programming problem when implementing LZ77 is the fast search for the longest prefix α of B which repeats in W.
- + A brute-force algorithm that checks the occurrence of every prefix of B in W, via a linear backward scan, would be very time-consuming and thus unacceptable for compressing Gbs files. Fortunately, this process can be accelerated by using a suitable data structure.
- + Gzip, the most popular implementation of LZ77, uses a hash table to determine α and find its previous occurrence in W.
- + The idea is to store in the hash table all 3-grams occurring in W, namely all triplets of contiguous characters, by using as key the 3-gram and as its satellite data the position in B where that 3-gram occurs. Since a 3-gram may repeat multiple times in W, the hash table saves for a given 3-gram all of its multiple occurrences, sorted by increasing position in S. This way, when W shifts to the right because of the emission of the pair $\langle d, p \rangle$, the hash table can be updated by deleting the '3-grams starting at $W[1; p]$, and inserting the '3-grams starting at $B[1; p]$.
- + The search for α is implemented as follows
 - + first, the 3-gram $B[1; 3]$ is searched in the hash table. If it does not occur, then Gzip emits the phrase $\langle o, c \rangle$, and the parsing advances of one single character. Otherwise, it determines the list L of occurrences of $B[1; 3]$ in W.
 - + second, for each position i in L (which is expressed as absolute position in S), the algorithm compares character-by-character $S[i; n]$ against B in order to compute their longest common prefix. At the end, the position $i' \in L$, sharing this longest common prefix is determined, as well as it is found α
 - + finally, let p be the current position of B in S, the algorithm emits the pair $\langle p-i', \alpha \rangle$ and advances the parsing of α positions.
- + Triplets are encoded with Huffman on two alphabets: that of lengths and characters, and that of distances. $\langle o, c \rangle$ is only represented with the Huffman encoding of c, $\langle d, l \rangle$ is represented in reverse. If the decoder finds c, it knows to expect a character, if it encounters l it expects an integer.

On the size of the window

- + Every variant of LZ77 has limitations on W. For instance: gzip $W=32 \times 2^{10}$, 7-zip: $W \leq 2^{30}$.
- + Experiment with a highly repetitive file consisting of similar DNA sequences.



Lab exercises

- + Portfolio: Write a program in a programming language of your choice, that given a text T outputs the LZ₇₇ encoding and the LZss encoding of the text. A parameter of the program should be W , the length of the search buffer.
 - + Compare the number of triplets denoted by z produced by LZ₇₇(T) and the number of the equal-letter runs denoted by r produced by the BWT(T). Test experimentally whether the following relations hold:
 - + $r = O(z \log^2 n)$
 - + $z = O(r \log n)$,
- where n is the length of the input text

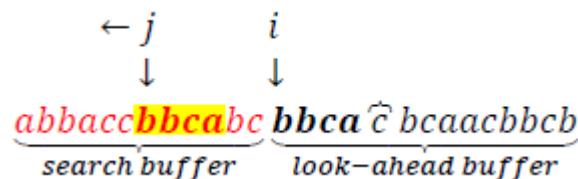


Dictionary-based Compression (part 2)

Friday April 28, 2023

LZ77 encoding

- + The encoding is done using a sliding window consisting of two parts: a **search buffer**, containing the portion already encoded, and a **look-ahead buffer**, containing the segment yet to be encoded.
- + The separation of the two buffers is achieved by using a pointer i which identifies the beginning of the look-ahead buffer. A second pointer j , starting with i , will examine the search buffer for **the longest sequence (if any) that is a prefix for the look-ahead buffer**.



- + The compressed message consists of a sequence of triplets (o, l, s)
 - o is the distance between i and j (offset)
 - l is the length of the prefix (phrase)
 - s is the symbol appearing on the look-ahead buffer after the prefix

The encoding process

INPUT	OUTPUT
<u>babbababbaabbaabaabaaa</u> <i>look-ahead buffer</i>	(0,0, b)
<u>b</u> <u>abbababbaabbaabaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a)
<u>ba</u> <u>bbbababbaabbaabaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b)
<u>babb</u> <u>ababbaabbaabaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b) (3,2, a)
<u>babbaba</u> <u>bbaabbaabaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b) (3,2, a) (5,3, a)
<u>babbababbaaa</u> <u>bbaabbaabaaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b) (3,2, a) (5,3, a) (4,4, b)
<u>babbababbaabbaab</u> <u>aa baaa</u> <i>search buffer</i> <i>look-ahead buffer</i>	(0,0, b) (0,0, a) (2,1, b) (3,2, a) (5,3, a) (4,4, b) (3,5, a)

The decoding process

INPUT	OUTPUT
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$ b$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$b a$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$\widehat{b} \, a \, _{\widehat{b}} \, b$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$b \, \widehat{ab} \, b \, _{\widehat{ab}} \, a$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$ba \, \widehat{bba} \, ba \, _{\widehat{bba}} \, a$
$(0,0, b)$ $(0,0, a)$ $(2,1, b)$ $(3,2, a)$ $(5,3, a)$ $(4,4, b)$ $(3,5, a)$	$babbaba \, \widehat{bbaa} \, _{\widehat{bbaa}} \, b$

LZ77 - encoding

- + Encode the text $S[1..N]$ using LZ77, with a sliding window of W characters

$p=1$

WHILE $p < N$ {

 search for the longest match for $S[p\dots]$ in $S[p-W \dots p-1]$.

 Suppose the match occurs at position $p-m$, with length l

 output the triple $< p-m, l, S[p+l] >$

$p=p + l + 1$

}

LZ77 - decoding

- + Decode the text $S[1..N]$ using LZ77, with a sliding window of W characters

$p=1$

FOREACH triple $\langle f, l, c \rangle \{$

$S[p \dots p + l - 1] = S[p - f \dots p - f + l - 1]$

Suppose the match occurs at position $p-f$, with length l

$S[p+l] = c$

$p=p+l+1$

}

LZ algorithms

- + Ziv and Lempel, in their 1977 paper, described their contribution as follows :
“[...] universal coding scheme which can be applied to any discrete source and whose performance is comparable to certain optimal fixed code book scheme designed for completely specified sources [...]”
- + That is, unlike Huffman encoding and arithmetic encoding, information on the characteristics of the source was implicitly deduced from the repetitiveness of the text.

LZ77: pro and contra

- + LZ77's compressor is based on a sliding window (search buffer) $W[1:w]$ which contains a portion of the input sequence that has been processed so far, typically consisting of the last w characters, and a look-ahead buffer B which contains the suffix of the text still to be processed.
- + For instance, in the figure $W = \text{aabbababb}$ of size 9, and $B = \text{baababaabbaa\$}$.

$\leftarrow \dots \boxed{\text{aabbababb}} \text{ baababaabbaa\$} \rightarrow$

- + The compressor operates in two main stages: parsing and encoding.
- + Parsing consists of transforming the input sequence S into a sequence of triples of integers (called phrases). Encoding turns these triples into a (compressed) bit stream by applying either a statistical compressor (i.e. Huffman or Arithmetic) to each triplet-component individually, or an integer encoding scheme.
- + LZ77 searches for the longest prefix of B which occurs as a substring of WB . The size of the dictionary is quadratic in W 's length.
- + Let us consider the following parsing steps

$\boxed{\text{aabbabab}}$	\Rightarrow	$\langle 0, 0, a \rangle$
$\boxed{a\text{abbabab}}$	\Rightarrow	$\langle 1, 1, b \rangle$
$\boxed{\text{aab}\mid\text{babab}}$	\Rightarrow	$\langle 1, 1, a \rangle$
$\boxed{\text{aabba}\mid\text{bab}}$	\Rightarrow	$\langle 2, 3, EOF \rangle$

LZ77: pro and contra

- + The decoding process is very fast.
- + The longer is the window W , the longer are possibly the phrases, the fewer is their number and thus possibly the shorter is the compressed output; but of course, in terms of compression time, the longer is the time to search for the longest copied substring
Vice versa, the shorter is W , the worse is the compression ratio but the faster is the compression time.

Storer and Szymanski

- + Storer and Szymanski in 1982 observed that in the parsing process two situations may occur: a longest match has been found, or it has not.
- + In the former case it is not reasonable to add the character following (third component in the triple), given that we anyway advance in the input sequence. In the latter case it is not reasonable to emit two o's (first two components in the triple) and thus waste one integer encoding. The simplest solution to these two inefficiencies is to always output a pair, rather than a triple, with the form: $\langle d, | \alpha | \rangle$ or $\langle o, c \rangle$.
- + This variant of LZ77 is named LZss.

LZss: example

- + Lzss encoding for the sequence aabbabab.

aabbabab	\Rightarrow	$\langle 0, a \rangle$
a abbabab	\Rightarrow	$\langle 1, 1 \rangle$
aa bbabab	\Rightarrow	$\langle 0, b \rangle$
aab babab	\Rightarrow	$\langle 1, 1 \rangle$
aabb abab	\Rightarrow	$\langle 3, 2 \rangle$
aabbab ab	\Rightarrow	$\langle 2, 2 \rangle$

Gzip

- + Gzip is a very effective variant of Lz77. It is distributed by the GNU project (gzip = GNU zip)
home page of the project gzip: www.gzip.org

Gzip

- + The key programming problem when implementing LZ77 is the fast search for the longest prefix α of B which repeats in W.
- + A brute-force algorithm that checks the occurrence of every prefix of B in W, via a linear backward scan, would be very time-consuming and thus unacceptable for compressing Gbs files. Fortunately, this process can be accelerated by using a suitable data structure.
- + Gzip, the most popular implementation of LZ77, uses a hash table to determine α and find its previous occurrence in W.
- + The idea is to store in the hash table all 3-grams occurring in W, namely all triplets of contiguous characters, by using as key the 3-gram and as its satellite data the position in B where that 3-gram occurs. Since a 3-gram may repeat multiple times in W, the hash table saves for a given 3-gram all of its multiple occurrences, sorted by increasing position in S. This way, when W shifts to the right because of the emission of the pair $\langle d, p \rangle$, the hash table can be updated by deleting the '3-grams starting at $W[1; p]$, and inserting the '3-grams starting at $B[1; p]$.
- + The search for α is implemented as follows
 - + first, the 3-gram $B[1; 3]$ is searched in the hash table. If it does not occur, then Gzip emits the phrase $\langle o, c \rangle$, and the parsing advances of one single character. Otherwise, it determines the list L of occurrences of $B[1; 3]$ in W.
 - + second, for each position i in L (which is expressed as absolute position in S), the algorithm compares character-by-character $S[i; n]$ against B in order to compute their longest common prefix. At the end, the position $i' \in L$, sharing this longest common prefix is determined, as well as it is found α
 - + finally, let p be the current position of B in S, the algorithm emits the pair $\langle p-i', \alpha \rangle$ and advances the parsing of α positions.
- + Triplets are encoded with Huffman on two alphabets: that of lengths and characters, and that of distances. $\langle o, c \rangle$ is only represented with the Huffman encoding of c, $\langle d, l \rangle$ is represented in reverse. If the decoder finds c, it knows to expect a character, if it encounters l it expects an integer.

Lab exercises

- + Portfolio: Write a program in a programming language of your choice, that given a text T outputs the LZ₇₇ encoding and the LZss encoding of the text. A parameter of the program should be W , the length of the search buffer.
 - + Compare the number of triplets denoted by z produced by LZ₇₇(T) and the number of the equal-letter runs denoted by r produced by the BWT(T). Test experimentally whether the following relations hold:
 - + $r = O(z \log^2 n)$
 - + $z = O(r \log n)$,
- where n is the length of the input text

Family of LZ78 compressors

- it has restrictions on which substring can be referenced (but this avoids some inefficiency)
- decoding is slower than LZ77 and require more memory
- + does not have a window to limit how far back substring can be referenced. Note that LZ77 cannot have too large buffers, so it does not take full advantage of the regularity of the text to be compressed.
- + one of its variant, LZW, is widely used in many popular compression systems

Referentiable strings

- + The text prior to the current coding position is parsed in substrings, and only parsed phrases can be referenced
- + Previous phrases are numbered in sequence, and the output is a list of pairs

<previous phrase, next character>

- + This unseen combination is stored as a new phrase

An example

Phrases

a b a a b a b a a

Output

An example

a b a a b a b a a

Phrases

- o <null>

Output

An example

a b a a b a b a a

Phrases

- o <null>

Output

An example

a b a a b a b a a

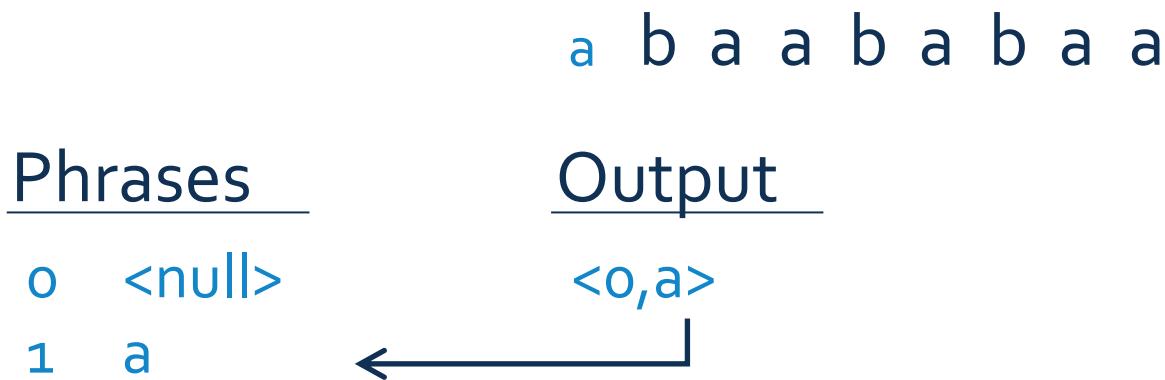
Phrases

o <null>

Output

<o,a>

An example



An example

a b a a b a b a a

Phrases

0 <null>
1 a

Output

<0,a>

An example

a b a a b a b a a

Phrases

0 <null>
1 a

Output

<o,a> <o,b>

An example

	a b a a b a b a a
<u>Phrases</u>	<u>Output</u>
0 <null>	<o,a> <o,b>
1 a	
2 b	

An example

a b a a b a b a a

Phrases

0 <null>

1 a

2 b

Output

<o,a> <o,b>

An example

a b a a b a b a a

Phrases

0 <null>

1 a

2 b

Output

<0,a> <0,b> <1,a>

An example

	a b a a b a b a a
<u>Phrases</u>	<u>Output</u>
0 <null>	<o,a> <o,b> <1,a>
1 a	
2 b	
3 aa	

An example

a b a a b a b a a

Phrases

0 <null>

1 a

2 b

3 aa

Output

<0,a> <0,b> <1,a>

An example

Phrases

0 <null>
1 a
2 b
3 aa

Output

a b a a b a b a a
<0,a> <0,b> <1,a> <2,a>

An example

	a b a a b a b a a	
<u>Phrases</u>		<u>Output</u>
0 <null>		<0,a> <0,b> <1,a> <2,a>
1 a		
2 b		
3 aa		
4 ba		

A large black bracket is drawn from the '<2,a>' entry in the 'Output' column to the 'ba' entry in the 'Phrases' column at row 4.

A horizontal arrow points from the 'ba' entry in the 'Phrases' column to the '4' index below it.

An example

Phrases

0 <null>
1 a
2 b
3 aa
4 ba

Output

a b a a b a b a a
<0,a> <0,b> <1,a> <2,a>

An example

a b a a b a b a a

Phrases

- 0 <null>
- 1 a
- 2 b
- 3 aa
- 4 ba

Output

<0,a> <0,b> <1,a> <2,a> <4,a>

An example

	a b a a b a b a a	
<u>Phrases</u>		<u>Output</u>
0 <null>		<0,a> <0,b> <1,a> <2,a> <4,a>
1 a		
2 b		
3 aa		
4 ba		
5 baa		

A horizontal arrow points from the bottom of the 'Phrases' column to the bottom of the 'Output' column. A vertical line connects the bottom of the 'Output' column to the bottom-right corner of the slide area.

An example

Phrases

0	<null>
1	a
2	b
3	aa
4	ba
5	baa

Output

<0,a>	<0,b>	<1,a>	<2,a>	<4,a>
-------	-------	-------	-------	-------



a b a a b a b a a

Only these phrases can be referred

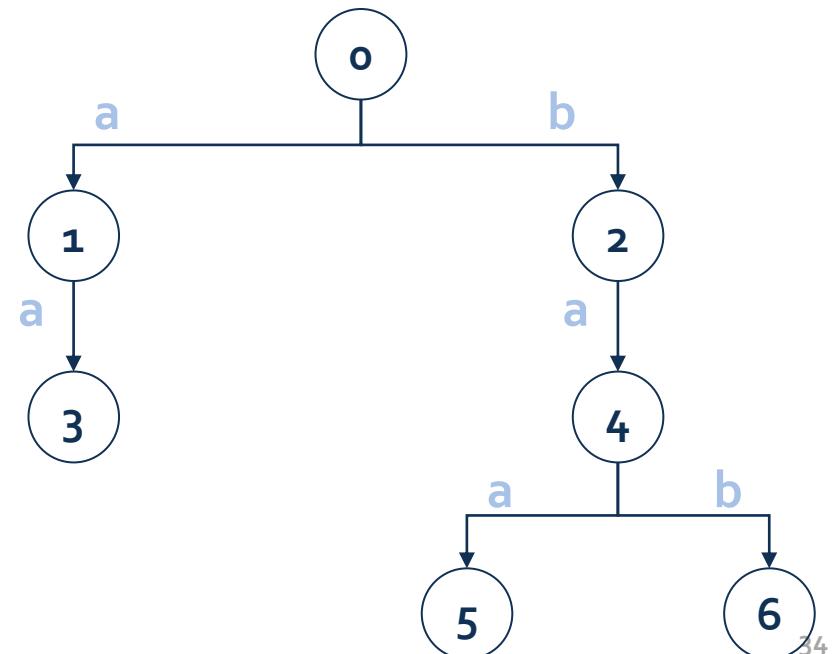
- This avoids the inefficiency of having more than one coded representation for the same string, as usual in LZ77

How to store the phrases

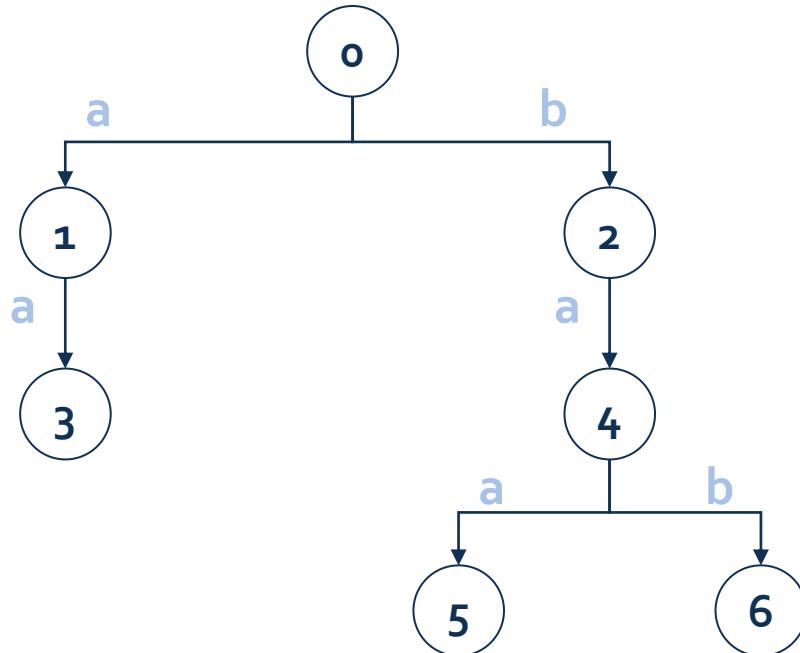
- It is crucial for the efficiency of the algorithm that the phrases are stored in a clever way
- It can be realized by using a trie

Phrases

0	<null>
1	a
2	b
3	aa
4	ba
5	baa



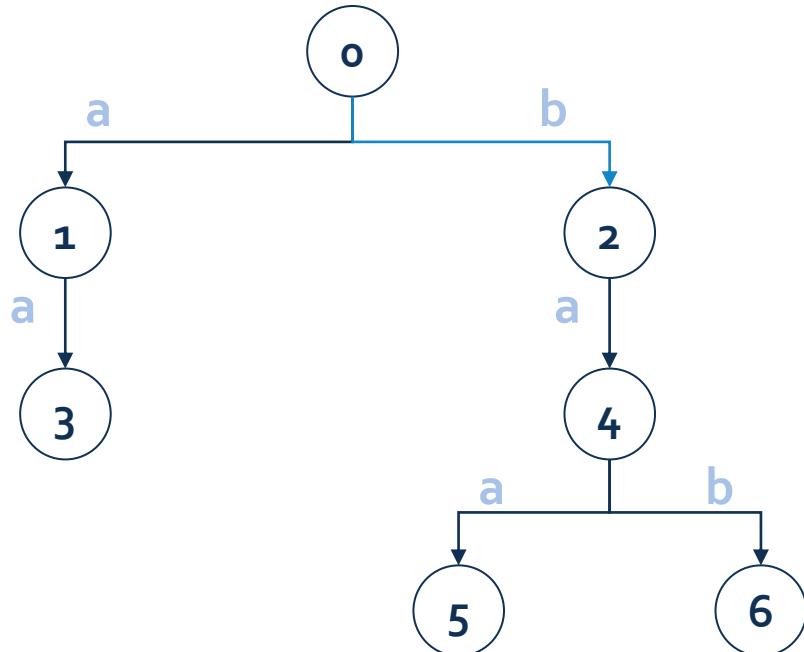
How to store the phrases



baab

- + The characters of each phrase give the path from the root to a leaf
- + The character to be encoded are used to traverse the trie until the end of the path
- + The last node contains the phrase number to output
- + A new node is added with next input character, to form a new phrase.

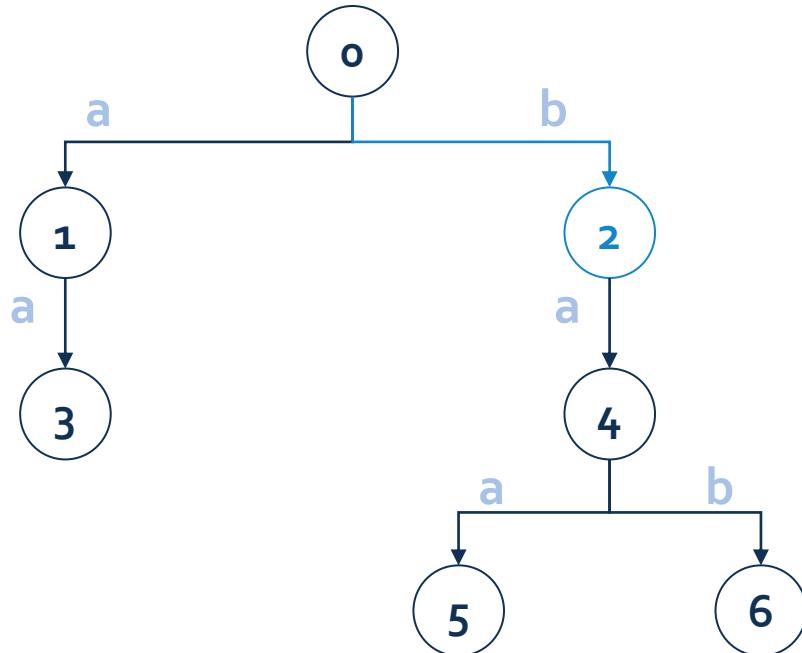
How to store the phrases



baab

- + The characters of each phrase give the path from the root to a leaf
- + The character to be encoded are used to traverse the trie until the end of the path
- + The last node contains the phrase number to output
- + A new node is added with next input character, to form a new phrase.

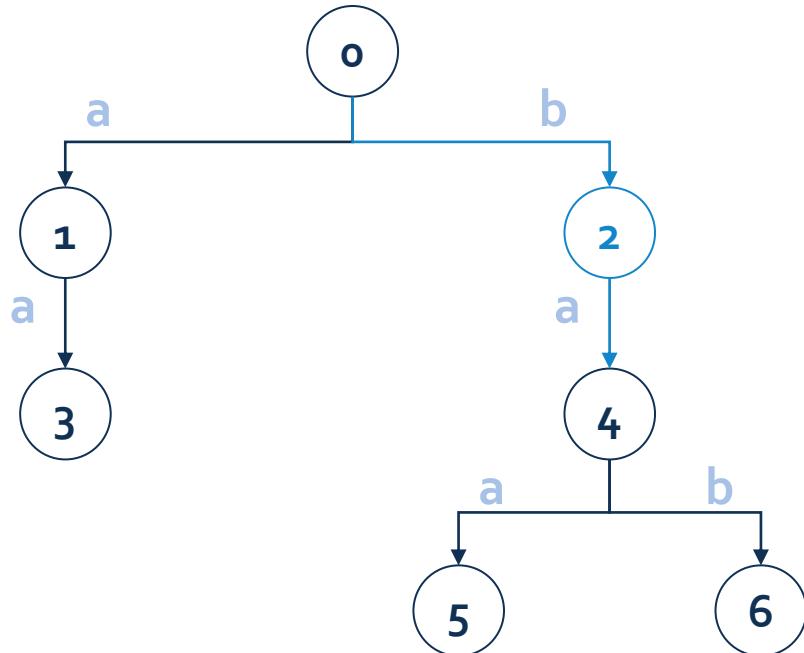
How to store the phrases



baab

- + The characters of each phrase give the path from the root to a leaf
- + The character to be encoded are used to traverse the trie until the end of the path
- + The last node contains the phrase number to output
- + A new node is added with next input character, to form a new phrase.

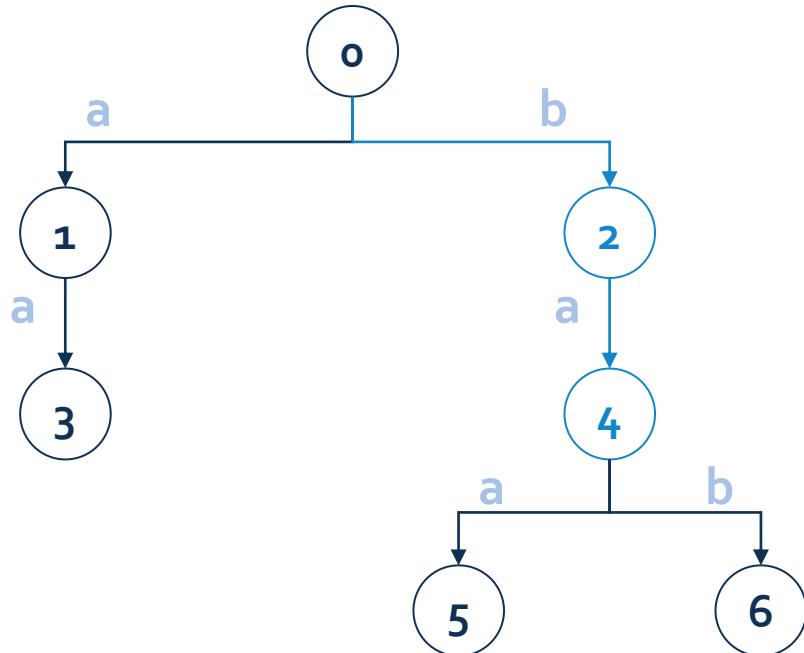
How to store the phrases



baab

- + The characters of each phrase give the path from the root to a leaf
- + The character to be encoded are used to traverse the trie until the end of the path
- + The last node contains the phrase number to output
- + A new node is added with next input character, to form a new phrase.

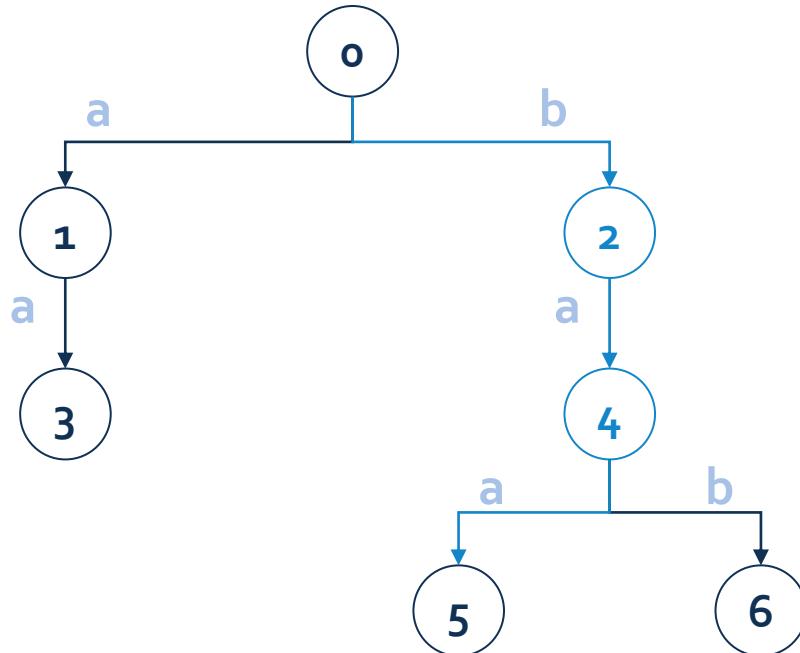
How to store the phrases



baab

- + The characters of each phrase give the path from the root to a leaf
- + The character to be encoded are used to traverse the trie until the end of the path
- + The last node contains the phrase number to output
- + A new node is added with next input character, to form a new phrase.

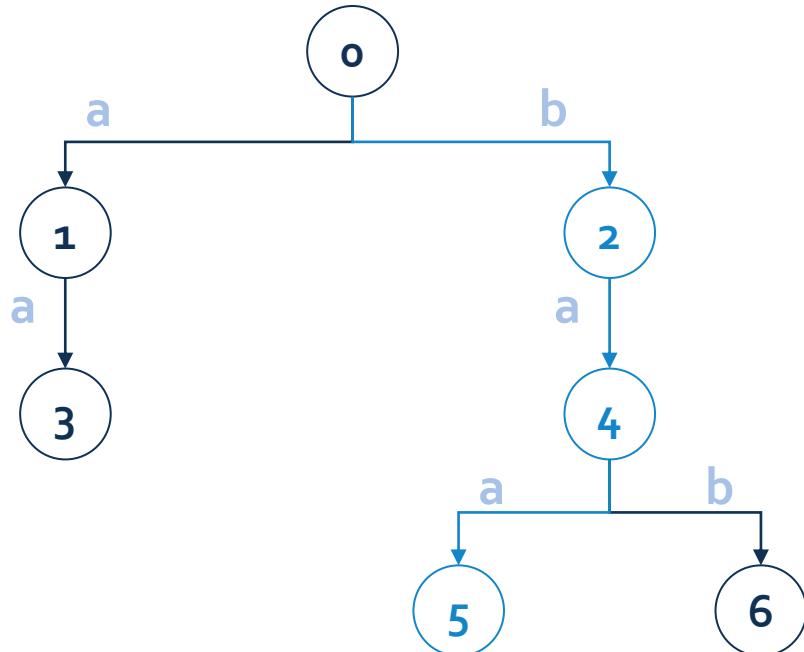
How to store the phrases



baab

- + I caratteri di ogni frase specificano un percorso dalla radice ad una foglia
- + I caratteri da codificare sono usati per attraversare il trie fino a che il percorso non termina
- + L'ultimo nodo contiene il numero della frase da emettere
- + Un nuovo nodo è aggiunto, a formare una nuova frase con il prossimo carattere di input

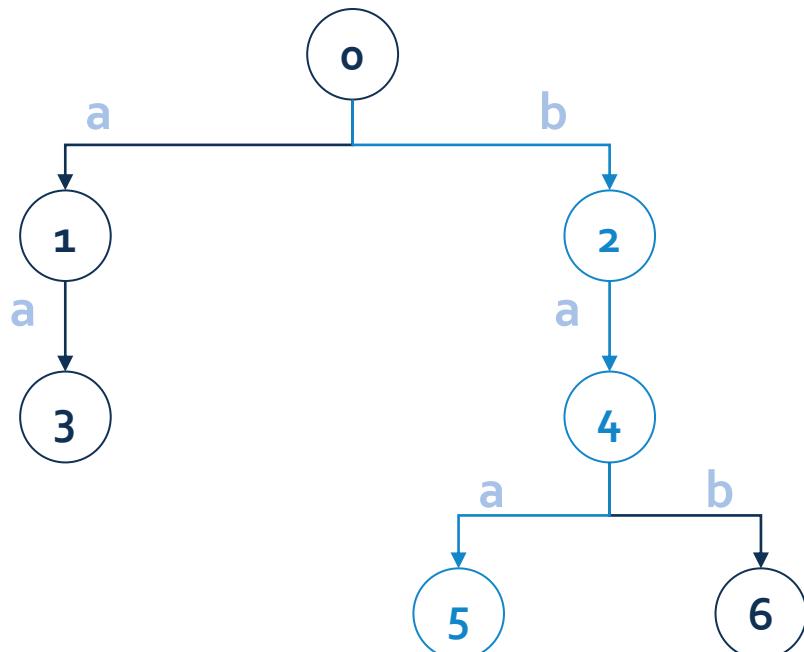
How to store the phrases



baab

- + The characters of each phrase give the path from the root to a leaf
- + The character to be encoded are used to traverse the trie until the end of the path
- + The last node contains the phrase number to output
- + A new node is added with next input character, to form a new phrase.

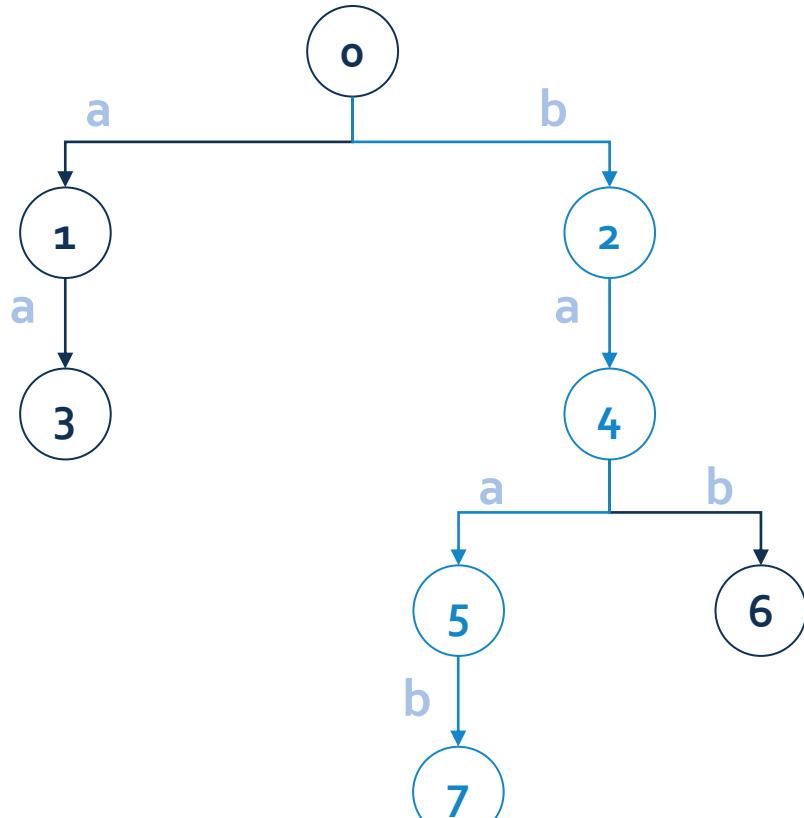
How to store the phrases



baab <5,b>

- + The characters of each phrase give the path from the root to a leaf
- + The character to be encoded are used to traverse the trie until the end of the path
- + The last node contains the phrase number to output
- + A new node is added with next input character, to form a new phrase.

How to store the phrases



baab

<5,b>

- + The characters of each phrase give the path from the root to a leaf
- + The character to be encoded are used to traverse the trie until the end of the path
- + The last node contains the phrase number to output
- + A new node is added with next input character, to form a new phrase.

The encoding process

INPUT	OUTPUT	CODEBOOK
$\underline{c} caccbcabcaba$	$(0, c)$	$\{\text{'''} = 0, c = 1\}$
$\underline{c} \underline{ca} ccbcabcaba$	$(0, c) (1, a)$	$\{\text{'''} = 0, c = 1, ca = 2\}$
$\underline{cca} \underline{cc} bcabcaba$	$(0, c) (1, a) (1, c)$	$\{\text{'''} = 0, c = 1, ca = 2, cc = 3\}$
$\underline{ccacc} \underline{b} cabcaba$	$(0, c) (1, a) (1, c) (0, b)$	$\{\text{'''} = 0, c = 1, ca = 2, cc = 3, \}$ $b = 4$
$\underline{ccaccc} \underline{cab} caba$	$(0, c) (1, a) (1, c) (0, b) (2, b)$	$\{\text{'''} = 0, c = 1, ca = 2, cc = 3, \}$ $b = 4, cab = 5$
$\underline{ccacccbcab} \underline{caba}$	$(0, c) (1, a) (1, c) (0, b) (2, b) (5, a)$	$\{\text{'''} = 0, c = 1, ca = 2, cc = 3, \}$ $b = 4, cab = 5, caba = 6$

The decoding process

INPUT	OUTPUT	CODEBOOK
(0, c) (1, <i>a</i>) (1, <i>c</i>) (0, <i>b</i>) (2, <i>b</i>) (5, <i>a</i>)	c	{ $''' = 0, c = 1 \}$
(0, <i>c</i>) (1, a) (1, <i>c</i>) (0, <i>b</i>) (2, <i>b</i>) (5, <i>a</i>)	<i>c</i> <u>ca</u>	{ $''' = 0, c = 1, ca = 2 \}$
(0, <i>c</i>) (1, <i>a</i>) (1, c) (0, <i>b</i>) (2, <i>b</i>) (5, <i>a</i>)	<i>cca</i> <u>cc</u>	{ $''' = 0, c = 1, ca = 2, cc = 3 \}$
(0, <i>c</i>) (1, <i>a</i>) (1, <i>c</i>) (0, b) (2, <i>b</i>) (5, <i>a</i>)	<i>ccacc</i> <u>b</u>	{ $''' = 0, c = 1, ca = 2, cc = 3, b = 4 \}$
(0, <i>c</i>) (1, <i>a</i>) (1, <i>c</i>) (0, <i>b</i>) (2, b) (5, <i>a</i>)	<i>ccaccb</i> <u>cab</u>	{ $''' = 0, c = 1, ca = 2, cc = 3, b = 4, cab = 5 \}$
(0, <i>c</i>) (1, <i>a</i>) (1, <i>c</i>) (0, <i>b</i>) (2, <i>b</i>) (5, a)	<i>ccaccbca</i> <u>aba</u>	{ $''' = 0, c = 1, ca = 2, cc = 3, b = 4, cab = 5, caba = 6 \}$

A problem

- + The trie data structure continues to grow during coding, and eventually growth must be stopped to avoid an excessive use of memory
- + There are several strategies
 - + The trie can be rebuilt from scratch
 - + The trie can be used as it is, without any update
 - + The trie can be partially rebuilt by using the last part of the text (and this avoids the penalty of starting from scratch)

LZ78 vs. LZ77

- + LZ78 encoding can be faster. Huffman and AC can be used to encode the pairs.
- + LZ78 decoding is slower because the decoder must also store the phrases..

LZ78: exercise

- + Find the LZ78 parsing of the text

aabbababbbaababaa

- + Solution: <0,a><1,b><0,b><2,a><3,b><3,a><4,b><1,a>

LZ78 - exercise

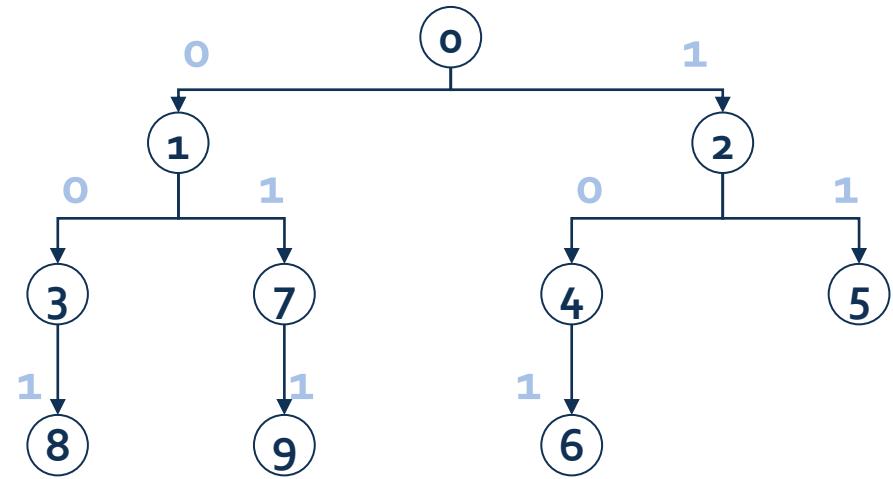
- + Find the LZ78 parse for the following string and show the trie with the phrases

0100101110101001011

SOL. <0,0><0,1><1,0><2,0><2,1><4,1><1,1><3,1><7,1>

PHRASES

0	<null>	8	001
1	0	9	011
2	1		
3	00		
4	10		
5	11		
6	101		
7	01		



LZW algorithm: a variant of LZ78

- + It is one of the most popular variants of Lempel-Ziv 78 (Welch 1984)
- + It is the basis of many popular compressors or compressor schemes, such as TIFF, GIF, UNIX utility compress and uncompress
- + The main difference between LZW and LZ78 is that LZW only encodes phrase numbers without any additional characters

LZW variant

- + A new phrase is constructed by appending the first character of the next phrase.
- + At the beginning, the codebook contains only the list of the characters of the alphabet
- + This codebook does not have to be sent, as is the case with Huffman, but is reconstructed at the time of decoding.

Encoding process

- + The codebook is initialized

INPUT	OUTPUT	CODEBOOK
$bocababbcbcbaaaabbc$		$\{a = 1, b = 2, c = 3\}$

Encoding process

- + The longest phrase in the codebook is searched for
- + Such a phrase is encoded
- + A new phrase is added to the codebook by concatenating the previous phrase with the next symbol

INPUT	OUTPUT	CODEBOOK
$\underline{bc} \underline{ababbc} bcbaaaabbc$	2	$\{a = 1, b = 2, c = 3, bc = 4\}$
$\underline{b} \underline{ca} \underline{babbc} bcbaaaabbc$	2, 3	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5\}$
$\underline{bc} \underline{ab} \underline{abb} bcbcbaaaabbc$	2,3,1	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6\}$
$\underline{bca} \underline{ba} \underline{bbc} bcbcbaaaabbc$	2,3,1,2	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, ba = 7\}$
$\underline{bcab} \underline{abb} \underline{cbc} bcbaaaabbc$	2,3,1,2,6	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, ba = 7, abb = 8\}$
$\underline{bcabab} \underline{bcb} \underline{cba} bcbaaaabbc$	2,3,1,2,6,4	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, ba = 7, abb = 8, bcb = 9\}$
$\underline{bcababbc} \underline{bcb} \underline{aa} bcbcbaaaabbc$	2,3,1,2,6,4,9	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, ba = 7, abb = 8, bcb = 9, bcbba = 10\}$
$\underline{bcababbc} \underline{bcb} \underline{aa} \underline{aab} bcbcbaaaabbc$	2,3,1,2,6,4,9,1	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, ba = 7, abb = 8, bcb = 9, bcbba = 10, aaa = 11\}$
$\underline{bcababbc} \underline{bcb} \underline{aa} \underline{aaa} bcbcbaaaabbc$	2,3,1,2,6,4,9,1,11	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, ba = 7, abb = 8, bcb = 9, bcbba = 10, aaa = 11, aaa = 12\}$
$\underline{bcababbc} \underline{bcb} \underline{aa} \underline{abb} bcbcbaaaabbc$	2,3,1,2,6,4,9,1,11,8	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, ba = 7, abb = 8, bcb = 9, bcbba = 10, aaa = 11, aaa = 12, abbc = 13\}$
$\underline{bcababbc} \underline{bcb} \underline{aa} \underline{abb} bcbcbaaaabbc$	2,3,1,2,6,4,9,1,11,8,3	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, ba = 7, abb = 8, bcb = 9, bcbba = 10, aaa = 11, aaa = 12, abbc = 13\}$

Decoding process

INPUT	OUTPUT	Codebook
2, 3, 1, 2, 6, 4, 9, 1, 11, 8, 3	b	{a = 1, b = 2, c = 3}
2, 3, 1, 2, 6, 4, 9, 1, 11, 8, 3	\underline{bc}	{a = 1, b = 2, c = 3, bc = 4}
2, 3, 1, 2, 6, 4, 9, 1, 11, 8, 3	$\underline{b} \underline{ca}$	{a = 1, b = 2, c = 3, bc = 4, ca = 5}

Decoding process

$\underline{2,3,1,2,6,4,9,1,11,8,3}$	$b \cancel{c} \underline{ab}$	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6 \end{cases}$
$\underline{2,3,1,2,6,4,9,1,11,8,3}$	$b \cancel{ca} \underline{ba} \underline{b}$	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7 \end{cases}$
$\underline{2,3,1,2,6,4,9,1,11,8,3}$	$b \cancel{cab} \underline{abb} \underline{c}$	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8 \end{cases}$
$\underline{2,3,1,2,6,4,9,1,11,8,3}$	$b \cancel{cabab} \underline{bc} ?$	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8 \end{cases}$

$\underline{2,3,1,2,6,4,9,1,11,8,3}$	$b \cancel{cabab} \underline{bc} \underline{b} \underline{c} ?$	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bc? = 9 \end{cases}$
--------------------------------------	---	--

$\underline{2,3,1,2,6,4,9,1,11,8,3}$	$b \cancel{cabab} \underline{bcb} \underline{cb}$	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9 \end{cases}$
$\underline{2,3,1,2,6,4,9,1,11,8,3}$	$b \cancel{cababbc} \underline{bcba}$	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ bcba = 10 \end{cases}$

Decoding process

2,3,1,2,6,4,9,1,11,8,3	<i>bcababbcbeb</i> <u>a</u> ?	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ beba = 10 \end{cases}$
2,3,1,2,6,4,9,1,11,8,3	<i>bcababbcbeb</i> <u>a a</u> ?	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ beba = 10, aa = 11 \end{cases}$
2,3,1,2,6,4,9,1,11,8,3	<i>bcababbcbeb</i> <u>aa a</u>	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ beba = 10, aa = 11 \end{cases}$
2,3,1,2,6,4,9,1,11,8,3	<i>bcababbcbeb</i> <u>aa a bb</u>	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ beba = 10, aa = 11, aaa = 12 \end{cases}$
2,3,1,2,6,4,9,1,11,8,3	<i>bcababbcbebaaaabbc</i>	$\begin{cases} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ beba = 10, aa = 11, aaa = 12 \end{cases}$

Which is the performance?

- + Lempel and Ziv show that:
 - + Under certain assumptions about the source, LZ77 and LZ78 are optimal, i.e. their compression ratio approaches the entropy
- + The notion of optimality implies that assuming that the string is infinite and that it is produced by a source that always behaves in the same way, then the compression ratio tends towards entropy.
- + In recent studies the redundancy has been estimated. For LZ78 is $O(1/\log n)$, where n is the length of the string.
- + Such results do not hold for all sources.

Recall the empirical order-0 entropy

- + It is based on the probability distribution implicitly defined by the input, i.e. if s is the input of size n on an alphabet with h symbols

$$H_0(s) = - \sum_{i=1}^h \frac{n_i}{n} \log \frac{n_i}{n}$$

- + We assume that $0 \log 0 = 0$
- + The value $|s|H_0(s)$ represents the size of the output of an ideal compressor that uses $-\log \frac{n_i}{n}$ bits to encode the i -th symbol of the alphabet.

Order-k empirical entropy

- + Using zero-order entropy is often not enough to measure the performance of a compressor. For example, if we want to compress an English text, we would benefit both from knowledge of the frequencies but also from the fact that these frequencies may depend on the context.
- + If we consider a context of length k :
- +
$$H_k(s) = -\frac{1}{|s|} \sum_{w \in A^k} \left(\sum_{i=1}^h n_{wa_i} \log \frac{n_{wa_i}}{n} \right)$$
- + It means that $|s|H_k(s)$ represents the size of the output of an ideal compressor that uses $-\log \frac{n_{wa_i}}{n}$ bits for the symbol a_i when it appears after the context w .
- + $H_k(s)$ is a decreasing function on k

LZ78 evaluation

- + In 1992 Hansel, Perrin and Simon proved that if t is the number of phrases then

$$LZ78(s) \leq |s|H_0(s) + t \log \frac{|s|}{t} + \theta(t)$$

- + Manzini and Kosaraju in 2000 proved that for strings with low entropy LZ78 does not work well because of long runs of identical symbols

[S.R.Kosaraju, G. Manzini, « Compression of low entropy strings with Lempel-Ziv Algorithms» Siam J. Computing, 29 (3), pp 893-911, 2000.]

Optimality

- + A compression algorithm is *coarsely optimal* if for all k there exists a function $f_k(n)$ tending to 0 (for n approaching to infinity) and such that for all sequences S of increasing length the compression ratio of the algorithm is at most
$$H_k(s) + f_k(|s|).$$
- + Plotnik et al. proved the coarse optimality of LZ78.
- + Kosaraju and Manzini noticed that the notion of coarse optimality does not necessarily imply a good algorithm because, if the entropy of the string S approaches zero, the algorithm can compress badly. This observation makes the par with the one we made for Huffman, related to the extra-bit needed for each encoded symbol. That extra-bit was ok for large entropies, but it was considered bad for entropies approaching 0.
- + PROPOSITION: There exist strings for which the compression ratio achieved by LZ78 is at least $g(|s|) H_0(s)$ with $g(n)$ such that $\lim_{n \rightarrow \infty} g(n) = \infty$.
- + Dim: Let us consider the string 01^{n-1} , whose 0-order entropy is $\Theta(\log n/n)$. LZ78 produces $\Theta(\sqrt{n})$ phrases. Therefore $g(n) = \sqrt{n}/\log n$

Evaluation of LZ77

- + Kosaraju and Manzini proved that
- + $LZ_{77}(s) \leq 8 |s|H_0(s) + \text{terms of lower order}$
- + PROPOSITION: LZ₇₇ is coarsely optimal with respect to the k-th order entropy , for every k greater than or equal to 0.
- + PROPOSITION: LZ₇₇ with a bounded buffer is not coarsely optimal.
- + PROOF. It is shown that however one chooses the window, one can find a string whose compression ratio exceeds the entropy of order k.

λ -Optimality

- + Kosaraju and Manzini introduced the notion of λ -optimality, i.e. the compression ratio must be upper bounded by $H_k(T) + o(H_k(T))$.
(o little = it approaches to o more rapidly)
- + LZ77, with no assumption on the buffer, is coarsely and also 8-optimal with respect to H_0 . It is not when $k \geq 1$.
- + LZ78 is not λ -optimal

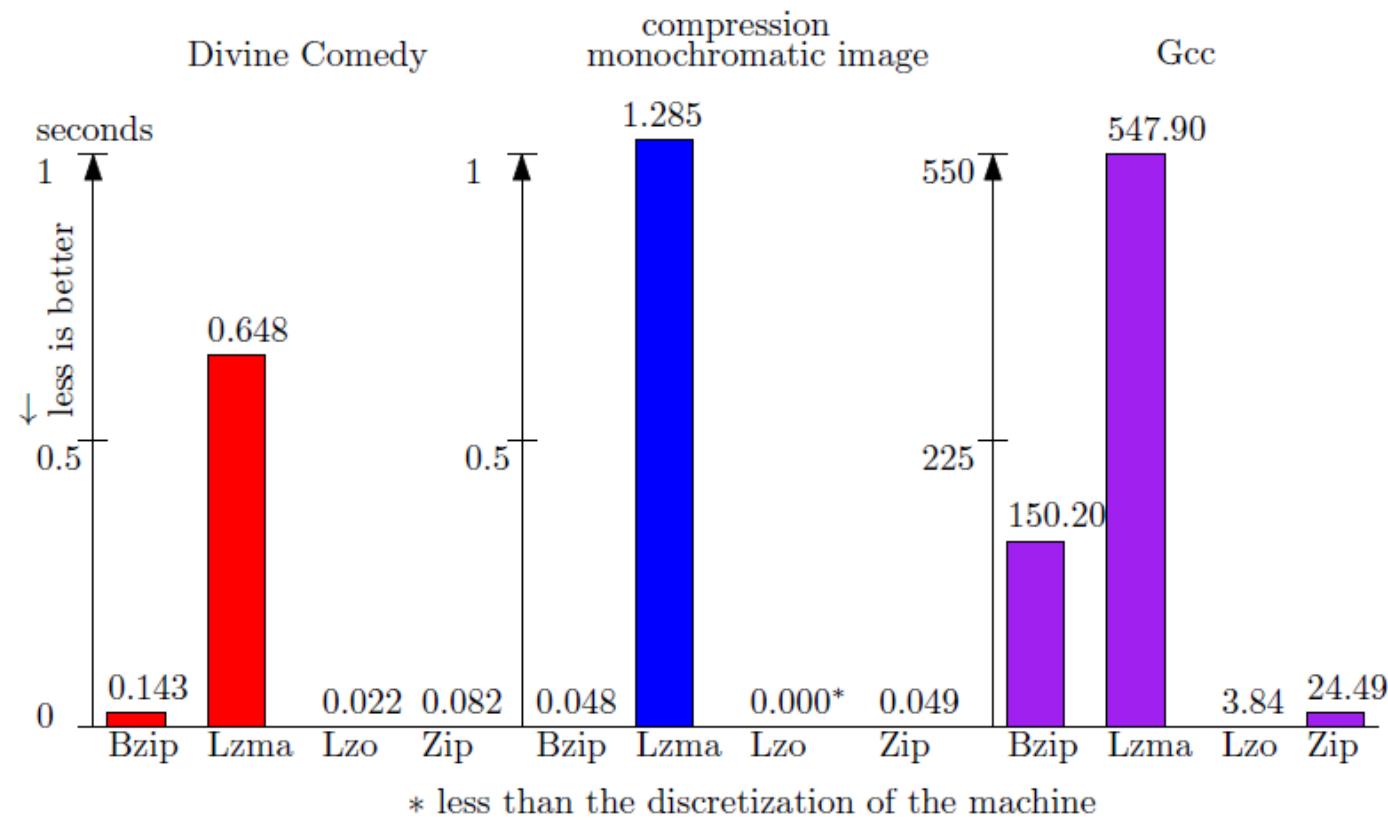
Kosaraju and Manzini variant

- + They had the idea of combining LZ78 with the RLE (Run -length encoding) algorithm.
- + RLE is a technique frequently used in image compression, i.e. when the input contains long runs of equal symbols.
- + A run of equal d symbols of length m is transmitted by encoding the run length m, i.e. by binary encoding of m, where d represents 1.
- + For instance dddd->dod
- + Other versions encode (d,5), i.e. (d,101)
- + They introduce the algorithm LZ78-RL that applies LZ78 to the string RLE encoded.
- + $LZ78-RL(s) \leq 3 |s|H_0(s) + \text{terms of lower order}$

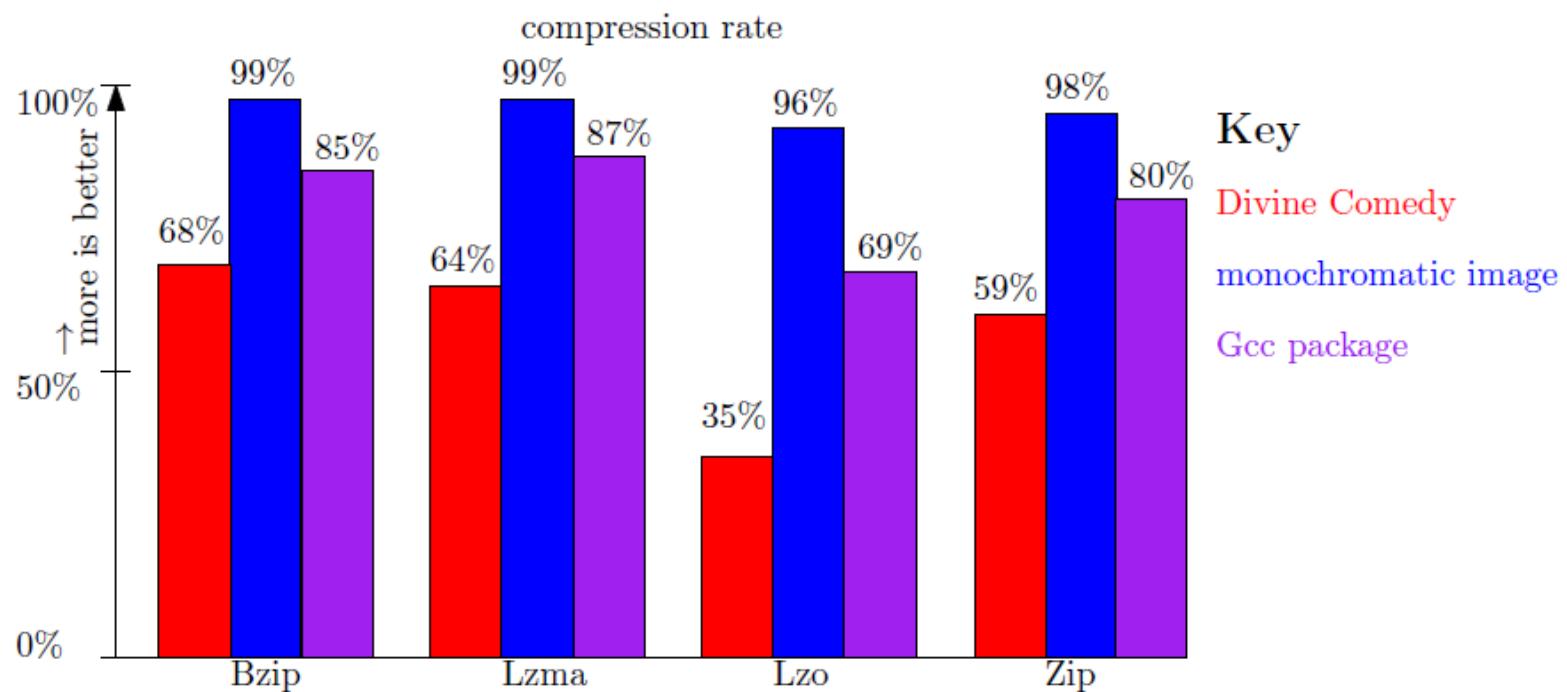
Comparison of compressors

- + In some experiments reported by P. Ferragina "The magic of Algorithms! Lectures on some algorithmic pearls.", in which a BWT-based compressor is compared with LZMA (Lempel-Ziv-Markov chain algorithm), LZO (LZ-Oberhumer zip), ZIP.
- + Tests are realized with PC with 2GB RAM (using ramfs), AMD Athlon(tm)X2 Dual- Core QL-64, running Linux.
- + Three datasets: "The Divine Comedy", Grayscale image, P package gcc-4.4.3

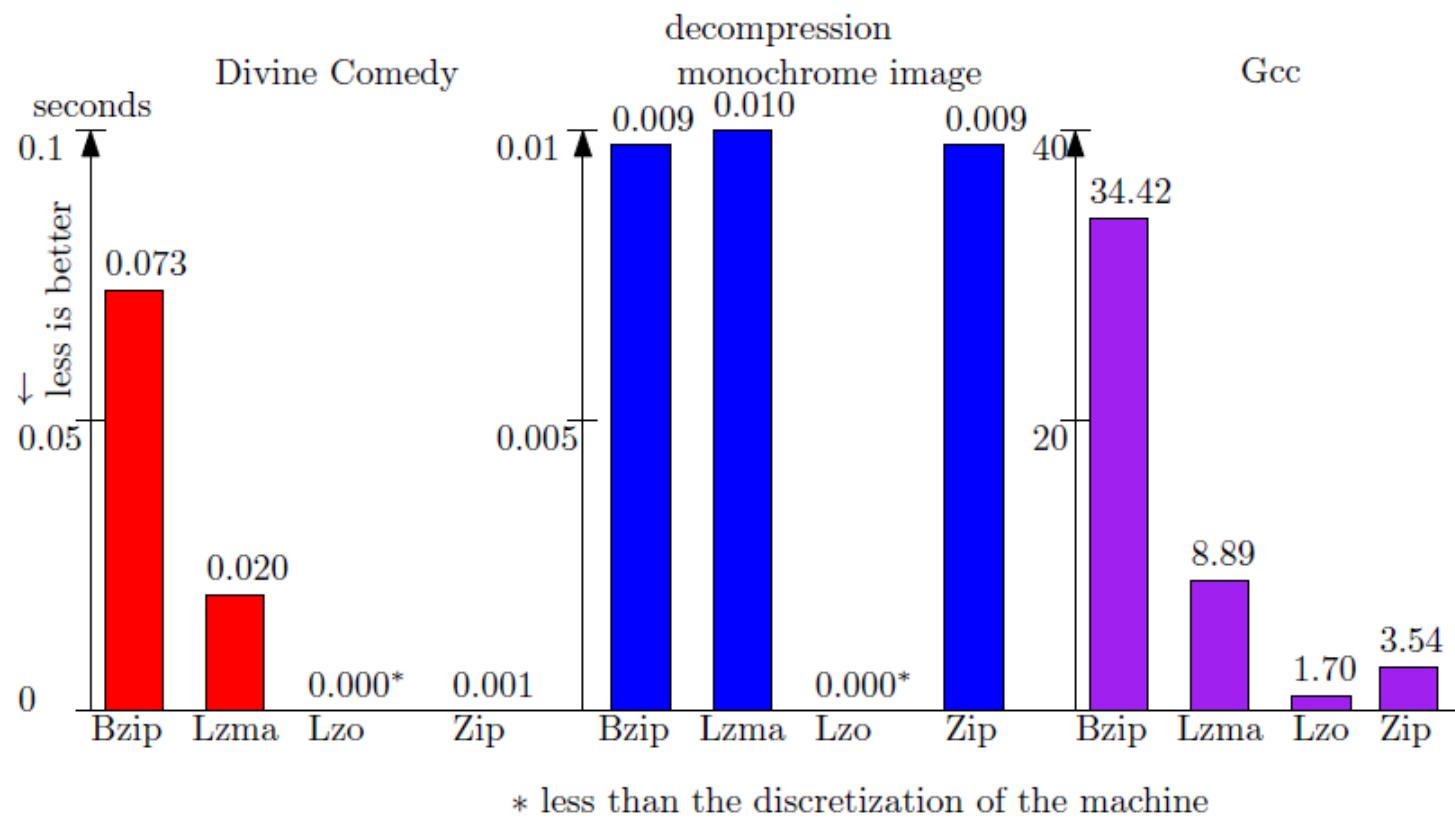
Experiments: compression speed



Experiments: compression saving



Experiments: decompression time



Comparison with statistical compressors

- + Statistical compressors (Huffman and AC) are evaluated in terms of entropy of order o , i.e.

$$|out_{stat}(X)| \approx |X|H_0(X)$$

- + They are not sensitive to repetitiveness. If we consider highly repetitive texts, for example X^k it is easy to verify that

$$|out_{stat}(X^k)| \approx |X^k|H_0(X^k) = |X^k|H_0(X) = k|X|H_0(X) \approx k|out_{stat}(X)|$$

- + When LZ77 is applied to X^k , a substring of the type X^{k-1} has a previous occurrence (as a prefix), therefore it will be encoded by $(|X|, (k-1)|X|, \text{symbol})$, then

$$|out_{lz}(X^k)| \leq |out_{lz}(X)| + c, \quad c \text{ constant.}$$

- + Exercise: let us consider the string $S=(ab)^n$. Which is the order o entropy? Which is the Huffman encoding? Which is the LZ parsing with all LZ-based compressors?

Exercises

- + Let $z_{77}(T)$ the number of phrases defined by LZ77
- + Is it possible to verify that :
 - + $z_{77}(T) = z_{77}(T^R)$?
 - + $z_{77}(ST) = z_{77}(S)z_{77}(T)$?
- + Given an input text, compare the number of phrases produced by LZ77, LZ78 e LZW.
- + Design an algorithm that computes in linear time the LZss parsing given its LPF array. For each j $\text{LPF}[j]$ stores the maximal length of factors that start at position j and at a previous position.



FM-Index

(Full-text index in Minute space)

Friday May 5th, 2023

Motivation

- + Simultaneously combines text compression with indexing (not needing the original text)
- + In practice, it combines the BWT with a few small auxiliary structures
- + It is a structure introduced by Paolo Ferragina and Giovanni Manzini in "Opportunistic data structures with applications." *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on.* IEEE, 2000.
- + Deeply studied in:
Paolo Ferragina, Giovanni Manzini. " Indexing compressed text ". Journal of the ACM 52(4): 552-581 (2005).
- + **Goal: Given a pattern P, count its occurrences and locate it by looking only at the compressed text (or portions of it).**
- + It does so efficiently
 - + Time:
 - + $O(p)$ for counting the occurrences.
 - + $O(p+occ \log^{1+\varepsilon} n)$ for locating all the occurrences, where ε is an arbitrary positive constant chosen during the construction of the index
 - + Space: At most $5n H_k(T) + O(n/\log^\varepsilon n)$.

How does it work?

- + It is based on the relationship between the *Burrows-Wheeler Transform* and the Suffix Array.
- + This is a compressed suffix array that encapsulates both the compressed text and the indexing information.
- + The full-text index on a T-text has two goals:
 - + **Count** – returns the number of the occurrences of P in T.
 - + **Locate** – finds all the positions of P in T.

First Step: compression

- Preprocessing of the text $T[1,..,n]$ by the **Burrows-Wheeler Transform**
 - Output $L[1,..,n]$ (permutation of T)
- Apply **Move-To-Front** on L
 - Output $L^{MTF}_{[1,..,n]}$
- Encode the equal-letter runs in L^{MTF} by using the **run-length encoding**
 - Output L^{rle}
- Compress L^{rle} by using a **variable-length prefix code**
 - Output Z (on the alphabet $\{0,1\}$)

Burrows-Wheeler Transform: remind

mississippi#
ississippi#m
ssissippi#mi
sissippi#mis
issippi#miss
sippi#missi
sippi#missis
ippi#mississ
ppi#mississi
pi#mississip
i#mississipp
#mississippi

Sort the rows



F	L
#	i
i	p
i	s
i	s
i	m
m	#
p	p
p	i
s	s
s	s
s	i
s	i

Burrows-Wheeler Transform: remind

mississippi#
ississippi#m
ssissippi#mi
sissippi#mis
issippi#miss
sippi#missi
sippi#missis
ippi#mississ
ppi#mississi
pi#mississip
i#mississipp
#mississippi

Sort the rows



F	L
#	i
i	p
i	s
i	s
i	m
m	#
p	p
p	i
s	s
s	s
s	i
s	i

Burrows-Wheeler Transform: remind

- Each column is a permutation of T.
- Given a row i, the character L[i] precedes F[i] in the text T.
- Consecutive characters in L are adjacent to similar strings in T.
- Therefore – L is likely to contain long runs of equal letters

F	L
#	i
i	p
i	s
i	s
i	m
i	#
m	p
p	i
p	s
s	s
s	i
s	i

How to construct the function LF

- + Two auxiliary structures are used:

- + $C[1, \dots, |\Sigma|]$: $C[c]$ contains the total number of characters in T that are alphabetically smaller than c (including repetitions)
- + $\text{Occ}(c, q)$: number of occurrences of the character c in the prefix $L[1, q]$

- + For each i ,
 $\text{LF}(i) = C[L[i]] + \text{Occ}(L[i], i)$

- + Example: In the case of the string
 $T = \text{mississippi}\#$,
 $\text{LF}[10] = C[s] + \text{Occ}(s, 10) = 8 + 4 = 12$

C	1	5	6	8
	i	m	p	s

F	L
# mississipp	i
i #mississip	p
i ppi#missis	s
i ssippi#mis	s
i ssissippi#	m
m ississippi	#
p i#mississi	p
p pi#mississ	i
sippi#missi	s
s issippi#mi	s
s sippi#miss	i
s sissippi#m	i
	12

Recover T from L

L

i
p
s
s
m

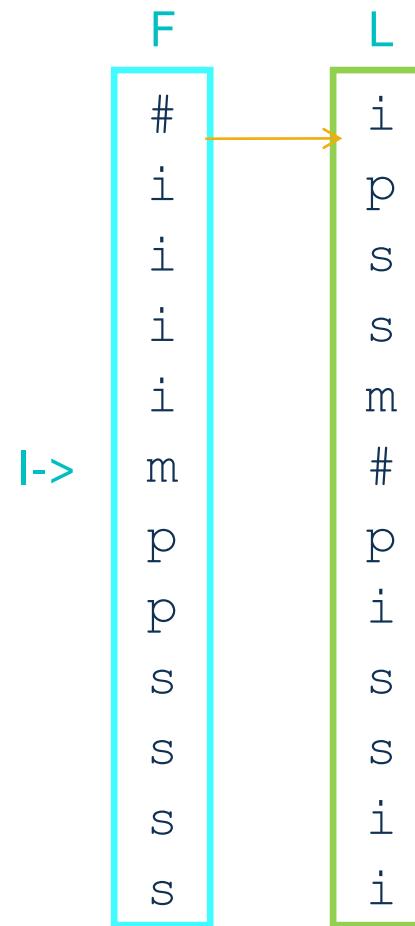
p
i
s
s
i
i

Recover T from L

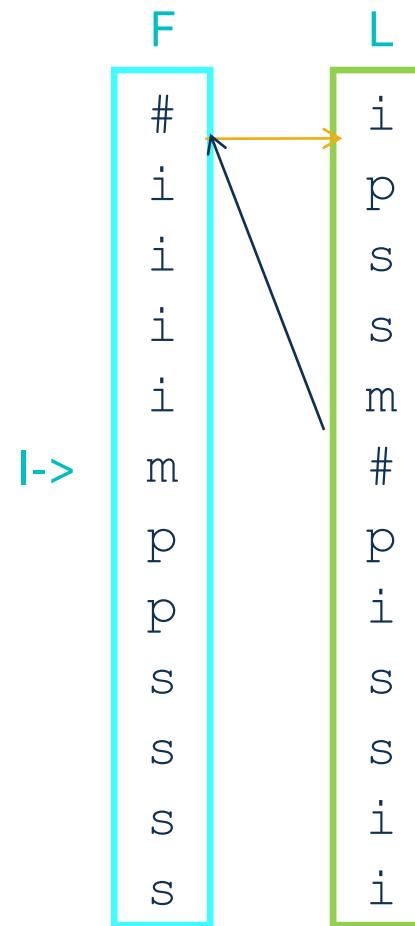
F	L
#	i
i	p
i	s
i	m
m	#
p	p
p	i
s	s
s	s
s	s
s	i

I->

Recover T from L

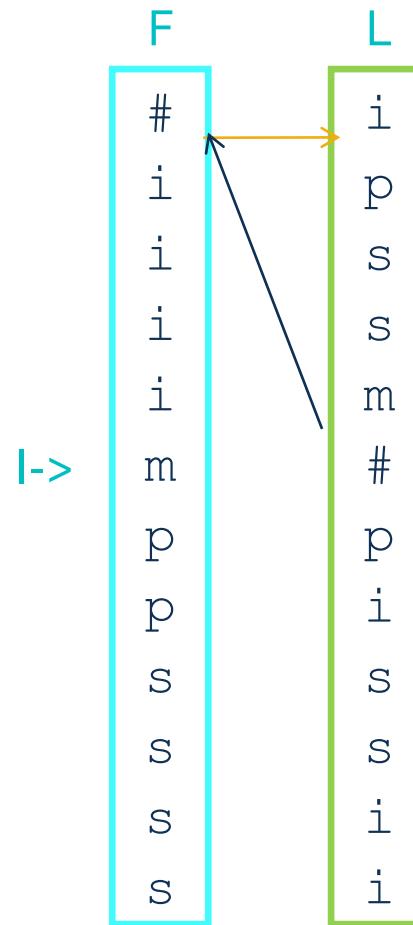


Recover T from L



Ricostruire T da L

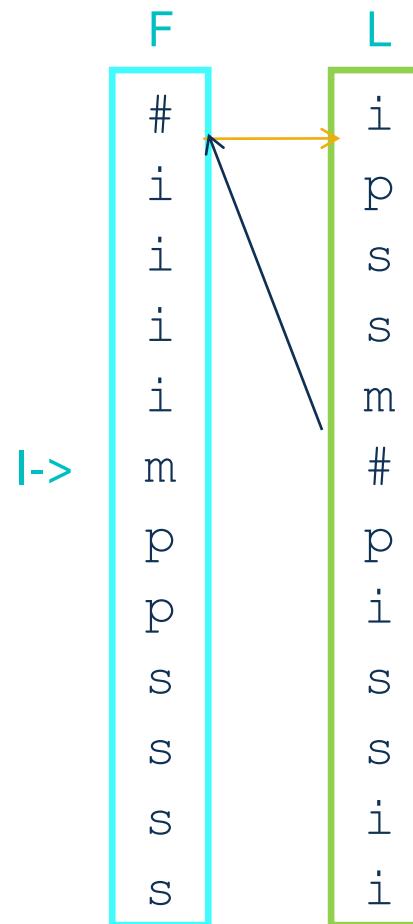
For $k=n-1$ down to 1



Recover T from L

$T[n]=\#$

For $k=n-1$ down to 1

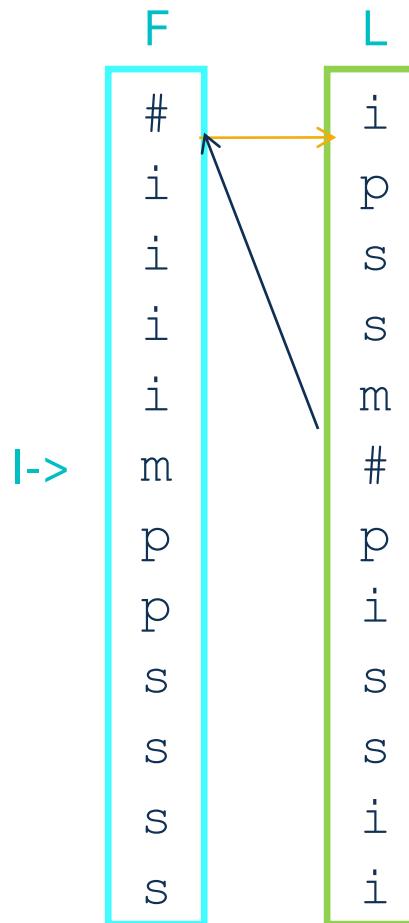


Recover T from L

$T[n]=\#$

$i=|L|$

For $k=n-1$ down to 1



Recover T from L

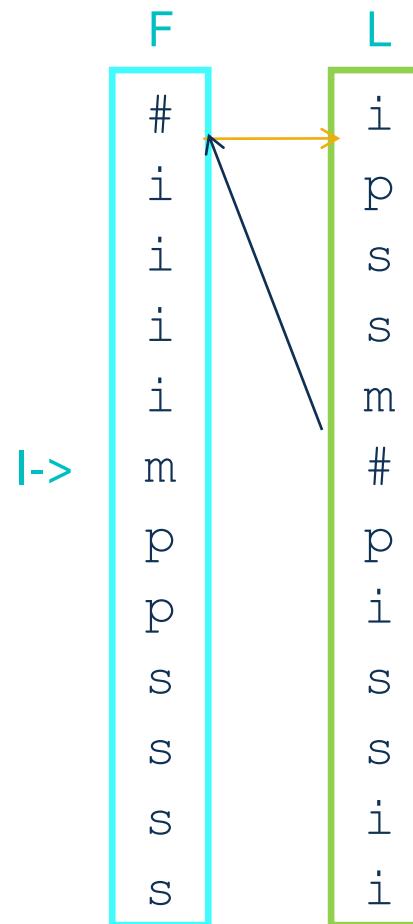
$T[n]=\#$

$i=1$

For $k=n-1$ down to 1

$T[k]=L[LF[i]]$

$i=LF[i]$



Recover T from L

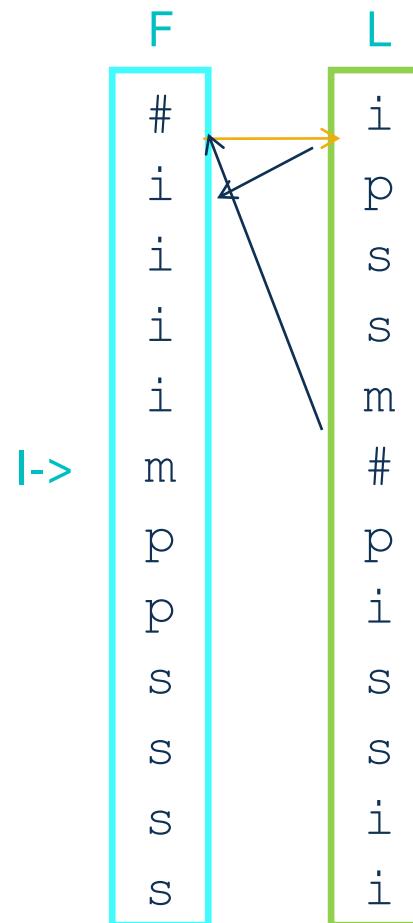
$T[n]=\#$

$i=1$

For $k=n-1$ down to 1

$T[k]=L[LF[i]]$

$i=LF[i]$



Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i

#	i	m	p	s
0	1	2	3	4

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i
1											

#	i	m	p	s
0	1	2	3	4

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i
1											

#	i	m	p	s
0	1	2	3	4

i	#	m	p	s
0	1	2	3	4

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i
1	3										

#	i	m	p	s
0	1	2	3	4

i	#	m	p	s
0	1	2	3	4

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i
1	3										

#	i	m	p	s
0	1	2	3	4

i	#	m	p	s
0	1	2	3	4

p	i	#	m	s
0	1	2	3	4

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i
1	3	4									

#	i	m	p	s
0	1	2	3	4

i	#	m	p	s
0	1	2	3	4

p	i	#	m	s
0	1	2	3	4

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i
1	3	4									

#	i	m	p	s
0	1	2	3	4

i	#	m	p	s
0	1	2	3	4

p	i	#	m	s
0	1	2	3	4

s	p	i	#	m
0	1	2	3	4

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i
1	3	4	0								

#	i	m	p	s
0	1	2	3	4

i	#	m	p	s
0	1	2	3	4

p	i	#	m	s
0	1	2	3	4

s	p	i	#	m
0	1	2	3	4

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.
- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted.
- We obtain the string L^{MTF} which is likely to be dominated by small numbers.
- Runs of consecutive equal letters become runs of zero in L^{MTF}

L

i	p	s	s	m	#	p	i	s	s	i	i
1	3	4	0	4	4	3	4	4	0	1	0

#	i	m	p	s
0	1	2	3	4

i	#	m	p	s
0	1	2	3	4

p	i	#	m	s
0	1	2	3	4

s	p	i	#	m
0	1	2	3	4

And so on...

Move to front

- Replace each character of L with the number of distinct characters seen since the previous occurrence of the same character.

- Consider the array $MTF[0, \dots, |\Sigma|-1]$ lexicographically sorted

- We obtain the string L^{MTF} which is likely to be dominated by small numbers.

- Runs of consecutive equal letters become runs of zero in L^{MTF}

L	i	p	s	s	m	#	p	i	s	s	i	i
1	2	2	2	2	3	3	3	3	4	4	1	0
1	2	2	2	2	3	3	3	3	4	4	1	0

- Bad example

- For larger texts there will be many runs of zeros and a large number of small numbers.

- The reason is that BWT produces runs of equal symbols.

0	1	2	3	4
---	---	---	---	---

Run length encoding: a variant

- Replace each sequence of consecutive zeroes 0^m with:
 - ($m+1$) in binary representation
 - First the least significant bit
 - Discard the most significant bit
- Such an encoding uses **2 new symbols – 0, 1**
- L^{rle} is defined on the alphabet
 $\{ 0, 1, 1, 2, \dots, |\Sigma|-1 \}$

L	i	p	s	s	m	#	p	i	s	s	i	i
L^{MTF}	1	3	4	0	4	4	3	4	4	0	1	0

Example

- $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
- $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
- $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
- $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
- $00000 \rightarrow 5+1 = 6 \rightarrow 110 \rightarrow 011 \rightarrow 01$
- $000000 \rightarrow 6+1 = 7 \rightarrow 111 \rightarrow 111 \rightarrow 11$
- $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

Run length encoding: a variant

- Replace each sequence of consecutive zeroes 0^m with:

- ($m+1$) in binary representation
- First the least significant bit
- Discard the most significant bit

- Such an encoding uses 2 new symbols – 0, 1

- L^{rle} is defined on the alphabet

{ 0, 1, 1, 2, ..., $|\Sigma|-1$ }

L	i	p	s	s	m	#	p	i	s	s	i	i
L^{MTF}	1	3	4	0	4	4	3	4	4	0	1	0

T = pipeMississippi#
is a more realistic example

Example

1. $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
2. $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
3. $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
4. $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
5. $00000 \rightarrow 5+1 = 6 \rightarrow 110 \rightarrow 011 \rightarrow 01$
6. $000000 \rightarrow 6+1 = 7 \rightarrow 111 \rightarrow 111 \rightarrow 11$
7. $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

Run length encoding: a variant

- Replace each sequence of consecutive zeroes 0^m with:
 - ($m+1$) in binary representation
 - First the least significant bit
 - Discard the most significant bit
- Such an encoding uses **2 new symbols – 0, 1**
- L^{rle} is defined on the alphabet
 $\{ 0, 1, 1, 2, \dots, |\Sigma|-1 \}$

L	i	p	s	s	m	#	p	i	s	s	i	i
L^{MTF}	1	3	4	0	4	4	3	4	4	0	1	0

Example

- $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
- $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
- $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
- $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
- $00000 \rightarrow 5+1 = 6 \rightarrow 110 \rightarrow 011 \rightarrow 01$
- $000000 \rightarrow 6+1 = 7 \rightarrow 111 \rightarrow 111 \rightarrow 11$
- $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

Run length encoding: a variant

- Replace each sequence of consecutive zeroes 0^m with:
 - ($m+1$) in binary representation
 - First the least significant bit
 - Discard the most significant bit
- Such an encoding uses 2 new symbols – 0, 1
- L^{rle} is defined on the alphabet
 $\{ 0, 1, 1, 2, \dots, |\Sigma|-1 \}$

L	i	p	s	s	m	#	p	i	s	s	i	i
L^{MTF}	1	3	4	0	4	4	3	4	4	0	1	0

Example

- $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
- $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
- $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
- $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
- $00000 \rightarrow 5+1 = 6 \rightarrow 110 \rightarrow 011 \rightarrow 01$
- $000000 \rightarrow 6+1 = 7 \rightarrow 111 \rightarrow 111 \rightarrow 11$
- $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

Run length encoding: a variant

- Replace each sequence of consecutive zeroes 0^m with:

- ($m+1$) in binary representation
- First the least significant bit
- Discard the most significant bit

- Such an encoding uses 2 new symbols – 0, 1

- L^{rle} is defined on the alphabet

{ 0, 1, 1, 2, ..., $|\Sigma|-1$ }

L
 L^{MTF}

Example

1. $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
2. $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
3. $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
4. $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
5. $00000 \rightarrow 5+1 = 6 \rightarrow 110 \rightarrow 011 \rightarrow 01$
6. $000000 \rightarrow 6+1 = 7 \rightarrow 111 \rightarrow 111 \rightarrow 11$
7. $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

Run length encoding: a variant

- Replace each sequence of consecutive zeroes 0^m with:
 - ($m+1$) in binary representation
 - First the least significant bit
 - Discard the most significant bit
- Such an encoding uses **2 new symbols – 0, 1**
- L^{rle} is defined on the alphabet
 $\{ 0, 1, 1, 2, \dots, |\Sigma|-1 \}$

L	i	p	p	p	s	s	m	e	i	p	#	i	s	s	i	i
L^{MTF}	2	4	0	0	5	0	5	5	4	4	5	2	5	0	1	0

Example

- $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
- $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
- $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
- $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
- $00000 \rightarrow 5+1 = 6 \rightarrow 110 \rightarrow 011 \rightarrow 01$
- $000000 \rightarrow 6+1 = 7 \rightarrow 111 \rightarrow 111 \rightarrow 11$
- $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

Run length encoding: a variant

- Replace each sequence of consecutive zeroes 0^m with:
 - ($m+1$) in binary representation
 - First the least significant bit
 - Discard the most significant bit
- Such an encoding uses **2 new symbols – 0, 1**
- L^{rle} is defined on the alphabet
 $\{ 0, 1, 1, 2, \dots, |\Sigma|-1 \}$

L	i	p	p	p	s	s	m	e	i	p	#	i	s	s	i	i
L^{MTF}	2	4	0	0	5	0	5	5	4	4	4	5	2	5	0	1
L^{rle}	2	4	1	5	0	5	5	4	4	5	2	5	0	1	0	

Example

- $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
- $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
- $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
- $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
- $00000 \rightarrow 5+1 = 6 \rightarrow 110 \rightarrow 011 \rightarrow 01$
- $000000 \rightarrow 6+1 = 7 \rightarrow 111 \rightarrow 111 \rightarrow 11$
- $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

Run length encoding

- Replace each sequence of consecutive zeroes 0^m with:
 - $(m+1)$ in binary representation
 - First the least significant bit
 - Discard the most significant bit
- Such an encoding uses 2 new symbols – 0, 1
- L^{rle} is defined on the alphabet
 $\{ 0, 1, 1, 2, \dots, |\Sigma|-1 \}$

How to recover m

- given the binary number $b_0 b_1, \dots, b_k$
- replace each bit b_j with a sequence of $(b_j + 1) \cdot 2^j$ zeroes
- $10 \rightarrow (1+1) \cdot 2^0 + (0+1) \cdot 2^1 = 4$
- this is equivalent to sum $2^k - 1$

Example

1. $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
2. $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
3. $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
4. $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
5. $00000 \rightarrow 5+1 = 6 \rightarrow 110 \rightarrow 011 \rightarrow 01$
6. $000000 \rightarrow 6+1 = 7 \rightarrow 111 \rightarrow 111 \rightarrow 11$
7. $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

Variable-length prefix code

- Compress L^{rle} as follows using the alphabet $\{0,1\}$:

$$1 \rightarrow 11 \quad 0 \rightarrow 10$$

- For $i = 1, 2, \dots, |\Sigma| - 1$

- $\lfloor \log(i+1) \rfloor$ zeroes
- Followed by the binary representation of $i+1$ that takes $1 + \lfloor \log(i+1) \rfloor$
- A total of $1 + 2 \lfloor \log(i+1) \rfloor$ bit

L	i	p	p	p	s	s	m	e	i	p	#	i	s	s	i	i
L^{MTF}	2	4	0	0	5	0	5	5	4	4	5	2	5	0	1	0
L^{rle}	2	4	1	5	0	5	5	4	4	5	2	5	0	1	0	

Example

1. $i=1 \rightarrow \lfloor \log(2) \rfloor$ 0's, $\text{bin}(2) \rightarrow 010$
2. $i=2 \rightarrow \lfloor \log(3) \rfloor$ 0's, $\text{bin}(3) \rightarrow 011$
3. $i=3 \rightarrow \lfloor \log(4) \rfloor$ 0's, $\text{bin}(4) \rightarrow 00100$
4. $i=4 \rightarrow \lfloor \log(5) \rfloor$ 0's, $\text{bin}(5) \rightarrow 00101$
5. $i=5 \rightarrow \lfloor \log(6) \rfloor$ 0's, $\text{bin}(6) \rightarrow 00110$
6. $i=6 \rightarrow \lfloor \log(7) \rfloor$ 0's, $\text{bin}(7) \rightarrow 00111$
7. $i=7 \rightarrow \lfloor \log(8) \rfloor$ 0's, $\text{bin}(8) \rightarrow 0001000$

Variable-length prefix code

- Compress L^{rle} as follows using the alphabet $\{0,1\}$:

$$1 \rightarrow 11 \quad 0 \rightarrow 10$$

- For $i = 1, 2, \dots, |\Sigma| - 1$

- $\lfloor \log(i+1) \rfloor$ zeroes
- Followed by the binary representation of $i+1$ that takes $1 + \lfloor \log(i+1) \rfloor$
- A total of $1 + 2 \lfloor \log(i+1) \rfloor$ bit

L	i	p	p	p	s	s	m	e	i	p	#	i	s	s	i	i
L^{MTF}	2	4	0	0	5	0	5	5	4	4	5	2	5	0	1	0
L^{rle}	2	4	1	5	0	5	5	4	4	5	2	5	0	1	0	

Example

1. $i=1 \rightarrow \lfloor \log(2) \rfloor$ 0's, $\text{bin}(2) \rightarrow 010$
2. $i=2 \rightarrow \lfloor \log(3) \rfloor$ 0's, $\text{bin}(3) \rightarrow 011$
3. $i=3 \rightarrow \lfloor \log(4) \rfloor$ 0's, $\text{bin}(4) \rightarrow 00100$
4. $i=4 \rightarrow \lfloor \log(5) \rfloor$ 0's, $\text{bin}(5) \rightarrow 00101$
5. $i=5 \rightarrow \lfloor \log(6) \rfloor$ 0's, $\text{bin}(6) \rightarrow 00110$
6. $i=6 \rightarrow \lfloor \log(7) \rfloor$ 0's, $\text{bin}(7) \rightarrow 00111$
7. $i=7 \rightarrow \lfloor \log(8) \rfloor$ 0's, $\text{bin}(8) \rightarrow 0001000$

Variable-length prefix code

011 00101 11 00110 10 00110 00110 00101 00101 00110 011 00110 10 010 10

Compress L using RLE

using the alphabet $\{0,1\}$:

L^{rle}	2	4	1	5	0	5	5	4	4	5	2	5	0	1	0
-----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

○ $1 \rightarrow 11$ $0 \rightarrow 10$

○ For $i = 1, 2, \dots, |\Sigma| - 1$

- $\lfloor \log(i+1) \rfloor$ zeroes
- Followed by the binary representation of $i+1$ that takes $1 + \lfloor \log(i+1) \rfloor$
- A total of $1 + 2 \lfloor \log(i+1) \rfloor$ bit

Example

1. $i=1 \rightarrow \lfloor \log(2) \rfloor$ 0's, $\text{bin}(2) \rightarrow 010$
2. $i=2 \rightarrow \lfloor \log(3) \rfloor$ 0's, $\text{bin}(3) \rightarrow 011$
3. $i=3 \rightarrow \lfloor \log(4) \rfloor$ 0's, $\text{bin}(4) \rightarrow 00100$
4. $i=4 \rightarrow \lfloor \log(5) \rfloor$ 0's, $\text{bin}(5) \rightarrow 00101$
5. $i=5 \rightarrow \lfloor \log(6) \rfloor$ 0's, $\text{bin}(6) \rightarrow 00110$
6. $i=6 \rightarrow \lfloor \log(7) \rfloor$ 0's, $\text{bin}(7) \rightarrow 00111$
7. $i=7 \rightarrow \lfloor \log(8) \rfloor$ 0's, $\text{bin}(8) \rightarrow 0001000$

The compressor

- + The following compressor is considered

BW_RLX=bwt+mtf+rle+PC

- + Which are its performance in terms of compression ratio?

$$|Z| \leq 5n H_k(T) + O(\log n)$$

Modified entropy

- + There is a bound in which $H_k^*(T)$ is used:
 - + *It is called Modified empirical entropy*
 - + It represents the maximum compression ratio we can obtain by using for each symbol a codeword depending on the context of length **at most** k (instead of a context of length k).

$$n H_k(T) \leq n H_k^*(T) \leq n H_k(T) + O(\log n)$$

- + In 2001, Manzini has proved that for each k, previous compressor is bounded by

$$5n H_k^*(T) + g(k, |\Sigma|)$$

Searching a pattern in a BWT-based compressed text



Pattern matching

- + Problem: Given T a text and Z the text compressed by BW_RLX. Given a pattern P we are searching for all the occurrences of P in T , with only Z available, and without decompression.
- + We will see how to solve the problem by considering only $BWT(T)$, and without recovering the original text.

Counting all the occurrences

- + Algorithm Backward-search
- + Only the output L of the BWT is used
- + Two auxiliary data structures:
 - + $C[1, \dots, |\Sigma|]$: $C[c]$ contains the total number of characters in T that are alphabetically smaller than c (including repetitions)
 - + $\text{Occ}(c, q)$: number of occurrences of the character c in the prefix $L[1, q]$

Example

- $C[]$ for $T = \text{mississippi}\#$
- $\text{occ}(s, 5) = 2$
- $\text{occ}(s, 12) = 4$

1	5	6	8
i	m	p	s

F	L
# mississipp i	1
i #mississip p	2
i ppi#missis s	3
i ssippi#mis s	4
i ssissippi# m	5
m ississippi #	6
p i#mississi p	7
p pi#mississ i	8
s ippi#missi s	9
s issippi#mi s	10
s sippi#miss i	11
s sissippi#m i	12

COUNT P in T: idea

This is unknown

```
#mississipp  
i#mississip  
ippi#missis  
issippi#mis  
ississippi#  
mississippi  
pi#mississi  
ppi#mississ  
ippi#missi  
issippi#mi  
ssippi#miss  
ssissippi#m
```

L

i
p
s
s
m

p
i
s
s
i
i

COUNT P in T: idea

P = si

This is unknown

```
#mississipp  
i#mississip  
ippi#missis  
issippi#mis  
issippi#  
mississippi  
pi#mississi  
ppi#mississ  
ippi#missis  
issippi#mi  
ssippi#miss  
ssissippi#m
```

L

i
p
s
s
m

p
i
s
s
i

Available
information

C

i	1
m	5
p	6
S	8

COUNT P in T: idea

$$P = S_i$$

First step

This is unknown

#mississippi
i#mississip
ippi#missis
issippi#mis
ississippi#
mississippi
pi#mississi
ppi#mississ
sippi#missi
issippi#mi
ssippi#miss
ssissippi#m

L

i
p
s
s
m

p
i
s
s
i

Available
information

C

i	1
m	5
p	6
S	8

COUNT P in T: idea

P = S_i

First step

This is unknown

Rows having «i» as a prefix

fr

#mississippi
i#mississip
ippi#missis
issippi#mis
issippi#
mississippi
pi#mississi
ppi#mississ
sippi#missi
issippi#mi
ssippi#miss
ssissippi#m

lr

L

i
p
s
s
m

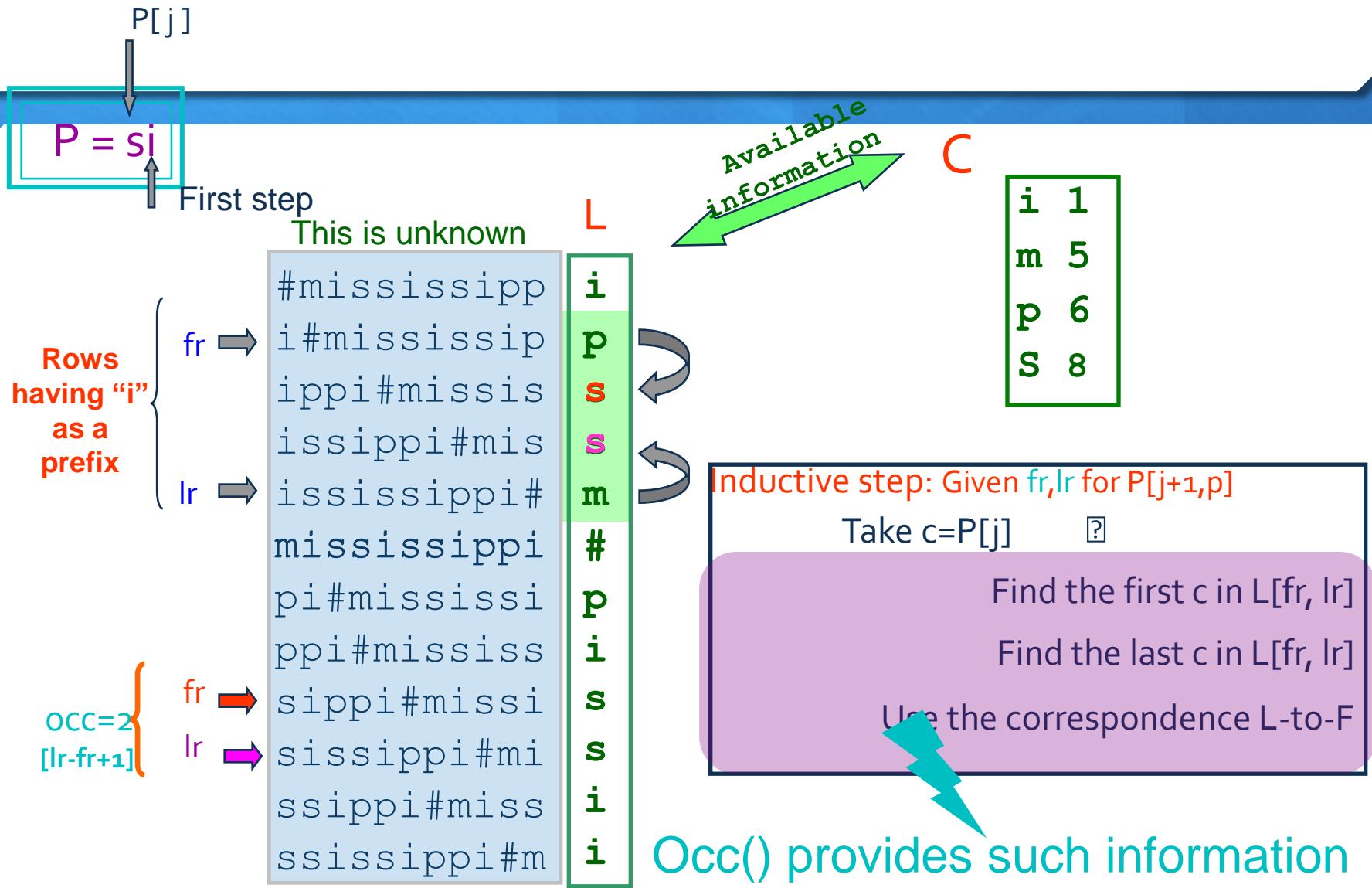
p
i
s
s
i

Available information

C

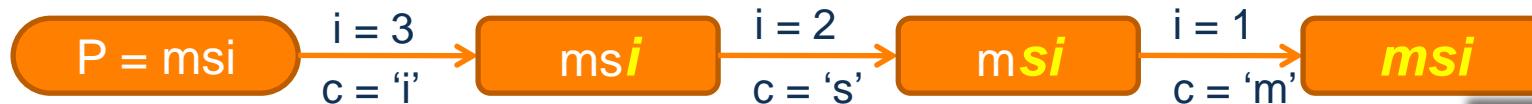
i	1
m	5
p	6
s	8

COUNT P in T: idea



Backward search for the pattern $P[1..p]$

- + It uses p iterations, from p downto 1



- + Recall that: Rows of the matrix = sorted suffixes of T

- + All the suffixes starting with P , represent a contiguous set of rows
- + Such a set starts at the position **First**
- + Ends at the position **Last**
- + Therefore, ($\text{Last} - \text{First} + 1$) gives the total number of occurrences of P

- + At the end of the i -th step,
First gives the first row
prefixed by $P[i..p]$,
and **Last** the last row
prefixed by $P[i..p]$.

F	L
# mississippi	i
i #mississip	p
p i#mississi	s
s i ssippi#mis	s
m i ssissippi#	m
p i#mississi p	
p pi#mississi	i
s ippi#missis	s
s issippi#mi	s
s sippi#miss	i
s sissippi#m	i

Algorithm `backward_search($P[1..p]$)`

- (1) $i \leftarrow p$, $c \leftarrow P[p]$, $\text{First} \leftarrow C[c] + 1$, $\text{Last} \leftarrow C[c + 1]$;
- (2) **while** (($\text{First} \leq \text{Last}$) and ($i \geq 2$)) **do**
- (3) $c \leftarrow P[i - 1]$;
- (4) $\text{First} \leftarrow C[c] + \text{Occ}(c, \text{First} - 1) + 1$;
- (5) $\text{Last} \leftarrow C[c] + \text{Occ}(c, \text{Last})$;
- (6) $i \leftarrow i - 1$;
- (7) **if** ($\text{Last} < \text{First}$) **then return** “no rows prefixed by $P[1..p]$ ” **else return** $\langle \text{First}, \text{Last} \rangle$.

BACKWARD-SEARCH EXAMPLE

○ $P = \text{pssi}$

- $i = 3$
- $c = 's'$
- $\text{First} = C['s'] + \text{Occ}('s', 1) + 1 = 8 + 0 + 1 = 9$
- $\text{Last} = C['s'] + \text{Occ}('s', 5) = 8 + 2 = 10$
- $(\text{Last} - \text{First} + 1) = 2$

F	L
# mississippi	1
i #mississipp	2
i ppi#mississ	3
i ssippi#missis	4
i ssissippi#mis	5
m ississippi #	6
p i#mississipp	7
p pi#mississi	8
s ippi#mississ	9
s ississippi#mi	10
s sippi#mississ	11
s sissippi#miss	12

C[] =	1	5	6	8
	i	m	p	s

Algorithm backward_search($P[1, p]$)

- ```

(1) $i \leftarrow p, c \leftarrow P[p], \text{First} \leftarrow C[c] + 1, \text{Last} \leftarrow C[c + 1];$
(2) while ((First \leq Last) and ($i \geq 2$)) do
(3) $c \leftarrow P[i - 1];$
(4) First $\leftarrow C[c] + \text{Occ}(c, \text{First} - 1) + 1;$
(5) Last $\leftarrow C[c] + \text{Occ}(c, \text{Last});$
(6) $i \leftarrow i - 1;$
(7) if (Last $<$ First) then return “no rows prefixed by $P[1, p]$ ” else return {First, Last}.

```

# BACKWARD-SEARCH EXAMPLE

○  $P = \text{pssi}$

- $i = 2$
- $c = 's'$
- $\text{First} = C['s'] + \text{Occ}('s', 8) + 1 = 8 + 2 + 1 = 11$
- $\text{Last} = C['s'] + \text{Occ}('s', 10) = 8 + 4 = 12$
- $(\text{Last} - \text{First} + 1) = 2$

First  
Last

| F            | L  |
|--------------|----|
| #            | 1  |
| mississippi  | 2  |
| i #mississip | 3  |
| ppi#missis   | 3  |
| ssippi#mis   | 4  |
| ssissippi#m  | 5  |
| issippi#     | 6  |
| i#mississi   | 7  |
| pi#mississi  | 8  |
| sippi#missi  | 9  |
| issippi#mi   | 10 |
| sippi#miss   | 11 |
| issippi#m    | 12 |

|         |   |   |   |   |
|---------|---|---|---|---|
| C[ ] =  | 1 | 5 | 6 | 8 |
| i m p s | i | m | p | s |

Algorithm backward\_search( $P[1, p]$ )

- (1)  $i \leftarrow p, c \leftarrow P[p], \text{First} \leftarrow C[c] + 1, \text{Last} \leftarrow C[c + 1];$
- (2) **while** ((First  $\leq$  Last) **and** ( $i \geq 2$ )) **do**
- (3)    $c \leftarrow P[i - 1];$
- (4)    $\text{First} \leftarrow C[c] + \text{Occ}(c, \text{First} - 1) + 1;$
- (5)    $\text{Last} \leftarrow C[c] + \text{Occ}(c, \text{Last});$
- (6)    $i \leftarrow i - 1;$
- (7) **if** (Last  $<$  First) **then return** “no rows prefixed by  $P[1, p]$ ” **else return**  $\langle \text{First}, \text{Last} \rangle$ .

# BACKWARD-SEARCH EXAMPLE

○  $P = \text{pssi}$

- $i = 1$
- $c = 'p'$
- $\text{First} = C['p'] + \text{Occ}('p', 10) + 1 = 6 + 2 + 1 = 9$
- $\text{Last} = C['p'] + \text{Occ}('p', 12) = 6 + 2 = 8$
- $(\text{Last} - \text{First} + 1) = 0$

First  
Last

| F | L  |
|---|----|
| # | 1  |
| i | 2  |
| i | 2  |
| i | 3  |
| i | 4  |
| i | 5  |
| m | 6  |
| p | 7  |
| p | 8  |
| s | 9  |
| s | 10 |
| s | 11 |
| s | 12 |

|        |   |   |   |   |
|--------|---|---|---|---|
| C[ ] = | 1 | 5 | 6 | 8 |
|        | i | m | p | s |

Algorithm backward\_search( $P[1, p]$ )

- (1)  $i \leftarrow p, c \leftarrow P[p], \text{First} \leftarrow C[c] + 1, \text{Last} \leftarrow C[c + 1];$
  - (2) **while** ((First  $\leq$  Last) **and** ( $i \geq 2$ )) **do**
  - (3)    $c \leftarrow P[i - 1];$
  - (4)    $\text{First} \leftarrow C[c] + \text{Occ}(c, \text{First} - 1) + 1;$
  - (5)    $\text{Last} \leftarrow C[c] + \text{Occ}(c, \text{Last});$
  - (6)    $i \leftarrow i - 1;$
  - (7) **if** (Last  $<$  First) **then return** “no rows prefixed by  $P[1, p]$ ” **else return**  $\langle \text{First}, \text{Last} \rangle$ .
-

# Backward search: first analysis

- + Backward-search has **p iterations**, and it is strongly affected from **the computation of Occ()** .
- + If we construct a bidimensional array OCC such that  $OCC[c][q]=\text{Occ}(c,q)$ , The procedure backward search takes  $O(p)$ . The table OCC needs  $O(n \log n)$  bit.
- + **THEOREM:** Combining OCC with the suffix array by using Backward search, we construct a full-text index of size  $O(n \log n)$  bit and query time  $O(p+occ)$
- + The FM-index has an implementation of the backward-search that works in  $O(p)$  by using just  $5n H_k(T) + o(n)$  bits.

# Lab Exercise

- + Write an implementation of the backward-search algorithm to find the number of occurrences of a pattern P in a text T by using the BWT (T)



# Locating a pattern in a compressed text

# LOCATE P IN T

- + Problem: Determine the positions in T of the  $(Last - First + 1)$  occurrences of the pattern P.

- + I.e.: for each  $i = first, first+1, \dots, Last$   
find the position in T of the suffix that is prefix of the rows we consider
- + Denote such a notation with  $pos(i)$

| pos(9) = 7 |   |   |   |   |   |   |   |   |    |    |    |
|------------|---|---|---|---|---|---|---|---|----|----|----|
| m          | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1          | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

| F | L                |
|---|------------------|
| # | mississippi i 1  |
| i | #mississip p 2   |
| i | ppi#missis s 3   |
| i | ssippi#mis s 4   |
| i | ssissippi# m 5   |
| m | issippi#issi # 6 |
| p | i#mississi p 7   |
| p | pi#mississ i 8   |
| { |                  |
| s | ippi#missi s 9   |
| s | issippi#mi s 10  |
| s | sippi#miss i 11  |
| s | issippi#m i 12   |

P = si

- + We cannot find  $pos(i)$  directly, but :
- + Given the row  $i(9)$  s ippi#missi s, we can find the row  $j(11)$  s sippi#miss i such that  $pos(j) = pos(i) - 1$
- + Such an algorithm is called  $backward\_step(i)$ 
  - + Running time  $O(1)$
  - + Use the previous structures

# BACKWARD\_STEP

- $L[i]$  precedes  $F[i]$  in  $T$ .
  - All the characters appears in the same relative order in  $L$  and  $F$ .
  - + So, we should just compute  $\text{Occ}(L[i], i) + C[L[i]]$

We don't have  $L[i]$ ,  
 $L$  is compressed!

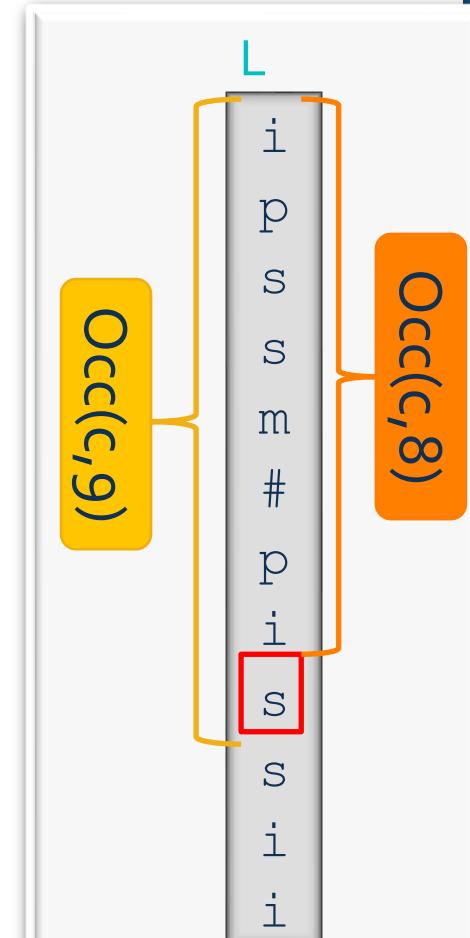
```
m issis
p i#mississipi
p pi#mississi 8
sippi#missis 9
sissippi#mis 10
sippi#missi 11
sissippi#m 12
```

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

i = 9

# BACKWARD\_STEP

- + Solution:
  - + Compare  $\text{Occ}(c, i)$  with  $\text{Occ}(c, i-1)$  for each  $c \in \Sigma \cup \{\#\}$
  - + The difference will be at  $c = L[i]$ .
  - + if  $L[i] = \#$  then return pos = 1, else return  $\text{Occ}(L[i], i) + C[L[i]]$ .
- + Call  $\text{Occ}()$  has a cost  $O(1)$
- +  $|\Sigma| = \Theta(1)$
- + So, ***backward\_step*** take the time  $O(1)$ .



|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

$$P = si \\ i = 9$$

# LOCATE P IN T: PREPROCESSING

- + The overall algorithm
- + Firstly, each row is marked each  $\lceil \log^{1+\varepsilon} n \rceil$  characters of T and their correspondent suffix in L.
- + For each marked row  $r_j$ , its position is stored  $\text{Pos}(r_j)$  in a data structure S.
- + For instance, query S to obtain  $\text{Pos}(r_3)$  will give 8.

| F | L             |
|---|---------------|
| # | mississippi i |
| i | #mississip p  |
| i | ppi#missis s  |
| i | ssippi#mis s  |
| i | ssissippi# m  |
| m | ississippi #  |
| p | i#mississi p  |
| p | pi#mississ i  |
| s | ippi#missi s  |
| s | issippi#mi s  |
| s | sippi#miss i  |
| s | sissippi#m i  |

first →

last →

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

P = si  
i = 9

# LOCATE P IN T

+ Given a row  $i$ , find Pos( $i$ ) as follows:

- If  $M_T[i]$  is a marked row, return Pos( $i$ ) from **S**. OK!
- Otherwise - use *backward\_step*( $i$ ) to find  $i'$  such that :  
 $\text{Pos}(i') = \text{Pos}(i) - 1$
- Repeat  $t$  times until a marked row is reached.
- Then – recover Pos( $i'$ ) from **S** and compute Pos( $i$ )  
by computing:  $\text{Pos}(i') + t$

| F | L             |
|---|---------------|
| # | mississippi i |
| i | #mississip p  |
| i | ppi#missis s  |
| i | ssippi#mis s  |
| i | ssissippi# m  |
| m | ississippi #  |
| p | i#mississi p  |
| p | pi#mississ i  |
| s | ippi#missi s  |
| s | issippi#mi s  |
| s | sippi#miss i  |
| s | sissippi#m i  |

first →

last →

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

$$P = si \\ i = 9$$

# The algorithm

---

**Algorithm** backward\_step( $i$ )

- (1) Compute  $L[i]$  comparing  $\text{Occ}(c, i)$  with  $\text{Occ}(c, i - 1)$  for every  $c \in \Sigma \cup \{\#\}$ .
- (2) **if** ( $L[i] = \#$ ) **then return** “ $\text{Pos}(i) = 1$ ”;
- (3) **else return**  $C[L[i]] + \text{Occ}(L[i], i)$ ;

**Algorithm** get\_position( $i$ )

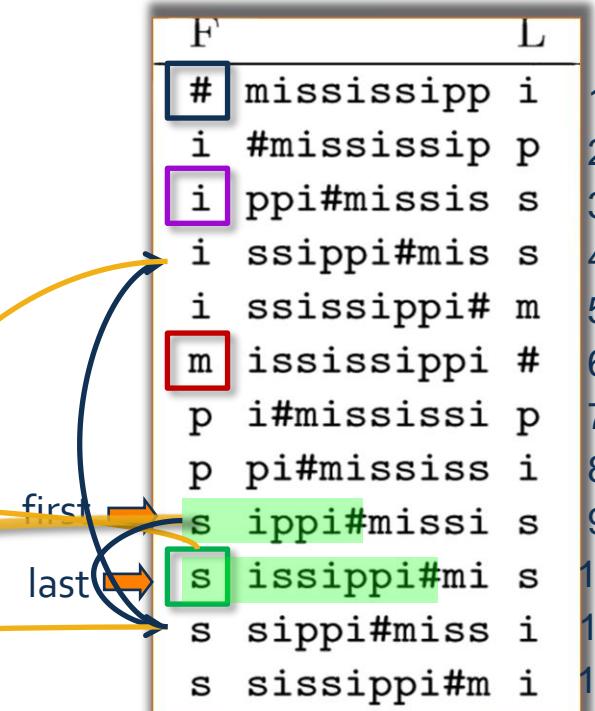
- (1)  $i' \leftarrow i$ ,  $t \leftarrow 0$ ;
  - (2) **while** row  $i'$  is not marked **do**
  - (3)      $i' \leftarrow \text{backward\_step}(i')$ ;
  - (4)      $t \leftarrow t + 1$ ;
  - (5) **return**  $\text{Pos}(i') + t$ ;
-

# LOCATE P IN T: EXAMPLE

## FIND “si”

- + For  $i = \text{last} = 10$ 
  - +  $M_T[10]$  is marked – get Pos(10) from  $S$  :
  - +  $\text{Pos}(10) = 4$
  
- + For  $i = \text{first} = 9$ 
  - +  $M_T[9]$  is not marked  $\rightarrow$  backward\_step(9)
  - +  $\text{backward\_step}(9) = M_T[11]$  ( $t = 1$ )
  - +  $M_T[11]$  is not marked.  $\rightarrow$  backward\_step(11)
  - +  $\text{Backward\_step}(11) = M_T[4]$  ( $t = 2$ )

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |



$P = si$   
 $i = 9$

# LOCATE P IN T

## Example – finding “si”

+ For  $i = \text{last} = 10$

+  $M_T[10]$  is marked – get  $\text{Pos}(10)$  da  $S$ :

+  $\text{Pos}(10) = 4$

+ For  $i = \text{first} = 9$

+  $M_T[9]$  is not marked.  $\rightarrow \text{backward\_step}(9)$

+  $\text{backward\_step}(9) = M_T[11] (t=1)$

+  $M_T[11]$  is not marked  $\rightarrow \text{backward\_step}(11)$

+  $\text{Backward\_step}(11) = M_T[4] (t=2)$

+  $M_T[4]$  is not marked  $\rightarrow \text{backward\_step}(4)$

+  $\text{Backward\_step}(4) = M_T[10] (t=3)$

+  $M_T[10]$  is marked – get  $\text{Pos}(10)$  from  $S$ .  $\text{Pos}(10) = 4$

+  $\text{Pos}(9) = \text{Pos}(10) + t = 4 + 3 = 7$

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

| F | L             |
|---|---------------|
| # | mississippi i |
| i | #mississipp p |
| i | ppi#missis s  |
| i | ssippi#mis s  |
| i | ssissippi# m  |
| m | ississippi #  |
| p | i#mississi p  |
| p | pi#mississi i |
| s | ippi#missi s  |
| s | issippi#mi s  |
| s | sippi#miss i  |
| s | sissippi#m i  |

first

last

P = si  
i = 9



# FM-Index (part 2)

(Full-text index in Minute space)

Monday May 8th, 2023

# First Step: compression

- Preprocessing of the text  $T[1,..,n]$  by the **Burrows-Wheeler Transform**
  - Output  $L[1,..,n]$  (permutation of T)
- Apply **Move-To-Front** on  $L$ 
  - Output  $L^{MTF}_{[1,...,n]}$
- Encode the equal-letter runs in  $L^{MTF}$  by using the **run-length encoding**
  - Output  $L^{rle}$
- Compress  $L^{rle}$  by using a **variable-length prefix code**
  - Output  $Z$  (on the alphabet  $\{0,1\}$ )

# The compressor

- + The following compressor is considered

BW\_RLX=bwt+mtf+rle+PC

- + Which are its performance in terms of compression ratio?

$$|Z| \leq 5n H_k(T) + O(\log n)$$

# Modified entropy

- + There is a bound in which  $H_k^*(T)$  is used:
  - + *It is called Modified empirical entropy*
  - + It represents the maximum compression ratio we can obtain by using for each symbol a codeword depending on the context of length **at most** k (instead of a context of length k).

$$n H_k(T) \leq n H_k^*(T) \leq n H_k(T) + O(\log n)$$

- + In 2001, Manzini has proved that for each k, previous compressor is bounded by

$$5n H_k^*(T) + g(k, |\Sigma|)$$

# Searching a pattern in a BWT-based compressed text



# Pattern matching

- + Problem: Given  $T$  a text and  $Z$  the text compressed by BW\_RLX. Given a pattern  $P$  we are searching for all the occurrences of  $P$  in  $T$ , with only  $Z$  available, and without decompression.
- + During the previous lecture we have seen how to count and locate the occurrence of a pattern with only the  $BWT(T)$  available, and without recovering the original text.

# Counting all the occurrences

- + Algorithm Backward-search
- + Only the output L of the BWT is used
- + Two auxiliary data structures:
  - +  $C[1, \dots, |\Sigma|]$ :  $C[c]$  contains the total number of characters in T that are alphabetically smaller than c (including repetitions)
  - +  $\text{Occ}(c, q)$ : number of occurrences of the character c in the prefix  $L[1, q]$

## Example

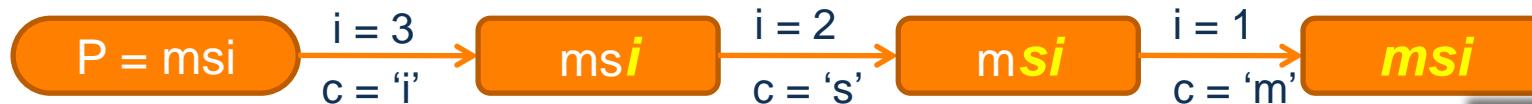
- $C[ ]$  for  $T = \text{mississippi}\#$
- $\text{occ}(s, 5) = 2$
- $\text{occ}(s, 12) = 4$

|   |   |   |   |
|---|---|---|---|
| 1 | 5 | 6 | 8 |
| i | m | p | s |

| F              | L  |
|----------------|----|
| # mississipp i | 1  |
| i #mississip p | 2  |
| i ppi#missis s | 3  |
| i ssippi#mis s | 4  |
| i ssissippi# m | 5  |
| m ississippi # | 6  |
| p i#mississi p | 7  |
| p pi#mississ i | 8  |
| s ippi#missi s | 9  |
| s issippi#mi s | 10 |
| s sippi#miss i | 11 |
| s sissippi#m i | 12 |

# Backward search for the pattern $P[1..p]$

- + It uses  $p$  iterations, from  $p$  downto 1



- + Recall that: Rows of the matrix = sorted suffixes of T

- + All the suffixes starting with  $P$ , represent a contiguous set of rows
- + Such a set starts at the position **First**
- + Ends at the position **Last**
- + Therefore, ( $\text{Last} - \text{First} + 1$ ) gives the total number of occurrences of  $P$

- + At the end of the  $i$ -th step,  
**First** gives the first row  
prefixed by  $P[i..p]$ ,  
and **Last** the last row  
prefixed by  $P[i..p]$ .

| F             | L |
|---------------|---|
| # mississippi | i |
| i #mississip  | p |
| i ppi#missis  | s |
| i ssippi#mis  | s |
| i ssissippi#  | m |
| m ississippi  | # |
| p i#mississi  | p |
| p pi#mississ  | i |
| s ippi#missi  | s |
| s issippi#mi  | s |
| s sippi#miss  | i |
| s sissippi#m  | i |

Algorithm `backward_search( $P[1..p]$ )`

- (1)  $i \leftarrow p$ ,  $c \leftarrow P[p]$ ,  $\text{First} \leftarrow C[c] + 1$ ,  $\text{Last} \leftarrow C[c + 1]$ ;
- (2) **while** (( $\text{First} \leq \text{Last}$ ) and ( $i \geq 2$ )) **do**
- (3)      $c \leftarrow P[i - 1]$ ;
- (4)      $\text{First} \leftarrow C[c] + \text{Occ}(c, \text{First} - 1) + 1$ ;
- (5)      $\text{Last} \leftarrow C[c] + \text{Occ}(c, \text{Last})$ ;
- (6)      $i \leftarrow i - 1$ ;
- (7) **if** ( $\text{Last} < \text{First}$ ) **then return** “no rows prefixed by  $P[1..p]$ ” **else return**  $\langle \text{First}, \text{Last} \rangle$ .

# Backward search: first analysis

- + Backward-search has  $p$  iterations, and it is strongly affected from the computation of  $\text{Occ}()$ .
- + If we construct a bidimensional array  $\text{OCC}$  such that  $\text{OCC}[c][q] = \text{Occ}(c, q)$ , The procedure backward search takes  $O(p)$ . The table  $\text{OCC}$  needs  $O(n \log n)$  bit.
- + THEOREM: Combining  $\text{OCC}$  with the suffix array by using Backward search, we construct a full-text index of size  $O(n \log n)$  bit and query time  $O(p + occ)$
- + The FM-index has an implementation of the backward-search that works in  $O(p)$  by using just  $5n H_k(T) + o(n)$  bits.



# Locating a pattern in a compressed text

# LOCATE P IN T

- + Problem: Determine the positions in T of the  $(Last - First + 1)$  occurrences of the pattern P.

- + I.e.: for each  $i = first, first+1, \dots, Last$   
find the position in T of the suffix that is prefix of the rows we consider
- + Denote such a notation with  $pos(i)$

| pos(9) = 7 |   |   |   |   |   |   |   |   |    |    |    |
|------------|---|---|---|---|---|---|---|---|----|----|----|
| m          | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1          | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

| F | L                |
|---|------------------|
| # | mississippi i 1  |
| i | #mississip p 2   |
| i | ppi#missis s 3   |
| i | ssippi#mis s 4   |
| i | ssissippi# m 5   |
| m | issippi#issi # 6 |
| p | i#mississi p 7   |
| p | pi#mississ i 8   |
| { |                  |
| s | ippi#missi s 9   |
| s | issippi#mi s 10  |
| s | sippi#miss i 11  |
| s | issippi#m i 12   |

P = si

- + We cannot find  $pos(i)$  directly, but :
- + Given the row  $i(9)$  s ippi#missi s, we can find the row  $j(11)$  s sippi#miss i such that  $pos(j) = pos(i) - 1$
- + Such an algorithm is called  $backward\_step(i)$ 
  - + Running time  $O(1)$
  - + Use the previous structures

# BACKWARD\_STEP

- $L[i]$  preceeds  $F[i]$  in  $T$ .
- All the characters appears in the same relative order in  $L$  and  $F$ .
- + So, we should just compute  $\text{Occ}(L[i], i) + C[L[i]]$

We don't have  $L[i]$ ,  
 $L$  is compressed!

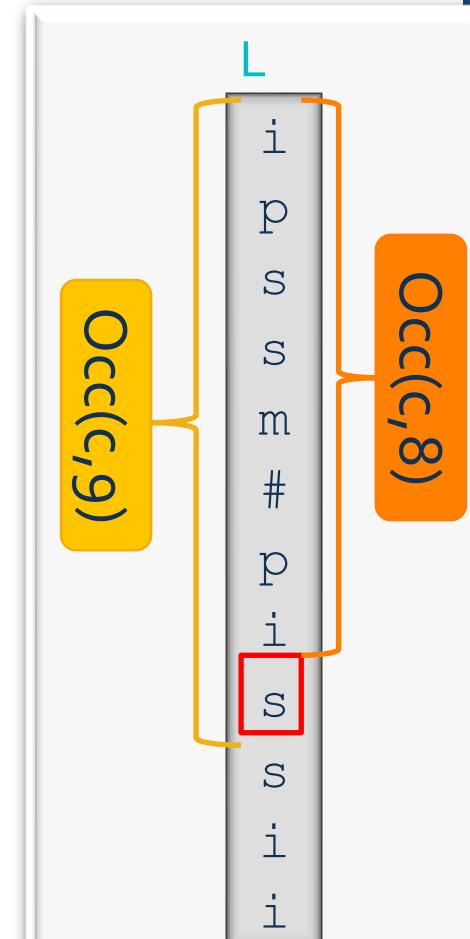
|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

$$P = si \\ i = 9$$



# BACKWARD\_STEP

- + Solution:
  - + Compare  $\text{Occ}(c, i)$  with  $\text{Occ}(c, i-1)$  for each  $c \in \Sigma \cup \{\#\}$
  - + The difference will be at  $c = L[i]$ .
  - + if  $L[i] = \#$  then return pos = 1, else return  $\text{Occ}(L[i], i) + C[L[i]]$ .
- + Call Occ() has a cost  $O(1)$
- +  $|\Sigma| = \Theta(1)$
- + So, ***backward\_step*** take the time  $O(1)$ .



|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

$$P = si \\ i = 9$$

# LOCATE P IN T: PREPROCESSING

- + The overall algorithm
- + Firstly, each row is marked each  $\lceil \log^{1+\varepsilon} n \rceil$  characters of T and their correspondent suffix in L.
- + For each marked row  $r_j$ , its position is stored  $\text{Pos}(r_j)$  in a data structure S.
- + For instance, query S to obtain  $\text{Pos}(r_3)$  will give 8.

| F | L             |
|---|---------------|
| # | mississippi i |
| i | #mississip p  |
| i | ppi#missis s  |
| i | ssippi#mis s  |
| i | ssissippi# m  |
| m | ississippi #  |
| p | i#mississi p  |
| p | pi#mississ i  |
| s | ippi#missi s  |
| s | issippi#mi s  |
| s | sippi#miss i  |
| s | sissippi#m i  |

first →

last →

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

P = si  
i = 9

# LOCATE P IN T

+ Given a row  $i$ , find Pos( $i$ ) as follows:

- If  $M_T[i]$  is a marked row, return Pos( $i$ ) from **S**. OK!
- Otherwise - use *backward\_step*( $i$ ) to find  $i'$  such that :  
 $\text{Pos}(i') = \text{Pos}(i) - 1$
- Repeat  $t$  times until a marked row is reached.
- Then – recover Pos( $i'$ ) from **S** and compute Pos( $i$ )  
by computing:  $\text{Pos}(i') + t$

| F | L             |
|---|---------------|
| # | mississippi i |
| i | #mississip p  |
| i | ppi#missis s  |
| i | ssippi#mis s  |
| i | ssissippi# m  |
| m | ississippi #  |
| p | i#mississi p  |
| p | pi#mississ i  |
| s | ippi#missi s  |
| s | issippi#mi s  |
| s | sippi#miss i  |
| s | sissippi#m i  |

first →

last →

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

$$P = si \\ i = 9$$

# The algorithm

---

**Algorithm** backward\_step( $i$ )

- (1) Compute  $L[i]$  comparing  $\text{Occ}(c, i)$  with  $\text{Occ}(c, i - 1)$  for every  $c \in \Sigma \cup \{\#\}$ .
- (2) **if** ( $L[i] = \#$ ) **then return** “ $\text{Pos}(i) = 1$ ”;
- (3) **else return**  $C[L[i]] + \text{Occ}(L[i], i)$ ;

**Algorithm** get\_position( $i$ )

- (1)  $i' \leftarrow i$ ,  $t \leftarrow 0$ ;
  - (2) **while** row  $i'$  is not marked **do**
  - (3)      $i' \leftarrow \text{backward\_step}(i')$ ;
  - (4)      $t \leftarrow t + 1$ ;
  - (5) **return**  $\text{Pos}(i') + t$ ;
-

# POS(): ANALYSIS

+ A marked row can be found in  $\log^{1+\varepsilon} n$  iterations.

+ Each iteration uses backward\_step, which is O(1).

$$O(\log^{1+\varepsilon} n)$$

+ Finding a single position

All the occurrences of P in T has a cost :

$$O(occ \cdot \log^{1+\varepsilon} n)$$

If we query S for membership query is O(1)!!

$$\log^{1+\varepsilon} n$$

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

| F | L            | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|--------------|---|---|---|---|---|---|---|---|---|----|----|----|
| # | mississippi  | i |   |   |   |   |   |   |   |   |    |    |    |
| i | #mississipp  | p |   |   |   |   |   |   |   |   |    |    |    |
| i | ppi#mississ  | s |   |   |   |   |   |   |   |   |    |    |    |
| i | ssippi#mis   | s |   |   |   |   |   |   |   |   |    |    |    |
| i | ssissippi#m  | s |   |   |   |   |   |   |   |   |    |    |    |
| m | ississippi # |   |   |   |   |   |   |   |   |   |    |    |    |
| p | i#mississipp |   |   |   |   |   |   |   |   |   |    |    |    |
| p | pi#mississi  |   |   |   |   |   |   |   |   |   |    |    |    |
| s | ippi#mississ |   |   |   |   |   |   |   |   |   |    |    |    |
| s | issippi#mi   |   |   |   |   |   |   |   |   |   |    |    |    |
| s | sippi#miss   |   |   |   |   |   |   |   |   |   |    |    |    |
| s | sissippi#m   |   |   |   |   |   |   |   |   |   |    |    |    |

first

last

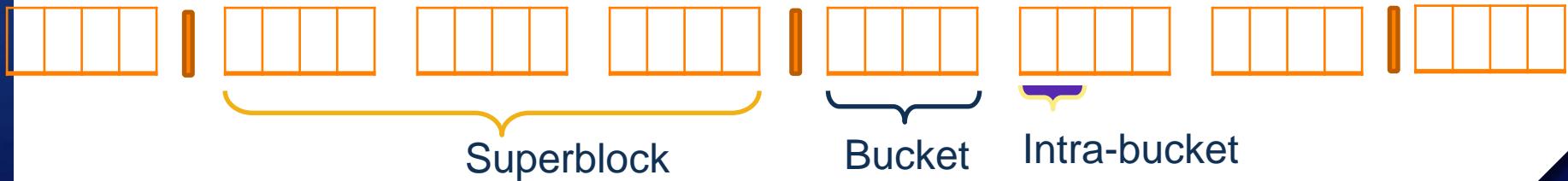
$$P = si \\ i = 9$$

# The computation of Occ()

- + Both in Backward search and in Backward step procedures the computation depends on the computation of the function Occ()
- + Recall that in the FM-index we don't have  $L = \text{BWT}(T)$ , but we have  $Z = \text{BW\_RLX}(T) = \text{PC}(\text{rle}(\text{mtf}(\text{bwt}(T))))$

# Backward search in FM-index

- + We can prove that  $\text{Occ}( )$  can be realized in time  $O(1)$  by using  $|Z| + O\left(n \cdot \frac{\log \log n}{\log n}\right)$  bits in the worst case
- + One can prove that counting the occurrences takes  $O(p)$  time, by using at most  $5n H_k(T) + O\left(n \cdot \frac{\log \log n}{\log n}\right)$  bits
- +  $Z$  is partitioned in **Bucket** and **Superblock** with some auxiliary data structures
- + **General idea:** Summing the occurrences of the characters in 3 steps



# How to implement Occ()

We assume that each run of zeroes is entirely contained in a bucket

## + Bucket

- + L is partitioned in substrings of length  $l = \Theta(\log n)$  denoted with

$$BL_i = L[(i-1)l + 1, il] \quad i = 1, \dots, n/l$$

- + Such a partition induces a partition in  $L^{MTF}$ , denoted  $BL_i^{MTF}$

|                      |                                 |                                           |
|----------------------|---------------------------------|-------------------------------------------|
| T = pipeMississippi# | n =  T  =  L  = 16              | $ BL_i^{MTF}  = 4$                        |
| L                    | i p p p s s m e i p # i s s i i | $L^{MTF}$ 2 4 0 0 5 0 5 5 4 4 5 2 5 0 1 0 |

- + Applying *run-length encoding and prefix-free encoding* on each bucket we obtain  $n/\log(n)$  buckets of **variable length** denoted  $BZ_i$



# Computing $\text{Occ}(c, q)$

- + In order to compute  $\text{Occ}(c, q)$ , the string  $L[1, q]$  is logically partitioned in the following substrings:
  1. the longest prefix of  $L[1, q]$  having length a multiple of  $l^2$
  2. the longest prefix of the remaining suffix having length a multiple of  $l$
  3. The remaining suffix of  $L[1, q]$ . Note that such a substring is a prefix of a bucket containing the character  $L[q]$
- +  $\text{Occ}(c, q)$  is computed by summing the occurrences of  $c$  in each of the previous substrings.
- + For instance, if  $l = 20$ ,  $\text{Occ}(c, 1393)$  is computed by summing the occurrences of  $c$  in  $L[1, 1200]$ ,  $L[1201, 1380]$ ,  $L[1381, 1393]$ .
- + To realize such a computation we use tables and arrays described in the following

# Computing Occ(): case 1

## + Superblocks

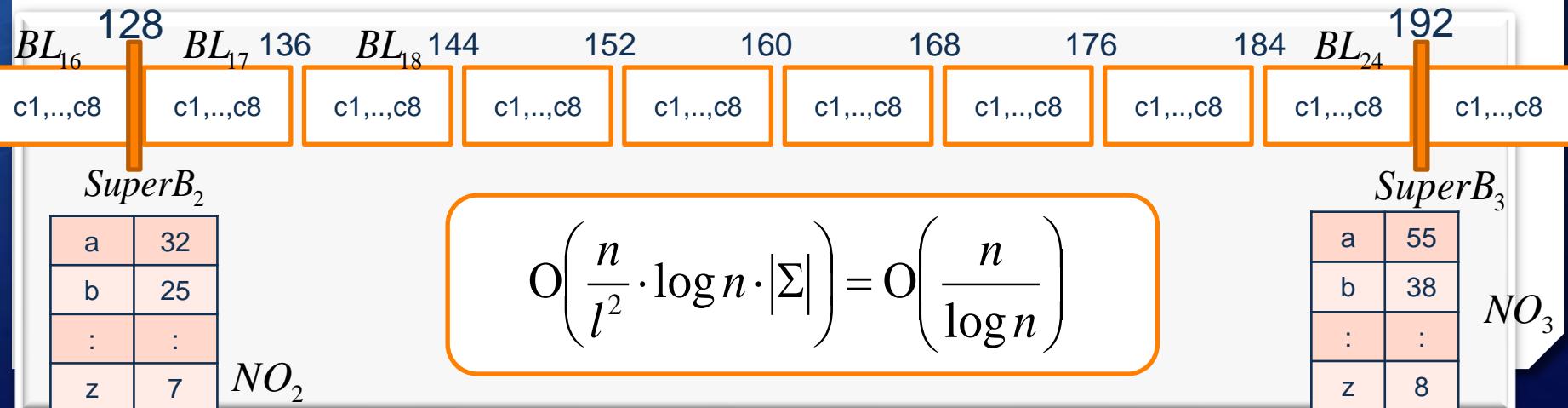
+ L is partitioned into superblocks of length  $l^2 = \Theta(\log(n)^2)$

$$|T| = |L| = 256 \quad |BL_i^{MTF}| = 8 \quad i = 1, 2, \dots, 32$$

$$|SuperB_j| = 64 \quad j = 1, 2, 3, 4$$

+ We create a **table for each superblock**, maintaining for **each character  $c \in \Sigma$** , the number of occurrences of  $c$  in  $L$ , until the end of a given superblock.

+ This means that  $NO_j[1..|\Sigma|]$  stores the occurrences of the characters in  $L[1, \dots, j \cdot l^2]$



# Computing Occ(): case 1

## + Superblocks

- + L is partitioned into superblocks

$$|T| = |L| = 256$$

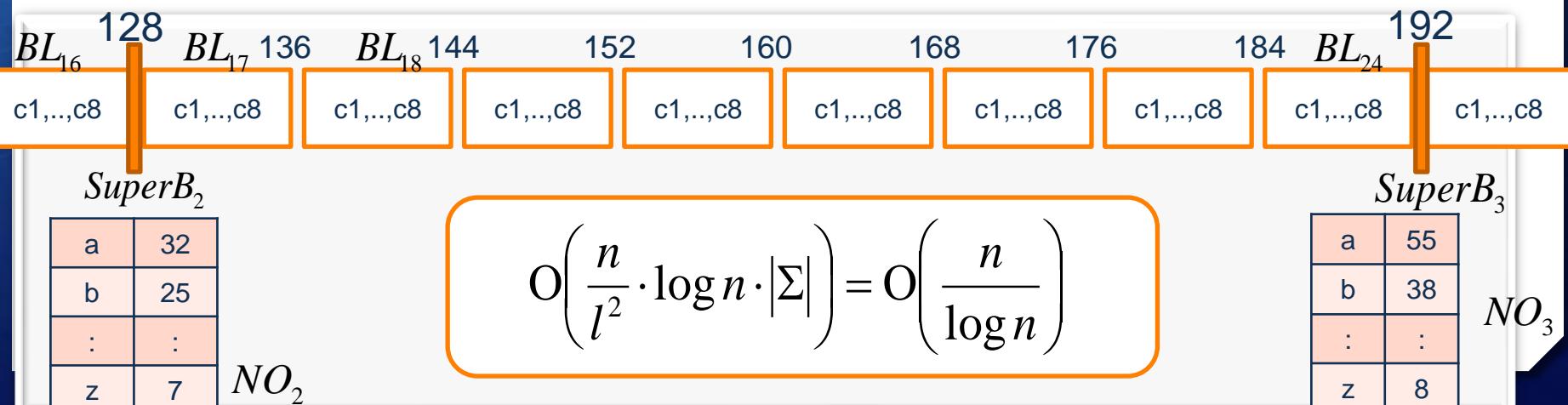
$$|BL_i|$$

$$|SuperB_j|$$

An array  $W[1..n/l^2]$  is used to store at position  $j$  the sum of the size of the buckets  $BZ_1, \dots, BZ_{jl}$  corresponding to the  $L[1..jl^2]$  after the compression step

- + We create a **table for each superblock** to store the number of occurrences of  $c$  in  $L[1..jl^2]$

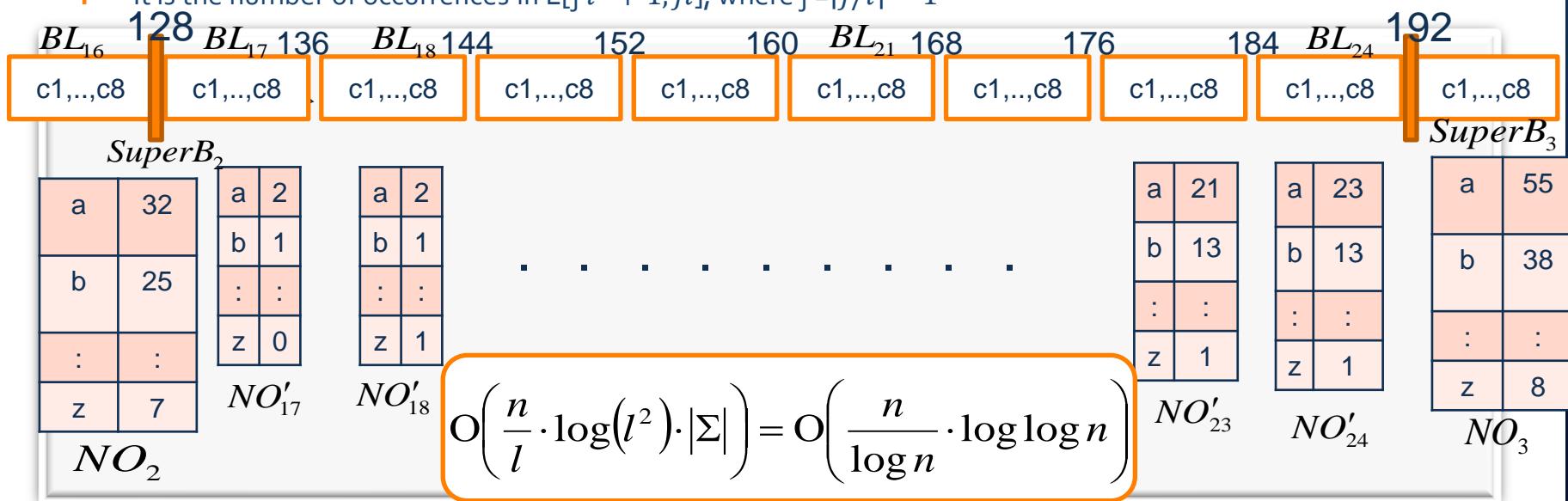
- + This means that  $NO_j[1..|\Sigma|]$  stores the occurrences of the characters in  $L[1, \dots, j \cdot l^2]$



# Computing Occ():case 2

## + Let us consider the Buckets

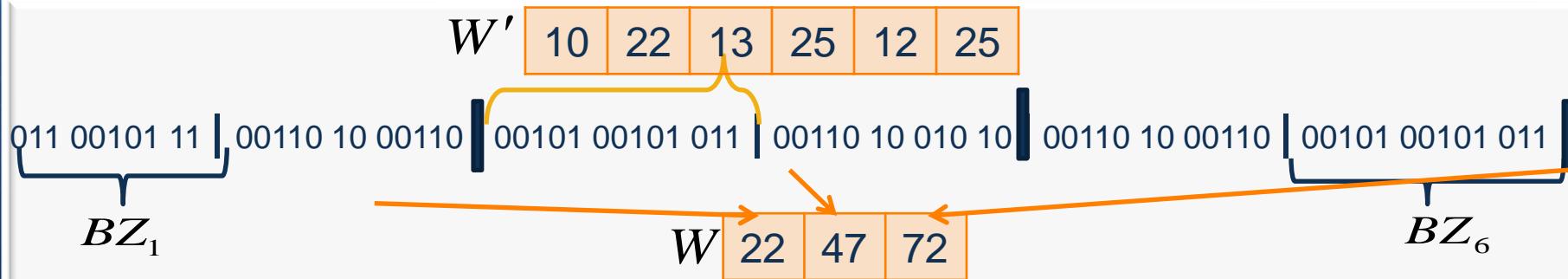
- + We create an analogous table for each bucket, in which we count the characters starting from the current **superblock**. It is denoted by  $NO'_j$
- + It is the number of occurrences in  $L[j'l^2 + 1, jl]$ , where  $j' = [j/l] - 1$



- + The only thing that remains to be done is searching inside a bucket.
  - + For instance,  $\text{Occ}(c, 164)$  will require counting in  $BL_{21}$
  - + But we have only the **compressed** string Z
  - + We need extra structures, i.e. the size of the BZ blocks.

# Size of the blocks

- + Array  $W[1, \dots, n/l^2]$  maintains for each  $SuperB_j$ , the sum of the size of the compressed buckets  $BZ_1, \dots, BZ_{j \cdot \log n}$  (in bits). They represent the portions of  $Z$  corresponding to  $L[1, jl^2]$
- + Array  $W'[1, \dots, n/l]$  maintains for each bucket  $BZ_i$ , the sum of the sizes of the buckets ( $BZ_i$  included), from the beginning of the superblock. Each block has size  $l \cdot (1 + 2\lfloor \log |\Sigma| \rfloor)$



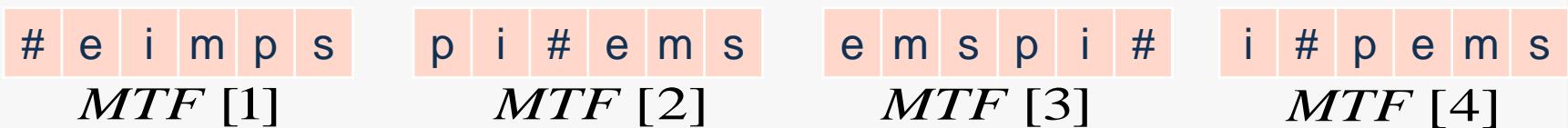
$$|W| = O\left(\frac{n}{l^2} \cdot \log n\right) = O\left(\frac{n}{\log n}\right)$$

$$|W'| = O\left(\frac{n}{l} \cdot \log(l^2)\right) = O\left(\frac{n}{\log n} \cdot \log \log n\right)$$

# Computing Occ(): case 3

- + We have the compressed bucket  $BZ_i$  | 011 00101 11 |
  - + And we have  $h$  (how many characters to check in  $BZ_i$ ).
  - + Note that  $BZ_i$  contains Move-To-Front compressed informations!
- We need additional structures!
- + For each  $i$ , before encoding the bucket  $BL_i$  by Move-To-Front, we store the state of the MTF table

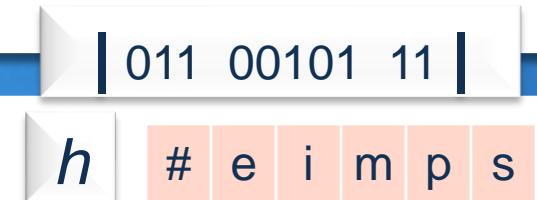
| L         | T = pipeMississippi# |   |   |   |   |   |   |   |   |   |   |   |   |   |   | $ T  =  L  = 16$ |
|-----------|----------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|------------------|
| $L^{MTF}$ | i                    | p | p | p | s | s | m | e | i | p | # | i | s | s | i | i                |
|           | 2                    | 4 | 0 | 0 | 5 | 0 | 5 | 5 | 4 | 4 | 5 | 2 | 5 | 0 | 1 | 0                |



$$O\left(\frac{n}{\log n} \cdot |\Sigma| \log |\Sigma|\right) = O\left(\frac{n}{\log n}\right)$$

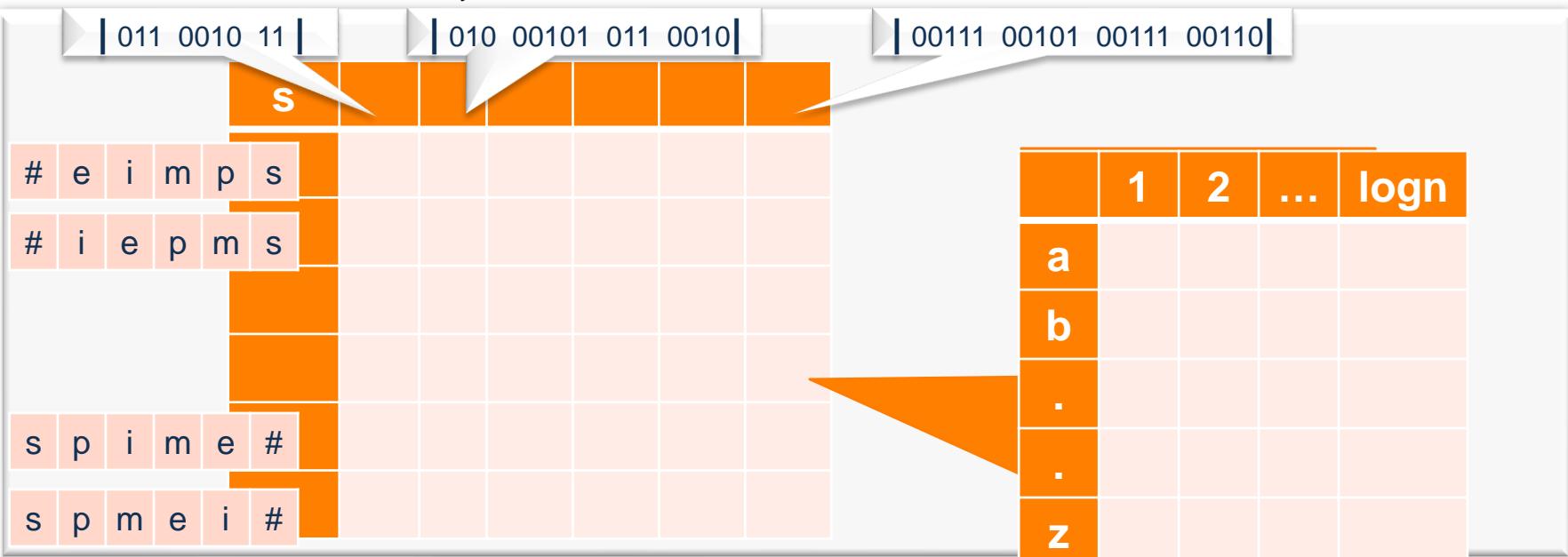
# Computing Occ()

- How to use  $MTF_i$  to count inside  $BZ_i$ ?



- Final structure!

- We construct a table  $S[c, h, BZ_i, MTF[i]]$ , in which we store the number of occurrences of 'c' in the first  $h$  characters of  $BL_i$ 
  - It is possible since  $BZ_i$  and  $MTF_i$  fully determine  $BL_i$



- Maximal size of a compressed bucket:  $BZ_i$

$$l \cdot (1 + 2 \lfloor \log |\Sigma| \rfloor)$$

- Number of possible compressed buckets:

$$2^{l \cdot (1 + 2 \lfloor \log |\Sigma| \rfloor)} = 2^t$$

- Number of possible MTF tables:

$$2^{|\Sigma| \log |\Sigma|}$$

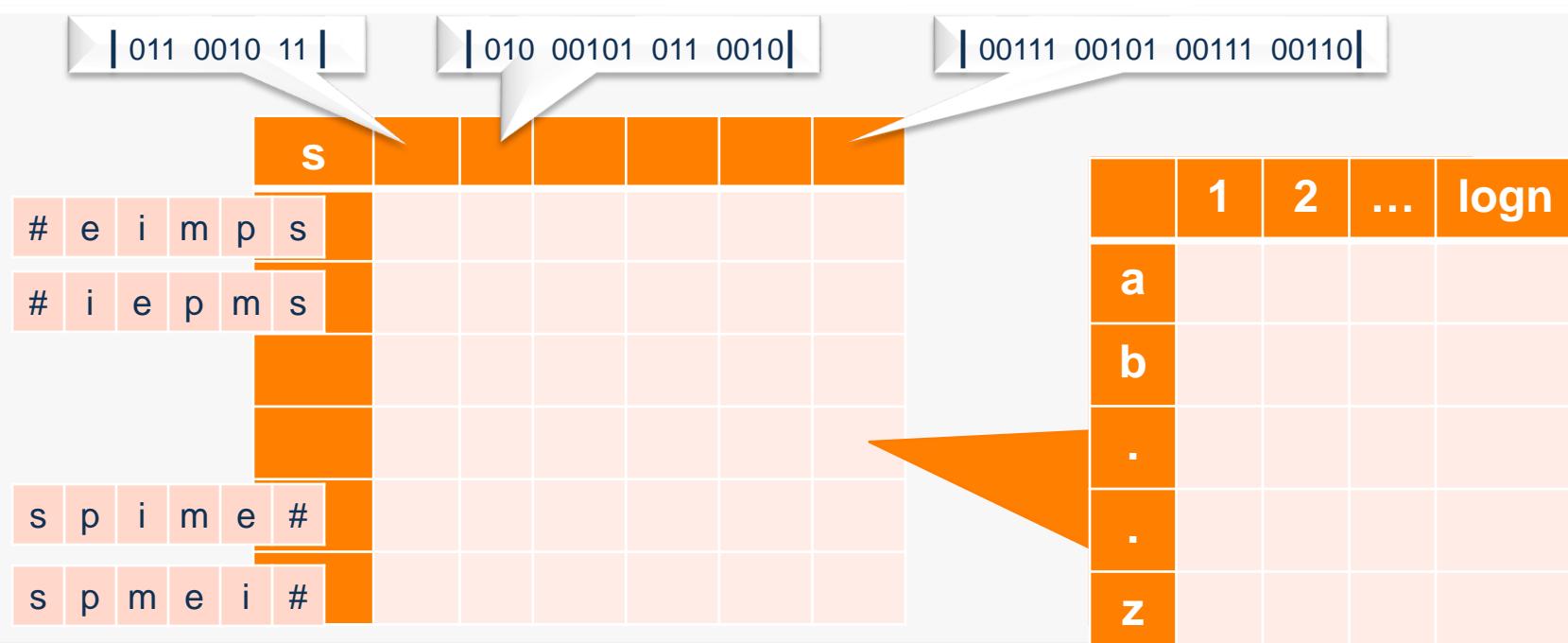
- Size of the internal table:

$$|\Sigma| l$$

- Size of each entry:

$$\log l$$

- Total size of  $S[ ]$ :  $O(2^t l \log l) = O(n \cdot \log n \cdot \log \log n)$



# How to compute Occ(c,q)

Occ(k,166)

Which bucket contains q?

$$\text{Find } i \text{ s.t. } BL_i \cdot i = \lceil \frac{q}{\log n} \rceil$$

Find the position in  $BL_i$

$$h = q - (i-1) \cdot \log n$$

$NO_t[c]$

Find the superblock  $SuperB_t$  that precedes  $BL_i$

$NO'_{i-1}[c]$

Find the position of  $BZ_i$  in Z:

Occurrences  
in  $BL_i[1, h]$

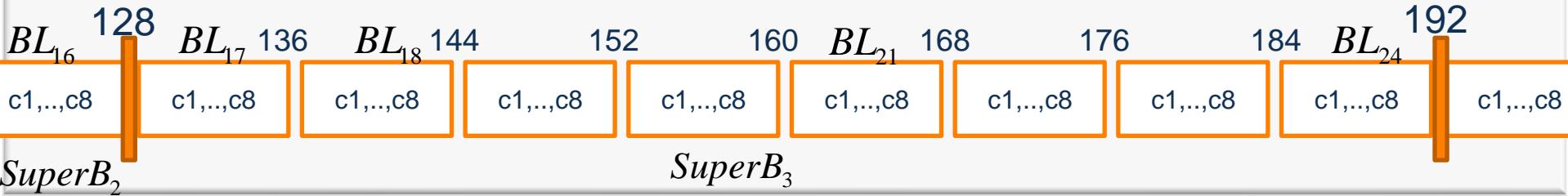
$$166/8 = 20.75 \\ \rightarrow i = 21$$

$$h = 166 - 20 \cdot 8 \\ \rightarrow h = 6$$

$$(21/8) = 2.625 \\ \rightarrow t = 2$$

$S[c, h, BZ_i, MTF[i]]$

Position of the compressed bucket  
 $W[2] + W'[20] + 1$



# How many bits?



- Maximal size of a compressed bucket:  $BZ_i$

$$l \cdot (1 + 2 \lfloor \log |\Sigma| \rfloor)$$

- Number of possible compressed buckets:

$$2^{l \cdot (1 + 2 \lfloor \log |\Sigma| \rfloor)} = 2^t$$

- Number of possible MTF tables:

$$2^{|\Sigma| \log |\Sigma|}$$

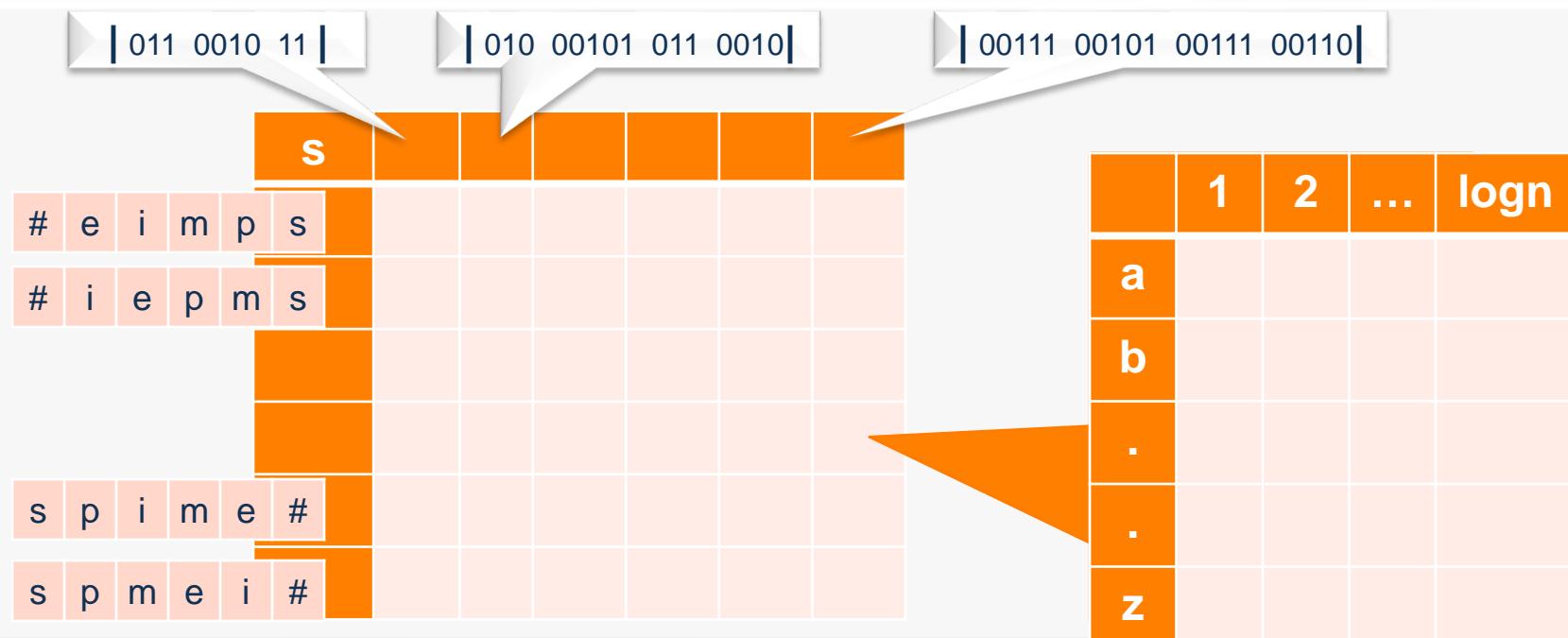
- Size of the internal table:

$$|\Sigma| l$$

- Size of each entry:

$$\log l$$

- Total size of  $S[ ]$ :  $O(2^t l \log l) = O(n \cdot \log n \cdot \log \log n)$



- Maximal size of a compressed bucket:  $BZ_i$
    - Number of possible compressed buckets:
- $$l \cdot (1 + 2 \lfloor \log |\Sigma| \rfloor)$$
- $$2^{l \cdot (1 + 2 \lfloor \log |\Sigma| \rfloor)} = 2^t$$
- Number of possible MTF tables:
- $$2^{|\Sigma| \log |\Sigma|}$$
- Size of the internal table:
    - Size of each entry:
- $$\begin{matrix} |\Sigma| l \\ \log l \end{matrix}$$
- Total size of  $S[ ]$ :
- $$O(2^t l \log l) = O(n \cdot \log n \cdot \log \log n)$$

- However, a linear size index is not interesting when  $n$  is large.
- We want to maintain a sublinear size, i.e:  $O(n^\alpha), \alpha < 1$
- Choose the size of the bucket  $l$  such that  $l \cdot (1 + 2 \lfloor \log |\Sigma| \rfloor) = \alpha \log n \quad \alpha < 1$
- We obtain  $2^{\alpha \log n} = n^\alpha = O(n) \rightarrow |S| = O(n^\alpha \cdot \log n \cdot \log \log n)$

# ANALYSING OCC( )

- Summing up:

- NO &  $W$  require  $O\left(\frac{n}{\log n}\right)$
- $NO'$  &  $W'$  require  $O\left(\frac{n}{\log n} \cdot \log \log n\right)$
- MTF require  $O\left(\frac{n}{\log n}\right)$
- S needs  $O(n^\alpha \cdot \log n \cdot \log \log n)$

- Total Size:  $|Z| + O\left(\frac{n}{\log n} \cdot \log \log n\right)$
- Total Time:  $O(1)$

# e i m p s

$W'$  10 22 13 25 12 25

MTF<sub>1</sub>

011 00101 11 | 00110 10 00110 | 00101 00101 011 | 00110 10 010 10 | 00110 10 0

|   |    |
|---|----|
| a | 23 |
| b | 13 |
| : | :  |
| z | 1  |

$NO'_1$

|   |    |
|---|----|
| a | 55 |
| b | 38 |
| : | :  |
| z | 8  |

$NO_2$

$W$  22 47 72



# TABLE S FOR LOCATE: SPACE

$$O\left(\frac{n}{\log^{1+\varepsilon} n}\right)$$

- + Number of marked rows
- + The implementation of  $S$  uses  $O(\log \log n)$  bits for each marked row,

and  $O(\log n)$  additional bits to store  $\text{Pos}(i)$

- + Then  $S$  requires  $O\left(\frac{n}{\log^{1+\varepsilon} n}(\log \log n + \log n)\right)$

We have seen that for counting the occurrences we need

$$|Z| + O\left(\frac{n}{\log n} \cdot \log \log n\right) \text{ bits, then we choose } \varepsilon \text{ between 0 and 1.}$$

Decreasing  $S$ , does not reduce the asymptotic space  $O\left(\frac{n}{\log n} \cdot \log \log n\right)$

Moreover, the larger the value of  $\varepsilon$ , the larger the search time,  
the smaller the space

| F | L             |
|---|---------------|
| # | mississippi i |
| i | #mississip p  |
| i | ppi#missis s  |
| i | ssippi#mis s  |
| i | ssissippi# m  |
| m | ississippi #  |
| p | i#mississi p  |
| p | pi#mississ i  |
| s | ippi#missi s  |
| s | issippi#mi s  |
| s | sippi#miss i  |
| s | sissippi#m i  |

P = si  
i = 9

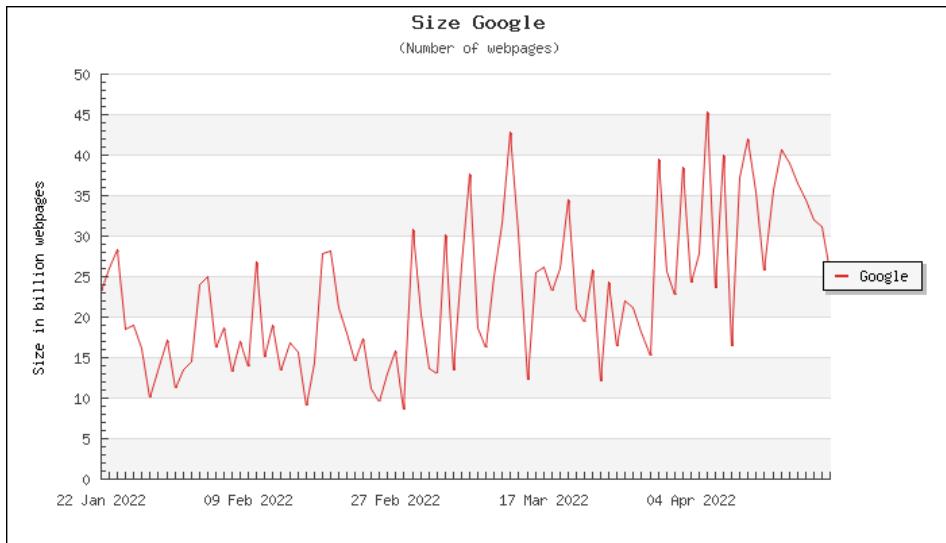
|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| m | i | s | s | i | s | s | i | p | p  | i  | #  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# How to manage this Big DATA

- + In addition to acquisition and distribution issues, EFFICIENT METHODOLOGIES FOR THE STORAGE AND ANALYSIS OF THESE DATA ARE ESSENTIAL
- + TWO IMPORTANT TASKS

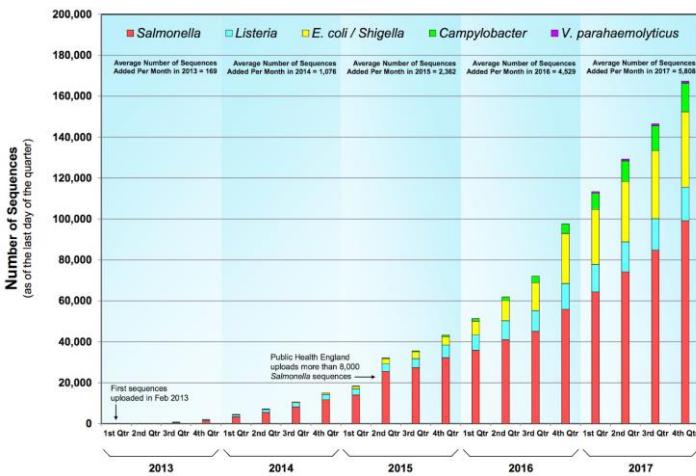
COMPRESSORS  
reduce the size

INDEXING :  
searching for a  
pattern



# Big Data vs Moore's Law

- + The evolution of technology certainly allows for better data management, but still one pays a price proportional to the size of the data in terms of amount of computation, data storage, network usage, energy consumption, etc. This becomes very "problematic in those contexts (such as Bioinformatics or Astronomy) where data growth has outpaced Moore's law". (Navarro, ACM Comput. Surv., 2022)
- + NEW IDEAS AND ALGORITHMS ARE NEEDED!



The enormous amount of data available challenges our ability to manage and analyse it even in compressed form



# Good news: highly repetitive texts!

- + Especially in the case of textual data, many fast-growing data collections have the characteristic of being very repetitive, i.e. their information content is orders of magnitude smaller than their size.
  - + Genome databases typically store many genomes of the same species (PANGENOMES). Two human genomes differ by about 0.1%.
  - + WIKIPEDIA collects multiple versions of each paper, in 2015 it stored in 10 terabytes over 20 versions per paper (i.e. quasi-repeats), which grow faster than the original papers
  - + A software repository with versions such as GitHub stored over 20 terabytes in 2016 and also reported over 20 versions per project. A ratio of commits (new versions) to creates (new projects) of approximately 20 was observed
  - + A 40%-80% degree of duplication was observed in tweets, emails, web pages, and software repositories
- + Compressors using text statistics are not able to make the most of this repetitiveness.
- + GOAL: Searching new compression paradigms that allow for efficient compression of large, highly repetitive datasets, enabling efficient indexing operations i.e. counting and localisation of patterns, using the compressed data directly (without decompressing)

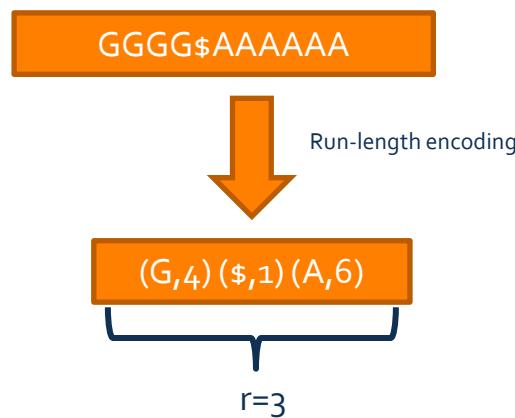
# A new data structure : R-index

- + Described in [Gagie, Navarro, Prezza: J. ACM 2020]
- + It is an evolution of the FM-index. It is based on BWT

# Takes advantage from a low number of runs

- + EXAMPLE: W=AGAAGAAGAG\$

```
$AGAAGAAGAG
AAGAAAGAG$AG
AAGAG$AGAAG
AG$AGAAGAAG
AGAAGAAGAG$
AGAAGAG$AGA
AGAG$AGAAAGA
G$AGAAGAAGA
GAAGAAGAG$A
GAAGAG$AGAA
GAG$AGAAGAA
```



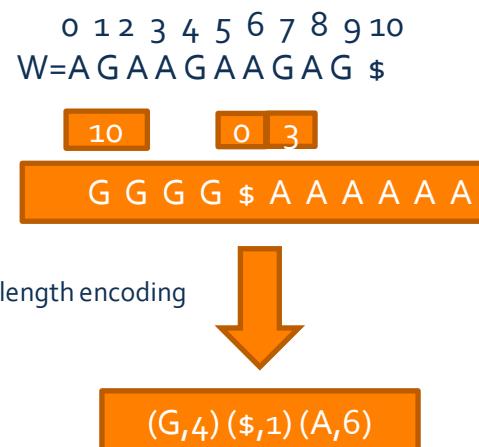
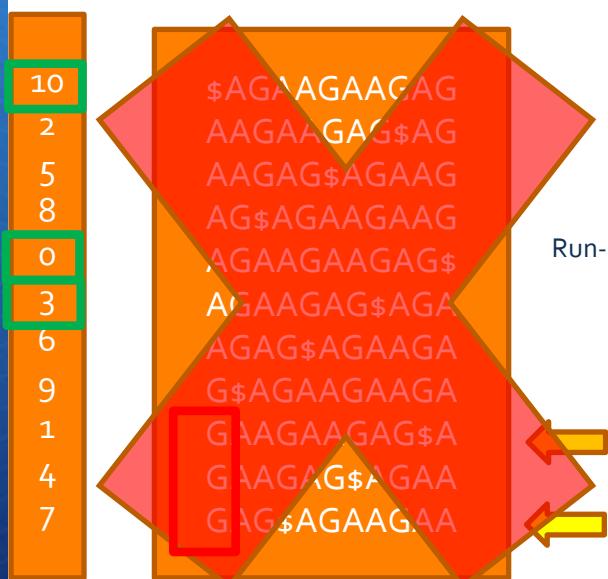
| Dataset       | $\sigma$ | $n$         | $r_f$     |
|---------------|----------|-------------|-----------|
| DNA           | 10       | 629,140,006 | 1,287,509 |
| boost         | 96       | 629,145,600 | 62,026    |
| einstein      | 194      | 629,145,600 | 958,672   |
| world_leaders | 89       | 46,968,181  | 573,487   |

H. Bannai et al., Theoret. Comput. Sci. 2020

- + In highly repetitive text collections, there tends to be fewer runs of the same symbols
- + DNA: A pseudo-real DNA sequence consisting of 629145 copies of a DNA sequence of length 1000 (Human Genome) where each character has been mutated with probability  $10^{-3}$ .
- + boost: concatenated versions of the GitHub boost library.
- + einstein: concatenated versions of the Wikipedia paper for Albert Einstein.
- + world\_leaders: a collection of all pdf files of CIA world leaders from January 2003 to December 2009

# Locate the occurrences of a pattern

- + QUERY: what and where are the positions where P=GA appears?



- + Only  $r$  positions are stored for characters that are at the beginning of runs in the BWT (and other small auxiliary data structures)
- + LF matching is used, i.e. equal characters that are consecutive in the BWT correspond to equal and consecutive characters in the first column of the BWT matrix.

# Lab Exercise

- + Write the implementation of the function to recover the original text from its BWT.
- + Write an implementation of the backward-search algorithm to find the number of occurrences of a pattern P in a text T by using the BWT (T)

```
def bw(s):
 n = len(s)
 m = sorted([s[i:n]+s[0:i] for i in range(n)])
 #l = m.index(s)
 L = ".join([q[-1] for q in m])
 return L
```

# Experiments on synthetic data

- + Low complexity words: finite Sturmian words
- + How to generate: Let  $d_1, d_2, \dots, d_n, \dots$  be sequence of natural numbers, with  $d_1 \geq 0$  and  $d_i > 0$  for  $i = 2, \dots, n, \dots$
- + Consider the following sequence of words  $\{s_n\}_{n \geq 0}$ :  $s_0 = b$ ,  $s_1 = a$ , and  $s_{n+1} = s_n^{d_n} s_{n-1}$  per  $n > 1$ .
- + Each word  $s_n$  is a finite Sturmian word. Fibonacci words are obtained by considering the sequence  $(1, 1, 1, 1, \dots)$
- + Which is the number of BWT-runs for such words. Which is the number of phrases of LZ77 or Lzss?



# Kolmogorov complexity

Monday, May 15<sup>th</sup>, 2023

# Algorithmic information theory

- + Algorithmic information theory, is a branch of computer science and mathematics that studies the relationship between information and computation. It deals with the concepts of information, randomness, and complexity, and provides a framework for understanding the limits of what can be computed and the limits of what can be known. The theory was developed by Soviet mathematician Andrey Kolmogorov in the 1960s and has since been extended by other researchers.
- + In a 1965 paper, Kolmogorov observed that Shannon's theory had some limitations. In fact, Shannon's theory of information, introduced in 1948, is based on the probability distribution of a source. However, Kolmogorov was not completely satisfied with Shannon's formalisation of the notion of information. The main criticism was that it allows the amount of information to be measured only of variables generated by a source and not of objects at hand. Through entropy, Shannon can calculate the information associated with a probability distribution on texts or images, but not the information contained in a given text or image.
- + Algorithmic information theory links the notion of information with the notion of complexity. That is, the more difficult a string is to represent, the more information it carries. This makes it possible to speak of information associated with specific strings or texts without referring to the context in which the given text was produced.

# Example

- +  $W=11111111$ 
  - + According to Shannon, it does not carry information. The string is actually not very complex to describe.
- +  $W=10101010$ 
  - + What is Shannon entropy?
- +  $W=1001110010$ 
  - + What is Shannon entropy? Is it more difficult to describe than the previous one?

# Random strings

- + Given the two strings

S\_1="00010001000100010001000100010001000100010001"

S\_2="1110010111001101101011010011001111101101",

Which one is random? Why?

# The first string is not random

- + In fact it has a clear and evident regularity: output ten times the strings with 3 0's followed by 1.
- + Then, is the second string random? Why?

# And the second string?

- + It could have a hidden rule.
- + For example, could you get a rule like the following?
  - + Take the tenth prime number
  - + Sum the age of your colleague X
  - + Reverse the order of the digits
  - + If the number is even add 7, otherwise multiply by 8
  - + Convert the number to binary notation
- + There are an infinite number of rules, if not this one we could certainly find one that generates our string.
- + So, when is a string random?

# Information and algorithmic complexity

- + Intuitively a string is random if it is complex to describe.
- + Kolmogorov in 1965 gave a formal definition in which he linked the amount of information in a text to the complexity needed to describe it.
- + Kolmogorov defines the amount of information contained in an object or in a string  $x$  as **minimum length of the shortest program (in a given programming language) that can produce the object or the string as its output**.  
Such a length is the **Kolmogorov complexity** of  $x$ .
- + In the same period, analogous concepts were formulated independently by two other mathematicians, Solomonoff (American of Russian origin) in 1964 and Chaitin (Argentinian-American) in 1966

# Kolmogorov complexity: formal definition

- + Let us consider a universal Turing machine (i.e. capable of simulating any Turing machine)  $U$ . Equivalently, we can consider a computer with infinite (or arbitrarily large) memory on which any programming language can run.
- + Given a universal Turing machine  $U$ , or equivalently given a programming language, the amount of information contained in a string  $S$  turns out to be the number:

$$K_U(S) = \min\{l(p) \mid U(p) = S\}$$

where  $p$  is a program (expressed in bits),  $l(p)$  is its length and  $U(p)$  denotes the string produced by machine  $U$  when executing program  $p$ . Complexity has been defined for binary strings, but it can be extended to other objects.

- + Kolmogorov complexity provides a way to quantify the amount of information or complexity present in a string, independent of any specific encoding or representation

# Let us come back to the example

- + For the first string, a program could be:

for 1 to 10

    print "0001"

    next

- + This message is shorter than

    print "0001000100010001000100010001000100010001"

since "print", "next" etc. need only a few bits to be encoded.

# Invariance of K

- + Theorem: Given two universal Turing machine U and V, then for each string  $x$ , one has that  $|K_U(x) - K_V(x)| \leq C(U, V)$ , where  $C(U, V)$  depends only on U and V and it does not depend on the string  $x$ .
- + Proof: Let  $p_v$  be the shortest program for the machine V generating the string  $x$ . Let  $s_v$  be the program that simulates the machine V by using the machine U. Then, a program  $p$  for U is  $s_v p_v$ . This means that  $I(p) = I(s_v) + I(p_v) = c_v + I(p_v)$ , where  $c_v$  is a constant. This implies that  $K_U(x) \leq K_V(x) + c_v$ . Analogously,  $K_V(x) \leq K_U(x) + c_U$ . The thesis follows by considering  $C(U, V)$  the maximum value between the two constants.

# Algorithmic randomness and Kolmogorov complexity

- + Kolmogorov complexity is closely related to the concept of algorithmic randomness. A string is considered algorithmically random if it has high Kolmogorov complexity, meaning that it cannot be significantly compressed or described by any algorithm or pattern shorter than the string itself. Random strings have no discernible structure or regularity.
- + For example, a sequence of coin tosses that alternate between heads and tails (HTHTHT...) can be described concisely by a simple algorithm, making its Kolmogorov complexity relatively low. On the other hand, a completely random sequence of coin tosses (e.g., HTHHTTTHT...) cannot be compressed or described by any concise algorithm, leading to a high Kolmogorov complexity.

# Random strings

- + The fundamental idea is that if the complexity of a string (the length of the shortest description) is equal to the length of the string itself, then the description (the string from which the description method returns the string to be described) is not substantially different from the string itself, i.e., put more explicitly, it is the string itself that is the shortest description of itself
- + In this case, the string is called uncompressible.
- + Interpreting this as a 'high complexity' meaning, we say that a string is random if it is uncompressible.

# Random strings: do they exist?

- + A string  $x$  is random (also called Kolmogorov random) if  $K(x) \geq |x|$
- + For every  $n > 0$ , there exist uncompressible strings of length  $n$ . Let us consider all the programs having length smaller than  $n$ , their number is  $2^n - 1$ . This means that there exists at least one string having a program of length at least  $n$ .

# Kolmogorov complexity is not computable

- + THEOREM: There is no algorithm or computer program that can calculate the exact Kolmogorov complexity  $K(x)$  of an arbitrary string  $x$ .
- + The proof of the non-computability of Kolmogorov complexity is derived from the fact that there is no universal algorithm that can compute the shortest program for any input.
- + The non-computability of Kolmogorov complexity is related to the halting problem of Turing machines. The halting problem, as formulated by Alan Turing, is the problem of determining, given a Turing machine and an input, whether the machine will eventually halt or run indefinitely on that input. Turing's proof of the undecidability of the halting problem proved that there is no algorithm or Turing machine that can solve this problem for all possible inputs.
- + If there were a computable algorithm to calculate the exact Kolmogorov complexity, it would essentially solve the halting problem. Given a Turing machine and an input, one could check if there exists a program that produces the desired output and has a shorter length than any program that does not halt. If such a shorter program exists, it would imply that the Turing machine halts; otherwise, it would run indefinitely. Since the halting problem is undecidable, it follows that the exact Kolmogorov complexity is also non-computable. Calculating the exact Kolmogorov complexity would require solving the halting problem, which is impossible for all inputs.

# Applications: comparing strings

- + Although the exact Kolmogorov complexity is not computable, it is still a valuable theoretical concept that helps us reason about information and complexity.
- + We can approximate the Kolmogorov complexity by considering specific programming languages or using compression algorithms as proxies. These approximations provide insights and practical applications in various fields, even if they do not yield the exact Kolmogorov complexity value.
- + In 2004 Chen, Li and Vitanyi introduced a similarity measure between strings that uses Kolmogorov complexity, called **Information Distance**:  
$$E(x, y) = \max\{K(x|y), K(y|x)\}$$
- + In real experiments, the Kolmogorov complexity is approximated by using real compressors.

# Problem

- + The goal of lossless text compression is to reduce the size of a given string by exploring the regularities of the text to be compressed.
- + Kolmogorov's complexity cannot be calculated. Since we cannot reach this limit, it can only be approximated.
- + One of the goals of research today is to find a model that captures or measures the degree of regularity of a text. There are many candidates, measures  $r$  and  $z$  being among them.
- + Entropies, even empirical entropies of order 0 or order  $k$ , are not always able to represent the lower bound of the compression of any text. For example, it can be proven that if the text contains many repetitions, the empirical entropy of order  $k$  remains roughly the same.
- + This particular weakness has generated much interest in recent decades in algorithms capable of directly exploiting text repetition to beat the lower bound of entropy on highly repetitive texts.
- + Both dictionary-based compressors (including grammar-based compressors) and BWT-based compressors fit into this context.
- + At this point, a natural question is whether there is a unifying framework for all compression schemes and a measure that can be used to better evaluate the effectiveness of all compressors.



# Complexity of a text

# Complexity of a text

- + The Kolmogorov complexity of a string  $x$  is defined as the length of the smallest program used by a universal Turing machine to generate the string  $x$ .
- + This Turing machine model for string representation is too powerful to be exploited effectively.
- + The idea was to weaken the string representation model from Turing machines to context-free grammars.
- + Being able to capture the regularities of a text is in strong correlation with the rules used to generate the text itself (inductive inference). This is also related to the rules for learning a language.

# Smallest grammar problem

- + Given a string  $x$ , which is the smallest CF grammar generating precisely the string  $x$ ?
- + For instance: given the string  $x=a\ rose\ is\ a\ rose\ is\ a\ rose$ , the smallest CF grammar generating  $x$  is:
  - +  $S \rightarrow BBA; A \rightarrow a\ rose; B \rightarrow A\ is;$
- + The size of a grammar is defined as the total number of symbols on the right-hand side of all rules. In the example above, the grammar has size 14.
- + More formally: let  $G=(\Sigma, V, P, S)$  be a CF grammar, where  $P=\{A_1 \rightarrow v_1, A_2 \rightarrow v_2, \dots, A_k \rightarrow v_k\}$ . Then  $|G|=\sum_{i=1}^k |v_i|$  is the size of  $G$ .
- + Smallest grammar problem (SGP): Given a string  $x$ , finding the CF grammar  $G$  having minimum size such that  $L(G)=\{x\}$ .

# Example

- + Let us consider the following strings:
  - +  $f_0 = b, f_1 = a, f_n = f_{n-1}f_{n-2}$
  - + Let us consider  $f_6 = abaababaabaab$ . It has been shown that the smallest grammar (in *Chomsky normal form*) is  $G_6$  ( $F_6$  is the axiom)  
 $F_0 \rightarrow b$   
 $F_1 \rightarrow a$   
 $F_2 \rightarrow F_1 F_0$   
 $F_3 \rightarrow F_2 F_1$   
 $F_4 \rightarrow F_3 F_2$   
 $F_5 \rightarrow F_4 F_3$   
 $F_6 \rightarrow F_5 F_4$
  - + In this case  $|G_6| = 12$ . In general,  $|G_n| = 2n$ .

A grammar is in the Chomsky normal form if the productions have the following form:  
 $A \rightarrow BC$ , or  $A \rightarrow a$ , or  $S \rightarrow \epsilon$ , where  $A, B, C$  are non-terminal symbols  
(with  $B, C$  different from the axiom  $S$ ),  $a$  is a terminal symbol.

# SGP is NP-complete

- + Going from the Turing machine to context-free grammars has reduced the complexity of the problem from the reign of undecidability to mere intractability.
- + Theorem (Gold 1967, Storer 1982): SGP is NP-complete.
- + However, it is shown that one can efficiently approximate the 'grammatical complexity' of a string.

# Algorithmic approximation

- + We are looking for a polynomial approximation algorithm A such that for each string w produces a grammar G such that  $L(G)=\{w\}$
- + We will focus on the approximation ratio of algorithm A, i.e.

$$\rho_A(n) = \max_{w \in \Sigma^n} \frac{|A(w)|}{|G_{\min}(w)|}$$

where  $A(w)$  is the grammar produced by A and  $G_{\min}$  is the grammar of minimal size for w.

- + We are interested on the worst case of the algorithm.
- + Theorem (Charikar et al. 2005): There is no polynomial approximation algorithm A for SGP such that  $\rho_A(n) < \frac{8569}{8568} \cong 1,000116\dots$  (unless P=NP)

# SEQUITUR

- + The algorithm was introduced by Craig Nevill-Manning and Ian H. Witten in “Identifying hierarchical structure in sequences: A linear-time algorithm,” *J. Artificial Intell.*, vol. 7, pp. 67–82, 1997.
- + It has been improved by the algorithm SEQUENTIAL introduced in E. H. Yang and J. C. Kieffer, “Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform—Part one: Without context models,” *IEEE Trans. Inf. Theory*, vol. 46, no. 3, pp. 755–777, May 2000.

# How does SEQUITUR work?

- + We start with the empty grammar with rule  $S \rightarrow \epsilon$
- + We scan the input  $w$  once from left to right and at each step the two constraints DIGRAM UNIQUENESS and RULE UTILITY must be respected:
  - + DIGRAM UNIQUENESS: no pair of adjacent symbols appears more than once in the grammar. When SEQUITUR observes a new symbol  $Y$ , it adds it to rule  $S$ . The last two symbols of rule  $S$  (the new symbol  $Y$  and its predecessor  $X$ ) form a new digram. If this digram  $XY$  appears elsewhere in the grammar, the first constraint has been violated. To restore it, a new rule is formed with the digram  $XY$  on the right-hand side, with a new non-terminal symbol  $A$  on the left, i.e.  $A \rightarrow XY$ . The two original digrams are replaced by  $A$ .
  - + RULE UTILITY: all non-terminal symbols must be used more than once in the right-hand parts of all productions of the grammar, i.e. if a symbol appears only once, it must be removed from the grammar and its appearance replaced with the symbols from which it was created.

# Example

+ Let  $w=abcdbcabcd$

| symbol number | the string so far | resulting grammar       | remarks                    |
|---------------|-------------------|-------------------------|----------------------------|
| 1             | a                 | $S \rightarrow a$       |                            |
| 2             | ab                | $S \rightarrow ab$      |                            |
| 3             | abc               | $S \rightarrow abc$     |                            |
| 4             | abcd              | $S \rightarrow abcd$    |                            |
| 5             | abcdb             | $S \rightarrow abcdb$   |                            |
| 6             | abcdbc            | $S \rightarrow abcdcb$  | bc appears twice           |
|               |                   | $S \rightarrow aAdA$    | enforce digram uniqueness  |
|               |                   | $A \rightarrow bc$      |                            |
| 7             | abcdcba           | $S \rightarrow aAdAa$   |                            |
|               |                   | $A \rightarrow bc$      |                            |
| 8             | abcdbcab          | $S \rightarrow aAdAab$  |                            |
|               |                   | $A \rightarrow bc$      |                            |
| 9             | abcdbcabc         | $S \rightarrow aAdAabc$ | bc appears twice           |
|               |                   | $A \rightarrow bc$      |                            |
|               |                   | $S \rightarrow aAdAaA$  | enforce digram uniqueness. |
|               |                   | $A \rightarrow bc$      | aA appears twice           |
|               |                   | $S \rightarrow BdAB$    | enforce digram uniqueness  |
|               |                   | $A \rightarrow bc$      |                            |
|               |                   | $B \rightarrow aA$      |                            |
| 10            | abcdbcabcd        | $S \rightarrow BdABd$   | Bd appears twice           |
|               |                   | $A \rightarrow bc$      |                            |
|               |                   | $B \rightarrow aA$      |                            |
|               |                   | $S \rightarrow CAC$     | enforce digram uniqueness. |
|               |                   | $A \rightarrow bc$      | B is only used once        |
|               |                   | $B \rightarrow aA$      |                            |
|               |                   | $C \rightarrow Bd$      |                            |
|               |                   | $S \rightarrow CAC$     | enforce rule utility       |
|               |                   | $A \rightarrow bc$      |                            |
|               |                   | $C \rightarrow aAd$     |                            |

# Performance of SEQUITUR

- + Theorem: SEQUITUR is linear in space and time  
(An index is constructed with the digrams of the text)
- + Theorem: The approximation ration of SEQUITUR is
$$O\left(\left(\frac{n}{\log n}\right)^{\frac{3}{4}}\right)$$

# Exercises

- + Apply SEQUITUR to  $f_6$
- + Apply SEQUITUR to the string: abcabcdefabcdef
- + Apply SEQUITUR to the string: abababcdabcde
- + Implement SEQUITUR

# RE-PAIR (Recursive Pairing)

- + It is an algorithm introduced in N. J. Larsson and A. Moffat, “Offline dictionary-based compression,” *Proc. IEEE*, vol. 88, no. 11, pp. 1722–1732, Nov. 2000.
- + It has linear time and space complexity, but with a large multiplicative constant.
- + It has been recently studied in T. Gagie, T. I, G. Manzini, G. Navarro, H. Sakamoto, Y. Takabatake “Rpair: Rescaling RePair with Rsync” SPIRE 2019: 35-44, in which Re-Pair can be applied on very large input. Experiments show that a very repetitive text of about 50GB can be compressed in less than an hour, using less than 650MB RAM with very competitive compression ratios.

# How does Re-Pair work?

1. Identify symbols X and Y such that XY is the most frequent pair of adjacent symbols (without overlap). If no pair appears more than once, stop.
2. Introduce a new symbol A and replace all occurrences of XY with A.
3. Repeat from Step 1.

# Example

+ w=aaabcaabaaabcabdabd

| aa | ab | bc | ca | ba | bd | da |
|----|----|----|----|----|----|----|
| 3  | 5  | 2  | 2  | 1  | 2  | 1  |

+ w=aaAcaAaaAcAdAd

| aa | aA | Ac | ca | Aa | Ad | dA |
|----|----|----|----|----|----|----|
| 2  | 3  | 2  | 1  | 1  | 2  | 1  |

+ w=aBcBaBcAdAd

| aB | Bc | cB | Ba | aB | cA | Ad | dA |
|----|----|----|----|----|----|----|----|
| 2  | 2  | 1  | 1  | 1  | 1  | 2  | 1  |

+ w=aBcBaBcCC

+ w=aDBaDCC

+ w=EBaECC is the compressed string and {A->ab, B->aA, C->Ad, D->Bc, E->aD} is the dictionary.

# Approximation ratio

- + Theorem: The approximation ratio of RE-PAIR is  $O\left(\left(\frac{n}{\log n}\right)^{\frac{2}{3}}\right)$
- + Exercise: Apply Re-Pair to  $f_6$

# How to obtain a CNF by RE-PAIR

- + Initial step: All terminal symbols in the input string are replaced with non-terminal symbols. This creates unary productions.
- + Replacement step: The algorithm picks an arbitrary bigram which has the most nonoverlapping occurrences in the string, and then replaces all possible occurrences of the bigram with a new non-terminal symbol.
- + The algorithm repeats the same process recursively for the string obtained after the replacement of the bigrams, until no bigrams have two or more non-overlapping occurrences in the string. It is clear that the productions created in the replacement stage are all binary
- + Final step: Trivial binary productions are created from the sequence of non-terminal symbols which are obtained after the last replacement. This ensures that the resulting grammar is in the Chomsky normal form. We remark that when distinct bigrams have the most non-overlapping occurrences in the string, then the choice of the bigrams to replace depends on each implementation of RePair.

# EXERCISE

- + Apply previous version of RE-PAIR to  $f_6$
- + It has been proved that RePair grammars are the smallest grammars for Fibonacci words [Mieno, Inenaga, Horiyama 2022]

# LZ78

- + LZ78 can be seen as a generator of grammars
- + Example:  $w=aabbababaabb$   
 $LZ78(w)=(0,a)(1,b)(0,b)(2,a)(3,a)(2,b)$
- + The pair  $(o,c)$  is encoded by  $T \rightarrow c$ , the pair  $(i,c)$  is encoded by  $T \rightarrow Uc$  where  $U$  is the non-terminal symbol associated to the  $i$ -th pair.

$$\begin{array}{l} S \rightarrow X_1 X_2 X_3 X_4 X_5 X_6 \\ X_1 \rightarrow a \quad X_3 \rightarrow b \quad X_5 \rightarrow X_3 a \\ X_2 \rightarrow X_1 b \quad X_4 \rightarrow X_2 a \quad X_6 \rightarrow X_2 b \end{array}$$

# Approximation ratio

- + Theorem: The approximation ratio of LZ78 is  $O\left(\left(\frac{n}{\log n}\right)^{\frac{2}{3}}\right)$
- + Exercise: Construct the grammar LZ78 for  $f_6$

# LZ77

- + Let us consider the following variant of LZ77.

| Index:  | 1 | 2 | 3      | 4      | 5 | 6 | 7 | 8      | 9 | 10 | 11 | 12 | 13 |
|---------|---|---|--------|--------|---|---|---|--------|---|----|----|----|----|
| LZ77:   | a | b | (1, 2) | (2, 3) |   |   | c | (1, 5) |   |    |    |    |    |
| String: | a | b | a      | b      | b | a | b | c      | a | b  | a  | b  | b  |

- + Let us consider  $f_6 = a\ b\ a\ aba\ baaba\ ab$ .  
 $LZ77(f_6) = a\ b\ (1, 1)\ (1, 3)\ (2, 5)\ (1, 2)$

- + It has been proved (Rytter 2003) that the number of phrases of LZ77 applied to the string  $x$  is a lower bound for the size of a generic grammar for  $x$ .

# Lab exercises

1. Apply the Sequitur algorithm to the following input sequence: "abacdabeabcdfabcd".
2. Apply the RE-PAIR algorithm to the following input sequence: "aaabbcccccdddaaabbb".
3. Compare the compression results of Sequitur and RE-PAIR algorithms for the input sequence: "abcabcbcabcabc".



# String Attractors

Wednesday May 17th, 2023

# Motivations

- + In many of the applications where data grows rapidly (genomic data, versioned documents, social networks, ...), a high degree of repetitiveness in the data is observed, meaning that most of the content of each element is the same as the content of other elements.
- + Compression algorithms alone may not be sufficient. Compressed data structures are needed to perform operations such as searching or data analysis.

# Motivations

- + One of the goals of current research is to find a model that captures or measures the degree of regularity in a text.
- + Entropies, including empirical order-0 or order-k entropies, are not always able to represent the lower bound of compression for any text. For example, it can be shown that if a text contains many repetitions, the empirical order-k entropy remains approximately the same.
- + This particular weakness has generated a lot of interest in algorithms that directly exploit the repetitiveness of the text to surpass the lower bound of entropy on highly repetitive texts. Both dictionary-based compressors (including grammar-based compressors) and Burrows-Wheeler Transform (BWT)-based compressors fall within this context.
- + At this point, it is natural to wonder if there exists a principle that can unify the various compression strategies to better establish their effectiveness or relate various measures of repetitiveness, such as the number of BWT runs, the number of LZ phrases, the smallest grammar size, etc.
- + Is there a complexity measure that allows for an effective evaluation, even on repetitive texts, of the performance of compression algorithms and compressed indexing structures?

# String attractors

- + The notion of string attractor has been introduced in the paper titled "At the Roots of Dictionary Compression: String Attractors" by D. Kempa and N. Prezza appeared among the accepted papers at the 50th Annual ACM SIGACT Symposium on the Theory of Computing (STOC 2018). This conference is considered one of the most prestigious in the field of theoretical computer science.
- + Based on the observation that the repetitiveness of a string can be defined in terms of the cardinality of the set of distinct substrings, the researchers introduced a very simple combinatorial object called a "string attractor," which captures the complexity of this set.

# String attractor

- + A string attractor is a set of positions within a string such that all distinct substrings have an occurrence that crosses at least one position in the string attractor.
- + Despite the simplicity of this definition, it has been shown that many compressors, particularly those based on dictionaries and the Burrows-Wheeler Transform (BWT), can be interpreted as algorithms that approximate the smallest string attractor of a given string.
- + This discovery opens the possibility of defining new measures of repetitiveness for strings.

# String attractor: formal definition

- + A string attractor for a string  $T$  over an alphabet  $\Sigma$  of size  $\sigma$  is a set of  $\gamma$  positions  $\Gamma = \{j_1, \dots, j_\gamma\}$  such that each substring  $T[i..j]$  has an occurrence  $T[i'..j'] = T[i..j]$  with  $j_k \in [i', j']$ , for some  $j_k \in \Gamma$
- + The set  $\{1, 2, \dots, n\}$  is trivially a string attractor. We are interested to the string attractor, denoted by  $\Gamma^*$ , of minimal size  $\gamma^*$
- + Each string attractor satisfies  $|\Gamma| \geq \sigma$ .
- + Example: let  $T = CDAB\underline{CCDAB}\underline{CCA}$ 
  - +  $\Gamma^* = \{4, 7, 11, 12\}$
- + To verify that  $\Gamma$  is a string attractor, it is sufficient to check that every substring located between positions in  $\Gamma$  has an occurrence that intersects a position of the attractor. There is no need to perform this verification for other strings.

# Property

- + PROPOSITION: Let  $\Gamma$  be a string attractor of size  $\gamma$  for a string  $S$ . Then,  $S$  contains at most  $\gamma k$  distinct  $k$ -mers (factors of length  $k$ ) for every  $1 \leq k \leq |S|$ .
- + Proof: By definition, every distinct  $k$ -mer that appears in the string has an occurrence that intersects a position of the attractor. Therefore, in order to count the number of distinct  $k$ -mers, we can focus our attention on the regions of size  $2k - 1$  that overlap with the positions of the attractor. The upper bound of  $\gamma k$  follows easily.

# Exercises

+ Finding a string attractor for the following strings:

1. 0100101001001
2. 0000001111111
3. 0110100110010110
4. ababababbababa
5. aaababbb
6. abacabadabacaba
7. abacabadabacabaеабакадабакаба

# Exercise

- + Let us consider the Thue-Morse words generated by using the rule (also called morphism)  $0 \rightarrow 01 \quad 1 \rightarrow 10$  (starting from 0)

0. 0

1. 01

2. 0110

3. 01101001

4. 0110100110010110

5. 01101001100101101001011001101001

...

Verify if  $\{3, 5, 6\}$ ,  $\{3, 6, 9, 12\}$  or  $\{3; 6; 12; 17; 24\}$  are string attractors for some of the Thue-Morse words. What about  $\{2, 6, 8, 12\}$ ? And  $\{8, 12, 16, 24\}$ ?

# NP-complete problem

- + Let us consider the problem: Given a string  $T$  construct a string attractor of minimal size.
- + It is a NP-complete problem.
- + It is possible to find approximations in polynomial time.

# Relationship with compressors

- + THEOREM: Let  $z$  be the number of factors in the Lempel-Ziv 77 factorization of a string  $T$ . Then,  $T$  has a string attractor of size  $z$ .

# Proof

- + Let us consider the following variant of LZ77.

| Index:  | 1 | 2 | 3     | 4     | 5 | 6 | 7 | 8 | 9     | 10 | 11 | 12 | 13 |
|---------|---|---|-------|-------|---|---|---|---|-------|----|----|----|----|
| LZ77:   | a | b | (1,2) | (2,3) |   |   | c |   | (1,5) |    |    |    |    |
| String: | a | b | a     | b     | b | a | b | c | a     | b  | a  | b  | b  |

- + EXAMPLE: Let us consider  $f_6 = a\ b\ a\ aba\ baaba\ ab$ .  
 $LZ77(f_6) = (0, a) (0, b) (1, 1) (1, 3) (2, 5) (1, 2)$
- + If we include in  $\Gamma_{LZ77}$  all the positions at the end of a phrase, it is known that every substring of the text has an occurrence that intersects a phrase boundary. Therefore, we can conclude that  $\Gamma_{LZ77}$  is a valid attractor for  $T$ .

# Relationship with compressors

- + THEOREM: Let  $r$  be the number of runs of equal letters in the Burrows-Wheeler Transform of  $T$ . Then,  $T$  has an attractor of size  $r$ .
- + Proof idea: We define  $\Gamma_{\text{BWT}}(T)$  the set of positions at the end of the runs in the BWT (Burrows-Wheeler Transform) of  $T$ .

# Relationship with compressors

- + THEOREM: Let  $G = \{X_i \rightarrow a_i, i=1, \dots, g'\} \cup \{X_i \rightarrow A_i B_i, i=1, \dots, g''\} \cup \{Y_i \rightarrow Z_i^{l_i}, l_i \geq 2, i=1, \dots, g'''\} \cup \{W_i \rightarrow K_i[l_i, r_i], i=1, \dots, g''''\}$  be a context-free grammar (also called a collage system) of size  $g = g' + g'' + g''' + g''''$  that generates a word  $w$ . Then,  $w$  has an attractor of size at most  $g$ .
- + THEOREM: Let  $\gamma^*$  be the size of the smallest string attractor for the string  $T \in \Sigma^n$ , and  $H_k$  denotes the empirical entropy of order  $k$  of  $T$ . Then,  $\gamma^* \log n \leq nH_k + o(n \log \sigma)$  for  $k \in o(\log_\sigma n)$ .

# From string attractors to compressors

- + THEOREM: Given a string  $T \in \Sigma^n$  and an attractor  $\Gamma$  of size  $\gamma$  for  $T$ , we can construct a context-free grammar (also called a Straight Line Program) for  $T$  of size  $O(\gamma \log^2(n/\gamma))$ .
- + THEOREM: Let  $g^*$  be the size of the smallest context-free grammar that generates  $T$ ,  $z$  be the size of the LZ77 parsing,  $\gamma^*$  be the size of the smallest attractor, and  $r$  be the number of runs of equal letters in the BWT of  $T$ . Then:

$$g^*, z \in O\left(\gamma^* \log^2\left(\frac{n}{\gamma^*}\right)\right)$$

$$g^*, z \in O\left(r \log^2\left(\frac{n}{r}\right)\right).$$

# Some recent results on LZ77 e BWT

- +  $z = O(r \log n)$ : According to the work by Gagie, Navarro, and Prezza in their paper "On the approximation ratio of Lempel-Ziv parsing" (2018), it has been proved that the number of LZ77 phrases, denoted as  $z$ , is upper-bounded by  $O(r \log n)$ , where  $n$  is the length of the text and  $r$  is the number of runs in the BWT.
- +  $r = O(z \log^2 n)$ : In the paper "Resolution of the Burrows-Wheeler Transform Conjecture" by Kempa and Kociumaka in April 2020, it is stated that the number of runs in the BWT, denoted as  $r$ , is upper-bounded by  $O(z \log^2 n)$ , where  $n$  is the length of the text and  $z$  is the number of LZ77 phrases.
- + Additionally, Kempa and Kociumaka propose an algorithm that converts LZ77 parsing into run-length compressed BWT in  $O(z \log^7 n)$  time complexity.

# Some recent results

- +  $\gamma^*$  (the size of the smallest string attractor) serves as a lower bound for almost all other measures of repetitiveness. However, it is not known whether it is achievable, i.e., whether we can represent  $S$  in  $O(\gamma^*)$  space.
- + On the other hand, Kempa and Prezza (2018) show that space  $O(\gamma \log(n/\gamma))$  is sufficient not only to encode  $S$  but also to provide logarithmic-time access to any  $S[i]$ .
- + Christiansen et al. (2020) also proved how to support indexed searches within  $O(\gamma \log(n/\gamma))$  space.
- + More recently, it has been proven that  $g^* = O(\gamma \log(n/\gamma))$  by Kociumaka (2021).

# Combinatorial properties

- + A string  $T[1..n]$  is considered repetitive when the cardinality of the set  $SUB_T = \{T[i..j] \mid 1 \leq i \leq j \leq |T|\}$  of its distinct substrings is much smaller than the maximum number of distinct substrings that could appear in a string of the same length over the same alphabet.
- + Note that  $T$  can be seen as a compact representation of  $SUB_T$ . This observation suggests a simple way to capture the degree of repetitiveness of  $T$ , namely the cardinality of  $SUB_T$ . Then, the cardinality of  $SUB_T$  can therefore be regarded as a measure of the complexity or repetitiveness of  $T$ .
- + Proposition: Given the string  $T$ ,  
$$|SUB_T| \leq \sum_{k=1}^n \min\{\sigma^k, n - k + 1, \gamma^* k\},$$
where  $\gamma^*$  is the size of the smallest string attractor.

# Another measure

- +  $\delta(S) = \max\{\text{SUB}(k)/k, 1 \leq k \leq n\}$ , where  $\text{SUB}(k)$  is the number of distinct factors of length  $k$ .
- + It is not difficult to prove that  $\delta(S) \leq \gamma(S)$  for every string  $S$ . As we have seen before, since every substring of length  $k$  in  $S$  has a copy that includes some of its  $\gamma$  attractors, there can be at most  $k\gamma$  distinct substrings, i.e.,  $\text{SUB}(k) \leq k\gamma$  for all  $k$ .

# Combinatorial properties

- + We denote by  $\ell_{\max}$  the length of the longest repeated substring. It is also indicative of the repetitiveness of the string. It can be shown that a smaller minimum attractor implies a larger  $\ell_{\max}$
- + PROPOSIZIONE. Let  $S$  be a string, let  $\gamma^*$  be the size of the smallest string attractor and let  $\ell_{\max}$  be the length of the smallest repeated substring. Therefore,  $\ell_{\max} \geq \frac{n-\gamma^*}{\gamma^* + 1}$  and, equivalently,  $\gamma^* \geq \frac{n-\ell_{\max}}{\ell_{\max} + 1}$ .
- + Proof: Consider the maximum distance  $j-i$  between any two positions in the smallest string attractor for  $S$ . Then, the substrings  $S[i+1\dots j-1]$  of length  $j-i-1$  between these two positions are not covered by any element of the attractor. This means that  $S[i+1\dots j-1]$  must have another occurrence that crosses some other element of the attractor, i.e., it repeats. The smallest maximum distance among the  $\gamma^*$  elements of the attractor occurs when the elements are equidistant. The claim easily follows after observing that the repeated substring should not cross any element of the attractor (there are  $n-\gamma^*$  remaining positions) and the  $\gamma^*$  elements divide the text into  $\gamma^*+1$  factors.

# Combinatorial properties

- + PROPOSITION: Let  $w$  be a string and  $w^r$  its reverse string.  
Then  $\gamma^*(w) = \gamma^*(w^r)$ .
- + PROPOSITION: Let  $u$  and  $v$  be two strings. Then  $\gamma^*(uv) \leq \gamma^*(u) + \gamma^*(v) + 1$ .
- + Although  $\gamma^*$  is sensitive to the concatenation operation, the following proposition shows that there exist words  $w=uv$  such that  $\gamma^*(u) > \gamma^*(w)$ .
- + PROPOSITION: The measure  $\gamma^*$  is not monotone. Let us consider, for instance, the strings  $abbba^nab$  and  $abbba^nabb$ .
- + PROPOSITION: Let  $w$  be a string over the alphabet  $\Sigma$ . Then,  
$$\gamma^*(w^n) \leq \gamma^*(w) + 1$$

# Lab exercises

- + Portfolio: Write a program in a programming language of your choice that, given a string  $T$  and a set  $\Gamma$  of positions, check if  $\Gamma$  is a string attractor for  $T$ .
- + Moreover, find a string attractor for a given text  $T$ , By using the partition LZ77 and the BWT.
- + Let us consider the finite Sturmian words. One can prove that finite Sturmian words have as smallest string attractor two consecutive positions. Verify if there exist other binary strings with the same length having the same property.
- + Given the set of binary strings of length  $n$ , find how many and which ones have a minimum attractor of size 2

# Finite Sturmian words

- + How to generate: Let  $d_1, d_2, \dots, d_n, \dots$  be sequence of natural numbers, with  $d_1 \geq 0$  and  $d_i > 0$  for  $i = 2, \dots, n, \dots$   
..
- + Consider the following sequence of words  $\{s_n\}_{n \geq 0}$ :  $s_0 = b$ ,  $s_1 = a$ , and  $s_{n+1} = s_n^{d_n} s_{n-1}$  per  $n > 1$ .
- + Each word  $s_n$  is a finite Sturmian word. Fibonacci words are obtained by considering the sequence  $(1, 1, 1, 1, \dots)$

# Lab exercises

## + Alternatively:

Portfolio: Write a program in a programming language of your choice that, given a string  $T$ , construct the grammar produced by the algorithm RE-PAIR.

Implement also the construction of the grammar in the Chomsky normal form.

Compute also the size of the grammar in both cases.



# Compression benchmark

Friday, May 19, 2023

# Social Network and Compression



+ Facebook in 2015 published details on how it provides preview images with only 200 bytes each.

<https://code.fb.com/android/the-technology-behind-preview-photos/>

The technology behind preview photos

Aug 31, 2017  
Optimizing 360 photos at scale

Oct 16, 2018  
TIP Summit 2018: Advancing efforts to improve global connectivity

Mar 19, 2018  
PERFORMANCE @ SCALE

<https://code.fb.com/ml-applications/optimizing-360-photos-at-scale/>

# Netflix

- + In 2015, Netflix announced that it was about to start modifying its compression technology for its videos; the idea is to be able to modulate the type of algorithm on the type of video, depending on the complexity of the content itself.

A Medium Corporation [US] | <https://medium.com/netflix-techblog/per-title-encode-optimization-7e99442b62a2>

**M** THE NETFLIX TECH BLOG Sign in Get started

## Per-Title Encode Optimization

 Netflix Technology Blog Follow  
Dec 14, 2015 · 11 min read

We've spent years developing an approach, called per-title encoding, where we run analysis on an individual title to determine the optimal encoding recipe based on its complexity. Imagine having very involved action scenes that need more bits to encapsulate the information versus unchanging landscape scenes or animation that need less. This allows us to deliver the same or better experience while using less bandwidth, which will be particularly important in lower bandwidth countries and as we expand to places where video viewing often happens on mobile networks.

Never miss a story from **Netflix TechBlog**, when you sign up for Medium. Learn more [GET UPDATES](#)



# Snappy (2011)

The screenshot shows a web browser displaying the official GitHub Pages site for the Snappy project. The URL in the address bar is [google.github.io/snappy/](https://google.github.io/snappy/). The page has a blue header with the Snappy logo and tagline "A fast compressor/decompressor". On the right side, there's a GitHub logo button labeled "View project on GitHub". Below the header, there's a detailed description of the Snappy library, mentioning its speed and compression ratio compared to zlib. It also notes its widespread use at Google. A download section offers ".zip" and ".tar.gz" file formats. At the bottom, it states that the project is maintained by Google and was generated using GitHub Pages.

Non sicuro | google.github.io/snappy/

# Snappy

A fast compressor/decompressor

View project on GitHub

Snappy is a compression/decompression library. It does not aim for maximum compression, or compatibility with any other compression library; instead, it aims for very high speeds and reasonable compression. For instance, compared to the fastest mode of zlib, Snappy is an order of magnitude faster for most inputs, but the resulting compressed files are anywhere from 20% to 100% bigger. On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more.

Snappy is widely used inside Google, in everything from BigTable and MapReduce to our internal RPC systems. (Snappy has previously been referred to as "Zippy" in some presentations and the likes.)

The latest release at the time of writing is 1.1.3.

For more information, please see the [README](#). Benchmarks against a few other compression libraries (zlib, LZO, LZF, FastLZ, and QuickLZ) are included in the source code distribution. The source code also contains a [formal format specification](#), as well as a specification for a [framing format](#) useful for

Download .zip file

Download .tar.gz file

is maintained by [google](#).

This page was generated by [GitHub Pages](#) using the [Architect](#) theme by [Jason Long](#).

# LZ4 – Extremely fast (2011)

- Operating Systems :  [Linux](#) (\*).  [FreeBSD](#) (\*).  [Illumos](#) (\*)
- File Systems :  [OpenZFS](#) (\*).  [Hammer2](#) (\*).  [SquashFS](#) (\*).  [LessFS](#) (\*).  [LeoFS](#) (\*)  [Grub](#)
- Big Data :  [Hadoop](#) (\*).  [cassandra](#)  [Cassandra](#) (\*).  [Hbase](#) (\*).  [Rarelogic](#),  [Spark](#) (\*)  [Spark](#) (\*), [Hustle](#) (\*)
- Search Engine :  [Lucene](#) (\*).  [Apache Solr](#)  [Solr](#) (\*)
- Database :  [MySQL](#) (\*)  [Tokudb](#),  [Delphix](#) (\*).  [infiniSQL](#) (\*).  [RocksDB](#) (\*).  [OlegDB](#) (\*)
- Games :  [Battlefield 4](#) (\*).  [Guild Wars 2](#) (\*).  [Sine Mora](#) (\*).  [1000 Tiny Claws](#)
- Games :  [Panzer Tactics HD](#)

# Brotli: LZ77 and Huffman encoding

The screenshot shows a web browser displaying a blog post from the Google Open Source Blog. The URL in the address bar is <https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html>. The page header includes the Google Open Source logo and navigation links for PROJECTS, COMMUNITY, DOCS, and BLOG. The main content features a large title "Google Open Source Blog" and a subtitle "The latest news from Google on open source releases, major projects, events, and student outreach programs." Below the title is the main article: "Introducing Brotli: a new compression algorithm for the internet". The article is dated Tuesday, September 22, 2015. The text discusses the development of Brotli, its integration into various compression solutions, and its use in modern compression needs like web font compression. To the right of the article, there is a search bar labeled "Search blog..." and a sidebar titled "Popular Posts" with links to other blog articles.

Google Open Source

PROJECTS COMMUNITY DOCS BLOG

## Google Open Source Blog

The latest news from Google on open source releases, major projects, events, and student outreach programs.

### Introducing Brotli: a new compression algorithm for the internet

Tuesday, September 22, 2015

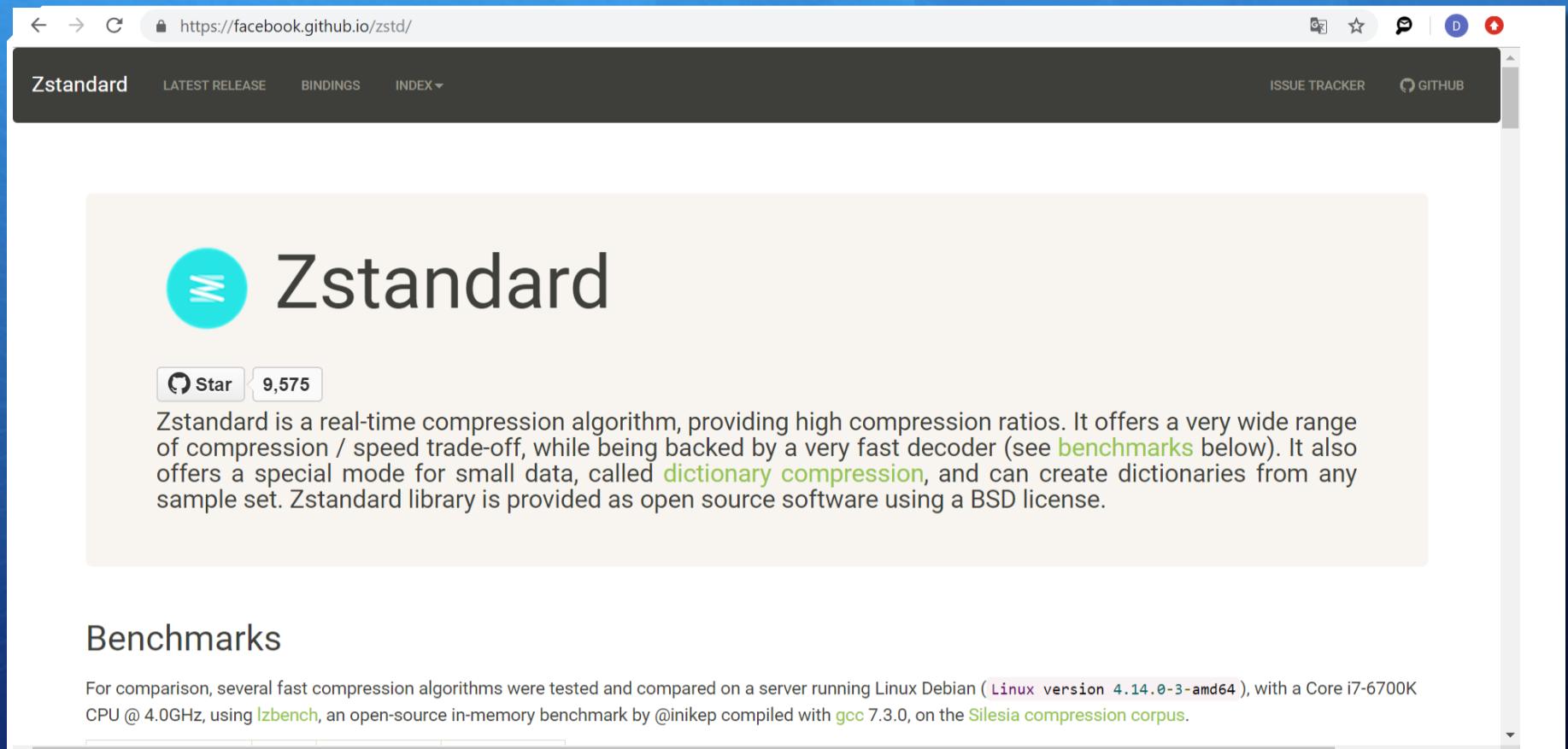
At Google, we think that internet users' time is valuable, and that they shouldn't have to wait long for a web page to load. Because fast is better than slow, two years ago we published the [Zopfli compression algorithm](#). This received such positive feedback in the industry that it has been integrated into many compression solutions, ranging from PNG optimizers to preprocessing web content. Based on its use and other modern compression needs, such as [web font compression](#), today we are excited to announce that we have developed and open sourced a new algorithm, the [Brotli compression algorithm](#).

Search blog...

Popular Posts

- Introducing Git protocol version 2
- Open sourcing ClusterFuzz
- Google Summer of Code: 15 years strong!
- Introducing Data Transfer

# Zstandard (2015)

A screenshot of a web browser displaying the official Zstandard GitHub repository at <https://facebook.github.io/zstd/>. The page has a dark header with navigation links for "LATEST RELEASE", "BINDINGS", "INDEX", "ISSUE TRACKER", and "GITHUB". The main content area features the Zstandard logo (a stylized 'Z' inside a circle) and the word "Zstandard". Below this, there's a "Star" button with the number "9,575". A descriptive paragraph explains that Zstandard is a real-time compression algorithm with high compression ratios and a fast decoder, supporting dictionary compression and provided under a BSD license. At the bottom, a section titled "Benchmarks" provides details about testing against other algorithms like LZ4 and LZ4HC.

## Benchmarks

For comparison, several fast compression algorithms were tested and compared on a server running Linux Debian ([Linux version 4.14.0-3-amd64](#)), with a Core i7-6700K CPU @ 4.0GHz, using [lzbench](#), an open-source in-memory benchmark by [@inikep](#) compiled with [gcc](#) 7.3.0, on the [Silesia compression corpus](#).

# LZFSE (2015)

The screenshot shows a web browser window with the URL <https://www.infoq.com/news/2016/07/apple-lzfse-lossless-opensource>. The page is titled "Apple Open-Sources its New Compression Algorithm LZFSE". The main content discusses the introduction of LZFSE by Apple, its performance, and its relationship to ANS entropy coding. Below the main content, there is a sidebar for related content.

DEVELOPMENT

## Apple Open-Sources its New Compression Algorithm LZFSE

LIKE DISCUSS PRINT BOOKMARKS

JUL 02, 2016 • 1 MIN READ

by Sergio De Simone

FOLLOW

Apple has open-sourced its new lossless compression algorithm, [LZFSE](#), introduced last year with iOS 9 and OS X 10.10. According to Apple, LZFSE provides the same compression gain as ZLib level 5 while being 2x–3x faster and with higher energy efficiency.

LZFSE is based on Lempel-Ziv and uses [Finite State Entropy coding](#), based on Jarek Duda's work on [Asymmetric Numeral Systems](#) (ANS) for entropy coding. Shortly, ANS aims to "end the trade-off between speed and rate" and can be used both for precise coding and very fast encoding, with support for data encryption. LZFSE is one of a [growing number](#) of compression libraries that use ANS in place of the more traditional [Huffman](#) and [arithmetic coding](#).

### RELATED CONTENT

**Swift 5 Now Officially Available**  
MAR 28, 2019

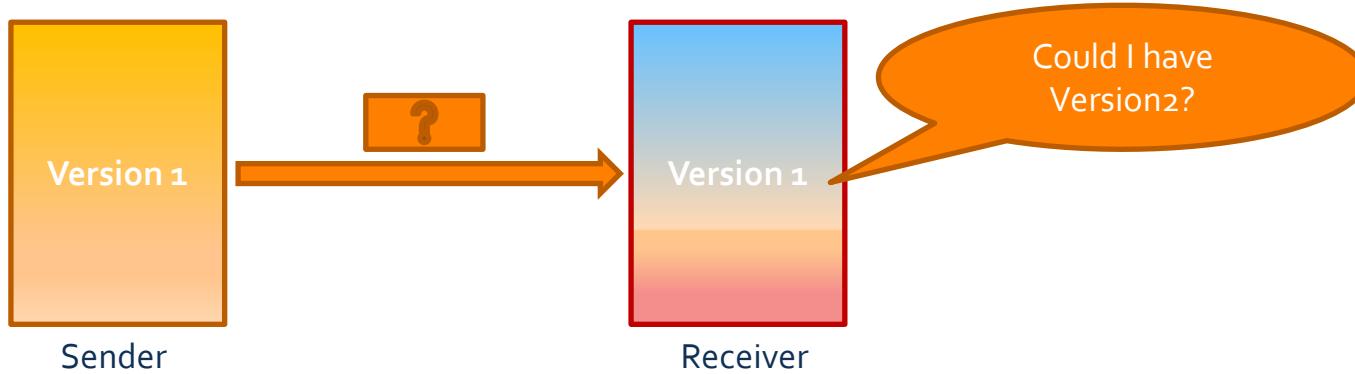
**F# 4.6 Released with Anonymous Records, Improved Performance**  
APR 08, 2019

**Vector Performance Monitoring Tool Adds eBPF, Unified Host-Container Metrics Support**  
MAR 23, 2019

**Google Launches "Season of Docs" Program to Improve Open Source Documentation**  
MAR 21, 2019

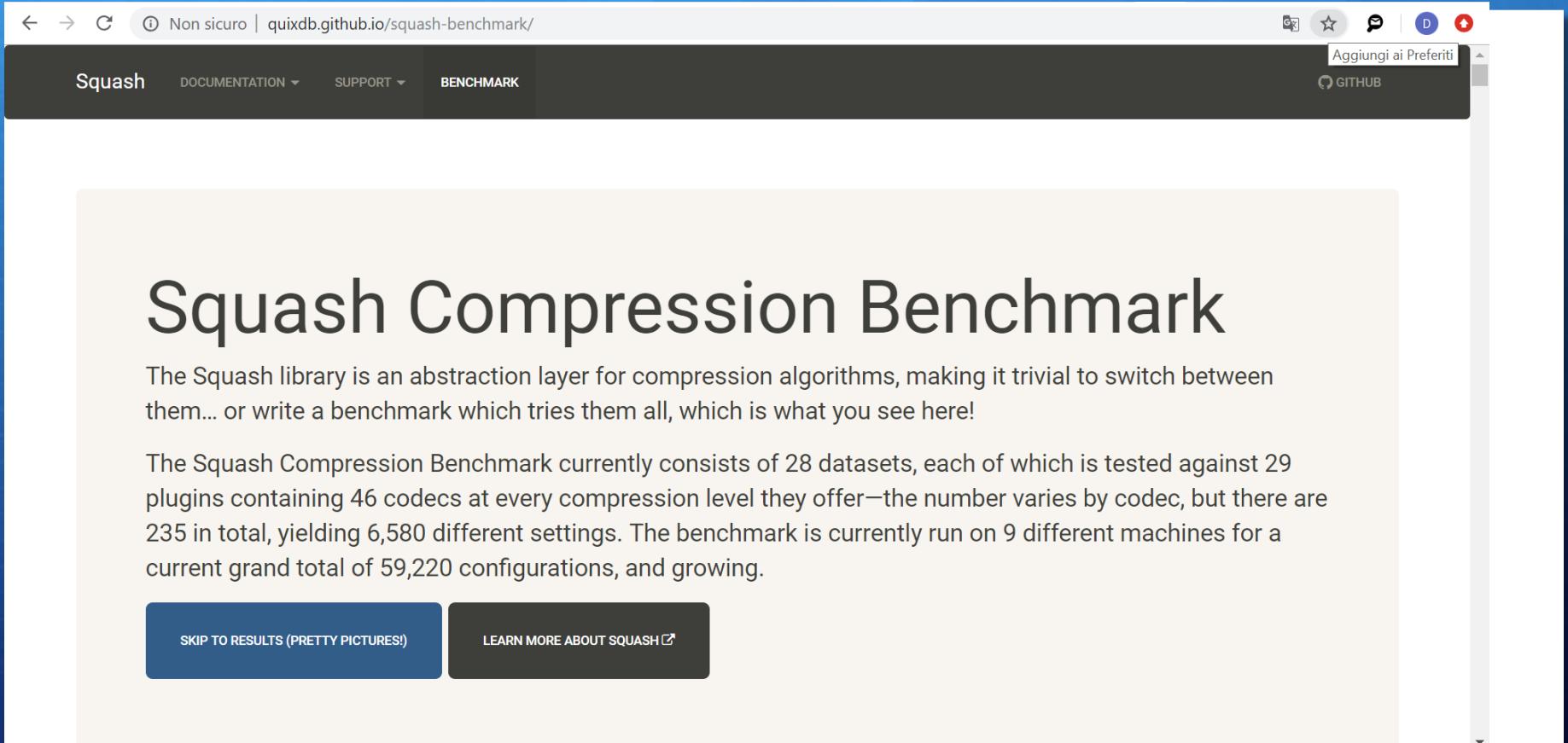
**ImageSharp: Funding an Open Source Project**  
MAR 18, 2019

# LZ77 scheme to update apps and softwares



- + The concatenation **Version 1Version2** is compressed, start to scan **Version2** (**Version 1** is used as a dictionary)
  
- + A similar approach to synchronize data

# Which compressor should we choose?

A screenshot of a web browser displaying the Squash Compression Benchmark page. The URL in the address bar is "Non sicuro | quixdb.github.io/squash-benchmark/". The page has a dark header with navigation links for "Squash", "DOCUMENTATION", "SUPPORT", and "BENCHMARK". On the right side of the header are icons for a star (Bookmark), a speech bubble (Feedback), a GitHub logo, and a red circular arrow (Refresh). A button labeled "Aggiungi ai Preferiti" (Add to Favorites) is also visible. The main content area features a large title "Squash Compression Benchmark" in a bold, sans-serif font. Below the title is a paragraph explaining the library's purpose: "The Squash library is an abstraction layer for compression algorithms, making it trivial to switch between them... or write a benchmark which tries them all, which is what you see here!". Another paragraph details the benchmark's scope: "The Squash Compression Benchmark currently consists of 28 datasets, each of which is tested against 29 plugins containing 46 codecs at every compression level they offer—the number varies by codec, but there are 235 in total, yielding 6,580 different settings. The benchmark is currently run on 9 different machines for a current grand total of 59,220 configurations, and growing." At the bottom of the page are two buttons: "SKIP TO RESULTS (PRETTY PICTURES!)" and "LEARN MORE ABOUT SQUASH".

## Squash Compression Benchmark

The Squash library is an abstraction layer for compression algorithms, making it trivial to switch between them... or write a benchmark which tries them all, which is what you see here!

The Squash Compression Benchmark currently consists of 28 datasets, each of which is tested against 29 plugins containing 46 codecs at every compression level they offer—the number varies by codec, but there are 235 in total, yielding 6,580 different settings. The benchmark is currently run on 9 different machines for a current grand total of 59,220 configurations, and growing.

[SKIP TO RESULTS \(PRETTY PICTURES!\)](#)

[LEARN MORE ABOUT SQUASH](#)

# Choose a dataset

A screenshot of a web browser window displaying a dataset selection interface. The URL in the address bar is `Non sicuro quixdb.github.io/squash-benchmark/`. The main content area has a heading "Choose a dataset" with a link icon. Below it is a note: "Different codecs can behave very differently with different data. Some are great at compressing text but horrible with binary data, some excel with more repetitive data like logs. Many have long initialization times but are fast once they get started, while others can compress/decompress small buffers almost instantly." Another note below states: "This benchmark is run against many standard datasets. Hopefully one of them is interesting for you, but if not don't worry—you can use Squash to easily run your own benchmark with whatever data you want. That said, if you think you have a somewhat common use case, please [let us know](#)—we may be interested in adding the data to this benchmark." A "Note" section indicates that the default dataset is selected randomly. A table lists seven datasets with columns for Name, Source, Description, and Size.

| Name           | Source                           | Description                                                          | Size       |
|----------------|----------------------------------|----------------------------------------------------------------------|------------|
| alice29.txt    | Canterbury Corpus                | English text                                                         | 148.52 KiB |
| asyoulik.txt   | Canterbury Corpus                | Shakespeare                                                          | 122.25 KiB |
| cp.html        | Canterbury Corpus                | HTML source                                                          | 24.03 KiB  |
| dickens        | Silesia Corpus                   | Collected works of Charles Dickens                                   | 9.72 MiB   |
| enwik8         | Large Text Compression Benchmark | The first $10^8$ bytes of the English Wikipedia dump on Mar. 3, 2006 | 95.37 MiB  |
| fields.c       | Canterbury Corpus                | C source                                                             | 10.89 KiB  |
| fireworks.jpeg | Snappy                           | A JPEG image                                                         | 120.21 KiB |

# Choose a machine

Non sicuro quixdb.github.io/squash-benchmark/     

## Choose a machine

If you think certain algorithms are always faster, you've got another thing coming! Different CPUs can behave very differently with the same data.

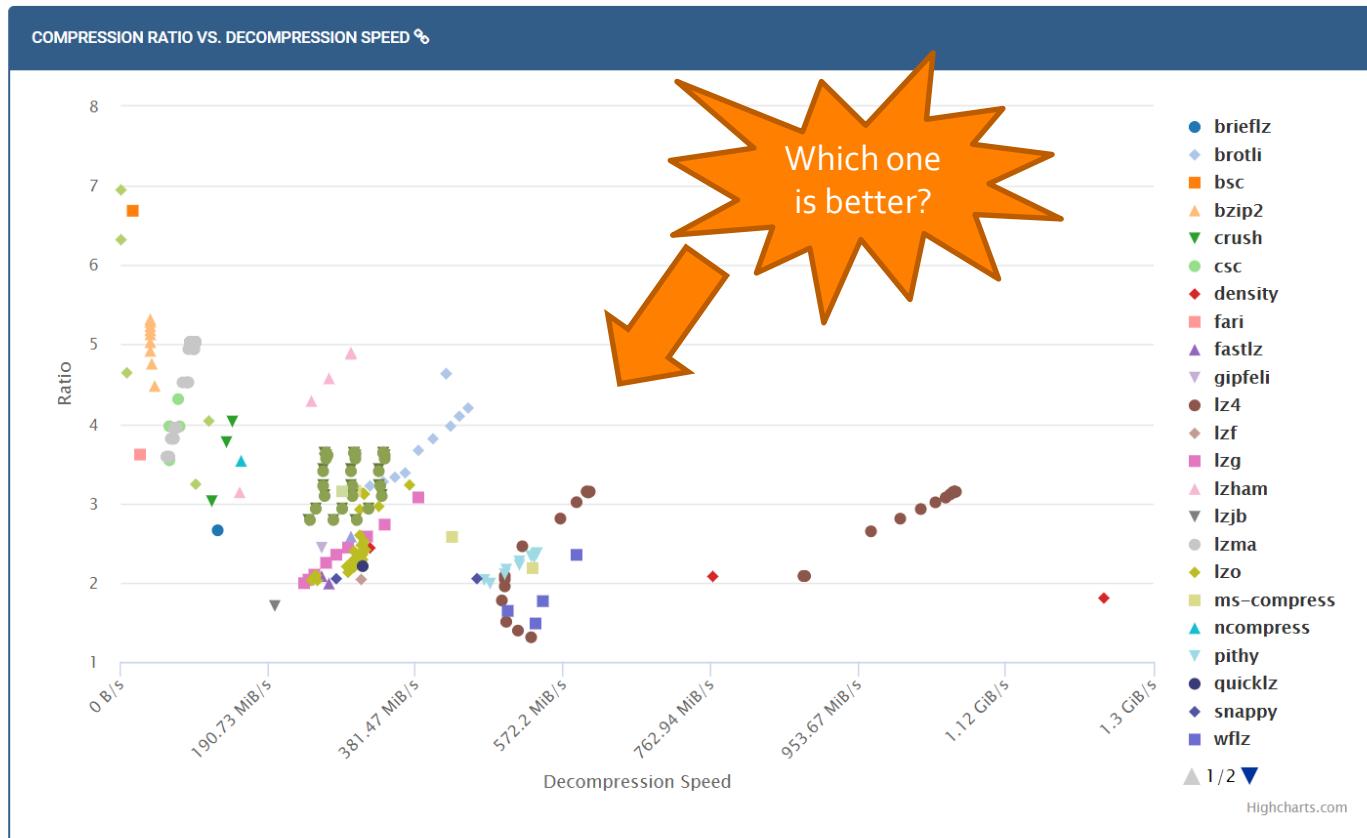
The Squash benchmark is currently run on many of the machines I have access to—this happens to be fairly recent Intel CPUs, and a mix of ARM SBCs. There is an entry in the [FAQ](#) with more details.

**Note**

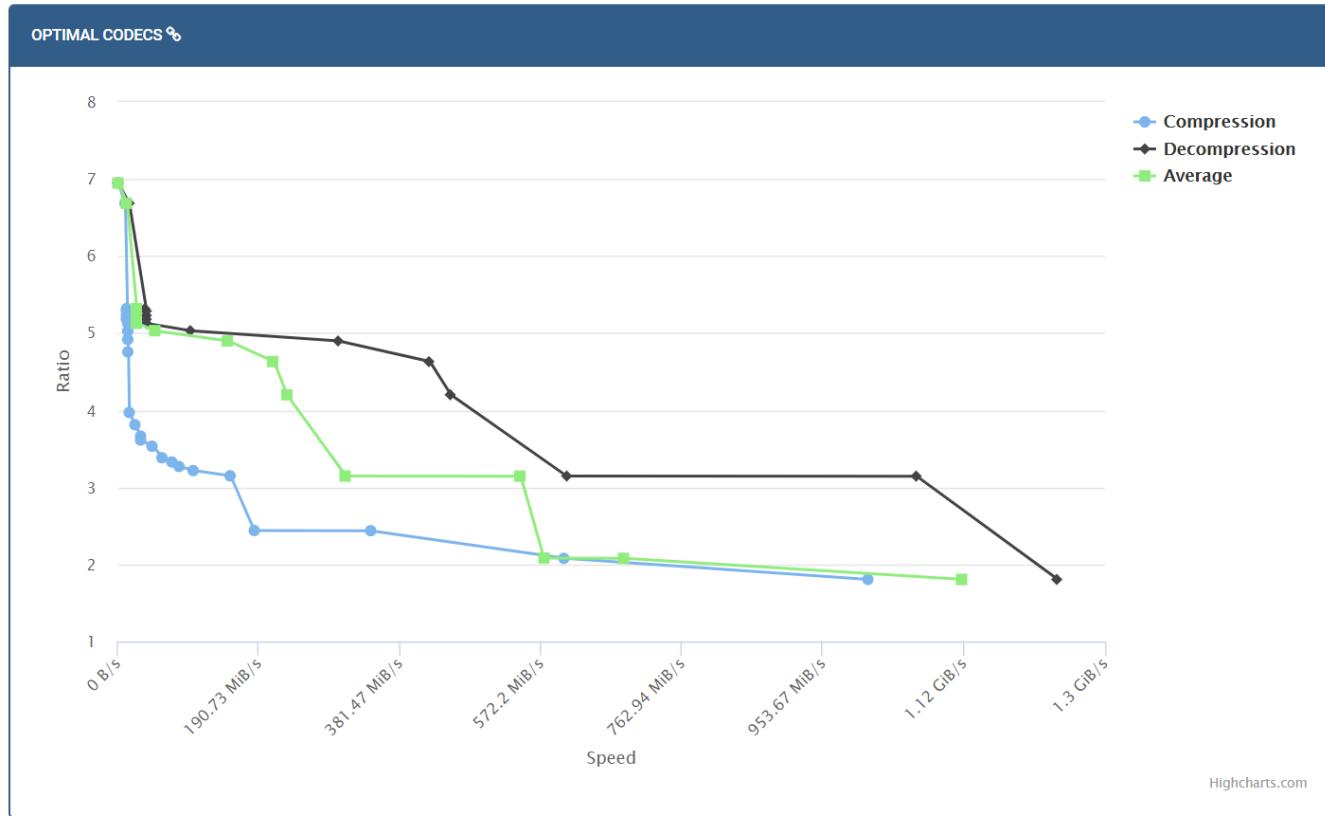
The default machine is selected randomly.

| Name           | Status | CPU/SoC                           | Architecture | Clock Speed | Memory  | Platform                     | Distro          | Kernel  | Compiler  | CSV                                                                                   |
|----------------|--------|-----------------------------------|--------------|-------------|---------|------------------------------|-----------------|---------|-----------|---------------------------------------------------------------------------------------|
| beagleboard-xm | ✓      | Texas Instruments DM3730          | armv7l       | 1 GHz       | 512 MiB | BeagleBoard-xM revision B    | Ubuntu 15.04    | 4.1.5   | gcc-4.9.2 |    |
| e-desktop      | ✓      | Intel® Core™ i3-2105              | x86_64       | 3.1 GHz     | 8 GiB   | Asus P8H61-H                 | Fedora 22       | 4.1.4   | gcc-5.1.1 |    |
| hoplite        | ✓      | Intel® Core™ i7-2630QM            | x86_64       | 2 GHz       | 6 GiB   | Toshiba Satellite A660-X     | Fedora 22       | 4.1.4   | gcc-5.1.1 |    |
| odroid-c1      | ✓      | Amlogic S805                      | armv7l       | 1.5 GHz     | 1 GiB   | ODROID-C1                    | Ubuntu 14.04.4  | 3.10.80 | gcc-4.9.2 |    |
| peltast        | ✓      | Intel® Xeon® Processor E3-1225 v3 | x86_64       | 3.2 GHz     | 20 GiB  | Lenovo ThinkServer TS140     | Fedora 22       | 4.1.6   | gcc-5.1.1 |    |
| phalanx        | ✓      | Intel® Atom™ D525                 | x86_64       | 1.8 GHz     | 4 GiB   | Asus AT5NM10T-I              | Fedora 22       | 4.1.4   | gcc-5.1.1 |    |
| raspberry-pi-2 | ✓      | Broadcom BCM2709                  | armv7l       | 900 MHz     | 1 GiB   | Raspberry Pi 2 Model B       | Raspbian Jessie | 4.1.6   | gcc-4.9.2 |    |
| s-desktop      | ✓      | Intel® Core™ i5-2400              | x86_64       | 3.1 GHz     | 4 GiB   | Asus P8Z68-V                 | Fedora 22       | 4.1.4   | gcc-5.1.1 |  |
| satellite-a205 | ✓      | Intel® Celeron® Processor 540     | x86_64       | 1.86 GHz    | 1 GiB   | Toshiba Satellite A205-S5805 | Fedora 21       | 4.1.6   | gcc-4.9.2 |  |

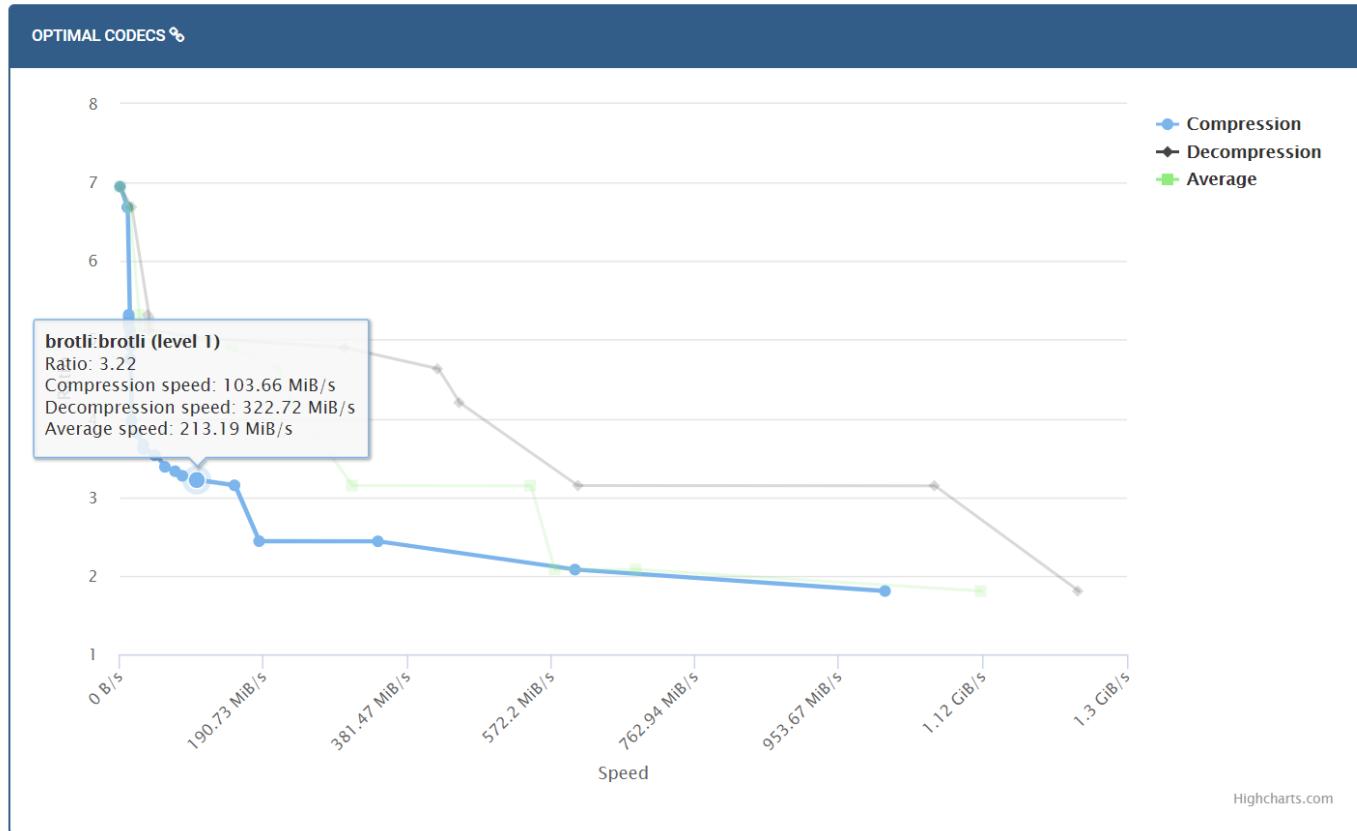
# Results



# Which algorithms are optimal?

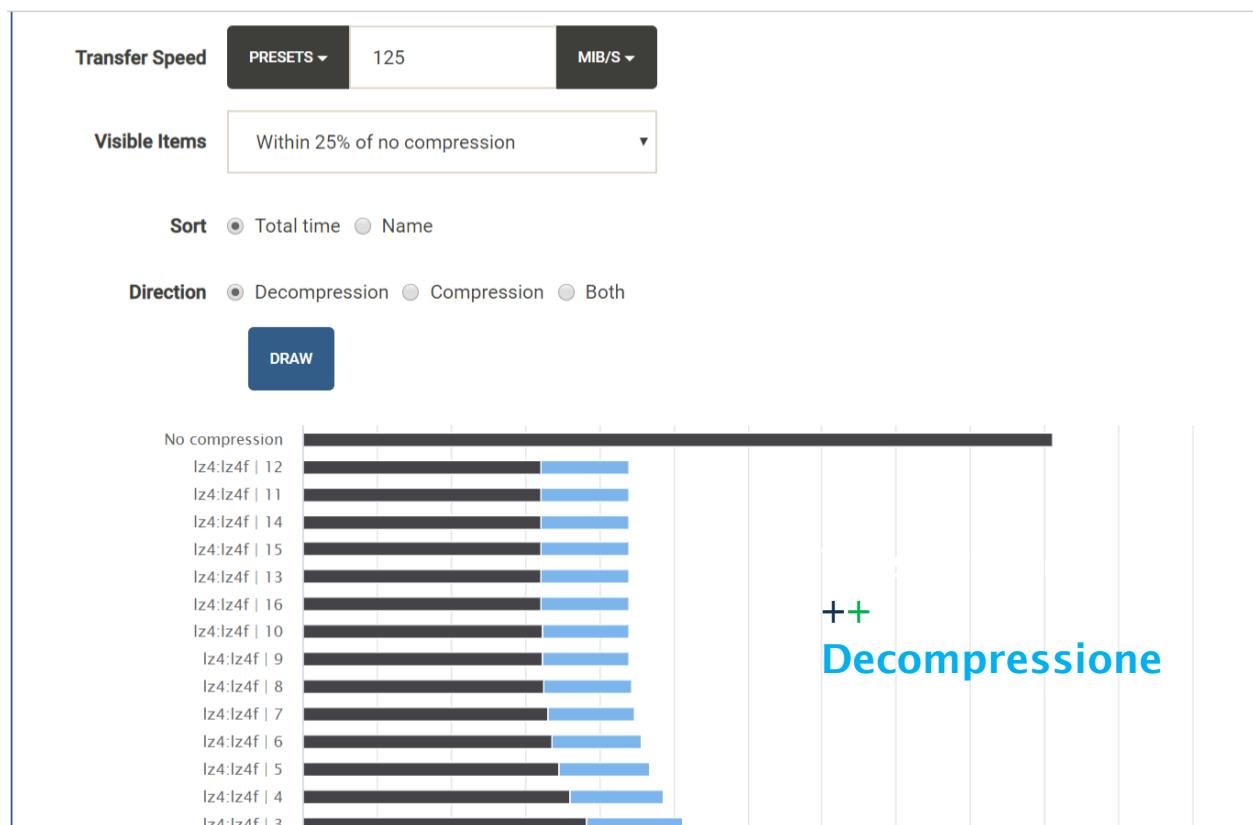


# Which algorithms are better?



# Sometimes it is better not to compress

- + If uploading an uncompressed file takes 1 second, it only makes sense to compress it if the decompression and transfer time is shorter!



# Lab Exercises: possible exam questions

- + Determine if the following code  $\{01, 011, 1101, 11011, 11101\}$  is UD?
- + Given an input text in which the characters in the alphabet have the following frequencies: A: 10 B: 15 C: 30 D: 20 E: 25, construct the Huffman code.
- + Given a sequence of integers  $\{10, 5, 8, 3, 12\}$ , which is the gamma encoding?
- + Given the text abbccddddd, which is its arithmetic coding?
- + Given the text ABCDEABCDEFGHABC, which is the output of the SEQUITUR algorithm?
- + Consider a set of source symbols  $S = \{A, B, C, D\}$ . Is it possible to construct a binary prefix code with corresponding codeword lengths  $L = \{2, 2, 3, 3\}$ ? Let us consider now a set of source symbols  $S = \{A, B, C, D, E\}$  with corresponding codeword lengths  $L = \{2, 2, 2, 3, 3\}$ .
- + Compute BWT(banana)
- + Compute LZ77 parsing of the string abcdeabcdeabcde
- + Compute LZ78 parsing of the string abcdeabcdeabcde

# Next meetings

- + May 22 and 29 at 10.00 for clarifications and exercises.
- + Exams (sessione estiva):
  - + 8 June 2023, 10:00 (9 June 2023, 10:00 a.m. for Erasmus students)
  - + 20 June 2023, 15:00
  - + 6 July 2023, 10:00