



UNIVERSITÀ DEGLI STUDI  
DI MODENA E REGGIO EMILIA

# Lecture notes for Multimedia Data Processing

## Data Compression

Last updated on 27/02/2023

# The Importance of Compression

- The amount of data we interact with is increasing dramatically.
- Communication skills do not have the same development.
- Multimedia data have given further impetus to this need:
  - The Holy Bible in text format occupies 4.41 MB (4,627,239 bytes)
  - A 1280x1024 color photo with 24 bits/pixel occupies 3.75 MB (3,932,160 bytes)
  - A 4-minutes song in CD format (44.1 KHz, 16 bit/sample, Stereo) occupies 40.38 MB (42.336.000 bytes)
  - A 1-minute video in PAL format (704x576 at 24 bit/pixel, 25 fps) occupies 1.7 GB (1,824,768,000 bytes)
- The technology has therefore followed this evolution also in the field of generic coding, but above all with technical specifications for audio and video.

# Terminology

- *Message*: some information concerning the specific problem or area under consideration (a song, a sound, an image or a video).
- *Symbols*: elementary parts that make up the message (bytes, letters of the alphabet, pixel brightness, audio sample value, etc.)
- You can describe the same message with a different set of symbols.
- *Encoding*: numeric representation used for symbols (for example one byte for each letter).
- The word "compression" is referred to all the operations required to obtain a message encoding with a size (possibly) reduced w.r.t to its original encoding.
- The *decoding* of a compressed message can take place in exact form (lossless) or approximate (lossy).

# Perfect compression does not exist!

- An algorithm capable of compressing everything does not exist and cannot exist.
- To be more precise, given all the possible messages that can be represented with  $n$  bits, it is not possible to represent them with  $n-1$  bits.
- Therefore, compressions that always work are not possible.
- This is not so trivial! As an example, the US patent 5,533,051 entitled "Methods for Data Compression" (existing) says that it can reduce any message by at least one bit.
- So how do we compress?
- We need to create a *model* of our system able to describe the probability with which the message is found, then encoding the less likely messages with more bits and the more likely ones with less bits.
- The *encoder* is the algorithm that select how to represent messages.

# Information Theory

- C. E. Shannon was one of the pioneers in this field. He formalized a measure capable of representing the amount of information or uncertainty of a set of possible symbols  $S$ :

$$H(S) = \sum_{x \in S} p(x) \log_2 \frac{1}{p(x)} = - \sum_{x \in S} p(x) \log_2 p(x)$$

- with  $p(x)$  the probability of a symbol  $x$ .
- This measure is characterized by multiple mathematical properties that won't be analyzed here.
- It basically says that most likely messages in a set contain less information.
  - This may seem strange, but if I tell you that there will be 30 degrees tomorrow and we are in winter, I am giving you more information w.r.t. a communication informing you that there will be 10 degrees tomorrow.
- Beyond philosophical aspects, this measure allowed Shannon to develop a whole mathematical model that has conditioned the history of communication.
- This measure limits the maximum transmission capacity of a communication system.

# Entropy and Compression

- Under the unlucky suggestion of Von Neumann and given its similarity with the entropy in statistical mechanics, the measure  $H(S)$  has been called entropy by Shannon.
- In general it is possible to say that the average encoding length  $C_S$  of a set of symbols  $S$  is given by:

$$\bar{L}(C_S) = \sum_{(x,w) \in C_S} p(x)L(w)$$


- where  $C_S$  is a set with all the pairs symbol-code and  $L(w)$  is the length in bits of the code  $w$ .
- The following relation is true:


$$H(S) \leq \bar{L}(C_S), \forall C_S$$


- This relationship limits the maximum compression obtainable with variable length representations of the symbols to be transmitted.
- The only way to overcome this limit is to define a new symbols system with less entropy, able of transmitting the same concept.


# Encoding of Numbers


- [illegible]


1 


2 


3 


4 


5 

6 

7 

8 

9 

10 

# Unary coding

- A direct translation of the previously described approach for coding numbers is given by substituting 1 to I and adding a 0 instead of a space at the end of each number, which would give: 1 = 10, 2 = 110, 3 = 1110, 4 = 11110, ...
- This is not very smart, because there is a 1 at the beginning of all numbers, which gives no information. So the *unary coding* is given by the following schema (any binary encoding has a dual form):

1	0	1
2	10	01
3	110	001
4	1110	0001
5	11110	00001
6	111110	000001
7	1111110	0000001
...	...	...



# Unary coding

- Is unary coding a *good* encoding? Is it an *optimal* encoding?

# Unary coding

- Is unary coding a *good* encoding? Is it an *optimal* encoding?
- As it is usually the case, the answer is “it depends”. On what?

# Unary coding

- Is unary coding a *good* encoding? Is it an *optimal* encoding?
- As it is usually the case, the answer is “it depends”. On what?
- It depends on *the probability distribution* of the symbols. Based on that, it can be optimal, good, average or horrible!
- For which distribution is unary coding optimal? Optimal means that there is no other encoding which has a lower  $\bar{L}(C_S)$ . This is guaranteed only if  $\bar{L}(C_S) = H(S)$ . Let's make both quantities explicit, taking into account that  $S = [1, +\infty)$ :

$$\sum_{i=1}^{+\infty} p_i L_i = \sum_{i=1}^{+\infty} p_i \log_2 \frac{1}{p_i}$$

- Moreover  $L_i = i$ , so:

$$\sum_{i=1}^{+\infty} p_i \cdot i = \sum_{i=1}^{+\infty} p_i \log_2 \frac{1}{p_i}$$

- that gives us:

# Unary coding

$$i = \log_2 \frac{1}{p_i} \rightarrow 2^i = \frac{1}{p_i} \rightarrow p_i = \frac{1}{2^i}$$

- Basically we are saying that unary coding is optimal if the symbols are distributed as negative powers of two, that is if half of the values are 1, a quarter is 2, an eighth is 3, a sixteenth is 4, and so on.
- On a side note, we are talking of numbers, but of course any bijection with those numbers can be used to map codes and symbols.
- So if we have letters in which half of the values are *f*, a quarter is *w*, an eighth is *n*, a sixteenth is *s*, ..., we can map unary codes to those symbols, obtaining an optimal encoding.
- A negative exponential with base 2 is pretty steep and goes to zero very quickly.
- Let's see an alternative universal code

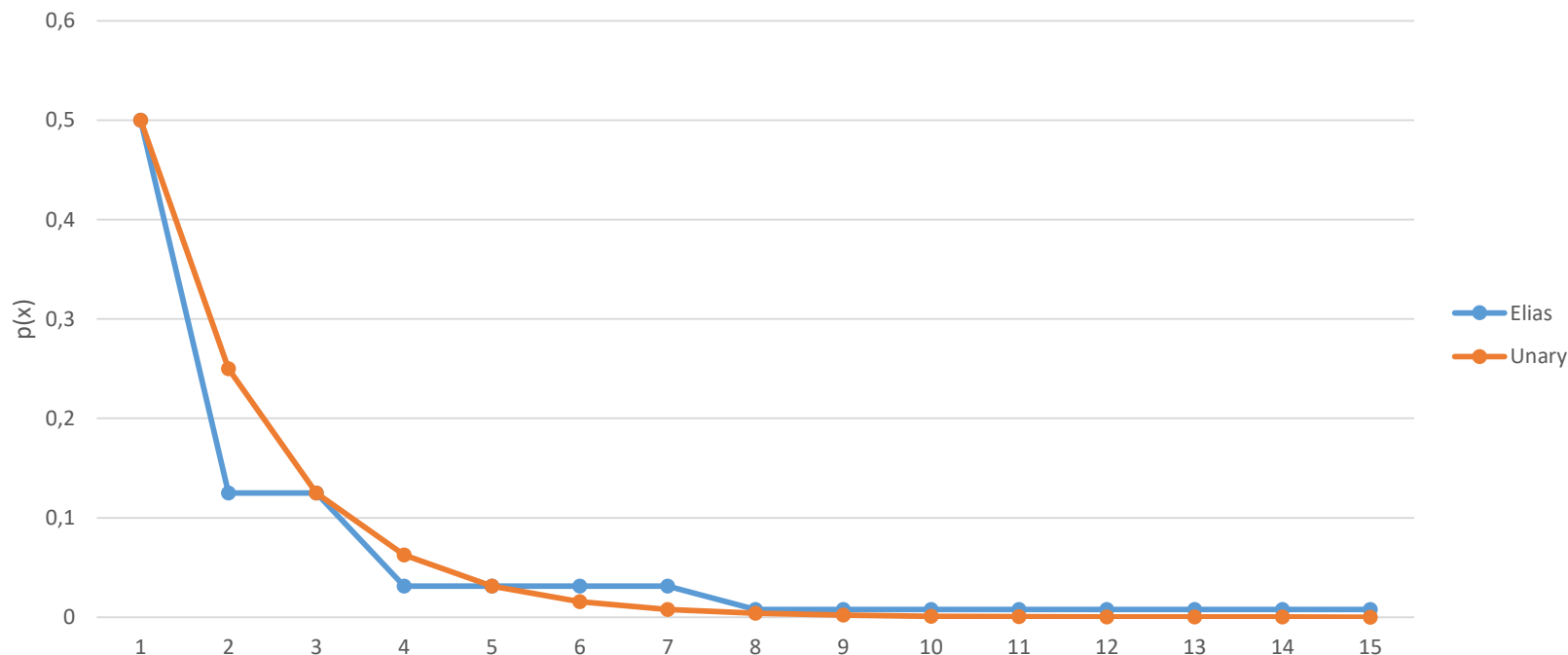
# Elias $\gamma$ coding

- Still making the hypothesis that small numbers are more probable than larger ones, we can divide them in classes based on their binary representation.
- The Elias gamma code is the simplest of the Elias codes (we won't see all of them) and is defined as follows.
- To code a natural number  $x \in \mathbb{N} = \{1, 2, 3, \dots\}$ , write its binary representation preceded by  $\lfloor \log_2 x \rfloor$  zeros. Notice that  $\lfloor \log_2 x \rfloor + 1$  is the number of digits required to write  $x$  in binary:

1	1	0
2	010	101
3	011	100
4	00100	11011
5	00101	11010
6	00110	11001
7	00111	11000
8	0001000	1110111
9	0001001	1110110
...	...	...

# Elias $\gamma$ coding

- In practice you can see a unary code which identifies the *class* to which each number belongs to, followed by a binary number that identifies the specific number in the class.
- This means that the assumption is that numbers in each class are uniformly distributed (so the binary encoding is optimal) and that the classes are distributed as negative powers of two.



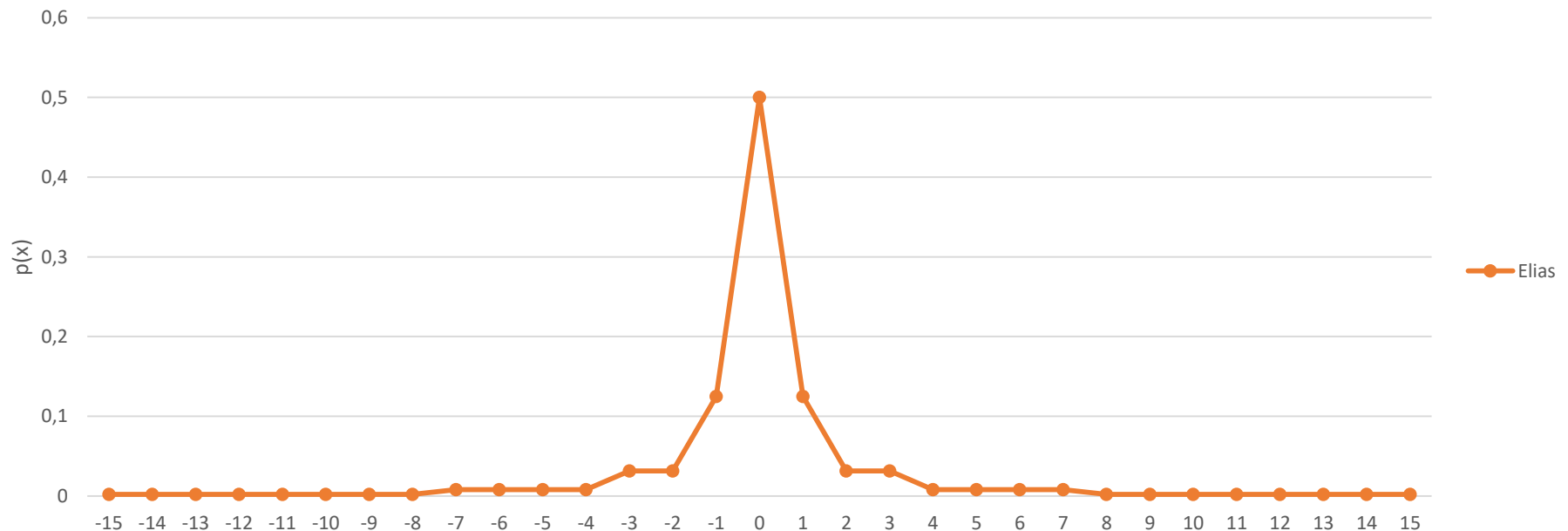
## Elias $\gamma$ coding

- Elias  $\gamma$  coding is particularly apt to encode positive *and negative* numbers in multimedia applications, where numbers which have a small absolute value are more likely than those with a larger one.
- What transformation from  $(-\infty, +\infty)$  to  $[1, +\infty)$  should we use?

# Elias $\gamma$ coding

- Elias  $\gamma$  coding is particularly apt to encode positive *and negative* numbers in multimedia applications, where numbers which have a small absolute value are more likely than those with a larger one.
- What transformation from  $(-\infty, +\infty)$  to  $[1, +\infty)$  should we use?
- For example:

$$\text{map}(x) = \begin{cases} 1 - 2x & x \leq 0 \\ 2x & x > 0 \end{cases}$$





# Algorithm for the Optimal Encoding

- In 1952, David A. Huffman published an algorithm to generate an optimal variable length coding.
- It is not possible to obtain encodings with a lower average length.
- This is true only if we consider codes that represent each symbol with an integer number of bits. But this encoding is a cornerstone of data compression, used today in JPEG compression for example.
- This algorithm starts from the assumption that we aim at generating variable length codes such that it is not necessary to add anything between one to the other to distinguish the end of a code and the beginning of the following.
- For example, if you want to encode a set of 4 symbols you can use the codes 01, 102, 111, 202 (using 3 values: 0, 1 and 2).
- The sequence 1111022020101111102 can only be interpreted as 111-102-202-01-01-111-102.
- If we used the codes 11, 111, 102, 02, this would not be possible.

# Huffman Encoding

- The algorithm consists of an a priori estimate of the probability of each symbol, followed by the construction of a system of codes that depends on this probability in order to represent the most probable symbols with fewer bits.
- The possible symbols must therefore be sorted in increasing order of probability.
- Take the two symbols with lower probability and combine them into a new set with probability given by the sum of the original probabilities (i.e., the probability that one or the other event occurs).
- You reorder the symbols/sets (eliminating the two combined) and start again the process.
- This procedure generates a tree that can be traversed from the root (the last combination) to the leaves by assigning a 0 if the least probable branch is chosen and a 1 if the other is chosen, or vice versa.
- The code is obviously never univocal since the choice to issue 0 or 1 is arbitrary.

# Example

Original Message Ensemble	Message Probabilities											
	Auxiliary Message Ensembles											
	1	2	3	4	5	6	7	8	9	10	11	12
0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20
0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18
0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10
0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10
0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10
0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
*0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

# Example

$i$	$P(i)$	$L(i)$	$P(i)L(i)$	$Code$
1	0.20	2	0.40	10
2	0.18	3	0.54	000
3	0.10	3	0.30	011
4	0.10	3	0.30	110
5	0.10	3	0.30	111
6	0.06	4	0.24	0101
7	0.06	5	0.30	00100
8	0.04	5	0.20	00101
9	0.04	5	0.20	01000
10	0.04	5	0.20	01001
11	0.04	5	0.20	00110
12	0.03	6	0.18	001110
13	0.01	6	0.06	001111
			$L_{av} = 3.42$	

# Probabilities Estimation

- As said, a probabilities estimation of the different symbols is needed.
- There are mainly 3 approaches for this purpose:
  - **Static a priori estimation**: the probabilities of different symbols can be estimated on a huge amount of data in order to have a very reliable statistical model and knowing a priori both to the encoder and to the decoder.
  - **Static estimation of the sequence**: two passes are performed on the data to be compressed. The first pass carries out the estimate of the probabilities, while the second pass carries out the coding. It is therefore necessary to also transmit the probabilities before the message.
  - **Dynamic estimation**: this approach starts with a static a priori estimate (for example a uniform distribution), then encodes/decodes the first symbol, updates the probability model and then the encoding, and so on. It is not necessary to transmit the distribution, but the computational load can become significant. Methods that use this technique are called Adaptive Huffman Coding.

# A Complete Example

- Let's suppose we want to compress the following message:

piccolo esempio di compressione dati con  
l'algoritmo di huffman

- The message is composed of 63 characters also counting the spaces and the apostrophe.
- The symbols are 19 and their distribution is as follows:

o,spazio=8

i=7

c,e,m=4

p,l,s,d,n,a=3

r,t,f=2

',g,h,u=1

# A Complete Example

o=8  
spazio=8  
i=7  
c=4  
e=4  
m=4  
p=3  
l=3  
s=3  
d=3  
n=3  
a=3  
r=2  
t=2  
f=2  
'=1  
g=1  
h=1  
u=1

o=8  
spazio=8  
i=7  
c=4  
e=4  
m=4  
p=3  
l=3  
s=3  
d=3  
n=3  
a=3  
(h+u)=2  
r=2  
t=2  
f=2  
'=1  
g=1

o=8  
spazio=8  
i=7  
c=4  
e=4  
m=4  
p=3  
l=3  
s=3  
d=3  
n=3  
a=3  
('+g)=2  
(h+u)=2  
r=2  
t=2  
f=2

# A Complete Example

o=8  
spazio=8  
i=7  
(t+f)=4  
c=4  
e=4  
m=4  
p=3  
l=3  
s=3  
d=3  
n=3  
a=3  
('+g)=2  
(h+u)=2  
r=2

o=8  
spazio=8  
i=7  
((h+u)+r)=4  
(t+f)=4  
c=4  
e=4  
m=4  
p=3  
l=3  
s=3  
d=3  
n=3  
a=3  
('+g)=2

o=8  
spazio=8  
i=7  
(a+('+g))=5  
((h+u)+r)=4  
(t+f)=4  
c=4  
e=4  
m=4  
p=3  
l=3  
s=3  
d=3  
n=3



# A Complete Example

o=8  
spazio=8  
i=7  
(d+n)=6  
(a+('+g))=5  
((h+u)+r)=4  
(t+f)=4  
c=4  
e=4  
m=4  
p=3  
l=3  
s=3

o=8  
spazio=8  
i=7  
(l+s)=6  
(d+n)=6  
(a+('+g))=5  
((h+u)+r)=4  
(t+f)=4  
c=4  
e=4  
m=4  
p=3

o=8  
spazio=8  
(m+p)=7  
i=7  
(l+s)=6  
(d+n)=6  
(a+('+g))=5  
((h+u)+r)=4  
(t+f)=4  
c=4  
e=4

# A Complete Example

(c+e)=8  
o=8  
spazio=8  
(m+p)=7  
i=7  
(l+s)=6  
(d+n)=6  
(a+('+g))=5  
(((h+u)+r)=4  
(t+f)=4

(((h+u)+r)+(t+f))=8  
(c+e)=8  
o=8  
spazio=8  
(m+p)=7  
i=7  
(l+s)=6  
(d+n)=6  
(a+('+g))=5

((d+n)+(a+('+g)))=11  
(((h+u)+r)+(t+f))=8  
(c+e)=8  
o=8  
spazio=8  
(m+p)=7  
i=7  
(l+s)=6

# A Complete Example

$(i+(l+s))=13$   
 $((d+n)+(a+('g)))=11$   
 $((h+u)+r)+(t+f)=8$   
 $(c+e)=8$   
 $o=8$   
 $spazio=8$   
 $(m+p)=7$

$(spazio+(m+p))=15$   
 $(i+(l+s))=13$   
 $((d+n)+(a+('g)))=11$   
 $((h+u)+r)+(t+f)=8$   
 $(c+e)=8$   
 $o=8$

$((c+e)+o)=16$   
 $(spazio+(m+p))=15$   
 $(i+(l+s))=13$   
 $((d+n)+(a+('g)))=11$   
 $((h+u)+r)+(t+f)=8$

# A Complete Example

$$(((d+n)+(a+('+g))))+(((h+u)+r)+(t+f)))=19$$

$$((c+e)+o)=16$$

$$(spazio+(m+p))=15$$

$$(i+(l+s))=13$$

$$((spazio+(m+p))+ (i+(l+s)))=28$$

$$(((d+n)+(a+('+g))))+(((h+u)+r)+(t+f)))=19$$

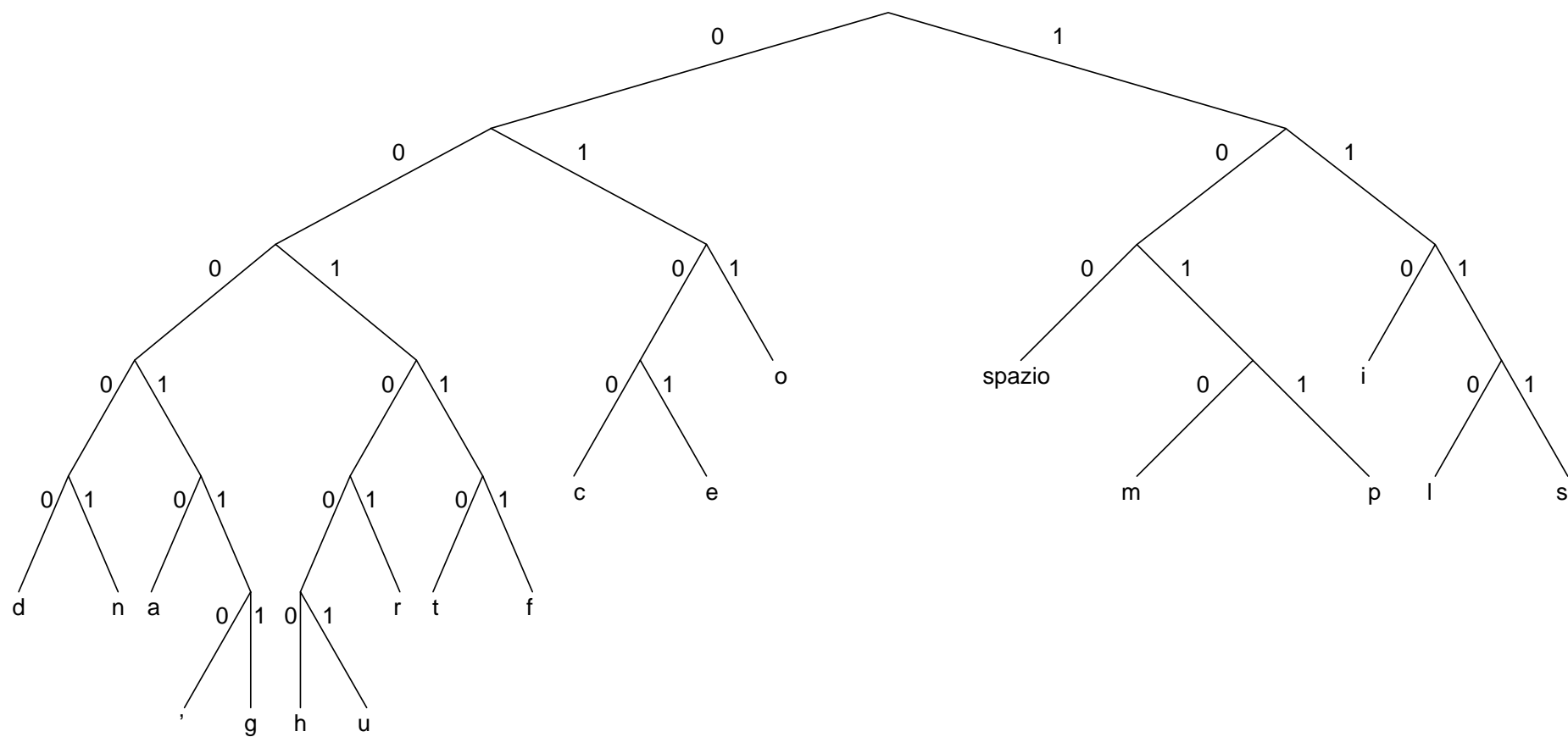
$$((c+e)+o)=16$$

$$((((d+n)+(a+('+g))))+(((h+u)+r)+(t+f))))+((c+e)+o))=35$$

$$((spazio+(m+p))+ (i+(l+s)))=28$$

$$((((((d+n)+(a+('+g))))+(((h+u)+r)+(t+f))))+((c+e)+o))+((spazio+(m+p))+ (i+(l+s))))=63$$

# The Resulting Tree



# The Resulting Encoding

Symbol	Encoding	Length (l)	Occurrences (o)	l*o
o	011	3	8	24
spazio	100	3	8	24
i	110	3	7	21
c	0100	4	4	16
e	0101	4	4	16
m	1010	4	4	16
p	1011	4	3	12
l	1110	4	3	12
s	1111	4	3	12
d	00000	5	3	15
n	00001	5	3	15
a	00010	5	3	15
r	00101	5	2	10
t	00110	5	2	10
f	00111	5	2	10
'	000110	6	1	6
g	000111	6	1	6
h	001000	6	1	6
u	001001	6	1	6
			tot	252

# Classical Encoding with Fixed Length

Symbol	Encoding	Length (l)	Occurrences (o)	$l \cdot o$
o	00000	5	8	40
spazio	00001	5	8	40
i	00010	5	7	35
c	00011	5	4	20
e	00100	5	4	20
m	00101	5	4	20
p	00110	5	3	15
l	00111	5	3	15
s	01000	5	3	15
d	01001	5	3	15
n	01010	5	3	15
a	01011	5	3	15
r	01100	5	2	10
t	01101	5	2	10
f	01110	5	2	10
'	01111	5	1	5
g	10000	5	1	5
h	10001	5	1	5
u	10010	5	1	5
			tot	315

# Canonical Huffman Code

- To decode a message encoded with the Huffman algorithm, it is therefore necessary to have a table that associates the corresponding code with each symbol.
- Transmitting this table would require for each symbol to transmit the symbol itself (with a certain number of bits), the length (with a certain number of bits) and the code bits (how many specified by the length).
- Among all the possible codes that can be generated with the Huffman algorithm, however, it is possible to choose two (dual) which can only be obtained from the lengths associated with the symbols: it is therefore useless to transmit the code table.
- These are called *canonical* encodings!
- The algorithm to obtain one of the two (the one that starts from 0) is very simple: it starts from the value 0 represented with 1 bit. Then, for each symbol, shift value to the left is performed until you reach the required length (zeros are added to the right until the length is reached). That is the code for current symbol. Then increment value and go to the next symbol.



# Canonical Huffman Code

Symbol	Length	Canonical code from 0	Canonical code from 1
o	3	000	111
spazio	3	001	110
i	3	010	101
c	4	0110	1001
e	4	0111	1000
m	4	1000	0111
p	4	1001	0110
l	4	1010	0101
s	4	1011	0100
d	5	11000	00111
n	5	11001	00110
a	5	11010	00101
r	5	11011	00100
t	5	11100	00011
f	5	11101	00010
'	6	111100	000011
g	6	111101	000010
h	6	111110	000001
u	6	111111	000000

Notice that the codes start from all 0 you get to all 1. The coding that starts from all 1 is the complement of the other.

# The Context

- In this algorithm, we consider symbols that come from a source without memory, which generate symbols independent one another with a certain probability.
- Albeit the assumption is very convenient it almost never occurs.
- This text, for example, does not come from a source of this type; in fact there are links between one letter and the next one: the probability of observing a p after an n in an Italian text is zero (unless there are grammatical errors!).
- If we encode characters pairs, we could obtain a more efficient encoding in terms of average length of the codes.
- The context can be used in different ways, employing it to change the message in order to produce less uniform distribution of messages, or using it to calculate conditional probabilities.
- In some cases it is possible that the context itself is enough to reduce the size of the messages thus not requiring additional systems for the compression.

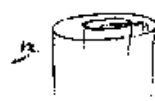
# Run Length Encoding

- It is the simplest context-based compression system.
- It encodes sequences (runs) of equal elements instead of single elements. It doesn't work very well for text, but it has noticeable effects on black and white images.
- Having the sequence aaaabbbbcccccaaaa it can be translated into (4, a) (3, b) (5, c) (4, a).
- With this kind of algorithms, several implementation aspects come into play that can make the difference:
  - How do I distinguish the number of occurrences from the occurrences themselves?
  - Do I always send encoded symbols or not?
  - Do I insert a code that can switch from coding to non-coding?
- Several implementations are possible, and these must be adapted to the specific application.
- The messages obtained must be encoded. It would be possible to apply the Huffman algorithm to the different codes or pairs of codes obtained.

# Example

- The image on the right is a 518x723 pixels scan at 1 bit per pixel, i.e. black or white.
- The uncompressed image takes in memory 46.815 bytes.
- Using a very simple RLE algorithm, one byte is sent to indicate the number of black pixels, then one to indicate the number of white pixels and so on. If there are 255 or more pixels with the same color, the next byte still indicates pixels of that color.
- With this algorithm I obtained a 10.519 bytes file.
- This file contains many white runs and the devised algorithms works very well.

①



$$r = 4.1 \cdot 10^{-2} \text{ m} \quad R = 3.77 \cdot 10^{-2} \text{ m}$$

$$h = 37 \cdot 10^{-3} \text{ m} \quad R_1 = 6.18 \cdot 10^{-2} \text{ m}$$

$$S_1 = 26.1 \text{ mC/m}^2 \quad P_1 = -18 \text{ nC/m}^2$$

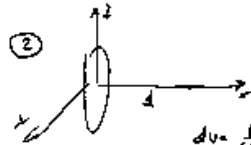
$$\lambda_1^* = V_1 \cdot 2\pi R_1 \quad \lambda_2^* = V_2 \cdot 2\pi R_2$$

$$E_1 = \frac{\lambda_1^*}{2\pi \epsilon_0} \frac{1}{h_1} \quad E_2 = \frac{\lambda_2^*}{2\pi \epsilon_0} \frac{1}{h_2} = \frac{\lambda_2^*}{2\pi \epsilon_0} \frac{1}{h_2}$$

$$E_1 = 2.19 \cdot 10^9 \text{ N/C} \quad E_2 = 4.36 \cdot 10^9 \text{ N/C}$$

Radiation esterna      Radiation interna

②



$$R = 3 \text{ m} \quad Q = 9 \cdot 10^{-9} \text{ C}$$

$$d = 4 \text{ m} \quad q = -6 \cdot 10^{-12} \text{ C}$$


$$dV = \frac{1}{4\pi \epsilon_0} \frac{dQ}{r} \quad \frac{dQ}{r} = \frac{1}{4\pi \epsilon_0} \frac{dQ}{\sqrt{R^2 + d^2}}$$

$$V = \frac{1}{4\pi \epsilon_0} \frac{Q}{\sqrt{R^2 + d^2}}$$

$$V_0 = 4.2 \text{ V} \quad V_1 = 27 \text{ V}$$

$$U = (V_1 - V_0) \cdot q = 64.3 \cdot 10^{-19} \text{ Joule}$$

③

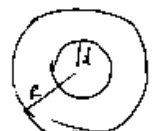


$$\frac{dB}{dt} = 10 \text{ mT/s} \quad r = 4 \text{ cm}$$

$$\oint \mathbf{E} \cdot d\mathbf{l} = -\frac{d\Phi_B}{dt} \quad E = \frac{1}{2} \frac{dB}{dt} r$$

$$E = \frac{0.01}{2} = 5 \cdot 10^{-3} \text{ V/m}$$

④



$$I_2 = 2 \text{ mA/A} = 2 \cdot 10^{-3} \text{ A/m}^2$$

$$d = 50 \cdot 10^{-3} \text{ m}$$

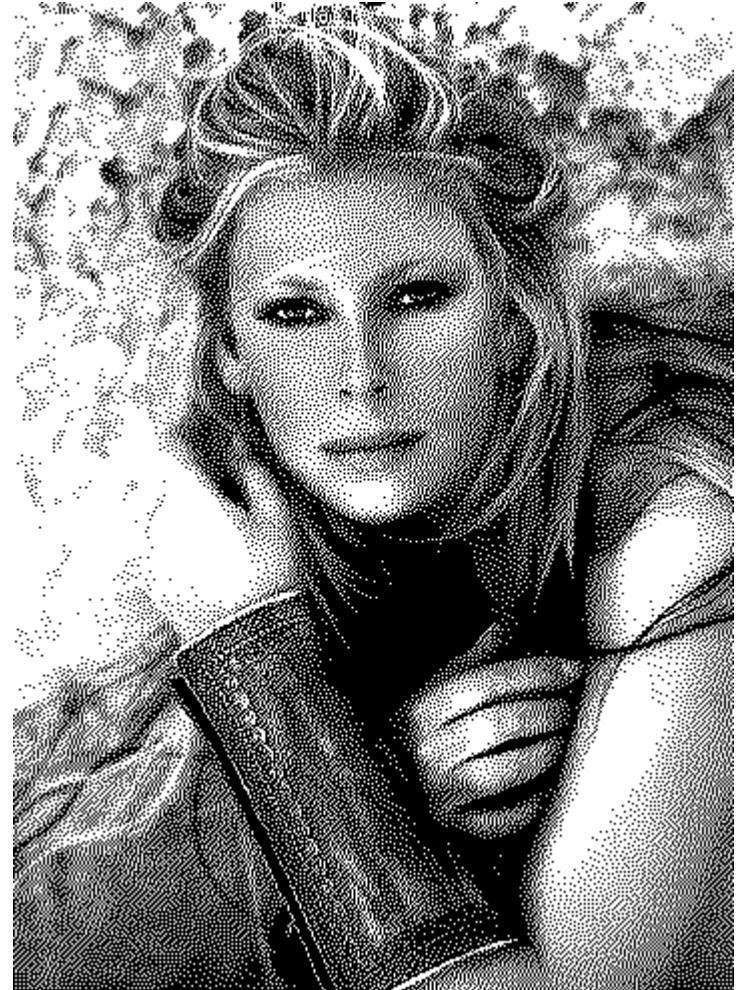
$$\oint \mathbf{B} \cdot d\mathbf{l} = \mu_0 \int I_2 dS \Rightarrow B = \frac{\mu_0 I_2 d}{2}$$

$$B = 6.3 \cdot 10^{-7} \text{ T}$$

$$\frac{dB}{dt} = 5 \text{ V/m} = 2.7 \cdot 10^{-11} \text{ V/m}$$

# Example

- The image at the right is at 1 bpp of 360x494 pixels which occupies 22,230 bytes in memory.
- The "compression" with the previous algorithm produces a 78.920 bytes file!
- This is because the algorithm uses at least one byte for each run.
- It is possible to obtain better results, but in this case the model we are using does not correspond to the data that is being modeled.



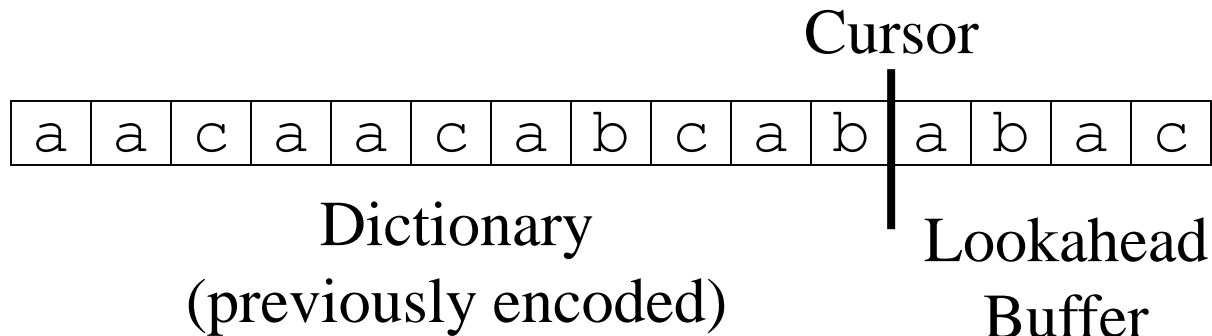
## RLE Variation: Packbits

- In the PostScript standard, a variant of the RLE was used. This variation is inspired by an algorithm known as Packbits.
- The RunLengthEncode filter encodes the data in a simple byte format consisting of *commands* for the decoder.
- Each command consists of a byte indicating the length  $L$  followed by 1-128 bytes of data.
- If  $L$  is between 0 and 127 the command is a copy: the following  $L + 1$  (1-128) bytes are copied as they are (no encoding).
- If  $L$  is between 129 and 255 the command is a run: the next byte is output  $257-L$  times (2-128 times).
- If  $L$  is 128 the command is End Of Data. The value 128 is called the EOD marker.

# Lempel-Ziv Algorithms

- LZ77 (sliding window algorithm)
  - Variations: LZSS (Lempel-Ziv-Storer-Szymanski)
  - Applications: `gzip`, Squeeze, LHA, PKZIP, ZOO
- LZ78 (dictionary base technique)
  - Variations : LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
  - Applications: `compress`, GIF, CCITT (in modems), ARC, PAK
- Traditionally, the LZ77 was the best, but very slow. In the version used by `gzip`, however, it is similar to the LZ78.

# LZ77: Lempel-Ziv Sliding Window



- The window containing the dictionary and the buffer are of fixed length and scroll with the cursor.
- At each step:
  - Send out the triplet (P, L, C)  
P = position of the longest match in the dictionary, relative to the current position (backward distance from the cursor)  
L = length of the longest match  
C = next character to be buffered after the correspondence.
  - Move the window L + 1 steps forward



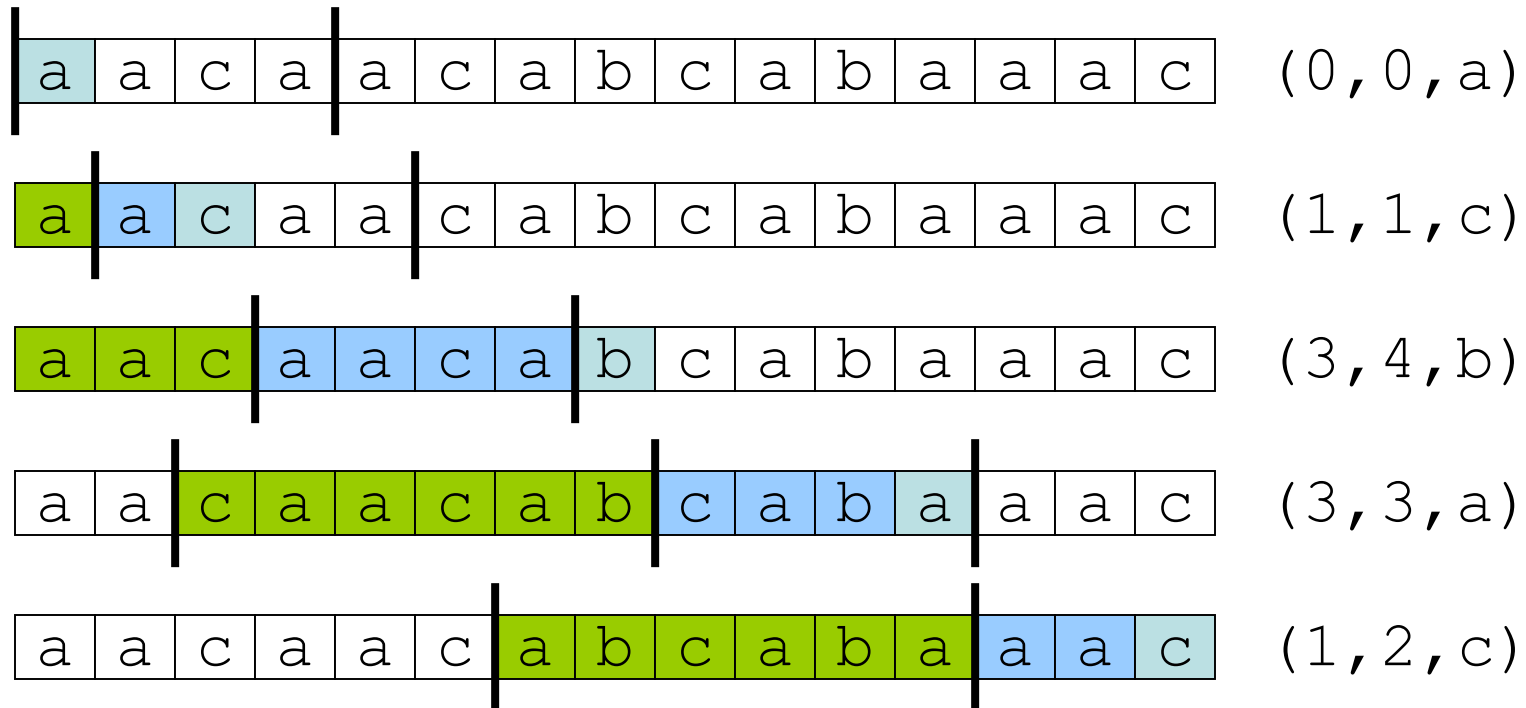
## LZ77: Decoding

- The decoder has the same dictionary window as the encoder and builds it dynamically.
- For each message, the decoder has just to move back in the dictionary, insert a copy at the current position and then insert the additional character.
- What does it mean if  $L > P$ , that is, if only a part of the message is present in the dictionary?
- E.g.: dictionary = abcd, symbol to decode = (2, 9, e)
- You simply copy from the dictionary (which is also the current output) starting from the cursor with something like this:

```
for (i = 0; i < length; i++)  
    out[cursor+i] = out[cursor-offset+i];
```

- Output = abcdcdcdcdcdce

# LZ77: Example

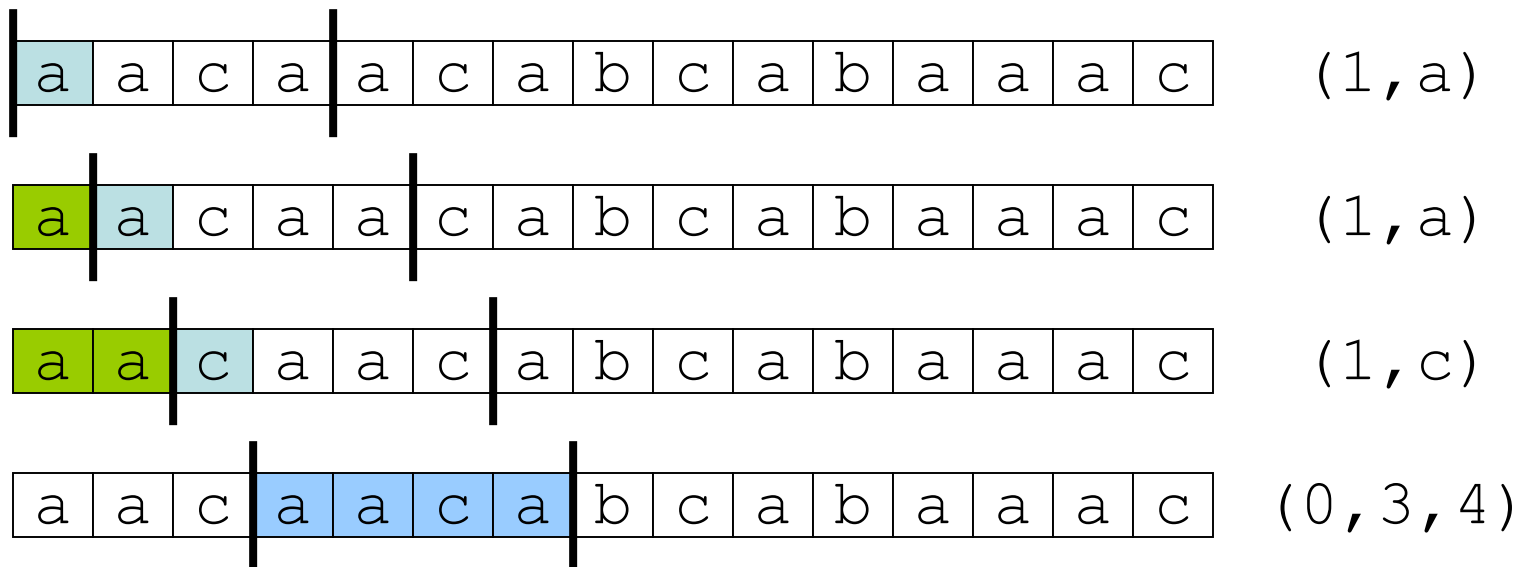


# LZ77: Optimizations Employed by `gzip`

LZSS: variation of the LZ77, it sends as output two different codes:

`(0, position, length)` or `(1, character)`

The second form is typically used when the length is less than 3.



# Optimizations Employed by `gzip`

- Position, length and characters are coded with Huffman
- Some special rules (which we will not see) are introduced to prevent the use of the longest string right now if we can do better in the next step  
→ **better an egg today than a hen tomorrow!**
- From what concerns the implementation, hash tables are used to store the dictionary:
  - Hash based on strings of length 3.
  - The best match is found within the correct bucket.
  - Limits are imposed on the length of the search.
  - The data is stored in the bucket in position order.

# Theory Behind the LZ77

- The sliding window LZ algorithm has been proven to be asymptotically optimal [Wyner-Ziv, 94]
- Sufficiently long strings will be compressed up to the limit of the entropy of the message when the search window tends to infinity:

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \rightarrow \infty} H_n$$

- This using logarithmic position encoding.
- The problem is that "sufficiently long" means really very long! Usually not tractable from the computational point of view: the cost of the research is too high.

# LZ78: Lempel-Ziv with Dictionary

- Basic algorithm:
  - A dictionary of words is kept with an integer identifier for each entry (from an implementation point of view this aspect becomes fundamental: a structure like a *trie* can modify performance dramatically)
  - Encoding cycle
    - find the longest string **S** in the dictionary matching the string starting at current position
    - send the identifier of **S** and the next (after the correspondence) character **c**
    - add the string **Sc** to the dictionary
  - Decoding employs the same method to build the dictionary and looks for identifiers (no need to send the dictionary)

# LZ78: Encoding Example

	Output	Dictionary
a a b a a c a b c a b c b	(0, a)	1 = a
a a b a a c a b c a b c b	(1, b)	2 = ab
a a b a a c a b c a b c b	(1, a)	3 = aa
a a b a a c a b c a b c b	(0, c)	4 = c
a a b a a c a b c a b c b	(2, c)	5 = abc
a a b a a c a b c a b c b	(5, b)	6 = abcb

# LZ78: Decoding Example

Input		Dictionary													
(0, a)	<table><tr><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	a													1 = a
a															
(1, b)	<table><tr><td>a</td><td>a</td><td>b</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	a	a	b											2 = ab
a	a	b													
(1, a)	<table><tr><td>a</td><td>a</td><td>b</td><td>a</td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	a	a	b	a	a									3 = aa
a	a	b	a	a											
(0, c)	<table><tr><td>a</td><td>a</td><td>b</td><td>a</td><td>a</td><td>c</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	a	a	b	a	a	c								4 = c
a	a	b	a	a	c										
(2, c)	<table><tr><td>a</td><td>a</td><td>b</td><td>a</td><td>a</td><td>c</td><td>a</td><td>b</td><td>c</td><td></td><td></td><td></td><td></td></tr></table>	a	a	b	a	a	c	a	b	c					5 = abc
a	a	b	a	a	c	a	b	c							
(5, b)	<table><tr><td>a</td><td>a</td><td>b</td><td>a</td><td>a</td><td>c</td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td><td>c</td><td>b</td></tr></table>	a	a	b	a	a	c	a	b	c	a	b	c	b	6 = abcb
a	a	b	a	a	c	a	b	c	a	b	c	b			



## LZW (Lempel-Ziv-Welch) [Welch84]

- You can avoid sending the extra character **c** in output, but still add **Sc** to the dictionary!
- The dictionary is initialized with byte values from 0 to 255 (all possible single bytes that can be encountered). This initialization is fixed and therefore does not require any data transmission.
- The decoder rebuilds the code and the dictionary based only on the codes received.
- There is a special case that must be handled separately.

# LZW: Encoding Example

	Output	Dictionary
a a b a a c a b c a b c b	97	256=aa
a a b a a c a b c a b c b	97	257=ab
a a b a a c a b c a b c b	98	258=ba
a a b a a c a b c a b c b	256	259=aac
a a b a a c a b c a b c b	99	260=ca
a a b a a c a b c a b c b	257	261=abc
a a b a a c a b c a b c b	260	262=cab
a a b a a c a b c a b c b	98	263=bc
a a b a a c a b c a b c b	99	264=cb
	98	

# LZW: Decoding Example

Input

Dictionary

97	a	a	b	a	a	c	a	b	c	a	b	c	b	
97	a	a	b	a	a	c	a	b	c	a	b	c	b	256=aa
98	a	a	b	a	a	c	a	b	c	a	b	c	b	257=ab
256	a	a	b	a	a	c	a	b	c	a	b	c	b	258=ba
99	a	a	b	a	a	c	a	b	c	a	b	c	b	259=aac
257	a	a	b	a	a	c	a	b	c	a	b	c	b	260=ca
260	a	a	b	a	a	c	a	b	c	a	b	c	b	261=abc
98	a	a	b	a	a	c	a	b	c	a	b	c	b	262=cab
99	a	a	b	a	a	c	a	b	c	a	b	c	b	263=bc

...

# Special Case

- Let's take the following example: albalalaica
- The LZW compression is as follows:

```

out << 97      dict[256]=«al»
out << 108     dict[257]=«lb»
out << 98      dict[258]=«ba»
out << 256     dict[259]=«ala»
out << 259     dict[260]=«alai»
out << 105     dict[261]=«ic»
out << 99      dict[262]=«ca»
out << 97

```

- The decompression is as follows:

```

97 → a
108 → l      dict[256]=«al»
98 → b      dict[257]=«lb»
256 → al    dict[258]=«ba»
259 → ??? ←

```

We still didn't create  
entry 259 in the  
dictionary!

## Special Case

- When the code that we should have created in the current step is found, it means that this is equal to the last decoded code plus its first symbol.
- In the example case, the code 259 will be «al» with an extra «a» at the end, which means «ala»:

97 → a

108 → l      dict[256]=«al»

98 → b      dict[257]=«lb»

256 → al      dict[258]=«ba»

259 → ala      dict[259]=«ala»

105 → i      dict[260]=«alai»

99 → c      dict[261]=«ic»

97 → a      dict[262]=«ca»

- This happens frequently in repeated sequences, typical of run-length.

# A Problem to be Solved in LZ78 and LZW

- What if the dictionary gets too big?
  - When the dictionary reaches a certain size you delete it and start again as if the file started here (technique used in GIF: simple)
  - When the compression is not particularly effective (threshold on the ratio between the compression and the minimum input size) you delete the dictionary and start again (used in the unix `compress` utility)
  - The entry of the dictionary used least recently is deleted (least-recently-used or LRU) when the dictionary reaches a certain size (used in BTLZ, the standard of British Telecom)

# Summary on Lempel-Ziv Algorithms

- Both the LZ77 and LZ78 and their variations keep a "dictionary" of the recently viewed strings.
- The differences are:
  - How the dictionary is stored
  - How it is extended
  - How it is indexed
  - How the elements are removed
- They adapt well to changes in the data probability distribution (for example in a ZIP archive with different types of files).
- The first versions of these algorithms did not use a probabilistic encoding (Huffman or similar) and had rather poor performances (e.g. 4.5 bits/character for English texts)
- More "modern" versions (e.g. `gzip`) employ probabilistic encoding as a *second pass* and compress much more.
- They are becoming obsolete, but the ideas used are the basis of all the latest algorithms.