1 Information representation and multimedia

In this chapter, you will learn about

- binary magnitudes, binary prefixes and decimal prefixes
- binary, denary and hexadecimal number systems
- how to carry out binary addition and subtraction
- the use of hexadecimal and binary coded decimal (BCD) number systems
- the representation of character sets (such as ASCII and Unicode)
- how data for a bit-mapped image is encoded
- how to estimate the file size for a bit-map image
- image resolution and colour depth
- encoding of vector graphics
- the representation of sound in a computer
- the effects of changing sampling rate and resolution on sound quality
- the need for file compression methods (such as lossy and lossless formats)
- how to compress common file formats (such as text files, bit-map images, vector graphics, sound files and video files).

WHAT YOU SHOULD ALREADY KNOW

Try these four questions before you read this chapter.

1 What are the column weightings for the binary number system?

2 Carry out these binary additions. Convert your answers to denary.

a) 0 0 1 1 0 1 0 1 + 0 1 0 0 1 0 0 0

b) 0 1 0 0 1 1 0 1 + 0 1 1 0 1 1 1 0

c) 0 1 0 1 1 1 1 1 + 0 0 0 1 1 1 1 0

d) 0 1 0 0 0 1 1 1 + 0 1 1 0 1 1 1 1

e) 1 0 0 0 0 0 0 1 + 0 1 1 1 0 1 1 1

f) 1 0 1 0 1 0 1 0 + 1 0 1 0 1 0 1 0

3 What are the column weightings for the hexadecimal (base 16) number system?

4 Carry out these hexadecimal additions. Convert your answers to denary.

a) 1 0 7 + 2 5 7

b) 2 0 8 + A 1 7

c) A A A + 7 7 7

d) 1 F F + 7 F 7

e) 1 4 9 + F 0 F

f) 1 2 5 1 + 2 5 6 7

g) 34AB + C00A

h) A001 + D77F

i) 1009 + 9FF1

j) 2777 + ACF1

1.1 Data representation

Key terms

Binary – base two number system based on the values 0 and 1 only.

Bit – abbreviation for binary digit.

One’s complement – each binary digit in a number is reversed to allow both negative and positive numbers to be represented.

Two’s complement – each binary digit is reversed and 1 is added in right-most position to produce another method of representing positive and negative numbers.

Sign and magnitude – binary number system where left-most bit is used to represent the sign (0 = + and 1 = –); the remaining bits represent the binary value.

Hexadecimal – a number system based on the value 16 (uses the denary digits 0 to 9 and the letters A to F).

Memory dump – contents of a computer memory output to screen or printer.

Binary-coded decimal (BCD) – number system that uses 4 bits to represent each denary digit.

ASCII code – coding system for all the characters on a keyboard and control codes.

Character set – a list of characters that have been defined by computer hardware and software. It is necessary to have a method of coding, so that the computer can understand human characters.

Unicode – coding system which represents all the languages of the world (first 128 characters are the same as ASCII code).

1.1.1 Number systems

Every one of us is used to the decimal or denary (base 10) number system. This uses the digits 0 to 9 which are placed in ‘weighted’ columns.

10000	1000	100	10	units
3	1	4	2	1

The denary number represented above is thirty-one thousand, four hundred and twenty-one.

(Note that dealing with decimal fractions is covered in Chapter 13 since this is slightly more complex.)

Designers of computer systems adopted the **binary** (base 2) number system since this allows only two values, 0 and 1. No matter how complex the system, the basic building block in all computers is the binary number system. Since computers contain millions and millions of tiny ‘switches’, which must be in the ON or OFF position, this lends itself logically to the binary system. A switch in the ON position can be represented by 1; a switch in the OFF position can be represented by 0. Each of the **bi**nary dig**its** are known as **bits**.

1.1.2 Binary number system

The binary system uses 1s and 0s only which gives these corresponding weightings:

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

A typical binary number would be:

1	1	1	0	1	1	1	0

Converting from binary to denary and from denary to binary

It is fairly straightforward to change a binary number into a denary number. Each time a 1 appears in a column, the column value is added to the total. For example, the binary number above is:

128 + 64 + 32 + 8 + 4 + 2 = 238 (denary)

The 0 values are simply ignored when calculating the total.

The reverse operation – converting from denary to binary – is slightly more complex. There are two basic ways of doing this.

Consider the conversion of the denary number, 107, into binary …

Method 1

This method involves placing 1s in the appropriate position so that the total equates to 107.

128	64	32	16	8	4	2	1
0	1	1	0	1	0	1	1

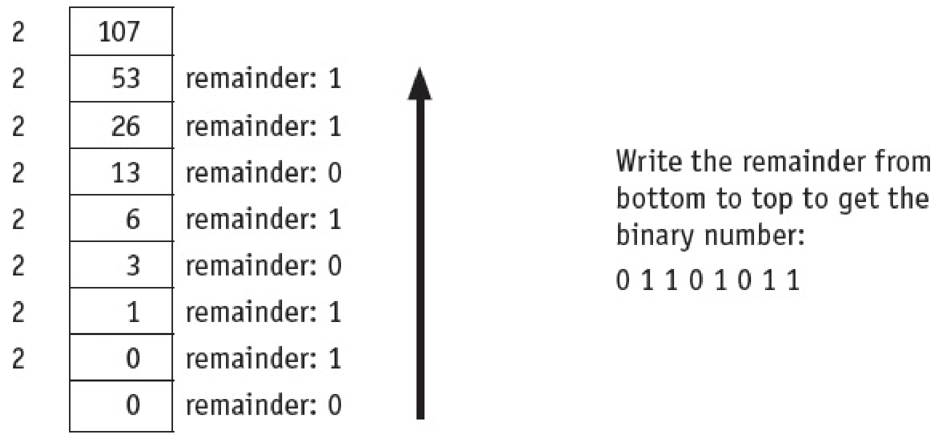
ACTIVITY 1A

Convert these binary numbers into denary.

a) 0 0 1 1 0 0 1 1
b) 0 1 1 1 1 1 1 1
c) 1 0 0 1 1 0 0 1
d) 0 1 1 1 0 1 0 0
e) 1 1 1 1 1 1 1 1
f) 0 0 0 0 1 1 1 1
g) 1 0 0 0 1 1 1 1
h) 0 0 1 1 0 0 1 1
i) 0 1 1 1 0 0 0 0
j) 1 1 1 0 1 1 1 0

Method 2

This method involves successive division by 2; the remainders are then written from bottom to top to give the binary value.



2	107	
2	53	remainder: 1
2	26	remainder: 1
2	13	remainder: 0
2	6	remainder: 1
2	3	remainder: 0
2	1	remainder: 1
2	0	remainder: 1
	0	remainder: 0

Write the remainder from bottom to top to get the binary number:
0 1 1 0 1 0 1 1

ACTIVITY 1B

Convert these denary numbers into binary (using either method).

a) 4 1
b) 6 7
c) 8 6
d) 1 0 0
e) 1 1 1
f) 1 2 7
g) 1 4 4
h) 1 8 9
i) 2 0 0
j) 2 5 5

Binary addition and subtraction

Up until now we have assumed all binary numbers have positive values. There are a number of methods to represent both positive and negative numbers. We will consider:

- one’s complement
- two’s complement.

In **one’s complement**, each digit in the binary number is inverted (in other words, 0 becomes 1 and 1 becomes 0). For example, 0 1 0 1 1 0 1 0 (denary value 90) becomes 1 0 1 0 0 1 0 1 (denary value −90).

In **two’s complement**, each digit in the binary number is inverted and a ‘1’ is added to the right-most bit. For example, 0 1 0 1 1 0 1 0 (denary value 90) becomes:

```
    1  0  1  0  0  1  0  1
+                         1
=   1  0  1  0  0  1  1  0   (since 1 + 1 = 0, a carry of 1) = denary value −90
```

Throughout the remainder of this chapter, we will use the two's complement method to avoid confusion. Also, two's complement makes binary addition and subtraction more straightforward. The reader is left to investigate one's complement and the **sign and magnitude** method in binary arithmetic.

Now that we are introducing negative numbers, we need a way to represent these in binary. The two's complement uses these weightings for an 8-bit number representation:

−128	64	32	16	8	4	2	1

This means:

−128	64	32	16	8	4	2	1
1	1	0	1	1	0	1	0
0	0	1	0	0	1	1	0

The first example is: −128 + 64 + 16 + 8 + 2 = −38

EXTENSION ACTIVITY 1A

Show the column headings for a system that uses 16 bits to represent a binary number.

The second example is: 32 + 4 + 2 = 38

The easiest way to convert a number into its negative equivalent is to use two's complement. For example, 104 in binary is 0 1 1 0 1 0 0 0.

To find the binary value for −104 using two's complement:

```
invert the digits:   1  0  0  1  0  1  1  1   (+104 in denary)
add 1:                                    1
which gives:         1  0  0  1  1  0  0  0   = −104)
```

ACTIVITY 1C

Convert these denary numbers into 8-bit binary numbers using two's complement where necessary. Use these binary column weightings:

−128	64	32	16	8	4	2	1

a) +114
b) +61
c) +96
d) −14
e) −116

Binary addition

Consider Examples 1.1 and 1.2.

Example 1.1

Add 0 0 1 0 0 1 0 1 (37 in denary) and 0 0 1 1 1 0 1 0 (58 in denary).

Solution

−128	64	32	16	8	4	2	1
0	0	1	0	0	1	0	1
				+			
0	0	1	1	1	0	1	0
				=			
0	1	0	1	1	1	1	1

This gives us 0 1 0 1 1 1 1 1, which is 95 in denary; the correct answer.

Example 1.2

Add 0 1 0 1 0 0 1 0 (82 in denary) and 0 1 0 0 0 1 0 1 (69 in denary).

Solution

−128	64	32	16	8	4	2	1
0	1	0	1	0	0	1	0
				+			
0	1	0	0	0	1	0	1
				=			
1	0	0	1	0	1	1	1

This gives us 1 0 0 1 0 1 1 1, which is –105 in denary (which is clearly nonsense). When adding two positive numbers, the result should always be positive (likewise, when adding two negative numbers, the result should always be negative). Here, the addition of two positive numbers has resulted in a negative answer. This is due to the result of the addition producing a number which is outside the range of values which can be represented by the 8 bits being used (in this case +127 is the largest value which can be represented, and the calculation produces the value 151, which is larger than 127 and, therefore, out of range). This causes overflow; it is considered in more detail in Chapter 13.

Binary subtraction

To carry out subtraction in binary, we convert the number being subtracted into its negative equivalent using two’s complement, and then *add* the two numbers.

Example 1.3

Carry out the subtraction 95 – 68 in binary.

Solution

1 Convert the two numbers into binary:

95 = 0 1 0 1 1 1 1 1

68 = 0 1 0 0 0 1 0 0

2 Find the two's complement of 68:

invert the digits:	1	0	1	1	1	0	1	1	
add 1:								1	
which gives:	1	0	1	1	1	1	0	0	= −68

3 Add 95 and −68:

	−128	64	32	16	8	4	2	1
	0	1	0	1	1	1	1	1
				+				
	1	0	1	1	1	1	0	0
				=				
1	0	0	0	1	1	0	1	1

The additional ninth bit is simply ignored leaving the binary number 0 0 0 1 1 0 1 1 (denary equivalent of 27, which is the correct result of the subtraction).

Example 1.4

Carry out the subtraction 49 – 80 in binary.

Solution

1 Convert the two numbers into binary:

49 = 0 0 1 1 0 0 0 1

80 = 0 1 0 1 0 0 0 0

2 Find the two's complement of 80:

invert the digits:	1	0	1	0	1	1	1	1	
add 1:								1	
which gives:	1	0	1	1	0	0	0	0	= −80

3 Add 49 and −80:

−128	64	32	16	8	4	2	1
0	0	1	1	0	0	0	1
			+				
1	0	1	1	0	0	0	0
			=				
1	1	1	0	0	0	0	1

This gives us 1 1 1 0 0 0 0 1, which is −31 in denary; the correct answer.

ACTIVITY 1D

Carry out these binary additions and subtractions using these 8-bit column weightings:

−128 64 32 16 8 4 2 1

a) 0 0 1 1 1 0 0 1 + 0 0 1 0 1 0 0 1
b) 0 1 0 0 1 0 1 1 + 0 0 1 0 0 0 1 1
c) 0 1 0 1 1 0 0 0 + 0 0 1 0 1 0 0 0
d) 0 1 1 1 0 0 1 1 + 0 0 1 1 1 1 1 0
e) 0 0 0 0 1 1 1 1 + 0 0 0 1 1 1 0 0
f) 0 1 1 0 0 0 1 1 − 0 0 1 1 0 0 0 0
g) 0 1 1 1 1 1 1 1 − 0 1 0 1 1 0 1 0
h) 0 0 1 1 0 1 0 0 − 0 1 0 0 0 1 0 0
i) 0 0 0 0 0 0 1 1 − 0 1 1 0 0 1 0 0
j) 1 1 0 1 1 1 1 1 − 1 1 0 0 0 0 1 1

Measurement of the size of computer memories

The byte is the smallest unit of memory in a computer. Some computers use larger bytes, such as 16-bit systems and 32-bit systems, but they are always multiples of 8. 1 byte of memory wouldn’t allow you to store very much information; so memory size is measured in these multiples. See Table 1.1.

Name of memory size	Equivalent denary value (bytes)
1 kilobyte (1 KB)	1 000
1 megabyte (1 MB)	1 000 000
1 gigabyte (1 GB)	1 000 000 000
1 terabyte (1 TB)	1 000 000 000 000
1 petabyte (1 PB)	1 000 000 000 000 000

Table 1.1 Memory size and denary values

The system of numbering shown in Table 1.1 only refers to some storage devices, but is technically inaccurate. It is based on the SI (base 10) system of units where 1 kilo is equal to 1000. A 1 TB hard disk drive would allow the storage of 1×10^{12} bytes according to this system. However, since memory size is actually measured in terms of powers of 2, another system has been proposed by the International Electrotechnical Commission (IEC); it is based on the binary system. See Table 1.2.

Name of memory size	Number of bytes	Equivalent denary value (bytes)
1 kibibyte (1 KiB)	2^{10}	1 024
1 mebibyte (1 MiB)	2^{20}	1 048 576
1 gibibyte (1 GiB)	2^{30}	1 073 741 824
1 tebibyte (1 TiB)	2^{40}	1 099 511 627 776
1 pebibyte (1 PiB)	2^{50}	1 125 899 906 842 624

Table 1.2 IEC memory size system

This system is more accurate. Internal memories (such as RAM) should be measured using the IEC system. A 64 GiB RAM could, therefore, store 64×2^{30} bytes of data (68 719 476 736 bytes).

See Section 1.2 for examples of how to calculate the size of a file.

1.1.3 Hexadecimal number system

The **hexadecimal** system is very closely related to the binary system. Hexadecimal (sometimes referred to as simply hex) is a base 16 system with the weightings:

1048576	65536	4096	256	16	1
(16^5)	(16^4)	(16^3)	(16^2)	(16^1)	(16^0)

Because it is a system based on 16 different digits, the numbers 0 to 9 and the letters A to F are used to represent hexadecimal digits.

A = 10, B = 11, C = 12, D = 13, E = 14 and F = 15.

Since 16 = 2^4, **four** binary digits are equivalent to each hexadecimal digit. Table 1.3 summarises the link between binary, hexadecimal and denary.

Binary value	Hexadecimal value	Denary value
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	8	8
1 0 0 1	9	9
1 0 1 0	A	10
1 0 1 1	B	11
1 1 0 0	C	12
1 1 0 1	D	13
1 1 1 0	E	14
1 1 1 1	F	15

Table 1.3 The link between binary, hexadecimal and denary

Converting from binary to hexadecimal and from hexadecimal to

binary

Converting from binary to hexadecimal is a fairly easy process. Starting from the right and moving left, split the binary number into groups of 4 bits. If the last group has less than 4 bits, then simply fill in with 0s from the left. Take each group of 4 bits and convert it into the equivalent hexadecimal digit using Table 1.3.

Examples 1.5 and 1.6 show you how this works.

Example 1.5

Convert 1 0 1 1 1 1 1 0 0 0 0 1 from binary to hexadecimal.

Solution

First split it into groups of 4 bits:

1 0 1 1	1 1 1 0	0 0 0 1

Then find the equivalent hexadecimal digits:

B	E	1

Example 1.6

Convert 1 0 0 0 0 1 1 1 1 1 1 1 0 1 from binary to hexadecimal.

Solution

First split it into groups of 4 bits:

1 0	0 0 0 1	1 1 1 1	1 1 0 1

The left group only contains 2 bits, so add in two 0s to the left:

0 0 1 0	0 0 0 1	1 1 1 1	1 1 0 1

Now find the equivalent hexadecimal digits:

2	1	F	D

ACTIVITY 1E

Convert these binary numbers into hexadecimal.

a) 1 1 0 0 0 0 1 1
b) 1 1 1 1 0 1 1 1
c) 1 0 0 1 1 1 1 1 1 1
d) 1 0 0 1 1 1 0 1 1 1 0
e) 0 0 0 1 1 1 1 0 0 0 0 1
f) 1 0 0 0 1 0 0 1 1 1 1 0
g) 0 0 1 0 0 1 1 1 1 1 1 1 1 0
h) 0 1 1 1 0 1 0 0 1 1 1 0 0

i) 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1
j) 0 0 1 1 0 0 1 1 1 1 0 1 0 1 1 1 0

Converting from hexadecimal to binary is also straightforward. Using the data from Table 1.3, simply take each hexadecimal digit and write down the 4 bit code which corresponds to the digit.

Example 1.7

Convert this hexadecimal number to its binary equivalent.

4 5 A

Solution

Using Table 1.3, find the 4-bit code for each digit:

0 1 0 0   0 1 0 1   1 0 1 0

Put the groups together to form the binary number:

0 1 0 0 0 1 0 1 1 0 1 0

Example 1.8

Convert this hexadecimal number to its binary equivalent.

B F 0 8

Solution

Using Table 1.3:

1 0 1 1   1 1 1 1   0 0 0 0   1 0 0 0

Then put all the digits together:

1 0 1 1 1 1 1 1 0 0 0 0 1 0 0 0

Use of the hexadecimal system

This section reviews two uses of the hexadecimal system.

Memory dumps

It is much easier to work with:

B 5 A 4 1 A F C

than it is to work with:

1 0 1 1 0 1 0 1 1 0 1 0 0 1 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 0 0

So, hexadecimal is often used when developing new software or when trying to trace errors in programs. When the memory contents are output to a printer or monitor, this is known as a

memory dump.

ACTIVITY 1F

Convert these hexadecimal numbers into binary.

a) 6 C

b) 5 9

c) A A

d) A 0 0

e) 4 0 E

f) B A 6

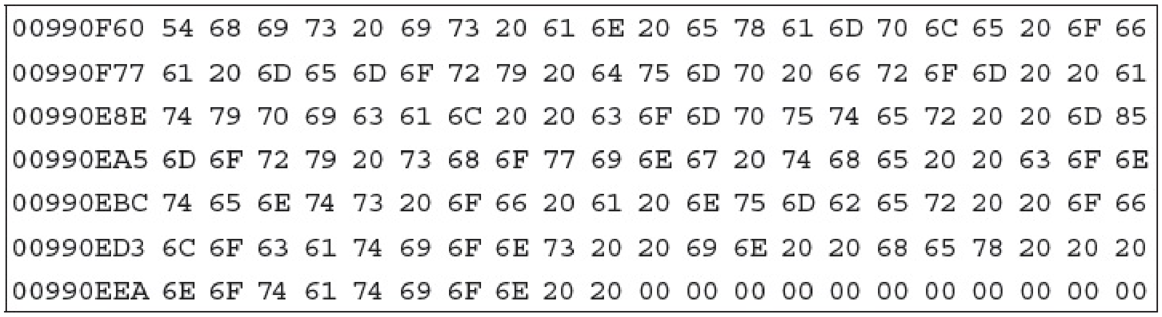
g) 9 C C

h) 4 0 A A

i) D A 4 7

j) 1 A B 0

A program developer can look at each of the hexadecimal codes (as shown in Table 1.4) and determine where the error lies. The value on the far left shows the memory location, so it is possible to find out exactly where in memory the fault occurs. Using hexadecimal is more manageable than binary. It is a powerful fault-tracing tool, but requires considerable knowledge of computer architecture to be able to interpret the results.



```
00990F60 54 68 69 73 20 69 73 20 61 6E 20 65 78 61 6D 70 6C 65 20 6F 66
00990F77 61 20 6D 65 6D 6F 72 79 20 64 75 6D 70 20 66 72 6F 6D 20 20 61
00990E8E 74 79 70 69 63 61 6C 20 20 63 6F 6D 70 75 74 65 72 20 20 6D 85
00990EA5 6D 6F 72 79 20 73 68 6F 77 69 6E 67 20 74 68 65 20 20 63 6F 6E
00990EBC 74 65 6E 74 73 20 6F 66 20 61 20 6E 75 6D 62 65 72 20 20 6F 66
00990ED3 6C 6F 63 61 74 69 6F 6E 73 20 20 69 6E 20 20 68 65 78 20 20 20
00990EEA 6E 6F 74 61 74 69 6F 6E 20 20 00 00 00 00 00 00 00 00 00 00 00
```

Table 1.4 Memory dump

1.1.4 Binary-coded decimal (BCD) system

The **binary-coded decimal (BCD)** system uses a 4-bit code to represent each denary digit:

0 0 0 0 = 0	0 1 0 1 = 5
0 0 0 1 = 1	0 1 1 0 = 6
0 0 1 0 = 2	0 1 1 1 = 7
0 0 1 1 = 3	1 0 0 0 = 8
0 1 0 0 = 4	1 0 0 1 = 9

Therefore, the denary number 3 1 6 5 would be 0 0 1 1 0 0 0 1 0 1 1 0 0 1 0 1 in BCD format.

The 4-bit code can be stored in the computer either as half a byte or two 4-bit codes stored together to form one byte. For example, using 3 1 6 5 again …

Method 1: four single bytes

0	0	0	0	0	0	1	1	3
0	0	0	0	0	0	0	1	1
0	0	0	0	0	1	1	0	6
0	0	0	0	0	1	0	1	5

Method 2: two bytes

0	0	1	1	0	0	0	1	3	1
0	1	1	0	0	1	0	1	6	5

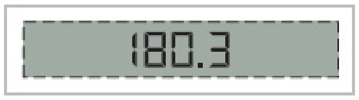
ACTIVITY 1G

1 Convert these denary numbers into BCD format.
   a) 2 7 1
   b) 5 0 0 6
   c) 7 9 9 0

2 Convert these BCD numbers into denary numbers.
   a) 1 0 0 1 0 0 1 1 0 1 1 1
   b) 0 1 1 1 0 1 1 1 0 1 1 0 0 0 1 0

Uses of BCD

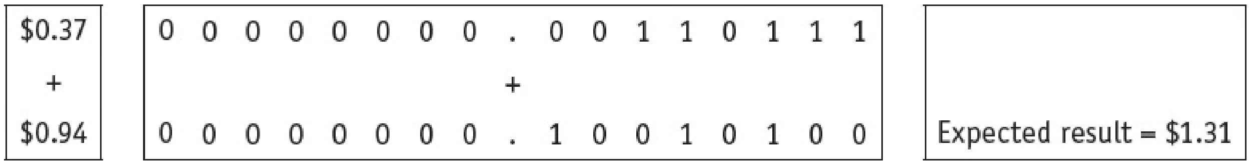
The most obvious use of BCD is in the representation of digits on a calculator or clock display.



Each denary digit will have a BCD equivalent value which makes it easy to convert from computer output to denary display.

As you will learn in Chapter 13, it is nearly impossible to represent decimal values exactly in computer memories which use the binary number system. Normally this doesn't cause a major issue since the differences can be dealt with. However, when it comes to accounting and representing monetary values in computers, *exact* values need to be stored to prevent significant errors from accumulating. Monetary values use a fixed-point notation, for example $1.31, so one solution is to represent each denary digit as a BCD value.

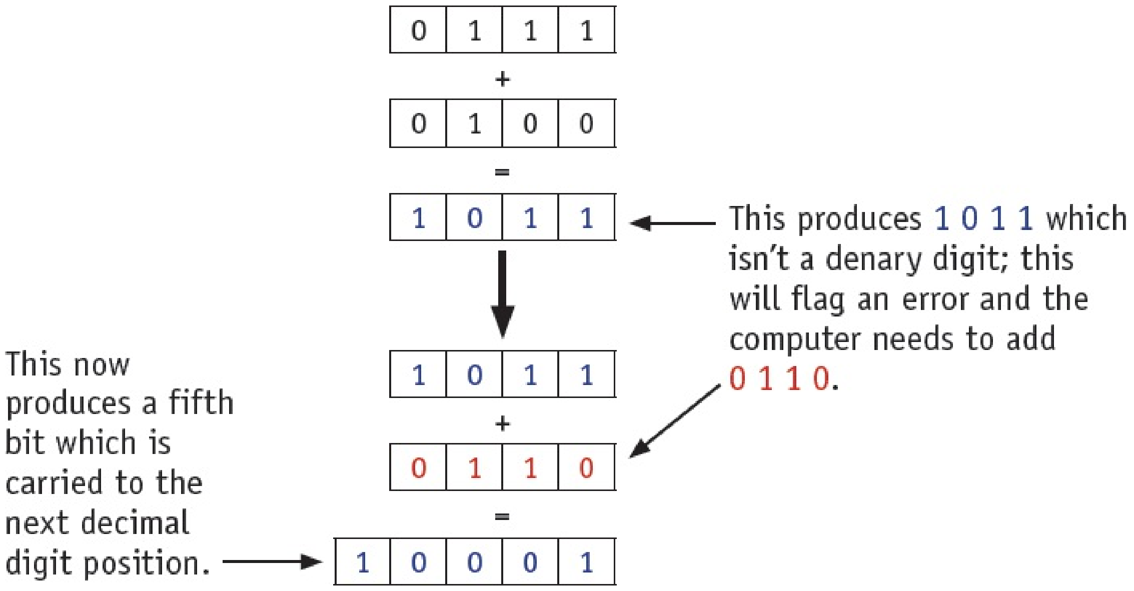
Consider adding $0.37 and $0.94 together using fixed-point decimals.



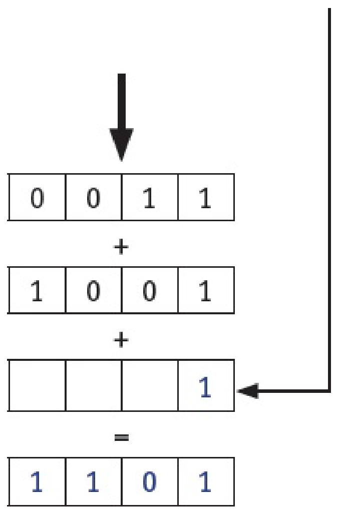
Using binary addition, this sum will produce:

0 0 0 0 0 0 0 0 . 1 1 0 0 1 0 1 1 which produces 1 1 0 0 (denary 12) and 1 0 1 1 (denary 11), which is clearly incorrect. The problem was caused by 3 + 9 = 12 and 7 + 4 = 11, as neither 12 nor 11 are single denary digits. The solution to this problem, enabling the computer to store monetary values accurately, is to add 0 1 1 0 (denary 6) whenever such a problem arises. The computer can be programmed to recognise this issue and add 0 1 1 0 at each appropriate point.

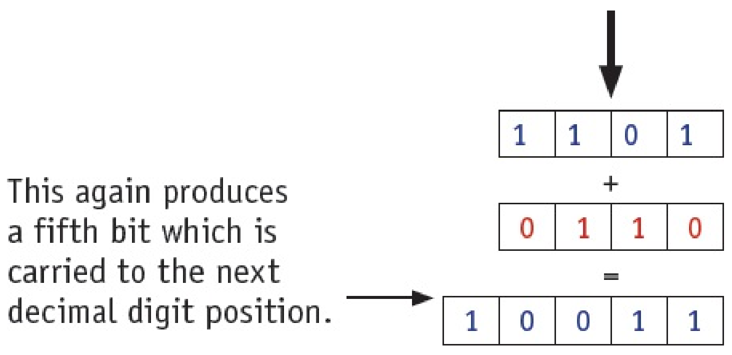
If we look at the example again, we can add .07 and .04 (the two digits in the second decimal place) first.


0 1 1 1
+
0 1 0 0
=
1 0 1 1
This produces 1 0 1 1 which isn't a denary digit; this will flag an error and the computer needs to add 0 1 1 0.
1 0 1 1
+
0 1 1 0
=
1 0 0 0 1
This now produces a fifth bit which is carried to the next decimal digit position.

Now we will add .3 and .9 together (the two digits in the first decimal place) remembering the carry bit from the addition above:

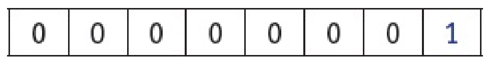


This produces 1 1 0 1 which isn't a denary digit; this will flag an error and the computer again needs to add 0 1 1 0.

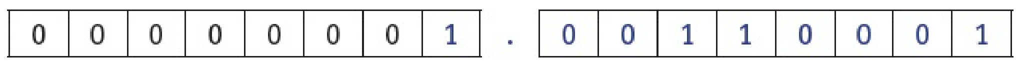


1 1 0 1
+
0 1 1 0
=
1 0 0 1 1
This again produces a fifth bit which is carried to the next decimal digit position.

Adding 1 to 0 0 0 0 0 0 0 0 produces:



Final answer:



which is 1.31 in denary – the correct answer.

ACTIVITY 1H

Carry out these BCD additions.

a) 0.45 + 0.21

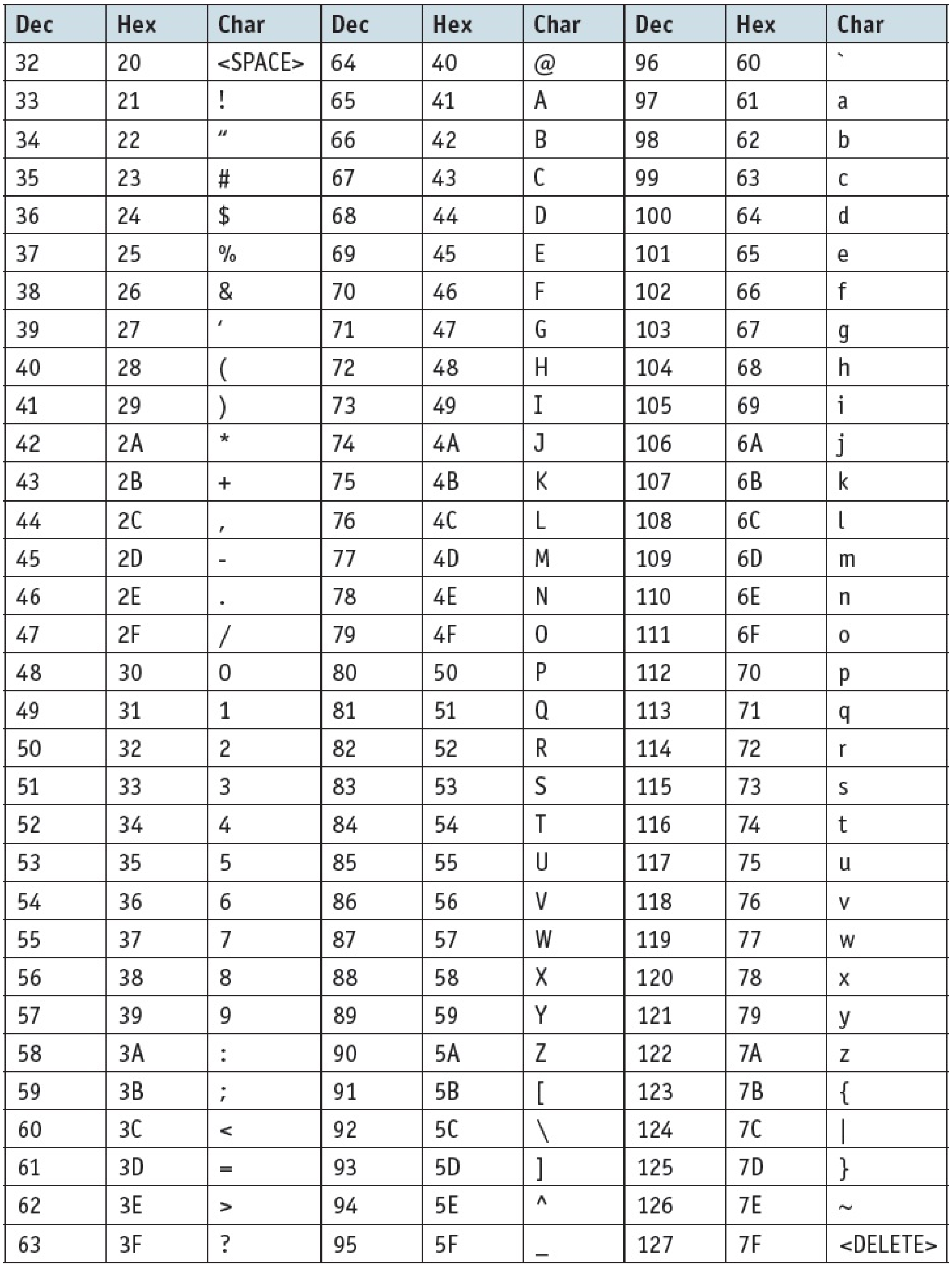
b) 0.66 + 0.51

c) 0.88 + 0.75

1.1.5 ASCII codes and Unicodes

The **ASCII code** system (American Standard Code for Information Interchange) was set up in 1963 for use in communication systems and computer systems. The newer version of the code was published in 1986. The standard ASCII code **character set** consists of 7-bit codes (0 to 127 denary or 0 to 7F in hexadecimal); this represents the letters, numbers and characters found on a standard keyboard together with 32 control codes (which use up codes 0 to 31 (denary) or 0 to 19 (hexadecimal)).

Table 1.5 shows part of the standard ASCII code table (only the control codes have been removed from the table).



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	<SPACE>	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	“	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	‘	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	<DELETE>

▲ **Table 1.5** Part of the ASCII code table

Notice the storage of characters with uppercase and lowercase. For example:

a	1	1	0	0	0	0	1	hex 61 (lower case)
A	1	0	0	0	0	0	1	hex 41 (upper case)
y	1	1	1	1	0	0	1	hex 79 (lower case)
Y	1	0	1	1	0	0	1	hex 59 (uppercase)

Notice the sixth bit changes from 1 to 0 when comparing lower and uppercase characters. This makes the conversion between the two an easy operation. It is also noticeable that the character sets (such as a to z, 0 to 9, and so on) are grouped together in sequence, which speeds up usability.

Extended ASCII uses 8-bit codes (128 to 255 in denary or 80 to FF in hex). This allows for non-English characters and for drawing characters to be included.

Since ASCII code has a number of disadvantages and is unsuitable for some purposes, different methods of coding have been developed over the years. One coding system is called **Unicode**. Unicode allows characters in a code form to represent all languages of the world, thus supporting many operating systems, search engines and internet browsers used globally. There is overlap with standard ASCII code, since the first 128 (English) characters are the same, but Unicode can support several thousand different characters in total. As can be seen in Tables 1.5 and 1.6, ASCII uses one byte to represent a character, whereas Unicode will support up to four bytes per character.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	154	9A	Ü	180	B4	┤	206	CE	╬	232	E8	Φ
129	81	ü	155	9B	¢	181	B5	╡	207	CF	╧	233	E9	Θ
130	82	é	156	9C	£	182	B6	╢	208	D0	╨	234	EA	Ω
131	83	â	157	9D	¥	183	B7	╖	209	D1	╤	235	EB	δ
132	84	ä	158	9E	Pts	184	B8	╕	210	D2	╥	236	EC	∞
133	85	à	159	9F	ƒ	185	B9	╣	211	D3	╙	237	ED	ø
134	86	å	160	A0	á	186	BA	║	212	D4	╘	238	EE	ε
135	87	ç	161	A1	í	187	BB	╗	213	D5	╒	239	EF	∩
136	88	ê	162	A2	ó	188	BC	╝	214	D6	╓	240	F0	≡
137	89	ë	163	A3	ú	189	BD	╜	215	D7	╫	241	F1	±
138	8A	è	164	A4	ñ	190	BE	╛	216	D8	╪	242	F2	≥
139	8B	ï	165	A5	Ñ	191	BF	┐	217	D9	┘	243	F3	≤
140	8C	î	166	A6	ª	192	C0	└	218	DA	┌	244	F4	⌠
141	8D	ì	167	A7	º	193	C1	┴	219	DB	█	245	F5	⌡
142	8E	Ä	168	A8	¿	194	C2	┬	220	DC	▄	246	F6	÷
143	8F	Å	169	A9	⌐	195	C3	├	221	DD	▌	247	F7	≈
144	90	É	170	AA	¬	196	C4	─	222	DE	▐	248	F8	°
145	91	æ	171	AB	½	197	C5	┼	223	DF	▀	249	F9	▪▪
146	92	Æ	172	AC	¼	198	C6	╞	224	E0	α	250	FA	▪
147	93	ô	173	AD	¡	199	C7	╟	225	E1	ß	251	FB	√
148	94	ö	174	AE	«	200	C8	╚	226	E2	Γ	252	FC	^{3}
149	95	ò	175	AF	»	201	C9	╔	227	E3	π	253	FD	^{2}
150	96	û	176	B0	░	202	CA	╩	228	E4	Σ	254	FE	■
151	97	ù	177	B1	▒	203	CB	╦	229	E5	σ	255	FF	□
152	98	ÿ	178	B2	▓	204	CC	╠	230	E6	µ			
153	99	Ö	179	B3	│	205	CD	═	231	E7	τ			

▲ **Table 1.6** Extended ASCII code table

The Unicode consortium was set up in 1991. Version 1.0 was published with five goals, these were to

- create a universal standard that covered all languages and all writing systems
- produce a more efficient coding system than ASCII
- adopt uniform encoding where each character is encoded as 16-bit or 32-bit code
- create unambiguous encoding where each 16-bit or 32-bit value always represents the same character (it is worth pointing out here that the ASCII code tables are not standardised and versions other than the ones shown in tables 1.5 and 1.6 exist)
- reserve part of the code for private use to enable a user to assign codes for their own characters and symbols (useful for Chinese and Japanese character sets).

A sample of Unicode characters are shown in Table 1.7. As can be seen from the table, characters used in languages such as Russian, Greek, Romanian and Croatian can now be represented in a computer).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
01A0	Ơ	ơ	Ƣ	ƣ	Ƥ	ƥ	Ʀ	Ƨ	ƨ	Ʃ	ƪ	ƫ	Ƭ	ƭ	Ʈ	Ư
01B0	ư	Ʊ	Ʋ	Ƴ	ƴ	Ƶ	ƶ	Ʒ	Ƹ	ƹ	ƺ	ƻ	Ƽ	ƽ	ƾ	ƿ
01C0	ǀ	ǁ	ǂ	ǃ	Ǆ	ǅ	ǆ	Ǉ	ǈ	ǉ	Ǌ	ǋ	ǌ	Ǎ	ǎ	Ǐ
01D0	ǐ	Ǒ	ǒ	Ǔ	ǔ	Ǖ	ǖ	Ǘ	ǘ	Ǚ	ǚ	Ǜ	ǜ	ǝ	Ǟ	ǟ
01E0	Ǡ	ǡ	Ǣ	ǣ	Ǥ	ǥ	Ǧ	ǧ	Ǩ	ǩ	Ǫ	ǫ	Ǭ	ǭ	Ǯ	ǯ
01F0	ǰ	Ǳ	ǲ	ǳ	Ǵ	ǵ	Ƕ	Ƿ	Ǹ	ǹ	Ǻ	ǻ	Ǽ	ǽ	Ǿ	ǿ
0200	Ȁ	ȁ	Ȃ	ȃ	Ȅ	ȅ	Ȇ	ȇ	Ȉ	ȉ	Ȋ	ȋ	Ȍ	ȍ	Ȏ	ȏ
0210	Ȑ	ȑ	Ȓ	ȓ	Ȕ	ȕ	Ȗ	ȗ	Ș	ș	Ț	ț	Ȝ	ȝ	Ȟ	ȟ
0220	Ƞ	ȡ	Ȣ	ȣ	Ȥ	ȥ	Ȧ	ȧ	Ȩ	ȩ	Ȫ	ȫ	Ȭ	ȭ	Ȯ	ȯ
0230	Ȱ	ȱ	Ȳ	ȳ	ȴ	ȵ	ȶ	ȷ	ȸ	ȹ	Ⱥ	Ȼ	ȼ	Ƚ	Ⱦ	ȿ
0240	ɀ	Ɂ	ɂ	Ƀ	Ʉ	Ʌ	Ɇ	ɇ	Ɉ	ɉ	Ɋ	ɋ	Ɍ	ɍ	Ɏ	ɏ
0250	ɐ	ɑ	ɒ	ɓ	ɔ	ɕ	ɖ	ɗ	ɘ	ə	ɚ	ɛ	ɜ	ɝ	ɞ	ɟ
0260	ɠ	ɡ	ɢ	ɣ	ɤ	ɥ	ɦ	ɧ	ɨ	ɩ	ɪ	ɫ	ɬ	ɭ	ɮ	ɯ
0270	ɰ	ɱ	ɲ	ɳ	ɴ	ɵ	ɶ	ɷ	ɸ	ɹ	ɺ	ɻ	ɼ	ɽ	ɾ	ɿ
0280	ʀ	ʁ	ʂ	ʃ	ʄ	ʅ	ʆ	ʇ	ʈ	ʉ	ʊ	ʋ	ʌ	ʍ	ʎ	ʏ
0290	ʐ	ʑ	ʒ	ʓ	ʔ	ʕ	ʖ	ʗ	ʘ	ʙ	ʚ	ʛ	ʜ	ʝ	ʞ	ʟ
02A0	ʠ	ʡ	ʢ	ʣ	ʤ	ʥ	ʦ	ʧ	ʨ	ʩ	ʪ	ʫ	ʬ	ʭ	ʮ	ʯ
02B0	ʰ	ʱ	ʲ	ʳ	ʴ	ʵ	ʶ	ʷ	ʸ	ʹ	ʺ	ʻ	ʼ	ʽ	ʾ	ʿ

▲ **Table 1.7** Sample of Unicode characters

1.2 Multimedia

Key terms

Bit-map image – system that uses pixels to make up an image.

Pixel – smallest picture element that makes up an image.

Colour depth – number of bits used to represent the colours in a pixel, e.g. 8 bit colour depth can represent $2^8 = 256$ colours.

Bit depth – number of bits used to represent the smallest unit in, for example, a sound or image file – the larger the bit depth, the better the quality of the sound or colour image.

Image resolution – number of pixels that make up an image, for example, an image could contain 4096 × 3192 pixels (12 738 656 pixels in total).

Screen resolution – number of horizontal and vertical pixels that make up a screen display. If the screen resolution is smaller than the image resolution, the whole image cannot be shown on the screen, or the original image will become lower quality.

Resolution – number of pixels per column and per row on a monitor or television screen.

Pixel density – number of pixels per square centimetre.

Vector graphics – images that use 2D points to describe lines and curves and their properties that are grouped to form geometric shapes.

Sampling resolution – number of bits used to represent sound amplitude (also known as bit depth).

Sampling rate – number of sound samples taken per second.

Frame rate – number of video frames that make up a video per second.

Images can be stored in a computer in two common formats: bit-map image and vector graphic.

1.2.1 Bit-map images

Bit-map images are made up of **pixels** (picture elements); the image is stored in a two-dimensional matrix of pixels.

Pixels can take different shapes, such as ▮▮▮ or ●.

When storing images as pixels, we have to consider

- at least 8 bits (1 byte) per pixel are needed to code a coloured image (this gives 256 possible colours by varying the intensity of the blue, green and red elements)
- true colour requires 3 bytes per pixel (24 bits), which gives more than one million colours
- the number of bits used to represent a pixel is called the **colour depth**.

EXTENSION ACTIVITY 1B

Find out how HTML is used to control the colour of each pixel on a screen. How is HTML used in the design stage of a web page screen layout?

In terms of images, we need to distinguish between **bit depth** and colour depth; for example, the number of bits that are used to represent a single pixel (bit depth) will determine the colour depth of that pixel. As the bit depth increases, the number of possible colours which can be represented also increases. For example, a bit depth of 8 bits per pixel allows 256 (2^8) different colours (the colour depth) to be represented, whereas using a bit depth of 32 bits per pixel results in 4 294 967 296 (2^{32}) different colours. The impact of bit depth and colour depth is considered later.

We will now consider the actual image itself and how it can be displayed on a screen. There are two important definitions here:

- **Image resolution** refers to the number of pixels that make up an image; for example, an image could contain 4096 × 3192 pixels (12 738 656 pixels in total).
- **Screen resolution** refers to the number of horizontal pixels and the number of vertical pixels that make up a screen display (for example, if the screen resolution is smaller than the image resolution then the whole image cannot be shown on the screen or the original image will now be a lower quality).

We will try to clarify the difference by using an example.

Figure 1.1 has been taken by a digital camera using an image resolution of 4096 × 3192 pixels:

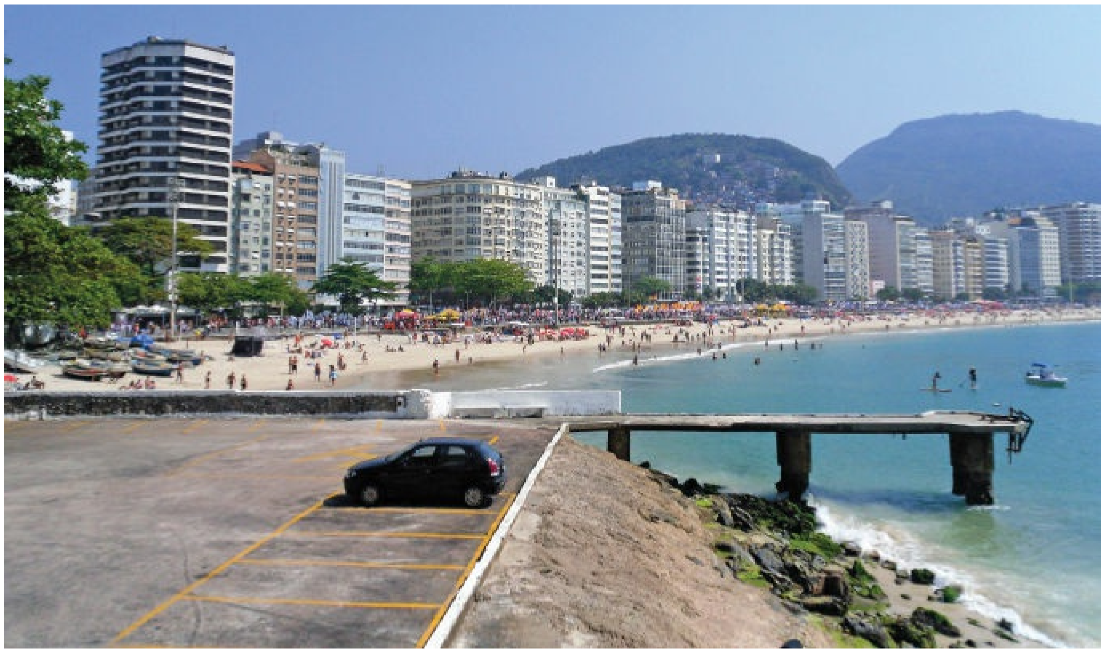


Figure 1.1 Image taken by a digital camera

Suppose we wish to display Figure 1.1 on a screen with screen resolution of 1920 × 1080. To display this image the web browser (or other software) would need to re-size Figure 1.1 so that it now fits the screen. This could be done by removing pixels so that it could now be displayed, or part of the image could be cropped (and, in this case, rotated through 90°) as shown in Figure 1.2.

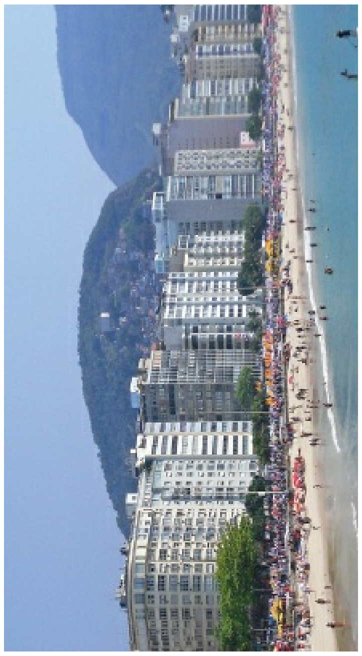


Figure 1.2 Image cropped and rotated through 90°

However, a lower **resolution** copy of Figure 1.1 (for example, 1024 × 798) would now fit on the screen without any modification to the image. We could simply zoom in to enlarge it to full screen size; however, the image could now become pixelated (in other words, the number of pixels per square inch (known as the **pixel density**) is smaller, causing deterioration in the image quality).

We will now consider a calculation which shows how pixel density can be calculated for a given screen. Imagine we are using an Apple iPhone 8 which has 5.5-inch screen size and screen resolution of 1920 pixels × 1080 pixels:

1 add together the squares of the resolution size (($1920^2 + 1080^2$) = (3 686 400 + 16 640) = 4 852 800)

2 find the square root $\left(\sqrt{4852800} = 2202.907\right)$

3 divide by screen size (2202.907 ÷ 5.5 = 401)

This gives us the pixel density of 401 pixels per square inch (ppi) (which is the same as the published figure from the manufacturer).

A pixel-generated image can be scaled up or scaled down; it is important to understand that this can be done when deciding on the resolution. The resolution can be varied on many cameras before taking, for example, a digital photograph. When magnifying an image, the number of pixels that makes up the image remains the same but the area they cover is now increased. This means some of the sharpness could be lost. This is known as the pixel density and is key when scaling up photographs. For example, look at Figure 1.3.

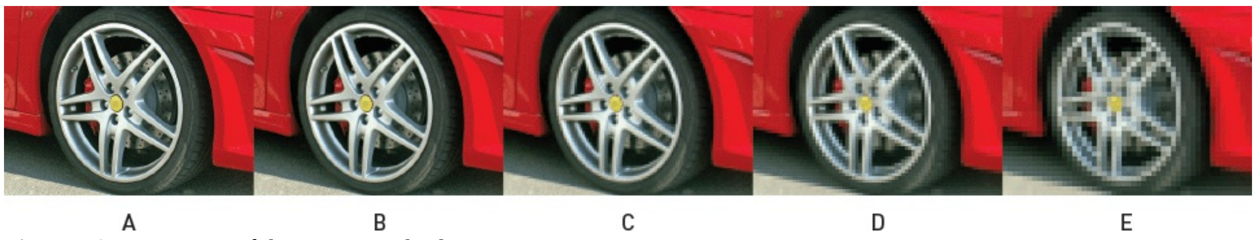


Figure 1.3 Five images of the same car wheel

Image A is the original. By the time it has been scaled up to make image E it has become pixelated ('fuzzy'). This is because images A and E have different pixel densities.

The main drawback of using high resolution images is the increase in file size. As the number of pixels used to represent the image is increased, the size of the file will also increase. This impacts on how many images can be stored on, for example, a hard drive. It also impacts on the time to download an image from the internet or the time to transfer images from device to device. Bit-map images rely on certain properties of the human eye and, up to a point, the amount of file compression used (see Section 1.3 File compression). The eye can tolerate a certain amount of resolution reduction before the loss of quality becomes significant.

Calculating bit-map image file sizes

It is possible to estimate the file size needed to store a bit-map image. The file size will need to take into account the image resolution and bit depth.

For example, a full screen with a resolution of 1920 × 1080 pixels and a bit depth of 24 requires 1920 × 1080 × 24 bits = 49 766 400 bits for the full screen image.

EXTENSION ACTIVITY 1C

Calculate the file size needed to store the screen image on a UHD television.

Dividing by 8 gives us 6 220 800 bytes (equivalent to 6.222 MB using the SI units or 5.933 MiB using IEE units). An image which does not occupy the full screen will obviously result in a smaller file size.

Note: when saving a bit-map image, it is important to include a file header; this will contain items such as file type (.bmp or .jpeg), file size, image resolution, bit depth (usually 1, 8, 16, 24 or 32), any type of data compression employed and so on.

1.2.2 Vector graphics

Vector graphics are images that use 2D points to describe lines and curves and their properties that are grouped to form geometric shapes. Vector graphics can be designed using computer aided design (CAD) software or using an application which uses a drawing canvas on the screen. See Figure 1.4.

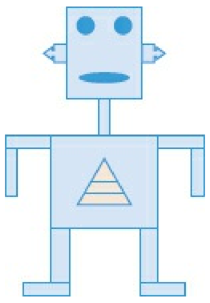


Figure 1.4 Drawing of a robot made up of a number of geometric shapes

A vector graphic will contain a drawing list (included in a file header) that is made up of

- the command used for each object that makes up the graphic image
- the attributes that define the properties that make up each object (for example consider the ellipse of the robot's mouth – this will need the position of the two centres, the radius from centres, the thickness and style of each line, the line colour and any fill colour used)
- the relative position of each object will also need to be included
- the dimensions of each object are not defined, but the relative positions of objects to each other in the final graphic need to be defined; this means that scaling up the vector graphic image will result in no loss of quality.

When printing out vector graphics it is usually necessary to first convert it into a bit-map image to match the format of most printers.

Comparison between vector graphics and bit-map images

Vector graphic images	**Bit-map images**
made up of geometric shapes which require definition/attributes	made up of tiny pixels of different colours
to alter/edit the design, it is necessary to change each of the geometric shapes	possible to alter/edit each of the pixels to change the design of the image
they do not require large file size since it is made up of simple geometric shapes	because of the use of pixels (which give very accurate designs), the file size is very large
because the number of geometric shapes is limited, vector graphics are not usually very realistic	since images are built up pixel by pixel, the final image is usually very realistic

file formats are usually .svg, .cgm, .odg	file formats are usually .jpeg, .bmp, .png

Table 1.8 Comparison between vector graphics and bit-map images

It is now worth considering whether a vector graphic or a bit-map image would be the best choice for a given application. When deciding which is the better method, we should consider the following:

- Does the image need to be resized? If so, a vector graphic could be the best option.
- Does the image need to be drawn to scale? Again, a vector graphic is probably the best option.
- Does the image need to look real? Usually bit-map images look more realistic than vector graphics.
- Are there file restrictions? If so, it is important to consider whether vector graphic images can be used; if not, it would be necessary to consider the image resolution of a bit-map image to ensure the file size is not too large.

For example, when designing a logo for a company or composing an ‘exploded diagram’ of a car engine, vector graphics are the best choice.

However, when modifying photographs using photo software, the best method is to use bit-map images.

1.2.3 Sound files

Sound requires a medium in which to travel through (it cannot travel in a vacuum). This is because it is transmitted by causing oscillations of particles within the medium. The human ear picks up these oscillations (changes in air pressure) and interprets them as sound. Each sound wave has a frequency and wavelength; the amplitude specifies the loudness of the sound.

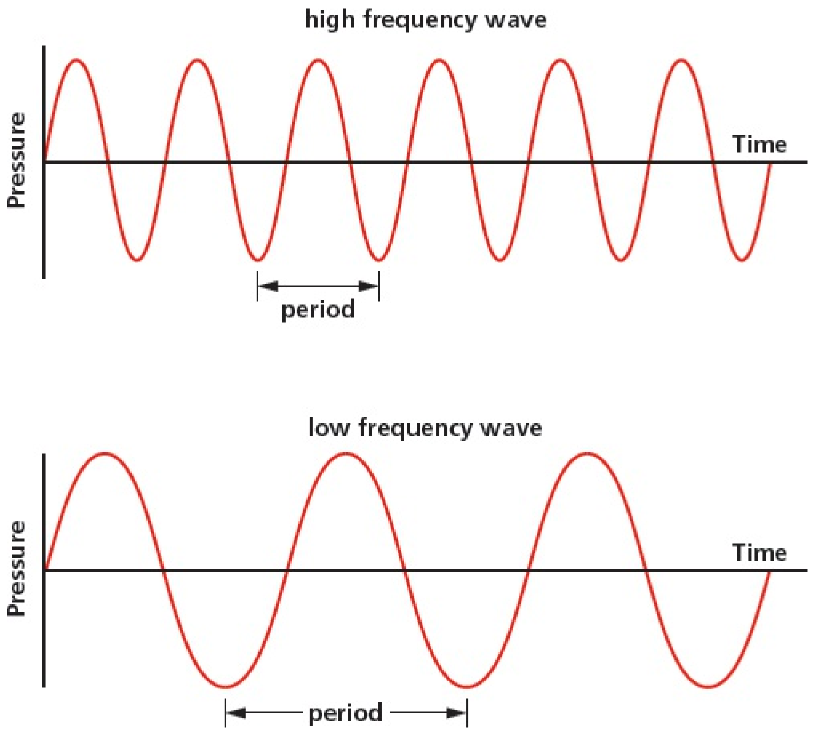


Figure 1.5 High and low frequency wave signals

Sound is an analogue value; this needs to be digitised in order to store sound in a computer. This is done using an analogue to digital converter (ADC). If the sound is to be used as a music file, it is often filtered first to remove higher frequencies and lower frequencies which are outside the range of human hearing. To convert the analogue data to digital, the sound waves are sampled at a given time rate. The amplitude of the sound cannot be measured precisely, so approximate values are stored.

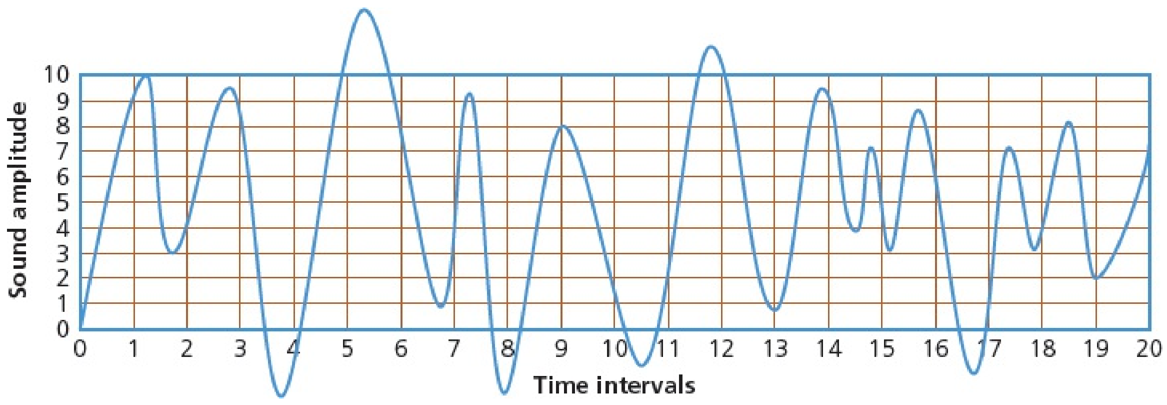


Figure 1.6 A sound wave

Figure 1.6 shows a sound wave. The *x*-axis shows the time intervals when the sound was sampled (0 to 20), and the *y*-axis shows the amplitude of the sampled sound (the amplitudes above 10 and below 0 are filtered out in this example).

At time interval 1, the approximate amplitude is 9; at time interval 2, the approximate amplitude is 4, and so on for all 20 time intervals. Because the amplitude range in Figure 1.6 is 0 to 10, then 4 binary bits can be used to represent each amplitude value (for example, 9 would be represented by the binary value 1001). Increasing the number of possible values used to represent sound amplitude also increases the accuracy of the sampled sound (for example, using a range of 0 to 127 gives a much more accurate representation of the sound sample than using a range of, for example, 0 to 10). This is known as the **sampling resolution** (also known as the bit depth).

Sampling rate is the number of sound samples taken per second. The higher the sampling rate and/or sampling resolution, the greater the file size. For example, a 16-bit sampling resolution is used when recording CDs to give better sound quality.

So, how is sampling used to record a sound clip?

- The amplitude of the sound wave is first determined at set time intervals (the sampling rate).
- This gives an approximate representation of the sound wave.
- The sound wave is then encoded as a series of binary digits.

Using a higher sampling rate or larger resolution will result in a more faithful representation of the original sound source.

Pros	**Cons**
larger dynamic range	produces larger file size
better sound quality	takes longer to transmit/download sound files
less sound distortion	requires greater processing power

Table 1.9 The pros and cons of using a larger sampling resolution when recording sound

Recorded sound is often edited using software. Common features of such software include the ability to

- edit the start/stop times and duration of a sample
- extract and save (or delete) part of a sample
- alter the frequency and amplitude of a sample
- fade in and fade out
- mix and/or merge multiple sound tracks or sources
- combine various sound sources together and alter their properties
- remove ‘noise’ to enhance one sound wave in a multiple of waves (for example, to identify and extract one person’s voice out of a group of people)
- convert between different audio formats.

1.2.4 Video

This section considers the use of video and extends beyond the syllabus. While this is not specifically mentioned in the syllabus, it has been included here for completeness. Many specialist video cameras exist. However, most digital cameras, smart phones and tablets are also capable of taking moving images by ‘stitching’ a number of still photos (frames) together. They are often referred to as DV (digital video) cameras; they store compressed photo frames at a speed of 25 MB per second – this is known as motion JPEG.

In both single frame and video versions, the camera picks up the light from the image and turns it into an electronic signal using light-sensitive sensors. In the case of the DV cameras, these signals are automatically converted into a compressed digital file format.

When recording video, the **frame rate** refers to the number of frames recorded per second.

1.3 File compression

Key terms

Lossless file compression – file compression method where the original file can be restored following decompression.

Lossy file compression – file compression method where parts of the original file cannot be recovered during decompression, so some of the original detail is lost.

JPEG – Joint Photographic Expert Group – a form of lossy file compression based on the inability of the eye to spot certain colour changes and hues.

MP3/MP4 files – file compression method used for music and multimedia files.

Audio compression – method used to reduce the size of a sound file using perceptual music shaping.

Perceptual music shaping – method where sounds outside the normal range of hearing of humans, for example, are eliminated from the music file during compression.

Bit rate – number of bits per second that can be transmitted over a network. It is a measure of the data transfer rate over a digital telecoms network.

Run length encoding (RLE) – a lossless file compression technique used to reduce text and photo files in particular.

It is often necessary to reduce the file size of a file to either save storage space or to reduce the time taken to stream or transmit data from one device to another (see Chapter 2). The two most common forms of file compression are **lossless file compression** and **lossy file compression**.

Lossless file compression

With this technique, all the data from the original file can be reconstructed when the file is uncompressed again. This is particularly important for files where loss of any data would be disastrous (such as a spreadsheet file of important results).

Lossy file compression

With this technique, the file compression algorithm eliminates unnecessary data (as with MP3 and **JPEG** formats, for example).

Lossless file compression is designed to lose none of the original detail from the file (such as Run-Length Encoding (RLE) which is covered later in this chapter). Lossy file compression usually results in some loss of detail when compared to the original; it is usually impossible to reconstruct the original file. The algorithms used in the lossy technique have to decide which parts of the file are important (and need to be kept) and which parts can be discarded.

We will now consider file compression techniques applied to multimedia files.

1.3.1 File compression applications

MPEG-3 (MP3) and MPEG-4 (MP4)

MPEG-3 (MP3) uses technology known as **audio compression** to convert music and other sounds into an MP3 file format. Essentially, this compression technology will reduce the size of a normal music file by about 90%. For example, an 80 MB music file on a CD can be reduced to 8 MB using MP3 technology.

MP3 files are used in MP3 players, computers or mobile phones. Music files can be downloaded or streamed from the internet in a compressed format, or CD files can be converted to MP3 format. While streamed or MP3 music quality can never match the 'full' version found on a CD, the quality is satisfactory for most purposes.

But how can the original music file be reduced by 90% while still retaining most of the music quality? This is done using file compression algorithms that use **perceptual music shaping.**

Perceptual music shaping removes certain sounds. For example

- frequencies that are outside the human hearing range
- if two sounds are played at the same time, only the louder one can be heard by the ear, so the softer sound is eliminated.

This means that certain parts of the music can be removed without affecting the quality too much. MP3 files use what is known as a lossy format, since part of the original file is lost following the compression algorithm. This means that the original file cannot be put back together again. However, even the quality of MP3 files can be different, since it depends on the **bit rate** – this refers to the number of bits per second used when creating the file. Bit rates are between 80 and 320 kilobits per second; usually 200 kilobits or higher gives a sound quality close to a normal CD.

MPEG-4 (MP4) files are slightly different to MP3 files. This format allows the storage of multimedia files rather than just sound. Music, videos, photos and animation can all be stored in the MP4 format. Videos, for example, could be streamed over the internet using the MP4 format without losing any real discernible quality (see Chapter 2 for notes on video streaming).

EXTENSION ACTIVITY 1D

Find out how file compression can be applied to a photograph without noticeably reducing its quality. Compare this to run-length encoding (RLE), described below.

Photographic (bit-map) images

When a photographic file is compressed, both the file size and quality of image are reduced. A common file format for images is JPEG, which uses lossy file compression. Once the image is subjected to the JPEG compression algorithm, a new file is formed and the original file can no longer be constructed. A JPEG will reduce the raw bit-map image by a factor of between 5 and 15, depending on the quality of the original.

Vector graphics can also undergo some form of file compression. Scalable vector graphics (.svg) are defined in XML text files which, therefore, allows them to be compressed.

Run-length encoding (RLE)

Run-length encoding (RLE) can be used to compress a number of different file formats.

It is a form of lossless/reversible file compression that reduces the size of a string of adjacent, identical data (such as repeated colours in an image).

A repeating string is encoded into two values.

The first value represents the number of identical data items (such as characters) in the run. The second value represents the code of the data item (such as ASCII code if it is a keyboard character).

RLE is only effective where there is a long run of repeated units/bits.

Using RLE on text data

Consider the text string ‘aaaaabbbbccddddd’.

Assuming each character requires 1 byte, then this string needs 16 bytes. If we assume ASCII code is being used, then the string can be coded as follows:

a	a	a	a	a	b	b	b	b	c	c	d	d	d	d	d
05 97					04 98				02 99		05 100				

This means we have five characters with ASCII code 97, four characters with ASCII code 98, two characters with ASCII code 99, and five characters with ASCII code 100. Assuming each number in the second row requires 1 byte of memory, the RLE code will need 8 bytes. This is half the original file size.

One issue occurs with a string such as ‘cdcdcdcdcd’, where compression is not very effective. To cope with this we use a flag. A flag preceding data indicates that what follows are the number of repeating units (for example, 255 05 97 where 255 is the flag and the other two numbers indicate that there are five items with ASCII code 97). When a flag is not used, the next byte(s) are taken with their face value and a run of 1 (for example, 01 99 means one character with ASCII code 99 follows).

Consider this example:

String	aaaaaaaa	bbbbbbbbbb	c	d	c	d	c	d	eeeeeeee
Code	08 97	10 98	01 99	01 100	01 99	01 100	01 99	01 100	08 101

The original string contains 32 characters and would occupy 32 bytes of storage.

The coded version contains 18 values and would require 18 bytes of storage.

Introducing a flag (255 in this case) produces:

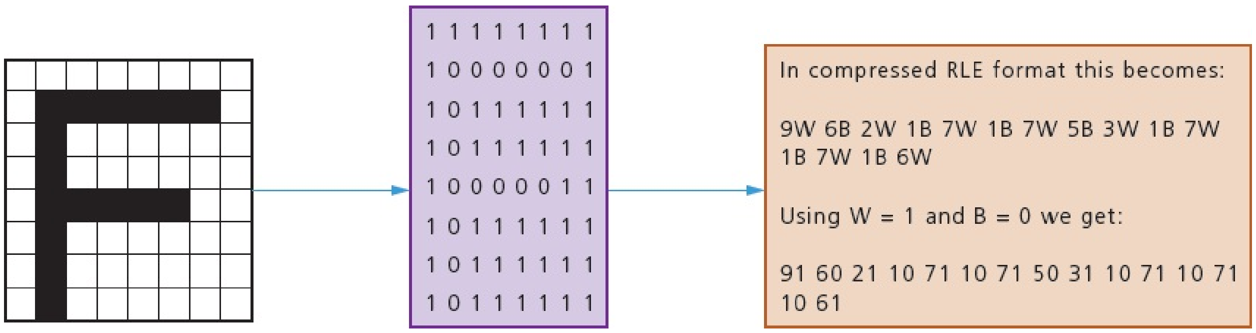
255 08 97 255 10 98 99 100 99 100 99 100 255 08 101

This has 15 values and would, therefore, require 15 bytes of storage. This is a reduction in file size of about 53%.

Using RLE with images

Black and white images

Figure 1.7 shows the letter F in a grid where each square requires 1 byte of storage. A white square has a value 1 and a black square a value of 0.



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

In compressed RLE format this becomes:

9W 6B 2W 1B 7W 1B 7W 5B 3W 1B 7W 1B 7W 1B 6W

Using W = 1 and B = 0 we get:

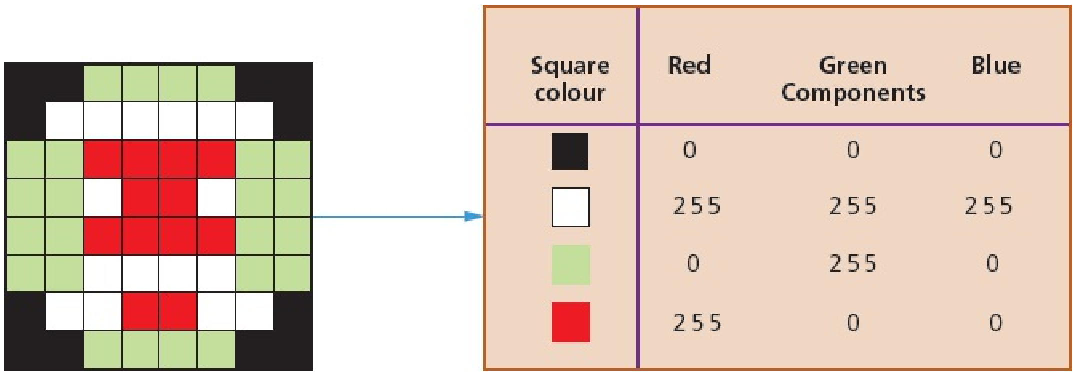
91 60 21 10 71 10 71 50 31 10 71 10 71 10 61

Figure 1.7 Using RLE with a black and white image

The 8 × 8 grid would need 64 bytes; the compressed RLE format has 30 values, and therefore needs only 30 bytes to store the image.

Coloured images

Figure 1.8 shows an object in four colours. Each colour is made up of red, green and blue (RGB) according to the code on the right.



Square colour	Red	Green Components	Blue
⬛	0	0	0
⬜	255	255	255
🟩	0	255	0
🟥	255	0	0

Figure 1.8 Using RLE with a coloured image

This produces the following data:

2 0 0 0 4 0 255 0 3 0 0 0 6 255 255 255 1 0 0 0 2 0 255 0 4 255 0 0 4 0 255 0 1 255 255 255 2 255 0 0 1 255 255 255 4 0 255 0 4 255 0 0 4 0 255 0 4 255 255 255 2 0 255 0 1 0 0 0 2 255 255 255 2 255 0 0 2 255 255 255 3 0 0 0 4 0 255 0 2 0 0 0

The original image (8 × 8 square) would need 3 bytes per square (to include all three RGB values). Therefore, the uncompressed file for this image is 8 × 8 × 3 = 192 bytes.

The RLE code has 92 values, which means the compressed file will be 92 bytes in size. This gives a file reduction of about 52%. It should be noted that the file reductions in reality will not be as large as this due to other data which needs to be stored with the compressed file (such as a file header).

1.3.2 General methods of compressing files

All the above file compression techniques are excellent for very specific types of file. However, it is also worth considering some general methods to reduce the size of a file without the need to use lossy or lossless file compression:

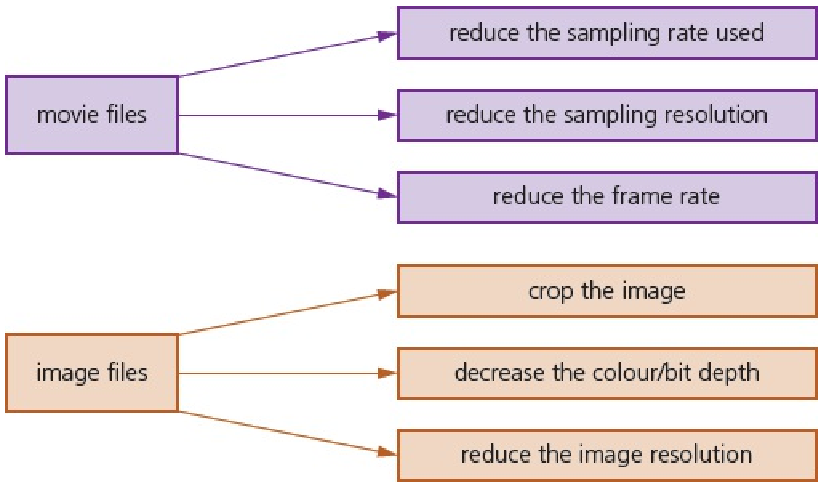

movie files
reduce the sampling rate used
reduce the sampling resolution
reduce the frame rate
image files
crop the image
decrease the colour/bit depth
reduce the image resolution

Figure 1.9 General methods of compressing files

ACTIVITY 1I

1 a) What is meant by *lossless* and *lossy* file compression?
   b) Give an example of a lossless file format and an example of a lossy file format.

2 a) Describe how music picked up by a microphone is turned into a digitised music file in a computer.
   b) Explain why it is often necessary to compress stored music files. Describe how the music quality is essentially retained.

3 a) What is meant by *run length encoding*?
   b) Describe how RLE compresses a file. Give an example in your description.

4 a) Describe the differences between bit-map images and vector graphics.
   b) A software designer needs to incorporate images into her software to add realism. Explain what she needs to consider when deciding between using bit-map images and vector graphics in her software.

End of chapter questions

1 a) The following bytes represent binary integers using the two's complement form. State the equivalent denary values.

i) 0 1 0 0   1 1 1 1

[1]

ii) 1 0 0 1   1 0 1 0

[1]

iii) Write the integer −53 in two's complement form.

[1]

iv) Write the maximum possible range of numbers using the two's complement form of an 8-bit binary number.
Give your answers in denary.

[2]

b) i) Write the denary integer 798 in binary-coded decimal (BCD) format.

[1]

ii) Write the denary number that is represented by the following BCD number.

1	0	0	1	0	1	1	1	0	1	1	1	0	1	1	0

[2]

c) Give **one** use of binary-coded decimal system.

[1]

2 A software developer is using a microphone and a sound editing app to collect and edit sounds for his new game.
When collecting sounds, the software developer can decide on the sampling resolution he wishes to use.

a) i) State what is meant by *sampling resolution*.

[1]

ii) Describe how sampling resolution will affect how accurate the stored digitised sound will be.

[2]

b) The software developer will include images in his new game.

i) Explain the term *image resolution*.

[1]

ii) The software developer is using 16-colour bit-map images.
State the number of bits required to encode data for one pixel of his image.

[1]

iii) One of the images is 16 384 pixels wide and 512 pixels high.
The developer decides to save it as a 256-colour bit-map image.
Calculate the size of the image file in gibibytes.

[3]

iv) The bit-map image will contain a *header*.
State **two** items you would expect to see in the header.

[2]

v) Give **three** features you would expect to see in the sound editing app.

[3]

3 The editor of a movie is finalising the music score. They will send the final version of the score to the movie producer by email attachment.

a) Describe how *sampling* is used to record the music sound clips.

[3]

b) The music sound clips need to undergo some form of data compression before the music editor can send them via email.
Identify the type of compression, lossy or lossless, they should use.
Give a justification for your answer.

[3]

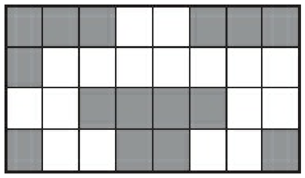
c) One method of data compression is known as *run length encoding (RLE).*

i) Explain what is meant by RLE.

[3]

ii) Show how RLE would be used to produce a compressed file for the image below.
Write down the data you would expect to see in the RLE compressed format (you may assume that the grey squares have a code value of 85 and the white squares have a code value of 255).

[4]



4 a) Write the denary numbers 60, 27 and −27 in 8-bit binary two's complement form.

[3]

b) Show the result of the addition 60 + 27 using 8-bit binary two's complement form. Show all of your working.

[2]

c) Show the result of the subtraction 60 − 27 using 8-bit binary two's complement form.

[2]

d) Give the result of the following addition.

0 1 0 1 1 0 0 1

+

0 1 1 0 0 0 0 1

Explain why the expected result is not obtained.

[2]

5 a) Carry out 0.52 + 0.83 using binary-coded decimal (BCD). Show all of your working.

[4]

b) i) Define the term *hexadecimal*.

[1]

ii) Give **two** uses of the hexadecimal system.

[2]

iii) Convert the following binary number into hexadecimal.

0 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0

[2]

6 a) Convert the denary number 95 into binary coded decimal (BCD).

[1]

b) Using two's complement, carry out the binary subtraction:
0 0 1 0 0 0 1 1 – 0 1 0 0 0 1 0 0
and convert your answer into denary.

[3]

c) Convert the denary number 506 into hexadecimal.

[1]