

4. Organisation and structure of data

Organisation and structure of data

Representation of numbers

The nature of data

Data is made up of raw facts and figures and can be represented in many different forms including text, numbers, pictures, sounds and video clips. Information can be derived from data when it is processed.

Why data needs to be converted into binary format

You will need to be familiar with three different counting systems. These are denary, binary and hexadecimal.

Denary

The first counting system that you need to be familiar with is the **denary** counting system, also known as the **Base 10** or decimal counting system. In the denary counting system, the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are used to represent numbers. The number *138* for example, actually means 1 'hundred', 3 'tens' and 8 'units'. This gives the total one hundred and thirty-eight:

10^2	10^1	10^0
100	10	1
1	3	8

Binary

The second counting system that you need to be familiar with is the **binary** counting systems, also known as the **Base 2** counting system. In order for data to be processed by a computer system, it must be converted into binary format. This is because computer systems can only store and process **Binary digITs**, also known as **BITs**. A **BIT** is either a 1 or 0. You may think of this as a light switch, where the switch is either **ON** or **OFF**:

- If the switch is ON it is stored as the digit 1.
- If the switch is OFF it is stored as the digit 0.

A binary number is a string of BITS, for example 10001010.

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

The binary number 10001010 is therefore a binary representation of the denary number 138 ($128 + 8 + 2$).

Hexadecimal numbers representing binary numbers

The third counting system that you need to be familiar with is the **hexadecimal** counting system, also known as the **Base 16** counting system. In the hexadecimal counting system, the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are used to represent 1–9 and then the characters A, B, C, D, E and F are used to represent 10–15. The hexadecimal number 8A for example:

16^2	16^1	16^0
256	16	1
0	8	A

The hexadecimal number 8A therefore represents 8 ‘sixteens’ and 10 ‘units’. This gives the total one hundred and thirty-eight. Remember that A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.

INTERESTING FACT

Up until the late 20th century, traditional Chinese weights and measurements used in the marketplace used the hexadecimal counting system. Other cultures used different base counting systems, e.g. the ancient Babylonians used a **Base 60** counting system.

Hexadecimal is widely used as binary numbers can be quickly converted into hexadecimal numbers that are more convenient for humans to use. For example, a telephone conversation where you might read out the binary number

10001010 could cause confusion. It would be easier to simply say 8A and mistakes are less likely to be made.

Denary to binary and hexadecimal, binary to denary and hexadecimal, hexadecimal to binary and denary

In this section, we will discuss how to convert between different number systems.

Denary to binary

One way of converting a denary number to a binary number is by drawing a Base 2 table from the right to the left.

128	64	32	16	8	4	2	1

In this example, we will convert the denary number 198 into a binary number. Take 198 and see if it is more than the first number on the left. In this case, 128 is the number on the left and so we write a 1 under the heading 128.

128	64	32	16	8	4	2	1
1							

We now deduct 128 from our original denary number, which leaves 70. The next number in our Base 2 table is 64. If the number remaining, 70, is more than the next number on the left, 64, write the number 1 under the heading 64.

128	64	32	16	8	4	2	1
1	1						

We now repeat the process again and deduct 64 from 70, which leaves 6. The next number in our Base 2 table is 32. If the number remaining, 6, is more than the next number on the left, 32, write the number 1 under the heading 32. However, in this case the number remaining is less than the next number on the left, so we write a 0 under the heading 32.

128	64	32	16	8	4	2	1
1	1	0					


This process is repeated until you reach the final heading and the binary number for the denary number 198 is found:

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

The binary number for the denary number 198 is therefore 11000110 (128 + 64 + 4 + 2).

Denary to hexadecimal

You may wish to convert a denary number into a hexadecimal number. To do this, take the number 198 from our previous example and draw a **Base 16** table, from right to left, as before.



256	16	1

Take 198 and see if it is more than the first number on the left. In this case, 256 is the number on the left and so we write a 0 under the heading 256.

256	16	1
0		

The next number in our **Base 16** table is 16. If the number remaining, 198, is more than the next number on the left, 16, work out how many 16s are needed without going over the number remaining. In this case it is C ($C = 12$, $12 \times 16 = 192$).

Remember that A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

256	16	1
0	C	

We now deduct 192 from our remaining denary number, 198, which leaves 6. The next number in our **Base 16** table is 1. If the number remaining, 6, is more than the next number on the left, 1, work out how many 1s are needed without going over the number remaining. In this case it is 6.

256	16	1
0	C	6

The hexadecimal number for the denary number 198 is therefore C6.

Hint

You may find it easier to convert a denary number into a binary number first and then into a hexadecimal number. See the example *binary to hexadecimal* below.

Binary to denary

To convert a binary number into a denary number, draw a Base 2 table from right to left and populate the table with the binary number you are converting. In this case we will use 00100011.

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

Simply convert the binary number into a denary number by adding the headings with a 1 under them: $32 + 2 + 1 = 35$. The denary number for the binary number 00100011 is therefore 35.

Binary to hexadecimal

To convert a binary number into a hexadecimal number, there is a shortcut that you can use by drawing a Base 2 table from right to left and then populating the table with the binary number you are converting. In this case we will use 00101011.

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

Now split the Base 2 table into two smaller 4-bit Base 2 tables.

128	64	32	16
0	0	1	0

8	4	2	1
1	0	1	1

Now change the headings of the left 4-bit table.

8	4	2	1
0	0	1	0

8	4	2	1
1	0	1	1

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

2

B

Now take the hexadecimal number of each 4-bit table and this is the converted hexadecimal number.

256	16	1
0	2	B

The hexadecimal number for the binary number 00101011 is therefore 2B.

Hexadecimal to denary

You may wish to convert a hexadecimal number into a denary number. To do this you may take the number C6 and draw a Base 16 table, from right to left as before.

256	16	1
0	C	6

Now multiply each heading to obtain the denary converted number.

$$\begin{array}{r} C(12) \times 16 = 192 \\ 6 \times 1 = \quad 6 \quad + \\ \hline 198 \end{array}$$

The denary number for the hexadecimal number C6 is therefore 198.

Hexadecimal to binary

To convert a hexadecimal number into a binary number, there is a shortcut that you can use similar to the one above by drawing two 4-bit Base 2 tables from right to left.

8	4	2	1

8	4	2	1

In this example, we will convert the hexadecimal number 2B into a binary number. First start by representing the first number, 2, in the left table.

8	4	2	1
0	0	1	0

8	4	2	1


Then complete the second table by representing B in the right table, remembering that B = 11.

8	4	2	1
0	0	1	0

8	4	2	1
1	0	1	1

Now re-label the headings in the left table as shown below and join the two 4-bit tables together to make one 8-bit table.

128	64	32	16
0	0	1	0



8	4	2	1
1	0	1	1

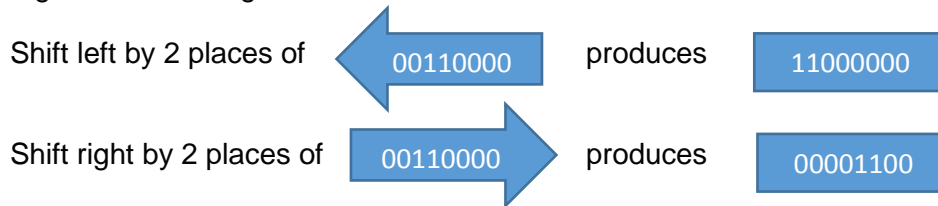
128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	1

And so, the hexadecimal number 2B can be represented as 00101011 in binary number form.

Arithmetic shift functions.

Shifts are manipulations of bit patterns. A shift involves moving the bits in a specified direction, either left or right, by a specified number of places

e.g. for an 8 bit register

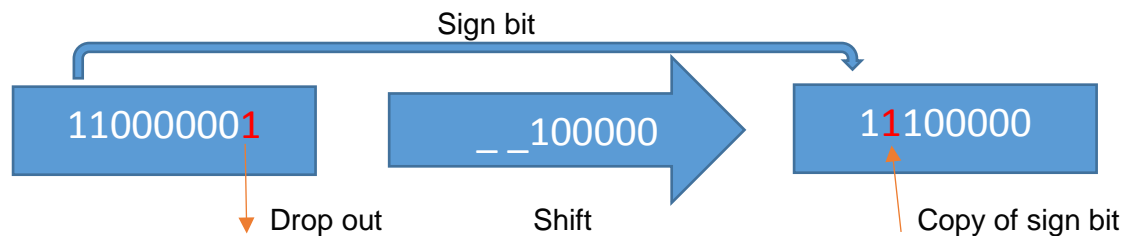


Arithmetic shifts can be used for division and multiplication.

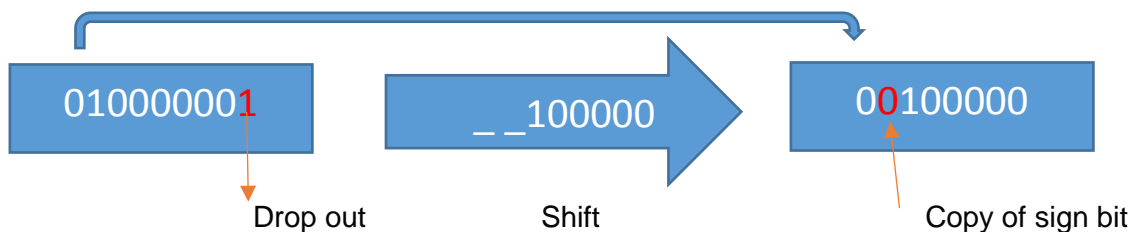
Arithmetic shift right

This operation preserves the sign of a number and will divide a binary number by 2 at each shift. It will work for positive and negative numbers.

At each shift the right hand bit is lost and a copy of the sign bit is inserted to the left.



Negative integer in Two's complement: - 63 right shift 1 place = - 32



Positive integer in Two's complement: 65 right shift 1 place = 32

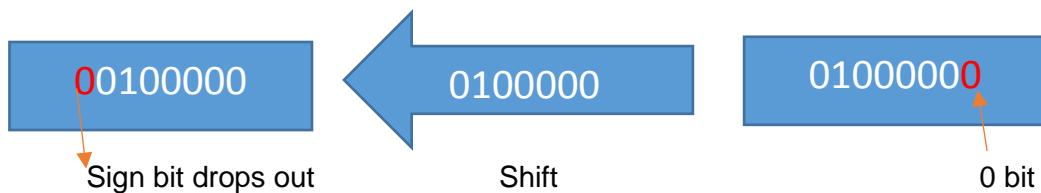
Note when the right bit has value i.e. when the shift is applied to an odd number the result is rounded down to the next even integer.

Arithmetic shift left

Similar, but at each shift the sign bit is lost and a 0 bit is moved in to the right. The effect of each shift is to multiply the integer $\times 2$. The process can be repeated until the sign bit is changed, at which point overflow occurs.



Negative integer in Two's complement: -64 left shift 1 place = -128



Positive integer in Two's complement: 32 left shift 1 place = 64



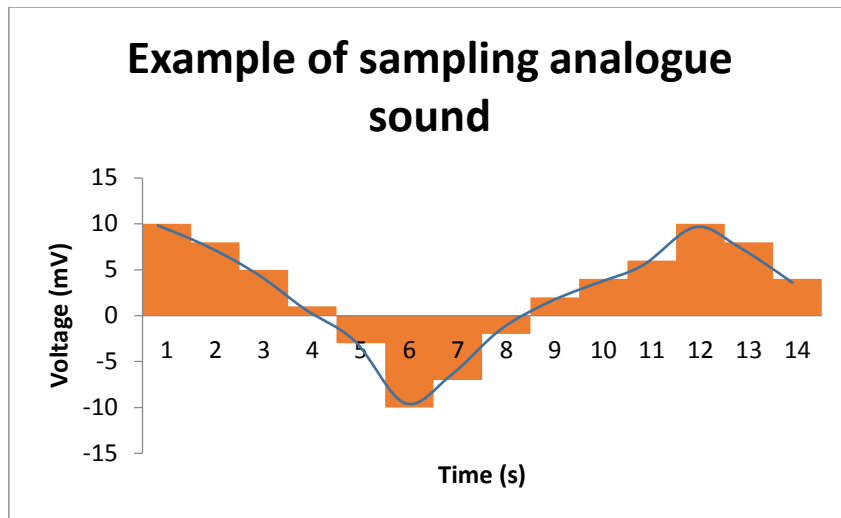
Positive integer in Two's complement: 64 left shift 1 place = overflow

The shift changes the sign bit and overflow occurs.

How sound can be sampled and stored digitally

As we have already established, a computer system is only able to store and process binary digits, as it is a digital device. Since this is the case, how can sound be stored as it is an analogue signal not digital? If an analogue signal, such as sound, is sent to a computer system, it has to be converted into a digital signal before it can be processed.

Sound is converted into a digital signal by a process called *sampling*. Sampling is where hardware, such as a microphone, measures the level of sound many times per second and records this as binary digits.



The number of times that the sound level is sampled per second is called the *sampling frequency*. The higher the sampling frequency, the better the quality of the sound recorded. A typical sampling frequency is 44,000 times per second, also known as 44 kHz. This is the sampling frequency used on most audio CDs.

Sound sampled at 44 kHz in stereo will produce a large amount of data and as such, this data may need to be compressed. When sound files are compressed, data is removed to reduce the size. This reduction in size means that

DIGITAL AUDIO QUALITY

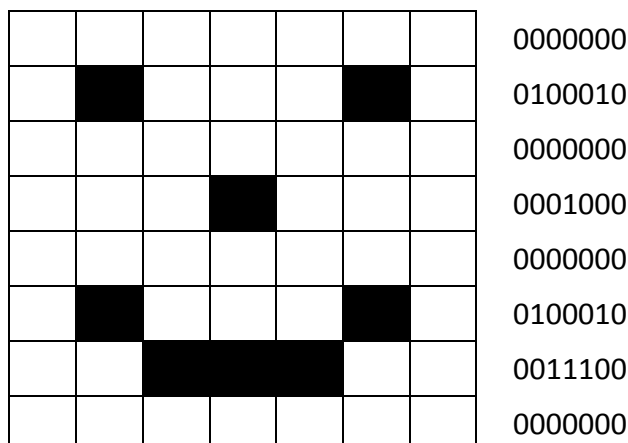
Sample rate – the number of audio samples captured every second

Bit depth – the number of bits available for each clip

Bit rate – the number of bits used per second of audio

How an image is represented by pixels in binary format

Images on a computer system are made up of thousands of small coloured dots, known as pixels (short for picture elements). Bitmap images are stored as an array of pixels. A black and white bitmap image will store a 1 for a black pixel and 0 for a white pixel.



0000000
 0100010
 0000000
 0001000
 0000000
 0100010
 0011100
 0000000

This bitmap image can be represented using a 56 bits (or 7 bytes).

A colour bitmap image is stored by replacing the 1s and 0s above with a longer number that represents how much red, green and blue (RGB) is required in the colour of each pixel; this is known as *colour depth*. In a 256-colour palette, the image would require 1 byte of storage per pixel – so we would need 448 bits (or 56 bytes) to store the image above in colour. There are other colour depths available, which can store more information about the colours in each pixel of an image. The more information stored about the colour of each pixel, the larger the file size becomes.

You may also have heard of vector images. These images do not store the data by pixels, but are a set of instructions for drawing a geometric shape. The advantages of a vector image are that they can be scaled without loss of quality (pixilation etc.) and use less storage space.

Images require a large amount of storage space and as such, may need to be compressed.

Why metadata needs to be included in an image file (including height, width, colour depth)

The term metadata refers to 'data about data'. Key properties that are needed to display an image correctly are stored as metadata. Data such as an image's height, width and colour depth are typical examples of data stored in the metadata about an image. Without metadata, a computer system may render an image incorrectly on screen, such as displaying all pixels in one row.

Other data may also be stored in the metadata of an image file, such as the date the image was made, the geographical location of a photograph.

Binary numbers representing characters

A **character** can be a letter, digit, space, punctuation mark or various other symbols. When characters are stored on a computer system, they are stored as a binary number.

It is important that computer systems recognise that characters can be represented differently by other computer systems; otherwise data could not be exchanged between computers.

The terms 'character set', Unicode and American Standard Code for Information Interchange (ASCII)

In order to allow for data exchange between computer systems, **character sets** were devised. A character set is a table that maps a character with a unique binary number.

One such character set is the 7-bit American Standard Code for Information Interchange (ASCII). Part of the ASCII character set, that includes printable characters only, can be seen in the table overleaf.

INTERESTING FACT

Before the widespread adoption of graphical user interfaces, programmers used the ASCII character set to design simple interfaces. Try searching for some on the Internet.

Denary	Binary	Hex	Character
32	100000	20	space
33	100001	21	!
34	100010	22	"
35	100011	23	#
36	100100	24	\$
37	100101	25	%
38	100110	26	&
39	100111	27	'
40	101000	28	(
41	101001	29)
42	101010	2A	*
43	101011	2B	+
44	101100	2C	,
45	101101	2D	-
46	101110	2E	.
47	101111	2F	/
48	110000	30	0
49	110001	31	1
50	110010	32	2
51	110011	33	3
52	110100	34	4
53	110101	35	5
54	110110	36	6
55	110111	37	7
56	111000	38	8
57	111001	39	9
58	111010	3A	:
59	111011	3B	;
60	111100	3C	<
61	111101	3D	=
62	111110	3E	>
63	111111	3F	?

Denary	Binary	Hex	Character
64	1000000	40	@
65	1000001	41	A
66	1000010	42	B
67	1000011	43	C
68	1000100	44	D
69	1000101	45	E
70	1000110	46	F
71	1000111	47	G
72	1001000	48	H
73	1001001	49	I
74	1001010	4A	J
75	1001011	4B	K
76	1001100	4C	L
77	1001101	4D	M
78	1001110	4E	N
79	1001111	4F	O
80	1010000	50	P
81	1010001	51	Q
82	1010010	52	R
83	1010011	53	S
84	1010100	54	T
85	1010101	55	U
86	1010110	56	V
87	1010111	57	W
88	1011000	58	X
89	1011001	59	Y
90	1011010	5A	Z
91	1011011	5B	[
92	1011100	5C	\
93	1011101	5D]
94	1011110	5E	^
95	1011111	5F	_

Denary	Binary	Hex	Character
96	1100000	60	`
97	1100001	61	a
98	1100010	62	b
99	1100011	63	c
100	1100100	64	d
101	1100101	65	e
102	1100110	66	f
103	1100111	67	g
104	1101000	68	h
105	1101001	69	i
106	1101010	6A	j
107	1101011	6B	k
108	1101100	6C	l
109	1101101	6D	m
110	1101110	6E	n
111	1101111	6F	o
112	1110000	70	p
113	1110001	71	q
114	1110010	72	r
115	1110011	73	s
116	1110100	74	t
117	1110101	75	u
118	1110110	76	v
119	1110111	77	w
120	1111000	78	x
121	1111001	79	y
122	1111010	7A	z
123	1111011	7B	{
124	1111100	7C	
125	1111101	7D	}
126	1111110	7E	~

Using the ASCII character set, the character A would be stored as the binary number 1000001.

The problem with using this ASCII character set is that it is only able to represent 128 different characters and computer systems need to be able to store more characters than this. For example, you may have noticed that the £ character is missing from the table above. As a result, other character sets were developed and used to allow computer systems to store more characters.

Unicode is a standard character set that has combined and replaced many others. It was originally an extension to the ASCII character set and it contains many of the characters used around the world.

Data types such as: integer, real, Boolean, character, string

Many different data types can be stored on a computer system. The data types that are commonly used are as follows:

Data type	Description	Examples
Integer	Whole numbers, positive or negative	42, -11, 0
Real	Numbers, including fractions or decimal points	12.9, -17.50, 28.0
Boolean	True or false	1 or 0
Character	Letter, digit, space, punctuation mark or various other symbols	'A', 'b', '7', '?'
String	A sequence of characters	'Computer science' 'The cat sat on the mat'

Data structures

A data structure is a specific way of organising data within memory so that it can be processed efficiently. There will be a relationship between the data items that will vary according to the type of data structure being used.

Static data structure

A static data structure is designed to store a known number of data items. The values of the data can be changed but the memory size is fixed. An array is an example of a static data structure; we can change the values of the elements in the array but we cannot alter the memory size allocated to the array. Memory is allocated at compile time.

As static data structures store a fixed number of data items they are easier to program, as there is no need to check on the size of the data structure or the number of items stored.

Dynamic data structure

Dynamic data structures are designed to allow the data structure to grow or shrink at runtime. It is possible to add new elements or remove existing elements without having to consider memory space. Memory is allocated at runtime.

Dynamic data structures make the most efficient use of memory but are more difficult to program, as you have to check the size of the data structure and the location of the data items each time you use the data.

List

A list is a data structure that has the data items stored in the order they were originally added to memory. If the list is made up of a set number of data items, then it can be a static data structure. If the list can vary in the number of data items, then it will be a dynamic data structure.

Array

An array is a data structure that can hold a fixed number of data items, which must be of the same data type i.e. real, integer, string etc. The data items in an array are known as elements. An array is an example of a static list.

The elements in an array are identified by a number that indicates their position in the array. This number is known as the index. The first element in an array usually has an index of 0.

Elements	37	11	42	6	26	56	4	76
Index	0	1	2	3	4	5	6	7

- There are 8 elements in this array.
- The index always starts at position 0.
- Each element can be accessed using its index. The element at index 5 is 56.
- This type of array is known as a one-dimensional array.

Using one-dimensional arrays

A one-dimensional array can be used to store a list of data in memory that can be used by a program at runtime. There are basic operations that can be carried out on data in a one-dimensional array.

Traversing

Traversing an array simply means using a loop to use each element of the array in a section of a program. If you wanted to print out the contents of an array called myArray that has 10 elements you would use a 'for . . . next' loop. Like this:

```
1  
2 myArray[10]           'dimensions the array  
3  
4 for i = 0 to 9       'sets the loop  
5     print myArray[i] 'prints each element in sequence  
6 next in              'end of loop  
7  
8
```

Insertion

You can add an element to an array at a given index.

```
myArray[3] = 27
```

This would store the value 27 at the index 3 of the array.

Deletion

You can delete an element from an array.

```
myArray[6] = ""
```

This would leave the memory at index 6 blank.

Search

Arrays can be search using the index or the value stored at the index.

Two-dimensional arrays

Often the data we want to process comes in the form of a table. The data in a two-dimensional array must all be of the same data type.

For example, your teacher may have a spreadsheet of your class' test results.

Pupil	Test 1	Test 2	Test 3	Test 4	Test 5
Sam	23	5	34	4	23
Maisie	25	7	12	6	12
Elsie	34	8	45	8	32
Frank	9	4	43	9	26

Elements in a two-dimensional array are indexed by two numbers – one for the row it is in and one for the column.

		0	1	2	3	4
	Pupil	Test 1	Test 2	Test 3	Test 4	Test 5
0	Sam	23	5	34	4	23
1	Maisie	25	7	12	6	12
2	Elsie	34	8	45	8	32
3	Frank	9	4	43	9	26

If the two-dimensional array is called testMark then the command to declare this array would be:

```
testMark [4,5]
```

In this declaration, the 4 refers to the number of pupils (rows) and the 5 to the number of tests (columns). The index for Sam's marks for Test 2 would be [0,1].

To print out the pupils test marks you would need to use one loop inside another loop:

```
1  
2 testMark[4,5]  
3  
4 for i = 0 to 3  
5     for j = 0 to 4  
6         print testMark[i, j]  
7     next j  
8 next i  
9  
10
```

Records

Arrays can only hold data if it is all of the same data type. If you need to hold related data of different data types you will need to use a data structure called a record. A record will be made up of information about one person or thing. Each piece of information in the record is called a field.

For example, an after school club wants to store data about its members' emergency contact information.

Record Structure

Field Name	Field Type	Example data
Membership Number	Integer	1074
First name	String	Sara
Surname	String	Davies
Date of Birth	Date	12/07/2004
Contact Name	String	Mrs Davies
Contact phone number	String	07564 191919

Key field

Each record in a file should have a key field. That is an item of data that is unique and can be used to identify the individual record. In this example the membership number would be the key field.

Files

For a computer to function it must have data flowing through the central processing unit (CPU) under the control of a program. More often than not this data will have come from a stored file.

A program will load the file from secondary storage, such as a hard disc, into the computer's memory. The data will be manipulated by the CPU and then output. The output could be another data file, screen images or a document.

Data stored in a file will have a structure and organisation known as the file format. A data file will be made up of records. It would be most efficient for the fields in a record to be stored next to each other so that the data can be read into the record data structure in memory for processing by the CPU.

In summary – files are made of records of the same structure and records are made up of fields containing information about one person or item.

Validation

Validation is a process to check that input data is reasonable or sensible. Frequently used validation algorithms include;

Presence checks	Used to prevent further progress if a required field is left blank.
Format checks	Used to ensure data matches a specific pattern, such as dd/mm/yyyy for a date. Input masks are often used to create format checks on database forms.
Length checks	Used to ensure an input data string is a sensible length e.g. number of characters in 'firstName' to be between 3 and 16
Type checks	Used to ensure input data is a particular data type e.g. quantity ordered to be integer or cost to be real.
Range checks	Used to ensure input data lies within a specified range e.g. over time hours to be > 0 and < 15.

Look up checks can also be used to ensure that input data matches an item in a list of valid entries e.g. input look up "none"; "vegetarian"; "vegan" will limit the acceptable input to one of three entries.

Verification

Verification is a process for checking data is correct. It can be carried out as a user enters data, such as via a keyboard and also when data is copied from one part of a system to another. Copying should not change the data.

Examples of verification of user input include double entry and screen based verification.

Double entry involves comparing two versions of data input. e.g. "re-enter your email address". A verification algorithm will compare the two versions and inform the user if they are not identical.

Screen based verification requires the user to check a display of input data and confirm that it is correct.

More sophisticated verification algorithms apply calculations to input data. e.g. to produce the check digits of bar codes. Repeating the calculations and checking that the result is the same can verify the data.

Similar verification algorithms, including parity checks and checksums, can be used when sending data between computers to check that the data has not been corrupted during transmission.

Algorithm design - examples

- A. Design an algorithm that will sensibly validate the input data for a diving competition, where entrants must be between 16 and 19 years of age. The competition is to involve 5 dives, with each dive being awarded a score of between 0 and 20.

The algorithm should verify the entrants date of birth, check that they are eligible for competition and ensure that sensible scores are recorded.

- B. The table includes input data for a payroll system. Some of the data can be sensibly validated.

Surname	String
National Insurance (NI) number	Standard format LL123456L
Job title	Apprentice, semi-skilled, skilled, supervisor
Week no.	Integer
Full time	Y or N, Full time = 38 hours per week
Hours worked	Integer, hours worked in current week, maximum 10 hours overtime in one week. Overtime rate = 1.5 x pay rate
Pay rate	Real, hourly pay rate, max £15.00 / hour

Write validation checks that will help to ensure that input is sensible.