Computer Science Y7

Unit 1Computational Thinking

Lesson 1

Computational Thinking

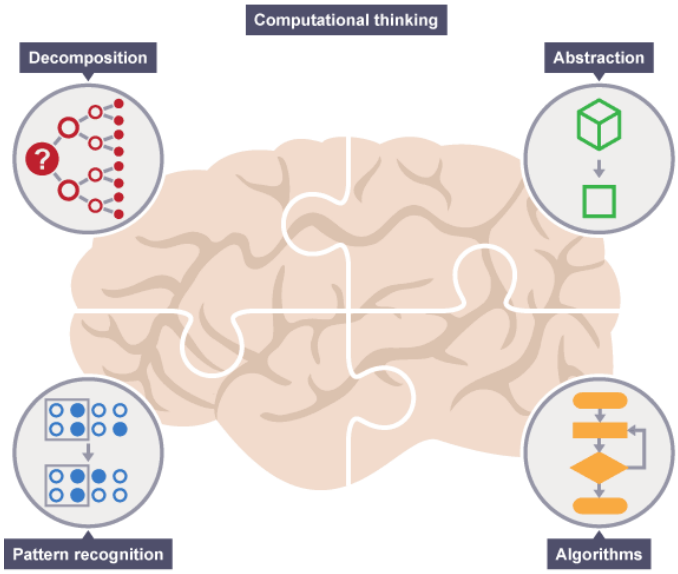
What is computational thinking?

Computers can be used to help us solve problems. However, before a problem can be tackled, the problem itself and the ways in which it could be solved need to be understood. Computational thinking allows us to do this.

The four cornerstones of computational thinking

- **decomposition** - breaking down a complex problem or system into smaller, more manageable parts
- **pattern recognition** – looking for similarities among and within problems
- **abstraction** – focusing on the important information only, ignoring irrelevant detail
- **algorithms** - developing a step-by-step solution to the problem, or the rules to follow to solve the problem

Each cornerstone is as important as the others. They are like legs on a table - if one leg is missing, the table will probably collapse. Correctly applying all four techniques will help when programming a computer.



Computational thinking in practice

A complex problem is one that, at first glance, we don't know how to solve easily. Computational thinking involves taking that complex problem and breaking it down into a series of small, more manageable problems (**decomposition**). Each of these smaller problems can then be looked at individually, considering how similar problems have been solved previously (**pattern recognition**) and focusing only on the important details, while ignoring irrelevant information (**abstraction**). Next, simple steps or rules to solve each of the smaller problems can be designed (**algorithms**).
Finally, these simple steps or rules are used to **program** a computer to help solve the complex problem in the best way.

Thinking computationally

Computational thinking enables you to work out exactly what to tell the computer to do.
For example, if you agree to meet your friends somewhere you have never been before, you would probably plan your route before you step out of your house. You might consider the routes available and which route is 'best' - this might be the route that is the shortest, the quickest, or the one which goes past your favourite shop on the way. You'd then follow the step-by-step directions to get there. In this case, **the planning part is like computational thinking,** and **following the directions is like programming.**

Being able to turn a complex problem into one we can easily understand is a skill that is extremely useful. In fact, it's a skill you already have and probably use every day.

For example, it might be that you need to decide what to do with your group of friends. If all of you like different things, you would need to decide:

- what you could do
- where you could go
- who wants to do what
- what you have previously done that has been a success in the past
- how much money you have and the cost of any of the options
- what the weather might be doing
- how much time you have

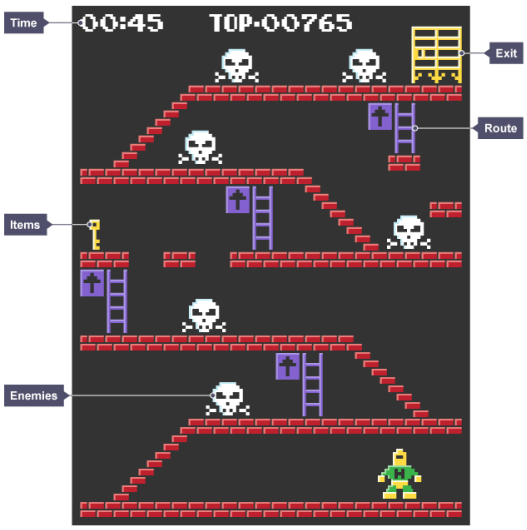
From this information, you and your friends could decide more easily where to go and what to do – in order to keep most of your friends happy. You could also use a computer to help you to collect and analyse the data to devise the best solution to the problem, both now and if it arose again in the future, if you wished.

Another example might occur when playing a videogame. Depending on the game, in order to complete a level you would need to know:

- what items you need to collect, how you can collect them, and how long you have in which to collect them
- where the exit is and the best route to reach it in the quickest time possible
- what kinds of enemies there are and their weak points

From these details you can work out a strategy for completing the level in the most efficient way.

If you were to create your own computer game, these are exactly the types of questions you would need to think about and answer before you were able to program your game.



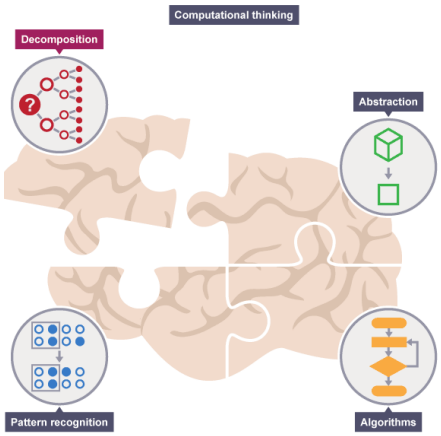
Both of the above are examples of where computational thinking has been used to solve a complex problem:

- each complex problem was broken down into several small decisions and steps (eg where to go, how to complete the level – decomposition)
- only the relevant details were focused on (eg weather, location of exit – abstraction)
- knowledge of previous similar problems was used (pattern recognition...to work out a step by step plan of action (algorithms)

Lesson 2

What is decomposition?

Decomposition is one of the four cornerstones of Computer Science. It **involves breaking down a complex problem or system into smaller parts that are more manageable and easier to understand**. The smaller parts can then be examined and solved, or designed individually, as they are simpler to work with.



Why is decomposition important?

If a problem is not decomposed, it is much harder to solve. Dealing with many different stages all at once is much more difficult than breaking a problem down into a number of smaller problems and solving each one, one at a time. Breaking the problem down into smaller parts means that each smaller problem can be examined in more detail.

Similarly, trying to understand how a complex system works is easier using decomposition. For example, understanding how a bicycle works is more straightforward if the whole bike is separated into smaller parts and each part is examined to see how it works in more detail.

Example 1: Brushing our teeth

To decompose the problem of how to brush our teeth, we would need to consider:

- which toothbrush to use
- how long to brush for?
- how hard to press on our teeth?
- what toothpaste to use

Example 2: Solving a crime

It is only normally when we are asked to do a new or more complex task that we start to think about it in detail – to decompose the task.

Imagine that a crime has been committed. Solving a crime can be a very complex problem as there are many things to consider.

For example, a police officer would need to know the answer to a series of smaller problems:

- what crime was committed
- when the crime was committed
- where the crime was committed
- what evidence there is
- if there were any witnesses
- if there have recently been any similar crimes

The complex problem of the committed crime has now been broken down into simpler problems that can be examined individually, in detail.

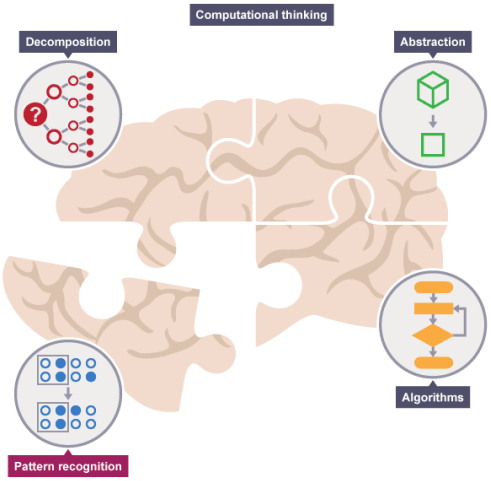
Decomposing creating an app

Imagine that you want to create your first app. This is a complex problem - there are lots of things to consider.

How would you decompose the task of creating an app?

Lesson 3

What is pattern recognition?

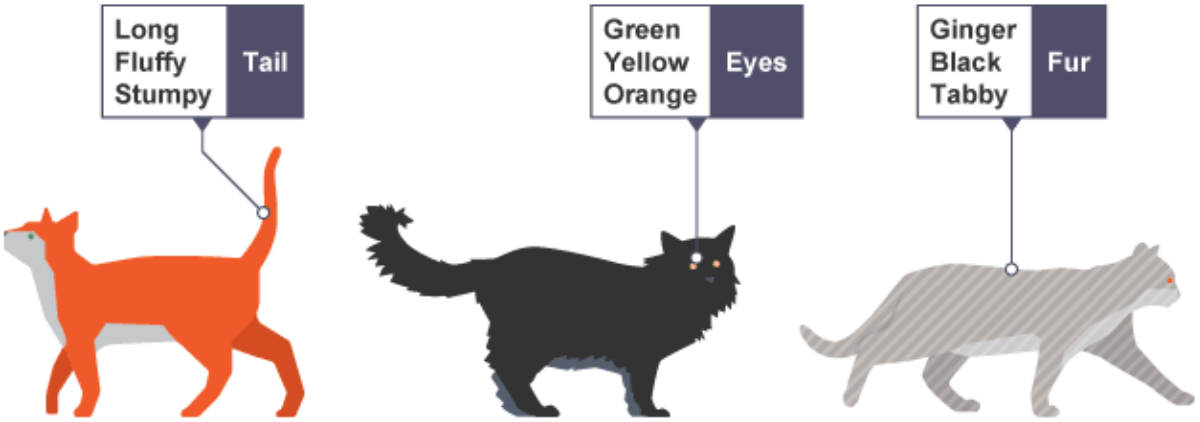


What are patterns?

Imagine that we want to draw a series of cats.
All cats share common characteristics. Among other things **they all have eyes, tails and fur**. They also like to eat fish and make meowing sounds.

Because we know that all cats have eyes, tails and fur, we can make a good attempt at drawing a cat, simply by including these common characteristics.
In computational thinking, these characteristics are known as patterns. **Once we know how to describe one cat we can describe others, simply by following this pattern.** The only things that are different are the specifics:

- one cat may have green **eyes**, a long **tail** and black **fur**
- another cat may have yellow **eyes**, a short **tail** and striped **fur**



Why do we need to look for patterns?

Finding patterns is extremely important. Patterns make our task simpler. Problems are easier to solve when they share patterns, because we can use the same problem-solving solution wherever the pattern exists.

The more patterns we can find, the easier and quicker our overall task of problem solving will be.

If we want to draw a number of cats, finding a pattern to describe cats in general, eg they all have eyes, tails and fur, makes this task quicker and easier.

We know that all cats follow this pattern, so we don't have to stop each time we start to draw a new cat to work this out. From the patterns we know cats follow, we can quickly draw several cats.

What happens when we don't look for patterns?

Suppose we hadn't looked for patterns in cats. Each time we wanted to draw a cat, we would have to stop and work out what a cat looked like. This would slow us down.

We could still draw our cats - and they would look like cats - but each cat would take far longer to draw. This would be very inefficient, and a poor way to go about solving the cat-drawing task.

In addition, if we don't look for patterns we might not realise that all cats have eyes, tails and fur. When drawn, our cats might not even look like cats. In this case, because we didn't recognise the pattern, we would be solving the problem incorrectly.

Recognising patterns

To find patterns in problems we look for things that are the same (or very similar) in each problem. It may turn out that no common characteristics exist among problems, but we should still look.
Patterns exist **among different problems** and **within individual problems**. We need to look for both.

Patterns among different problems

To find patterns among problems we look for things that are the same (or very similar) for each problem.
For example, decomposing the task of baking a cake would highlight the need for us to know the solutions to a series of smaller problems:

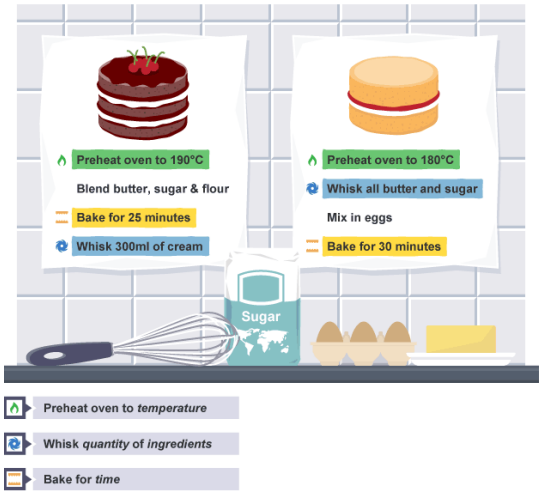
- what kind of cake we want to bake?
- what ingredients we need and how much of each
- how many people we want to bake the cake for
- how long we need to bake the cake for
- when we need to add each ingredient
- what equipment we need

Once we know how to bake one particular type of cake, we can see that baking another type of cake is not that different - because patterns exist.

For example:

- each cake will need a precise quantity of specific ingredients
- ingredients will get added at a specific time
- each cake will bake for a specific period of time

Once we have the patterns identified, we can work on common solutions between the problems.



Patterns within problems

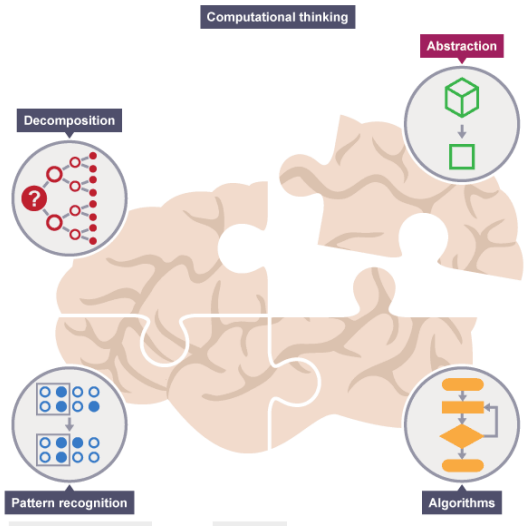
Patterns may also exist within the smaller problems we have decomposed to.

If we look at baking a cake, we can find patterns within the smaller problems, too.
For example, for ‘each cake will need a precise quantity of specific ingredients’, each ingredient needs:
identifying (naming), a specific measurement
Once we know how to identify each ingredient and its amount, we can apply that pattern to all ingredients. Again, all that changes is the specifics.

Lesson 3

What is abstraction?

Abstraction is one of the four cornerstones of Computer Science. It involves filtering out – essentially, ignoring - the characteristics that we don't need in order to concentrate on those that we do.



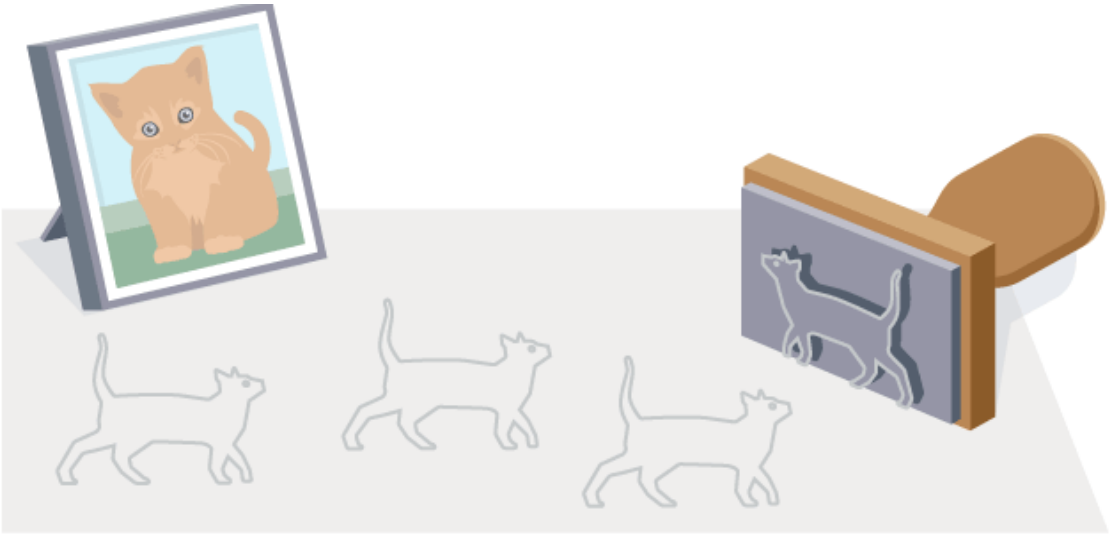
In computational thinking, when we decompose problems, we then look for patterns among and within the smaller problems that make up the complex problem.

Abstraction is the process of filtering out – ignoring - the characteristics of patterns that we don't need in order to concentrate on those that we do. It is also the filtering out of specific details. From this we create a representation (idea) of what we are trying to solve.

What are specific details or characteristics?

In pattern recognition we looked at the problem of having to draw a series of cats.
We noted that all cats have general characteristics, which are common to all cats, eg eyes, a tail, fur, a liking for fish and the ability to make meowing sounds. In addition, each cat has **specific characteristics**, such as **black** fur, a **long** tail, **green** eyes, a love of **salmon**, and a **loud** meow. **These details are known as specifics**.
In order to draw a basic cat, we **do** need to know that it has a tail, fur and eyes. These characteristics are relevant. We **don't** need to know what sound a cat makes or that it likes fish. These characteristics are irrelevant and can be filtered out. We **do** need to know that a cat has a tail, fur and eyes, but we **don't** need to know what size and colour these are. These specifics can be filtered out.

From the general characteristics we have (tail, fur, eyes) we can build a basic idea of a cat, ie what a cat basically looks like. Once we know what a cat looks like we can describe how to draw a basic cat.



Why is abstraction important?

Abstraction allows us to create a general idea of what the problem is and how to solve it. The process instructs us to remove all specific detail, and any patterns that will not help us solve our problem. This helps us form our idea of the problem. This idea is known as a ‘model’.

If we don’t abstract, we may end up with the wrong solution to the problem we are trying to solve. With our cat example, if we didn’t abstract we might think that all cats have long tails and short fur. Having abstracted, we know that although cats have tails and fur, not all tails are long and not all fur is short. In this case, abstraction has helped us to form a clearer model of a cat.

How to abstract

Abstraction is the gathering of the general characteristics we need and the filtering out of the details and characteristics that we do not need.

When baking a cake, there are some general characteristics between cakes. For example:

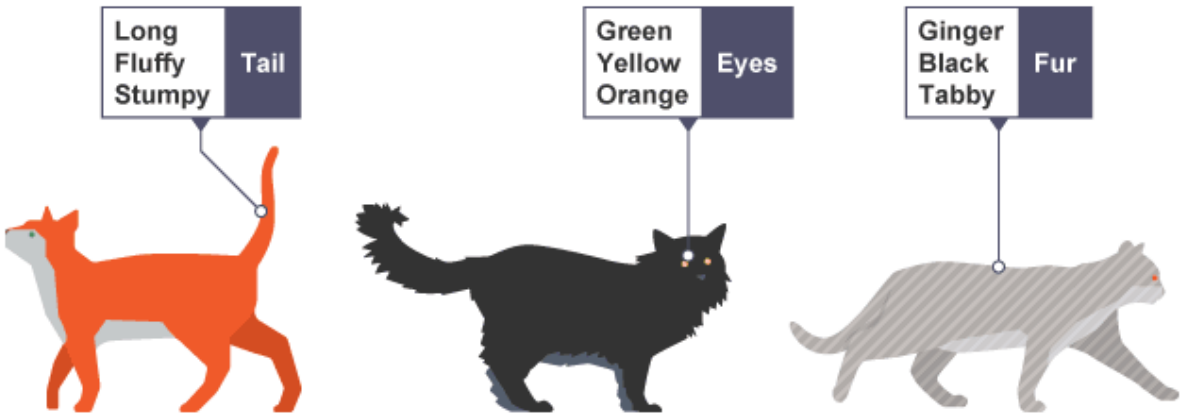
- a cake needs ingredients
- each ingredient needs a specified quantity
- a cake needs timings

When abstracting, we remove specific details and keep the general relevant patterns.

General patterns	Specific details
We need to know that a cake has ingredients	We don't need to know what those ingredients are
We need to know that each ingredient has a specified quantity	We don’t need to know what that quantity is
We need to know that each cake needs a specified time to bake	We don't need to know how long the time is

Creating a model

A model is a general idea of the problem we are trying to solve.
For example, a model cat would be any cat. Not a specific cat with a long tail and short fur - **the model represents all cats**. From our model of cats, we can learn what any cat looks like, using the patterns all cats share.

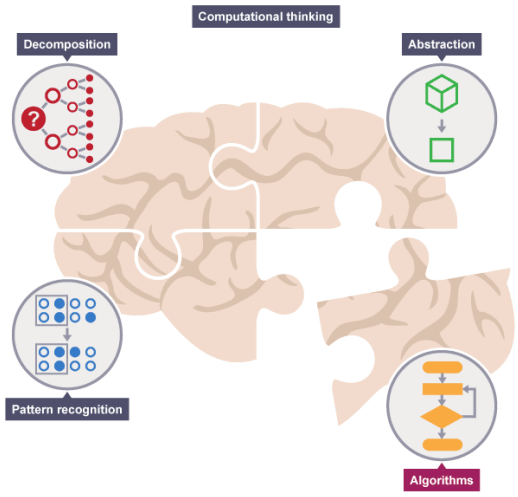


Similarly, when baking a cake, a model cake wouldn’t be a specific cake, like a sponge cake or a fruit cake. Instead, the model would represent all cakes. From this model we can learn how to bake any cake, using the patterns that apply to all cakes.
Once we have a model of our problem, we can then design an algorithm to solve it.

Lesson 4

What is an algorithm?

Algorithms are one of the four cornerstones of Computer Science. **An algorithm is a plan, a set of step-by-step instructions to solve a problem.** If you can tie shoelaces, make a cup of tea, get dressed or prepare a meal then you already know how to follow an algorithm.



In an algorithm, each instruction is identified and the order in which they should be carried out is planned. Algorithms are often used as a starting point for creating a computer program, and they are sometimes written as a flowchart or in pseudocode. If we want to tell a computer to do something, we have to write a computer program that will tell the computer, step-by-step, exactly what we want it to do and how we want it to do it. **This step-by-step program will need planning, and to do this we use an algorithm.**

Computers are only as good as the algorithms they are given. If you give a computer a poor algorithm, you will get a poor result – hence the phrase: 'Garbage in, garbage out.'

Algorithms are used for many different things including calculations, data processing and automation.

Making a plan

It is important to plan out the solution to a problem to make sure that it will be correct. Using computational thinking and decomposition we can break down the problem into smaller parts and then we can plan out how they fit back together in a suitable order to solve the problem.

This order can be represented as an algorithm. An algorithm must be clear. It must have a starting point, a finishing point and a set of clear instructions in between.

Representing an algorithm: Pseudocode

There are two main ways that algorithms can be represented - pseudocode and flowcharts. Most programs are developed using programming languages. These languages have specific syntax that must be used so that the program will run properly. Pseudocode is not a programming language, it is a simple way of describing a set of instructions that does not have to use specific syntax.

Writing in pseudocode is similar to writing in a programming language. Each step of the algorithm is written on a line of its own in sequence. Usually, instructions are written in uppercase, variables in lowercase and messages in sentence case.

In pseudocode, INPUT asks a question. OUTPUT prints a message on screen.
A simple program could be created to ask someone their name and age, and to make a comment based on these. This program represented in pseudocode would look like this:

```
OUTPUT 'What is your name?'
INPUT user inputs their name
STORE the user's input in the name variable
OUTPUT 'Hello' + name
OUTPUT 'How old are you?'
INPUT user inputs their age
STORE the user's input in the age variable
IF age >= 70 THEN
        OUTPUT 'You are aged to perfection!'
ELSE
        OUTPUT 'You are a spring chicken!'
```

Representing an algorithm: Flowcharts

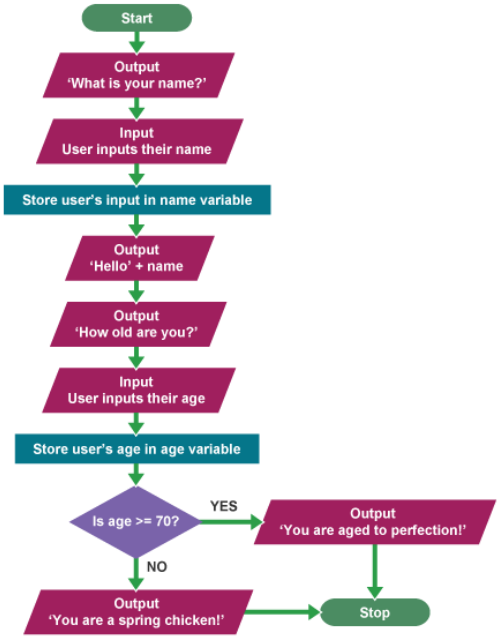
A flowchart is a diagram that represents a set of instructions. Flowcharts normally use standard symbols to represent the different instructions. There are few real rules about the level of detail needed in a flowchart. Sometimes flowcharts are broken down into many steps to provide a lot of detail about exactly what is happening. Sometimes they are simplified so that a number of steps occur in just one step.

Flowchart symbols

Flowchart symbols

Name	Symbol	Usage
Start or Stop	Start/Stop	The beginning and end points in the sequence.
Process	Process	An instruction or a command.
Decision	Decision	A decision, either yes or no.
Input or Output	Input/Output	An input is data received by a computer. An output is a signal or data sent from a computer.
Connector		A jump from one point in the sequence to another.
Direction of flow		Connects the symbols. The arrow shows the direction of flow of instructions.

A simple program could be created to ask someone their name and age, and to make a comment based on these. This program represented as a flowchart would look like this:



Lesson 5

Evaluating solutions

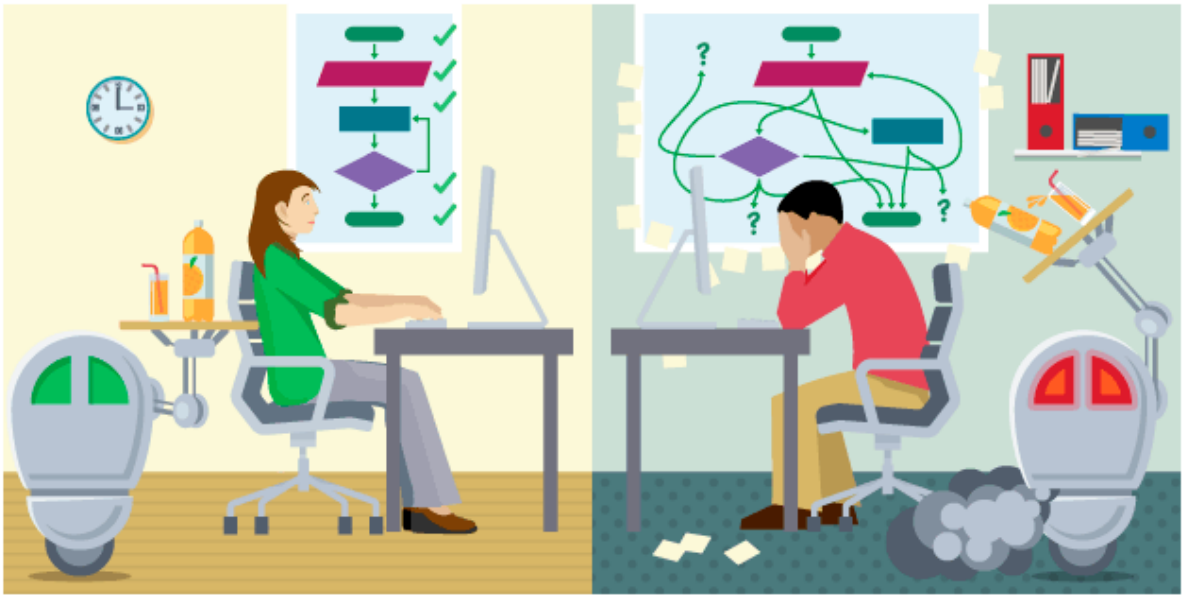
What is evaluation?

Once a solution has been designed using computational thinking, it is important to make sure that the solution is fit for purpose.

Evaluation is the process that allows us to make sure our solution does the job it has been designed to do and to think about how it could be improved.
Once written, an algorithm should be checked to make sure it:

- is easily understood – is it fully decomposed?
- is complete – does it solve every aspect of the problem?
- is efficient – does it solve the problem, making best use of the available resources (eg as quickly as possible/using least space)?
- meets any design criteria we have been given

If an algorithm meets these four criteria it is likely to work well. The algorithm can then be programmed.
Failure to evaluate can make it difficult to write a program. Evaluation helps to make sure that as few difficulties as possible are faced when programming the solution.



Why do we need to evaluate our solutions?

Computational thinking helps to solve problems and design a solution – an algorithm – that can be used to program a computer. However, if the solution is faulty, it may be difficult to write the program. Even worse, the finished program might not solve the problem correctly.

Evaluation allows us to consider the solution to a problem, make sure that it meets the original design criteria, produces the correct solution and is fit for purpose - before programming begins.

What happens if we don't evaluate our solutions?

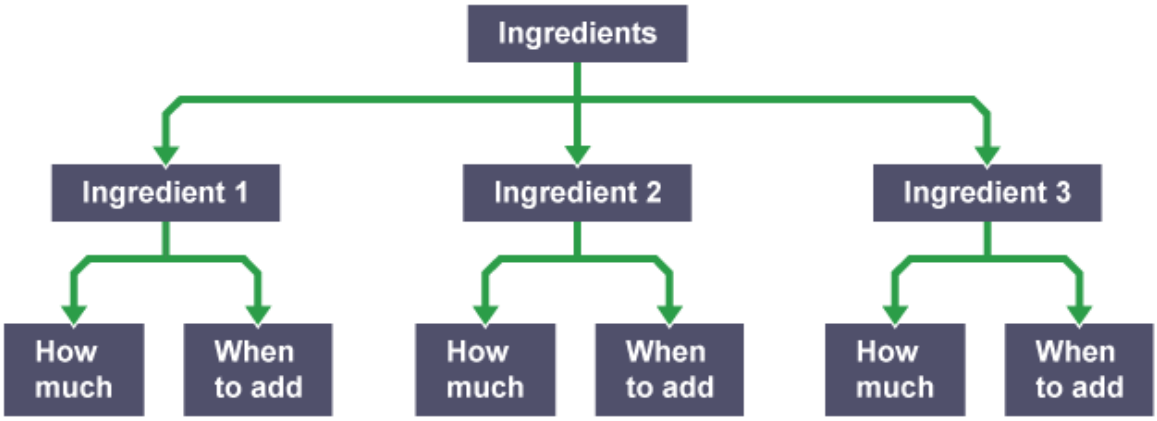
Once a solution has been decided and the algorithm designed, it can be tempting to miss out the evaluating stage and to start programming immediately. **However, without evaluation any faults in the algorithm will not be picked up, and the program may not correctly solve the problem, or may not solve it in the best way.**
Faults may be minor and not very important. For example, if a solution to the question 'how to draw a cat?' was created and this had faults, all that would be wrong is that the cat drawn might not look like a cat. However, faults can have huge – and terrible – effects, e.g. if the solution for an aeroplane autopilot had faults.

Ways that solutions can be faulty

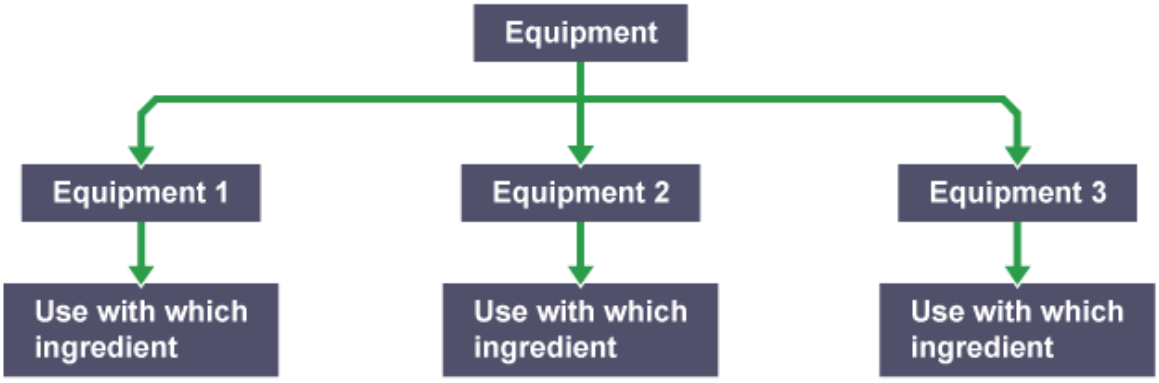
We may find that solutions fail because:
it is **not fully understood** - we may not have properly decomposed the problem
it is **incomplete** - some parts of the problem may have been left out accidentally
it is **inefficient** – it may be too complicated or too long
it **does not meet the original design criteria** – so it is not fit for purpose
A faulty solution may include one or more of these errors.

Solutions that are not properly decomposed

If computational thinking techniques are applied to the problem of how to bake a cake, on decomposing the problem, it is necessary to know:
what kind of cake to bake?
what ingredients are needed, how much of each ingredient, and when to add it
how many people the cake is for
how long to bake the cake for?
what equipment is needed
A diagram of a further decomposition of **ingredients** would look like this:



At the moment, a diagram of the further decomposition of **equipment** would look like this: the 'Equipment' part is not properly broken down (or decomposed). Therefore, if the solution - or algorithm – were created from this, baking the cake would run into problems. The algorithm would say what equipment is needed, but not how to use it, so a person could end up trying to use a knife to measure out the flour and a whisk to cut a lump of butter, for example. This would be wrong and would, of course, not work. Ideally, then, 'Equipment' should be decomposed further, to state which equipment is needed and which ingredients each item is used with.



The problem occurred here because the problem of which equipment to use and which ingredients to use it with hadn't been fully decomposed.

Computer Science Y7

Unit 2 Algorithms

Lesson 1

Designing an algorithm

An **algorithm** is a plan, a logical step-by-step process for solving a problem. Algorithms are normally written as a **flowchart** or in **pseudocode**.
The key to any problem-solving task is to guide your thought process. The most useful thing to do is keep asking 'What if we did it this way?' Exploring **different** ways of solving a problem can help to find the **best** way to solve it.

When designing an algorithm, consider if there is more than one way of solving the problem.

When designing an algorithm there are two main areas to look at:

- the **big picture** - What is the final goal?
- the **individual stages** – What hurdles need to be overcome on the way to the goal?

Understanding the problem

Before an algorithm can be designed, it is important to check that the problem is completely understood. There are a number of basic things to know in order to really understand the problem:

- What are the **inputs** into the problem?
- What will be the **outputs** of the problem?
- In what order do **instructions** need to be carried out?
- What decisions need to be made in the problem?
- Are any areas of the problem repeated?

Once these basic things are understood, it is time to design the algorithm.

Pseudocode

Most programs are developed using programming languages. These languages have specific syntax that must be used so that the program will run properly. Pseudocode is not a programming language, it is a simple way of describing a set of instructions that does not have to use specific syntax.

Common pseudocode notation

There is no strict set of standard notations for pseudocode, but some of the most widely recognised are:
INPUT – indicates a user will be inputting something
OUTPUT – indicates that an output will appear on the screen
WHILE – a loop (iteration that has a condition at the beginning)
FOR – a counting loop (iteration)
REPEAT – UNTIL – a loop (iteration) that has a condition at the end
IF – THEN – ELSE – a decision (selection) in which a choice is made
any instructions that occur inside a selection or iteration are usually indented

Using pseudocode

Pseudocode can be used to plan out programs. Planning a program that asks people what the best subject they take is, would look like this in pseudocode:

```
REPEAT
        OUTPUT 'What is the best subject you take?'
        INPUT user inputs the best subject they take
        STORE the user's input in the answer variable
        IF answer = 'Computer Science' THEN
                OUTPUT 'Of course it is!'
        ELSE
                OUTPUT 'Try again!'
UNTIL answer = 'Computer Science'
```

Flowcharts

A flowchart is a diagram that represents a set of instructions. Flowcharts normally use standard symbols to represent the different types of instructions. These symbols are used to construct the flowchart and show the step-by-step solution to the problem.

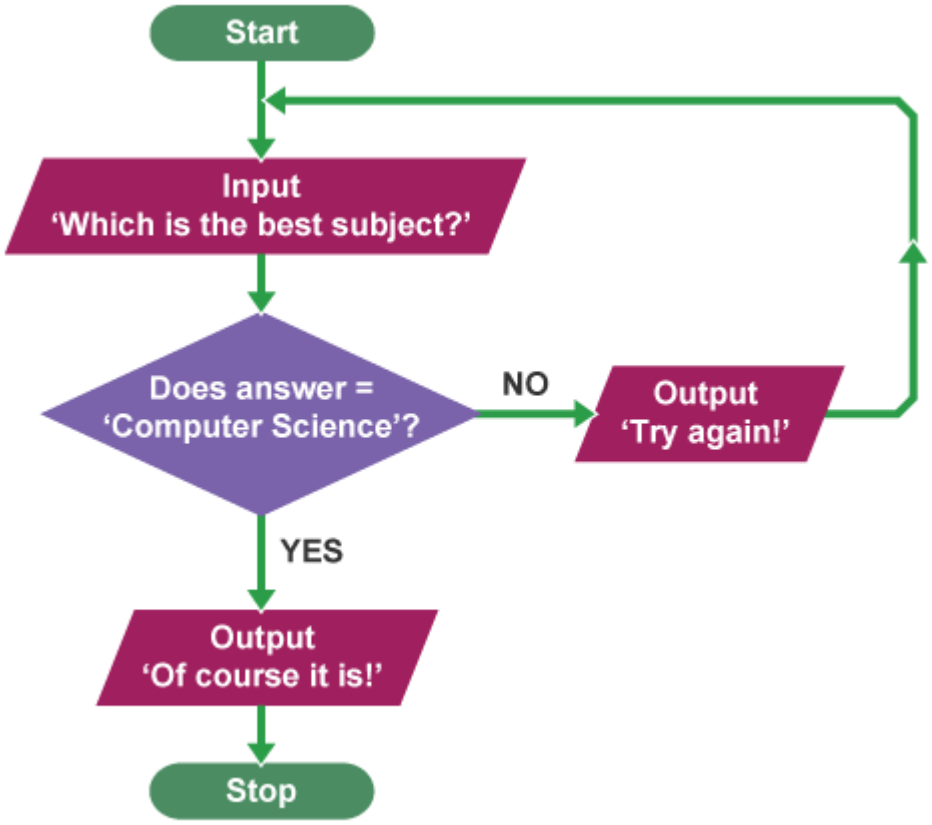
Using flowcharts

Flowcharts can be used to plan out programs. Planning a program that asks people what the best subject they take is, would look like this as a flowchart:

Name	Symbol	Usage
Start or Stop		The beginning and end points in the sequence.
Process		An instruction or a command.
Decision		A decision, either yes or no.
Input or Output		An input is data received by a computer. An output is a signal or data sent from a computer.
Connector		A jump from one point in the sequence to another.
Direction of flow		Connects the symbols. The arrow shows the direction of flow of instructions.

Using flowcharts

Flowcharts can be used to plan out programs. Planning a program that asks people what the best subject they take is, would look like this as a flowchart



Lesson 2

Why do we need searching algorithms?

We often need to find one particular item of data amongst many hundreds, thousands, millions or more. For example, you might need to find someone's phone number on your phone, or a particular business's address in the UK.

This is why searching algorithms are important. Without them you would have to look at each item of data – each phone number or business address – individually, to see whether it is what you are looking for. In a large set of data, it will take a long time to do this. Instead, a searching algorithm can be used to help find the item of data you are looking for.

Search algorithms prevent you from having to look through lots of data to find the information you are searching for

There are many different types of searching algorithms. Two of them are **serial search** and **binary search**

Serial search

Searching for a **keyword** or value is the foundation of many computer **programs**. The most basic kind of search is a serial search.
Criteria are set up before the search begins. The search then starts with the first item and then moves to each item in turn, until either a match is found or it reaches the end of the data with no match found.

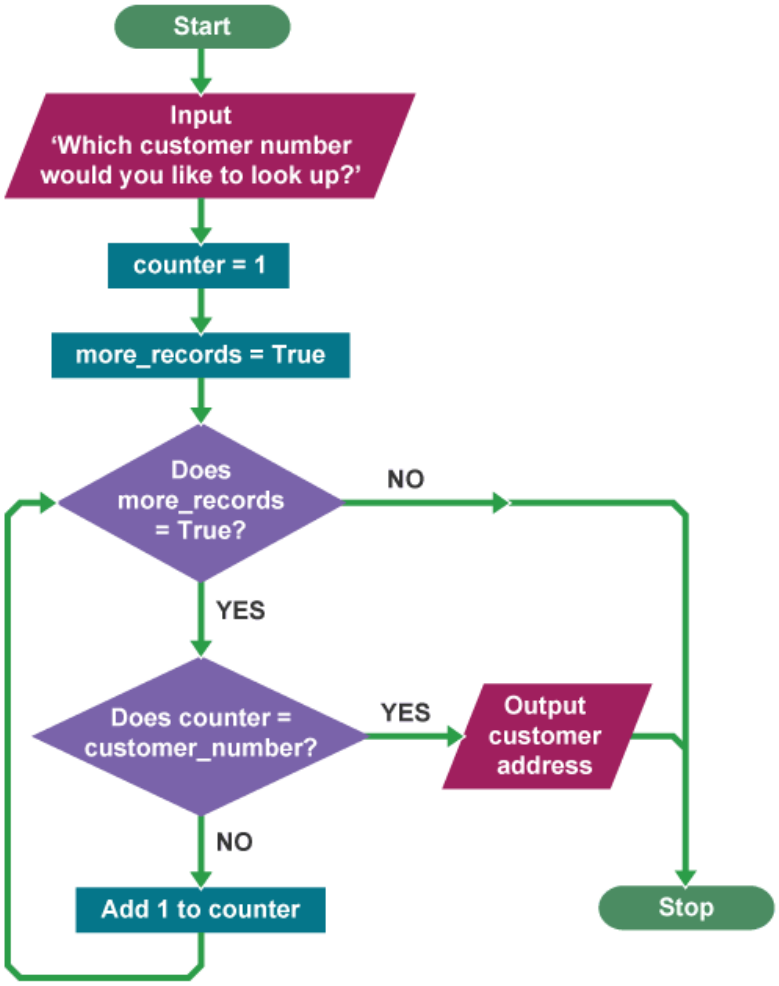
Example

Imagine that you have a **database** of sales made to customers. You need to deliver the goods that customer number 150 has bought, so need to find their address in the database.
The criterion is set first - **Find the address for customer 150**.

A serial search will begin at customer 1 and will search through each customer in turn until it reaches customer 150. It will then output the address for this customer. If it does not find customer 150, a message will be output to say that the customer is not found.
In **pseudocode** this would look like:

```
OUTPUT "Which customer number would you like to look up?"
INPUT user inputs customer number
STORE the user's input in the customer_number variable
counter = 1
(we need to count the number of records that we have searched through)
more_records = True
(we need a flag to say if more records are available to search through
or if we have reached the end of the database)
WHILE more_records = True:
        IF counter = customer_number THEN
                OUTPUT customer address
                Exit the loop
ELSE
        add 1 to counter
```

As a flowchart, this would look like:



Binary search

Binary search is a faster method for searching for an item that is in an **ordered list**. **An ordered list is one where the sequence of items in the list is important.** An ordered list does not necessarily contain a sequence of numbers (eg 1, 2, 3, 4) or characters (eg A, B, C, D). It might also contain, eg a list of names in alphabetical order, a list of files from smallest to largest or a list of records from earliest to most recent.

A binary search algorithm takes the data and keeps dividing it in half until it finds the item it is looking for.

Example

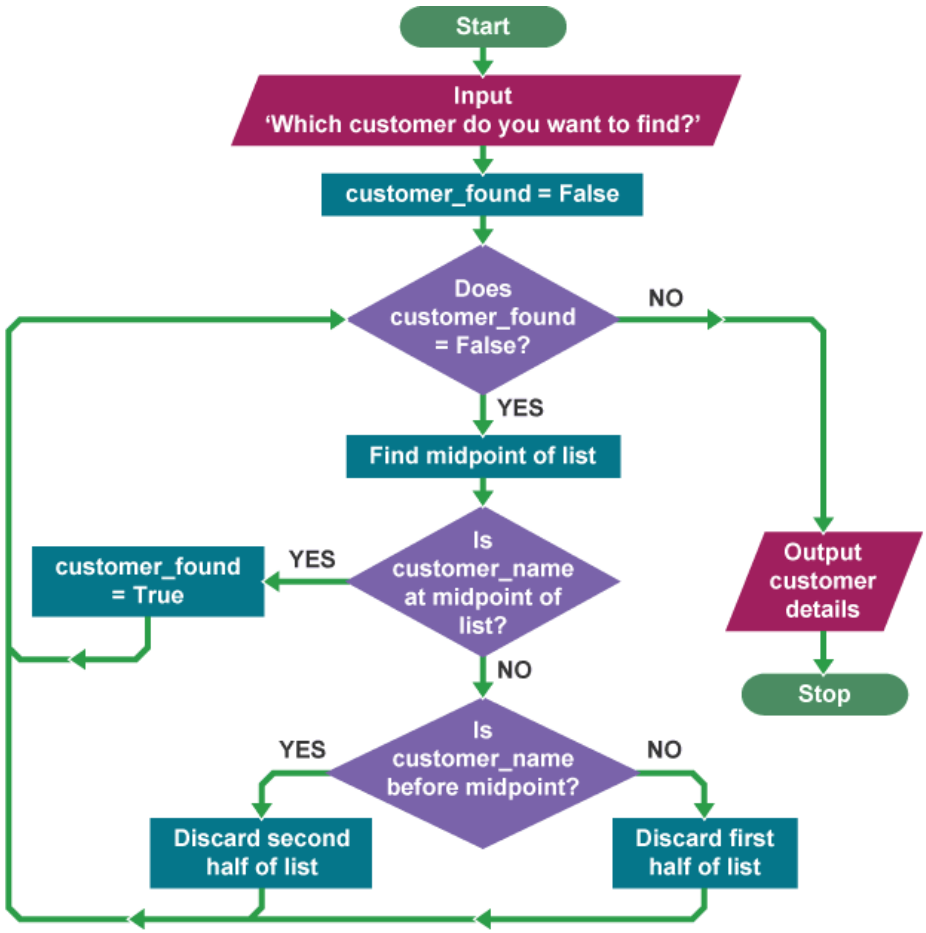
Imagine that you have a database of customers and want to search for the customer John Smith. We first need the database to be ordered into alphabetical order by surname. We then search for the record 'John Smith' by surname.

The binary search will split the database in half, and compare the midpoint (the middle name) to the name 'John Smith'. It will see whether the midpoint comes before or after

‘Smith’ and will throw away the set of records that doesn’t contain ‘Smith’ as a surname. It will keep dividing the records in that way until it reaches two records left, one of which is ‘John Smith’. It will then throw away the other record and output John Smith’s details. Binary search effectively divides the data in half and throws away, or ‘bins’ the half that does not contain the search term.
In pseudocode this would look like:

```
OUTPUT "Which customer do you want to find?"
INPUT user inputs John Smith
STORE the user's input in the customer_name variable
customer_found = False
(we need to create a flag that identifies if the customer is found)
WHILE customer_found = False:
        Find midpoint of list
        IF customer_name = record at midpoint of list THEN
                customer_found = True
        ELSE IF customer comes before the midpoint THEN
                throw away the second half of the list
        ELSE
                throw away the first part of the list
OUTPUT customer details
```

As a flowchart, this would look like:



Comparison of searches

Different algorithms might be best used in different situations. For example, sometimes an algorithm won't work with a particular set of data, and in some instances one algorithm will be much quicker or more efficient than another.

Serial search

One of the main advantages of a serial search is that it is a very simple algorithm, which makes it very easy to write a computer program to carry it out. It can also be used on any set of data regardless of type and whether or not it is sorted.
The biggest problem with a serial search is that it is very slow. For example, when searching through a database of everyone in the UK to find a particular name, it might be necessary to search through 60 million names before you found the one you wanted.

Binary search

One of the main advantages of a binary search is that it is much quicker than a serial search because the data that needs to be searched halves with each step. For example, it is possible to search through 1024 values and find the one you want within 10 steps, every time.

The biggest problem with a binary search is that you can only use this if the data is sorted into an order.

Lesson 3

Why do we need sorting algorithms?

A sorting **algorithm** will put items in a list into an order, such as alphabetical or numerical order. For example, a list of customer names could be sorted into alphabetical order by surname, or a list of people could be put into numerical order by age.

Sorting a list of items can take a long time, especially if it is a large list. A computer **program** can be created to do this, making sorting a list of data much easier. There are many types of sorting algorithms. Two of them are **bubble sort** and **bucket sort**.

Bubble sort

A bubble sort algorithm goes through a list of data a number of times, comparing two items that are side by side to see which is out of order. It will keep going through the list of data until all the data is sorted into order. Each time the algorithm goes through the list it is called a 'pass'.

Example

Imagine that you have a list of people who you want to sort by age, from youngest to oldest. A bubble sort can do this.

The list of ages is:

41, 15, 17, 32, 18, 28, 77 and 54

First pass

The highlighted numbers are the numbers that are being compared.

41, 15, 17, 32, 18, 28, 77, 54

This is the list before it is sorted.
41, 15, 17, 32, 18, 28, 77, 54

The first two numbers are compared. 15 is smaller than 41 so they switch places.
15, **41, 17,** 32, 18, 28, 77, 54

The next two numbers are compared. 17 is smaller than 41 so they switch places.
15, 17, **41, 32,** 18, 28, 77, 54

The next two numbers are compared. 32 is smaller than 41 so they switch places.
15, 17, 32, **41, 18,** 28, 77, 54

The next two numbers are compared. 18 is smaller than 41 so they switch places.
15, 17, 32, 18, **41, 28,** 77, 54

The next two numbers are compared. 28 is smaller than 41 so they switch places.

15, 17, 32, 18, 28, **41, 77,** 54

The next two numbers are compared. 41 is smaller than 77 so no change occurs.
15, 17, 32, 18, 28, 41, **77, 54**

The next two numbers are compared. 54 is smaller than 77 so they switch places.

15, 17, 32, 18, 28, 41, 54, 77
This is what the list looks like after the first pass.

The list is now more ordered than it was originally, but it isn't yet fully in order of youngest to oldest. The list needs to go through another pass to compare the numbers again, so it can be sorted further.

Second pass

15, 17, 32, 18, 28, 41, 54, 77

This is what the list looks like after the first pass.
15, 17, 32, 18, 28, 41, 54, 77

The first two numbers are compared. 15 is smaller than 17 so no change occurs.
15, **17, 32,** 18, 28, 41, 54, 77

The next two numbers are compared. 17 is smaller than 32 so no change occurs.
15, 17, **32, 18** 28, 41, 54, 77

The next two numbers are compared. 18 is smaller than 32 so they switch places.
15, 17, 18, **32, 28,** 41, 54, 77

The next two numbers are compared. 28 is smaller than 32 so they switch places.
15, 17, 18, 28, **32, 41,** 54, 77

The next two numbers are compared. 32 is smaller than 41 so no change occurs.
15, 17, 18, 28, 32, **41, 54,** 77

The next two numbers are compared. 41 is smaller than 54 so no change occurs.
15, 17, 18, 28, 32, 41, **54, 77**

The next two numbers are compared. 54 is smaller than 77 so no change occurs.

15, 17, 18, 28, 32, 41, 54, 77

This is what the list looks like after the second pass.

The set of data is now in order from youngest to oldest, but the algorithm does not know this yet as it made some changes in the second pass. The algorithm will only recognise that the list is in order if it makes no changes in a pass. The algorithm therefore needs to run for a third pass to compare the numbers again. As no changes will be made, the algorithm will then recognise that the data is in order.

If the data being sorted is a large set of data, it may take several passes to get the data sorted.

Lesson 4

Sequencing in algorithms

An **algorithm** is a plan, a set of step-by-step instructions to solve a problem. There are three basic building blocks (constructs) to use when designing algorithms:

- sequencing
- selection
- iteration

These building blocks help to describe solutions in a form ready for **programming**.

What is sequencing?

Algorithms consist of instructions that are carried out (performed) one after another. **Sequencing is the specific order in which instructions are performed in an algorithm**.

For example, a very simple algorithm for brushing teeth might consist of these steps:

1. put toothpaste on toothbrush
2. use toothbrush to clean teeth
3. rinse toothbrush

Each step is an instruction to be performed. Sequencing is the order in which the steps are carried out.

Why is sequencing important?

It is crucial that the steps in an algorithm are performed in the right order - otherwise the algorithm will not work correctly. Suppose the steps for the teeth-cleaning algorithm were in this sequence:

1. use toothbrush to clean teeth
2. put toothpaste on toothbrush
3. rinse toothbrush

A toothbrush would still be used to clean the teeth and toothpaste would still be put on the brush. But because steps 1 and 2 are in the wrong sequence the teeth wouldn't get cleaned with the toothpaste, and the toothpaste would be wasted.

A human would realise they had forgotten to add toothpaste at the start of the process, but a computer would not know that anything was wrong.

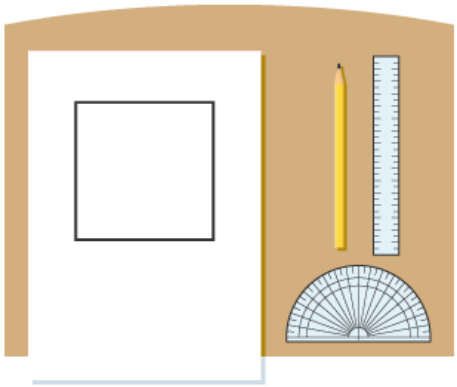
A computer can only do what it is programmed to do. If the steps are programmed in the wrong sequence, the computer will perform the tasks in this sequence – even if this is incorrect.

Sequencing in practice: Drawing a square

An algorithm to get a computer to draw a square on the screen might consist of these steps:

1. draw a 3 cm line
2. turn left 90 degrees
3. draw a 3 cm line
4. turn left 90 degrees
5. draw a 3 cm line
6. turn left 90 degrees
7. draw a 3 cm line

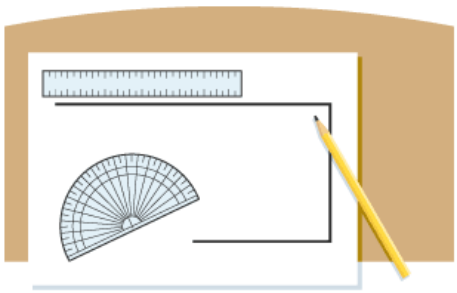
The steps in this algorithm are in the correct sequence. This algorithm would result in a perfect square.



If there was a mistake when designing the algorithm, and the steps in this sequence were placed like this:

1. draw a 3 cm line
2. turn left 90 degrees
3. draw a 3 cm line
4. turn left 90 degrees
5. draw a 3 cm line
6. draw a 3 cm line
7. turn left 90 degrees

This algorithm would create this shape, rather than a perfect square:



Because step 6 is in the wrong sequence (it should switch with step 7), this algorithm failed. However, fixing the algorithm - and the square - is easy, because there are only seven steps to look through for the error.

Complex algorithms may have hundreds, if not thousands, of steps. It is critical to make sure all steps in the algorithm are in the correct sequence before programming begins. Once programmed, trying to find an instruction in the wrong sequence can be extremely difficult.

Representing sequencing

There are two ways of representing **algorithms**:

- **pseudocode**
- a **flowchart**

Representing sequencing in pseudocode

Writing in pseudocode is rather like writing in a **programming language**. Each step of the algorithm is written on a line of its own, in sequence.

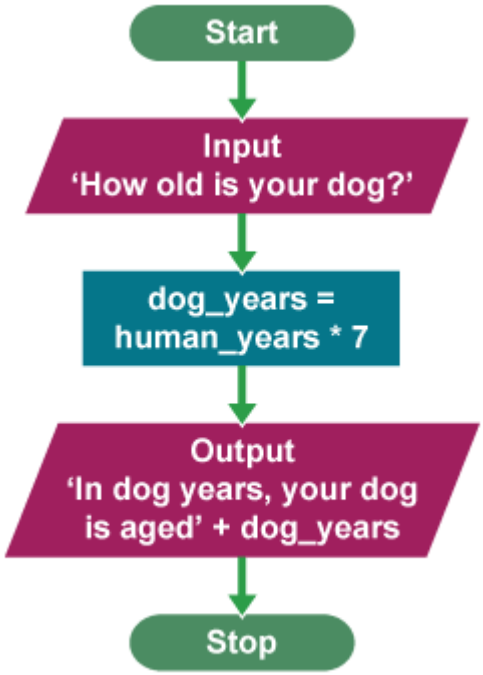
Consider this simple algorithm for calculating how old a dog is in dog years. It contains three steps, all in sequence:

- ask how old the dog is in human years
- multiply human years by seven to find out how old the dog is in dog years
- print the answer on the screen

In pseudocode, the algorithm would look like this:

```
OUTPUT 'How old is your dog?'
INPUT user inputs their dog's age in human years
STORE the user's input in the human_years variable
dog_years = human_years * 7
OUTPUT 'In dog years, your dog is aged' + dog_years
```

Representing sequencing in a flowchart



Lesson 5

Selection

Selection in algorithms

An **algorithm** is a plan, a set of step-by-step instructions to solve a problem. There are three basic building blocks (constructs) to use when designing algorithms:

- sequencing
- selection
- iteration

These building blocks help to describe solutions in a form ready for **programming**.

What is selection?

Selection is a decision or question.

At some point in an algorithm there may need to be a question because the algorithm has reached a step where one or more options are available. Depending on the answer given, the algorithm will follow certain steps and ignore others.

Why is selection important?

Selection allows us to include more than one path through an algorithm.

Without selection, different paths would not be included in algorithms. This means that the solutions created would not be realistic.

Selection in practice: How old are you?

Algorithms consist of a set of **instructions** that are carried out (performed) one after another. Sometimes there may be more than one path (or set of steps) that can be followed. At this point, a decision needs to be made. This point is known as **selection**.

For example, a simple algorithm can be created to determine correct bus fares. The steps could be:

1. ask how old you are
2. if you are under 16, pay half fare
3. otherwise pay full fare

The decision comes in step 2. If you are aged less than 16, one fare is charged. Otherwise, a different fare is charged.

IF…THEN…ELSE

Selection allows several paths to be included in an algorithm. In algorithms (and in programming), selection is usually represented by the instructions **IF**, **THEN** and **ELSE**.

- **IF** represents the **question**
- **THEN** points to what to do if the answer to the question is **true**
- **ELSE** points to what to do if the answer to the question is **false**

Using this method, the fare-related algorithm now reads like this:

1. ask how old you are
2. IF you are under 16, THEN pay half fare
3. ELSE pay full fare

If you try this algorithm using 15 as your age, the answer to the question at step 2 is true, so the algorithm tells you to pay half fare.

If you try the algorithm using 16 as your age, the answer to the question at step 2 is false, so the algorithm tells us to pay full fare.

Two different paths are followed according to the answer given to a particular question.

Representing selection

There are two ways of representing **algorithms**:

- **pseudocode**
- a **flowchart**

Representing selection in pseudocode

Writing in pseudocode is rather like writing in a **programming language**. Each step of the algorithm is written on a line of its own, in sequence.

Look at this simple six-step algorithm for comparing your dog's age with your own:

- ask how old the dog is in human years
- multiply human years by seven to find out how old the dog is in dog years
- print the answer on the screen
- ask how old you are
- if the dog's age in dog years is older than your age, say 'Your dog is older than you!'
- otherwise, say 'Your dog is not older than you.'

In pseudocode, the algorithm would look like this:

```
OUTPUT 'How old is your dog?'
INPUT user inputs their dog's age in human years
STORE the user's input in the human_years variable
dog_years = human_years * 7
OUTPUT 'In dog years, your dog is aged ' + dog_years
OUTPUT 'How old are you?'
INPUT user inputs their age
STORE the user's input in the user_age variable
IF dog_years > user_age THEN
        OUTPUT 'Your dog is older than you!'
ELSE
        OUTPUT 'Your dog is not older than you.'
```

NOTE: when using pseudocode, you do not need to use THEN with IF and ELSE - although you can if you want to.

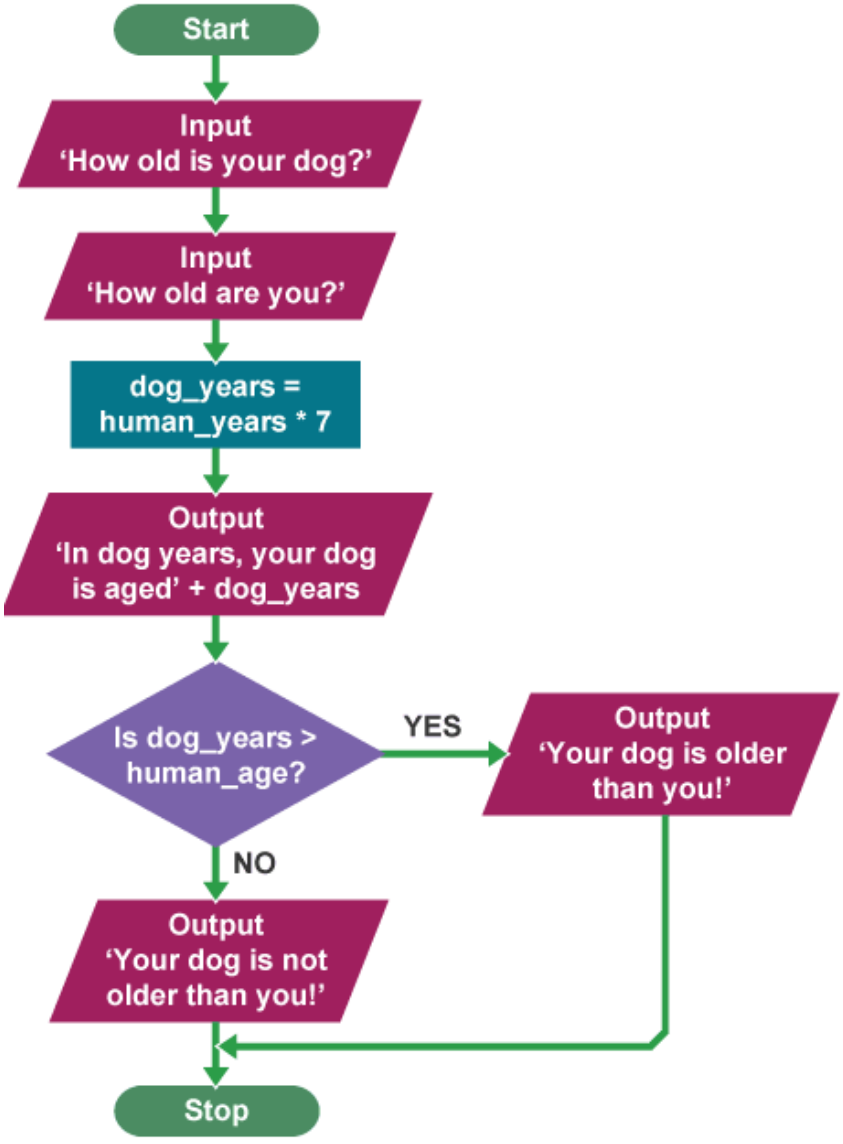
Representing selection in a flowchart

A **flowchart** uses a diagram to explain the steps of an **algorithm**.

Consider this simple six-step algorithm for comparing your dog's age with your own:

- ask how old the dog is in human years
- multiply human years by seven to find out how old the dog is in dog years
- print answer on the screen
- ask how old you are
- if the dog's age in dog years is older than your age, say 'Your dog is older than you!'
- otherwise, say 'Your dog is not older than you.'

In a flowchart, this sequence would look like this:



The ELSE IF instruction

The **ELSE IF** instruction allows there to be more than two paths through an **algorithm**. Any number of ELSE IF **instructions** can be added to an algorithm. It is used along with other instructions:

- **IF** represents a **question**
- **THEN** points to what to do if the answer to the question is **true**
- **ELSE IF** represents another **question**
- **THEN** points to what to do if the answer to that question is **true**
- **ELSE IF** represents another **question**
- **THEN** points to what to do if the answer to that question is **true**
- **ELSE** points to what to do if the answer to the question is **false**

Using this method, the bus fare algorithm could be improved like this:

1. ask how old you are
2. IF you are under 5, THEN pay no fare
3. ELSE IF you are under 16, THEN pay half fare
4. ELSE IF you are an OAP, THEN pay no fare
5. ELSE pay full fare

Lesson 6

Iteration

Iteration in algorithms

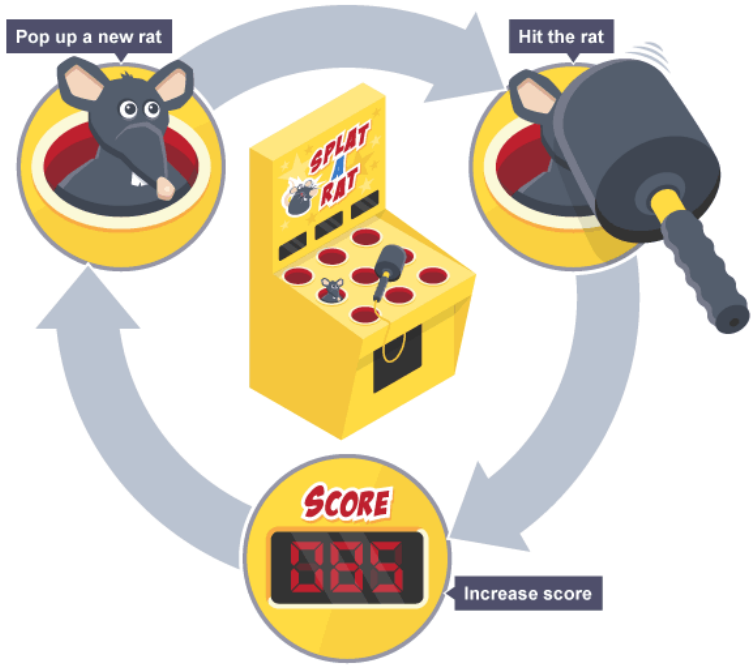
An **algorithm** is a plan, a set of step-by-step instructions to solve a problem. There are three basic building blocks (constructs) to use when designing algorithms:

- sequencing
- selection
- iteration

These building blocks help to describe solutions in a form ready for **programming**.

What is iteration?

Iteration in programming means repeating steps, or **instructions**, over and over again. This is often called a 'loop'.



Algorithms consist of instructions that are carried out (performed) one after another. Sometimes an algorithm needs to repeat certain steps until told to stop or until a particular **condition** has been met.

Iteration is the process of repeating steps.

For example, a very simple algorithm for eating breakfast cereal might consist of these steps:

- put cereal in bowl
- add milk to cereal
- spoon cereal and milk into mouth
- repeat step 3 until all cereal and milk is eaten
- rinse bowl and spoon

Why is iteration important?

Iteration allows algorithms to be simplified by stating that certain steps will repeat until told otherwise. This makes designing algorithms quicker and simpler because they don't need to include lots of unnecessary steps.

Iteration in practice: Cleaning teeth

A simple algorithm can be created for cleaning teeth. Suppose a person has ten top teeth. To make sure that every one of the top teeth is cleaned, the algorithm would look something like this:

1. put toothpaste on toothbrush
2. use toothbrush to clean tooth 1
3. use toothbrush to clean tooth 2
4. use toothbrush to clean tooth 3
5. use toothbrush to clean tooth 4
6. use toothbrush to clean tooth 5
7. use toothbrush to clean tooth 6
8. use toothbrush to clean tooth 7
9. use toothbrush to clean tooth 8
10. use toothbrush to clean tooth 9
11. use toothbrush to clean tooth 10
12. rinse toothbrush

Steps 2 through to 11 are essentially the same step repeated, just cleaning a different tooth every time. Iteration can be used to greatly simplify the algorithm. Look at this alternative:

1. put toothpaste on toothbrush
2. use toothbrush to clean a tooth
3. move onto next tooth
4. repeat steps 2 and 3 until all teeth are clean
5. rinse toothbrush

This algorithm is much simpler. However, there is a problem – how do we know when all teeth are clean (step 4)? A condition is needed to solve this problem.

Conditions and counters

A **condition** is a situation that is checked every time an **iteration** occurs.
The following is a 12-step **algorithm** for cleaning a person's upper teeth, and supposes that they have ten top teeth:

1. put toothpaste on toothbrush
2. use toothbrush to clean tooth 1
3. use toothbrush to clean tooth 2
4. use toothbrush to clean tooth 3
5. use toothbrush to clean tooth 4
6. use toothbrush to clean tooth 5
7. use toothbrush to clean tooth 6
8. use toothbrush to clean tooth 7
9. use toothbrush to clean tooth 8
10. use toothbrush to clean tooth 9
11. use toothbrush to clean tooth 10
12. rinse toothbrush

The **condition**, in this case, will be to check if the number of teeth cleaned equals ten. If that condition is **False** (the number of teeth cleaned is less than ten), then another iteration occurs. If the condition is **True** (the number of teeth cleaned equals ten), then no more iterations occur.

It is also important to keep a count of how many teeth have been cleaned.

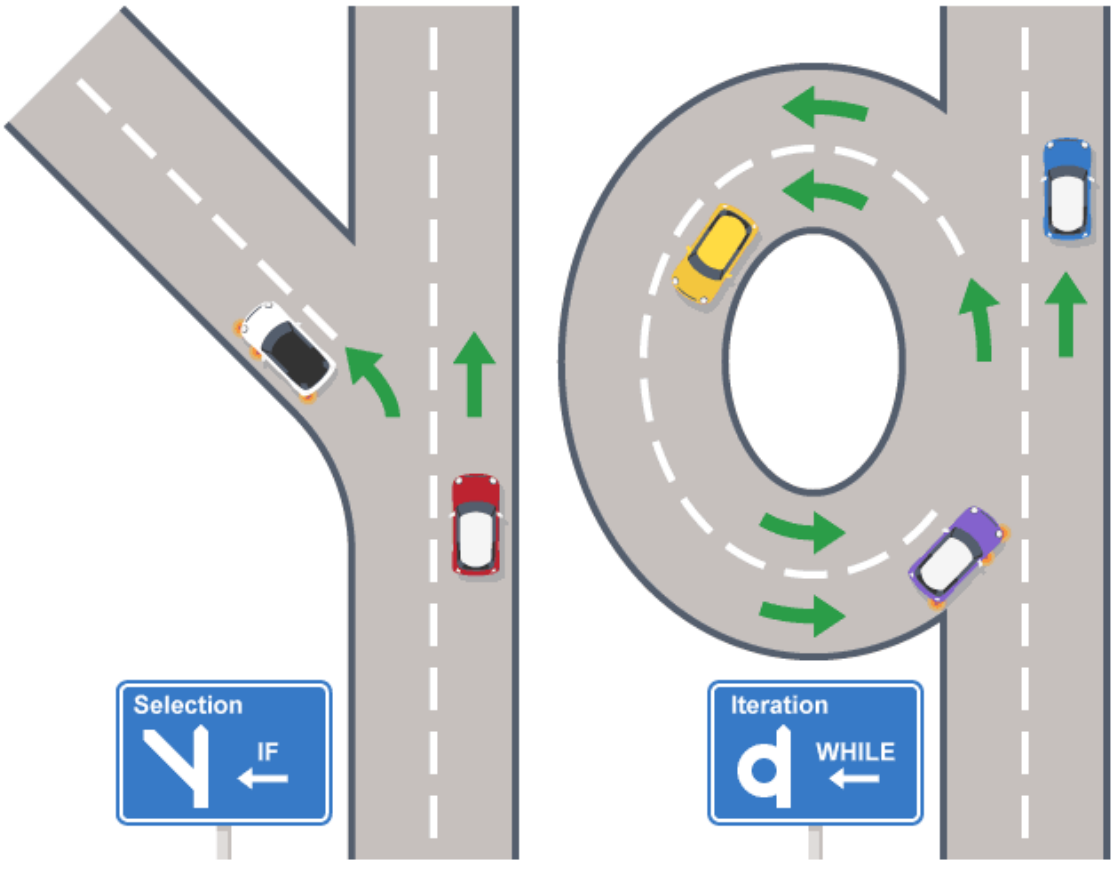
A counter is used to do this.

The counter can be used to see if the condition has been met. This is known as **testing** the condition. **The test sees whether the condition is True or False**.

If a condition and a counter are included, the algorithm would look like this:

1. set number_of_teeth_cleaned to 0
2. put toothpaste on toothbrush
3. use toothbrush to clean a tooth
4. increase number_of_teeth_cleaned by 1
5. if number_of_teeth_cleaned < 10 then go back to step 3
6. rinse toothbrush

The counter is called **number_of_teeth_cleaned**, and the condition is '**if number_of_teeth_cleaned < 10**'. This is very similar to **selection** as two routes are created. However, with iteration there is a loop back into the algorithm, rather than creating and keeping two separate routes as would occur with selection.



Using iteration, the number of steps has reduced from 12 in the first teeth-cleaning algorithm to just six in the algorithm with iteration.

Amending iteration

Most adults have 32 teeth in total. To amend the original algorithm to clean 32 teeth, another 22 steps (one for each additional tooth) would have to be added to the first algorithm - making this algorithm 34 steps long. However, in the algorithm with iteration, all that is needed is to change the condition from ten teeth to 32. The total number of steps in that algorithm remains unchanged:

1. set number_of_teeth_cleaned to 0
2. put toothpaste on toothbrush
3. use toothbrush to clean a tooth
4. increase number_of_teeth_cleaned by 1
5. if number_of_teeth_cleaned < 32 then go back to step 3
6. rinse toothbrush

Representing iteration

There are two ways of representing **algorithms**:

- **pseudocode**
- a **flowchart**

Representing iteration in pseudocode

Writing in pseudocode is rather like writing in a **programming language**. Each step of the algorithm is written on a line of its own, in sequence.

Consider this simple six-step algorithm for cleaning your teeth:

1. set number_of_teeth_cleaned to 0
2. put toothpaste on toothbrush
3. use toothbrush to clean a tooth
4. increase number_of_teeth_cleaned by 1
5. if number_of_teeth_cleaned < 32 then go back to step 3
6. rinse toothbrush

With each iteration, a decision needs to be made as to whether to continue iterating or not. Decisions are represented as selection.

In pseudocode, the algorithm would look like this:

```
number_of_teeth_cleaned = 0
put toothpaste on toothbrush
REPEAT
        procedure clean_tooth
        number_of_teeth_cleaned = number_of_teeth_cleaned + 1
UNTIL number_of_teeth_cleaned = 32
rinse toothbrush
```

Representing iteration in a flowchart

A flowchart uses a diagram to explain the steps of an algorithm.

In a flowchart, the algorithm would look like this:

Computer Science Y7

Unit 3 Binary

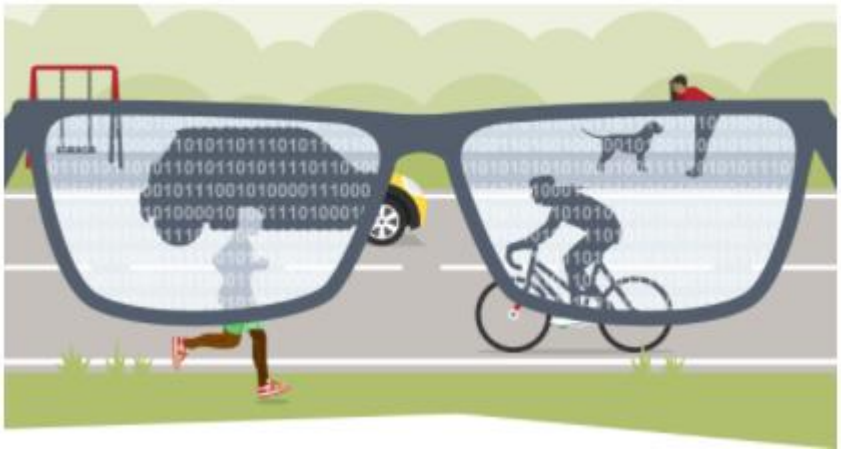
Lesson 1

Binary

How computers see the world

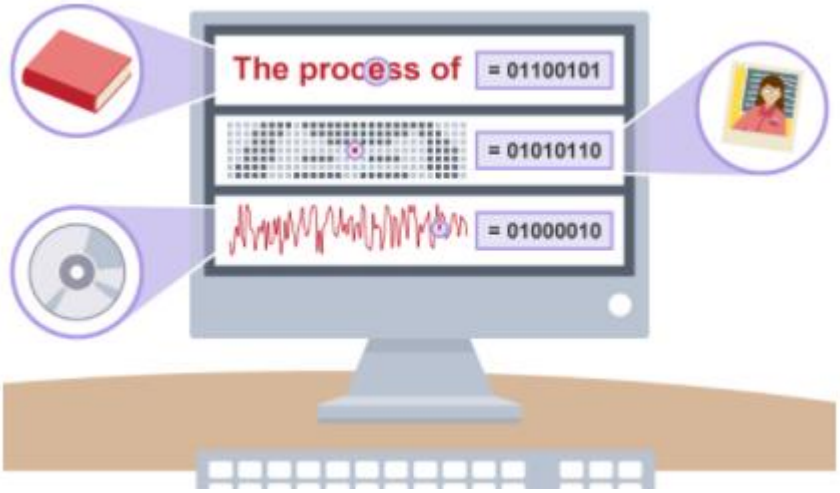
There are a number of very common needs for a computer, including the need to store and view **data**.
Computers use electrical signals that are on or off, so they have to see everything as a series of **binary** numbers. This data is represented as a sequence of 1s and 0s (on and off).

All data that we want a computer to process needs to be converted into this binary format.



What is binary?

Binary is a number system that only uses two digits: 1 and 0. All information that is processed by a computer is in the form of a sequence of 1s and 0s. Therefore, all data that we want a computer to process needs to be converted into binary.



The binary system is known as a ‘base 2’ system. This is because:

- there are only two digits to select from (1 and 0)
- when using the binary system, data is converted using the power of two.

Converting from binary to denary

People use the **denary** (or decimal) number system in their day-to-day lives. This system has 10 digits that we can use: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.

Converting from binary to denary

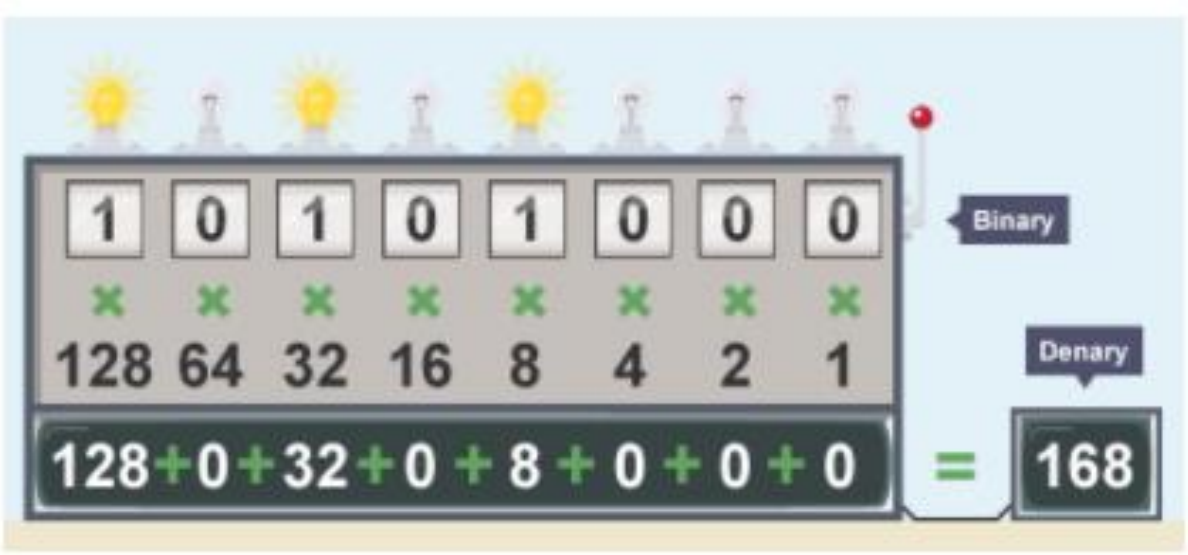
To convert a **binary** number to denary, start by writing out the binary place values. In denary, the place values are 1, 10, 100, 1000, etc – each place value is 10 times bigger than the last. In binary, each place value is 2 times bigger than the last (ie increased by the power of 2). The first few binary place values look like this:

128	64	32	16	8	4	2	1

Working out the value of 1010 1000:

128	64	32	16	8	4	2	1
1	0	1	0	1	0	0	0
1×128 +	**0**×64 +	**1**×32 +	**0**×16 +	**1**×8 +	**0**×4 +	**0**×2 +	**0**×1
128 +	0 +	32 +	0 +	8 +	0 +	0 +	0

So **1010 1000** in binary is equal to **168** in denary.



Converting from denary to binary

There is a very simple method to convert a **denary** number into a **binary** number. Let's take the number 199.

Start by writing out the first few binary **place values** (128, 64, 32, 16, 8, 4, 2, 1).

128	64	32	16	8	4	2	1

Start at the far left point and say **"Can 128 be taken away from 199?"**. If it can, do that.
199 – 128 = 71. Because 128 could be taken off, put a 1 in the '128' place value column:

128	64	32	16	8	4	2	1
1							

Now repeat for 64: **71 – 64 = 7**

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

And again for 32: **7 – 32** won't work, so put a 0 in that place value column.

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

Try again for 16: **7 – 16** won't work, so add a 0 to that place value column.

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

Next is 8: **7 – 8** won't work. Add a 0 to the '8' place value column.

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

Try again for 4: **7 – 4 = 3**, so add a 1 to the '4' place value column.

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

Next try 2: **3 – 2 = 1**, so add a 1 to the '2' place value column.

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

And finally, 1: **1 – 1 = 0** – add a 1 to the '1' place value column.

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

This means that 199 as a binary number is 1100 0111.

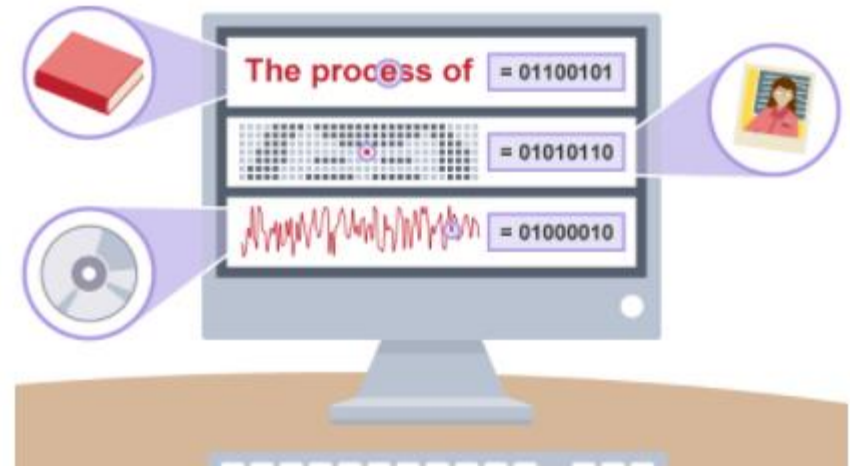
Note that binary numbers are usually written in blocks of four, separated by a space (eg 0111 1011). In denary, numbers are often written in blocks of three (eg 6 428 721).

Lesson 2

Representing text, images and sound

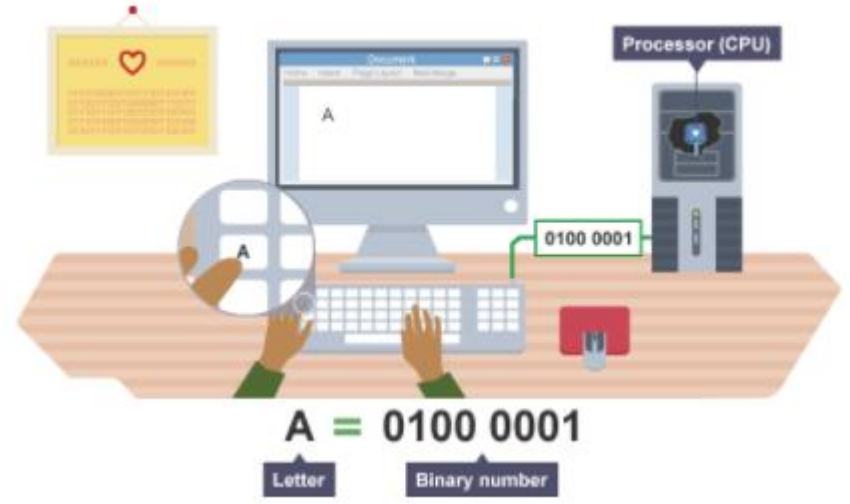
Representing data

All data inside a computer is transmitted as a series of electrical signals that are either **on** or **off**. Therefore, in order for a computer to be able to process any kind of data, including text, images and sound, they must be converted into **binary** form. If the data is not converted into binary – a series of 1s and 0s – the computer will simply not understand it or be able to process it.



Representing text

When any key on a keyboard is pressed, it needs to be converted into a binary number so that it can be processed by the computer and the typed character can appear on the screen.



A code where each number represents a character can be used to convert text into binary. One code we can use for this is called **ASCII**. The ASCII code takes each character on the keyboard and assigns it a binary number. For example:

- the letter ‘a’ has the binary number 0110 0001 (this is the denary number 97)
- the letter ‘b’ has the binary number 0110 0010 (this is the denary number 98)
- the letter ‘c’ has the binary number 0110 0011 (this is the denary number 99)

Text characters start at **denary** number 0 in the ASCII code, but this covers special characters including punctuation, the return key and control characters as well as the number keys, capital letters and lower case letters.

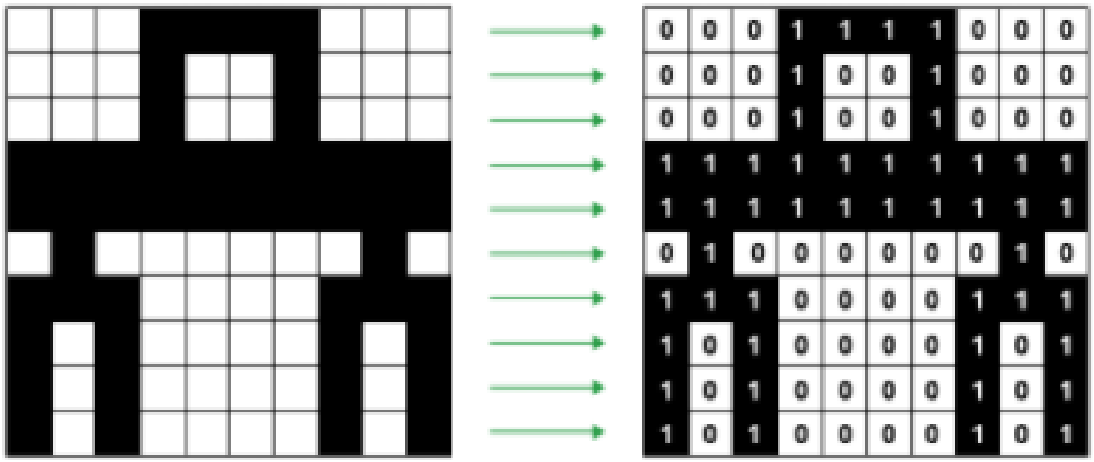
ASCII code can only store 128 characters, which is enough for most words in English but not enough for other languages. If you want to use accents in European languages or larger alphabets such as Cyrillic (the Russian alphabet) and Chinese Mandarin then more characters are needed. Therefore another code,called **Unicode**, was created. This meant that computers could be used by people using different languages.

Representing images

Images also need to be converted into **binary** in order for a computer to process them so that they can be seen on our screen. Digital images are made up of **pixels**. Each pixel in an image is made up of binary numbers.

If we say that 1 is black (or on) and 0 is white (or off), then a simple black and white picture can be created using binary.
To create the picture, a grid can be set out and the squares coloured (1 – black and 0 – white). But before the grid can be created, the size of the grid needs be known. This data is called **metadata** and computers need metadata to know the size of an image. If the metadata for the image to be created is 10x10, this means the picture will be 10 pixels across and 10 pixels down.

This example shows an image created in this way:



Adding colour

The system described so far is fine for black and white images, but most images need to use colours as well. Instead of using just 0 and 1, using four possible numbers will allow an image to use four colours. In binary this can be represented using two **bits** per pixel:

- 00 – white
- 01 – blue
- 10 – green
- 11 – red

While this is still not a very large range of colours, adding another binary digit will double the number of colours that are available:

- 1 bit per pixel (0 or 1): two possible colours
- 2 bits per pixel (00 to 11): four possible colours
- 3 bits per pixel (000 to 111): eight possible colours
- 4 bits per pixel (0000 – 1111): 16 possible colours
- …
- 16 bits per pixel (0000 0000 0000 0000 – 1111 1111 1111 1111): over 65 000 possible colours

The number of bits used to store each pixel is called the **colour depth**. Images with more colours need more pixels to store each available colour. This means that images that use lots of colours are stored in larger files.

Image quality

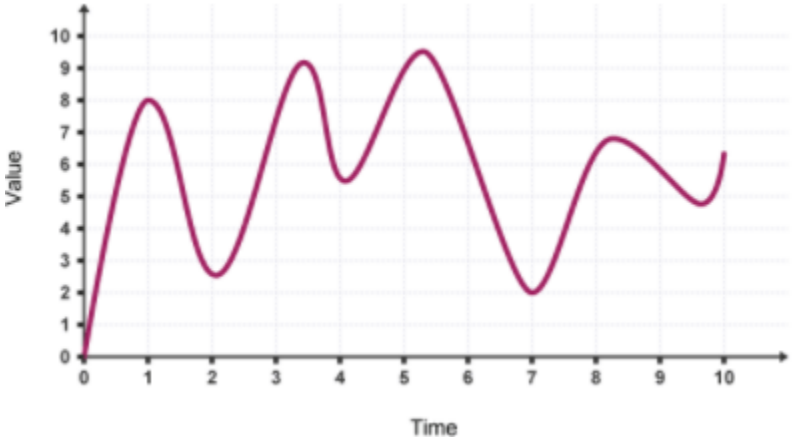
Image quality is affected by the **resolution** of the image. The resolution of an image is a way of describing how tightly packed the pixels are.

In a low-resolution image, the pixels are larger so fewer are needed to fill the space. This results in images that look blocky or **pixelated**. An image with a high resolution has more pixels, so it looks a lot better when you zoom in or stretch it. The downside of having more pixels is that the file size will be bigger.

Representing sound

Sound needs to be converted into **binary** for computers to be able to process it. To do this, sound is captured - usually by a microphone - and then converted into a **digital** signal.

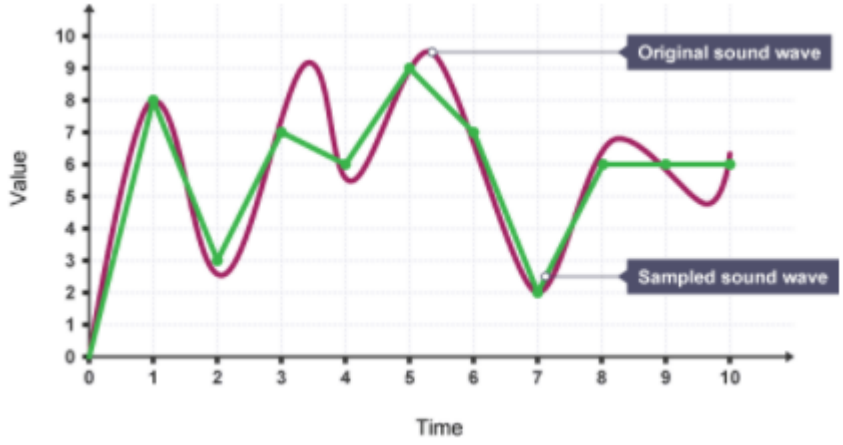
An **analogue** to digital converter will sample a sound wave at regular time intervals. For example, a sound wave like this can be sampled at each time sample point:



The samples can then be converted to binary. They will be recorded to the nearest whole number.

Time sample	1	2	3	4	5	6	7	8	9	10
Denary	8	3	7	6	9	7	2	6	6	6
Binary	1000	0011	0111	0110	1001	0111	0010	0100	0110	0110

If the time samples are then plotted back onto the same graph, it can be seen that the sound wave now looks different. This is because sampling does not take into account what the sound wave is doing in between each time sample.



This means that the sound loses quality as data has been lost between the time samples. The way to increase the quality and store the sound at a quality closer to the original, is to have more time samples that are closer together. This way, more detail about the sound can be collected, so when it's converted to digital and back to analogue again it does not lose as much quality.

The frequency at which samples are taken is called the **sample rate**, and is measured in Hertz (Hz). 1 Hz is one sample per second. Most CD-quality audio is sampled at 44 100 or 48 000 KHz.

Compression

Why compress files?

Processing power and storage space is very valuable on a computer. To get the best out of both, it can mean that we need to reduce the file size of text, image and audio **data** in order to transfer it more quickly and so that it takes up less storage space.
In addition, large files take a lot longer to **download** or **upload** which leads to web pages, songs and videos that take longer to load and play when using the internet.
Compression addresses these issues.

Any kind of data can be compressed. There are two main types of compression: lossy and lossless.

Lossy compression

Lossy compression removes some of a file's original data in order to reduce the file size. This might mean reducing the numbers of colours in an image or reducing the number of samples in a sound file. This can result in a small loss of quality of an image or sound file.
A popular lossy compression method for images is the **JPEG**, which is why most images on the internet are JPEG images. A popular lossy compression method for sounds is **MP3**. **Once a file has been compressed using lossy compression, the discarded data cannot be retrieved again**.

Lossless compression

Lossless compression doesn't reduce the quality of the file at all. No data is lost, so lossless compression allows a file to be recreated exactly as it was when originally created.
There are various algorithms for doing this, usually by looking for patterns in the data that are repeated. **Zip** files are an example of lossless compression.
The space savings of lossless compression are not as good as they are with lossy compression.

Computer Science Y7

Unit 4 Safety and Responsibility

Lesson 1

Online dangers

The internet is a fantastic resource that helps us to learn, share, communicate and find entertainment. It has billions of users who use it for legitimate reasons. However, there are others who use the internet for illegal and unsavoury purposes.

There are several dangers that we might come across when online:

- **malware**
- **phishing**
- **cyberbullying**

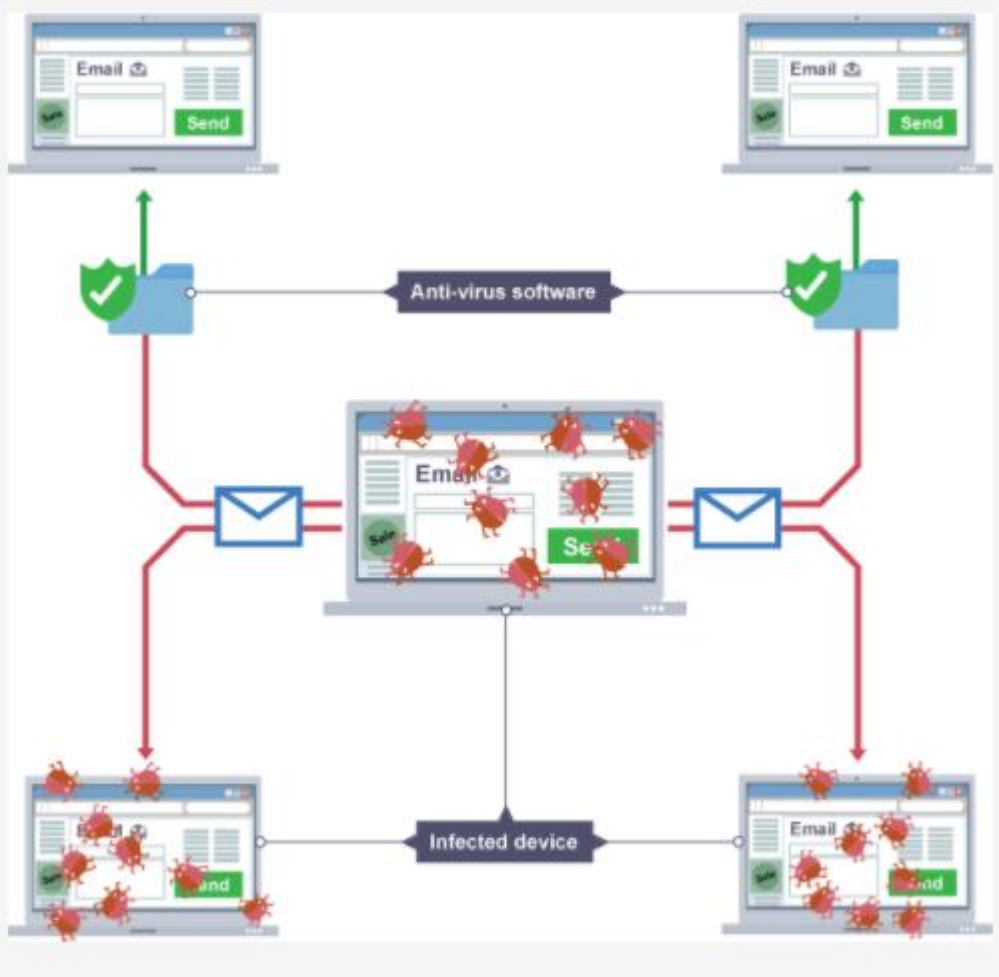
Taking simple precautions can help to reduce or prevent exposure to such dangers.

Malware

Malware (short for malicious software) are **programs** that install and run on your computer without your knowledge or consent. Malware is often downloaded from email attachments or websites that are not properly protected. Some websites are designed to trick you into **downloading** and installing malware. Once installed, malware usually (but not always) causes harm to the computer or user by deleting **data** or gaining access to personal information.

Some malware is designed to copy itself and spread to other computers via email attachments that are sent from the infected computer. This type of malware is known as a **virus**. Another type of malware spies on users' activities, usually to steal financial details and passwords. This type of malware is known as **spyware**. Other types of malware include **trojans** and **zombies**.

Once infected, it can be difficult to rid a computer of malware. Special programs called anti-virus software are required to clean malware off a computer.



Phishing

You might receive emails from someone pretending to be someone you know. The emails are designed to trick you into giving away personal information, such as your usernames and passwords. For example, an email might appear to be sent from a social media website. It might say that a password needs resetting, and might provide a link to reset it. The link would lead to a fake website which looks exactly like the real site. This site will capture your details, allowing an unsavoury character access to your accounts.

Phishing emails are often quite easy to spot. Although at first glance they may look very much like a genuine email from a company or website, on closer examination they often contain spelling or grammatical errors. Always remember that it is rare that a website will genuinely send you an email asking for your username and password.

Unsavoury characters and cyberbullying

Most people use the internet safely and responsibly, but some use it for illegal or unsavoury activities. They do it to make money or to behave badly in a situation where they think that no-one can identify them.
It is not always possible to be sure who sends emails or posts a comment to a social media account. Many people pretend to be someone else online, sometimes for fun but sometimes because they intend to harm others, through cyberbullying, theft or **trolling**.

Trolling is the term given to people who visit other peoples' social media accounts and leave distasteful messages.
Cyberbullying is the bullying of another person using the internet, mobile phones and other digital devices. Cyberbullying can take the form of posts on forums or social media, text messages or emails, all with the aim of hurting the victim.

Staying safe from malware

There are a number of ways to protect against **malware**:

- **antivirus** software
- **firewall**
- showing caution, by:
  - not opening emails from senders who we do not recognise
  - not installing programs downloaded illegally

Antivirus software protects the computer from malware such as viruses and spyware. Antivirus software scans the computer for known **malware**. If it finds malware, it safely removes it. To remain effective, antivirus software must be regularly updated so that it can recognise and remove as many forms of malware as possible.

A firewall is either a piece of hardware or software that monitors communications coming in from and going out to the internet. Both forms work on a similar basis. The firewall looks for unauthorised communications from malware. Any such communications are blocked by the firewall, preventing the malware from completing its task.

Through the course of a day you may receive emails from senders that you do not recognise. Such emails often contain malware hidden in attachments, or links to fake websites where malware can be downloaded and installed without your knowledge. **Delete such emails without opening them.**

Lots of programs and games that can be illegally **downloaded** using file-sharing tools and websites contain malware. As well as breaking the law by downloading these, you could be installing malware onto your computer.

Staying safe from phishing

Phishing emails are often quite easy to spot. If in doubt about an email, delete it immediately. **Do not follow any links contained in the email**. Instead, **go to the website directly**, and try to log in there.

Staying safe from unsavoury characters and cyberbullying

The easiest way to stay safe online is to stay in control of personal information given out.
Never disclose important details such as name, telephone number, address or school.
Never accept someone as a 'friend' on social media simply because they claim to know another friend of yours.
Always be cautious about what you say online.
Never agree to meet anyone in person that you've only known online.
If somebody does start sending you messages that offend or upset you, tell an adult that you trust.

Protection from online bullying and harassment

Cyberbullying is an extremely unpleasant and upsetting experience. There are several authorised websites that offer advice on how to stay safe online and what to do if cyberbullying occurs:

- **BBC Webwise**
- **Childline**
- **ThinkUKnow** run by the Child Exploitation and Online Protection Centre (CEOP)

The **Bullying UK** helpline is available on **0808 800 2222**, and **Childline** can be contacted on **0800 1111**.

Lesson 2

Bias and reliability

The internet contains a wealth of information. This information can be used to learn about new things or to verify facts.
However, much of the information on the internet is either biased in some way or incorrect.
Information that is biased or incorrect loses its value. When information has no value, it is of no use to us.

We need to be able to distinguish between information that is valuable (of use to us) and that which is not.

What is bias?

Biased information is information that is written from a particular perspective or point of view.

When we write, we often – either purposefully or accidentally – introduce bias. Information that contains bias may be:

- personal opinion
- a statement that has no factual basis
- **prejudiced** in favour of or against a person, product, situation or idea

Examples

Look at the following examples of information about a film:

- "I think this film is the best animated film of all time." This statement is clearly personal opinion, and as such should be treated with caution. Someone else might say the film is poor.
- "In twenty years' time, people will say this film is the best animated film ever." There is no factual basis to this statement. How can the person who wrote it know what people will think in the future?
- "Like all animated films, this one is great!" This information contains prejudice – the writer clearly has a passion for animation. Someone who does not like animation may say all animated films are poor.

In each case, bias has distorted the information about the film.

What is reliability?

Incorrect information is information that is wrong, out of date or inaccurate.

Websites may contain information that is incorrect for any of these reasons:

- **wrong –** the facts stated are incorrect
- **out of date –** the facts may have been correct when the website was produced, but are no longer correct
- **inaccurate –** the facts may be largely correct, but may contain some errors

When information is correct, it is 'reliable'. Reliable information has value. The less reliable the information, the less valuable it is.

Recognising bias and unreliability

Biased information also loses its value. Information of little value may:

- mislead us
- misinform us
- cause us to make an incorrect deduction
- cause us to make a poor decision

Suppose we used the internet to research the health benefits of cleaning our teeth. One website, owned and produced by a dental company, might tell us that we need to clean our teeth ten times every day. Another website, written by an individual, might state that cleaning our teeth is a waste of time.

By following the advice from the first one, we might spend more money than we need to on teeth-cleaning products and damage our teeth by cleaning them too much. By following the advice from the other, we might suffer from poor dental hygiene.

Factors to consider

Biased information is influenced by a point of view. When analysing information for bias, there are certain factors to look for:

- **Source – who has produced the information?** Information from an authoritative, well-known organisation or person is likely to have value. Information from **wikis** and **blogs** may be less valuable because they are not authoritative – anyone can update a wiki or write a blog. As such, they may contain bias or inaccuracies. Remember, though, that a company may overstate claims about their products or services, whilst understating those of their competitors.

- **Opinion or fact – does the website state facts or opinions?** Opinions are points of view, not facts. Whilst opinions should be considered and may be interesting, as information they have less value than facts.

- **Statements without facts – does the website contain statements that cannot be backed up by facts?** Such statements are opinions, and have little value.

- **Date of publication – when was the website was last updated?** Websites that have not been updated for a long time may no longer be accurate.

Lesson 3

The law and ethics

Computers and the law

Computers are fantastic - they help us to learn, share, communicate and find entertainment. However, it is also possible for computers to be used to aid illegal activities. An understanding of computer-related laws in the United Kingdom is needed to make sure we stay on the right side of the law.

Computers might be used unlawfully in many ways, for example:

- allowing someone to illegally share your personal data
- helping to steal financial information, such as credit card numbers or bank account details
- helping to illegally copy and distribute films, television programmes and music
- extorting information or blackmailing someone

Additionally, the internet allows people to commit crimes remotely, for example a hacker could gain access to a computer on the other side of the world. Laws are required to help deter such activities.

There are three laws to consider:

- **Computer Misuse Act**
- **Copyright, Designs and Patents Act**
- **Data Protection Act**

Computer Misuse Act

The **Computer Misuse Act** attempts to discourage people from using computers for illegal purposes. There are three separate parts to the Act:

1. It is illegal to access data stored on a computer unless you have permission to do so. Unauthorised access is often referred to as **hacking**.

2. It is illegal to access data on a computer when that material will be used to commit further illegal activity, such as fraud or blackmail.

3. It is illegal to make changes to any data stored on a computer when you do not have permission to do so. If you access and change the contents of someone's files without their permission, you are breaking the law. This includes installing a **virus** or other **malware** which damages or changes the way the computer works.

The maximum punishment for breaking this law is a £5000 fine or several years' imprisonment.

However, one key part of the law is that **intent must be proved**. If a computer is not well protected, someone could accidentally access its data without meaning to. Someone might also accidentally change a document without realising it. For anyone to be found guilty, it has to be shown that they intentionally accessed and changed data.

Copyright, Designs and Patents Act

The **Copyright, Designs and Patents Act** exists to protect our creations.

When anyone creates something, they own it. What they create might include:

- a picture, drawing or photograph
- a video, television programme or film
- text, such as a book, article or report
- a game

Copyright is a legal means of ensuring that content creators can protect what they create. **Copyright is applied automatically - it is not necessary to register copyright or to use a © symbol.** Work is automatically protected by copyright unless the copyright holder chooses to give that right away.

Copyright gives the copyright holder exclusive rights to publish, copy, distribute and sell their creation. No one else can use the work without permission. Copyright on a piece of work lasts for a long time, although the rules about how long are quite complicated and vary from country to country.
When you buy something, such as a book, film or music CD, the copyright holder grants permission for you to use it as part of the sale. This is called a **licence**. The licence is generally only for you to use.

When using computers, unless you have permission with regard to a particular copyrighted material, it is illegal to:

- make copies
- publish
- distribute
- sell copies

This applies to any copyrighted material, such as music, films, games and television programmes. The internet has made it extremely easy to access copyrighted material illegally. **If you download a music track, film, game or programme without the copyright holder's permission, you are breaking the law.**

Supermarkets earn their money by selling food and other products. If someone takes their products without paying, the supermarket doesn't make any money. In the same way, musicians, photographers, film makers and artists earn their money by selling products. **If someone takes their products without paying, the person who created the work doesn't make any money.**

There are some situations where it is legal to copy, publish, distribute or sell material. These are:

- when you are the copyright holder
- when you have the copyright holder's permission
- when the copyright holder has chosen to give up their copyright

Data Protection Act

It is increasingly common for **personal details** to be stored on computers. The **Data Protection Act** exists to protect such details. This personal data includes items such as:

- name and address
- date of birth
- medical records
- school and employment records
- religion

Personal data is private and should only be accessible by authorised people. Also, digital files stored on computers can be easy to access, copy and share. Protection is needed to make sure that our personal data is kept private and not altered or deleted. **The Data Protection Act exists to ensure our data is properly looked after**.
In addition, **everyone has the right to see what data is held about them by an organisation, and to have that data corrected or deleted if incorrect**.