جامعة اللاذقية
كلية الهندسة المعلوماتية

Multimedia Systems

Lossless Compression Algorithms

Lecture 5

Data Compression

More and more data into digital form

– Libraries, museums, governments

– To be stored without any loss

Data Compression is the process of encoding or converting data in such a way that it consumes less memory space. Data compression reduces the number of resources required to **store** and **transmit** data.

It can be done in two ways- lossless compression and lossy compression. Lossy compression reduces the size of data by removing unnecessary information, while there is no data loss in lossless compression.

Data Compression

Original
976 KB

compressed
834 KB

Compression Ratio

A measure of how many times the original file size has been **reduced**.

Compression ratio = B0 / B1

The number of bits *before* compression is B0

The number of bits *after* compression is B1

Compression ratio must be **larger than 1.0** ;

The higher the compression ratio, the better the lossless compression scheme.

Example:

Original Size = 1000 KB , Compressed Size = 250 KB

CR= 1000/250= 4

The compressed file is 4 times smaller than the original.

Compression Savings %

The percentage of the original size that has been eliminated after compression.

A higher percentage means more space saved.

Compression saving = ((B0-B1)/B0)*100 %

For example:

Original Size = 1000 KB , Compressed Size = 250 KB

Savings= 75% (Compression removed 75% of the original file size).

Same data may give: CR = 4 and Savings = 75%

Basics of Information Theory

What is entropy?

Entropy in data compression denote the **randomness** of the data that you are inputting to the compression algorithm. That means the more random the text is, the lesser you can compress it.

- entropy H of an image is the theoretical minimum # of bits/pixel required to encode the image without loss of information
- No matter what coding scheme is used, it will never use fewer than H bits per pixel

Basics of Information Theory

The entropy of an information source – With alphabet S = {s1, s2,...sn}

$$\eta = H(s) = \sum_{i=0}^{n} p_i \log_2 \frac{1}{p_i} = -\sum_{i=0}^{n} p_i \log p_i$$

- P_i is the probability that symbol s_i in S will occur
- log2 ($1/p_i$) indicate the amount of information contained in characters.

For example:

the probability of n in a manuscript is 1/32, so

The amount of information is 5 bits

A character sting nnn require 15 bits to code.

Basics of Information Theory

Example:

Suppose a system has 4 states outcome, each outcome has probability ¼

$$\eta = 4 \times \frac{1}{4} \times \log_2 \frac{1}{(\frac{1}{4})} = 2bits$$

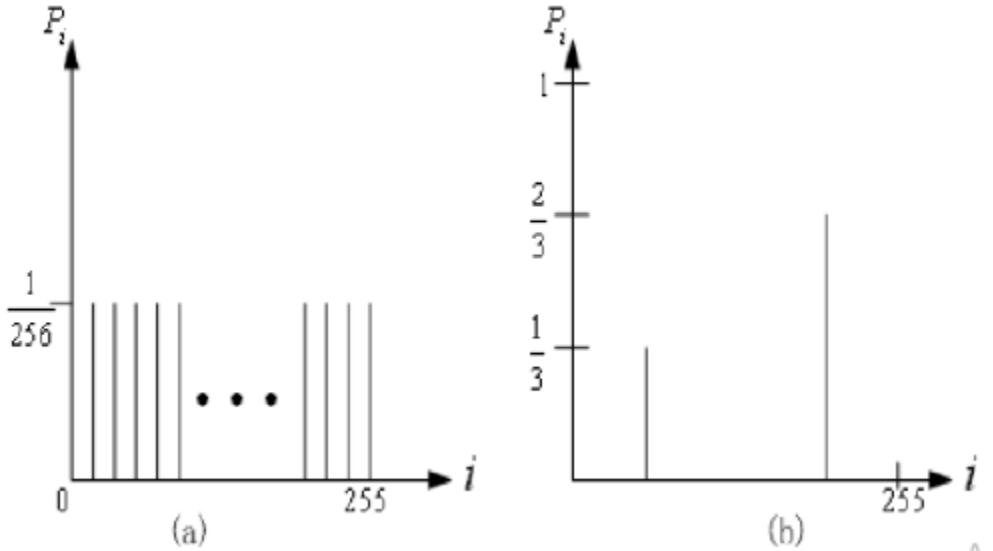
If one state had probability ½,the other three had probability 1/6 :

$$\eta = \frac{1}{2} \times \log_2 \frac{1}{(\frac{1}{2})} + 3 \times \log_2 \frac{1}{(\frac{1}{6})} = 1.795 < 2bits$$

The most-occurring one means fewer bits to send.

Basics of Information Theory

- The definition of entropy is aimed at identifying often-occurring symbols as short codewords
- For example, **E** occurs frequently in English, so we would give it a shorter code than **Q**.
- Example: - If the information source S is a gray level digital image. - Each s_i is a gray level intensity ranging between 0 and 255 - The image histogram is used to calculate the frequency pi of having pixels with gray- level intensity i in the image



Basics of Information Theory

The entropy of the two above images –

The entropy of image (a) is :

$$\eta = 256 \times \frac{1}{256} \times \log_2 \frac{1}{(\frac{1}{256})} = 8 bits$$

The entropy of image (b) is :

$$\eta = \frac{1}{3} \times \log_2 \frac{1}{(\frac{1}{3})} + \frac{2}{3} \times \log_2 \frac{1}{(\frac{2}{3})} = 0.92 bits$$

The entropy is greater when the probability is flat and smaller when it is more peaked

Basics of Information Theory

The entropy η is a weighted-sum of terms; $\log_2(1/p)$

hence it represents the average amount of information contained per symbol in the source S.

The entropy η specifies the lower bound for the average number of bits to code each symbol in S,

$$\eta \leq \bar{l}$$

$\bar{l}$ the average length (measured in bits) of the codewords produced by the encoder

Lossy vs. Lossless Compression

Lossy means that your image will "*lose*" some data, such as the number of pixels or colors. This loss is usually not noticeable to the naked eye and is enough for the web.

Unlike lossy compression, **lossless** compression lets you decompress an image without losing quality or information.

Lossless Compression

Basic Techniques

- Run-Length Encoding

Entropy coding / Statistical Methods

- Shannon-Fano Coding
- Huffman Coding
- Adaptive Huffman Coding
- Arithmetic Coding

Dictionary Methods

- The Lempel-Ziv

RLC (RUN-LENGTH CODING)

One of the simplest forms of data compression

Basic idea:

If symbols of information source tend to form continuous groups, code one such symbol and the length of the group instead of coding each symbol individually.

Examples

- Silence in audio data, Pauses in conversation
- Bitmaps (1 bit plane)
- Blanks in text or program source files
- Backgrounds in images

RLC (RUN-LENGTH CODING)

10	10	10	10	10	10	10	10
10	10	10	10	10	12	12	12
10	10	10	10	10	12	12	12
0	0	0	10	10	10	0	0
5	5	5	0	0	0	0	0
5	5	5	10	10	9	9	10
5	5	5	4	4	4	0	0
0	0	0	0	0	0	0	0

8x8 block of a gray level image

First row :10,8

Second row :10,5,12,3

Third row:10 5 12 3

Fourth row : 0,3,10,3,0,2

Fifth row :5 3 0 5

Sixth row :5,3,10,2,9,2,10,1

Seventh row :5,3,4,3,0,2

Eighth row :0,8

Horizontal RLC

RLC (RUN-LENGTH CODING)

Input sequence:

0,0,-3,5,1,0,-2,0,0,0,0,2,-4,3,-2,0,0,0,1,0,0,-2

Run-length sequence (1): (15 samples)

(0,2)(-3,1)(5,1)(1,1)(0,1)(-2,1)(0,4)(2,1)(-4,1)(3,1)(-2,1)(0,3)(1,1)(0,2)(-2,1)

Run-length sequence (2): (10 samples)

(# zeros to skip, next non-zero value)

(2,-3)(0,5)(0,1)(1,-2)(4,2)(0,-4)(0,3)(0,-2)(3,1)(2,-2)

- Method 2 reduces the number of samples to code
- It is effective when data stream contains long runs of zeros
- Implementation is simple

RLC（RUN-LENGTH CODING）

Example 1:

x='AAAAAABBBBCCC000AAB00000000DDDDEFAB0000‘

- Length of x before compression: 39
- Compression (A,6)(B,4)(c,3)(0,3)(A,2)(B,1)(0,8)(D,4)(E,1)(F,1)(A,1)(B,1)(0,4)
- Length of x after compression is 26
- Compression ratio =39/26= 1.5

RLC（RUN-LENGTH CODING）

Example 2:

x='Variety is the spice of life'

Length of x before compression is 28

- Compression

(V,1)(a,1)(r,1)(i,1)(e,1)(t,1)(y,1) (‘ ’,1)(i,1)(s,1) (‘ ’,1)(t,1) (h,1)(e,1)

(‘ ’,1) (s,1) (p,1)(i,1)(c,1)(e,1)(‘ ’,1)(o,1)(f,1)(‘ ’,1)(l,1)(l,1)(f,1)(e,1)

- Length of x after compression is 56
- Compression ratio =28/56= 0.5 < 1

Variable-Length Coding - Shannon Fano

Shannon Fano Algorithm is an entropy encoding technique for lossless data compression of multimedia.

It assigns a code to each symbol *based on their probabilities* of occurrence. It is a variable-length encoding scheme, that is, the codes assigned to the symbols will be of varying lengths.

The **Top-Down** manner

1. **Sort** the symbols according to the frequency count of their occurrences
2. Recursively divide the symbols into two parts, each with approximately the same number of counts(HOW??), until all parts contain only one symbol

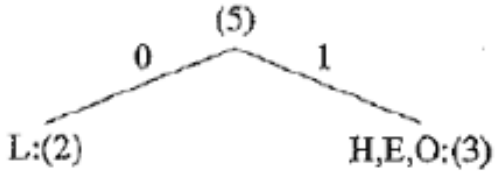
- A way of implementing the above procedure is to build a binary tree.

Variable-Length Coding - Shannon Fano

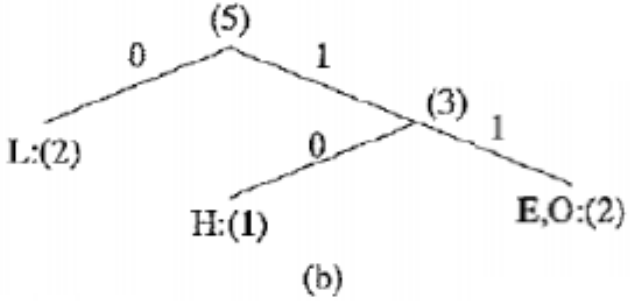
Example: Hello

Symbol	H	E	L	O
Count	1	1	2	1

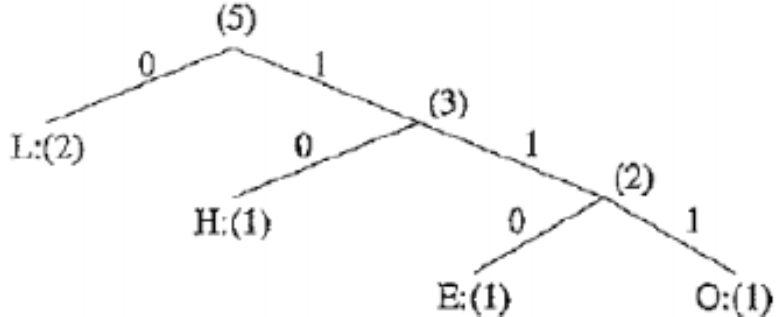
After iteration 1



After iteration 2



After iteration 3



Variable-Length Coding - Shannon Fano

Example: the entropy of example:

$$\eta = \frac{2}{5}\times\log_2\frac{5}{2}+\frac{1}{5}\log_2 5+\frac{1}{5}\log_2 5+\frac{1}{5}\log_2 5 = 1.92$$

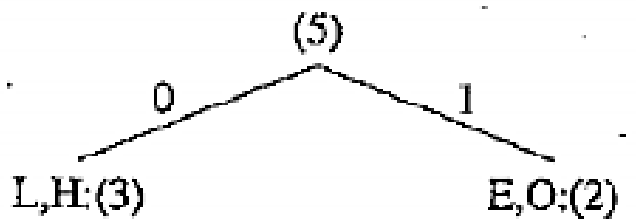
The result of applying shanon-fano on "Hello" word:

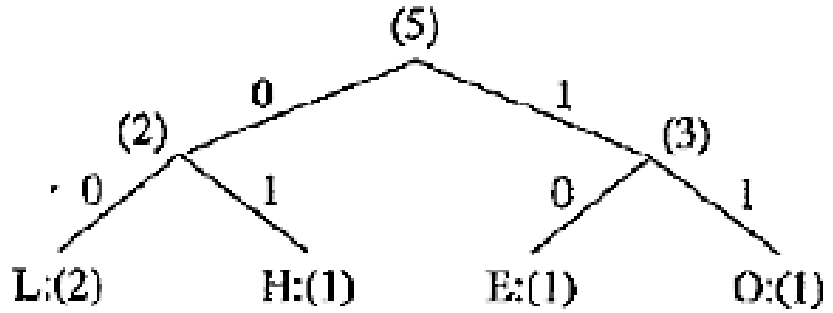
symbol	Count	$\text{Log}_2 P_i^{-1}$	Code	Number of bits used
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
Total number of bits : 10				

$$\bar{l} == (2+2+3+3)/5 = 10/5 = 2 \qquad \eta \le \bar{l}$$

Variable-Length Coding - Shannon Fano

Is the outcome of shanon-fano unique?:





(a)

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	Number of bits used
L	2	1.32	00	4
H	1	2.32	01	2
E	1	2.32	10	2
O	1	2.32	11	2
TOTAL number of bits:				10

(b)

Variable-Length Coding - Huffman coding

A **bottom-up** manner:

1. Initialization: put all symbols on the list sorted according to their frequency counts

2. Repeat until the list has only one symbol left :

- From the list, pick two symbols with the lowest frequency counts, form a Huffman sub-tree that has these two symbols as child nodes and create a parent node for them.
- Assign the sum of the children's frequency counts to the parent and insert it into the list, such that the order is maintained.
- Delete the children from the list.

3. Assign a codeword for each leaf based on the path from the root

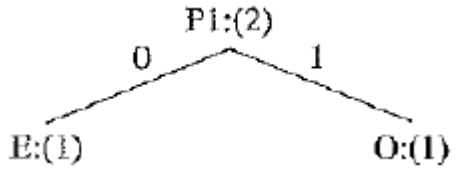
Variable-Length Coding - Huffman coding

Example: Hello

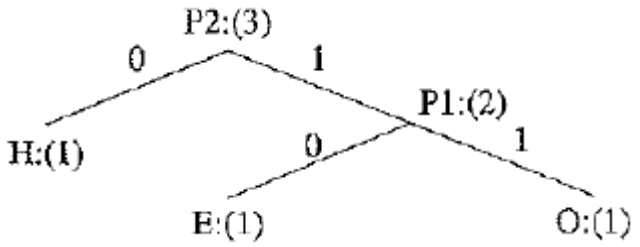
Symbol	H	E	L	O
Count	1	1	2	1

Symbol	L	H	E	O
Count	2	1	1	1

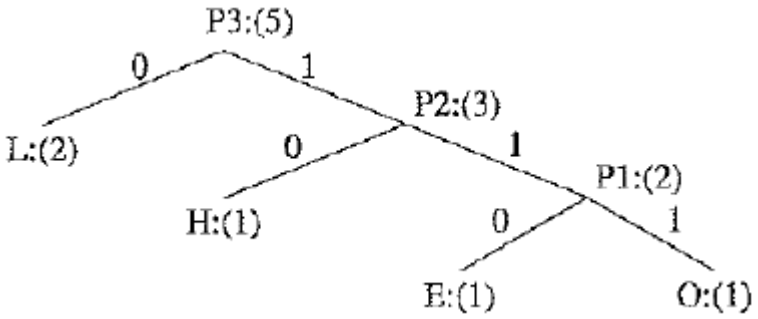
After iteration 1



After iteration 2



After iteration 3



Symbol	L	H	E	O
Code	0	10	110	111

The average number of bits used to code each character is (2+2+3+3)/5=2 bits

Variable-Length Coding - Huffman coding

For above example, Huffman coding generate the same coding result as Shannon-Fano algorithm

Another example:

A:(15), B:(7), C:(6), D:(6) and E:(5)
Shannon-Fano needs 89 bits ;
Huffman needs 87 bits

Huffman ➔ minimum

The statistics and/or coding tree are sent before the data to be decompressed

Properties of Huffman Coding

1. Unique Prefix Property: No Huffman code is a prefix of any other Huffman code - precludes any ambiguity in decoding.

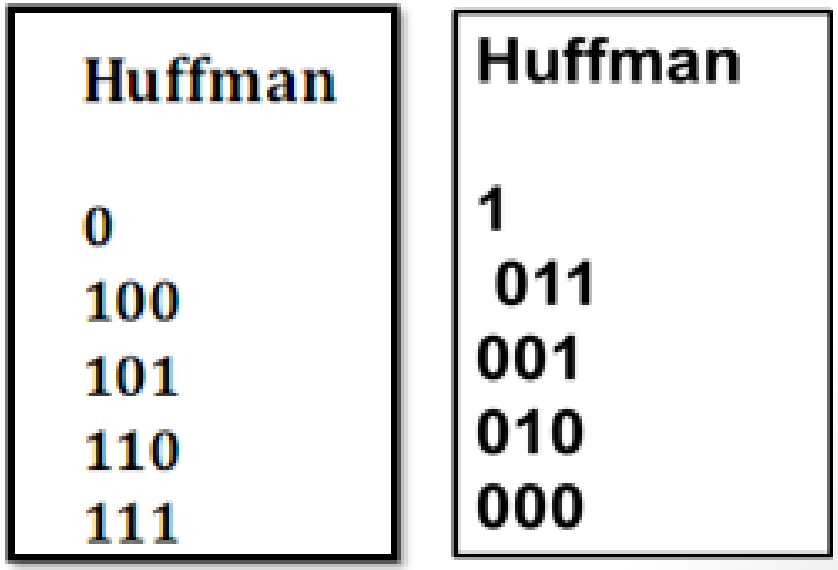
2. Optimality: minimum redundancy code - proved optimal for a given data model (i.e., a given, accurate, probability distribution):

    - The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit (LSB).
    - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.

Huffman Coding

x='ABBAAAACDEAAABBBDDEEAAACCCAEDDDABBAAEC'

Character	Frequency
‘A’	15
'B’	7
‘C’	6
‘D’	6
‘E’	5



Huffman Decoding

We need (encoding stream ,symbol ➔code) for decoding process

Use (symbol ➔ code) to build huffman tree

Decompress encoding data (leaf ➔ symbol)

Huffman Decoding

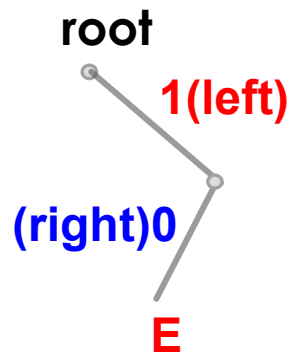
Example: Retrieve the data from the encoding stream '10000100010110111001' using this table:

E	25	10
I	15	01
A	15	111
D	12	001
C	7	1101
B	6	1100
G	6	0000
F	4	00010
H	1	00011

Huffman Decoding

Solution:

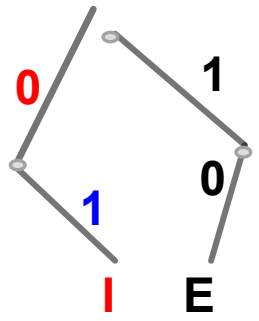
E	**25**	**10**
I	15	01
A	15	111
D	12	001
C	7	1101
B	6	1100
G	6	0000
F	4	00010
H	1	00011



Huffman Decoding

Solution:

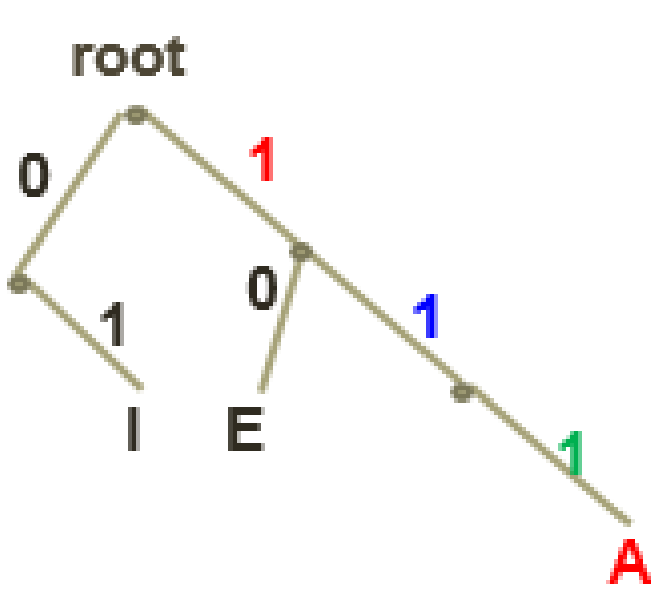
E	25	10
I	15	01
A	15	111
D	12	001
C	7	1101
B	6	1100
G	6	0000
F	4	00010
H	1	00011



Huffman Decoding

Solution:

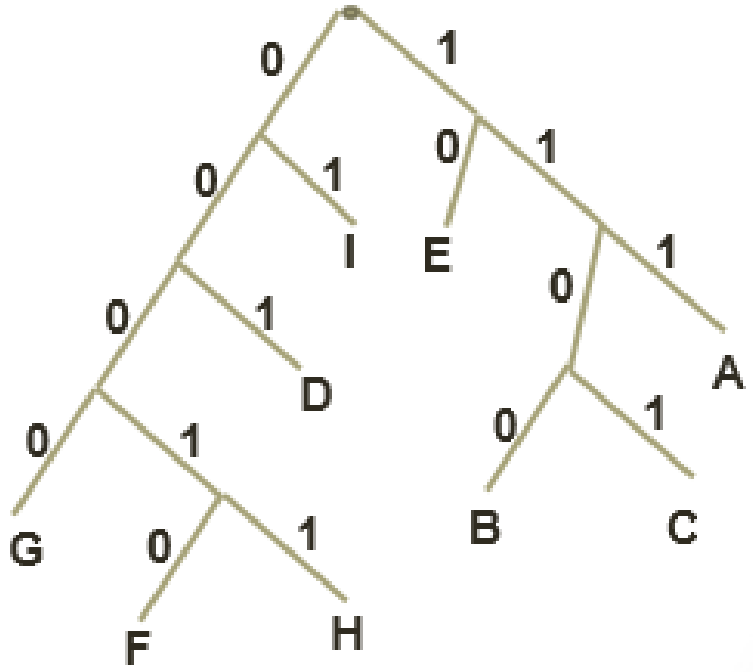
E	25	10
I	15	01
A	15	111
D	12	001
C	7	1101
B	6	1100
G	6	0000
F	4	00010
H	1	00011



Huffman Decoding

Solution:

E	25	10
I	15	01
A	15	111
D	12	001
C	7	1101
B	6	1100
G	6	0000
F	4	00010
H	1	00011

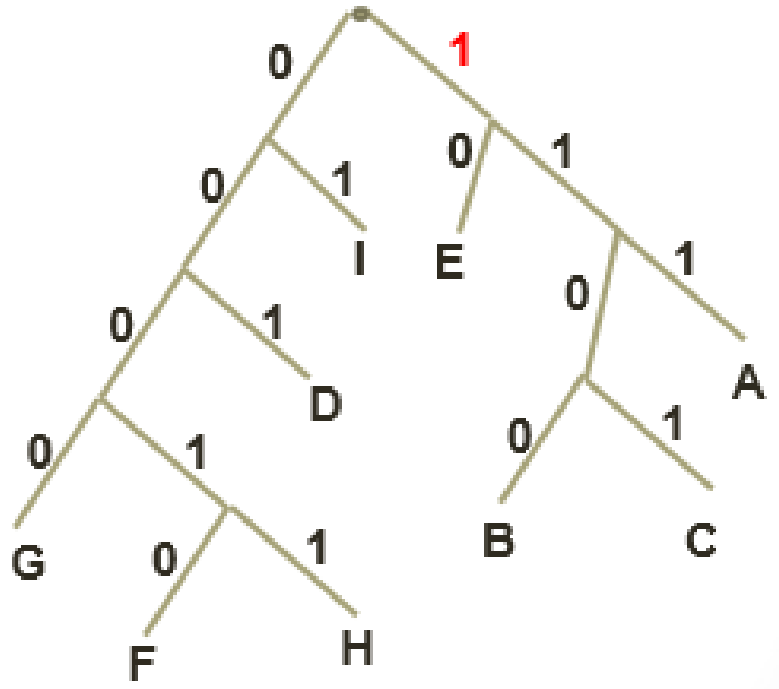


Huffman Decoding

Solution:

Decompress stream from tree

‘100001000101101110 01’



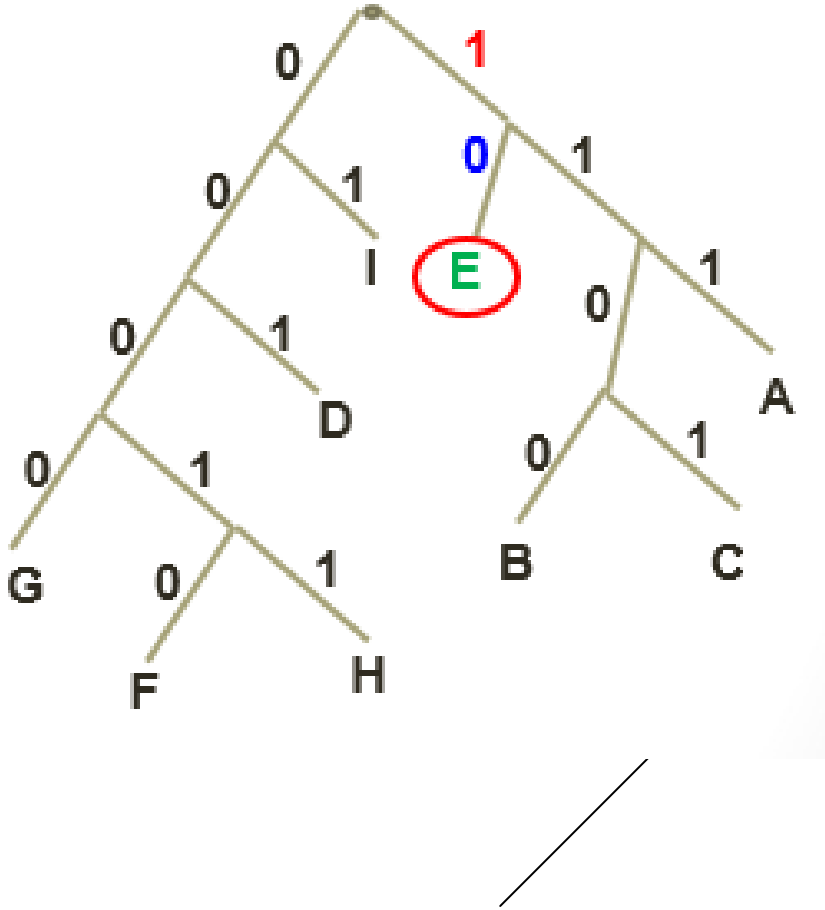
Huffman Decoding

Solution:

Decompress stream from tree

‘100001000101101110 01’

E

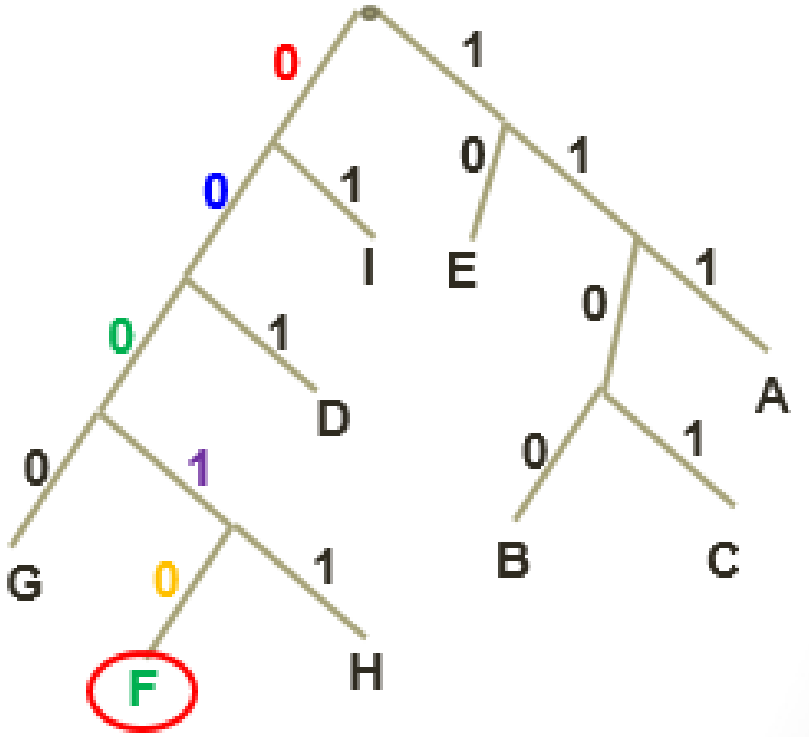


Huffman Decoding

Solution:

Decompress stream from tree

- ’10 00010 0010110111001’
- E F

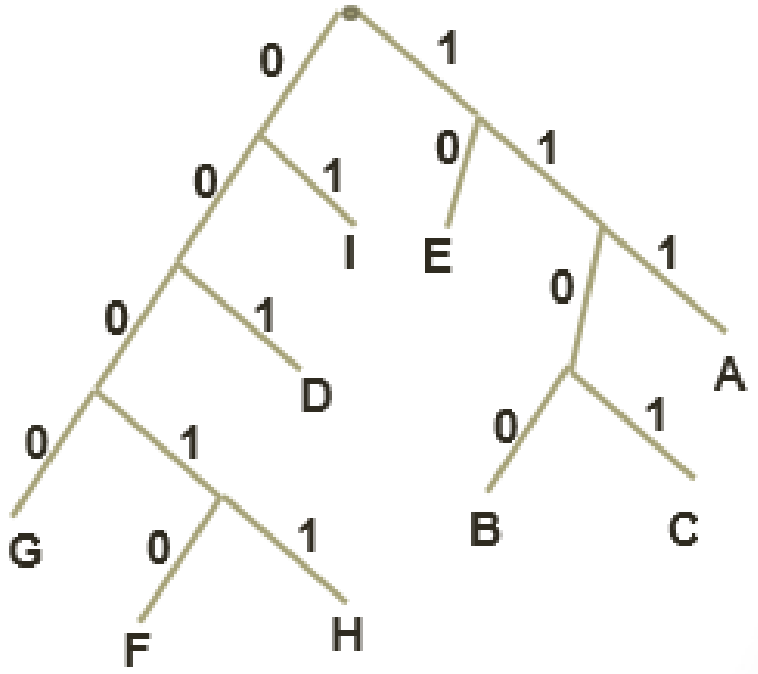


Huffman Decoding

Solution:

Decompress stream from tree

- ’10 00010 001 01 10 111 001’
- E F D I E A D



Arithmetic encoding:

- Entropy encoding
- Arithmetic coding is a more modern coding method that usually out-performs Huffman coding.
- Huffman coding encodes one symbol at a time and each symbol translates into an integral number of bits.
- Let's sacrifice the one to- one mapping between the symbol and its code word and encode the entire sequence of source symbols into one single code word (This is exactly the basis of arithmetic coding)

Arithmetic encoding:

Like Huffman coding, this too is a Variable Length Coding (**VLC**) scheme requiring a priori knowledge of the symbol probabilities.

Unlike Huffman coding, which assigns variable length codes to a fixed group symbols (usually of length one), arithmetic coding assigns variable length codes to a variable group of symbols.

Arithmetic encoding:

A message is represented by a half-open interval [a, b) where a and b are real numbers between 0 and 1.

Initially, the interval is [0, 1). When the message becomes longer, the length of the interval shortens and the number of bits needed to represent the interval increases.

Arithmetic encoding:

Example: suppose we have 3 symbols with probabilities:

Symbol	Probability
A	.5
B	.3
C	.2

Start with interval [0,1[

Divide it :

A = [0.0, 0.5)

B = [0.5, 0.8)

C = [0.8, 1.0)

We need to encode "**AB**"

A in interval: [0.0, 0.5) , we divide this interval again using the same probability ratios:

A inside the interval : [0.00, 0.25)

B inside the interval : [0.25, 0.40)

C inside the interval : [0.40, 0.50)

(Same as above: A = 50% of the new interval, B = 30%, C = 20%)

The second symbol is **B,** so the final interval is: [0.25 , 0.40)

Any number within the interval 0.25 to 0.40 represents the series "AB".

Arithmetic encoding:

Example: Encoding “eaii!” in Arithmetic Coding.

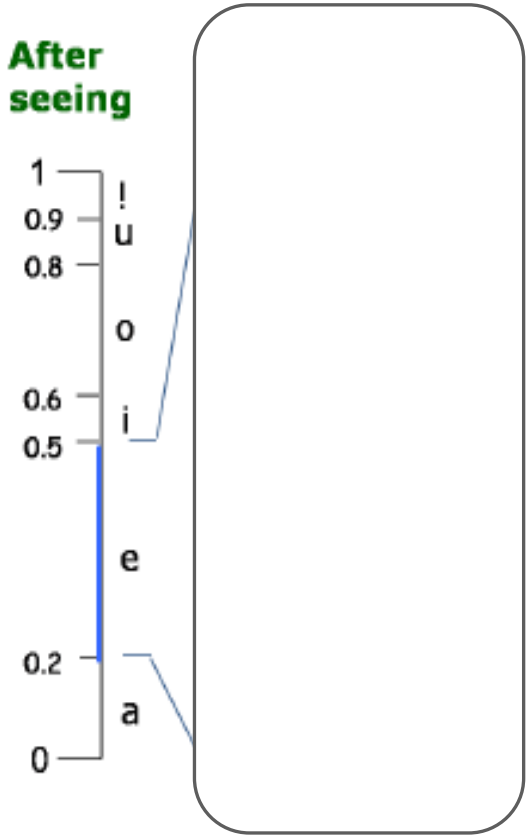
Symbol	Probability	Range
a	.2	[0 , 0.2(
e	.3	[0.2 , 0.5 (
i	.1	[0.5 , 0.6(
o	.2	[0.6 , 0.8(
u	.1	[0.8 , 0.9(
!	.1	[0.9 , 1 (

CDF (cumulative probability) is used here to partition the interval into sub-intervals

Arithmetic encoding:

e

Symbol	Probability	Range
a	.2	[0 , 0.2(
e	.3	[0.2 , 0.5 (
i	.1	[0.5 , 0.6(
o	.2	[0.6 , 0.8(
u	.1	[0.8 , 0.9(
!	.1	[0.9 , 1 (



L=0, H=1 ,r=1 e[0.2,0.5(→ current

new

L = L + Range * L of the symbol being coded=0+1*0.2=0.2

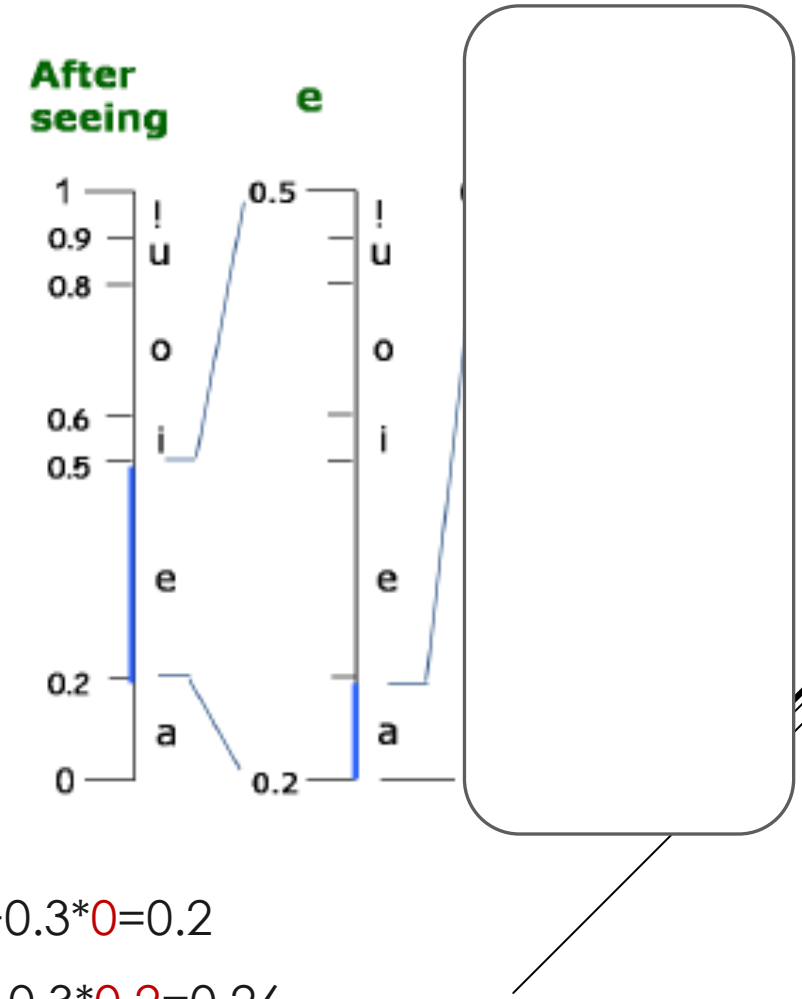
H = L + Range * H of the symbol being coded=0+1*0.5=0.5

Range=0.5-0.2=0.3

Arithmetic encoding:

ea

Symbol	Probability	Range
a	.2	[0 , 0.2(
e	.3	[0.2 , 0.5 (
i	.1	[0.5 , 0.6(
o	.2	[0.6 , 0.8(
u	.1	[0.8 , 0.9(
!	.1	[0.9 , 1 (



L=0.2, H=0.5 ,r=0.3 , a[0,0.2(→ current

new

L = L + Range * L of the symbol being coded=0.2+0.3*0=0.2

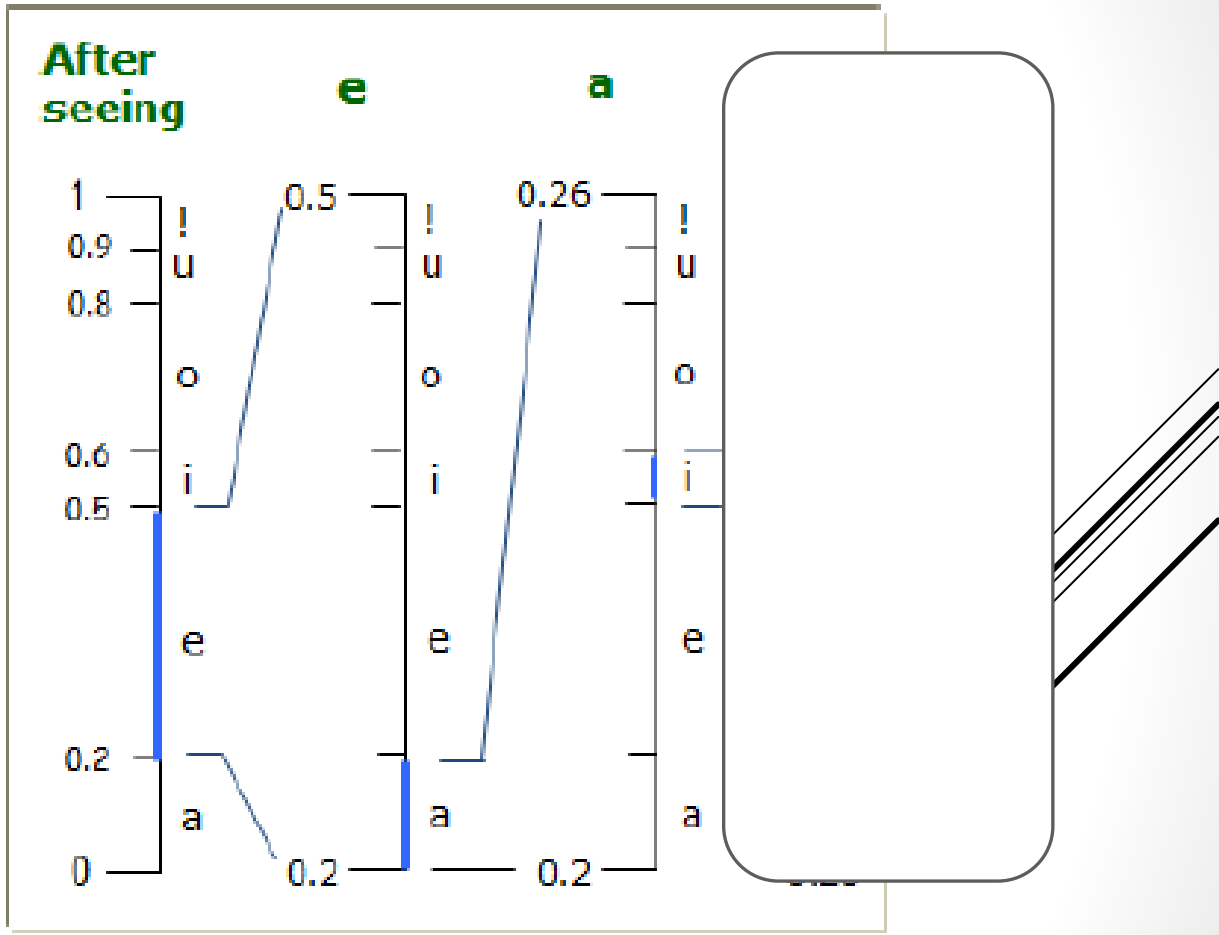
H = L + Range * H of the symbol being coded=0.2+0.3*0.2=0.26

Range=0.26-0.2=0.06

Arithmetic encoding:

eai

Symbol	Probability	Range
a	.2	[0 , 0.2(
e	.3	[0.2 , 0.5 (
i	.1	[0.5 , 0.6(
o	.2	[0.6 , 0.8(
u	.1	[0.8 , 0.9(
!	.1	[0.9 , 1 (



L=0.2, H=0.26,r=0.06 a[0.5,0.6) → current

new

L = L+ Range * L of the symbol being coded=0.2+0.06*0.5=0.23

H = L + Range * H of the symbol being coded=0.2+0.06*0.6=0.236

Range=0.236-0.23=0.006

Arithmetic encoding:

eaii

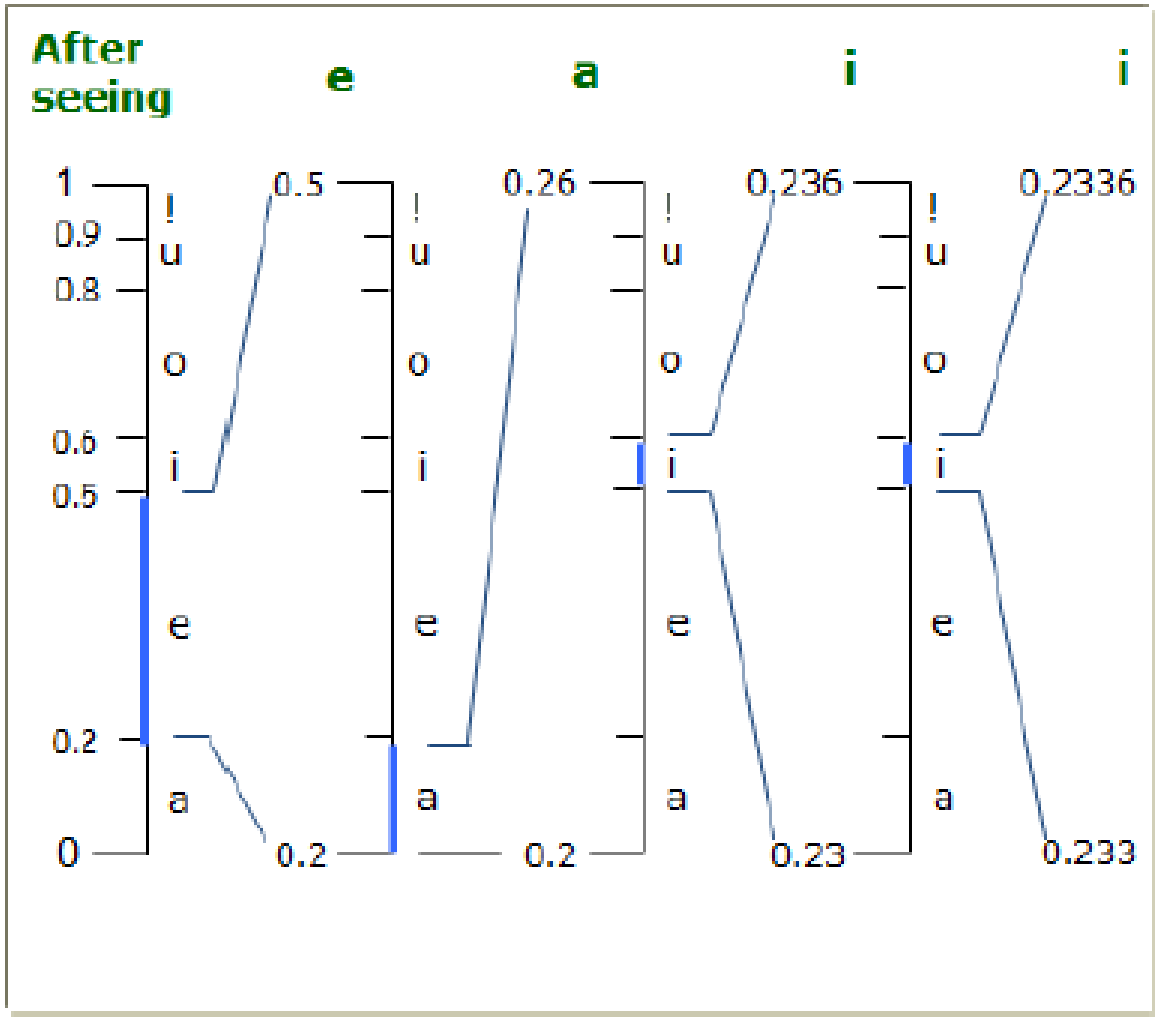
Symbol	Probability	Range
a	.2	[0 , 0.2(
e	.3	[0.2 , 0.5 (
i	.1	[0.5 , 0.6(
o	.2	[0.6 , 0.8(
u	.1	[0.8 , 0.9(
!	.1	[0.9 , 1 (

new

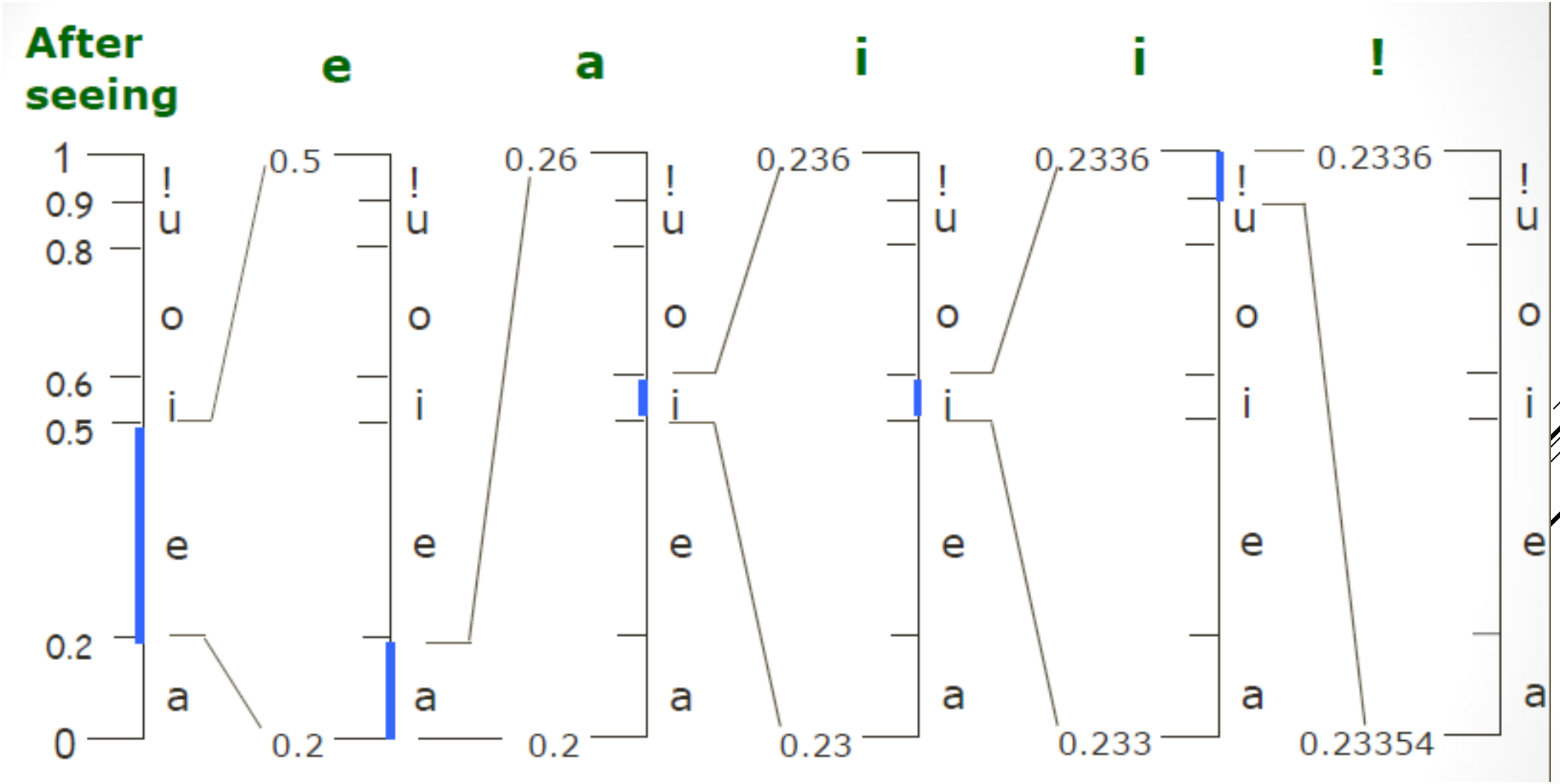
L = 0.233

H = 0.2336

Range=0.0006



Arithmetic encoding:



new

L =0.23354 , H =0.2336 , Range=0.00006

Arithmetic encoding:

```
BEGIN
    low = 0.0;    high = 1.0;    range = 1.0;

    while (symbol != terminator)
        {
          get (symbol);
          low =  low + range * Range_low(symbol);
          high = low + range * Range_high(symbol);
          range = high - low;
        }

    output a code so that low <= code < high;
END
```

Arithmetic encoding:

Symbol	Probability	Range
a	.2	[0 , 0.2(
e	.3	[0.2 , 0.5 (
i	.1	[0.5 , 0.6(
o	.2	[0.6 , 0.8(
u	.1	[0.8 , 0.9(
!	.1	[0.9 , 1 (

Symbol	Low	High	Range
	0	1	1
e	0.2	0.5	0.2
a	0.2	0.26	0.06
i	0.23	0.236	0.006
i	0.233	0.2336	0.0006
!	0.23354	0.2336	0.00006

New low and high range generated

Arithmetic encoding:

Final range= 0.00006

Final range= p(e)*p(a)* p(i)* p(i)* p(!)=0.3*0.2*0.1*0.1*0.1 = 0.00006

Final interval [0.23354 ,0.2336]

we use final interval to Generating Codeword for Encoder (next algorithm)

Arithmetic encoding - Generating Codeword for Encoder:

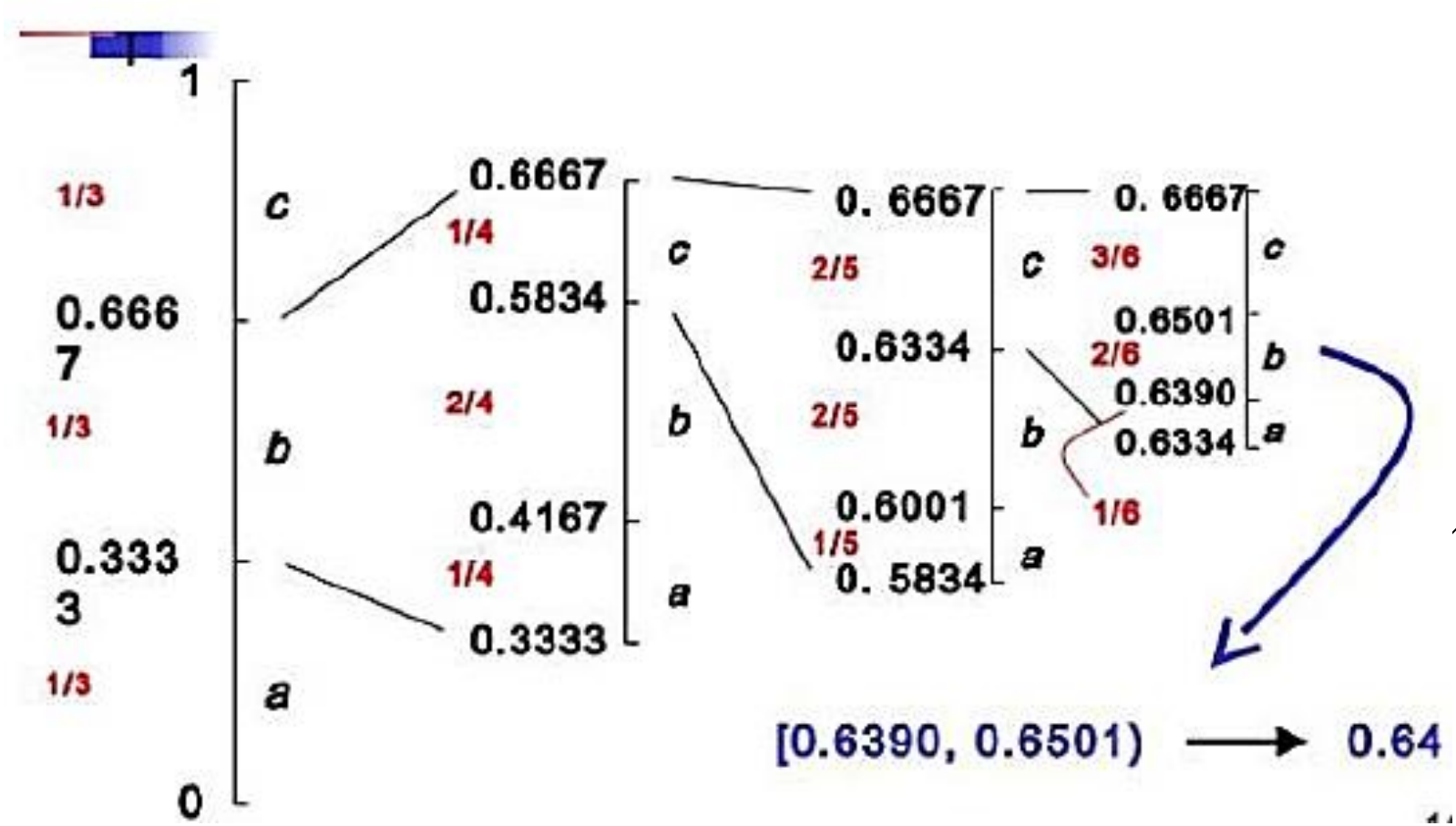
```
BEGIN
code = 0;
k = 1;
while (value(code) < low)
{
assign 1 to the kth binary fraction bit
if (value(code) > high)
replace the kth bit by 0
k = k + 1;
}
END
```

Binary fractional to decimal fractional(2^-1 + 2^-2 + 2^-3 ,….)

The output of arithmetic is stream of bits .number shown as decimal for explain ,convert to fractional binary to send

Arithmetic encoding - Generating Codeword for Encoder:

Example: Generate codeword for this interval: [0.6390,0.6501)



Arithmetic encoding - Generating Codeword for Encoder:

Example: Generate codeword for this interval: [0.6390,0.6501)

1- code=0 k=1

Is value(0) < low ??

0 < 0.639 yes → enter while loop

K1=1

is value(0.1)> 0.6501 ??

1*2^(-1)=0.5 > 0.6501 no → nothing

K=2

K-index	1
	1 ?
0.	1

Arithmetic encoding - Generating Codeword for Encoder:

Example: Generate codeword for this interval: [0.6390,0.6501)

2- code=0.1 k=2

Is value(0.1) < low ??

0.5 < 0.639 yes → enter while loop

K2=1

is value(0.11)> 0.6501 ??

1*2^(-1)+1*2^(-2) > 0.6501

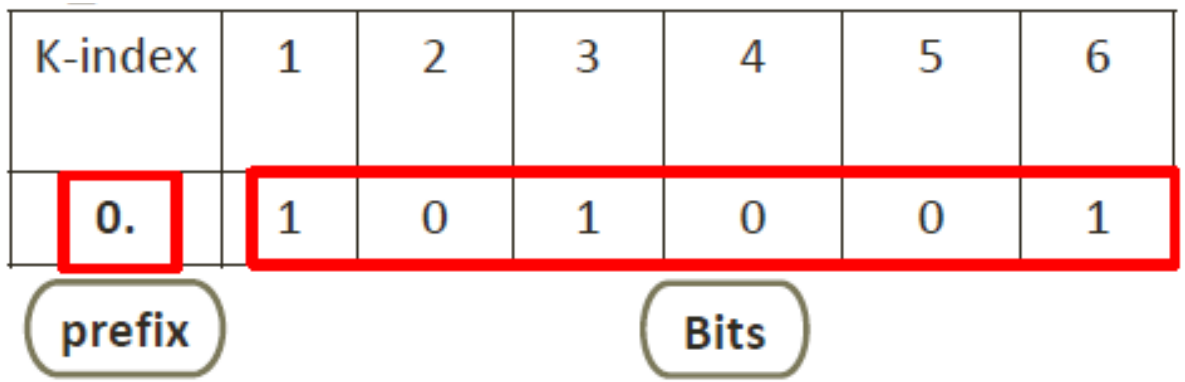
0.75 >0.6501?? Yes → k2=0

K=3

K-index	1	2
	1	1?
0.	1	0

Arithmetic encoding - Generating Codeword for Encoder:

Example: Generate codeword for this interval: [0.6390,0.6501)



Continue example

7- code=0.101001 k=7

Is value(0.101001) < low ??

0.64062 < 0.639 no → enter while loop NO

Stop algorithm

We send 0.64062 as a codeword of "bccb" message

Notice : low< 0.64062 <high

What is the number of bits ??

0.101001 fraction binary → [0. prefix] [101001 =6 bits]

1/2+1/8+1/64= 0.64062 (fraction decimal)

LZW compression encoding:

If you were to take a look at almost any data file on a computer, character by character, you would notice that there are many **recurring patterns**.

LZW is a data compression method that takes advantage of this repetition.

It generally performs best on files with repeated substrings

Where is LZW compression used?

- TIFF files
- GIF files
- PDF files

LZW compression encoding:

(Lempel-Ziv-Welch) LZW is a "dictionary"-based compression algorithm.

This means that instead of tabulating character counts and building trees (as for Huffman encoding), LZW encodes data by referencing a dictionary.

Unlike Huffman coding and arithmetic coding, this coding scheme *does not require a priori knowledge of the probabilities* of the source symbols

Thus, to encode a substring, only a single code number, corresponding to that substring's index in the dictionary, needs to be written to the output file.

LZW compression encoding:

Compression occurs when a single code is output instead of a string of characters ➔

(long message has lots of repetition e.g. text , monochrome images.)

It assigns a fixed length codeword to a variable length of symbols.

The coding starts with an initial dictionary, which is enlarged with the arrival of new symbol sequences

LZW compression encoding:

LZW compression works best for files containing lots of repetitive data. Files that are compressed but that do not contain any repetitive information at all can even grow bigger!

LZW compression is fast.

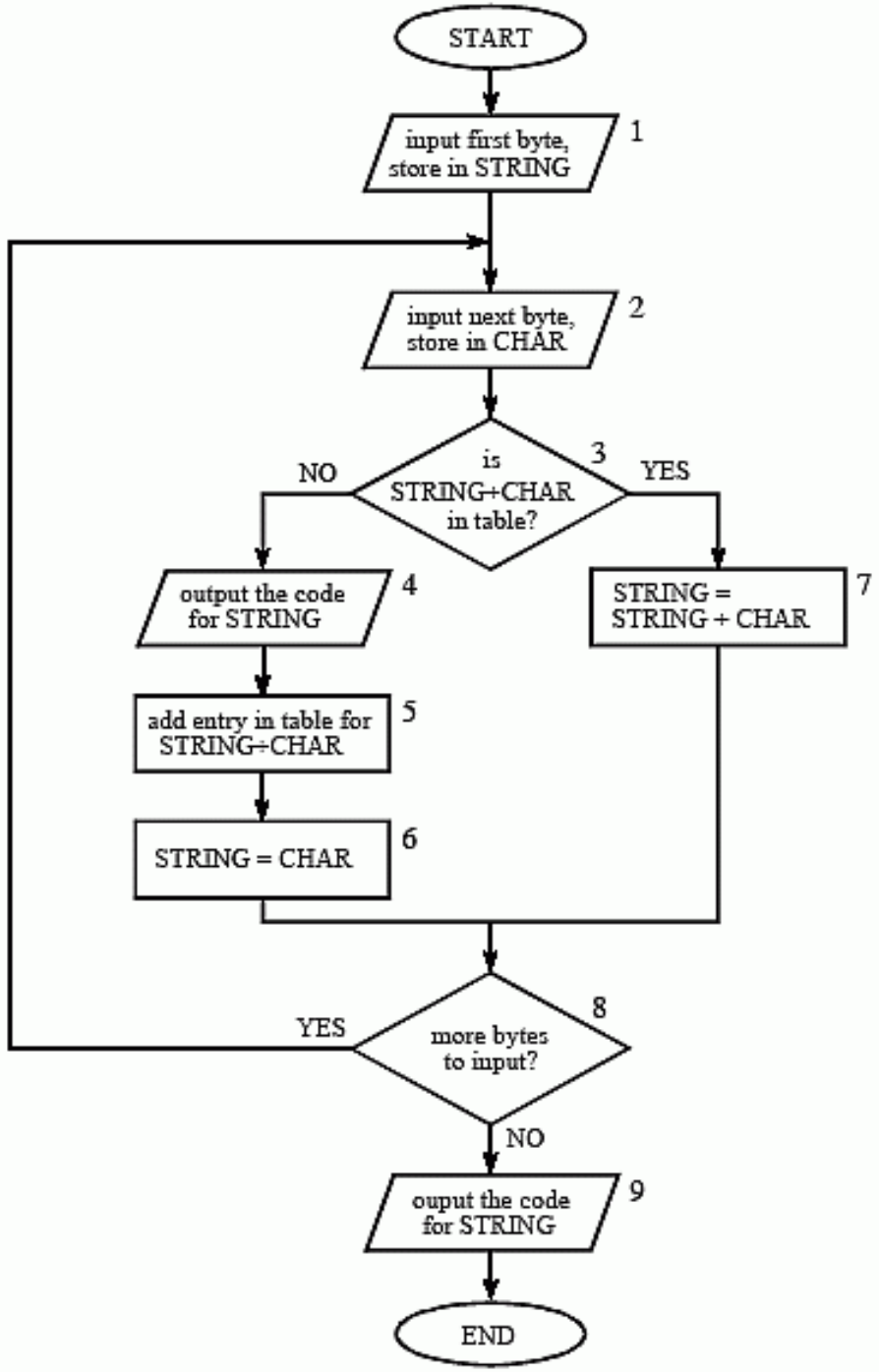
- **Statistical coding** requires two passes over the input (The first pass calculates probabilities, the second encodes the message)
- **dictionary coding** require only one.

LZW is easier to understand because it uses a strategy that programmers are familiar with-> using indexes into databases to retrieve information from large amounts of storage.

LZW compression Algorithm:

- STRING = get input character
- while there are still input characters do
  1. CHAR = get input character
  2. if STRING+CHAR exists is in the table then
     - STRING = STRING+CHAR
  3. else // STRING in table, STRING+CHAR not in table
     - output the code for STRING
     - add STRING+CHAR to the string table
     - STRING = CHAR
- end while
- output the code for STRING // Flush buffer to ouput last code

LZW compression encoding:



LZW compression encoding:

Example:

Input string: **ABABBABCABABBA**

Let's start with a very simple dictionary (also referred to as a string table), initially containing only three characters, with codes as follows:

code	string
1	A
2	B
3	C

LZW compression encoding:

Input string:

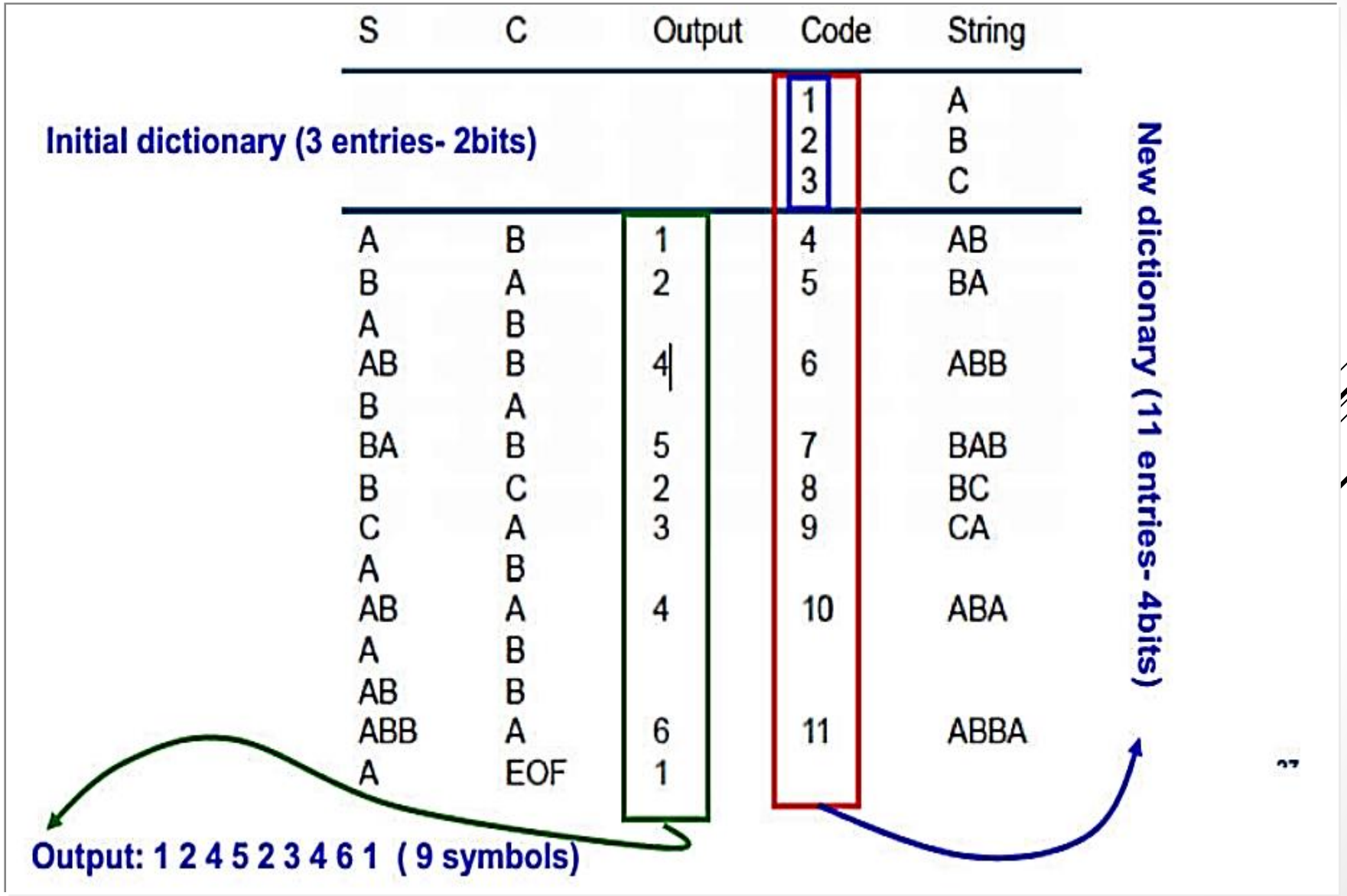
ABABBABCABABBA

String	char	output	code	string+char
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	A	4	10	ABA
A	B			
AB	B			
ABB	A	6	11	ABBA
A	EOF	1		

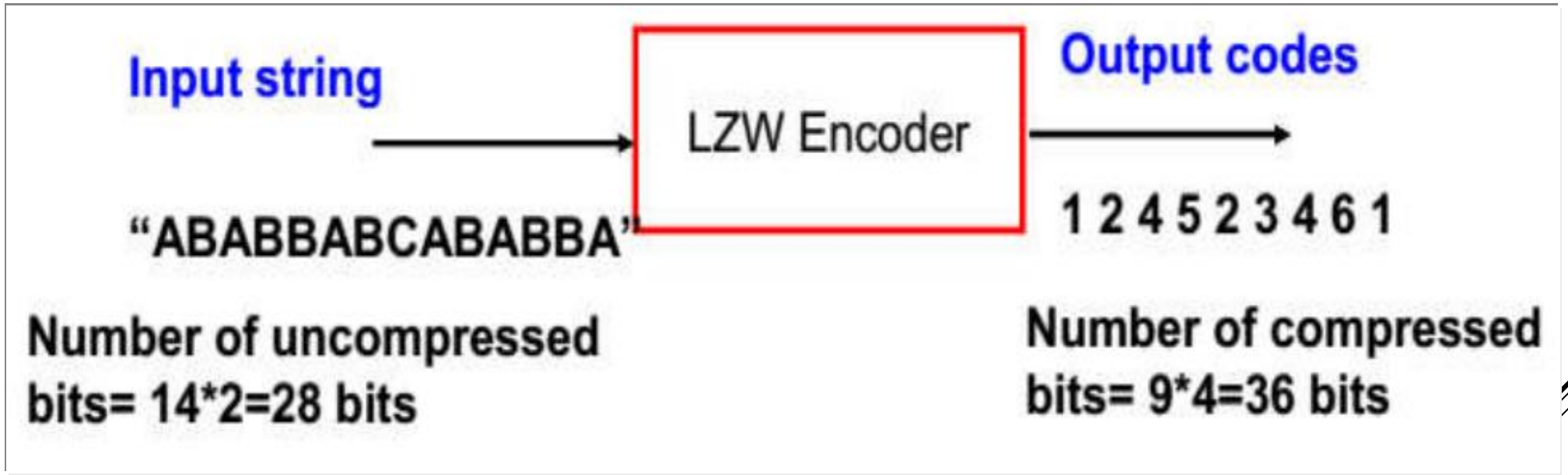
LZW compression encoding:

Input string:

ABABBABCABABBA



LZW compression encoding:



Compression Ratio= 28/36=0.77 message is short

LZW compression encoding:

- There is no need to transmit the dictionary from the encoder to the decoder. A Lempel-Ziv decoder builds an identical dictionary during the decoding process.

- LZW places longer and longer repeated entries into a dictionary, and then emits the code for an element, rather than the string itself, if the element has already been placed in the dictionary.

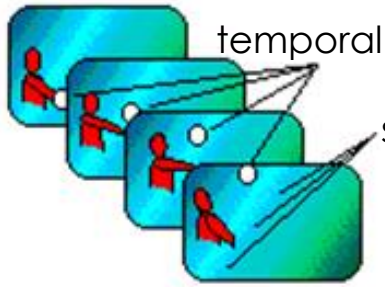
Lossy compression

Why is image compression possible ??

Image compression is largely possible by exploiting various kinds of redundancies which are typically present in an image.

The extent of redundancies may vary from image to image .

Redundancies in images may be categorized as follows –



(a) Statistical Redundancy

Statistical redundancy occurs due to the fact that pixels within an image tend to have very similar intensities as those of its neighborhood, except at the object boundaries or illumination changes.

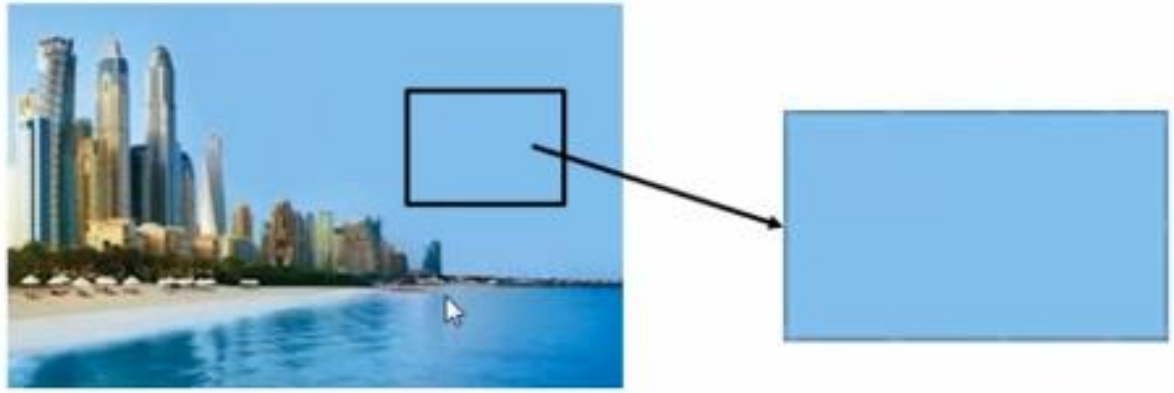
For **still images**, statistical redundancies are essentially spatial in nature.

Video signals exhibit yet another form of statistical redundancy and that is temporal.

For video, intensities of same pixel positions across successive frames tend to be very similar, unless there is large amount of motion present.

Why is image compression possible ??

Spatial Redundancy



Temporal Redundancy

Why is image compression possible ??

(a) Psychovisual Redundancy

- visual perception
- Stem from the fact that the human eye does not respond with equal intensity to all visual information
- The human visual system does not rely on quantitive analysis of individual pixel values when interpreting an image –an observer searches for distinct features and mentally combines them into recognizable groupings
- In this process certain information is relatively less important than other –this information is called Psychovisually redundant
- Psychovisually redundant image information can be removed - quantization →

lossy compression

End of JPEG compression