CAIE Computer Science IGCSE

1 - Data Representation

Advanced Notes

1.1 Number systems

What are number systems?

A number system determines how values are represented using digits; a number base defines the number of unique digits used in that system.
The most common number systems are:

- Denary (Base 10) - used by humans for counting
- Binary (Base 2) - used by computers to represent all data and instructions
- Hexadecimal (Base 16) - used by programmers for compact representation

Denary (base 10)

Denary is the number system that humans use to count, perhaps because we have ten fingers. Denary uses the ten digits 0 through to 9 to represent numbers.

Each digit in a decimal number has a place value based on powers of 10. The value of a digit depends on its position within the number. This is illustrated by the table below, which shows how the decimal number 237 is constructed using place values.

10^2	10^1	10^0
100	10	1
2	3	7

```
237 = (2×100) + (3×10) + (7×1)
```

Binary (base 2)

Binary is used by computer systems to store all data and instructions. This is because it has only two states, 0 or 1, which map directly to the two states of electronic components like transistors: on (1) or off (0) and other logic gates used to process and store data. This simplicity makes it easier to design, build, and maintain computer hardware. Therefore, data needs to be converted into a binary format to be processed by a computer.

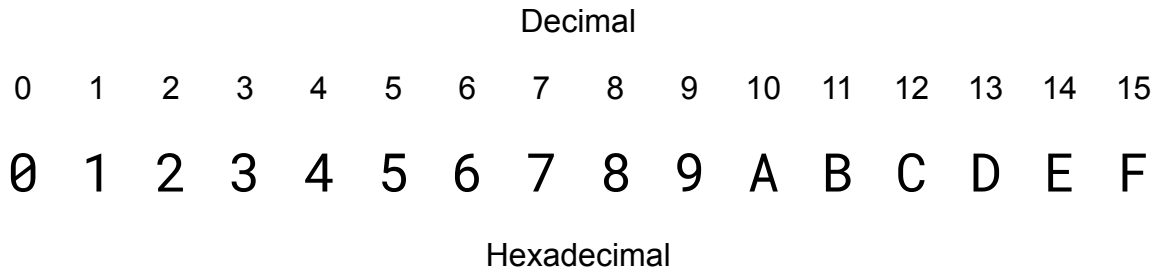
Each digit in a binary number has a place value based on powers of 2. This is illustrated by the table below, which shows how the binary number 1011 is constructed using place values - making it equal to 11 in denary.

2^3	2^2	2^1	2^0
8	4	2	1
1	0	1	1

```
1011 = (1×8) + (0×4) + (1×2) + (1×1) = 11 (decimal)
```

Hexadecimal (base 16)

In contrast to decimal, hexadecimal uses the digits 0 through to 9 followed by the uppercase characters A to F to represent the denary numbers 0 to 15.



Of all the number bases covered by this course, hexadecimal is the most compact. This means that it can represent the same number as binary or decimal while using far fewer digits. Each character in hexadecimal represents four bits in binary.

Each digit in a decimal number has a place value based on powers of 16. This is illustrated by the table below, which shows how the hexadecimal value 2F is constructed using place values - making it equal to 47 in decimal.

16^1	16^0
16	1
2	15 (because F represents 15)

```
2F = (2×16) + (15×1)  = 47 (decimal)
```

Converting Denary ↔ Binary

To convert binary → denary:

You can convert from binary to decimal by using place value headers. Starting with one and increasing in powers of two, placing larger values to the left of smaller values. For example, the binary number `10110010` could have place value headers added as follows:

128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
1	0	1	1	0	0	1	0

The binary number could then be converted to denary by adding together all of the place values with a binary one below them.

$$128 + 32 + 16 + 2 = 178$$

So the binary number `10110010` is equivalent to the decimal number 178.

To convert decimal → binary:

When converting from denary to binary, you use the same place value headers. Starting from the left hand side, you place a one if the value is less than or equal to your number, and a zero otherwise.

Once you've placed a one, you must subtract the value of that position from your number and continue as before, until your decimal number becomes 0.

Let's say we're converting the number 53 to binary. First, write out your place value headers in powers of two. Keep going until you've written a value that is larger than your number. For 53, we're going to go up to 64.

64	32	16	8	4	2	1

Now, starting from the left, compare the place value to your number. 64 is greater than 53 so we place a 0 under 64.

64	32	16	8	4	2	1
0						

Moving to the right, we see that 32 is lower than 53, so we place a 1 under 32.

64	32	16	8	4	2	1
0	1					

Because we've placed a 1, we have to subtract 32 from 53 to find what's left to be represented. In this case, 53 - 32 = 21.

We move to the right again and find 16, which is lower than 21, so we place a 1 under 16.

64	32	16	8	4	2	1
0	1	1				

Again, because we've placed a 1, we have to calculate a new value. 21 - 16 = 5.
Moving right, we find 8. This is larger than 5 so we place a 0.

64	32	16	8	4	2	1
0	1	1	0			

After moving right again, we find 4. As 4 is lower than 5, we place a 1.

64	32	16	8	4	2	1
0	1	1	0	1		

Having placed a 1, we must again calculate a new value. 5 - 4 = 1.

Moving right to find 2, we place a 0 as 2 is greater than 1.

64	32	16	8	4	2	1
0	1	1	0	1	0	

Moving right for the last time, we have 1. 1 = 1 so we place a 1.

64	32	16	8	4	2	1
0	1	1	0	1	0	1

Now that we've placed a 0 or a 1 under each place value, we have our answer. Although it's acceptable to remove any leading 0s, it may be preferable to add 0s to the start of your answer to make it a whole number of bytes (a multiple of 8 bits).

$$53 = 0110101 = 110101 = 00110101$$

Most Significant and Least Significant Bit

The most significant bit is the bit with the highest value, which is the leftmost 1 in a binary number. The least significant bit is the bit with the lowest value, which is the rightmost bit, whether it is a 0 or 1, in a binary number.

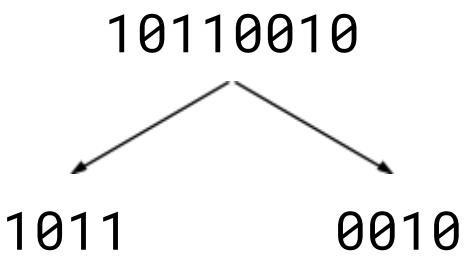
Adding additional 0s to the left of a binary number does not change its value, e.g. 11010 is the same as 00011010.

Converting Binary ↔ Hexadecimal

To convert binary → hex:

In order to convert from binary to hexadecimal, the binary number must first be split into nibbles. A nibble is four binary bits, or half a byte.

For example, the binary number 10110010 would be split into two nibbles:



Each binary nibble is then converted to decimal as in the previous example:

8	4	2	1
1	0	1	1

8 + 2 + 1 = 11

8	4	2	1
0	0	1	0

2 = 2

Once each nibble has been converted to decimal, the decimal value can be converted to its hexadecimal equivalent like so:

11 = B

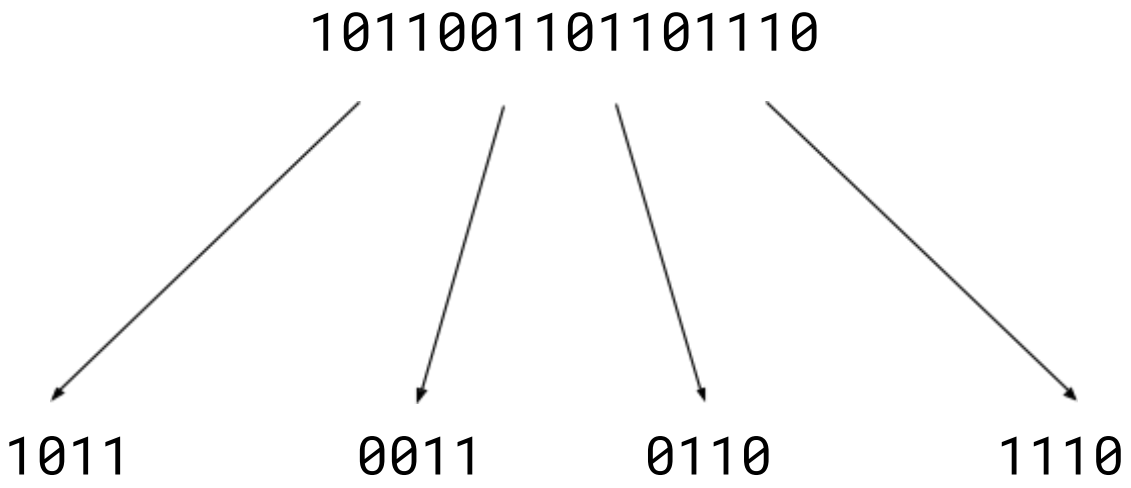
2 = 2

Finally, the hexadecimal digits are concatenated to form a hexadecimal representation:

10110010 = B2

For this specification, the maximum length of binary number that you could be asked to convert is 16-bit. The following example will show how you would convert a 16-bit binary number into hexadecimal.

For example, the binary number 1011001101101110 would be split into 4 nibbles.



Each binary nibble is then converted to decimal as in the previous example:

8 4 2 1	8 4 2 1	8 4 2 1	8 4 2 1
1 0 1 1	0 0 1 1	0 1 1 0	1 1 1 0
8 + 2 + 1 = 11	2 + 1 = 3	4 + 2 = 6	8 + 4 + 2 = 14

Once each nibble has been converted to decimal, the decimal value can be converted to its hexadecimal equivalent like so:

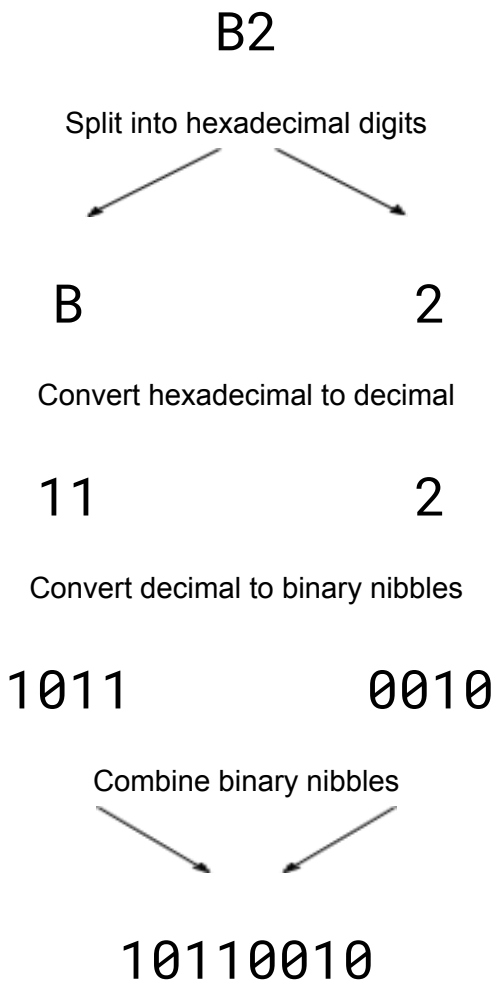
11 = B 2 = 2 6 = 6 14 = E

Finally, the hexadecimal digits are concatenated to form a hexadecimal representation. This representation has significantly fewer characters so is far easier for a human to read.

1011001101101110 = B26E

To convert hex → binary:

First, convert each hexadecimal digit to a decimal digit and then to a binary nibble before combining the nibbles to form a single binary number.



Converting Denary ↔ Hexadecimal

To convert denary → hex:

Combining the steps above:

1. Begin by converting the denary number into binary
2. Convert this binary number to hexadecimal

To convert hex → denary:

Combining the steps above:

1. Begin by converting the hexadecimal number into binary
2. Convert this binary number to denary.

How and why hexadecimal is used

Hexadecimal is a shorthand representation of binary: it is easier for people to read than binary, and it takes less time to type than binary. Therefore, hexadecimal representation is used because it is easier for humans to read and work with. However, hexadecimal does not offer any advantage to computers; computers always represent numbers using binary.

There are several areas within computer science where hexadecimal is used, for example:

- **Colour codes:** Hex is used to represent RGB colour values in HTML/CSS (e.g., #FE7C1B for a specific shade of orange).
- **Media Access Control (MAC) addresses:** Devices on a network have unique MAC addresses written in hex (e.g., 00:1A:2B:3C:4D:5E), representing 48-bit identifiers.

Binary addition

When adding binary numbers, there are three important rules to remember:

Binary add	Result	Carry
0 + 0	0	0
1 + 0	1	0
1 + 1	0	1 (carry)
1 + 1 + 1	1	1 (carry)

You'll only be expected to add two positive binary numbers of 8 bits.

A computer or a device has a predefined limit that it can represent or store, depending on the number of bits allocated. An overflow error can occur when the result of a binary addition is too large to be represented by the number of bits available. For example, if the result is greater than 255 in denary, requiring more than 8 bits, and you only have 8 bits available, then an overflow error will occur.

Example

Add binary integers 1011 and 1110.

	1	0	1	1
+	1	1	1	0

Place the two binary numbers above each other so that the digits line up.

	1	0	1	1
+	1	1	1	0
				1

Starting from the least significant bits (the right hand side), add the values in each column and place the total below. For the first column (highlighted), rule 2 from above applies.

	1	0	1	1
+	1	1	1	0
		^{1}	0	1

Move on to the next column. This time rule 3 applies. In this case there is a carry digit. Place a 1 in small writing under the next most significant bit's column.

	1	0	1	1
+	1	1	1	0
	^{1}	0^{1}	0	1

On to the next column, where there is a 0, a 1 and a small 1. In this case, rule 3 applies again. Therefore the result is 10. Because 10 is two digits long, the 1 is written in small writing under the next most significant bit's column.

	1	0	1	1
+	1	1	1	0
1	1^{1}	0^{1}	0	1

Moving on to the most significant column where there are three 1s. Rule 4 applies, so the result for this column is 11. The first digit of the result is written under the next most significant bit's column, but it can be written full size as there are no more columns to add.

1	1	0	0	1

Finally, the result is read off from the full size numbers at the bottom of each column. In this case, 1011 + 1110 = 11001.

Note: the overflow 1 (most significant bit) should be removed in questions unless stated otherwise. You should state that you have removed the overflow bit.

Binary shifts

A logical binary shift involves moving the bits of a binary number left or right. Bits shifted from the end of the register are lost and zeros are shifted in at the opposite end of the register. This means that the most significant bit(s) or least significant bit(s) are lost.

There are two types of binary shift:

- Left shift → moves all bits to the left (adds 0s on the right)
    - Same as multiplying by 2 for each place shifted
- Right shift → moves all bits to the right (adds 0s on the left)
    - Same as dividing by 2 for each place shifted

Example

In this example, we'll apply a binary left shift of 1 to the original binary number `00101100`. The effect of this is to multiply 44 by 2, making 88.

Original:	`00101100 (44)`
Shifted:	`01011000 (88)`

Two's complement

When using two's complement, the most significant bit of a binary number is given a negative place value. For an 8-bit number, the place values are:

-128	64	32	16	8	4	2	1

This allows negative numbers to be represented as low as -128. If the first bit is 1, the number is negative. If it's 0, the number is positive.

For example, 1011 represented using two's complement binary is equivalent to -5 in denary. The place value headers for this example are shown below:

-8	4	2	1
1	0	1	1

-8 + 2 + 1 = -5

To convert positive denary integer → 8-bit two's complement binary:

This procedure is almost identical to the standard denary to binary conversion, as covered in the notes above.

If you need the result to be 8 bits, pad it with leading 0s to the left of the number. You must make sure that the most significant bit isn't a 1, otherwise the result will become an incorrect negative number.

For example, converting the denary number 25 to binary will give the result 11001. If we need to make this 8-bits, then we can simply add leading zeros to the left of this result, giving the result 00011001.

To convert positive 8-bit two's complement binary → denary integer :

This procedure is identical to the standard binary integer to denary conversion, as covered in the notes above. When you write out the place values above the numbers, they should be as follows, where the most significant bit has a negative sign:

−128	64	32	16	8	4	2	1

To convert negative 8-bit two's complement binary → denary integer:

Carry out the following procedure:

1. Confirm that the most significant bit is 1. This indicates the number is negative.
2. Invert all the bits (change 0s to 1s and 1s to 0s).
3. Add 1 to the inverted binary number.
4. Convert the result to denary as a positive binary number.
5. Add a minus sign to make the answer negative.

For example, to convert 11101010 to denary:
Step 1: The first bit is 1, so the number is negative.
Step 2: Invert the bits → 00010101
Step 3: Add 1 → 00010101 + 1 = 00010110
Step 4: 00010110 = 22 in denary
Step 5: Make it negative → -22

The result is -22.

To convert a negative denary integer → 8-bit two's complement binary:

Carry out the following procedure:

1. Write the positive version of the number in binary.
2. Pad it with leading 0s to make it 8 bits long.
3. Invert all bits (change 0s to 1s and 1s to 0s).
4. Add 1 to the inverted binary number.

The final result should be 8 bits long and start with a 1.

For example, to convert -37 to 8-bit two's complement binary:
Step 1: +37 in binary is 100101
Step 2: Pad with leading 0s → 00100101
Step 3: Invert the bits → 11011010
Step 4: Add 1 → 11011010 + 1 = 11011011

The result is 11011011 in 8-bit two's complement binary.

To convert negative 8-bit two's complement → positive 8-bit two's complement:

Carry out steps 2, 3, and 4 of the previous algorithm:

1. Pad with leading 1s (as it is negative)
2. Invert all bits (change 0s to 1s and 1s to 0s).
3. Add 1 to the inverted number.

For example, to convert -61 to +61 in two's complement:
Step 1: -61 is 1000011 (7 bits) therefore pad with 1s → 11000011
Step 2: Invert the bits → 00111100
Step 3: Add 1 → 00111101 = 61

Note: the exact same process can be used to convert from positive to negative in two's complement

1.2 Text, sound and images

Representing text

Text must be converted into binary to be processed by a computer.

A character set, such as ASCII or Unicode, is a collection of characters and their corresponding binary values. Every character is assigned a unique binary code, known as a character code, using a standard such as ASCII or Unicode.

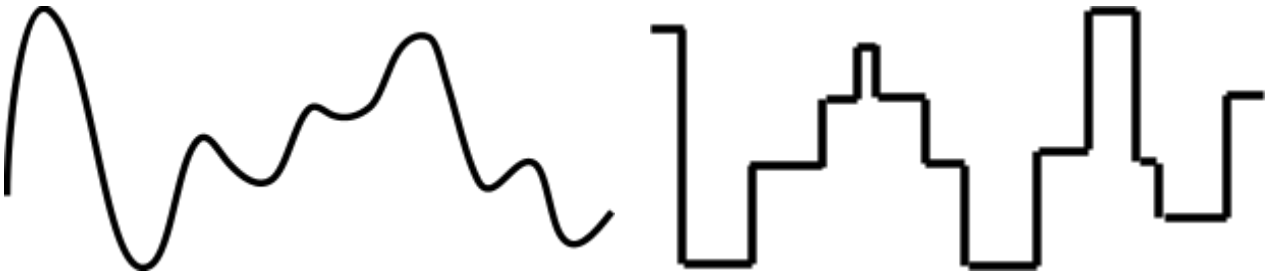
ASCII (American Standard Code for Information Interchange)

- Uses 7 bits to represent each character
- Can store 128 (2^7) characters
- Includes:
    - English letters (uppercase & lowercase)
    - Digits 0–9
    - Common symbols (@, #, etc.)
    - Control codes (like newline)

Unicode

- Uses 8-48 bits to represent each character, allowing it to represent a much wider range of different characters than ASCII, but requiring much more space.
- Supports many different languages (not just the Latin alphabet but also alphabets like Arabic, Cyrillic, Greek and Hebrew), and more symbols (such as emojis). This means that data such as text can be represented in a wider range of languages, making computers more accessible worldwide.

Representing sound



Analogue signal

Digital signal

Sound is analogue, meaning that its signal is a continuous wave that can take any value, not having a singular value. Computers cannot store continuous sound waves, so they take regular snapshots (samples) of the sound wave's amplitude. A sample is a measure of amplitude at a point in time - each sample is stored as a binary number.

The sampling rate is the number of samples taken in a second.

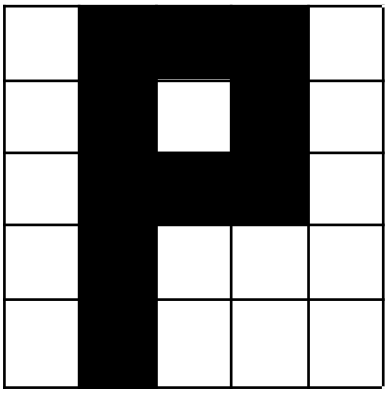
The sample resolution is the number of bits per sample (gives a more precise and accurate measure of the sound's amplitude at any one point).

The accuracy of the recording and the file size increases as the sample rate and resolution increase.

Representing images

Digital images are made up of a series of tiny squares called pixels (short for “picture elements”). A pixel is a single point in an image. Each pixel has a colour value, and this is stored in binary.

The value assigned to a pixel determines the colour of the pixel. The example below shows the binary representation of a simple image in which a 1 represents a black pixel and a 0 represents a white pixel.



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

The number of bits assigned to a pixel in an image is called its colour depth. In the example above, each pixel has been assigned one bit, allowing for 2 (2^1) different colours to be represented. If a colour depth of two bits were used, there would be four (2^2) different colours that each pixel could take, represented by the bit patterns 00, 01, 10 and 11.

The resolution refers to the number of pixels within an image. Resolution can be found by multiplying the image width in pixels by the image height in pixels.

The file size and quality of the image increase as the resolution and colour depth increase

1.3 Data storage and compression

Unit prefixes

Each binary digit is a bit of data. You'll often come across the following prefixes used for decimal numbers, and you need to be able to convert between them.

Unit	Symbol	Relative size
Bit	b	1 bit
Nibble		4 bits
Byte	B	8 bits
Kibibyte	KiB	1024 bytes
Mebibyte	MiB	1024 kibibytes
Gibibyte	GiB	1024 mebibytes
Tebibyte	TiB	1024 gibibytes
Pebibyte	PiB	1024 tebibytes
Exbibyte	EiB	1024 pebibytes

File size calculations

To calculate the file sizes of sound and image files, you can use the following formulas:

$$Sound\ file\ size\ =\ sample\ rate \times duration\ (s) \times bit\ depth$$

$$Image\ resolution\ =\ height\ (px) \times image\ width\ (px)$$

$$Image\ file\ size\ =\ colour\ depth \times image\ resolution$$

Note: file size calculations *must* use the measurement of 1024 and *not* 1000.

Data compression

Data compression is the process of reducing the file size of digital data without losing the original information (or with minimal acceptable loss). It is used to save storage space and speed up transmission, as well as reducing the bandwidth required. There are two types of image compression: lossy and lossless.

Lossy compression

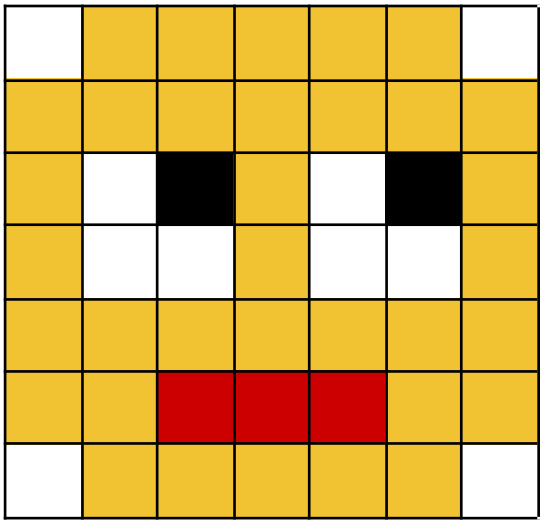
When using lossy compression, some information is permanently lost in the process of reducing the file's size. This could cause the quality of the file to be slightly reduced; the compressed file can never be fully restored to the original. This could be done by reducing the resolution of audio or reducing the colour depth of an image.

Lossless compression

In contrast to lossy compression, there is no permanent loss of information when using lossless compression. The size of a file can be reduced without decreasing its quality. Lossless compression methods use algorithms to find and compress patterns (e.g. repeated data).

Run length encoding (RLE)

Run length encoding (RLE) is a type of lossless compression, which reduces the size of a file by removing repeated information and replacing it with one occurrence of the repeated information followed by the number of times it is to be repeated.



$$00\ 11_5\ 00$$
$$11_7$$
$$11\ 00\ 01\ 11\ 00\ 01\ 11$$
$$11\ 00_2\ 11\ 00_2\ 11$$
$$11_7$$
$$11_2\ 10_3\ 11_2$$
$$00\ 11_5\ 00$$

The example uses the image of a face that was represented as a bitmap image earlier in these notes. Using RLE to replace repeated pixels with one pixel value and a number of repetitions has reduced the storage space required to represent the image.

The third row of pixels in the image has no repeated values and as such, couldn't be compressed by RLE. This highlights the fact that not all data is suitable for compression by run length encoding. For example, text is not suited to RLE at all, as it is unlikely to have many 'runs' of repeated letters.