# Data Compression

The purpose of data compression is to make the files smaller which means that:

- Less time / less bandwidth to transfer data
- Take up less space on the disk

Data compression can be applied to all sorts of files including images, sound and text.

**Lossy versus Lossless Compression**

**Lossless compression** - The original uncompressed representation can be recreated exactly from the compressed image. That is, we can reverse the compression and the uncompressed data file will be exactly (down to the last bit) the same as the original file.

**Lossy compression** reduces the size of the file but the original uncompressed data will not be able to be fully recreated. Lossy compression only approximates the uncompressed data and the original data will be irrecoverable. Some of the quality of the original file will be lost. The benefit is that files using lossy compression will be smaller than those using lossless compression.

**Run Length Encoding (RLE)**

Run Length Encoding is a compression method where sequences of the same values are stored in pairs of the value and the number of those values.

RLE is a lossless compression method.

For instance, the sequence:

```
0 0 0 1 1 0 1 1 1 1 0 1 1 1 1
```

would be represented as:

```
3 0 2 1 1 0 4 1 1 0 4 1
```

Given that there are 7 bits per ASCII character, the uncompressed size of an ASCII phrase is:

*size = number of characters (including spaces) x 7*

Suppose we have a sequence of the following ASCII characters:

```
Goooooooaaaaallll
```

Uncompressed size: 17 x 7 = 119 bits

With RLE compression this would be: `1G, 7o, 5a, 4l`

Suppose we store the number of each character in 3 bits and we the ASCII characters are 7 bits then we have:

```
001 1000111 111 1101111 101 1100001 100 1101100
```

RLE compressed: `7x4 + 3x4 = 40 bits`

**Dictionary based methods**

With dictionary-based compression methods we use a dictionary to encode and decode the message.

We are going to compress the following chorus from a song by the Beatles:
"We all live in a yellow submarine yellow submarine yellow submarine"

Each ASCII character is 7 bits and there is a total of 67 characters including spaces so the total message is 67 x 7 bits which is 469 bits.

Using the dictionary method there are 4 by 11 = 44 bits. Ignoring the dictionary, this is more than ten times compression on the original message. Of course using a repetitive message helps compress the message.

For a short message we are not reducing the volume of data because the dictionary itself will have volume and in fact we may even increase the volume of data compared with the original uncompressed message however for long messages the dictionary method of compression will save us space.

*Dictionary*

| word | Encoding |
|---|---|
| We | 0000 |
| all | 0001 |
| live | 0010 |
| in | 0011 |
| a | 0100 |
| yellow | 0101 |
| submarine | 0110 |

*Complete encoding*

| word | Encoding |
|---|---|
| We | 0000 |
| all | 0001 |
| live | 0010 |
| in | 0011 |
| a | 0100 |
| yellow | 0101 |
| submarine | 0110 |
| yellow | 0101 |
| submarine | 0110 |
| yellow | 0101 |
| submarine | 0110 |

WARNING: Sometimes compression will increase the size of the file. For instance, perform RLE compression on the following sequence:

```
1 0 1 0 1 0 1 0 1 0 1 0 1 (13 bits)
```

This would be:

```
11 01 11 01 11 01 11 01 11 01 11 01 11 (26 bits)
```

The size of the file has been doubled by applying a compression algorithm!

# Encryption

When transferring sensitive information from one place to another it is necessary to **encrypt** the message. **Encryption** means that the message is scrambled from its original **plaintext** into **ciphertext.** A **key** is required in order to understand the original message.

**Plaintext** is the original message

**Ciphertext** is the encrypted message

Given enough time, ciphertext and computational power nearly all methods of encryption can be cracked using **cryptoanalysis**.

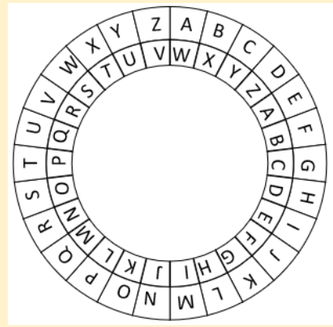
**Cryptoanalysis** is the breaking of encrypted codes without being given the key. Only the **Vernam** cipher has proven to be unbreakable.

**Caesar Cipher**
The Caesar cipher is a shift substitution cipher. It works by replacing a letter with another letter that is shifted along in the alphabet by a set number of places. The key is a letter and lets us know the number of letters we shift the alphabet by. So for instance a key of *E* means that we would shift by 4 places.

In the example below we are using a key W so we shift by 22 places to the left.

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |

This can be represented as a wheel. The letters on the outer wheel represent the original letters, the letters on the inside are the corresponding encrypted letter.




- The word MARCH would be encrypted as IWNYD
- To decrypt the message we use reverse the process. So XASWNA would be BEWARE

*Cryptoanalysis of Caesar Cipher*

- The Caesar cypher is not a secure method of encryption and is easy to crack using two methods.
- Try all 25 shifts and apply it to the cipher text. When you have a sensible plaintext message then you have cracked the encryption and found the key.
- Alternatively, if you had a long enough piece of cyphertext you could use frequency analysis of letters. E is the most commonly used letter, so the most commonly letter in the cyphertext will likely be a E and from there is it a trivial step to calculate the key

**Vernam (one-time pad) cipher**
With a Vernam cipher we have perfect security. It is the only cipher that has proven impossible to break using cryptoanalysis. To help achieve this the Vernam cipher has three features:

- The key is only used once
- The length of the key is at least long as the message that we want to send.
- It is truly random

*Applying the Vernam Cipher*

| Original Message | Hello |
|---|---|
| Convert to ASCII plain text | 1001000 1100101 1101100 1101100 1101111 |
| Generate a random key | !d7sY |
| Key in ASCII | 0100001 1100100 0110111 1110011 1011001 |
| Apply bitwise XOR to key and plain text | 1001000 1100101 1101100 1101100 1101111<br>0100001 1100100 0110111 1110011 1011001<br>1101001 0000001 1011011 0011111 0110110 |
| Encrypted message (cypher text) | 1101001 0000001 1011011 0011111 0110110 |
| To decrypt apply the XOR operator to the cypher text and the key | 1101001 0000001 1011011 0011111 0110110<br>0100001 1100100 0110111 1110011 1011001<br>1001000 1100101 1101100 1101100 1101111 |
| Plain text | 1001000 1100101 1101100 1101100 1101111 |