University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

Data compression
ضغط البيانات

Dr. Abdul Ameer Abuallah
أ.د. عبد الأمير عبد الله
Assistant Lecturer Zainab A. Yakoob
م.م. زينب علي يعكوب

cs.uotechnology.edu.iq

Introduction to Data Compression

1. <u>Previe</u>

Data compression has been ***pushed to the forefront*** of the computer science field. This is largely a result of the **rapid growth in the multimedia** market, and the advent of the **World Wide Web**, which makes the internet easily accessible for everyone.

Data compression addresses the problem of **reducing** the amount of **data required** to represent a digital file, so that it can be **stored or transmitted** so efficiently.

The **principle** of data compression is that, it compresses data by **removing redundancy** from the original data in the source file.

On the other hand**,** ***information theory*** tells us that the **amount of information** conveyed by an event **relates** to its **probability of occurrence**. An event that is less likely to occur is said to contain more information than an event that is more likely to occur. The amount of information of an event and its probability are thus **opposite**.

Amount of information $\downarrow$ probability $\uparrow$ compression $\uparrow$ probability $\uparrow$

It is obvious that information theory is the base theory that data compression rely on.

The problem of representing the source alphabet symbols Si in term of another system of symbols (usually the binary system consisting of the two symbols 0 & 1) is the main topic of ***coding theory***.

An optimum coding scheme will use **more bits** for the symbols that less likely to occur, and a **fewer bits** for the symbols that frequently occur.

Logically speaking, coding theory leads to information theory and information theory provides the bounds on what can be done by suitable encoding of the information. Thus the two theories are intimately related.

Before delivering into the details, we discuss important data compression terms:

2. Data Compression Terminology

Data compression is the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, or the compressed, stream) that has a smaller size. A stream is either a file or a buffer in memory. Data compression is popular because of two reasons:

1. People like to accumulate data and hate to throw anything away. No matter how big a storage device one has, sooner or later it is going to overflow. Data compression seems useful because it delays this inevitability.
2. People hate to wait a long time for data transfers. When sitting at the computer, waiting for a Web page to come in, or for a file to download, we naturally feel that anything longer than a few seconds is a long time to wait.

There are many known methods for data compression. They are based on different ideas, are suitable for different types of data, and produce different results, but they are all based on the **same principle**, namely, they compress data by ***removing redundancy*** from the original data in the source file. Any nonrandom collection data has some structure, and this structure can be exploited to achieve a smaller representation of the data, a representation where no structure is discernible.

The idea of compression by reducing redundancy suggests the **general law** of data compression, which is to "***assign short codes to common events (symbols or phrases) and long codes to rare events.***" There are many ways to implement this law, and an analysis of any compression method shows that, deep inside, it works by obeying the general law.

Type of Redundancy

1. ***Text redundancy*** :
   - In typical English text, for example, the letter E appears very often, while Z is rare. This is called ***alphabetic redundancy***, and suggests assigning variable-size codes to the letters, with E getting the shortest code and Z, the longest one.
   - Another type of redundancy, ***contextual redundancy***, is illustrated by the fact that the letter Q is almost always followed by the letter U (i.e., that certain diagrams and trigrams are more common in plain English than others).
2. ***Images redundancy*** is illustrated by the fact that in a nonrandom image, **adjacent pixels** tend to have **similar colors**.
3. ***Video redundancy*** is illustrated by the fact that in a nonrandom video **consecutive frame** tend to be similar.

The principle of compressing by removing redundancy also answers the following ***question***: "Why is it that an already compressed file cannot be compressed further?" The answer, of course, is that such a file ***has little or no redundancy***, so there is nothing to remove. An example of such a file is random text. When such a file is compressed, there is no redundancy to remove. If we assume that there was a possibility to compress an already compressed file, then successive compressions would reduce the size of the file until it becomes a single byte, or even a single bit. This, of course, is ridiculous since a single byte cannot contain the information present in an arbitrarily large file.)

Data compression is achieved by reducing redundancy, but this also makes the data **less reliable**, more prone to errors. Making data more reliable, on the other hand, is done by adding ***check bits and parity bits***, a process that increases the size of the codes, thereby increasing redundancy. Data compression and ***data reliability*** are thus opposites.

Before delving into the details, we discuss important data compression terms.

• The ***compressor*** or *encoder* is the **program** that compresses the raw data in the input stream and creates an output stream with compressed (low-redundancy) data. The ***decompressor*** or *decoder* converts in the opposite direction.

• The term "***stream***" is used throughout these lectures **instead of** "file". "Stream" is a more general term because the compressed data may be **transmitted directly** to the decoder, instead of being written to a file and saved. Also, the data to be compressed may be **downloaded** from a network instead of being input from a file.

• For the original input stream we use the terms *un encoded*, ***raw data***. The contents of the final, compressed, stream is considered the *encoded* or ***compressed data***. The term ***bit stream*** is also used in the literature to indicate the compressed stream.

3. Type of Data Compression

• A ***non adaptive*** compression method is **rigid** and does **not modify** its operations, its parameters, or its tables in response to the particular data being compressed. Such a method is best used to compress data that is all of a single type. Examples are the Group 3 and Group 4 methods for facsimile compression. They are specifically designed for facsimile compression and would do a poor job compressing any other data. In contrast, an ***adaptive*** method examines the raw data and **modifies** its operations and/or its parameters accordingly. An example is the adaptive Huffman method. Some compression methods use a 2-pass algorithm, where the first pass reads the input stream to collect statistics on the data to be compressed, and the second pass does the actual compressing using parameters set by the first pass. Such a method may be called ***semi adaptive***.

- ***Lossy / lossless compression****:* Certain compression methods are ***lossy.*** They achieve better compression by losing some information. When the compressed stream is decompressed, the result is not identical to the original data stream. Such a method makes sense especially in compressing images, movies, or sounds. If the loss of data is small, we may not be able to tell the difference. In contrast, text files, especially files containing computer programs, may become worthless if even one bit gets modified. Such files should be compressed only by a ***lossless*** compression method, also special purpose images like medical images, forensic images, NASA images are compressed using ***lossless*** compression methods.

- ***Symmetrical*** *compression* is the case where the compressor and decompressor use basically the same algorithm but work in "opposite" directions. Such a method makes sense for general work, where the same number of files is compressed as are decompressed. In an ***asymmetric*** compression method either the compressor or the decompressor may have to work significantly harder (i. e. each one uses a different algorithm).

4. Benefits of data compression

The digital representation of the data usually required a very large number of bits. In many applications, it is important to consider techniques for representing data with fewer bits, while maintaining an acceptable fidelity of data quality.

The main benefits of data compression are the follow :

1. Reducing the **storage** requirement or saving the storage space.
2. Potential **cost** saving associated with sending less data over communication channels (e.g. the **cost of call** is usually depend on its duration).
3. Compression can reduce the probability of **transmission error** occurring since **fewer** characters are transmitted when data is compressed.
4. By converting the original data that is represented by conventional code into a different (compressed) code, compression algorithms may **provide a level of security**.
5. **Reducing** the **time required for transmission** of the total original image by transmitting its compressed version.

5. Compression Performance

Most compression methods are ***physical***. They look only at the bits in the input stream and ignore the meaning of the data items in the input (e.g., the data items may be words, pixels, or sounds). Such a method translates one bit stream into another, shorter, one. The only way to make sense of the output stream (to decode it) is by knowing how it was encoded.

Compression Factor

Data compression involves reducing the size of data file, while retaining necessary information. The reduced file is called the compressed file and is used to reconstruct the original file, resulting in the decompressed file. The original file, before any compression is performed is called the uncompressed file. The ratio of the original, uncompressed file and the compressed file is referred to as the ***compression factor***. The compression factor is denoted by:

$$Compression\ Factor = \frac{size\ of\ the\ input\ stream}{size\ of\ the\ output\ sream}$$

$$= \frac{Uncompressed\ file\ size}{compressed\ file\ size} = \frac{sizeU}{sizeC}$$

It is often written as sizeU : sizeC.

In this case value greater than 1 indicates compression, and values less than 1 imply expansion. This measure seems natural to many people, since the bigger the factor, the better the compression.

Example 1

The original image is 256 × 256 pixel. Single-band (gray scale) 8 bits per pixel. This file size is 65,536 byte (64 k). After compression the image file size is became 6,554 bytes. Compute the compression Factor.

Sol

$$Compression\ Factor = \frac{Uncompressed\ file\ size}{compressed\ file\ size} = \frac{65536}{6554} = 9.9999 = 10$$

This can also be written as 10:1.

This is called "10 to 1 compression" or a "10 times compression" or it can be stated as "compressing the image to 1/10 its original size".

Bit Per Pixel

Another way to state the compression of an image is to use the terminology of ***bit per pixel***. For an N×N image.

$$\text{Bit per pixel} = \frac{Number\ of\ Bits\ (compressed\ file)}{Number\ of\ Pixels\ (\ original\ file\ )} = \frac{(8)\ (Number\ of\ Bytes)}{N \times N}$$

Example 2

Using the preceding example, with a compression factor of 65,536/6,554 bytes, we want to express this as bits per pixel. This is done by first finding the number of pixels in the image = 256×256=65,536 pixels. We then find the number of bits in the compressed image file = (6,554 bytes) (8 bits/bytes) = 52,432 bits. Now we can find the bits per pixel by taking the ratio:

Bit per pixel = $\frac{52,532}{65,536}$ = 0.8 bits/pixel.

The reduction in the file size is necessary to meet the bandwidth requirement for many transmission systems, as well as the storage requirement in computer data bases. The amount of data required for digital images is enormous. For example, a single 512 × 512, 8-bit image required 2,097,152 bits for storage. If we wanted to transmit this image over the World Wide Web, it would probably take minutes for transmission- too long for most people to wait.

Example 3

To transmit an RGB (true color) 512 × 512, 24-bit (8 bit / color) image via modem at 28.8 kbaud (kilobits/second), it would take about :

$$\frac{(512 \times 512 \text{ pixels})\ (24 \text{ bit/pixels})}{(28.8 \times 1024\ bits/second)} = 213 \text{ seconds} = 3.6 \text{ minutes}$$

Example 4

A colored video clip of 4 second duration with a frame size of 160×120 pixels and a frame rate of 30 frames per second, is to be transmitted via modem at 28.8 kbaud (kilobits/second), it would take about

$$\frac{(160 \times 120 \text{ pixels}) \left(24\frac{\text{bit}}{\text{pixels}}\right) (4 \text{ secons})(30 \text{ frame }/\text{second})}{(28.8 \times 1024\ bits/second)} = 1875 \text{ seconds}$$

$$= 31.25 \text{ minutes}$$

The above results show the necessity of data compression especially in images and movies transmission.

Self Information (ideally length of the code)

In ***Information theory*** , the function I of the probability Pi ($I = -\log_2 Pi$ bit) measures the amount of ***uncertainty, surprise, or information*** that the event contains.

If an event of ***low probability*** occurs, it causes ***greater surprise***, and hence ***conveys more information*** than the occurrence of an event of High probability. Thus the information is connected with the element of ***surprise***, which is a ***result of uncertainty***. **The more unexpected the event, the greater the surprise, and hence more information**. Thus the probability of occurrence is related to the information content. If **P** is the probability of occurrence of a message and **I** is the information gained from the message, it is evident that when **$P \to 1$ then $I \to 0$**, on the other hand when **$P \to 0$ then $I \to \infty$**, and in general a ***smaller P gives larger I.***

While in ***data compression*** the function (**$I = -\log_2 Pi$**) measure the **ideally length of the code**, hence, whenever the symbol has a high probability (i.e. frequently occur) it will be assigned a shorter code.

Probability $\downarrow$ **self information** $\uparrow$ **Probability** $\downarrow$ **Code length** $\uparrow$

Average Information -Entropy (ideally average length of the code)

In practice we are interested in the **average** information conveyed H (entropy), than in the specific information of each symbol where:

$H = -\sum_{i=1}^{n} Pi \log Pi$ bits \ symbol

The function H of the probability distribution P_i measures the amount of ***randomness*** (in other words information) the distribution contains. the ***more randomness*** that exist in the data the ***more information*** that data contains.

randomness $\uparrow$ Amount of information $\uparrow$

While in ***data compression*** the function (**H**) measure the **ideally average length of the code**, hence, ***whenever the source symbols have a high entropy those symbols will be assigned a longer code***. Hence, the ***more randomness*** that exist in the data , the ***more bits per pixel*** are required to represent the data and that leads to ***less compression.***

randomness $\uparrow$ average length of the code $\uparrow$ randomness $\uparrow$ compression $\downarrow$

It is obvious that information theory is the base theory that data compression based on.

Entropy هو مقياس لتشتت المعلومات.

كلما يزداد عدد ال Symbols كلما تكون H عالية (عشوائية عالية) وتكون Information عالية.

كلما تكون H عالية (عشوائية عالية) يكون ال Compression أقل.

entropyعالي ← Randomness عالي ← Information عالية ← Compression قليل

$H_{max}(x_i) = \log_2 M$ and this happen only when $P_i = \frac{1}{M}$ for all values of i

<u>Ex</u>

entropy للشخص الذي يعرف كلمتين no و yes (وباحتمالات متساوية) هو

$$H = -(\frac{1}{2}\log\frac{1}{2} + \frac{1}{2}\log\frac{1}{2}) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{2}\log\frac{1}{2}$$

$$= \frac{1}{2}\log 2 + \frac{1}{2}\log 2 = 1$$

entropy للشخص الذي يعرف ثلاث كلمات وباحتمالات متساوية هو

$$H = -(\frac{1}{3}\log\frac{1}{3} + \frac{1}{3}\log\frac{1}{3} + \frac{1}{3}\log\frac{1}{3}) = 1.58$$

entropy للشخص الذي يعرف اربع كلمات وباحتمالات متساوية هو

$$H = -(\frac{1}{4}\log\frac{1}{4} + \frac{1}{4}\log\frac{1}{4} + \frac{1}{4}\log\frac{1}{4} + \frac{1}{4}\log\frac{1}{4}) = 2$$

بينما entropy للشخص الذي يعرف كالمة واحدة فقط مثل كلمة" yes "

H = - 1 log 1 = 0

When calculating the ideally length of the code for the above example , we can see that the length of the code in case of 2 words is equal to - $\log\frac{1}{2}$ = 1 bits , also the length of the code in the in case of 4 words is equal to - $\log\frac{1}{4}$ = 2 bits, while its equal to – log $\frac{1}{8}$ = 3 bits in the case of 8 words. Hence, the ***more the entropy*** of the data , the ***more bits per pixel*** are required to represent the symbols of that data.

Source Coding

The problem of representing the source alphabet symbols S_i in term of another system of symbols (usually the binary system consisting of the two symbols 0 and 1) is the main topic of coding.

The two **main problems** of coding methods are the following:

1. Assigning codes that can be decoded unambiguously (i.e. the coder must provide a one-to-one mapping).
2. Assign codes with the minimum average size..

For the purposes of efficiency. The average code length

$$L = \sum_{i=1}^{n} P_i l_i$$

is minimized, where l_i is the length of the representation of the i_{th} symbol S_i.

the entropy function provides a lower bound on L ($L \geq H(x)$).

Source Code Efficiency

L = average length of the code

L = $\sum_{i=1}^{n} Pi\ \ li$ bits/symbol.

$$\xi_{code} = \frac{H(x)}{L} * 100\% \quad where\ \xi_{code} = \text{code Efficiency}$$

اي ان كل source *تقاس كفائته* ξ *بمقدار أقتراب* L *من ال* H(x) *((اي من الحالة المثالية لان* H *هو معدل* I *وان* I *هو ال* Ideally Length *للرمز)).*

ان غايتنا هي L تقترب من H ولكن عمليا $H(x) \leq L$.

<u>Redundancy of the Code</u>

Consider the four symbols *a1, a2, a3,* and *a4*. If they appear in our data strings with equal probabilities (= 0.25), then the entropy of the data is :

$H = -4\,(0.25 \log_2 0.25) = 2$ bit/symbol.

<u>Or</u> directly , from the theorem :

$H = \log_2 M = \log_2 4 = 2$ bits/symbol , because H reach it's maximum value when the symbols have equal probabilities.

Two is the smallest number of bits needed on the average to represent each symbol in this case. We can simply assign our symbols the four 2-bit codes 00, 01, 10, and 11. Since the probabilities are equal, The average length of the code is:

$L = \lceil \log_2 M \rceil = \lceil \log_2 4 \rceil = 2$.

the redundancy is :

$R = L - H = 2 - 2 = 0$.

Hence, ***the data cannot be compressed below 2 bits/symbol.***

Next, consider the case where the four symbols occur with different probabilities as shown in Table 1 , where *a1* appears in the data (on average) about half the time, *a2* and *a3* have equal probabilities, and *a4* is rare.

Symbol	Prob.	Code1
a1	0.49	1
a2	0.25	01
a3	0.25	001
a4	0.01	000

Table 1 : Variable-Size Codes.

In this case, the data has entropy :

$$H = -(0.49 \log_2 0.49 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.01 \log_2 0.01)$$
$$= -(-0.05 - 0.5 - 0.5 - 0.066) = 1.57 \text{ bits/symbol.}$$

The smallest number of bits needed, on average, to represent each symbol has dropped to 1.57 (i.e. the ideally length of the code is H = 1.57).

If we again assign our symbols the four 2-bit codes 00, 01, 10, and 11, the redundancy would be :

$$R = L - H$$
$$= \lceil \log_2 4 \rceil - 1.57 = 2 - 1.57 = 0.43 \text{ bits / symbol.}$$

That means , ***the data can be compressed below 2 bits/symbol.***

This suggests assigning *variable size codes* to the symbols. Code1 of Table 1 is designed such that the most common symbol, *a1*, is assigned the shortest code. When long data strings are transmitted using Code1, the average size of the code (the number of bits per symbol) is :

$L = \sum_{i=1}^{n} Pi\ li$

$= 1 \times 0.49 + 2 \times 0.25 + 3 \times 0.25 + 3 \times 0.01 = 1.77$ bits/symbol.

Which is very close to the minimum. The redundancy in this case is

$R = L - H = 1.77 - 1.57 = 0.2$ bits per symbol.

Source Coding Techniques

Variable length code

هذه التقنية تعنى بوجود الاحتمالية للرموز symbols حيث ان كل رمز symbols يخصص له code مختلف في الطول عن باقي ال symbols وذلك اعتمادا على أحتمالية ذلك ال symbols حيث ان ال symbols ذا الاحتمالية العالية يعطي code ذا طول قليل وبالعكس فان ال symbol ذا الاحتمالية القليلة يعطي code ذا طول كبير.

$$L = \sum_{i=1}^{M} P_i l_i \quad (bit/symbol)$$

However, variable length code bring with them a fundamental ***problem***, at the receiving end, how do you recognize each symbol of the code? In, for example, a binary system how do you recognize the end of one code word and the beginning of the next ?

If the ***probabilities*** of the frequencies of occurrence of the individual symbols are ***sufficiently different*** , then ***variable length encoding*** can be significantly ***more efficient*** than fixed –length encoding

$P_i \uparrow \Longrightarrow l_i \downarrow$

اي كلما تزيد عدد مرات تكرار ال symbol في ال text يجب ان يقل طول ال code المخصص لذلك ال symbol والعكس صحيح

<u>ملاحظة</u> يكون ال variable length code افضل **بكثير** من fixed length code عندما تكون احتماليات تكرار الرموز symbol مختلفة بصورة كبيرة .

Shannon – Fano method

To encode a message using Shannon-Fano method, you can follow the below steps :

1. Sort the symbols in descending order according to their probabilities.
2. Divide the list of symbols into two parts : upper and lower, so that the summation of the probabilities of the upper part is equal ***as possible*** to the summation of the lower part symbols.
3. Assign "0" code to each of the upper part symbols, and "1" code to each of the lower part symbols.
4. Divide each of the upper and lower part into upper and lower subdivision as in step (2) above, and assign the code "0" and "1" as in step (3) above.
5. Continue in step(4) until each subdivision contains only one symbols.

Note

The Shannon-fano method is ***easy*** to implement, but the code is produces is generally ***not good*** as that produced by the Huffman method, described in the next section.

Ex1

A source produce 5 independent symbols (x_1, x_2, x_3, x_4, x_5) with its corresponding probabilities 0.1, 0.3, 0.15, 0.25, 0.2 . design a binary code for the above source symbol using Shannon – fanon method .

Sol

symbols	P_i	code	l_i
x_2	0.3	0 0	2
x_4	0.25	0 1	2
x_5	0.2	1 0	2
x_3	0.15	1 1 0	3
x_1	0.1	1 1 1	3

$$L = \sum P_i\, l_i = 2*0.3 + 2*0.25 + 2*0.2 + 3*0.15 + 3*0.1$$
$$= 2.25 \text{ Bits\\symbol}$$

The entropy (the smallest number of bits needed, on average, to represent each symbol) is

$H = - \sum P_i \log P_i$

$= - (0.3 \log 0.3 + 0.25 \log 0.25 + 0.2 \log 0.2 + 0.15 \log 0.15 + 0.1 \log 0.1)$

$= 2.228$ Bits/symbol

$R = L – H = 2.25 - 2.228 = 0.022$

$\xi_{code} = \frac{H(x)}{L} * 100 \% = \frac{2.228}{2.25} * 100 \% = 99 \%$

Note

If we use fixed length coding

$L = \lceil \log_2 5 \rceil = \lceil 2.3219 \rceil = 3$

$R = L – H = 3 - 2.228 = 0.772$

$\xi_{code} = \frac{H(x)}{L} * 100 \% = \frac{2.228}{3} * 100 \% = 74 \%$

.: coding in Shannon –fanon is more efficient than coding in fixed length coding .

Ex2 :

A source produce 5 independent symbols (x_1, x_2, x_3, x_4, x_5) with its corresponding probabilities 0.1, 0.05, 0.25, 0.5, 0.1. design a binary code for the above source symbol using Shannon – fanon method .

Sol

symbols	P_i	code	l_i
x_4	0.5	0	1
x_3	0.25	1 0	2
x_1	0.1	1 1 0	3
x_5	0.1	1 1 1 0	4
x_2	0.05	1 1 1 1	4

$L = \sum P_i l_i = 1*0.5 + 2*0.25 + 3* 0.1 + 4* 0.1 + 4* 0.05$

$= 1.9$ Bits / symbol

The entropy (the smallest number of bits needed, on average, to represent each symbol) is

$H = - \sum P_i \log P_i$

$= - (0.5 \log 0.5 + 0.25 \log 0.25 + 0.1 \log 0.1 + 0.1 \log 0.1 + 0.05 \log 0.05)$

= 1.88 Bits /symbol

R = L – H = 1.9 – 1.88 = 0.02

$\xi_{\text{code}} = \frac{H(x)}{L} * 100\ \%$.

$= \frac{1.88}{1.9} * 100\ \% = 99\ \%$

Ex3 :

A source produce 5 independent symbols (x_1, x_2, x_3, x_4, x_5) with its corresponding probabilities 0.1, 0.35, 0.3, 0.05, 0.2. design a binary code for the above source symbol using Shannon – fanon method .

Sol

symbols	P_i	code	l_i
x_2	0.35	0	1
x_3	0.3	1 0	2
x_5	0.2	1 1 0	3
x_1	0.1	1 1 1 0	4
x_4	0.05	1 1 1 1	4

$L = \sum P_i\, l_i = 1*0.35 + 2*0.3 + 3*0.2 + 4*0.1 + 4*0.05$
$= 2.15$ Bit / symbol

The entropy (the smallest number of bits needed, on average, to represent each symbol) is

$H = -\sum P_i \log P_i$
$= -(\,0.35 \log 0.35 + 0.3 \log 0.3 + 0.2 \log 0.2 + 0.1 \log 0.1 + 0.05 \log 0.05\,)$
$= 2.062$ Bit/symbol

R = L – H = 2.15 - 2.062 = 0.088

$\xi_{\text{code}} = \frac{H(x)}{L} * 100\ \%$.

$= \frac{2.062}{2.15} * 100\ \% = 96\ \%$

Sol 2

symbols	P_i	code	l_i
x_2	0.35	0 0	2

x_3	0.3	0 1	2
x_5	0.2	1 0	2
x_1	0.1	1 1 0	3
x_4	0.05	1 1 1	3

$L = \sum P_i l_i = 1*0.35 + 2*0.3 + 2* 0.2 + 3* 0.1 + 3* 0.05$
$= 2.15$ Bit / symbol

Ex4 :

A source produce 7 independent symbols (x_1, x_2, …. , x_7) with its corresponding probabilities 0.10, 0.15, 0.10, 0.05, 0.25, 0.20, 015. design a binary code for the above source symbol using Shannon – fanon method .

Sol

Symbols	Prob.	Code	li
X5	0.25	00	2
X6	0.20	01	2
X2	0.15	100	3
X7	0.15	101	3
X1	0.10	110	3
X3	0.10	1110	4
X4	0.05	1111	4

The average size of this code is
$L = 0.25 \times 2 + 0.20 \times 2 + 0.15 \times 3 + 0.15 \times 3 + 0.10 \times 3 + 0.10 \times 4 + 0.05 \times 4$
$= 2.7$ bits/symbol.

The entropy (the smallest number of bits needed, on average, to represent each symbol) is
$H = -(0.2510g_2\, 0.25 + 0.2010g_2\, 0.20 + 0.1510g_2\, 0.15 + 0.151og_2\, 0.15$
$+ 0.10 \log_2 0.10 + 0.10 \log_2 0.10 + 0.0510g_2\, 0.05)$
$=$ 2.67 bits/ symbols.

$R = L – H = 2.7 - 2.67 = 0.03$

$\xi_{code} = \frac{H(x)}{L} * 100\ \%$.

$= \frac{2.67}{2.7} * 100\ \% = 99\ \%$

Huffman Coding method

Huffman makes the average number of binary digits per message nearly equal the Entropy (average bits of information per message). To encode a message using Huffman method, a tree must has to be constructed , with symbol at every leaf, from bottom to up. This is done by following the below steps:

1) Sort the symbols in descending order according to their probabilities .
2) Assign "0" and "1" code for the two symbols with the smallest probabilities .
3) Combine those two symbols in step (2) to construct a new symbol with probability equal to the summation of the two probabilities , then enter the new symbol in the list at a new position appropriate to its new probability .
4) Repeat step (2 and 3) until the list has only one symbol.
5) The code word of any symbol may be obtained by following the series of binary codes (0_s, 1_s) which has been assigned to that symbol.

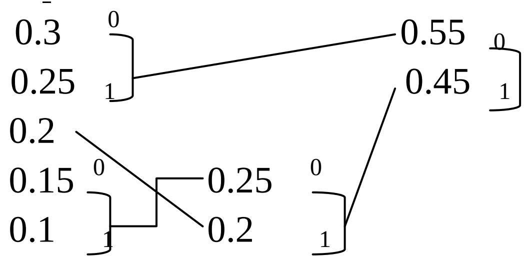
Note

It can be shown that the size of the Huffman code of a symbol a_i with probability Pi is always less than or equal to ⌈ -log Pi ⌉ (the ideally length of the code) .

Ex1

Design a binary code for the below source symbol using Huffman method.

X_i	P_i	code	l_i
x_1	0.3	00	2
x_2	0.25	01	2
x_3	0.2	11	2
x_4	0.15	100	3
x_5	0.1	101	3

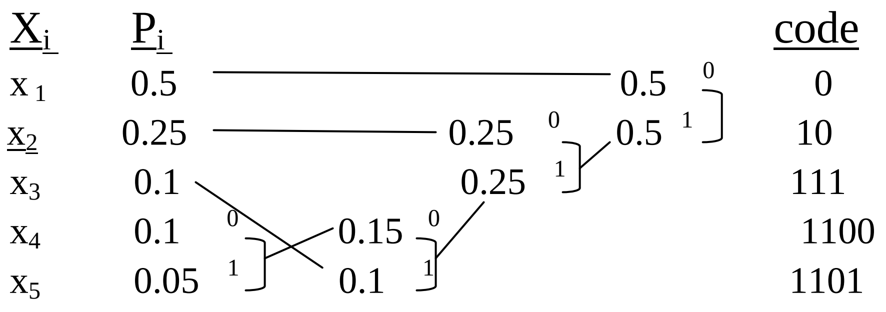


$L=\sum l_i\, p_i = 2.25$ bits\symbol

كفائة ال source هنا هي نفس الكفاءه اذا ما عملنا code بطريقة ال Shannon –fano

Ex2

Design a binary code for the below source using Huffman method .



X_i	P_i	code
x_1	0.5	0
x_2	0.25	10
x_3	0.1	111
x_4	0.1	1100
x_5	0.05	1101

$L=\sum l_i\, p_i= 2.25$

$=1.9$ bits\symbol

ملاحظة دائما يكون

$L_{huff} \le L_{shann}$

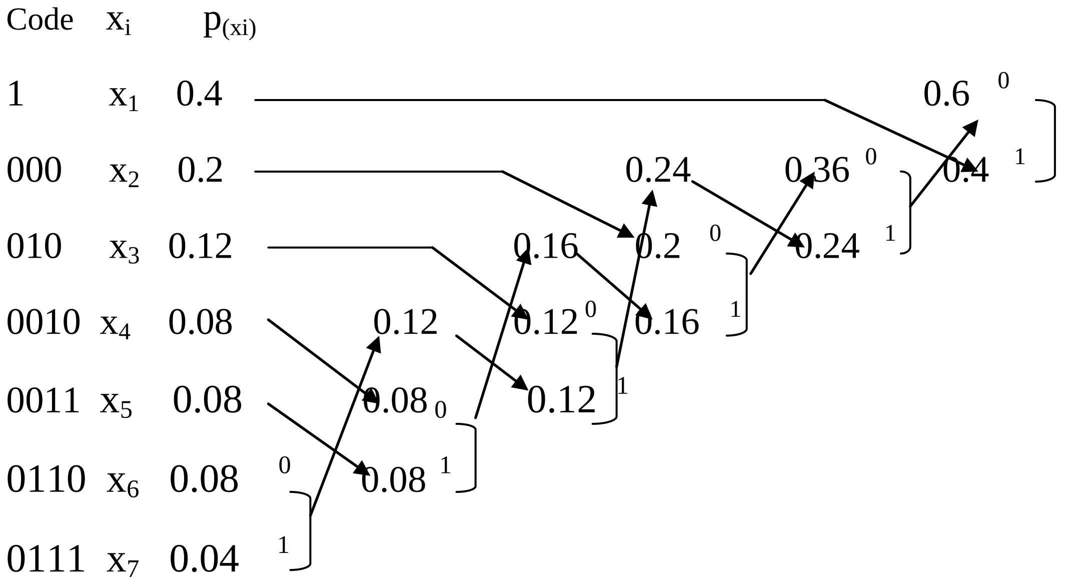
وان هناك حدود لل L هي

$H_{(x)} \le L \le H_{(x)+1}$

هذه الحدود تنطبق على ال variable length ولاتنطبق على fixed length حيث قد يكون الفرق في ال fixed length اكبر من ١ بكثير

Ex3

Design a binary code for the below source symbol using Huffman method



Code	x_i	$p_{(xi)}$
1	x_1	0.4
000	x_2	0.2
010	x_3	0.12
0010	x_4	0.08
0011	x_5	0.08
0110	x_6	0.08
0111	x_7	0.04

$L=\sum P_i\, l_i = 2.48$ bits/symbol

$H_{(x)}$=2.419 bits/symbol

Ƶcode $=\frac{2.419}{2.45} * 100\ \% = 97.54\ \%$

For Shannon –fanon

L= 2.52 bit/symbol

Ƶcode $=\frac{2.419}{2.52} * 100\ \% = 95.99\ \%$

Ex4

A source produce 7 independent symbols (x_1, x_2, x_3, x_4, x_5, x_6, x_7) with its corresponding probabilities 0.23, 0.17, 0.20 , 0.15, 0.08, 0.05, 0.12 design a binary code for the above source symbol using **Huffman** method.

Sol

Symbols	prob.	Code
X_1	0.23	10
X_3	0.20	11
X_2	0.17	000
X_4	0.15	001
X_6	0.12	011
X_5	0.08	0100
X4	0.05	0101

Extension of code

when we extend a source with order of **n** , we have the following equation for L and H

$L=\frac{Ln}{n}$

$H_{(x)} = \frac{Hn(x)}{n}$ where n is the order of extension

We also have the important relation for coding a source ,

$H_{(x)} \leq L \leq H_{(x)+1}$

Now , if we take the nth extension of the code the above relation is also applied , so

$H_{n(x)} \leq L_n \leq H_{n(x)+1}$

Where L_n is the average code length ,for the extended source

Since $L_n = n L$

And $H_{n(x)} = nH_{(x)}$

Then

$nH(x) \leq nL \leq n\, H_{(x)+1}$

بالقسم على n نحصل على القانون العام

$$\boxed{H_{(x)} \leq L \leq H_{(x)} + \frac{1}{n}}$$

المعادلة اعلاه تبين انه كلما وسعنا ال source output (اي كلما عملنا extension بـ order اكبر) يكون افضل لانة L ستقترب من H وهذه يؤدي الى

$$\xi_{code} \Longrightarrow 100\%$$

<u>Ex</u>

A binary source produce two symbols x_1 ,x_2 with probabilities $p(x_1) = 0.8$, $p(x_2) = 0.2$. find ξ for the 1st , 2nd , 3rd extension of binary code for the above source .

<u>Sol</u>

(1) n=1 , coding with extension

X_i	P_i	code	l_i
x_1	0.8	0	1
x_2	0.2	1	1

L=1 Bits /symbol

$H_{(x)} = - \sum P_i \log P_i$

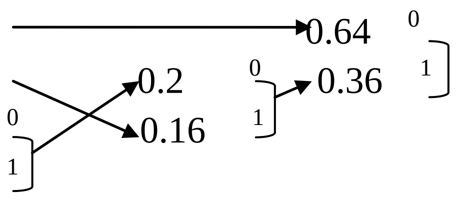
$= -(0.8 \log 0.8 + 0.2 \log 0.2)$

$=0.72$ Bits/symbol

$$\xi_{\text{code}} = \frac{H(x)}{L} * 100\% = \frac{0.72}{1} * 100\% = 72\ \%$$

(2) n=2 , coding with extension of order 2

Symbol	P_i	code	l_i
x_1x_1	0.64	0	1
x_1x_2	0.16	11	2
x_2x_1	0.16	100	3
x_2x_2	0.04	101	3



$L_2 = \sum l_i\ P_i$

$= 1*0.64 + 2*0.16 + 3*0.16 + 3*0.04 = 1.56$ bits\symbol

$L = \frac{L2}{2} = \frac{1.56}{2} = 0.78$ bits / symbol

$H_{2\,(x)} = -(0.64 \log 0.64 + 2*0.16 \log 0.16 + 0.04 \log 0.04)$

$= 1.44$

$H_{(x)} = \frac{H2(x)}{2} = \frac{1.44}{2} = 0.72$ bits/symbol

$$\xi_{\text{code}} = \frac{H(x)}{L} * 100\% = \frac{0.72}{0.78} * 100\ \% = 94\ \%$$

<u>ملاحظة</u> لاحظ ان سبب نقصان L هو انه اصبح لدينا رمزان هما x_1x_1 يرمزان ب bit واحد هو 1 وان احتمالية ورودها كبيرة وهي 0.64

(3) n= 3 ,coding with extension of order 3

Symbol	Prob.	code	l_i
$x_1x_1x_1$	0.512	0	1
$x_1x_1x_2$	0.128	1 0 0	3
$x_1x_2x_1$	0.128	1 0 1	3
$x_2x_1x_1$	0.128	1 1 0	3
$x_1x_2x_2$	0.032	1 1 1 0 0	5
$x_2x_1x_2$	0.032	1 1 1 0 1	5
$x_2x_2x_1$	0.032	1 1 1 1 0	5

$x_2x_2x_2$	0.008	1	1	1	1	1	5

ملاحظة: من الأكفاء اجراء عملية ال coding بأستخدام طريقة Huffman ولكن تم استخدام طريقة Shannon – fano للسهولة .

$L_3 = \sum P_i l_i = 2.184$

$L = \frac{L3}{3} = \frac{2.184}{3} = 0.728$ bits / symbol

$H_{3(x)} = -\sum P_i \log P_i = 2.16$

$H_{(x)} = \frac{H3(x)}{3} = \frac{2.16}{3} = 0.72$ bits/symbol

$\xi_{code} = \frac{H(x)}{L} * 100\% = \frac{0.72}{0.728} * 100\% = 98.9\%$

لاحظ ان الكفاءة زادت من ٧٢% الى ٩٤% عندما عملنا extensionبمقدار ٢ وزادت الى ٩٨,٩% عندما عملنا extension بمقدار ٣.

Prefix Code

prefix property: This property requires that once a ***certain bit pattern*** has been assigned as the code of a symbol, ***no other codes*** should start with that pattern (the pattern cannot be the ***prefix*** of any other code).

Hence, a ***prefix code*** is a variable-size code that satisfies the prefix property.

As we Know earlier, designing variable-size codes is done by following two principles:

(1) Assign short codes to the more frequent symbols and ,

(2) obey the prefix property.

Following these principles produces short, unambiguous codes, ***but not necessarily the best*** (i.e. shortest) ones.

Binary Representation of the Integers

The binary representation of the integers has two main ***disadvantages*** :

1. does not satisfy the prefix property.
2. In this representation the size ***n*** of the set of integers ***has to be known in advance***, since it determines the code size, which is $1 + \lfloor \log_2 n \rfloor$.

For example the size of the code of the integer is

Code size $(3) = 1 + \lfloor \log_2 3 \rfloor = 1 + \lfloor 1.585 \rfloor = 1+1 = 2$

Code size $(5) = 1 + \lfloor \log_2 5 \rfloor = 1 + \lfloor 2.322 \rfloor = 1+2 = 3$

Code size $(8) = 1 + \lfloor \log_2 8 \rfloor = 1 + \lfloor 3 \rfloor = 1+3 = 4$

Code size $(19) = 1 + \lfloor \log_2 19 \rfloor = 1 + \lfloor 4.24 \rfloor = 1+4 = 5$ its code is 10011

Code size $(37) = 1 + \lfloor \log_2 37 \rfloor = 1 + \lfloor 5.2 \rfloor = 1+5 = 6$ its code is 100101

Integer	binary representation	Code length
0	0	1
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
6	110	3
7	111	3
8	1000	4
9	1001	4

Table (1) : The Binary Representation of the Integers

Note : the ambiguity will happen between 1 and all the other numbers (except 0).

Ex:
Encoding this string 5,3,1,6 with the codes of table 1 yields *:*
101111110
Logically specking, the decoder does not know the size ***n*** of the set of integers (***which has to be known in advance*** e.g. ASCII code) , hence The decoder does not know whether to decode the string as 1|0|1|1|1|1|1|1|0 , which is 1,0,1,1,1,1,1,1,0 ; or as 10|11|11|11|0 , which is 2,3,3,3,0 *;* or as 101|111|110 , which is.. 5,7,6 and so on. Code of table 1 is thus ***ambiguous***.The decoder has to follow a constructed Binary tree).

In some applications, a prefix code is required to code a set of integers ***whose size is not known in advance***. Several such codes, most of which are presented later.

The Unary Code

The *unary code* of the nonnegative integer *n* is defined as *n* - 1 ones followed by a single 0 (Table 2). The length of the unary code for the integer *n* is thus *n* bits.

Integer	unary Code	Code length
1	0	1
2	10	2
3	110	3
4	1110	4
5	11110	5
6	111110	6
7	1111110	7
8	11111110	8
9	111111110	9

Table (2) : The Unary Code

Ex:
Assume that a number of integers are encoded into 1101111110010 using the unary code, decode this codeword.
Sol
The decoder starts at the root, reads the first bit "1", since this bit is not "0", the decoder precede foreword, the second bit "1" , also the decoder precede foreword, since the third bit is "0" ,it make the decode stop, and emits the integer

"3" which is the number of bits it has been read. It again returns to the root, reads the first bit "1", and so on until it read a bit "0", and emits the integer "7" which is the number of bits it has been read. It again returns to the root, reads the first bit "0", which force the decoder to stop and emit the integer "1", which is the number of bits it has been read. It again returns to the root, reads "1" , moves foreword , read "0" which force the decoder to stop and emit the integer "2", which is the number of integer it has been read.

Note

Its obvious that the unary code produces, unambiguous codes, ***but it's not necessarily the best ones*** (i.e. very long code).

Other Prefix Codes

Four more prefix codes are described in this section. We use $B(n)$ to denote the binary representation of integer n. Thus $|B(n)|$ is the length, in bits, of this representation. We also use $B'(n)$ to denote $B(n)$ without its most significant bit (which is always 1).

Code C1 is made of two parts. To code the positive integer n we first generate the unary code of $|B(n)|$ (the size of the binary representation of n), then append $B'(n)$ to it. An example is $n = 16 = 10000_2$. The size of B(16) is 5, so we start with the unary code 11110 and append $B'(16) = 0000$. The complete code is thus 11110|0000. Another example is $n = 5 = 101_2$, The size of B(5) is 3, so we start with the unary code 110 and append $B'(5) = 01$ the complete code is 110|01.the length of C1(n) is $2\lfloor \log_2 n \rfloor + 1$ bits.

Code $C2$ is a rearrangement of C1 where each of the $1 + \lfloor \log_2 n \rfloor$ bits of the first part (the unary code) of C1 is followed by one of the bits of the second part. Thus code $C2\ (16) = 101010100$ and $C2\ (5) = 10110$.

Code $C3$ starts with $|B(n)|$ coded in $C2$, followed by $B'(n)$. Thus 16 is coded as $C2\ (5) = 10110$ followed by $B'(16) = 0000$. The complete C3 (16) code is thus 10110|0000

Code $C4$ consists of several parts. We start with $B(n)$. To the left of this we write the binary representation of $|B(n)| - 1$ (the length of n, minus 1). This ***continues recursively, until a 2-bit length number is written***. A zero is then added to the right of the entire number, to make it decodable. To encode 16, we start with 10000, add $|B(16)| - 1 = 4 = 100_2$ to the left, then $|B(4)| - 1 = 2 = 10_2$ to the left of that and finally, a zero on the right. The result is 10|100|10000|0. To encode 5, we start with 101, add $|B(5)| - 1 = 2 = 10_2$ to the left, and a zero on the right. The result is 10|101|0.

Generating the Four prefix codes can be **summarized** in the below steps :
B(n) : is the binary representation of integer *n*.
B'(n) : denote *B(n)* without its most significant bit (which is always 1).
|***B(n)***| : is the length, in bits, of B(n).

1. C1(n) = U(|*B(n)*|) | *B'(n)*.
2. C2(n) = rearrangement (alternative representation) of C1.
3. C3(n) = C2(|*B(n)*|) | *B'(n)*.
4. C4(n) = [|*B(n)*| - 1] | *B(n)* | 0. Where [ ] denote the Binary representation.

<u>Ex1:</u>
Find the four other prefix codes for n = 13.
B(13) : 1101.
B'(13) : 101.
|***B(13)***| : 4.

1. C1(n) = U(|*B(n)*|) | *B'(n)*.
   C1(13) = U (4) | *B'(13)*.
   = 1110|101.

2. C2(13) = 1110110.

3. C3(n) = C2(|*B(n)*|) | *B'(n)*.
   C3(13) = C2(4) | *B'(13)*.

   B(4) : 100.
   B'(4) : 00.
   |***B(4)***| : 3.

   C1(4) = U(3) | *B'(4)*.
   = 110| 00.
   C2(4) = 10100.

   C3(13) = 10100|101.

4. C4(n) = [|*B(n)*| - 1] | *B(n)* | 0.
   C4(13) = [|*B(13)*| - 1] | *B(13)* | 0.
   = [4 - 1] | 1101 | 0.
   = 11|1101|0.

Step 4 is ***continues recursively, until a 2-bit length number is written***.

Ex2:

Find the four other prefix codes for n = 9.

B(9) : 1001.

B'(9) : 001.

|*B(9)*| : 4.

1. C1(n) = U(|*B(n)*|) | *B'(n)*.
   C1(9) = U (4) | *B'(9)*.
   = 1110|001.

2. C2(9) = 1010110.

3. C3(n) = C2(|*B(n)*|) | *B'(n)*.
   C3(9) = C2(4) | *B'(9)*.

   C2(4) = 10100. From Ex1.

   C3(9) = 10100|001.

4. C4(n) = [|*B(n)*| - 1] | *B(n)* | 0.
   C4(9) = [|*B(9)*| - 1] | *B(9)* | 0.
   = [4 - 1] | 1001 | 0.
   = 11|1001|0.

General Prefix Code

More prefix codes for the ***positive integers***, appropriate for special applications, may be designed by the following ***general approach***. Select positive integers *Vi* and combine them in a list *V* (which may be finite or infinite according to needs).

$V = [\ V_1 \quad V_2 \quad V_3 \ldots \quad 2^{i-1} \ldots . V_k]$

The code of the positive integer *n* is prepared in the three following steps:

1. Find *k* such that

$$\sum_{i=1}^{k-1} Vi < n \le \sum_{i=1}^{k} Vi$$

2. Compute the difference

$$d = \text{n} - \sum_{i=1}^{k-1} Vi - 1$$

.

dmax can be written in $\lceil \log_2 V_k \rceil$ bits. Hence, The number *d* is encoded, using the standard binary code, with $\lceil \log_2 V_k \rceil$ number of bits.

3. Encode *n* in two parts. Start with *k* encoded in some prefix code (here we use unary code for simplicity) , and concatenate the binary code of *d*.
Code = $U(k)|d_B$

Ex1

Encode the integer n = 10 , using the general prefix code.

The infinite sequence $V = [\ 1\ \ 2\ \ 4\ \ \underset{\overset{\uparrow}{10}}{8}\ \ \dots\ ]$.

1. The integer *n* = 10 satisfies
$\sum_{i=1}^{3} Vi < 10 \leq \sum_{i=1}^{4} Vi = 7 < 10 \leq 15.$

   Hence, *k* = 4.

2. $d = 10 - \sum_{i=1}^{3} Vi - 1 = 10 - 7 - 1 = 2$.

3. *k* = 4 encoded in unary (unary code is 1110), and *d* must be written in $\lceil \log_2 V_k \rceil = \lceil \log_2 8 \rceil = 3$ bits.

The general code of 10 is :
Code = $U(k)|d_B$
= $U(4)|2_B$ *2 must be represent in 3 bits.*
= 1110|010

Ex2

Encode the integer n = 19 , using the general prefix code.

The infinite sequence $V = [\ 1\ \ 2\ \ 4\ \ 8\ \ \underset{\overset{\uparrow}{19}}{16}\ \dots\ ]$.

1. The integer *n* = 19 satisfies
$\sum_{i=1}^{4} Vi < 19 \leq \sum_{i=1}^{5} Vi = 15 < 19 \leq 31.$

   Hence, *k* = 5.

2. $d = 19 - \sum_{i=1}^{4} Vi - 1 = 19 - 15 - 1 = 3$.

3. *k* = 5 encoded in unary (unary code is 11110), and *d* must be written in $\lceil \log_2 V_k \rceil = \lceil \log_2 16 \rceil = 4$ bits.

The general code of 19 is :
Code = $U(k)|d_B$

$= U(5)|3_B$ *3 must be represented in 4 bits*

$= 11110|0011$

The Golomb Code

The *Golomb code* for nonnegative integers *n* , can be an effective Huffman code. The code depends on the choice of a parameter ***b***. The first step is to compute the two quantities :

$$q = \lfloor \frac{n-1}{b} \rfloor \quad , \quad r = n - qb - 1,$$

(where the notation $\lfloor x \rfloor$ implies truncation of *x)*.

The Golomb code is constructed of two parts; the first is the value of $q + 1$, coded in unary, followed by r represented in binary depending on the selected base.

Golomb (n) = U (q + 1) | r

Choosing $b = 3$, e.g., produces three possible remainders, 0, 1, and 2. They are coded 0, 10, and 11, respectively. Choosing $b = 5$ produces the five remainders 0 through 4, which are coded 00, 01, 100, 101, and 110. Table 2 shows (r) code of the Golomb code for $b = 3$ and $b = 5$.

r	0	1	2	3	4
b = 3	0	10	11		
b = 5	00	01	100	101	110

Table 2 : r code in Golomb Codes for $b = 3$ and $b = 5$.

<u>Ex1</u>

Encode the integer n = 8 , using the Golomb code choosing b = 3.

$q = \lfloor \frac{n-1}{b} \rfloor$

$q = \lfloor \frac{8-1}{3} \rfloor = \lfloor \frac{7}{3} \rfloor = 2$

$r = n - qb - 1,$

$= 8 - 2 * 3 - 1 = 1$

Golomb (n) = U (q + 1) | r

Golomb (8) = U (3) | 1 *we obtain r from table 2*

= 110 | 10

<u>Ex2</u>

Encode the integer n = 7 , using the Golomb code choosing b = 5.

$q = \lfloor \frac{n-1}{b} \rfloor$

$q = \lfloor \frac{7-1}{5} \rfloor = \lfloor \frac{6}{5} \rfloor = 1$

$r = n - qb - 1,$

$= 7 - 5 * 1 - 1 = 1$

Golomb (n) = U (q + 1) | r

Golomb (7) = U (2) | 1 *we obtain r from table 2*

= 10 | 01

A Variant of Huffman coding

The Huffman method assumes that the ***frequencies*** of occurrence of all the symbols of the alphabet are ***known*** to the compressor. In practice, the frequencies are ***seldom, if ever, known*** in advance. One approach to this problem is for the compressor to ***read*** the original data ***twice***. The ***first time***, it just calculates the frequencies. The ***second time***, it compresses the data. Between the two passes, the compressor constructs the Huffman tree. Such a method is called semiadaptive and is normally too slow to be practical. The method that is used in practice is called adaptive (or dynamic) Huffman coding. This method is the basis of the UNIX compact program.

This variant of the adaptive Huffman method is ***simpler*** but ***less efficient***. The idea is to calculate a set of *n* variable-size codes based on equal probabilities, to assign those codes to the *n* symbols at random, and to change the assignments "on the fly," as symbols are being ***read and compressed***. The method is not efficient since the codes are ***not based*** on the ***actual probabilities*** of the symbols in the input stream. However, it is simpler to implement and also ***faster*** than the adaptive method described above, ***because*** it has to ***swap rows*** in a table, rather than ***update a tree***, when updating the frequencies of the symbols.

The main data structure is an ***n × 3 table*** where the three columns store the names of the *n* symbols, their frequencies of occurrence so far, and their codes. The table is always ***kept sorted by the second column***. When the frequency counts in the second column change, rows are swapped, but only ***columns 1 and 2 are moved.***The codes in ***column 3 never change***.

Figure 1 shows an example of four symbols and the behavior of the method when the string *"a2, a4, a4"* is compressed.

Figure 1 a shows the initial state. After the first symbol *a2* is read, its count is incremented, and since it is now the largest count, rows 1 and 2 are swapped (Figure 1 b). After the second symbol *a4* is read, its count is incremented and rows 2 and 4 are swapped (Figure 1c). Finally, after reading the last symbol *a4,*its count is the largest, so rows 1 and 2 are swapped (Figure 1d).

Name	Count	Code	Name	Count	Code	Name	Count	Code	Name	Count	Code
a1	0	0	*a2*	1	0	*a2*	1	0	*a4*	2	0
a2	0	10	*a1*	0	10	*a4*	1	10	*a2*	1	10
a3	0	110	*a3*	0	110	*a1*	0	110	*a1*	0	110
a4	0	111	*a4*	0	111	*a3*	0	111	*a3*	0	111

(*a*)   (*b*)   (*c*)   (*d*)

Figure 1 : Four Steps in a Huffman Variant coding.

Hence, the input string is *a2a4a4* $\Longrightarrow$ output is 0|10|0

<u>Ex</u>

Design a binary code for the source symbol shown in figure 2 using the variant of Huffman method, assume that the input stream is *a3a5a4a4a3a2a3*.

Name	Count	Code	Name	Count	Code	Name	Count	Code	Name	Count	Code
a1	0	0	*a3*	1	0	*a3*	1	0	*a3*	1	0
a2	0	10	*a1*	0	10	*a5*	1	10	*a5*	1	10
a3	0	110	*a2*	0	110	*a1*	0	110	*a4*	1	110
a4	0	1110	*a4*	0	1110	*a2*	0	1110	*a1*	0	1110
a5	*0*	1111	*a5*	*0*	1111	*a4*	*0*	1111	*a2*	*0*	1111

(*a*)   (*b*)   (*c*)   (*d*)

Name	Count	Code	Name	Count	Code	Name	Count	Code	Name	Count	Code
a4	2	0	*a4*	2	0	*a4*	2	0	*a3*	3	0
a3	1	10	*a3*	2	10	*a3*	2	10	*a4*	2	10
a5	1	110	*a5*	1	110	*a5*	1	110	*a5*	1	110
a1	0	1110	*a1*	0	1110	*a2*	1	1110	*a2*	1	1110
a2	*0*	1111	*a2*	*0*	1111	*a1*	*0*	1111	*a1*	*0*	1111

(e)   (f)   (g)   (h)

Figure 2 : eight Steps in a Huffman Variant coding.

input string is *a3a5a4a4a3a2a3* $\Longrightarrow$ 0|10|110|0|10|1110|0

The only point that can cause a **problem** with this method is ***overflow*** of the ***count fields***. If such a field is *k* bits wide, its maximum value is 2^k - 1, so it will overflow when incremented for the 2^k th time. This may happen if the size of the input stream is not known in advance, which is very common. Fortunately, we do not really need to know the counts, we just need them in sorted order, making it easy to solve this problem.

One solution is to count the input symbols and, ***after*** $\boldsymbol{2^k - 1}$ symbols ***are input*** and compressed, to (integer) ***divide*** all the count fields ***by 2*** (or ***shift*** them one position to the right, if this is easier).

Another, similar, solution is to ***check*** each ***count field*** every time it is ***incremented*** and, if it has ***reached*** its ***maximum*** value (if it consists of all ones), to (integer) ***divide*** all the count fields ***by 2*** as above. This approach requires fewer divisions but more complex tests.

Whatever solution is adopted should be used by both the compressor and decompressor.

MNP5

Microcom, Inc., a maker of **modems,** has developed a protocol (called MNP, for Microcom Networking Protocol) for use in its modems. Among other things, the MNP protocol specifies how to ***unpack bytes*** into individual bits before they are sent by the modem. These methods (especially MNP5) have become very popular and are currently used by most modern modems.

The MNP5 method is commonly used for data compression by modems. MNP5 method is a two-stage process that starts with run-length encoding, followed by adaptive frequency encoding.

First Stage : Run-length encoding , which has been described in RLE Text Compression, and it has been solved in MNP5 in different manner. When three or more identical consecutive bytes are found in the source stream, the compressor emits three copies of the byte onto its output stream, followed by a repetition count. When the decompressor reads three identical consecutive bytes, it knows that the next byte is a repetition count (which may be zero, indicating just three repetitions). A ***disadvantage*** of the method is that a run of three characters in the input stream results in four characters written to the output stream (**expansion**).A run of four characters results in **no compression**. Only runs longer than four characters do actually get compressed. Another, slight, problem is that the maximum count is artificially limited to 255.

Ex1

Use the first stage of MNP5 to compress the string

aaabccccddeeeeef

Sol : aaa3bccc4ddeee5f

Or aaa0bccc1ddeee2f

Where 0 means 3 , 1 means 4 and 2 means 5 in order to **maximize the limit** of the count by 3.

Calculating Compression Factor for MNP5 stage 1

As we state in RLE text compression that the compression factor (CF) is :

$$\text{Compression Factor} = \frac{N}{N - M(L-3)}$$

For calculating the CF for MNP5, just substitute 4 for 3.

Ex2

Calculate the CF for MNP5 stage1 (Run Length encoding) in which N = 1000, M = 50, L = 10.

For MNP5 the compression factor $= \frac{N}{N - M(L-4)}$

CF = 1000/[1000 - 50(10 - 4)] = 1.428.

You can see that RLE text compression gives better result than RLE MNP5 compression because CF is 1.538 in RLE text compression while CF is 1.428 in MNP5.

Ex3

for the following message "ffcccccfaaaaaafbbbbbbbbbbbbbf" calculate the CF for stage1 of MNP5 method.

The compressed message will be ffccc5faaa6fbbb**13**f

Or ffccc2faaa3fbbb**10**f in order to **maximize the limit** of the count by 3.

Note that the count 13 is encoded as ASCII code with only 1 byte length.

M = 3

$L = \frac{5+6+13}{3} = 8$

compression factor $= \frac{N}{N - M(L-4)} = \frac{29}{29 - 3(8-4)}$

$CF = \frac{29}{20-3} = \frac{29}{17} = 1.7$

Where **17** represent the uncompressed file length.

Second Stage : Adaptive Frequency Encoding , which operates on the bytes in the partially compressed stream generated by the first stage. Stage 2 is similar to the method of Variant of Huffman Code. It starts with a table of *256×2* entries, where each entry corresponds to one of the 256 possible 8-bit bytes (ASCII) from 00000000 to 11111111. The ***first column***, the frequency counts, is initialized to all zeros. ***Column 2*** is initialized to variable-size codes, called ***tokens***, that vary from a short "000|0" to a long "111|11111110". Column 2 with the tokens is shown in

Table 2 (which shows column 1 with frequencies of zero). ***Each token starts with a 3-bit header***, followed by some code bits.

The code bits (***with three exceptions***) are the two 1-bit codes 0 and 1, the four 2-bit codes 0 through 3, the eight 3-bit codes 0 through 7, the sixteen 4-bit codes, the thirty-two 5-bit codes, the sixty-four 6-bit codes, and the one hundred and twenty seven 7-bit codes. This provides for a total of 2 + 4 + 8 + 16 + 32 + 64 + 127 = 253 codes. The three exceptions are the first two codes "000|0" and "000|1", and the last code, which is "111|11111110" instead of the expected "111|1111111".

When stage 2 starts, all 256 entries of column 1 are assigned frequency counts of zero. When the next byte (e.g. B) is read from the input stream (actually, it is read from the output of the first stage), the corresponding token is written to the output stream, and the frequency of entry B is incremented by 1. Following this, ASCII code may be swapped to ensure that table entries with large frequencies always have the shortest tokens. **Notice** that only the ASCII code with its corresponding frequency are swapped, not the token. Thus the first entry always corresponds to token 000|0 and contains its frequency count. The ASCII code of this entry , however, may change from the original "00000000" to another ASCII code if other ASCII code achieve higher frequency counts.

Byte ASCII	Freq.	Token	Byte ASCII	Freq.	Token	Byte ASCII	Freq.	Token
0	0	000|0	16	0	100|0000	32	0	101|00000
1	0	000|1	17	0	100|0001	33	0	101|00001
2	0	001|0	18	0	100|0010	34	0	101|00010
3	0	001|1	19	0	100|0011	:		
4	0	010|00	20	0	100|0100	62	0	101|11110
5	0	010|01	21	0	100|0101	63	0	101|11110
6	0	010|10	22	0	100|1110	64	0	110|000000
7	0	010|11	23	0	100|1111	65	0	110|000001
8	0	011|000	24	0	100|1000	66	0	110|000010
9	0	011|001	25	0	100|1001	:		
10	0	011|010	26	0	100|1010	250	0	111|1111010
11	0	011|011	27	0	100|1011	251	0	111|1111011
12	0	011|100	28	0	100|1100	252	0	111|1111100
13	0	011|101	29	0	100|1101	253	0	111|1111101
14	0	011|110	30	0	100|1110	254	0	111|1111110
15	0	011|111	31	0	100|1111	255	0	111|11111110

Table 3. the MNP5 tokens

The ***frequency counts*** are stored in 8-bit fields. Each time a count is incremented, the algorithm ***checks*** to see whether it has ***reached its maximum*** value. If yes, all the counts are scaled down by (integer) ***dividing*** them ***by 2***.

Another, subtle, point has to do with interaction between the two compression stages. Recall that each repetition of three or more characters is replaced, in stage 1, by three repetitions, followed by a byte with the repetition count. When these four bytes arrive at stage 2, they **all** replaced by tokens, but the ***fourth one*** does **not cause** an ***increment*** to the ***frequency*** of that count.

Example: Suppose that the character with ASCII code 52 repeats six times. Stage 1 will generate the four bytes "52, 52, 52, 3," and stage 2 will **replace each** with a token, will increment the entry for "52" (entry 53 in the table) by 3, but ***will not increment the entry for "3"*** (which is entry 4 in the table). (The three tokens for the ***three bytes of "52"*** may ***all be different***, since tokens may be **swapped** after each "52" is read and processed.)

The efficiency of MNP5 is a result of both stages. The efficiency of stage 1 depends heavily on the original data. Stage 2 also depends on the original data, but to a smaller extent. ***Stage 2*** tends to identify the ***most frequent characters*** in the data and **assign** them the ***short codes***. A look at Table 2 shows that **32** of the 256 characters have tokens that are ***7 bits or fewer*** in length, thus resulting in compression. The other 224 characters have tokens that are 8 bits or longer. When one of these characters is replaced by a long token, the result is no compression, or even expansion.

The efficiency of MNP5 thus depends on how many characters dominate the original data. If all characters occur at the same frequency, expansion will result. **In** the other extreme case, if only four characters appear in the data, each will be assigned a 4-bit token, and the compression factor will be 2. Explain??

o **Exercise:** Assuming that all 256 characters appear in the original data with the same probability (1/256 each), what will the ***expansion factor*** in stage 2 be?

Sol

$$Compression\ Factor = \frac{Uncompressed\ file\ size}{compressed\ file\ size}$$

$$\text{CF} = \frac{256*8}{4*4+4*5+8*6+16*7+32*8+64*9+127*10+1*11} = \frac{2048}{16+20+48+112+256+576+1270+11}$$

$$= \frac{2048}{2309} = \mathbf{0.887} \text{ which mean expansion.}$$

Updating the Table

The process of updating the table of MNP5 codes by swapping rows can be done in two ways:

1. Sorting the entire table every time a frequency is incremented. This is simple in concept but **too slow** in practice, because the table is 256 entries long.
2. Using pointers in the table, and swapping pointers such that items with large frequencies will **point** to short codes.

Dictionary methods

Dictionary-based compression methods does not use a statistical model, nor do they use variable-size code. Instead they select **string** of symbols and encode each string as a **token (index),** using a dictionary. The dictionary holds string of symbols and it may be **static or dynamic** (adaptive). The **former** ***is permanent*** , some times ***allowing the additional of string but not deletion***, where as the **later** hold string previously found in the input stream, ***allowing for addition and deletion of string*** as a new input is being read.

Static dictionary

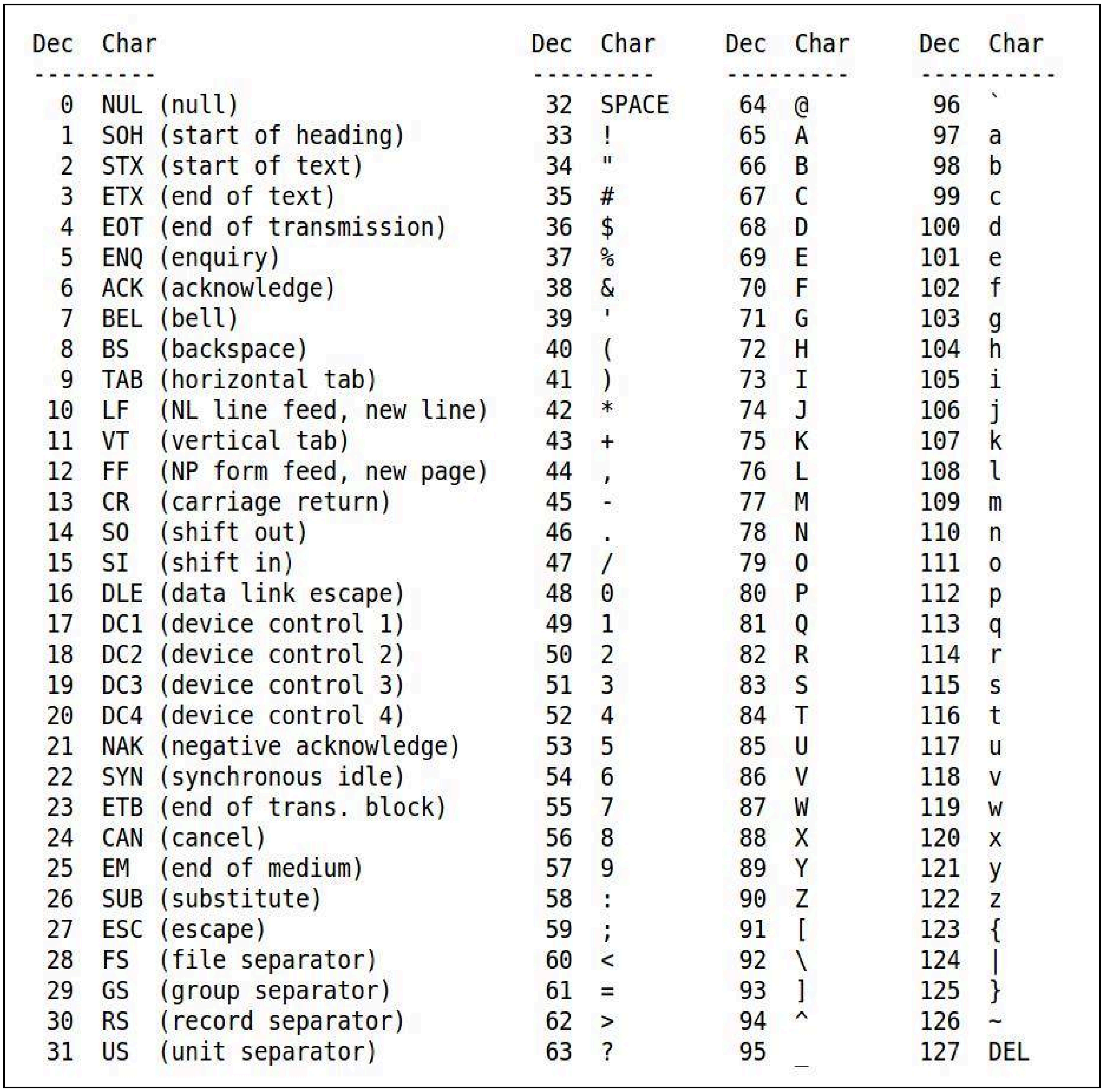
The simplest example of a static dictionary is the dictionary of English language used to compress English text. A word is read from the input stream and the dictionary is searched. If a match is found, an **index** to the dictionary is written into the output stream, otherwise, the uncompressed **word itself** is written.

As a result, the output stream contains ***index*** and ***row words*** and we need to distinguish between them, and to do that we must use an ***extra bit*** in every item written. In practice, a 19-bit index is sufficient to specify an item in a 2^{19} = 524288 word dictionary. Thus when a ***match is found***, we can write **20-bit** token consisting of a flag bit (perhaps is 0) followed by 19-bit index. When ***no match is found***, a flag of "1" is written, followed by the size of the un match word, followed by the word itself.

Example

Assuming that the word "bet" is found in dictionary entry 1025, it is encoded as 20-bit number 0|0000000010000000001.

Assuming that the word "xet" is not found, it is encoded 1|0000011|01111000|01100101|01110100. This is a 4-byte number where the 7-bit field 0000011 indicate that three more byte follow, while the three other bytes represent the ASCII code for the small letters x , e , t respectively.



Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Argument

Assuming that the size is written as a 7-bit number, and that an average word **size** is **five** characters, an uncompressed word occupies, on average, 6 bytes (= 48 bits) in the output stream. Compressing 48 bits into 20 is excellent, provided that it happens often enough. Thus, we have to answer the question; How ***many matches*** are needed in order to have ***overall compression***? We denote the probability of a match (the case where the word is found in the dictionary) by **P**. After reading and compressing **N** words, the size of the **output stream** will be N[20P + 48(1 - P)] = N[48 - 28P] bits. The size of the **input stream** is (assuming five characters per word) 40N bits. Compression is achieved when N[48 - 28P] < 40N, which implies P > 0.29. We need a matching rate of 29% or better to achieve compression.

Exercise: What compression factor do we get with P = 0.9?

As long as the input stream consist of English text, most words will be found in the dictionary. Other type of data, however, may not found. A file containing the source code of a computer program may contain words such as Cout, XOR, Malloc that may be not found in the English dictionary. This show that the static dictionary is not a good choice for a general-purpose compressor.

Adaptive dictionary

In general adaptive-based method is preferable. Such a method can start with an **empty** dictionary or with a **small ,** default dictionary, **add** words to it as they are found in the input stream, and **delete** old words, since a **big** dictionary mean **slow search**. If a match is found, then a token (**index**) will be written on the output stream, otherwise the uncompressed word should be written and also **added** to the dictionary. The last step in each iteration check to see whether an old word should be **deleted** from the dictionary.

The **advantages** of the adaptive dictionary are :

1. It involves **string** search and **match** operation, rather than numerical computations. many programmers prefer that.
2. Each of the encoder & decoder uses a different algorithm (this is an **asymmetric** compression). in statistical compression methods, the

decoder is normally the exact opposite of the encoder (**symmetric** compression).

LZ77 (Sliding Window)

The main idea of this method (Ziv 77) , is to use part of the previously seen input stream as the dictionary. The encoder maintains a window to the input stream and shifts the input in that window **from right to left** as strings of symbols are being encoded. The method is thus based on a **sliding window**. The window is divided into two parts. The part on the left is called the **search buffer**. This is the current dictionary, and it always includes symbols that have recently been input and encoded. The part on the right is the **look-ahead buffer,** containing text yet to be encoded. In practical implementations the search buffer is some ***thousands of bytes*** long, while the look-ahead buffer is only ***tens of bytes*** long. The vertical bar between the t and the e below represents the current dividing line between the two buffers. We thus assume that the text "sir␣sid␣eastman␣easily␣t" has already been compressed, while the text "eases␣sea␣sick␣seals" still needs to be compressed.

Search buffer	look-ahead buffer
sir␣sid␣eastman␣easily␣t	eases␣sea␣sick␣seals ....

The encoder scans the search buffer **backwards** (from right to left) looking for a match to the first symbol **"e"** in the look-ahead buffer. It finds one at the **"e"** of the word easily. This e is at a **distance** (offset) of 8 from the end of the search buffer. The encoder then matches as many symbols following the two e's as possible. Three symbols **"eas"** match in this case, so the **length of the match** is 3. The encoder then continues the backward scan, trying to find longer matches. In our case, there is one more match, at the word **eastman**, with distance 16, and it has the same length. ***The encoder selects the longest match or, if they are all the same length, the last one found*** and prepares the token (16, 3, *"e")*.

Selecting the last match, rather than the first one, simplifies the encoder, since it only has to keep track of the last match found. It is interesting to note that selecting the first match, while making the program somewhat more complex, also has an advantage. It selects the smallest offset. It would seem that this is not an advantage, since a token should have room for the largest possible offset.

Exercise: How does the decoder know whether the encoder selects the first match or the last match?

In general, an LZ77 token has three parts: **distance, length, and next symbol** in the look-ahead buffer (which, in our case, is the second **e** of the word teases). This token is written on the output stream, and ***the window is shifted to the right four positions: three positions for the matched string and one position for the next symbol.***

Search buffer	look-ahead buffer
sir␣sid␣eastman␣easily␣t ease	s␣sea␣sick␣seals ....

If the backward search yields no match, an LZ77 token with zero distance and length and with the unmatched symbol is written. This is also the reason a token has to have a third component. distances with zero offset and length are common at the beginning of any compression job, when the search buffer is empty or almost empty. The first five steps in encoding our example are the following:

Search buffer	look-ahead buffer
	sir␣sid␣eastman␣
s	ir␣sid␣eastman␣e
si	r␣sid␣eastman␣ea
sir	␣sid␣eastman␣eas
sir␣	sid␣eastman␣easi
sir␣sid	␣eastman␣easily␣t

The ***output token*** (**distance, length, next symbol**) of the first five steps are :

(0,0, "s")
(0,0, "i")
(0,0, "r")
(0,0, "␣")
(4,2, "d")

In practice, the ***search buffer*** may be a ***few thousand*** bytes long, so the **offset (distance)** size is typically **10-12** bits. In practice, the ***look-ahead buffer*** is only a ***few tens*** of bytes long, so the size of the **length** field is just a **few bits**. The size of the **symbol** field is typically **8 bits**. The **total size** of the **output token** (distance, length, next symbol) may typically be :

Output token = 11 + 5 + 8 = 24 bits.

Decoding

The decoder is much simpler than the encoder (LZ77 is thus an **asymmetric** compression method). It has to maintain a buffer, equal in size to the encoder's window. ***The decoder inputs a token, finds the match in its***

buffer, writes the match and the third token field on the output stream, and shifts the matched string and the third field into the buffer.

Because of the nature of the sliding window, the LZ77 method always compares the look-ahead buffer to the ***recently input text*** in the search buffer and ***never*** to text that ***was input long ago*** (and has thus been ***flushed out*** of the search buffer). The method thus implicitly assumes ***that patterns in the input data occur close together***. Data that satisfies this assumption will compress well.

argument

The basic LZ77 method was improved in several ways. One way to improve it is to use ***variable-size*** "offset" and "length" fields in the tokens. Another way is to increase the sizes of both buffers. ***Increasing the size*** of the ***search buffer*** makes it possible to find ***better matches***, but the tradeoff is an ***increased search time***. A large search buffer thus requires a more ***sophisticated data structure*** that allows for ***fast search*** .

Ex (LZ77 Coding)

Given the below search buffer and look-ahead buffer . apply the LZ77 compression method, show the resulted output token , and the dictionary content at each step.

Search buffer	look-ahead buffer
that␣is ␣my␣hat␣	this␣is␣his␣hair…..

Sol

Search buffer	look-ahead buffer
….that␣is ␣my␣hat␣	this␣is␣his␣hair
that␣is␣my␣hat␣thi	s␣is␣his␣hair
that␣is␣my␣hat␣this␣i	s␣his␣hair
that␣is␣my␣hat␣this␣is␣h	is␣hair
that␣is␣my␣hat␣this␣is␣his␣ha	ir
that␣is␣my␣hat␣this␣is␣his␣hair	

The ***output token*** (distance, length, next symbol) of the first five steps are :

(15,2, "i")
(12,2, "i")
(15,2, "h")
(4, 4, "a")
(24,1, "r")

Ex (LZ77 DeCoding)

Given the below search buffer. For the following ***input token*** , apply the LZ77 decompression method, show the resulted dictionary content at each step.

Search buffer

….that␣is ␣my␣hat␣

(15,2, "i") , (12,2, "i") , (15,2, "h") , (4,4, "a") , (24,1, "r")

LZ78

The LZ78 method ([Ziv 78] ***does not use*** any search buffer, look-ahead buffer, or sliding window. Instead, there is a dictionary of previously encountered strings. This dictionary ***starts empty*** (or almost empty), and its size is limited only by the amount of available memory. The encoder outputs two-field tokens. The ***first field*** is a pointer to the dictionary; the ***second*** is the code of a symbol. Tokens ***do not*** contain the ***length*** of a string, since this is implied in the dictionary. Each token corresponds to a string of input symbols, and that string is added to the dictionary after the token is written on the compressed stream. Nothing is ***ever deleted*** from the dictionary, which is both an *advantage* over LZ77 (since future strings can be compressed even by strings seen in the distant past) and a ***liability*** (since the dictionary tends to grow fast and to fill up the entire available memory).

The dictionary ***starts*** with the ***null string*** at position ***zero***. As symbols are input and encoded, strings are added to the dictionary at positions 1, 2, and so on. When the next symbol **x** is read from the input stream, the dictionary is searched for an entry with the one-symbol string **x**. ***If none are found, x is added to the next available position in the dictionary, and the token (0, x) is output.*** This token indicates the string "null **x**" (a concatenation of the null string and **x**). ***If an entry with x is found (at position 37, say), the next symbol y is read, and the dictionary is searched for an entry containing the two-symbol string xy. If none are found, then string xy is added to the next available position in the dictionary, and the token (37, y) is output.*** This token indicates the string **xy**, since 37 is the dictionary position of string **x**. The process continues until the end of the input stream is reached.

In general, the current symbol is read and becomes a one-symbol string. The encoder then tries to find it in the dictionary. ***If the symbol is found in the dictionary, the next symbol is read and concatenated with the first to form a two-symbol string that the encoder then tries to locate in the dictionary.*** As long as those strings are found in the dictionary, more symbols are read and concatenated to the string. At a certain point the string is not found in the dictionary, so the encoder adds it to the dictionary and outputs a token with the last dictionary match as its first field, and the last symbol of the string (the one that caused the search to fail) as its second field. Table 1. below shows the first 14 steps in encoding the string

"sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals".

dictionary	Token (output)	dictionary	Token (output)
0 **null**			
1 **"s"**	(0, "s")	8 **"a"**	(0, "a")
2 **"i"**	(0, "i")	9 **"st"**	(1, "t")
3 **"r"**	(0, "r")	10 **"m"**	(0, "m")
4 **"␣"**	(0, "␣")	11 **"an"**	(8, "n")
5 **"si"**	(1, "i")	12 **"␣ea"**	(7, "a")
6 **"d"**	(0, "d")	13 **"sil"**	(5,"1")
7 **"␣e"**	(4, "e")	14 **"y"**	(0, "y")

Table 1 : First 14 Encoding Steps in LZ78.

In each step, the string added to the dictionary is the one being encoded, minus its last symbol. ***In a typical compression run, the dictionary starts with short strings, but as more text is being input and processed, longer and longer strings are added to it.*** The size of the dictionary can either be fixed or may be determined by the size of the available memory each time the LZ78 compression program is executed. ***A large dictionary may contain more strings and thus allow for longer matches, but the tradeoff is longer pointers (and thus bigger tokens) and slower dictionary search***.

Since the total size of the dictionary is ***limited*** (because the number of bits that allocated to the dictionary pointer are 16 bits), it may fill up during compression. This, in fact, happens all the time except when the input stream is unusually ***small*.**

When the dictionary is ***full, delete*** some of the least recently used entries, to make room for new ones. Unfortunately there is no good algorithm to decide which entries to delete, and how many.

LZ78 Decoding

The LZ78 decoder works by building and maintaining the dictionary in the ***same way as the encoder*** does.

It reads its input stream (which consists of a pointers to the dictionary and its corresponding letter) and uses each pointer to retrieve uncompressed symbols from its dictionary and write them on its output stream.

1. The dictionary ***starts*** with the ***null string*** at position ***zero***. As symbols are input and decoded, strings are added to the dictionary at positions 1, 2, and so on.
2. Read the input stream (pointers to the dictionary and its corresponding letter) :
   a. If the pointer is equal to zero, ***then the letter is outputted***, ***and added to the next available position in the dictionary***.
   b. If the pointer is greater than zero , then concatenate the letter with the letter(s) in the dictionary that correspond to the pointer, ***the concatenated letters is outputted***, ***and added to the next available position in the dictionary***.

Ex

What is the output and the new dictionary entry of the LZ78 decompression method to the below code :
(0, "s"), (0, "i"), (0, "r"), (0, "␣"), (1, "i"), (0, "d"), (4, "e"), (0, "a"), (1, "t"), (0, "m"), (8, "n"), (7, "a") , (5,"1"), (0, "y")

Sol

dictionary	output	dictionary	output
0 **null**			
1 **"s"**	s	8 **"a"**	a
2 **"i"**	i	9 **"st"**	st
3 **"r"**	r	10 **"m"**	m
4 **"␣"**	␣	11 **"an"**	an
5 **"si"**	si	12 "**␣ea**"	␣ea
6 **"d"**	d	13 **"sil"**	si1
7 **"␣e"**	␣e	14 **"y"**	y

LZW

This is method is developed by Terry Welch in 1984. An LZW token consists of just a pointer to the dictionary. To best understand LZW, we will temporarily forget that the dictionary is a tree, and will think of it as an array of variable-size strings. The LZW method starts by initializing the dictionary to all the symbols in the alphabet. In the common case of 8-bit symbols, the first 256 entries of the dictionary (entries 0 through 255) are occupied before any data is input. Because the dictionary is initialized, the next input character will always be found in the dictionary. This is why an LZW token can ***consist of just a pointer*** and does not have to contain a character code as in LZ77 and LZ78.

The principle of LZW is that the encoder inputs symbols one by one and accumulates them in a string I. After each symbol is input and is concatenated to I, the dictionary is searched for string I. As long as I is found in the dictionary, the process continues. At a certain point, adding the next symbol x causes the search to fail; string I is in the dictionary but string Ix (symbol x concatenated to I) is not. At this point the encoder

(1) outputs the dictionary pointer that points to string I.

(2) saves string Ix (which is now called a *phrase)* in the next available dictionary entry.

(3) initializes string I to symbol x.

To illustrate this process, we again use the text string **"sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals".** The steps are as follows:

1. Initialize entries 0-255 of the dictionary to all 256 8-bit bytes.

2. The first symbol **"s"** is input and is found in the dictionary (in entry 115, since this is the ASCII code of **"s"**). The next symbol **"i"** is input, but **"si"** is not found in the dictionary. The encoder performs the following:
(1) outputs 115.
(2) saves string **"si"** in the next available dictionary entry (entry 256).
(3) initializes **I** to the symbol **"i".**

3. The **"r"** of sir is input, but string **"ir"** is not in the dictionary. The encoder :
(1) outputs 105 (the ASCII code of **"i").**
(2) saves string "**ir**" in the next available dictionary entry (entry 257).
(3) initializes **I** to the symbol **"r".**

Table 2 summarizes all the steps of this process. Table 3 shows some of the original 256 entries in the LZW dictionary plus the entries added during encoding of the string above.

The complete output stream is (***only the numbers are output***, not the strings in parentheses) as follows:
115 (s), 105 (i), 114 (r), 32 (␣), 256 (si), 100 (d), 32 (␣), 101 (e), 97 (a), 115 (s), 116 (t), 109 (m), 97 (a), 110 (n), 262 (␣e), 264 (as), 105 (i), 108 (1), 121 (y), 32 (␣), 116 (t), 263 (ea), 115 (s), 101 (e), 115 (s), 259 (us), 263 (ea), 259 (␣s), 105 (i), 99 (c), 107 (k), 281 (␣se), 97 (a), 108 (1), 115 (s), eof.

I	in dict?	new Entry	output	I	in diet?	new entry	output
s	Y			y	Y		
si	N	256-si	115 (s)	y␣	N	274-y␣	121 (y)
i	Y			␣	Y		
ir	N	257-ir	105 (i)	␣t	N	275-␣t	32 (␣)
r	Y			t	Y		
r␣	N	258-r ␣	114 (r)	te	N	276-te	116 (t)
␣	Y			e	Y		
␣s	N	259-␣s	32 (␣)	ea	Y		
s	Y			eas	N	277-eas	263 (ea)
si	Y			s	Y		
sid	N	260-sid	256 (si)	se	N	278-se	115 (s)
d	Y			e	Y		
d␣	N	261-d␣	100 (d)	es	N	279-es	101 (e)
␣	Y			s	Y		
␣e	N	262-␣e	32 (␣)	s␣	N	280-s␣	115 (s)
e	Y			␣	Y		
ea	N	263-ea	101 (e)	␣s	Y		
a	Y			␣se	N	281-␣se	259 (␣s)
as	N	264-as	97 (a)	e	Y		
s	Y			ea	Y		
st	N	265-st	115 (s)	ea␣	N	282-ea␣	263 (ea)
t	Y			␣	Y		
tm	N	266-tm	116 (t)	␣s	Y		
m	Y			␣si	N	283-␣si	259 (␣s)
ma	N	267-ma	109 (m)	i	Y		
a	Y			ic	N	284-ic	105 (i)
an	N	268-an	97 (a)	c	Y		
n	Y			ck	N	285-ck	99 (c)
n␣	N	269-n␣	110 (n)	k	Y		
␣	Y			k␣	N	286-k␣	107 (k)

␣e	Y			␣	Y		
␣ea	N	270-␣ea	262 (␣e)	␣s	Y		
a	Y			␣se	Y		
as	Y			␣sea	N	287-␣sea	281 (␣se)
asi	N	271-asi	264 (as)	a	Y		
i	Y			al	N	288-al	97 (a)
il	N	272-il	105 (i)	1	Y		
l	Y			1s	N	289-ls	108 (1)
ly	N	273-ly	108 (1)	s	Y		
				s,eof	N	115 (s)	

Table 2: LZW Encoding of "sir sid eastman easily teases sea sick seals".

Index	Symbol	Index	Symbol	Index	Symbol	Index	Symbol
0	NULL	110	n	262	␣e	276	te
1	SOH	:	:	263	ea	277	eas
:	:	115	s	264	as	278	se
32	Space	116	t	265	st	279	es
:	:	:	:	266	tm	280	s␣
97	a	121	y	267	ma	281	␣se
98	b	:	:	268	an	282	ea␣
99	c	255	255	269	n␣	283	␣si
100	d	256	si	270	␣ea	284	ic
101	e	257	ir	271	asi	285	ck
:	:	258	r␣	272	il	286	k␣
107	k	259	␣s	273	ly	287	␣sea
108	l	260	sid	274	y␣	288	al
109	m	261	d␣	275	␣t	289	ls

Table 3: An LZW Dictionary.

```
for i:=0 to 255 do
    append i as a 1-symbol string to the dictionary;
append λ to the dictionary;
di:=dictionary index of λ;
repeat
    read(ch);
    if «di,ch» is in the dictionary then
```

```
        di:=dictionary index of «di,ch»;
     else
        output(di);
        append «di,ch» to the dictionary;
        di:=dictionary index of ch;
     endif ;
until end-of-input;
```

The LZW Algorithm.

Above is a pseudo-code listing of the algorithm. We denote by λ the empty string, and by «a, b» the concatenation of strings a and b.

The line "append «di, ch» to the dictionary" is of special interest. It is clear that in practice, the dictionary may fill up. This line should therefore include a test for a full dictionary, and certain actions for the case where it is full.

Since the first 256 entries of the dictionary are occupied right from the start, pointers to the dictionary have to be longer than 8 bits. A simple implementation would typically use ***16-bit pointers***, which allow for a 64K-entry dictionary (where *64K* = 2^{16} = 65,536). Such a dictionary will, of course, ***fill up very quickly*** in all but the ***smallest compression*** jobs. Another interesting fact about LZW is that strings in the dictionary get ***only one character longer at a time***. It therefore takes a ***long time*** to get long strings in the dictionary, and thus a chance to achieve really good compression. We can say that LZW adapts slowly to its input data.

<> **Exercise 1** : Use LZW to encode the string "alf␣eats␣alfalfa". Show the encoder output and the new entries added by it to the dictionary.

<> **Exercise 2** : Analyze the LZW compression of the string "aaaa ... ".

LZW Decoding

In order to understand how the LZW decoder works, we should first recall the three steps the encoder performs each time it writes something on the output stream. They are **(1)** it outputs the dictionary pointer that points to string **I**, **(2)** it saves string **Ix** in the next available entry of the dictionary, and **(3)** it initializes string **I** to symbol **x**.

The decoder starts with the first entries of its dictionary initialized to all the symbols of the alphabet (normally 256 symbols). It then reads its input stream (which consists of pointers to the dictionary) and uses each pointer to retrieve uncompressed symbols from its dictionary and write them on its output stream. It also builds its dictionary in the same way as the encoder (this fact is usually expressed by saying that the encoder and decoder are *synchronized*, or that they work in *lockstep).*

In the first decoding step, the decoder inputs the first pointer and uses it to retrieve a dictionary item **I**. This is a string of symbols, and it is written on the decoder's output stream. String **Ix** needs to be saved in the dictionary, but symbol **x** is still unknown; it will be the first symbol in the next string retrieved from the dictionary.

In each decoding step after the first, the decoder inputs the next pointer, retrieves the next string **J** from the dictionary, writes it on the output stream, ***isolates its first symbol x***, and saves string **Ix** in the next available dictionary entry (after checking to make sure string **Ix** is not already in the dictionary). The decoder then moves **J** to **I** and is ready for the next step.

In our "sir␣sid ... " example, the first pointer that's input by the decoder is 115. This corresponds to the string **"s",** which is retrieved from the dictionary, gets stored in **I** and becomes the first thing write **"i"** is retrieved into **J** and is also written on the output stream. **J's *first symbol*** is concatenated with **I** , to form string **"si",** which does not exist in the dictionary, and is therefore added to it as entry 256. Variable **J** is moved to **I**, so **I** is now the string **"i".** The next pointer is 114, so string **"r"** is retrieved from the dictionary into **J** and is also written on the output stream. **J's** first symbol is concatenated with **I** , to form string **"ir"**, which does not exist in the dictionary, and is added to it as entry 257. Variable **J** is moved to **I**, so **I** is now the string **"r".** The next step reads pointer 32, writes "**␣**" on the output stream, and saves string "**r␣**".

o **Exercise 3 :** Decode the string "alf␣eats␣alfalfa" by using the encoding results from Exercise 1.

o **Exercise 4:** Assume a two-symbol alphabet with the symbols a and b. show the first few steps for encoding and decoding the string "ababab ... " .

Ex

Given the below basic dictionary

0	1	2	3	4	5	6	7	8
␣	a	c	f	h	n	o	s	t

What is the output and the new dictionary entry of the LZW decompression method to the below code :
7, 4, 1, 5, 6, 5, 0, 3, 258, 6, 0, 2, 258, 0, 259, 8.

Sol

256	257	258	259	260	261	262	263	264	265	266	267	268	269	270
sh	ha	an	no	on	n␣	␣f	fa	ano	o␣	␣c	ca	an␣	␣n	not

output	New entry
7(s)	
4(h)	256(sh)
1(a)	257(ha)
5(n)	258(an)
6(0)	259(no)
5(n)	260(on)
0 (␣)	261(n␣)
3(f)	262(␣f)
258(an)	263(fa)
6(o)	264(ano)
0(␣)	265(o␣)
2(c)	266(␣c)
258(an)	267(ca)
0(␣)	268(an␣)
259(no)	269(␣n)
8(t)	270(not)

Arithmetic Coding

The Huffman method is more efficient than the Shannon-Fano method, but either method rarely produces the best variable-size code. In fact these methods produce best results (codes whose average size equals the entropy and hence ***efficiency of 100 %***) only when the symbols have ***probabilities of occurrence that are negative powers of 2***. This is because these methods assign a code with an integral number of bits to each symbol in the alphabet. ***A symbol with probability 0.4 should ideally be assigned a 1.32-bit code, since $-\log_2 0.4 \sim 1.32$. The Huffman method, however, normally assigns such a symbol a code of 1 or 2 bits.***

Arithmetic coding overcomes this problem by ***assigning one (normally long) code to the entire input stream***, instead of assigning codes to the individual symbols. The method reads the input stream symbol by symbol and appends more bits to the code each time a symbol is input and processed. To understand the method, it is useful to imagine the resulting code as a number in the range [0,1). [The notation *[a, b)* means the range of real numbers from *a* to *b*, not including *b*. The range is "closed" at *a* and "open" at *b*.

The ***first step*** is to ***calculate***, or at least to estimate, the ***frequencies of occurrence of each symbol***. For best results, the exact frequencies are calculated by reading the entire input stream in the first pass of a two-pass compression job. If the program has good estimates of the frequencies from a different source, the first pass may be omitted.

Example 1

Given three symbols *a1, a2,* and *a3,* with probabilities *P1* = 0.4, *P2* = 0.5, and *P3* = 0.1, respectively. The interval [0,1) is divided among the three symbols by assigning each a subinterval proportional in size to its probability. The ***order*** of the ***subintervals is immaterial***. In our example, the three symbols are assigned the subintervals [0,0.4), [0.4,0.9), and [0.9,1.0).

Symbols	probability	Range
a3	0.1	[0.9 , 1.0)
a2	0.5	[0.4 , 0.9)
a1	0.4	[0.0 , 0.4)

To encode the string "**a2a2a2a3**", ***we start with the interval [0,1)***. The first symbol *a2* ***reduces*** this interval to the subinterval from its 40% point to its 90% point; hence, the result is [0.4,0.9).

The second *a2* ***reduces*** [0.4,0.9) in the same way (see note below) to [0.6,0.85), the third *a2* ***reduces*** this to [0.7,0.825), and the *a3* ***reduces*** this to the stretch from the 90% point of [0.7,0.825) to its 100% point, producing [0.8125,0.8250). ***The final code our method produces can be any number in this final range.***

The following rules summarize the main steps of arithmetic coding:

1. Start by defining the "current interval" as [0,1).
2. Repeat the following two steps for each symbol *s* in the input stream:
    - 2.1. Divide the current interval into subintervals whose sizes are proportional to the symbols' probabilities.
    - 2.2. Select the subinterval for *s* and define it as the new current interval.
3. When the entire input stream has been processed in this way, the output should be any number that uniquely identifies the current interval (i.e., ***any number inside the current interval).***

Table below show the calculation of the ranges of the arithmetic coding for the above example :

Char		**calculation of low and high**
a2	L	0 + (1 – 0) * 0.4 = 0.4
	H	0 + (1 – 0) * 0.9 = 0.9
a2	L	0.4 + (0.9 – 0.4) * 0.4 = 0.6
	H	0.4 + (0.9 – 0.4) * 0.9 = 0.85
a2	L	0.6 + (0.85 – 0.6) * 0.4 = 0.7
	H	0.6 + (0.85 – 0.6) * 0.9 = 0.825
a3	L	0.7 + (0.825 – 0.7) * 0.9 = 0.8125
	H	0.7 + (0.825 – 0.7) * 1.0 = 0.825

Where **Low** and **High** are being calculated according to:

NewLow = OldLow + Range * LowRange(X);
NewHigh = OldLow + Range * HighRange(X);

Where:

Range=OldHigh - OldLow

LowRange (X), HighRange (X) indicates the low and high limits of the range of symbol X, respectively.

For each symbol processed, the current interval gets smaller, so it takes more bits to express it, but the point is that ***the final output is a single number and does not consist of codes for the individual symbols***. The average code size can be obtained by dividing the size of the output (in bits) by the size of the input (in symbols).

Example 2

In this example, we show the compression steps for the short string "**SWISS␣MISS**". Table 1 shows the information prepared in the first step (the *statistical model* of the data). The five symbols appearing in the input may be arranged in any order. For each symbol, its frequency is first counted, followed by its probability of occurrence (the frequency divided by the string size, 10). The range [0,1) is then divided among the symbols, in any order, with each symbol getting a chunk, or a subrange, equal in size to its probability. Thus "s" gets the subrange [0.5,1.0) (of size 0.5), whereas the subrange of "I" is of size 0.2 [0.2,0.4).

The symbols and Probabilities in Table 1 are written on the output stream before any of the bits of the compressed code. ***This table will be the first thing input by the decoder (i.e. the decoder must receive this table before it start decompression).***

Symbols	probability	Range
S	0.5	[0.5 , 1.0)
W	0.1	[0.4 , 0.5)
I	0.2	[0.2 , 0.4)
M	0.1	[0.1 , 0.2)
␣	0.1	[0.0 , 0.1)

Table 1. Probabilities and Ranges of "**SWISS␣M1SS**" Symbols

The encoding process starts by defining two variables, Low and High, and setting them to 0 and 1, respectively. They define an interval [LOW, High). As symbols are being input and processed, ***the values of Low and High are moved closer together, to narrow the interval***.

After processing the first symbol "S", Low and High are updated to 0.5 and 1, respectively. ***The resulting code for the entire input stream will be a number in this range (0.5 ≤ Code < 1.0). The rest of the input stream will determine precisely where, in the interval [0.5,1), the final code will lie***.

Table 2 below show the calculation of the ranges of the arithmetic coding for the example 2 above :

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	$0.7175325 + (0.717535 - 0.7175325) \times 1.0 = 0.717535$

Table 2. the process of arithmetic encoding

The final code is the final value of Low 0.71753375 , recall that :

NewLow = OldLow + Range * LowRange(X);
NewHigh = OldLow + Range * HighRange(X);
where
Range=OldHigh - OldLow
LowRange (X), HighRange (X) indicate the low and high limits of the range of symbol X, respectively.

Decoding

The decoder works in the opposite way. It starts by inputting the symbols and their ranges (Table of Probabilities and Ranges of inputted Symbols).

Example 1

The decoder starts by reading the code 0.8125. The first digit is "8", so the decoder immediately knows that the entire code is a number of the form 0.8.... , this number is inside the sub range [0.4, 0.9) of a2, so the first symbol is a2. The decoder then ***eliminates the effect*** of symbol a2 from the code by subtracting the lower limit 0.4 of a2 and dividing by the width of the sub range of a2 (0.5). The result is 0.825, which tells the decoder that the next symbol is a2 (since the sub range of a2 is [0.4, 0.9).

Table below show the extracting of the symbols from the arithmetic code for the Example 1 above :

Code = 0.8125 $\Longrightarrow$ x = a2 because a2 rang is [0.4 , 0.9)
Code = (0.8125 – 0.4) / 0.5 = 0.825 $\Longrightarrow$ x = a2
Code = (0.825 – 0.4) / 0.5 = 0.85 $\Longrightarrow$ x = a2
Code = (0.85 – 0.4) / 0.5 = 0.9 $\Longrightarrow$ x = a3
Code = (0.9 – 0.9) / 0.1 = 0.0 $\Longrightarrow$ end of decoding

To eliminate the effect of symbol X from the code, the decoder performs the operation :

Code = (Code - LowRange (X)) / Range
where Range is the width of the sub range of X.

Example 2

The decoder start by reading the code 0.71753375, The first digit is "7", so the decoder immediately knows that the entire code is a number of the form 0.7.... , this number is inside the sub range [0.5, 1) of S, so the first symbol is S. The decoder then ***eliminates the effect*** of symbol S from the code by subtracting the lower limit 0.5 of S and dividing by the width of the sub range of S (0.5). The result is 0.4350675, which tells the decoder that the next symbol is W (since the sub range of W is [0.4, 0.5)).

To ***eliminate the effect*** of symbol X from the code, the decoder performs the operation

Code = (Code - LowRange (X)) / Range
where Range is the width of the sub range of X.

Table 3 summarizes the steps for decoding our example 2 above :

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

Table 3 . The process of Arithmetic Decoding

Skewed Probabilities Problem

The next example is of three symbols with probabilities as shown in Table below. Notice that the ***probabilities*** are ***very different***. One is large (97.5%) and the others much smaller. This is a case of ***skewed probabilities problem.***

Symbols	probability	Range
a1	0.001838	[0.998162, 1.0)
a2	0.975	[0.023162, 0.998162)
a3	0.023162	[0.0 , 0.023162)

Encoding the string ***a2a2ala3a3*** produces the strange numbers (accurate to 16 digits) in Table 4, where the two rows for each symbol correspond to the **Low** and **High** values, respectively. At first glance, it seems that the ***resulting code is longer*** than the original string.

Decoding this string is shown in Table 5 involves a ***special problem***. After eliminating the effect of *a1*, on line 3, ***the result is 0***. Earlier, we implicitly assumed that ***this means the end of the decoding process*****,** but now we know that there are two more occurrences of *a3* that should be decoded. These are shown on lines 4, 5 of the table. ***This problem always occurs when the last symbol in the input stream is the one whose subrange starts at zero***.

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

Table 4. Encoding the String a2a2a1a3a3

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	$0.0 - 0.0$	$= 0.0$	$/0.023162$	$= 0.0$

Table 5. Decoding the String a2a2a1a3a3

In order to distinguish between such a symbol and the end of the input stream, we need to define an additional symbol, the end-of-input (or end-of-file, ***eof***). ***This symbol should be added, with a small probability, to the frequency table (as the first symbol; see Table below), and it should be added and encoded at the end of the input stream.***

Symbols	probability	Range
eof	0.000001	[0.999999, 1.0)
a1	0.001837	[0.998162, 0.999999)
a2	0.975	[0.023162, 0.998162)
a3	0.023162	[0.0 , 0.023162)

Tables 6 and 7 show how the string ***a3a3a3a3eof*** is encoded into the number 0.0000002878086184764172, and then decoded properly. ***Without the eof*** symbol, a ***string of all a3s*** would have been ***encoded into a 0***.

a_3	$0.0 + (1.0 - 0.0) \times 0.0 = 0.0$
	$0.0 + (1.0 - 0.0) \times 0.023162 = 0.023162$
a_3	$0.0 + .023162 \times 0.0 = 0.0$
	$0.0 + .023162 \times 0.023162 = 0.000536478244$
a_3	$0.0 + 0.000536478244 \times 0.0 = 0.0$
	$0.0 + 0.000536478244 \times 0.023162 = 0.000012425909087528$
a_3	$0.0 + 0.000012425909087528 \times 0.0 = 0.0$
	$0.0 + 0.000012425909087528 \times 0.023162 = 0.0000002878089062853235$
eof	$0.0 + 0.0000002878089062853235 \times 0.999999 = 0.0000002878086184764172$
	$0.0 + 0.0000002878089062853235 \times 1.0 = 0.0000002878089062853235$

Table 6. Encoding the String a3a3a3a3eof

Char.	Code−low		Range
a_3	0.0000002878086184764172-0	=0.0000002878086184764172	/0.023162=0.000012425896661618912 47
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

Table 7. Decoding the String a3a3a3a3eof

Notice how the ***low value is 0 until the eof is input and processed***, and how the high value quickly approaches 0. Now is the time to mention that ***the final code does not have to be the final low value but can be any number between the final low and high values***. In the example of *a3a3a3a3eof,* the final code can be the much shorter number 0.0000002878086 (or 0.0000002878087 or even 0.0000002878088).

If the ***size*** of the ***input stream is known***, it is possible to do ***without an eof*** symbol. The encoder can start by writing this size (unencoded) on the output stream. The decoder reads the size, starts decoding, and ***stops*** when the decoded stream ***reaches this size***. If the decoder reads the compressed stream byte by byte, the encoder may have to add some zeros at the end, to make sure the compressed stream can be read in groups of 8 bits.