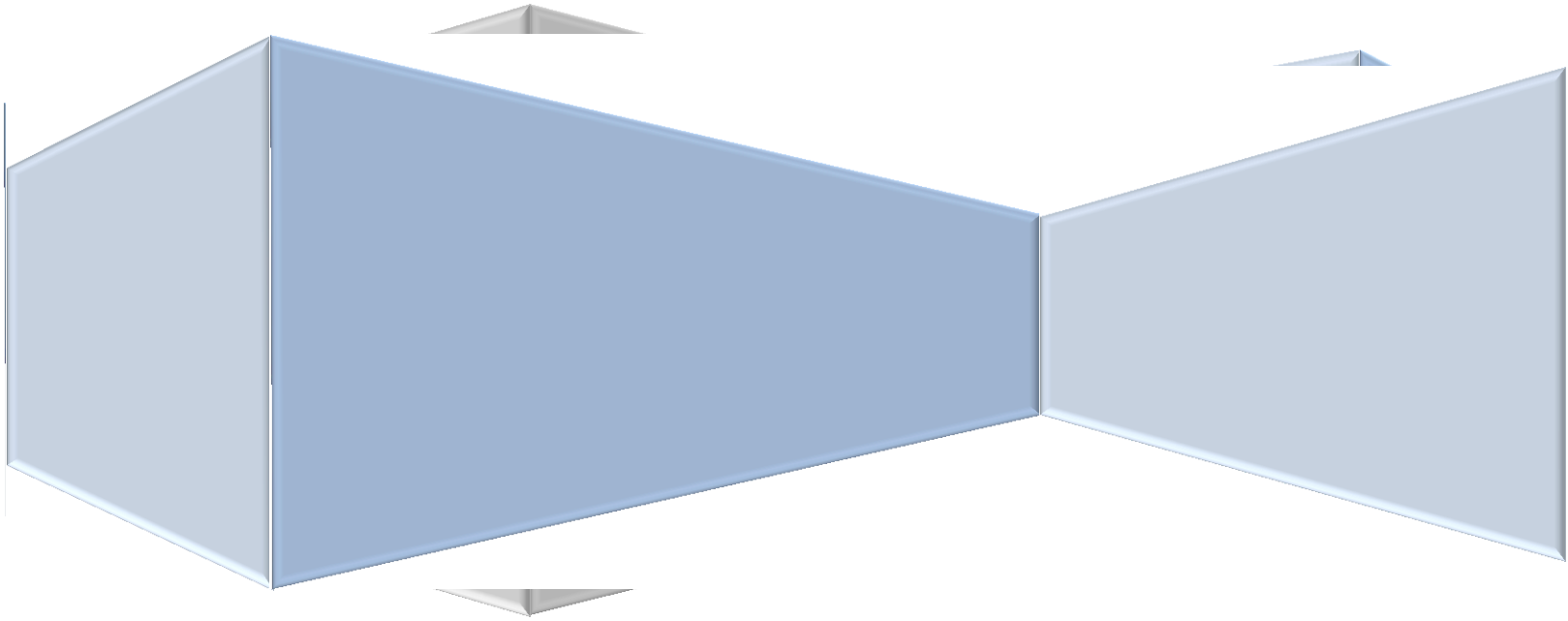
Data Compression

MSc. Amal S. Ajrash



Data Compression

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.

Data compression has come of age in the last 20 years; it is popular for 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.
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.

Data Compression: It is a key solution for space / time data transmission

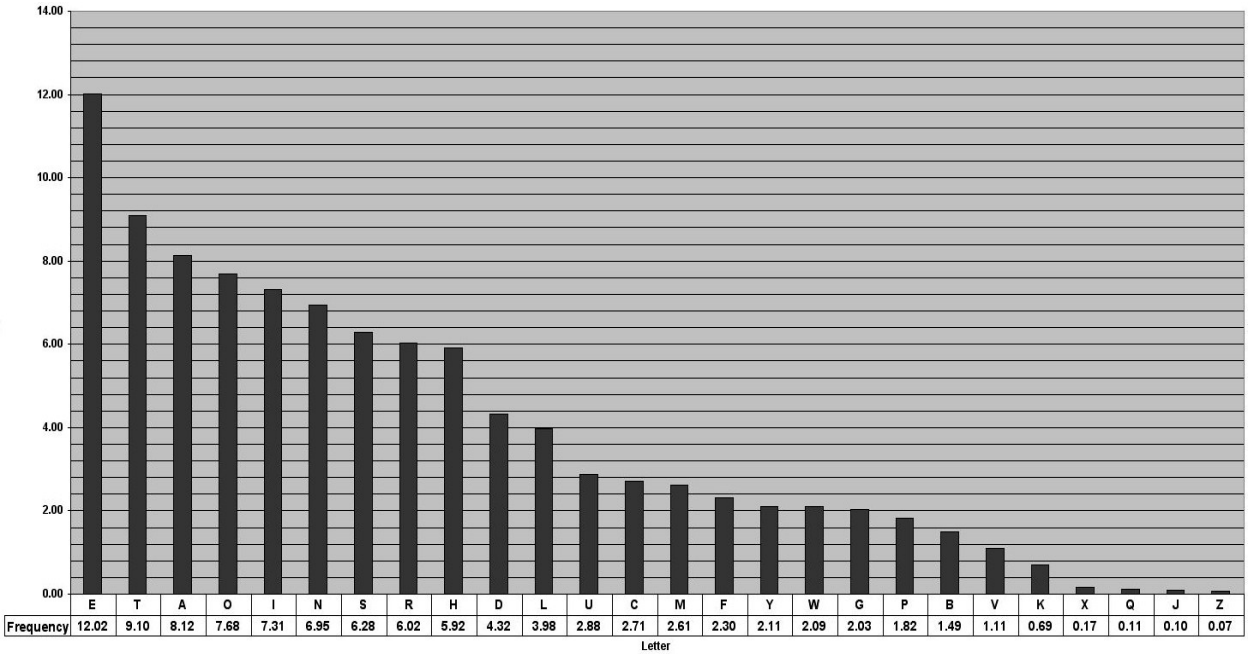
There are many known methods for data compression. They are based on different ***ideas***, are suitable for different types of data, and produce different results, but they are all based on the same principle, namely they compress data by ***removing redundancy*** from the original data in the source file.

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 getting the longest one.

English Letter Frequency (based on a sample of 40,000 words)

Letter	Count	Letter	Frequency
E	21912	E	12.02
T	16587	T	9.10
A	14810	A	8.12
O	14003	O	7.68

I	13318	I	7.31
N	12666	N	6.95
S	11450	S	6.28
R	10977	R	6.02
H	10795	H	5.92
D	7874	D	4.32
L	7253	L	3.98
U	5246	U	2.88
C	4943	C	2.71
M	4761	M	2.61
F	4200	F	2.30
Y	3853	Y	2.11
W	3819	W	2.09
G	3693	G	2.03
P	3316	P	1.82
B	2715	B	1.49
V	2019	V	1.11
K	1257	K	0.69
X	315	X	0.17
Q	205	Q	0.11
J	188	J	0.10
Z	128	Z	0.07



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).

- The most common diagrams (Two –character group) are “TH”, “IN”,”ER”,”AN”, and so on.
- The most common trigrams (three-character groups) are "THE", "AND", "THA", "ENT", and so on.

Redundancy in images is illustrated by the fact that in a nonrandom image, adjacent pixels tend to have similar colors.

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. In such text, each letter

occurs with ***equal probability***, so assigning them fixed-size codes does not add any redundancy. When such a file is compressed, there is no redundancy to remove.

- Another answer is that if it were possible 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 large file.

In order to compress a data file, the compression algorithm has to ***examine the data***, ***find redundancies in it***, and ***try to remove them***. Since the redundancies in data depend on the type of data (text, images, sound, etc.), any given compression method has to be developed for a specific type of data and performs best on this type. **There is no such thing as a universal, efficient data compression algorithm.**

There are many compression methods, some suitable for text and others for graphical data (still images or movies).

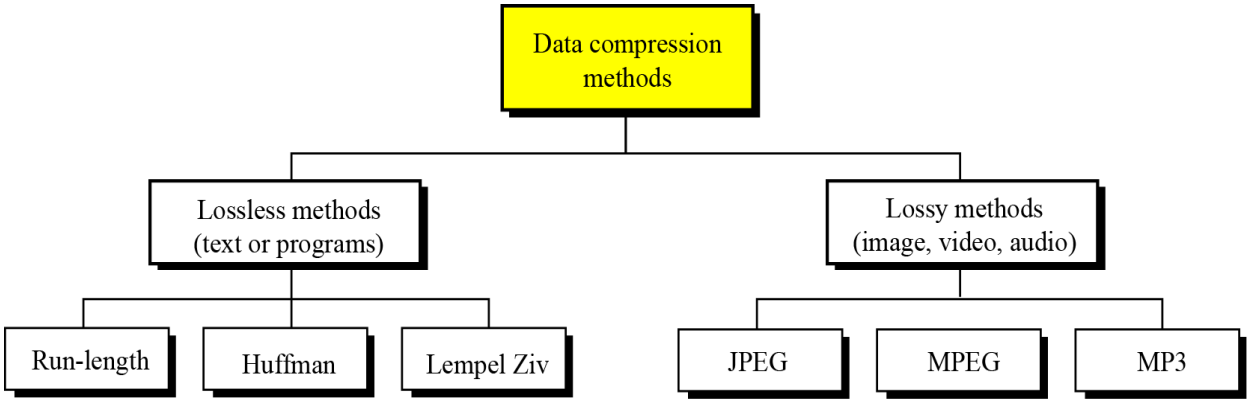
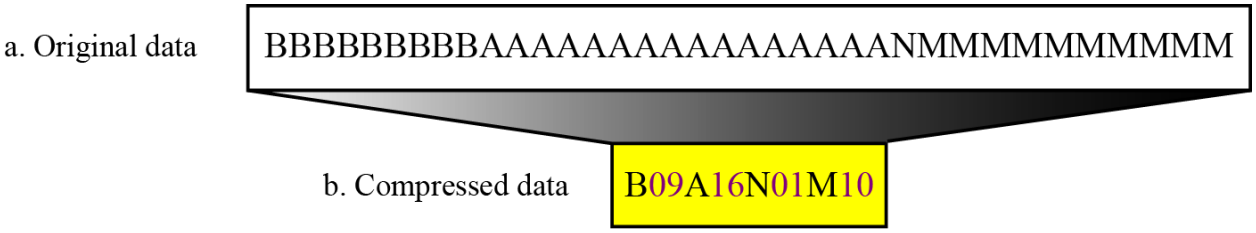


Figure_1: Data compression methods



Before delving into the details, we discuss important ***data compression terms***:-

1. 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. Note that the term encoding is very general and has wide meaning, but since we discuss only data compression, we use the name encoder to mean data compressor. The term codec is sometimes used to describe both the encoder and decoder. Similarly, the term ***companding*** is short for ***"compressing/expanding."***

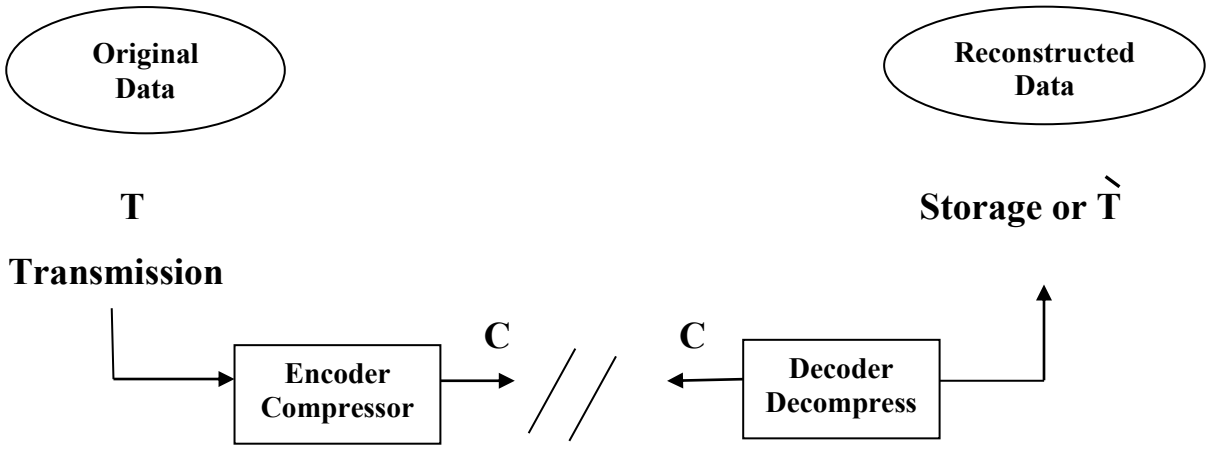


Figure _2 : Encoder /Decoder System (CODEC)

2. The term ***"stream"*** is used 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.
3. For the original input stream, we use the terms ***unencoded, raw, or original data***. The contents of the final, compressed, stream is considered the encoded or compressed data. The term ***bitstream*** is also used in the literature to indicate the compressed stream.

4. ***Lossless/Lossy compression*:**
   - In **lossless** data compression, the integrity of the data is preserved. ***The original data and the data after compression and decompression are exactly the same*** because, in these methods, the ***compression and decompression algorithms are exact inverses of each other***: ***no part of the data is lost in the process***. Redundant data is removed in compression and added during decompression. Lossless compression methods are normally used when we cannot afford to lose any data.
   - ***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**.** A lossy encoder must take advantage of the special type of data being compressed. It should delete only data whose absence would not be ***detected by our senses***, so it is often referred to as **a perceptive encoder**. ***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.
5. ***Symmetrical Compression: Symmetric compression*** method uses roughly the same algorithms, and performs the same amount of work, for compression as it does for decompression. For example, a data transmission application where compression and decompression are both being done on the fly will usually require a symmetric algorithm for the greatest efficiency. ***Asymmetric methods*** require substantially more work to go in one direction than they require in the other. Usually, the compression step takes far more time and system resources than the decompression step. In the real world this makes sense. ***For example, if we are making an image database in which an image will be compressed once for storage, but decompressed many times***

for viewing, then we can probably tolerate a much longer time for compression than for decompression. An asymmetric algorithm that uses much CPU time for compression, but is quick to decode, would work well in this case.

6. ***Adaptive, Semi-Adaptive, and Non-Adaptive Encoding :***Certain dictionary-based encoders are designed to compress only specific types of data. These ***non-adaptive*** encoders contain ***a static dictionary of predefined substrings*** that are known to occur with high frequency in the data to be encoded. A non-adaptive encoder designed specifically to compress English language text would contain a dictionary with predefined substrings such as "and", "but", "of", and "the", because these substrings appear very frequently in English text. ***An adaptive encoder,*** on the other hand, ***carries no preconceived heuristics about the data it is to compress***. Adaptive compressors, such as LZW, achieve data independence by building their dictionaries completely from scratch. They do not have a predefined list of static substrings and instead build phrases dynamically as they encode. ***Adaptive compression is*** capable of adjusting to any type of data input and of returning output using the best possible compression ratio. This is in contrast to non-adaptive compressors, which are capable of efficiently encoding only a very select type of input data for which they are designed. A mixture of these two dictionary encoding methods is the ***semi-adaptive encoding method***. A semi-adaptive encoder makes an initial pass over the data to build the dictionary and a second pass to perform the actual encoding. Using this method, an optimal dictionary is constructed before any encoding is actually performed.
7. ***Compression performance:*** Several quantities are commonly used to express the performance of a compression method.

1. The compression ratio is defined as :

$$CompressionRatio = \frac{size\ of\ the\ output\ stream}{size\ of\ the\ input\ stream}.$$

- A value of 0.6 means that the data occupies 60% of its original size after compression.
- Values greater than 1 mean an output stream bigger than the input stream (negative compression).
- The compression ratio can also be called bpb (bit per bit), since it equals the number of bits in the compressed stream needed, on average, to compress one bit in the input stream. In image compression, the same term, bpp stands for "bits per pixel." In modern, efficient text compression methods, it makes sense to talk about bpc (bits per character)—the number of bits it takes, on average, to compress one character in the input stream.

2. The inverse of the compression ratio is called the compression factor:

$Compressionfactor = \frac{sizeoftheinputstreamstream}{sizeoftheoutputstream}$.

- In this case, values 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.
- This measure is distantly related to the sparseness ratio. The expression 100 × (1 - compression ratio) is also a reasonable measure of compression performance. A value of 60 means that the output stream occupies 40% of its original size (or that the compression has resulted in savings of 60%).

3. Image Fidelity Criteria

- In lossy compression techniques, the decompressed image will not be identical to the original image. In such cases, we can define fidelity criteria that measure the difference between these two images. Fidelity criteria can be divided into two classes:

1) **Objective fidelity criteria**
2) **Subjective fidelity criteria**

The best instrument to measure image quality is the human eyes. Unfortunately, visual test are expensive to perform, so different statistical criteria can be used to measure the quality of the image.

1) The objective fidelity criteria:

The objective fidelity criteria are borrowed from digital signal processing and information theory and provide us with equations that can be used to measure the amount of distortion or error in the reconstructed (decompressed) image. Commonly used objective measures are:

- The mean square error (MSE)
- The signal to noise ratio (SNR)
- The peak signal to noise ratio (PSNR).

We can define the error between an original, uncompressed pixel value and the reconstructed (decompressed) pixel value as:

The mean square error is found by:

$$\mathbf{SNR} = \sqrt{\frac{\sum_{r=0}^{n-1}\sum_{c=0}^{n-1}\left[\ \bar{I}(r,c)\ \right]^2}{\sum_{r=0}^{n-1}\sum_{c=0}^{n-1}\left[\ \bar{I}(r,c) - I(r,c)\ \right]^2}}$$

The mean square error is found by:

$$MSE = \sqrt{1/N^2 \sum_{r=0}^{n-1} \sum_{c=0}^{n-1} [\ \bar{I}(r,c) - I(r,c)\ ]^2}$$

Another related metric, the peak signal to noise ratio, is defined as:

$$PSNR = 10\ Log_{10} \frac{(L-1)^2}{1/N^2 \sum_{r=0}^{n-1} \sum_{c=0}^{n-1} [\ \bar{I}(r,c) - I(r,c)\ ]^2}$$

Where L = the number of gray levels (e.g. for 8 bit, L=256)

The relationship between the SNR and MSE is ***reverse***. Its mean that when the MSE value of the image increase the SNR decrease and this will mean that the quality of the image is not good, but when the MSE value of the image decrease the SNR value will increase and in this case the quality of the image will be best.

These objective measures are often used in research because they are easy to generate and seemingly unbiased, but remember that these metrics are not necessarily correlated to our perception of an image.

2)The subjective fidelity criteria:

Subjective fidelity criteria require the definition of a qualitative scale to assess image quality. This scale can then be used by human test subjects to determine image fidelity.

Subjective testing is performed by creating a database of images to be tested, gathering a group of people that are representative of the desired population, and then having all the test subjects evaluate the images according to a predefined scoring criterion. The results are then analyzed statistically, typically using the averages and standard deviations as metrics. Subjective fidelity measures can be classified into three categories:

- The first type is referred to as impairment tests, where the test subjects score the images in terms of how bad they are.
- The second type is quality tests, where the test subjects rate the images in terms of how good they are.
- The third type is called comparison tests, where the images are evaluated on a side-by-side basis.

The comparison type tests are considered to provide the most useful results, as they provide a relative measure, which is the easiest metric for most people to determine.

Impairment and quality tests require an absolute measure, which is more difficult to determine in an unbiased fashion.

In Table (1) are examples of internationally accepted scoring scales for these three types of subjective fidelity measures.

Table (1) Subjective Fidelity Scoring Scales

Value	Rating	Description
1	Excellent	An image of extremely high quality, as good as you could desire.
2	Fine	An image of high quality, providing enjoyable viewing. Interference is not objectionable.
3	Passable	An image of acceptable quality. Interference is not objectionable.
4	Marginal	An image of poor quality; you wish you could improve it. Interference is somewhat objectionable.
5	Inferior	A very poor image, but you could watch it. Objectionable interference is definitely present.
6	Unusable	An image so bad that you could not watch it.

Data Compression Basic Techniques

Basic Techniques

The basic compression techniques are described here:

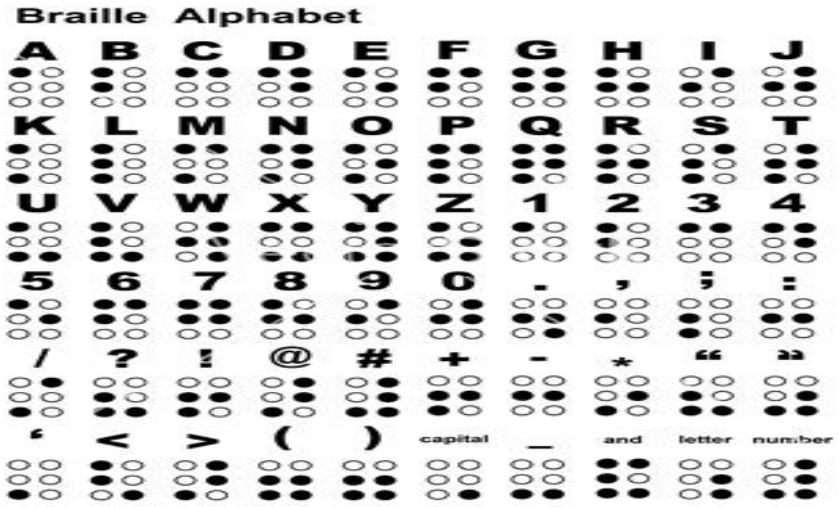
1. Intuitive Compression

This section discusses simple intuitive compression methods that have been used in the past. Today these methods are mostly of historical interest, since they are generally inefficient and cannot compete with the modern compression methods developed during the last several decades.

a. Braille code

This well-known code enables the blind to read. Developed by Louis Braille in the 1820s and is still in common use today.

Consists of groups (or cells) of 3 × 2 dots each embossed on thick paper. Each of the 6 dots in a group may be flat or raised, implying that the information content of a group is equivalent to 6 bits resulting in 64 possible groups.



b. Irreversible Text Compression

Sometimes it is acceptable to "compress" text by simply **throwing away some information**. This is called irreversible text compression or compaction. The decompressed text will not be identical to the original, so such methods are not general purpose; they can only be used in special cases.

c. Ad Hoc Text Compression

Here are some simple, intuitive ideas for cases where the compression must be reversible (lossless). If the text contains many spaces but they are not clustered, they may be removed and their positions indicated by a bit-string that contains a 0 for each text character that is not a space and a 1 for each space. Thus, the text: **Here are some ideas**
is encoded as the bit-string

00001000100001000000

followed by the text **Herearesomeideas**

2. Run-Length Encoding

The idea behind this approach to data compression is this: **If a data item d occurs n consecutive times in the input stream, replace the n occurrences with the single pair nd**. The n consecutive occurrences of a data item are called a run length of n, and this approach to data compression is called run-length encoding or RLE. We apply this idea first to text compression and then to image compression.

For example, consider a screen containing plain black text on a solid white background. There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text. A hypothetical scan line, with B representing a black pixel and W representing white, might read as follows:

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWW
WWWWWWWWWWWWBWWWWWWWWWWWWW

With a run-length encoding (RLE) data compression algorithm applied to the above hypothetical scan line, it can be rendered as follows:

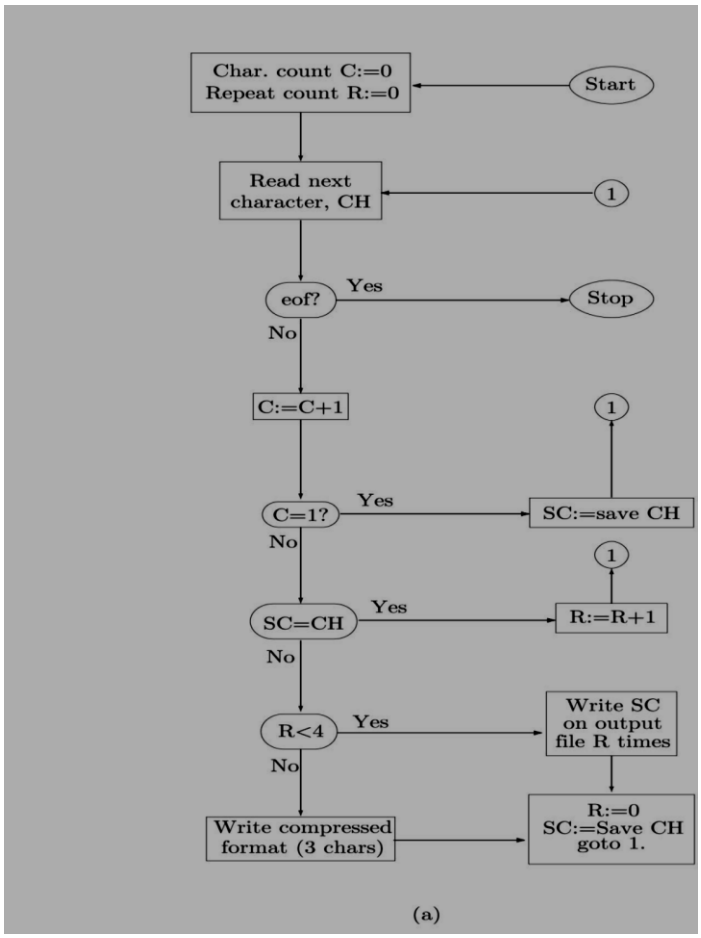
12W1B12W3B24W1B14W

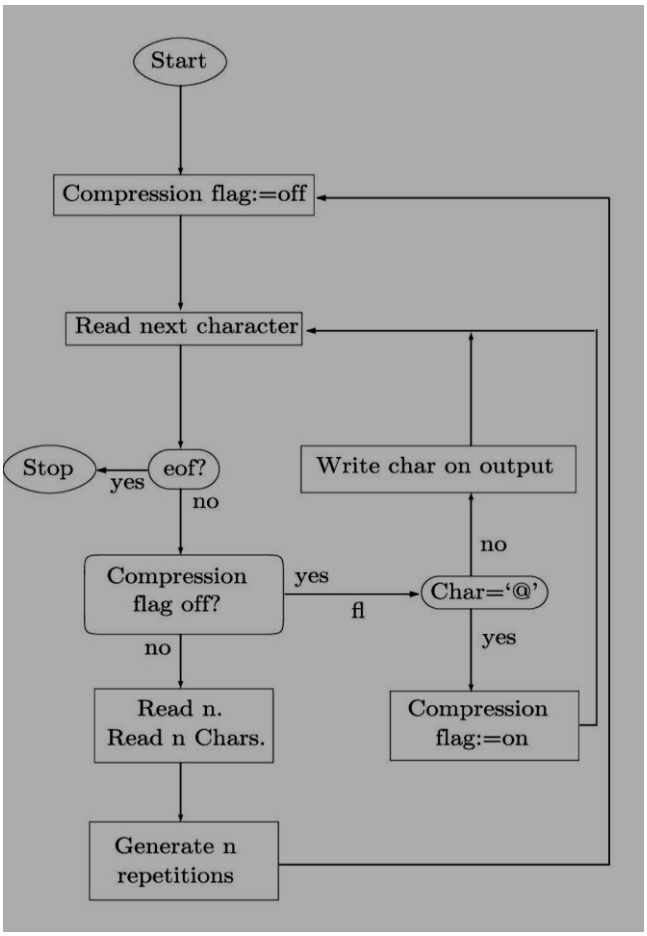
This can be interpreted as a sequence of twelve W, one B, twelve W, three B, etc.

a. RLE Text Compression

Just replacing 2._all_is_too_well with 2._a2_is_t2_we2 will not work. Even the string 2._a2l_is_t2o_we2l does not solve this problem. One way to solve this problem is to precede each repetition with a special escape character. If we use the character @ as the escape character, then the string 2._a@2l_is_t@2o_we@2l can be decompressed unambiguously. However, this string is longer than the original string, because it replaces two consecutive letters with three characters. We have to adopt the convention that only three or more repetitions of the same character will be replaced with a repetition factor. The main problems with this method are the following:

1. In English text there are not many repetitions. There are many “doubles”
but a “triple” is rare.
2. The character “@” may be part of the text in the input stream, in which case a different escape character must be chosen. Sometimes the input stream may contain every possible character in the alphabet.





RLE: a. Compression b. Decompression

b. RLE Image Compression

RLE can be used to compress grayscale images. Each run of pixels of the same intensity (gray level) is encoded as a pair **(run length, pixel value)**. The run length usually occupies one byte, allowing for runs of up to 255 pixels. The pixel value occupies several bits, depending on the number of gray levels (typically between 4 and 8 bits).

Example: An 8-bit deep grayscale bitmap that starts with

12, 12, 12, 12, 12, 12, 12, 12, 12, 35, 76, 112, 67, 87, 87, 87, 5, 5, 5, 5, 5, 5, 1, . . . is compressed into **9** ,12,35,76,112,67, **3** ,87, **6** ,5,1,. . . , where the bold numbers indicate counts.

The problem is to distinguish between a byte containing a grey scale value (suchas12) and one containing a count (such as 9).

Solution of RLE Problem

Here are some solutions (although not the only possible ones):

1. If the image is limited to just 128 grayscales, we can use one bit in each byte to indicate whether the byte contains a grayscale value or a count.

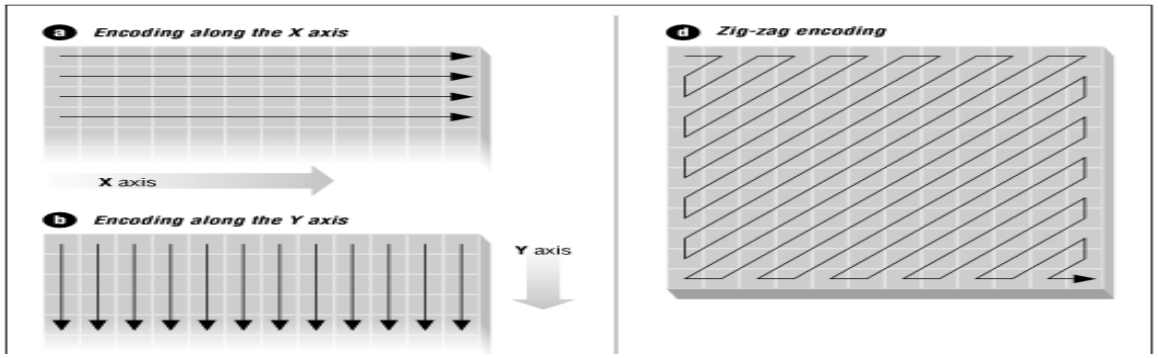
2. Put a 255in front of the count number. 255, 9, 12, 35, 76, 112, 67, 255, 3, 87, 255, 6, 5, 1

3. Use a byte to define the wherever the number is a count number or grayscale value. The count number will be marked by one and the rest is zero.

Example: the sequence 9,12,35,76,112,67, 3,87, 6,5,1,

becomes 10000010,9,12,35,76,112,67,3,87, 100..... ,6,5,1

RLE scanning



Disadvantage of image RLE

1. When the image is modified, the run lengths normally have to be completely redone.
2. The RLE output can sometimes be bigger than uncompressed image for complex image.

3. Imagine a picture with many vertical lines. When it is scanned horizontally, it produces very short runs, resulting in very bad compression.

4. A good, practical RLE image compressor should be able to scan the bitmap by rows, columns, or in zigzag and it may automatically try all three ways on every bitmap to achieve the best compression.

BMP Image Files

- BMP is the native format for image files in the Microsoft Windows operating system.
- BMP is a palette-based graphics file format for images with 1,2, 4, 8, 16, 24, or 32 bit planes.
- The image pixels, either in **raw format** or **compressed by RLE**.
- For images with eight bit planes, the compressed pixels are organized in pairs of bytes. The first byte of a pair is a **count C**, and the second byte is **a pixel value P** which is repeated C times.
- Thus, the pair 04 02 is expanded to the four pixels 02 02 02 02.

A count of 0 acts as an escape, and its meaning depends on the byte that follows:

1. **A zero byte followed by another zero** indicates end-of-line. The remainder of the current image row is filled with pixels of 00 as needed.
2. **A zero byte followed by 01** indicates the end of the image. The remainder of the image is filled up with 00 pixels.
3. **A zero byte followed by 02** indicates a skip to another position in the image. A 00 02 pair must be followed by two bytes indicating how many columns and rows to skip to reach the next nonzero pixel. Any pixels skipped are filled with zeros.

4. **A zero byte followed by a byte C greater than 2** indicates C raw pixels. Such a pair must be followed by the C pixels.

   Example:

   Assuming a 4×8 image with 8-bit pixels, the following sequence

   04 02, 00 04 a3 5b 12 47 , 01 f5, 02 e7, 00 02 00 01,01 99, 03 c1, 00 00, 00 04 08 92 6b d7 , 00 01

   Is the compressed representation of the 32 pixels

   02 02 02 02 a3 5b 12 47

   f5 e7 e7 00 00 00 00 00

   00 00 99 c1 c1 c1 00 00

   08 92 6b d7 00 00 00 00

3. Move-to-Front Coding

The basic idea of this method is to maintain the alphabet A of symbols as a list where frequently-occurring symbols are located near the front. A symbol s is encoded as the number of symbols that precede it in this list.

Thus if A=(t, h, e, s,...) and the next symbol in the input stream to be encoded is the e, it will be encoded as 2, since it is preceded by two symbols. There are several possible variants to this method; the most basic of them adds one more step: After symbol s is encoded, it is moved to the front of list A. Thus, after encoding the e, the alphabet is modified to A=(e, t, h, s,...). This move-to-front step reflects the expectation that once e

has been read from the input stream, it will be read many more times and will, at least for a while, be a common symbol.

The move-to-front method is locally adaptive, since it adapts itself to the frequencies of symbols in local areas of the input stream. The method produces good results if the input stream satisfies this expectation, i.e., if it contains concentrations of identical symbols (if the local frequency of symbols changes significantly from area to area in the input stream). We call this the concentration property. Here are the examples that illustrate the move-to-front idea. Assume the alphabet A=(a, b, c, d, m, n, o, p).

Example:

Compress this following string "DDBBAD"

A= {a, b, c, d}

Set A=

A	B	C	D
0	1	2	3

1)Read the first character "D" the index = 3, and move the D to the front of the list

D	A	B	C
0	1	2	3

2)Read the second character "D" the index = 3,0 and D is already in the front of the list.

D	A	B	C
0	1	2	3

3) Read the next character “B” the index =3,0,2 and move “B” to the front of the list.

B	D	A	C
0	1	2	3

4)Read the next character “B” the index = 3,0,2,0 and B is already in the front of the list.

B	D	A	C
0	1	2	3

5) Read the next character “A” the index =3,0,2,0,2 and move “A” to the front of the list.

A	B	D	C
0	1	2	3

6) Read the next character “D” the index =3,0,2,0,2, 2 and move “D” to the front of the list.

D	A	B	C
0	1	2	3

The Move to Front code is 3,0,2,0,2,2 for input string “DDBBAD”

In decompressing the same procedure will be performed but instead of reading the text the compression code will be read and make a mapping to the set A.

1)Compression code is 3,0,2,0,2,2

A	B	C	D
0	1	2	3

2)Read the first number 3 and make a mapping to the set A, the text= D and move D to the front.

D	A	B	C
0	1	2	3

3)Continue with the same procedure, until all code have been read the output will be string "DDBBAD"

4. Scalar Quantization

The dictionary definition of the term "quantization" is "to restrict a variable quantity to discrete values rather than to a continuous set of values."

In the field of data compression, quantization is employed in:

- If the data to be compressed is in the form of large numbers, quantization is used to convert it to small numbers. Small numbers take less space than large ones, so quantization generates compression. On the other hand, small numbers generally contain less information than large numbers, so quantization results in lossy compression.
- **Quantization:** Is a process of representing a large set of values with a smaller set of values.
- **Quantization** can be considered as **lossy compression**. Where it is easy to control the trade-off between **compression ratio** and the **amount of loss**.

For example is naive discrete quantization of an input stream of 8-bit numbers. We can simply delete the least-significant four bits of each data item. This method is simple but not very practical because too much information is lost.

Disadvantage of Scalar Quantization

- Because it is so simple, its use is limited to cases where much loss can be tolerated.
- Many image compression methods are lossy, but scalar quantization is not suitable for image compression because it creates annoying artifacts in the decompressed image.

This is why practical algorithms use vector quantization, instead of scalar quantization, for lossy (and sometimes lossless) compression of images and sound.

5. The Burrows-Wheeler Algorithm

- The algorithm consists of several stages and these stages are performed successively. With the decompression of the algorithm the data which compressed by the algorithm can be returned to their original data. Burrows-Wheeler (BW) method works in a block mode, where the input stream is read block by block and each block is encoded separately as one string. The method is therefore referred to as block sorting. The BW method is general purpose, it works well on images, sound, and text, and can achieve very high compression ratios (1 bit per byte or even better).

Structure of the Burrows-Wheeler

- The original algorithm consists of three stages:
  - The First Stage of the algorithm is the Burrows-Wheeler Transform. The aim of the first stage is to sort the characters of the input with the result that identical characters are close together, or better to sequences of identical characters.
  - The Second Stage of the algorithm is the Move-To-Front Transform. In this stage, the characters of the input which are still in a local context get assigned a global index value.
  - The Third Stage is the Entropy Coding. In this stage the real compression of the data takes place. Normally, the human coding is used as entropy coder.
- Process steps:

1. Order the input n (n is the length of the input) times among themselves, thereby rotate each row one character to the right compared to the previous row.

2. Sort the rows lexicographical.

The output of this stage is the L-column (we call L-column for the last column and F-column for the first column of the sorted matrix) and the index value of the sorted matrix that contains the original input.

- The procedure step by step with PANAMA as example input:

Step 1:

Index	F-column					L-column
0	P	A	N	A	M	A
1	A	P	A	N	A	M
2	M	A	P	A	N	A
3	A	M	A	P	A	N
4	N	A	M	A	P	A
5	A	N	A	M	A	P

Step 2:

Index	F-column					L-column
0	A	M	A	P	A	N
1	A	N	A	M	A	P
2	A	P	A	N	A	M
3	M	A	P	A	N	A
4	N	A	M	A	P	A
5	P	A	N	A	M	A

Output: $NPMAAA$ | 5

Move-To-Front Transform

- Process steps:

1. Save the index value of the global list Y which contains the first character of the input.

2. Move the saved character of the previous step in the global list on index position 0 and move all characters one position to the right which are located in the global list before the old position of the saved character

3. Repeat step 1 and 2 sequentially for the other characters of the input and use for all repetitions the modified global list from the previous repetition. The output of this stage consists of all saved index positions and the index value of the sorted matrix from the Burrows-Wheeler Transform which contains the original input.

- The procedure step by step with NPMAAA 5 (output of the example from the Burrows-Wheeler Transform) as example input. Use Y = [A, M, N, P] as global list.

Step 1:

Input: $\underline{N}$PMAAA

$Y = [A, M, \underline{N}, P] \Rightarrow$ *Save index position 2*

Step 2:

$Y = [A, M, \underline{N}, P] \Rightarrow Y = [\underline{N}, A, M, P]$

Step 3:

$Y = [N, A, M, \underline{P}] \Rightarrow$ *Save index position 3* $\Rightarrow Y = [\underline{P}, N, A, M]$

$Y = [P, N, A, \underline{M}] \Rightarrow$ *Save index position 3* $\Rightarrow Y = [\underline{M}, P, N, A]$

$Y = [M, P, N, \underline{A}] \Rightarrow$ *Save index position 3* $\Rightarrow Y = [\underline{A}, M, P, N]$

$Y = [\underline{A}, M, P, N] \Rightarrow$ *Save index position 0* $\Rightarrow Y = [\underline{A}, M, P, N]$

$Y = [\underline{A}, M, P, N] \Rightarrow$ *Save index position 0* $\Rightarrow Y = [\underline{A}, M, P, N]$

Output: $\boxed{233300}\boxed{5}$

- Output of this stage is the input for the next stage, the Entropy Coding.
- Entropy Coding using Zero Run-Length Coding

Example:

1. Increase all characters of the input which are greater as 0 by 1

Input: 233300 ⇒ 344400

2. Coding the sequences of zeros with string combinations of 0 and 1

We use at this point an example coding table.

Number of zeros	Coding string
1	0
2	1
3	00
4	01
5	10
6	11

344400 ⇒ Output: 34441

Notice from example

- In this example that the length of the output is one character smaller as the length of the input.
- For a larger input we can more profit from the Zero Run-Length Coding, because for example, we can coded 6 zeros with only 2 characters.

➢ **Decompression of Burrows-Wheeler Algorithm**

- During the decompression of the data the stages of the algorithm will be run in the reverse order compared to the compression.
- The technique of the Move-To-Front Back transform is analog to the Move-To-Front Transform of the compression with the different that there are index values instead of characters as input.
- The following code 233300 5 (output from the example of the Move-To-Front Transform) as example input and the output for this example is NPMAAA 5.
- The output of this stage is the input for the Burrows-Wheeler Back transform

6. Block Truncation Coding (BTC)

- The principle used by the block truncation coding (BTC) method and its variants is to quantize pixels in an image while preserving the first two or three statistical moments.
- In the basic BTC method, the image is divided into blocks (normally 4×4 or 8×8 pixels each).
- Assuming that a block contains n pixels with intensities p1 through pn, the first two moments are the mean and variance, defined as

$$\bar{p} = \frac{1}{n}\sum_{i=1}^{n} p_i,$$

$$\overline{p^2} = \frac{1}{n}\sum_{i=1}^{n} p_i^2,$$

- The standard deviation of the block is

$$\sigma = \sqrt{\overline{p^2} - \bar{p}}.$$

- The principle of the quantization is to select three values
- a threshold pthr
- a high value p+
- a low value p−

$$p_i \leftarrow \begin{cases} p^+, & \text{if } p_i \geq p_{\text{thr}}, \\ p^-, & \text{if } p_i < p_{\text{thr}}. \end{cases}$$

- Each pixel is replaced by either p+ or p−
- n+ the number of pixels in the current block that are greater than or equal to the threshold
- n− stands for the number of pixels less than the threshold
- The sum n+ + n− equals the number of pixels n in the block
- Preserving the first two moments is expressed by the two equations

$$n\bar{p} = n^- p^- - n^+ p^+, \quad n\overline{p^2} = n^-(p^-)^2 - n^+(p^+)^2.$$

$$p^- = \bar{p} - \sigma\sqrt{\frac{n^+}{n^-}}, \quad p^+ = \bar{p} + \sigma\sqrt{\frac{n^-}{n^+}}.$$

- These solutions are generally real numbers, but they have to be rounded to the nearest integer.

Example:

We select the 4×4 block of 8-bit pixels

$$\begin{pmatrix} 121 & 114 & 56 & 47 \\ 37 & 200 & 247 & 255 \\ 16 & 0 & 12 & 169 \\ 43 & 5 & 7 & 251 \end{pmatrix}$$

The mean is $\bar{p} = 98.75$

We count $n+ = 7$ pixels greater than the mean and $n- = 16 - 7 = 9$ pixels less than the mean

The standard deviation is $\sigma = 92.95$

The high and low values are $$p^+ = 98.75 + 92.95\sqrt{\frac{9}{7}} = 204.14, \quad p^- = 98.75 - 92.95\sqrt{\frac{7}{9}} = 16.78.$$

They are rounded to 204 and 17

The resulting block is $$\begin{pmatrix} 204 & 204 & 17 & 17 \\ 17 & 204 & 204 & 204 \\ 17 & 17 & 17 & 204 \\ 17 & 17 & 17 & 204 \end{pmatrix}$$

It is clear that the original 4×4 block can be compressed to the 16 bits $$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

plus the two 8-bit values 204 and 17; a total of 16 + 2×8 bits, compared to the 16×8 bits of the original block. The compression factor is

$$\frac{16 \times 8}{16 + 2 \times 8} = 4.$$

7. Statistical compression methods

7.1.Prefix Codes:

A prefix code is a variable-size code that satisfies the prefix property.

Disadvantage:

1. The binary representation of the integers does not satisfy the prefix property.

2. The size n of the set of integers has to be known in advance, since it determines the code size, which is 1 + [log2 n]. In some applications, a prefix code is required to code a set of integers whose size is not known in advance.

7.2Tunstall Code

The main advantage of variable-size codes is their variable size. Some codes are short, and it is this feature that produces compression. On the downside, variable-size codes are difficult to work with. It is definitely easier to deal with fixed-size codes, and the Tunstall codes described here are one example of how such codes can be designed.

The idea is to construct a set of fixed-size codes, each encoding a variable-size string of input symbols. Variable-to-fixed length source code

Suppose an alphabet consisting of two symbols A and B where A is more common. Given a typical string from this alphabet, we expect substrings of the form AA, AAA, AB, AAB, and B, but rarely strings of the form BB. We can therefore assign fixed-size codes to the following five substrings as follows. AA = 000, AAA = 001, AB = 010, ABA = 011, and B = 100.

This example is both bad and inefficient. It is bad, because AAABAAB can be encoded either as the four codes AAA, B, AA, B or as the three codes AA, ABA, AB; encoding is not unique. This happens because our five substrings don't satisfy the prefix property. This example is inefficient because only five of the eight possible 3-bit codes are used. An n bit Tunstall code should use all 2^n codes.

An algorithm is needed in order to develop the best n-bit Tunstall code for a given alphabet of M symbols and such an algorithm is Tunstall code.

- **Tunstall code Properties**

1. No input code is a prefix of another to assure unique encodability.
2. Minimize the number of bits per symbol.

- **The Tunstall algorithm**

Consider we want to encode an alphabet source S = {s1, s2, ..., sm} with m symbols with probability P(s1), P(s2), ..., P(sm). We start with a code table that consists of the symbols. We then iterate as long as the size of the code table is less than or equal to the number of codes 2n. Each iteration performs the following steps:

(i) Arrange the m source symbols in ***descending*** order of ***probability*** (s1, s2, ..., sm) with P(s1) >=P(s2) >=...>= P(sm). Select the symbol with ***largest probability*** in the table. Call it S.

(ii) ***Remove*** S and create a new source with 2m –1 symbols by ***splitting*** the symbol with the largest probability into m symbols with probabilities: P(s1)2, P(s1)P(s2), ..., P(s1)P(sm). Label the new symbols obtained as s1s1, s1s2, ..., s1sm.

(iii) Repeat step (ii) until ***the size of the code table is less than or equal to the number of codes 2n.***

(iv) Assign an ***equal-length binary sequence to eachnew source symbol.***

(v) Use each binary sequence to encode those sequences emitted by the original source which correspond to the labels of the new source symbols.

Example:

Given an alphabet with the three symbols ***A, B, and C (M = 3)***, with probabilities ***0.7, 0.2, and 0.1***, respectively, we decide to construct a set of 3-bit Tunstall codes (thus, n = 3). We start our code table as a tree with a root and three children (Figure 2.8a).

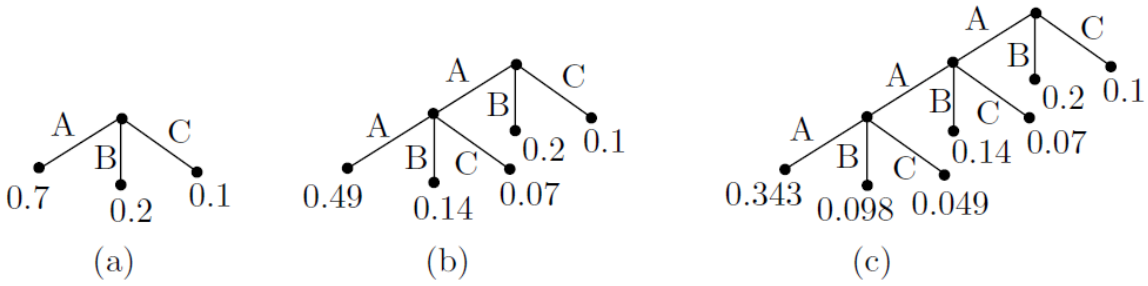


Figure 2.8: Tunstall Code Example.

1. In the first iteration, we select A and turn it into the root of a subtree with children AA, AB, and AC with probabilities ***0.49 (P(A)2),*** 0.14***(P(A)*P(B))*** , and 0.07 ***(P(A)*P(C))*** , respectively (Figure 2.8b).
2. The ***largest*** probability in the tree is that of node AA, so the second iteration converts it to the root of a subtree with nodes AAA, AAB, and AAC with probabilities 0.343, 0.098, and 0.049, respectively (Figure 2.8c).
3. After each iteration we count the number of codes in the tree.
4. After the second iteration there are seven codes in the tree, so the loop stops.

Seven 3-bit codes are arbitrarily assigned to elements AAA, AAB, AAC, AB, AC, B, and C. ***The eighth available code should be assigned to a substring that has the highest probability and also satisfies the prefix property***.

- **Bit Rate of Tunstall**

  • The length of the output code divided by the average length of the input code.

• Let ***pi*** be the ***probability*** of, and ***Li*** the ***length of input code i*** (length of tree node i) and let n be the length of the output code.

$$\text{avergae bit rate} = \sum_{i=1}^{m} \frac{n}{p_i l_i}$$

The ***average bit length*** of this code is easily computed as:

$$\frac{3}{3(0.343+0.098+0.049)+2(0.14+0.07)+0.2+0.1} = 1.37.$$

- **Tunstall Code advantage and disadvantage**

An important ***property*** of the Tunstall codes is their ***reliability***. If one bit becomes corrupt, only one code will get bad (error is restricted only to that code word), and the ***error does not propagate*** as happens in fixed-to-variable length codes (such as the Huffman code), normally, ***variable-size codes*** do not feature any reliability.

One ***disadvantage*** of the Tunstall algorithm is that it ***does not achieve*** the ***same coding efficiency*** as the Huffman code for the same complexity.

8. Shannon- Fano compression algorithm

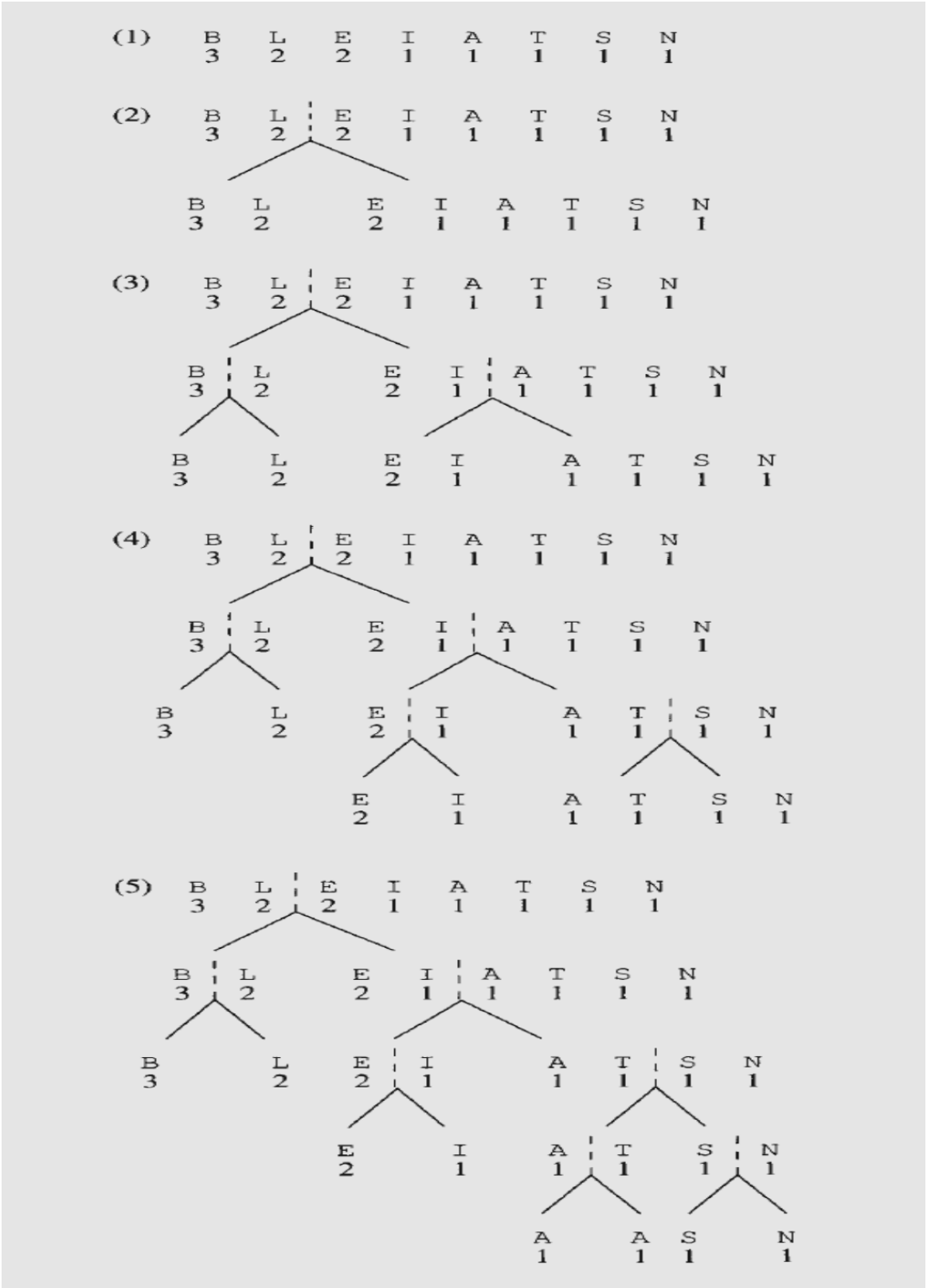
Shannon–Fano coding, is a technique for constructing a code based on a set of symbols and their probabilities (estimated or measured). It is suboptimal in the sense that it does not achieve the lowest possible expected code word length like Huffman coding; however unlike Huffman coding, it does guarantee that all code word lengths are within one bit of their theoretical ideal $-\log P(x)$. It Was the first algorithm to construct a set of the best variable-size codes.

Compression algorithm

This can be further refined to Shannon-Fano(S) algorithm.

Algorithm 4.5 Shannon-Fano encoding ideas

```
1: if S consists of only two symbols then
2:    add 0 to the codeword of the first symbol and 1 to the second
3: else
4:    if S has more than three symbols then
5:       divide S into 2 subsequence S1 and S2 with minimum probability dif-
         ference
6:       add a 0 to extend the codeword for each symbol in S1 and a 1 to those
         in S2
7:       Shannon-Fano(S1)
8:       Shannon-Fano(S2)
9:    end if
10: end if
```


(1)
B L E I A T S N
3 2 2 1 1 1 1 1
(2)
B L E I A T S N
3 2 2 1 1 1 1 1
B L E I A T S N
3 2 2 1 1 1 1 1
(3)
B L E I A T S N
3 2 2 1 1 1 1 1
B L E I A T S N
3 2 2 1 1 1 1 1
B L E I A T S N
3 2 2 1 1 1 1 1
(4)
B L E I A T S N
3 2 2 1 1 1 1 1
B L E I A T S N
3 2 2 1 1 1 1 1
B L E I A T S N
3 2 2 1 1 1 1 1
E I A T S N
2 1 1 1 1 1
(5)
B L E I A T S N
3 2 2 1 1 1 1 1
B L E I A T S N
3 2 2 1 1 1 1 1
B L E I A T S N
3 2 2 1 1 1 1 1
E I A T S N
2 1 1 1 1 1
A T S N
1 1 1 1
A A S N
1 1 1 1

- **Saving percentage**

Shannon Fano method produces better code when the splits are better, i.e., when the two subsets in every split have very close total probabilities. Carrying this argument to its limit suggests that perfect splits yield the best code.

See examples:-

	Prob.	Steps					Final
1.	0.25	1	1				:11
2.	0.20	1	0				:10
3.	0.15	0		1	1		:011
4.	0.15	0		1	0		:010
5.	0.10	0		0		1	:001
6.	0.10	0		0		0	1 :0001
7.	0.05	0		0		0	0 :0000

Table 2.14: Shannon-Fano Example.

The average size of this code is 0.25*2 +0.20*2 +0.15*3 + 0.15*3 + 0.10*3 + 0.10*4 + 0.05*4 = 2.7 bits / symbol

This is good result because the entropy (the smallest number of bits needed, on average, to represent each symbol) is

$$-(0.25\log_2 0.25 + 0.20\log_2 0.20 + 0.15\log_2 0.15 + 0.15\log_2 0.15 + 0.10\log_2 0.10 + 0.10\log_2 0.10 + 0.05\log_2 0.05) \approx 2.67.$$

What if we divide tree between third and fourth elements?

1.	0.25	1	1	1
2.	0.20	1	0	1
3.	0.15	1	0	0
4.	0.15	0	1	1
5.	0.10	0	1	0
6.	0.10	0	0	1
7.	0.05	0	0	0

0.25*3 + 0.20*3 + 0.15*3 +0.15*3 + 0.10*3 + 0.10*3 + 0.05*3 = 3

The code in the answer has longer average size because the split in this case not as good as those in the previous solution, this suggest that the Shannon- Fano method produces better code when split are better (when the two subset in every split have very close to the total probabilities).

- **Advantages of Shannon-Fano**

The algorithm produces fairly efficient variable-length encodings; when the two smaller sets produced by a partitioning are in fact of equal probability

- **Disadvantages of Shannon-Fano**

1) In Shannon-Fano coding, we cannot be sure about the codes generated. There may be two different codes for the same symbol depending on the way we build our tree.
2) Also, here we have no unique code i.e a code might be a prefix for another code. So in case of errors or loss during data transmission, we have to start from the beginning.
3) Shannon-Fano coding does not guarantee optimal codes.

9. Static Huffman coding

Huffman coding is a popular method for data compression. It serves as the basis for several popular programs run on various platforms. Some programs use just the Huffman method, while others use it as one step in a multistep compression process. The Huffman method is somewhat similar to the Shannon-Fano method. It generally produces better codes, and like the Shannon-Fano method, it produces the best code when the probabilities of the symbols are negative powers of 2. The main difference between the two methods is that Shannon-Fano constructs its codes top to bottom (from the leftmost to the rightmost bits), while Huffman constructs a code tree from the bottom up (builds the codes from right to left). The algorithm starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every

leaf, from the bottom up. This is done in steps, where at each step the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codes of the symbols.

Algorithm 4.2 Huffman encoding

INPUT: a sorted list of one-node binary trees $(t_1, t_2, \cdots, t_n)$ for alphabet $(s_1, \cdots, s_n)$ with frequencies $(w_1, \cdots, w_n)$

OUTPUT: a Huffman code with n codewords

1: initialise a list of one-node binary trees $(t_1, t_2, \cdots, t_n)$ with weight $(w_1, w_2, \cdots, w_n)$ respectively
2: **for** $k = 1; k < n; k = k + 1$ **do**
3: take two trees t_i and t_j with minimal weights $(w_i \leq w_j)$
4: $t \leftarrow merge(t_i, t_j)$ with weight $w \leftarrow w_i + w_j$,
where $left\_child(t) \leftarrow t_i$ and $right\_child(t) \leftarrow t_j$
5: $edge(t, t_i) \leftarrow 0$; $edge(t, t_j) \leftarrow 1$
6: **end for**
7: output every path from the root of t to a leaf, where $path_i$ consists of consecutive edges from the root to $leaf_i$ for s_i

Example: - build a Huffman coding for the text :

BLEIATSN

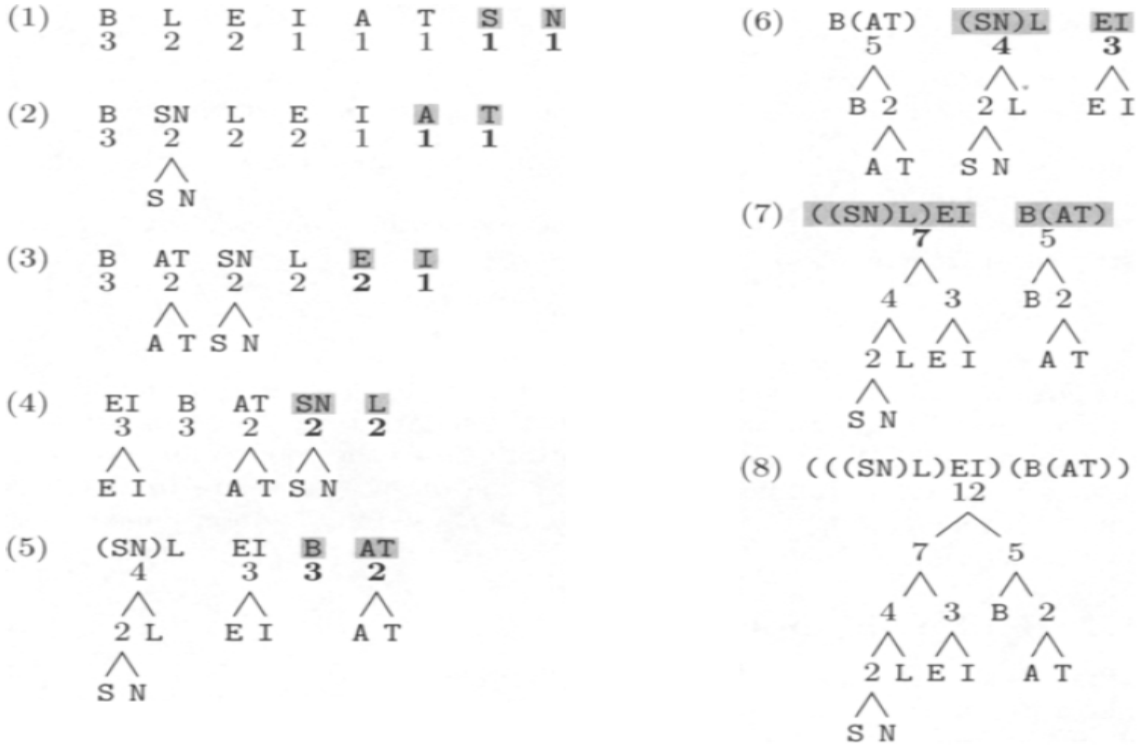


Figure 4.1: Building a Huffman tree

- **Saving Percentage**

consider the first example, with alphabet (B, L, E, I, A, T, S, N) with frequent (3,2,2,1,1,1,1,1). Huffman code (10,001,010,001,110,111,0000,0001) is derived with length (2,3,3,3,3,3,4,4) the total number of bits required by source BIL BEATS BEN 2*3+ 3*2*2 + 3*1*3 +4*1*2 =35 compare with 8 bits ASCII coding the source will be 8*12 =96 bits.

Huffman	ASCII/EBCDIC	Saving bits	Percentage
35	96	61	63.5%
		$96 - 35 = 61$	$61/96 = 63.5\%$

- **Advantages of Huffman coding**

1. Maximum compression ratio assuming correct probabilities of occurrences.
2. Easy to implement and fast
3. Not like run length encoding that work to compress only contiguous runs of symbols.
4. More accurate and clear than Shannon-Fano coding.

- **Disadvantages of Huffman coding**

1. Need two processes to encode / decode.
    - One create frequencies distributions.
    - One encode / decode data.
2. Encode compress one symbol each time.
3. What if the probabilities ensemble change with time or if we don't know probabilities a priori?
4. The extra bit $n = \sum p\, n \; < h(X) + 1$

   If h(x) is large then effect of (+1) is small, what if h(x) <1 bit ? so Huffman encoding uses at least one bit/ character encoding of blocks has its own problems.

References

1. Salomon, D. , Handbook of Data Compression, Fifth Edition, 2010.
2. Salomon, D., Data Compression the Complete Reference, Springer, 2007.
3. Gonzalez, R. , Digital Image Processing, Third Edition, 2008.
4. http://tcs.rwth-aachen.de/lehre/Komprimierung/SS2012/ausarbeitungen/Burrows-Wheeler.pdf
5. http://www.sps.ele.tue.nl/members/T.J.Tjalkens/info2/dictaten/info2.pdf