What is Data Compression?

Data compression requires the **identification** and **extraction** of **source redundancy**.

In other words, data compression seeks to **reduce the number of bits** used **to store** or **transmit information**.

There are wide ranges of compression methods that can be so unlike one another that they have little in common except that they compress data.

Data compression can be divided into **two main types**; **loss less** and **lossy** compression.

Lossless compression can recover the **exact original data** after compression. It is used mainly for compressing database records, spreadsheets or word processing files, where exact replication of the original is essential.

Lossy compression will result in a **certain loss of accuracy** in exchange for a substantial increase in compression. Lossy compression is more effective when used to compress graphic images and digitized voice where losses outside visual or aural perception can be tolerated. Most lossy compression techniques can be adjusted to different quality levels, gaining **higher accuracy** in exchange for **less effective compression**.

The **amount of compression** that can be achieved by a given algorithm depends on both the **amount of redundancy** in the **source** and the **efficiency** of its **extraction**.

The Need For Compression

In the past, storing **documents** were stored on **paper** and kept in **filing cabinets** have been very **inefficient** in terms of **storage space** and also the time taken to **locate** and **retrieve information** when required. Storing and accessing documents electronically through computers are now replacing this traditional method of storing documents. This has enabled us to manage things more **efficiently** and **effectively**, so that items can be located and **information extracted** without undue expense or inconvenience.

In terms of **storage**, the **capacity of a storage device** can be effectively increased with methods that **compress a body of data** on its way to a storage device and **decompresses** it when it is retrieved.

In terms of **communications**, compressing data at the sending end and decompressing data at the receiving end can effectively increase the bandwidth of a digital communication link.

At any given time, the ability of the **Internet** to transfer data is fixed. Thus, if data can effectively be **compressed** wherever possible, significant **improvements of data throughput** can be achieved. Many files can be **combined** into one **compressed document** making sending easier.
In **computer graphics**, we are interested in reducing the **size** of a **block of graphics** data so we can fit more information in a given physical **storage space**.

A Brief History of Data Compression

The late 40's were the early years of **Information Theory**; the idea of developing efficient new coding methods was just starting to be fleshed out. Ideas of **entropy**, **information content** and **redundancy** were explored. One popular notion held that if the probability of symbols in a message were known, there ought to be a way to code the symbols so that the message will take up less space.

The first well-known method for compressing digital signals is now known as **Shannon-Fano coding**. Shannon and Fano [~1948] simultaneously developed this algorithm that assigns binary codeword to unique symbols that appear within a given data file. While Shannon-Fano coding was a great leap forward, it had the unfortunate luck to be quickly superseded by an even more efficient coding system: Huffman Coding.

Huffman coding [1952] shares most characteristics of Shannon-Fano coding. Huffman coding could perform effective data compression by reducing the amount of redundancy in the coding of symbols. It has been proven to be the most efficient fixed-length coding method available.

In the last fifteen years, Huffman coding has been replaced by arithmetic coding. **Arithmetic coding** bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point output number. More bits are needed in the output number for longer, complex messages.

Dictionary-based compression algorithms use a completely different method to compress data. They encode variable-length strings of symbols as single ***tokens***. The token forms an **index** to a **phrase dictionary**. If the **tokens** are **smaller** than the **phrases,** they replace the phrases and compression occurs.

Two dictionary-based compression techniques called **LZ77** and **LZ78** have been developed. **LZ77** is a "**sliding window**" technique in which the dictionary consists of a set of **fixed-length phrases** found in a "window" into the **previously** seen text. LZ78 takes a completely different approach to building a dictionary. Instead of using fixed-length phrases from a window into the text, **LZ78 builds phrases up one symbol at a time**, **adding** a **new symbol** to an **existing phrase** when a **match occurs**.

2.1 Run Length Encoding (RLE)

Run-length Encoding, or **RLE** is a technique used to **reduce the size** of a **repeating string of characters**. This repeating string is called a ***run***, typically RLE encodes a **run of symbols** into **two bytes** , a **count** and a **symbol**. RLE can compress any type of data regardless of its information content, but the **content of data** to be compressed **affects the compression ratio**. RLE cannot achieve high compression ratios compared to other compression methods, but it is easy to implement and is quick to execute. Run-length encoding is supported by most bitmap file formats such as TIFF, BMP and PCX.

Compression is normally **measured** with the compression ratio :

Compression Ratio = original size / compressed size : 1

Consider a **character run of 15 'A' characters** which normally would require 15 bytes to store:

AAAAAAAAAAAAAAA

AAAAA becomes 15A

15A

With RLE, this would **only** require **two bytes to store**; the count (15) is stored as the first byte and the symbol (A) as the second byte.

Consider another example with **16 characters string** of:

000ppppppXXXXaaa

This string of characters **can be compressed** to form

3(0),6(p),4(X),3(a)

Hence, the **16-byte string** would **only require 8 bytes** of data **to represent** the string. In this case, RLE yields a **compression ratio of 2:1**.

In run-length encoding, **repetitive source** such as **a string of numbers** can be represented in a compressed form, for example,

1,4,5,1,4,5,1,4,5

can be **compressed** to form

3(1,4,5)

Thus, giving a compression ratio of = 9/4: 1 which is almost 2 : 1.

Another simple example is when we have a **source** of **incremental patterns** that can be compressed by **differencing**. This is done as shown, **given a set** of values

1,2,3,5,6,7,9

taking the **difference between two adjacent values** (eg. 2-1=1, 3-2=1, 5-3=2... etc) we will obtain

1,1,2,1,1,2

This result **could be further compressed** by representing it as repeated strings, i.e;

2(1,1,2)

Long runs are rare in certain types of data. For instance, in ASCII text files, long runs seldom occur. To **encode a run** in RLE, it is **required** that there is **a minimum of two characters** worth of information, **otherwise** a run of **single character** takes **more space**. In other words, when there is no **run**, there is **no compression**.

In most of the examples above, **compression is achieved** because there were long character runs. However if we observe the data example below,

XttmprsQssqznO

We obtain....

1(X),2(t),1(m),1(p),1(r),1(s),1(Q),2(s),1(q), 1(z),1(n),1(O)

which is being expanded from the original data.

2.2 Huffman Compression

Huffman compression **reduces** the **average code length** used **to represent** the **symbols of an alphabet**. Symbols of the source alphabet that occur frequently are assigned with **short length codes**. The general strategy is to allow the code length to vary from character to character and to ensure that the frequently occurring character has shorter codes.

Huffman compression is performed by constructing a **binary tree** using a simple example set.

1. This is done by **arranging the symbols** of the alphabets in **descending** order of probability.
2. Then repeatedly adding **two lowest probabilities** and resorting. This process goes on until the sum of probabilities of the **last two symbols is 1**.
3. Once this process is complete, a Huffman binary tree can be generated.
4. If we do not obtain a probability of 1 in the last two symbols, most likely there is a mistake in the process.
5. This probability of 1 that forms the last symbol is the **root** of the binary tree.

The **resultant codewords** are then formed by tracing the tree path **from the root node** to the endnodes codewords after assigning **0**s and **1**s to the branches.

A step-by-step **worked example** in constructing a Huffman binary tree is shown below:

Given a set of **symbols** with a list of relative **probabilities** of occurrence within a message.

m0	m1	m2	m3	m4
0.10	0.36	0.15	0.2	0.19

(1) List symbols in the order of **decreasing probability**.

m1	m3	m4	m2	m0
0.36	0.20	0.19	0.15	0.10

(2) Get **two** symbols with **lowest probability**. Give the combined symbol a **new name**.

m2	m0
0.15	0.10

Combines to form

A
0.25

(3) The **new list** obtained is shown below. Repeating the previous step will give us a new symbol for the **next two lowest probabilities**.

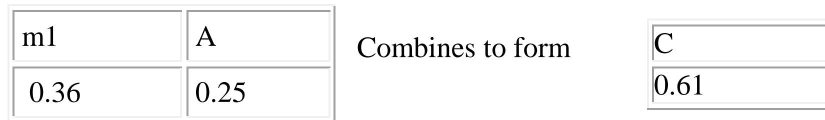
m1	A	m3	m4
0.36	0.25	0.20	0.19

m3	m4
0.20	0.19

Combines to form

B
0.39

(4) A **new list** is obtained. Repeating the previous step will give us a **new symbol** for the following **two lowest probabilities**.



B	m1	A
0.39	0.36	0.25

m1	A
0.36	0.25

Combines to form

C
0.61

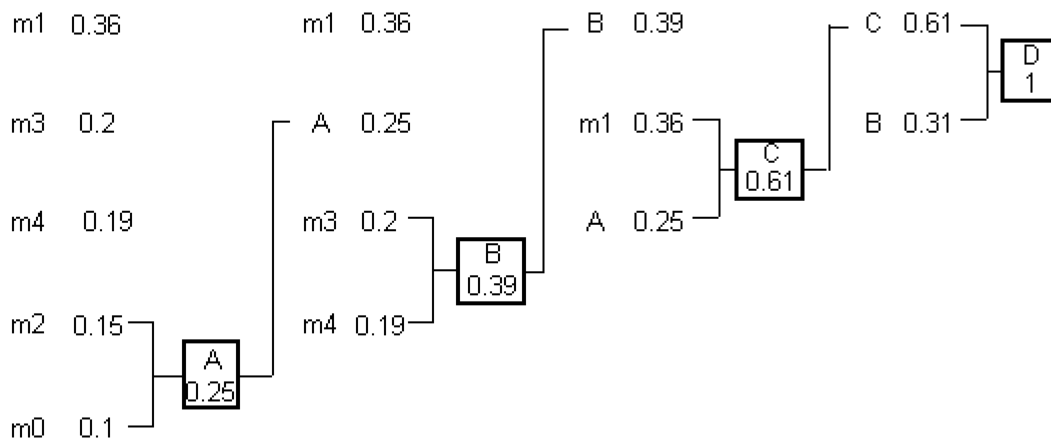
(5) Finally there is only one pair left and we simply combine them and name them as a **new symbol.**

B	C
0.39	0.61

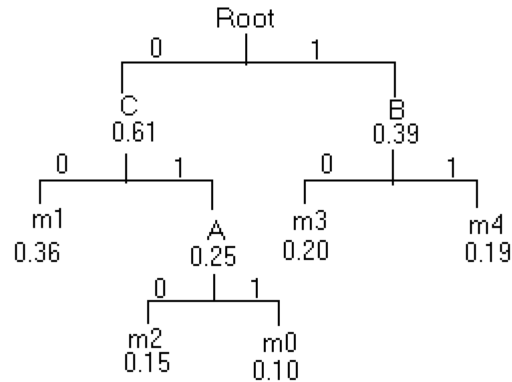
Combines to form

D
1.0

(6) Having finished these steps we have :



(7) Now, a **Huffman tree** can be constructed, **0**'s and **1**'s are assigned to the branches.



(8)The **resultant codewords** are formed by tracing the tree path from the root node to the codeword leaf.

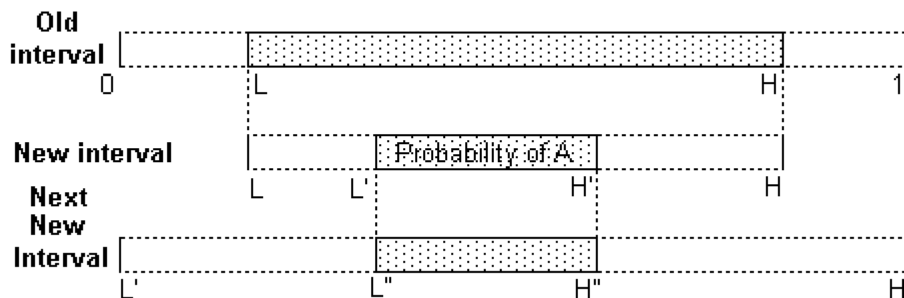
Symbols	**Probabilities**	**Codewords**
m0	0.10	011
m1	0.36	00
m2	0.15	010
m3	0.20	10
m4	0.19	11

Notice that **compression** is achieved by allocating **frequently occurring symbols** with **shorter codeword**.

2.3. Arithmetic Compression

Arithmetic compression is an alternative to Huffman compression; it enables characters to be represented as **fractional bit lengths**.
Unlike for Huffman compression, where fractional code lengths are not possible and the allocation of shorter codeword for more frequently occurring characters needs at least one-bit codeword no matter how high its frequency is.
Arithmetic coding works by **representing** a number by an **interval** of **real numbers greater** or **equal** to **zero**, but **less than one**. As a **message** becomes **longer**, the **interval** needed to represent it becomes **smaller and smaller**, and the **number of bits** needed to specify it **increases**.

The figure below shows the subdivision of the current interval based on the probability of the input symbol (A) that occurs next.



The **basic algorithm** for encoding a file using arithmetic coding works conceptually as follows:
(1) Begin with current **range [L,H)** initialised to **[0,1)**.
Note: We denote brackets [0,1) in such a way to show that it is equal to or greater than 0 but less than 1.

(2) For **each symbol** of the file, we perform **two steps**:

a) Subdivide the current interval into subintervals, one for each alphabet symbol.

b) Select the subinterval corresponding to the symbol that actually occurs next in the file and make it the new current interval.

(3) Output enough bits to **distinguish** the **current interval** from all **other possible interval**.
This table shows the source character and its current intervals according to its probability.

Source Characters	**Probabilities**	**Current Interval [Pi, P*j*)**
A	0.4	[0 , 0.4)
B	0.5	[0.4 , 0.9)
C	0.1	[0.9 , 1.00)

The **new intervals** can be obtained by **[L + P*i* (H-L) , L + P*j* (H-L)),** where **P*i*** and **P*j*** are the **cumulative probabilities.**

For example**,** the cumulative probabilities for character B and C are, **P*i* = 0.4 P*j* = 0.9** and **P*i* = 0.9 P*j* = 1.00** respectively. If we need to **encode** a file containing **symbols BBBC**, we obtain the following **new interval** shown in the table below.

Input	**New Interval [L, H)**
--	[0 , 1)
B	[0.4 , 0.9)
B	[0.6 , 0.85)
B	[0.7 , 0.825)
C	[0.8125 , 0.825)

In **summary**, the encoding process of arithmetic compression is simply one of **narrowing the range** of possible **number** with every **new symbol**. The **new range** is **proportional** to the **predefined probability** attached to that symbol.