

# Computational Thinking

How do we think about problems so that  
computers can help?

# Computational Thinking

## HOW DO WE THINK ABOUT PROBLEMS SO THAT COMPUTERS CAN HELP?

Computers are incredible devices: they extend what we can do with our brains. With them, we can do things faster, keep track of vast amounts of information and share our ideas with other people.

### What is computational thinking?

Getting computers to help us to solve problems is a two-step process:

1. First, we think about the steps needed to solve a problem.
2. Then, we use our technical skills to get the computer working on the problem.

Take something as simple as using a calculator to solve a problem in maths. First, you have to understand and interpret the problem before the calculator can help out with the arithmetic bit.

Similarly, if you're going to make a presentation, you need to start by planning what you are going to say and how you'll organise it before you can use computer hardware and software to put a deck of slides together.

In both of these examples, the **thinking** that is undertaken before starting work on a computer is known as computational thinking.

Computational thinking describes the processes and approaches we draw on when thinking about problems or systems in such a way that a computer can help us with these. Jeanette Wing puts it well:

**Computational Thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent. (Wing, 2010)**

The apparently clumsy term 'information-processing agent' is there because Wing wants us to understand that it's not just computers that can execute algorithms. People can (following instructions to make a cake), bees can (finding the shortest path to nectar), termites can (building a mound), and cells can (DNA is the program that cells execute).

Computational thinking is not thinking **about** computers or **like** computers. Computers don't think for themselves. Not yet, at least!

When we do computational thinking, we use the following concepts to tackle a problem:

- logical reasoning: predicting, analysing and explaining (see pages 8 – 12);
- algorithms: making steps and rules (see pages 12 – 20);
- decomposition: breaking problems or systems down into parts (see pages 20 – 23);
- abstraction: managing complexity, sometimes through removing unnecessary detail (see pages 23 – 27);
- generalisation:<sup>(1)</sup> spotting and using patterns and similarities (see pages 27 – 30);
- evaluation: making judgements (see pages 30 – 33).

This is not an exhaustive list, but it gives some helpful structure to the umbrella term of 'computational thinking'. Here is a picture that may help to clarify (Figure 1.1):

<sup>1</sup> The Barefoot Computing developed by Computing At School (CAS) for primary teachers refers to 'patterns' here, but the term 'generalisation' was used in CAS's computational thinking working group (see Czismadia et al., 2015).

# The Computational Thinkers

**concepts**

- Logic**  
Predicting & analysing
- Evaluation**  
Making judgements
- Algorithms**  
Making steps & rules
- Patterns**  
Spotting & using similarities
- Decomposition**  
Breaking down into parts
- Abstraction**  
Removing unnecessary detail

**approaches**

- Tinkering**  
Changing things to see what happens
- Creating**  
Designing & making
- Debugging**  
Finding & fixing errors
- Persevering**  
Keeping going
- Collaborating**  
Working together

We're all computational thinkers here!

When you think about it, whether we're parents, pupils or teachers - we're all natural computer scientists, capable of computational thinking.  
Our brains, like computers, process, debug and make simple algorithms every day!



Figure 1.1

Barefoot would like to acknowledge the work of Julia Briggs and the eLIM team at Somerset County Council for their contribution to this poster and Group DRP for their work on the design.

## What can you do with computational thinking?

Although computational thinking describes the sort of thinking that computer scientists and software developers engage in, plenty of other people think in this way too, and not just when it comes to using computers. The thinking processes and approaches that help with computing are really useful in many other domains too.

For example, the way a team of software engineers go about creating a new computer game, video editor or social networking platform is really not that different from how you and your colleagues might work together to plan a scheme of work or to organise an educational visit.

In each case:

- you think about the problem – it's not just trying things out and hoping for the best
- you take a complex problem and break it down into smaller problems;

- it's necessary to work out the steps or rules for getting things done;
- the complexity of the task needs to be managed, typically by focusing on the key details;
- the way previous projects have been accomplished can help;
- it's a good idea to take stock at the end to consider how effective your approach has been.

## How is computational thinking used in the curriculum?

Ideas like logical reasoning, step-by-step approaches (algorithms), decomposition, abstraction, generalisation and evaluation have wide applications for solving problems and understanding systems across (and beyond) the school curriculum. As pupils learn to use these approaches in their computing work, you and your colleagues should find that they become better at applying them to other work too.

During their time at primary school, pupils will already have used lots of aspects of computational thinking, and will continue to do so across the

curriculum in secondary education. It's worth making these connections explicit during computing lessons, drawing on the applications of computational thinking that your students will already be familiar with, as well as discussing these ideas with your colleagues teaching other subjects. For example:

- In English, students are encouraged to plan their writing, to think about the main events and identify the settings and the characters.
- In art, music or design and technology, students think about what they are going to create and how they will work through the steps necessary for this, by breaking down a complex process into a number of planned phases.
- In maths, pupils will identify the key information in a problem before they go on to solve it.

## Where does computational thinking fit in the new computing curriculum?

The national curriculum for computing puts computational thinking right at the heart of its ambition. It states:

**A high-quality computing education equips pupils to use computational thinking and creativity to understand and change the world. (Department for Education [DfE], 2013)**

Indeed, computational thinking remains part of the statutory curriculum entitlement for all up to the end of Key Stage 4:

**All pupils should be taught to ... develop and apply their analytic, problem-solving, design, and computational thinking skills. (DfE, 2013)**

Whilst programming (see pages 30 – 38) is an important part of the new curriculum, it would be wrong to see this as an end in itself. Rather, it is through the practical experience of programming that the insights of computational thinking can best be developed and exercised. Not all students will go on to get jobs in the software industry or make use of their programming in academic studies, but all **are** likely to find ways to apply and develop their computational thinking.

Computational thinking should not be seen as just a new name for 'problem-solving skills'. A key element of computational thinking is that it helps us make better use of computers in solving problems and understanding systems. It does help to solve problems and it has wide applications across other disciplines, but it is most obviously apparent, and probably most effectively learned, through the rigorous, creative processes of writing code – as discussed in the next section.



### Classroom activity ideas

- Traditional IT activities can be tackled from a computational thinking perspective. For example, getting students to create a short video presentation might begin by breaking the project down into short tasks (decomposition), thinking carefully about the best order in which to tackle these and drawing up a storyboard for the video (algorithms), learning about standard techniques in filming and editing and recognising how others' work could be used as a basis or even included here (generalisation), learning about, but not being overly concerned about technical elements of cameras and file formats (abstraction).
- If your school has cross-curricular projects or theme days, see if you can adopt a computational thinking approach to one of these. Take, as an example, putting on an end of year play: students could break the project down into a set of sub tasks, consider the order in which these need to be accomplished, assign tasks to individuals or groups and review how work is progressing towards the final outcome.
- There are strong links between computational thinking and the design–make–evaluate approach that's common in design and technology, and sometimes in other subjects.



### Further resources

Barba, L. (2016) *Computational thinking: I do not think it means what you think it means*. Available from <http://lorenabarba.com/blog/computational-thinking-i-do-not-think-it-means-what-you-think-it-means/>



Barefoot Computing (n.d.) *Computational thinking*. Available from <http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/computational-thinking/> (free, but registration required).

BBC Bitesize (n.d.) *Introduction to computational thinking*. Available from [www.bbc.co.uk/education/guides/zp92mp3/revision](http://www.bbc.co.uk/education/guides/zp92mp3/revision)

Berry, M. (2014) *Computational thinking in primary schools*. Available from <http://milesberry.net/2014/03/computational-thinking-in-primary-schools/>

Computer Science Teachers Association (n.d.) CSTA *computational thinking task force/Computational thinking resources*. Available from <http://csta.acm.org/Curriculum/sub/CompThinking.html>

Computing At School (n.d.) *Computational thinking*. Available from <http://community.computingschool.org.uk/resources/252>

Curzon, P., Dorling, M., Ng, T., et al. (2014) *Developing computational thinking in the classroom: A framework*. *Computing At School*. Available from <http://community.computingschool.org.uk/files/3517/original.pdf>

Google for Education (n.d.) *Exploring computational thinking*. Available from [www.google.com/edu/computational-thinking/index.html](http://www.google.com/edu/computational-thinking/index.html)

Google's MOOC on *Computational Thinking for Educators* (n.d.) Available from <https://computationalthinkingcourse.withgoogle.com/unit>

Harvard Graduate School of Education (n.d.) *Computational thinking with Scratch*. Available from <http://scratched.gse.harvard.edu/ct/defining.html>

Pólya, G. (1945) *How to solve it*. Princeton, NJ: Princeton University Press.

Selby, C. and Woollard, J. (2013) *Computational thinking: The developing definition*. University of Southampton. Available from <http://eprints.soton.ac.uk/356481/>

Wing, J.M., 2008. Computational thinking and thinking about computing. *Philosophical transactions of the royal society of London A: mathematical, physical and engineering sciences*, 366(1881), pp.3717-3725.

# Logical Reasoning

Can you explain why something happens?

At its heart, logical reasoning is about being able to explain why something is the way it is. It's also a way to work out why something isn't quite as it should be.

If you set up two computers in the same way, give them the same instructions (the program) and the same input, you can pretty much guarantee the same output. Computers don't make things up as they go along or work differently depending on how they happen to be feeling at the time. This means that they are **predictable**. Because of this we can use logical reasoning to work out exactly what a program or computer system will do.

It's well worth doing this in school: as well as **writing** and **modifying** programs, have pupils **read** programs that you give them; get them to **explain** a program to another student; encourage them to **predict** what their own or others' programs will do when given different test data; when their program doesn't work, encourage them to **form a hypothesis** of what is going wrong, and then **devise a test** that will confirm or refute that hypothesis; and so on. All of these involve logical reasoning.

At a basic level, pupils will draw on their previous experience of computing when making predictions about how a computer will behave, or what a program will do when run. They have some model of computation, a 'notional machine' (Sorva, 2013) which may be more or less accurate: one of our tasks as computing teachers is to develop and refine pupils' notional machines. This process of using existing knowledge of a system to make reliable predictions about its future behaviour is one part of logical reasoning.

As far back as Aristotle (1989; qv Chapter 22 of Russell, 1946), rules of logical inference were defined; these were expressed as syllogisms, such as:

- All men are mortal.
- Socrates is a man.
- Therefore Socrates is mortal.

This approach to logic, as an early sort of computational thinking, formed part of the trivium of classical and medieval education: it provides a grounding in the mechanics of thought and analysis, which is as relevant to education now as then.

## Boolean logic

Irish mathematician George Boole (2003) took the laws of logic a step further in the 19th century, taking them out of the realm of philosophy and locating them firmly inside mathematics. Boole saw himself as going under, over and beyond Aristotle's logic, by providing a mathematical foundation to logic, extending the range of problems that could be treated logically and increasing the number of propositions that could be considered to arbitrarily many.

Boole's system, subsequently named 'Boolean logic' after him, works with statements that are either true or false, and then considers how such statements might be combined using simple operators, most commonly AND, OR and NOT. In Boolean logic:

- (A AND B) is true if both statement A is true and statement B is true, otherwise it's false;
- A OR B is true if A is true, if B is true or if both A and B are true;
- NOT A is true if A is false, and vice versa.

Boole went on to establish the principles, rules and theorems for doing something very much like algebra with logical statements rather than numbers. It's a powerful way of analysing ideas, and worth some background reading; it's a mark of Boole's success that this system of logic remains in use today and lies at the heart of computer science and central processing unit (CPU) design.

One way of visualising Boole's operators is the combination of sets on Venn diagrams, where the members of the set are those that satisfy the conditions A or B.

X AND Y – the intersection of the two sets is where both condition X and condition Y are satisfied (Figure 1.2):

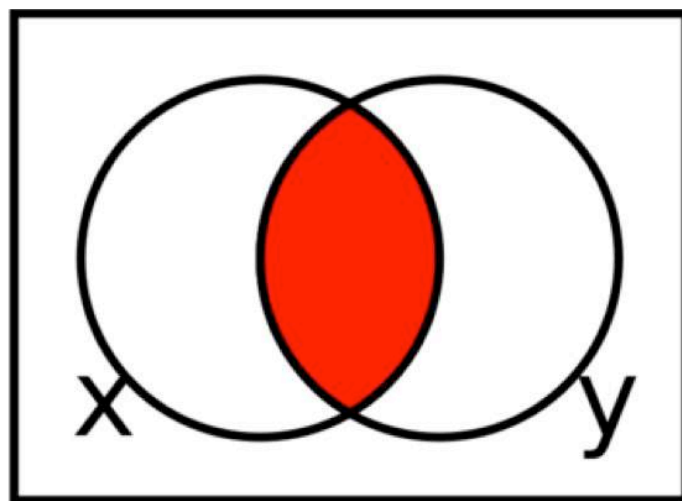


Figure 1.2

X OR Y – the union of the two sets is where either condition X or condition Y (or both) are satisfied (Figure 1.3):

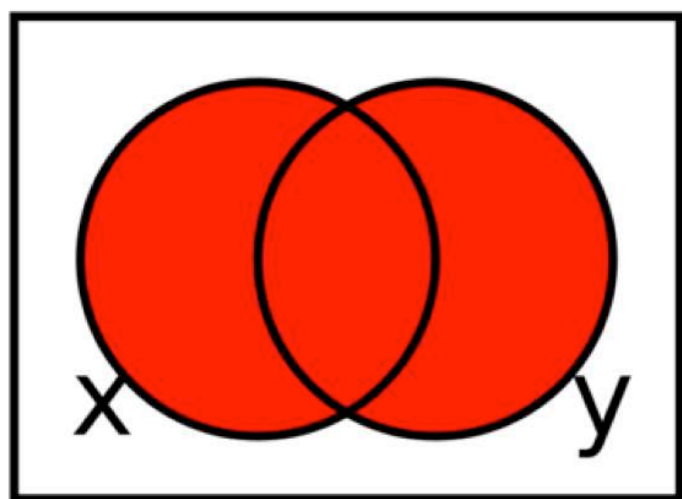


Figure 1.3

NOT X – the complement of the set, that is those elements that don't satisfy the condition X (Figure 1.4).

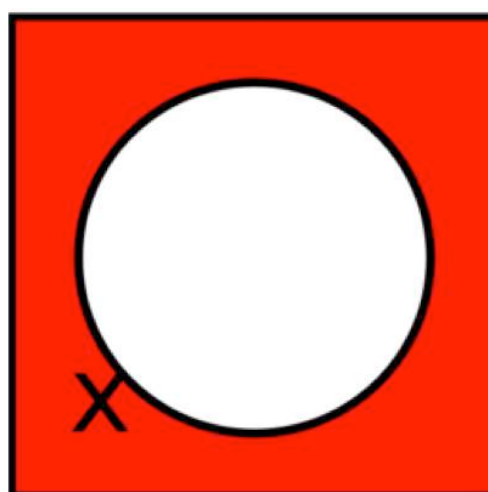
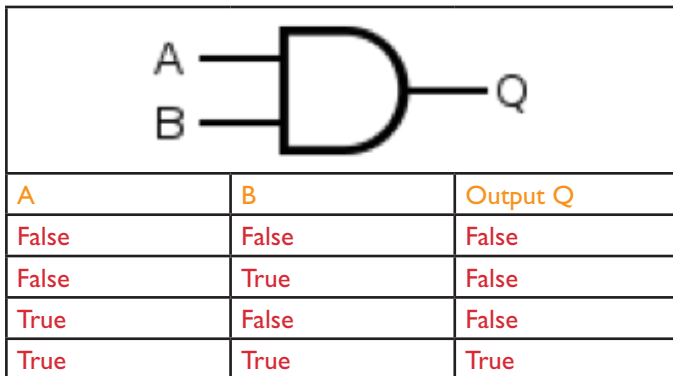


Figure 1.4

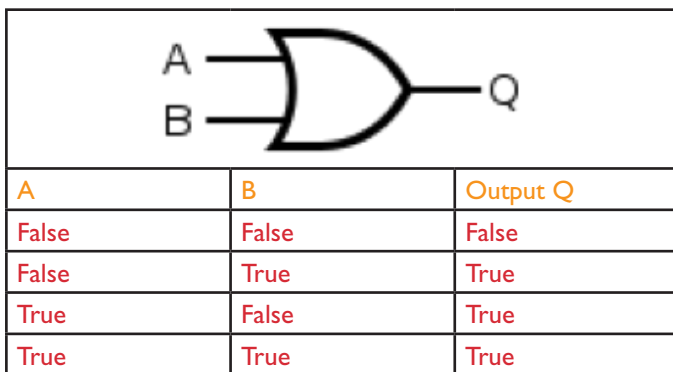
Sometimes we use Boolean operators like this when refining search results, for example results for ('Computing' OR 'computer science') AND 'CPD' AND (NOT ('primary' OR 'elementary')), although to be fair this sophistication is seldom needed these days – check out Google or Bing's advanced search pages to see how something similar to Boolean operators is used in modern search engines.

In computing, we can think of Boole's operators as gates, producing output depending on the inputs they are given – at a simple level, current flows through the gate depending on whether voltage is applied at the inputs, or the gate produces a binary 1 as output depending on the binary values provided at the inputs (see pages xx–yy). There are standard symbols for the different gates, making it easy to draw diagrams showing how systems of gates can be connected. We can use **truth tables** to show the relationship between the inputs and the outputs. These are the logic equivalents to times tables, listing all the possible inputs to a gate, or a system of gates, and the corresponding outputs.

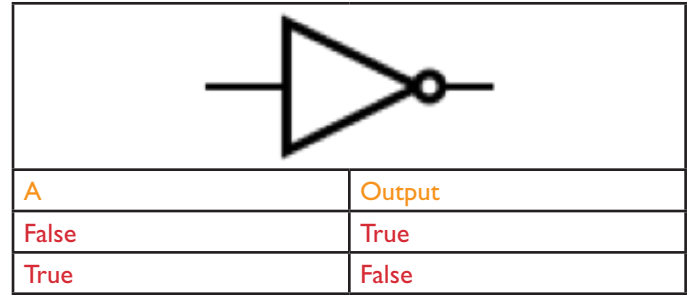
AND:



OR:



NOT:



It is possible to wire up simple electrical circuits to model the behaviour of these logic gates, using nothing more sophisticated than batteries, bulbs and switches: two switches in series model the behaviour of an AND gate, in parallel an OR gate, and a switch in parallel with a bulb would behave as a NOT gate, shorting the circuit when closed.

Boolean operators are also part of most programming languages, including Scratch, Python, Javascript and Small BASIC. They are used in conjunction with selection (if... then... else...) statements to control the flow of a program's execution. It is in this context that pupils are initially likely to encounter and make use of them. For example, game programming might make use of selection statements such as:

if (have candle) and (have match) then (show room)

Or

if (time remaining < 0) or (health < 0) then (game over)

When used in combination with binary representation, logical operators can be applied to each bit in binary numbers – we call these 'bitwise' operators – and this can be in quickly isolating particular parts of a byte or word; for example to get just the two least significant bits of a byte, AND the byte with 0b00000011.

How is logical reasoning used in computing?

Logic is fundamental to how computers work: deep inside the computer's CPU, every operation the computer performs is reduced to logical operations carried out on binary digits, using electrical signals. We return to these ideas in the section on technology. Operations at CPU level from binary

addition upwards can be carried out by digital circuits made from just combining AND, OR and NOT gates.

It is because everything a computer does is controlled by logic that we can use logic to reason about program behaviour.

Software engineers use logical reasoning all the time in their work. They draw on their internal mental models of how computer hardware, the operating system (such as Windows 10, OS X) and the programming language they are using all work, in order to develop new code that will work as they intend. They will also rely on logical reasoning when testing new software and when searching for and fixing the ‘bugs’ (mistakes) in their thinking (known as debugging – see page 35 & 77) or their coding when these tests fail.

Boolean operators are useful in many contexts beyond the digital circuits controlling CPUs, including refining sets of search results and controlling how programs operate.

There is a long history of getting computers to work with logic at a more abstract level, with these ideas laying the foundation for early work in artificial intelligence. The programming language Prolog is perhaps the best known and most widely used logical programming language and has applications in theorem proving, expert systems and natural language processing.

## Where does logical reasoning fit in the computing curriculum?

At primary school, Key Stage 1 pupils are expected to use logical reasoning to predict the behaviour of simple programs. This can include the ones they themselves write, but it might also include predicting what happens when they play a computer game or use a painting program. At Key Stage 2, pupils are expected to ‘use logical reasoning to explain how some simple algorithms work and to detect and correct errors in algorithms and programs’.

At Key Stage 3, pupils continue to use logical reasoning when thinking about programs, including ‘to compare the utility of alternative algorithms for the same problem’. They also encounter

Boolean logic, although some will have had a little experience of using logical operators in Scratch or other programming at primary school. They should ‘understand simple Boolean logic [for example, AND, OR and NOT] and some of its uses in circuits and programming’.

Logic has played a part in the school curriculum from classical times onwards, and your colleagues can do much to develop and build on pupils’ logical reasoning. For example, in science pupils should explain how they have arrived at their conclusions from the results of their experiments; in mathematics, they reason logically to deduce properties of numbers, relationships or geometric figures; in English, citizenship or history they might look for logical flaws in their own or others’ arguments.



### Classroom activity ideas

- Give pupils programs in Scratch, Python or other programming languages and ask them to explain what the program will do when run. Being able to give a reason for their thinking is what using logical reasoning is all about. Include some programs using logical operators in selection statements.
- In their own coding, logical reasoning is key to debugging (finding and fixing the mistakes in their programs). Ask the pupils to look at one another’s Scratch or Python programs and spot bugs, without running the code. Encourage them to test the programs to see if they can isolate exactly which bit of code is causing a problem. If pupils’ programs fail to work, get them to explain their code to a friend or even an inanimate object (for example, a rubber duck).
- Provide an opportunity for pupils to experiment with logic at a circuit level, perhaps using simple bulb/switch models or using LEDs and logic gates on simple integrated circuits on a breadboard.
- Encourage pupils to experiment with the advanced search pages in Google or Bing, perhaps also expressing their search query using Boolean operators.



- Pupils should make use of logic operators in selection statements when programming – game programs, for example text based adventures, provide many opportunities for this. Pupils could also experiment with bitwise logical operators in Python or TouchDevelop, or create blocks for other logical operators such as NAND and XOR in Snap!
- Ask pupils to think carefully about some school rules, for example those in the school's computer Acceptable Use Policy. Can they use logical reasoning to explain why the rules are as they are? Can they spot any logical contradictions in the policy?
- There are many games, both computer-based and more traditional, that draw directly on the ability to make logical predictions. Organise for the pupils to play noughts and crosses, Nim or chess. As they are playing, ask them to predict their opponent's next move. Let them play computer games such as Minesweeper or SimCity, as appropriate. Ask them to pause at certain points and tell you what they think will happen when they move next. Consider starting a chess club if your school doesn't already have one.

Cryan, D., Shatil, S. and Mayblin, B. (2008) *Introducing logic: A graphic guide*. London: Icon Books Ltd.

McInerny, D. (2005) *Being logical: A guide to good thinking*. New York, NY: Random House.

McOwan, P. and Curzon, P. (n.d.) *Brain-in-a-bag: Creating an artificial brain*. Available from [www.cs4fn.org/teachers/activities/braininabag/braininabag.pdf](http://www.cs4fn.org/teachers/activities/braininabag/braininabag.pdf)

*The P4C Co-operative* (n.d.) A co-operative providing resources and advice on philosophy for children. Available from [www.p4c.com/](http://www.p4c.com/)

*PhiloComp.net* (n.d.) Website highlighting the strong links between philosophy and computing. Available from [www.philocomp.net/](http://www.philocomp.net/)

## Algorithms

What is the best way to solve a problem?

An algorithm is a sequence of instructions or a set of rules to get something done.

You probably know the fastest route from school to home, for example, turn left, drive for five miles, turn right. You can think of this as an 'algorithm' – as a sequence of instructions to get you to your chosen destination. There are plenty of routes that will accomplish the same goal, but some are better (that is, shorter or faster) than others.

Indeed, we could think of strategies (that is, algorithms) for finding a good route, such as might be programmed into a sat nav. For example, taking a random decision at each junction is unlikely to be particularly efficient, but it will (perhaps centuries later) get you to school.

One approach to this problem could be to list all the possible routes between home and destination and simply choose the fastest. This isn't likely to be a particularly fast algorithm, as there are many, many possible routes (such as via Edinburgh, or round the M25 a few times), many of which can be immediately ignored.

Another algorithm would be to take the road closest to the direction you are heading; this will do a bit better, but might involve some dead ends and lots of narrow lanes.

### Further resources

Barefoot Computing (n.d.) *Logic: Predicting and Analysing*. Available from <http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/logic/> (free, registration required).

Boole, G. (1853) *An investigation of the rules of thought*. Mineola, NY: Dover Publications.

Bragg, M., Grayling, A. C., Millican, P., Keefe, R., (2010) *Logic*. BBC Radio 4 In Our Time. Available from [www.bbc.co.uk/programmes/b00vcqcx](http://www.bbc.co.uk/programmes/b00vcqcx)

Carroll, L. (1896) *Symbolic logic and the game of logic*. Mineola, NY: Dover Publications.

Cliff, D. (2013) *The joy of logic* (for BBC Four). Vimeo. Available from <https://vimeo.com/137147126>

Computer Science for Fun (n.d.) *The magic of computer science*. Available from [www.cs4fn.org/magic/](http://www.cs4fn.org/magic/)

Computer Science Unplugged (n.d.) *Databases unplugged*. Available from <http://csunplugged.org/databases>

Dijkstra's algorithm does a much better job of finding the shortest route, and, subject to some reasonable assumptions, there are even faster algorithms now.<sup>(2)</sup>

It's worth noting a couple of things here: calculating the shortest routes quickly comes not through throwing more computing power at the problem (although that helps) but through thinking about a better solution to the problem, and this lies at the heart of computer science; also Dijkstra's algorithm (and others like it) isn't just about finding the shortest route for one particular journey, it allows us to find the shortest path in any network (strictly, a graph), whether it be places on a road network, underground stations, states of a Rubik's Cube or film stars who have acted alongside one another.<sup>(3)</sup>

## How algorithms are expressed

There is sometimes confusion between what an algorithm is and the form in which it is expressed. Algorithms are written for people to follow rather than computers, but even so there is a need for a degree of precision and clarity in how algorithms are expressed. It is quite good enough to write the steps or rules of an algorithm out as sentences, as long as the meaning is clear and unambiguous, but some teachers find it helpful to have the steps of an algorithm written as a flowchart, such as in Figure 1.5:

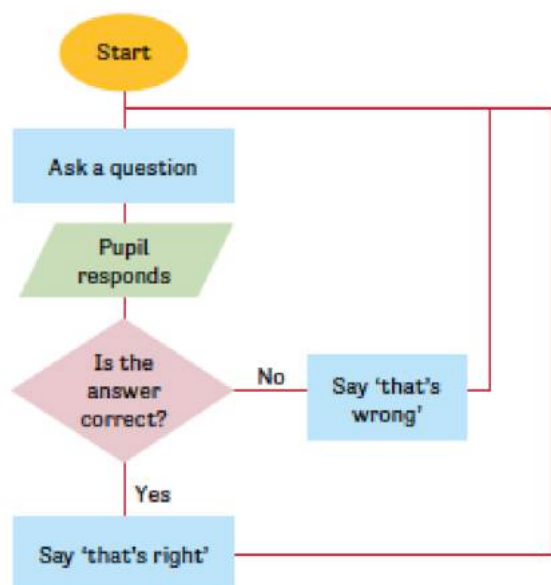


Figure 1.5 An example of a flow chart

Or algorithms can be expressed in pseudocode, which is perhaps best thought of as a half-way

house between human language and a programming language, borrowing many of the characteristics of the latter, whilst allowing some details to be left as implicit:

Repeat 10 times:

Ask a maths question

If the answer is right then:

Say well done!

Else:

Say think again!

At least one awarding organisation has developed its own more formal pseudocode for GCSE and A Level, allowing programming questions to be set and answered without preferring one programming language to another.

## How are algorithms used in the real world?

There are plenty of day-to-day situations in which life can be made a little simpler or more efficient through algorithmic thinking, from deciding where to park to finding socks in a drawer (Christian and Griffiths, 2016).

Search engines such as Bing or Google use algorithms to put a set of search results into order, so that more often than not, the result we are looking for is at the top of the front page (Brin and Page, 1998; qv Peng and Dabek, n.d.).

Your Facebook news feed is derived from your friends' status updates and other activity, but it only shows that activity which the algorithm (EdgeRank) thinks you will be most interested in seeing. The recommendations you get from Amazon, Netflix and eBay are algorithmically generated, based in part on what other people are interested in. There are even algorithms which can predict whether a movie or song will be a hit.

Credit ratings, interest rates and mortgage decisions are made by algorithms. Algorithmic trading now accounts for large volumes of the stocks and other financial instruments traded on the world's markets, including by pension funds.

<sup>2</sup> See [https://en.wikipedia.org/wiki/Shortest\\_path\\_problem](https://en.wikipedia.org/wiki/Shortest_path_problem)

<sup>3</sup> See <https://oracleofbacon.org/how.php>

Given the extent to which so much of their lives is affected by algorithms, it is worth pupils having some grasp of what an algorithm is.

## Where do algorithms fit in the computing curriculum?

At primary school, the computing curriculum expects pupils in Key Stage 1 to have an understanding of what algorithms are, and how they are used in programs on digital devices. Often young pupils will be introduced to the idea of an algorithm away from computers, in ‘unplugged’ classroom contexts. Pupils will go on to recognise that the same algorithm can be implemented as code in different languages and on different systems, from Bee Bots to Scratch Jr (Figure 1.6).



Figure 1.6 Scratch Jr programming for drawing a square

Key Stage 2 builds on this: pupils are expected to design programs with particular goals in mind, which will draw on their being able to think algorithmically, as well as using logical reasoning to explain algorithms and to detect and correct errors in them. Sometimes this will be through acting out or writing down what happens when an algorithm is followed rather than always through writing a program in Scratch to implement it.

In Key Stage 3, the national curriculum requirements include that pupils should ‘understand several key algorithms that reflect computational thinking [for example, ones for sorting and searching]’. There’s also the expectation that they ‘use logical reasoning to compare the utility of alternative algorithms for the same problem’, which is covered in the evaluation section on [page 30](#).

In developing your pupils’ understanding of key algorithms, you might start with ‘unplugged’ approaches, in which pupils get a good feel for how an algorithm operates, for example by playing ‘guess the number’ games or by sorting a set of masses into order using a pan balance. It can also be very useful for pupils to work through the steps of an algorithm, perhaps expressed as flowcharts or pseudocode, for themselves using pencil and paper. This was the approach advocated by Donald Knuth (1997) in *The art of computer programming*, when he wrote that:

**An algorithm must be seen to be believed, and the best way to learn what an algorithm is all about is to try it. The reader should always take pencil and paper and work through an example of each algorithm immediately upon encountering it.**

Don’t shy away from having pupils implement algorithms as code, in whatever programming language they are most familiar with. They’re likely to understand the algorithm better and to be able to recall it more clearly if they have to work out for themselves how it can be implemented and debug any mistakes that they make. It is also important that pupils maintain good habits of thinking about the algorithms they want to use before writing code to implement them. Think of the coding as like doing an experiment in science.

## Sorting and searching

The programme of study talks of ‘key algorithms’, particularly for search and sort, so we will look at some of these now. Why are sorting and searching distinguished in this way? For three reasons. First, it is particularly easy to explain what sorting and searching algorithms do and to explain why they are useful. Second, they are ubiquitous inside computer systems so knowledge of sorting and searching algorithms is particularly useful. Third, there are an astonishingly large number of algorithms known for sorting and searching, each useful in different circumstances. People write entire books about them!

## Search

Imagine trying to find the definition of a word in a dictionary – we have a number of possible approaches:

- we could pick a word at random, check if it's the right word, and repeat this until we get the one we want; or
- we could start at the beginning, checking each word in turn until we get to the word we want; or
- we could pick a word in the middle, decide whether our word is before or after that, pick a word in the middle of that half, and keep narrowing down until we get to the word we want.

These three approaches provide three possible algorithms for search, that is for finding a particular item in a list of data: a random search, a linear search and binary search.<sup>(4)</sup>

You can demonstrate these in class by taking three different approaches to a 'guess my number' game, thinking of a number and asking the class to use one or other of these algorithms to work out what it is (Figures 1.7–1.9):

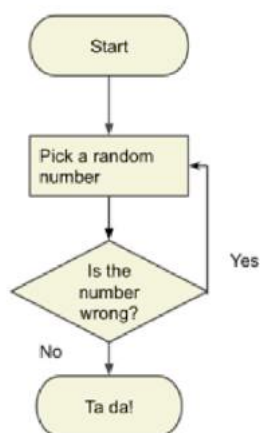


Figure 1.7 Random search

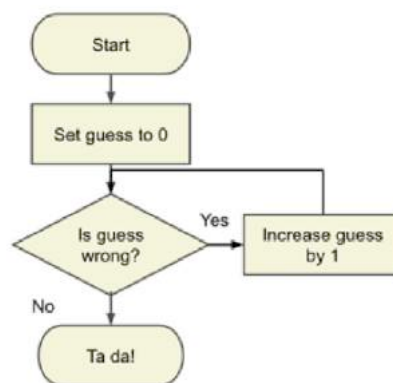


Figure 1.8 Linear search

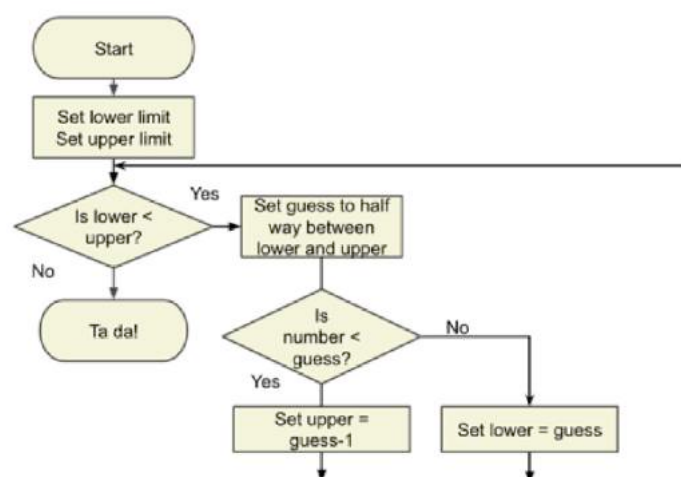


Figure 1.9 Binary search

After only a few goes at the game, or perhaps just by reasoning logically about the flowcharts here, it should be evident that the first two here are much slower than the third one, unless you happen to be very lucky. If we are trying to guess a number between, say, 0 and 127, the first two algorithms would take on average 64 steps to get there, sometimes more, sometimes less. If we use the binary search algorithm, we get there in just seven goes – try it! The efficiency gains get even more dramatic with a bigger range of starting numbers – guessing a number between one and a million would take on average 500,000 goes with either of the first two algorithms, but we would narrow down on the number in just 20 steps with binary search – try it!

Notice the recursive nature of the binary search – that we take the problem of finding a number between 0 and 127 and reduce it to a **simpler, but identically structured problem** of finding a number between 0 and 63 or 64 and 127 – and then apply the same binary search algorithm to solving this new problem.

4 See, for example, David J. Malan's demonstration for Harvard's CS50: <https://youtu.be/zFenJtAEzE?t=16m35s>



This recursive technique of reducing a problem to a simpler problem and then applying the same algorithm to that is called **divide and conquer**, and it is an excellent example of an algorithmic pattern that can be applied in lots of different contexts, including classical search problems like this, finding highest common factors (see Euclid's algorithm on [page 64](#)), sorting a list of numbers (see quicksort and merge sort on [pages 71 - 73](#)) and parsing algebraic expressions, computer code or even natural language.<sup>(5)</sup>

It is worth noting that binary search *only* works if the list is already in order: finding a word in a dictionary this way is only possible if the dictionary is already in alphabetical order. For an un-sorted list it would be hard to do better than starting at the beginning and working through to the end using linear search. Similarly, for the guess a number game, we can use this method because numbers are in order and we can ask the 'is it less than' question.

The problem of searching for information on the web ([see pages 123 - 124](#)) is obviously much more complex than simply finding a word in a dictionary or a secret number in a game, but even here one of the problems a search engine needs to solve is looking up the results in its index of the web for each of the keywords typed in by the user.

## Sort

In order to find things in a list quickly, time after time, it's really helpful if that list is in order. Imagine the problem of finding an email if all your messages were in some random sequence, or of finding a document on a computer if there was no way to sort a folder's contents into order, or of finding a contact on your phone if each time you had to look through a randomly-ordered list of everyone you knew, or of finding a book in the school library if they had never been put into any order. There are many, many areas where a list in some natural order seems an intuitive way to organise information. Notice that **once a list is in order** adding a new item into the list is easy – you just search for the right place, add the item and shuffle everything afterwards along a place. Getting the list into order in the first place is another matter.

It is well worth getting pupils to think about this problem for themselves, perhaps initially as an unplugged activity, sorting a group of their peers into height order or by birthday or using numbered slips of paper or a pan balance and pots with hidden masses. You can do this as a programming task once pupils have learnt how to manipulate a list in the programming language you are teaching, but as with any programming it's wise to have planned how to solve the problem (that is, the algorithm) before working with code.

There is quite a variety of sort algorithms around, some are much more efficient than others, and some are certainly easier to understand than others. For computing, it's worth teaching at least a couple so that pupils can use logical reasoning to compare them. Selection sort or bubble sort are good starting points, but ideally include quicksort or merge sort so that pupils can see how some algorithms are more efficient than others for the same problem.

## Selection sort

This is quite an intuitive approach.

- Start with a list.
- Find the smallest item\* and swap it with the item in the first place.
- Now, starting with the second item, look for the next smallest and swap it with the item in the second place.
- Starting with the third item, look for the next smallest and swap it with the item in the third place.
- Keep doing this until you get to the last item.

\* finding the smallest at each step involves comparing the smallest-so-far with all the other items after it until the end of the list, at which point the smallest-so-far must be the smallest of the group.

Try it for yourself!<sup>(6)</sup> Can you use logical reasoning to work out how this works? Could you explain the **idea** here to your pupils? Could you work out how many comparisons you would have to make to sort a list of 10 things?<sup>(7)</sup>

5 See [https://en.wikipedia.org/wiki/Divide\\_and\\_conquer\\_algorithms](https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms)

6 See the Harvard CS50 demonstration at [www.youtube.com/watch?v=f8hXR\\_Hvybo](http://www.youtube.com/watch?v=f8hXR_Hvybo)

7  $9+8+7+6+5+4+3+2+1=45$ .

## Bubble sort

Bubble sort has quite a lot in common with selection sort, but we make swaps as we go rather than just swapping the next smallest into the right place.

- Start with a list.
- Compare the first and the second item: if they are in the right order, that's fine, if not, swap them.
- Compare the second and the third item: if they are in the right order, that's fine, if not, swap them.
- Keep doing this until the end of the list: the last item should now be the largest of all.
- Now use the method above to work through everything up to the item before the last to find the second largest, then the list up to everything two before the end to get the third largest, then three before the end and so on until there's only one thing left.
- The list is now sorted.

Try it yourself!<sup>(8)</sup> Again, can you explain how this works? Could you explain it to a class? Can you predict how many comparisons it would take to sort a list of 10 things?<sup>(9)</sup>

## Quicksort

The quicksort algorithm wasn't discovered (or invented) until 1959 (published in 1961: Hoare, 1961), but it is still quite an intuitive method – it's also much faster, although it can be a bit trickier to program if the language does not support functions (see page 73). The idea is:

- Start with a list.
- If there's only one thing (or nothing) in the list, stop, as that's already sorted!
- Pick one item in the list which we will call the 'pivot'. Any item will do as the pivot, so you may as well choose the first item.
- Compare all the other items with the pivot, making two lists: one of things smaller than the pivot, the other of things larger than (or equal to) it.
- Now use this method to sort **both** of the unsorted lists.

- Your sorted list is made up of the sorted smaller items, the pivot, then the sorted larger items.

Again, try this!<sup>(10)</sup> Did you notice it was quicker than selection sort or bubble sort? Can you explain how this works? Can you think of a way to program this in a language you are familiar with? How many comparisons would it take to sort 10 things into order?<sup>(11)</sup>

## Merge sort

Whereas quicksort works top down, partitioning the list and then sorting each part, merge sort can be thought of as working from the bottom up.

Break the list into pairs, and then sort each pair into order. Now look at groups of four, that is two pairs, sorting them into order – this is quite easy as each of the pairs is already ordered and so the next item in the sorted group has to come from the beginning of each pair. Now look at groups of eight, sorting them into order by looking at the beginning of each sub-group of four and so on until the whole list is sorted. Here's an example:

Original list: 3 – 1 – 4 – 1 – 5 – 9 – 2 – 7

Sorting pairs: 1 3 – 1 4 – 5 9 – 2 7  
(4x1 comparisons)

Sorting quads: 1 1 3 4 – 2 5 7 9  
(2x3 comparisons)

Sorting all eight: 1 1 2 3 4 5 7 9  
(1x7 comparisons)

Again, to really get a feel for this, you need to try it for yourself!<sup>(12)</sup> Merge sort is as fast as quicksort, and is quite amenable to being run in parallel processing situations, where the work of a program is divided between multiple processors.

## Other algorithms

It would be wrong to leave pupils with the impression that the only interesting algorithms are about searching for things or sorting lists into order. Mathematics provides some rich territory for

8 See the Harvard CS50 demonstration at [www.youtube.com/watch?v=8Kp-8OGwphY](http://www.youtube.com/watch?v=8Kp-8OGwphY)

9 Also 45.

10 See the Harvard CS50 demonstration at [www.youtube.com/watch?v=aQiWF4E8fIQ](http://www.youtube.com/watch?v=aQiWF4E8fIQ)

11 This depends on the choice of pivot at each stage, but could be as low as  $9+4+4+1+1+1+1=21$ .

12 See the Harvard CS50 demonstration at [www.youtube.com/watch?v=EeQ8pwjQxTM](http://www.youtube.com/watch?v=EeQ8pwjQxTM)

pupils to experiment with algorithmic thinking for themselves, and exploring mathematical problems from a computer science standpoint seems likely to help develop pupils' mastery of mathematics.

## Testing for primes

Take as an example checking whether or not a number is prime (that is it cannot be divided by any number other than itself and one). A naive algorithm for this would be to try dividing by any number between 1 and the number itself; if none of them go in then the number must be prime. We can do **much** better than this though by thinking more carefully about the problem:

- We don't need to go up to the number – nothing past the half-way point could go in exactly.
- We don't need to go past the square root of the number – if something bigger than the square root goes in, then something less than the square root goes in as well (for example, because 50 goes into 1,000, we know 20 does too, as  $50 \times 20 = 1000$ ).
- If two doesn't go in, then no even numbers go in either; if three doesn't go in, no multiples of three can go in either; and so on.

So a quicker test for whether a number is prime is to try dividing by all the **primes** up to the square root of the number: if none of them go in, the number must be prime. (Try it!)

Big (well, very big) prime numbers play an important role in cryptography, and so algorithms to find and test whether numbers are prime have some important applications, but despite efficiencies such as the above algorithm, this is still a **hard** problem, without a guaranteed fast solution (as yet). There is, though, an algorithm that can tell if a number is **probably** prime, and you could run this many times to check whether a number is **almost** certainly prime or not (Miller, 1976).<sup>(13)</sup> Is this an algorithm? After all, it only says 'almost certainly prime'? Yes, it is an algorithm in the sense that it defines a precise sequence of steps that a computer can carry out. And how likely is that you would conclude 'N is prime' and be wrong? With a couple of hundred iterations, it is far more likely that an asteroid will strike the earth tomorrow than it is for N to be non-prime.

## Finding a list of primes

Finding a list of all the primes up to a certain limit has an interesting (and old) algorithmic solution. We could simply start at the beginning of our list and test each number in turn using the above algorithm. A bit slow, but it would get there, although we would need to watch out for the subtlety of needing a list of the primes up to the square root of each number to try dividing by.

We can do better than this though, using a method called the Sieve of Eratosthenes:

- Start with your list of numbers from 1 up to and including your limit.
- Cross out 1, as it's not prime.
- Take the next number not crossed out (initially 2) and cross out all the multiples of two – you can do this quickly by just counting on (initially in steps of 2).
- Repeat this step until the next number not crossed out is more than the square root of the limit.
- Anything not yet crossed out is a prime number.

+	2	3	4	5	6	7	8	9	+0
11	+2	13	+4	+5	+6	17	+8	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	+00

This is a really nice programming challenge too.<sup>(14)</sup>

## Finding the highest common factor

One more number theory problem, and again something with wide applications from implementing fractions arithmetic on a computer to internet cryptography, is to be able to find quickly the largest number which divides into a pair of numbers, that is the highest common factor or greatest common divisor.

A slow, naive algorithm would be to find all the numbers which divide into both and simply take the largest of these. A somewhat faster method is

<sup>13</sup> qv Simon Peyton Jones for CAS TV: <https://youtu.be/ixmbkp0QEDM?t=12m30s>

<sup>14</sup> Sieve programs in many languages: <http://c2.com/cgi/wiki?SieveOfEratosthenesInManyProgrammingLanguages>

the ‘ladder’ algorithm that is sometimes taught in schools:

- put the numbers side by side;
- find the smallest number (bigger than 1) that divides evenly into both;
- replace the numbers with how many times that number goes in;
- repeat until there is no number that can divide both;
- the highest common factor is then simply the product of the common prime factors.<sup>(15)</sup>

Better still is Euclid’s algorithm for this, which dates back to c.300 BCE.

The modern version of this uses modular arithmetic – that is finding the remainder on division by a number, but you can do this with nothing more sophisticated than repeated subtraction; it is also **really** fast:

- Start with two, non-zero numbers.
- Is the smaller number zero? If so, the highest common factor is the larger number (this won’t happen the first time around).
- Replace the larger number with the remainder you get when you divide by the smaller.
- Now repeat this process.

So, if we start with say 144 and 64, we get 16 and 64, then 0 and 16, so 16 is the highest common factor. Try it with some other numbers then have a go at writing a program to implement this.

## Estimating pi

Another very nice algorithm to estimate pi is this:

- throw a dart at a  $2r \times 2r$  board;
- see if it lands in the inscribed circle radius  $r$ .

The proportion of darts that land in the circle should be the area of the circle divided by the area of the board, that is  $\frac{\pi r^2}{4r^2}$ , which allows you to estimate pi! So in programmatic terms:

- choose two random numbers,  $x, y$  in  $-r$  to  $r$ ;
- See if  $x^2 + y^2$  is less than  $r^2$ . If so, the dart landed in the circle;
- Keep doing this, keeping track of how many landed inside and how many outside.

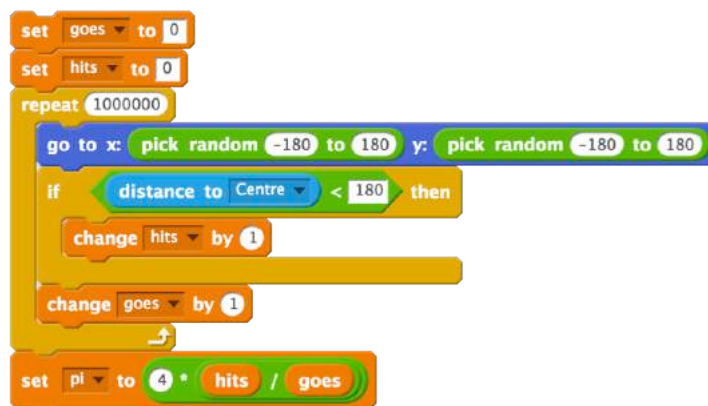


Figure 1.10 Scratch snippet to estimate pi, (see <https://scratch.mit.edu/projects/61893848/>)



## Classroom activity ideas

- It is worth starting with problems rather than solutions to encourage algorithmic thinking: set pupils the challenge of finding an algorithm that can find a short path through a graph or search a list or sort some numbers or test if a number is prime. Challenge them to find better algorithms to do this (see the evaluation on page xx).
- There’s much scope here for using ‘unplugged’ activities, that is activities that teach algorithmic thinking and some of these key algorithms without going near computers. You could play the ‘guess my number’ game as a class, trying to find a winning strategy or ask pupils to write instructions for finding a book in the library or for sorting out a set of masses. There are some great ideas online from CAS Barefoot, CS4FN and CS Unplugged.
- There’s much scope for algorithmic thinking in games and puzzles – can pupils work out the set of rules for playing an unbeatable game of noughts and crosses, Nim or Mastermind, or completing a Knights’s Tour of a (possibly small) chess board?
- Get pupils to think about when there are formal sets of rules or sequences of steps in other subject areas or in school life. In cookery, recipes have much in common with algorithms. In science, experimental methods do too. Challenge them to think of better algorithms for the same tasks (see the evaluation on page xx).

<sup>15</sup> See [www.youtube.com/watch?v=oKfwT-5DqsA](http://www.youtube.com/watch?v=oKfwT-5DqsA) for one presentation of this.



- Don't be afraid to get pupils implementing their algorithms as programs: it is the thinking that we are focussing on here, but there is much to be said for linking algorithms and coding together.



### Further resources

Bagge, P. (2014) *Flow charts in primary computing science*. Available from <http://philbagge.blogspot.co.uk/2014/04/flow-charts-in-primary-computing-science.html>

Cormen, T. (2013) *Algorithms unlocked*. MIT Press.

CS Field Guide (n.d.) *Algorithms*. Available from <http://csfieldguide.org.nz/en/chapters/algorithms.html>

*CS Unplugged* (2016) Available from <http://csunplugged.org/>

*CS4FN* (n.d.) Available from <http://www.cs4fn.org/>

Du Sautoy, M. (2015) *The secret rules of modern living* (for BBC Four).

Khan Academy (n.d.) *Algorithms*. Available from [www.khanacademy.org/computing/computer-science/algorithms](http://www.khanacademy.org/computing/computer-science/algorithms)

Peyton Jones, S. (2010) *Getting from A to B: Fast route-finding using slow computers*. Microsoft. Available from [www.microsoft.com/en-us/research/video/getting-from-a-b-fast-route-finding-using-slow-computers/](http://www.microsoft.com/en-us/research/video/getting-from-a-b-fast-route-finding-using-slow-computers/)

Peyton Jones, S. (2014) *Decoding the new computing programmes of study*. Computing at School. Available from <http://community.computingatschool.org.uk/resources/2936>

Slavin, K. (2011) *How algorithms shape our world*. TED. Available from [www.ted.com/talks/kevin\\_slavin\\_how\\_algorithms\\_shape\\_our\\_world?language=en](http://www.ted.com/talks/kevin_slavin_how_algorithms_shape_our_world?language=en).

Steiner, C. (2013) *Automate this: How algorithms came to rule our world*. New York, NY: Portfolio Penguin.

# Decomposition

How do I solve a problem by breaking it into smaller parts?

The process of breaking down a problem into smaller manageable parts is known as decomposition. Decomposition helps us solve complex problems and manage large projects.

This approach has many advantages. It makes a complex process or problem a manageable or solvable one – large problems are daunting but a set of smaller related tasks are much easier to take on. It also means that the task can be tackled by a team working together, each bringing their own insights, experience and skills to the task.

In modern computing, massively parallel processing can be used to significantly speed up some computing problems if they are broken down into parts that each processor can work on semi-independently of the others, using techniques such as MapReduce (Dean and Ghemawat, 2004).

How is decomposition used in the real world?

Decomposing problems into their smaller parts is not unique to computing: it is pretty standard in engineering, design and project management.

Software development is a complex process and so being able to break down a large project into its component parts is essential – think of all the different elements that need to be combined to produce a program like PowerPoint.

The same is true of computer hardware (see Figure 1.11): a smartphone or a laptop computer is itself composed of many components, often produced independently by specialist manufacturers and assembled to make the finished product, each under the control of the operating system and applications.

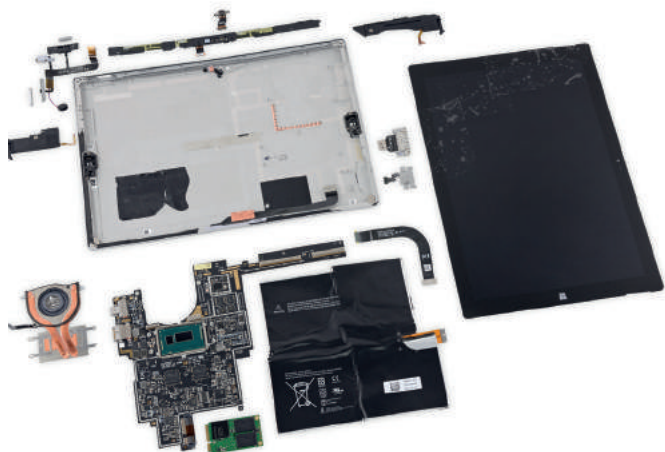


Figure 1.11 A tablet can be broken down (decomposed) into smaller components

You will have used decomposition to tackle big projects at school, just as programmers do in the software industry. For example, delivering your school's curriculum: typically this would be decomposed as years and subjects, further decomposed into terms, units of work and individual lessons or activities. Notice how the project is tackled by a team working together (your colleagues) and how important it is for the parts to integrate properly.

## Where does decomposition fit in the new computing curriculum?

In primary school pupils should have learnt to 'solve problems by decomposing them into smaller parts' (DfE, 2013); they should also have learnt to design and create a range of systems with particular goals in mind (here, system implies something with a number of interconnected components).

At Key Stage 3, there is an expectation that pupils will use modularity in their programming, using subroutines, procedures or function (see page 63). If their programs are to use a modular approach, this is something that they will need to take into account at the design stage too. For example, creating a complex turtle graphics figure such as in Figure 1.12 almost demands that pupils recognise that this is built up from a repeated square, and their program must include a set of instructions to draw a square. Similarly a turtle graphics program to draw a house is likely to include routines (perhaps as procedures) to draw doors, windows, a roof and so on.

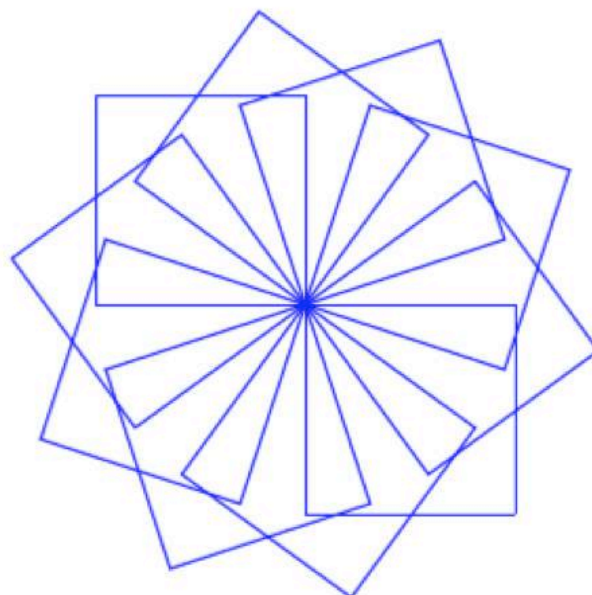


Figure 1.12

Notice that when we thought about the algorithm for quicksort earlier, we took for granted that we could partition (split) a list into those things smaller than our pivot and those which are larger than or equal to it – in implementing quicksort as code, we would need to implement this function too, if it is not already present in the programming language we are using.

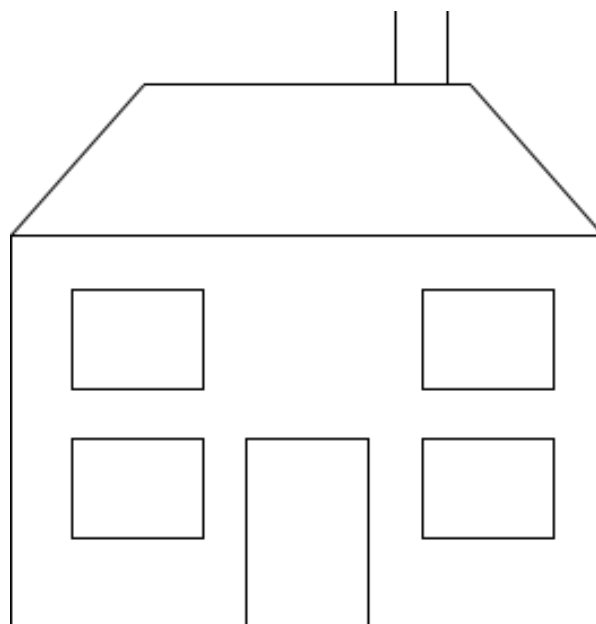


Figure 1.13

Divide and conquer algorithms like binary search and quicksort also use decomposition, but the distinctive feature of these is that the smaller problems have essentially the same structure as the larger one. This idea of the parts of a system or algorithm being similar to the whole is known as recursion: it is a very powerful way of looking

at systems with wide applications; for example the internet can be thought of as a network of networks, and each of those networks might be composed of still further networks. Recursive patterns occur in nature too: look for example at ferns, broccoli and other fractals. Pupils could draw representations of these using turtle graphics commands in Scratch, TouchDevelop, Small Basic or Python (see Figures 1.14–1.15):

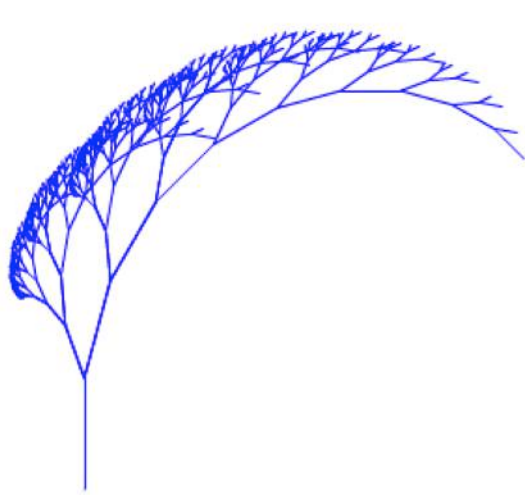


Figure 1.14

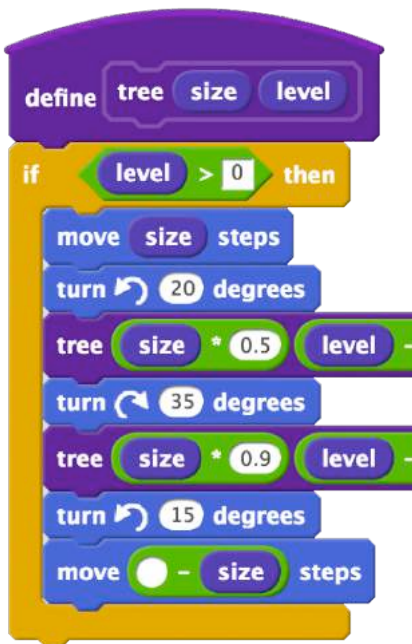


Figure 1.15

*Recursive procedure in Scratch to draw Figure 1.15.<sup>(16)</sup>*


```
def tree(level,size):
    if level > 0:
        forward(size)
        left(20)
```

```
tree(level-1,size*0.5)
right(35)
tree(level-1,size*0.9)
left(15)
forward(-size)
else:
    return
```

*Python turtle code for the same fractal.<sup>(17)</sup>*

As pupils plan their programs or systems encourage them to use decomposition: to work out what the different parts of the program or system must do and to think about how these are inter-related. For example, a simple educational game is going to need some way of generating questions, a way to check if the answer is right, some mechanism for recording progress such as a score and some sort of user interface, which in turn might include graphics, animation, interactivity and sound effects. Thinking of the game like this is essential to the planning process.

On larger projects, decomposition also makes it possible for pupils to work as a collaborative team, as team members can take responsibility for implementing each of these features, and then ensuring that they will work properly when combined. Plan opportunities for pupils to get some experience of working as a team on a software development project, and indeed other projects in computing. This could be media work such as animations or videos, shared online content such as a wiki or a challenging programming project such as making a computer game or a mobile phone app.



### Classroom activity ideas

- Organise for the pupils to tackle a large-scale programming project such as making a computer game, through decomposition. Even for a relatively simple game the project would typically be decomposed as follows: planning, design, algorithms, coding, animation, graphics, sound, debugging and sharing. A project like this would lend itself to a collaborative team-based approach with development planned over a number of weeks.

<sup>16</sup> See <https://scratch.mit.edu/projects/72176670/>  
<sup>17</sup> See <https://trinket.io/python/8996e36dda>

- Take the case off an old desktop computer and show the pupils how computers are made from systems of smaller components connected together. Depending on the components involved some of these can be disassembled further still, although it is likely to be better to look at illustrations of the internal architecture of such components.
- Organise for the pupils to carry out a collaborative project online, for example through developing a multi-page wiki site. For example, pupils could take the broad topic of e-safety, decompose this into smaller parts and then work collaboratively to develop pages for their wiki, exploring each individual topic. The process of writing these pages can be further decomposed through planning, research, drafting, reviewing and publishing phases.
- Introduce pupils to divide and conquer approaches and other applications of recursion through binary search, quicksort and experimenting with simple fractals using turtle graphics. Encourage them to look out for other occasions on which this powerful technique can be used.

Project Management Institute Educational Foundation (2011) *Project management toolkit for youth*. Available from <http://pmief.org/learning-resources/learning-resources-library/project-management-toolkit-for-youth>.

## Abstraction

How do you manage complexity?

For American computer scientist Jeanette Wing, credited with coining the term computational thinking, abstraction lies at the heart of this:

**The abstraction process – deciding what details we need to highlight and what details we can ignore – underlies computational thinking. (Wing, 2008)**

Abstraction is about simplifying things; identifying what is important without worrying too much about the detail. Abstraction allows us to manage complexity.

We use abstractions to manage the complexity of life in schools. For example, the school timetable is an abstraction of what happens in a typical week: it captures key information such as who is taught what subject where and by whom, but leaves to one side further layers of complexity, such as the learning objectives and activities planned in any individual lesson.

Pupils use abstraction in mathematics, when solving ‘real world’ problems, but mathematical abstractions are typically numerical, algebraic or geometrical, whereas in computing they can be far more general. In computing, abstractions are also multi-layered: computer systems are made up of boxes within boxes within boxes. We are able to tackle complex problems because others have built the components on which our solution depends – at one level we may be interested in playing a game but not concerned with how that game has been programmed; at another level we might be interested in the program but less so in the interpreter or compiler which converts that into machine code; at another with that machine code but less about how this is executed on the CPU or stored in physical memory. Wing puts it well:

### Further resources

*Apps for Good* (n.d.) Available from [www.appsforgood.org/](http://www.appsforgood.org/)

Barefoot Computing (2014) *Decomposition*. Available from <http://barefootcas.org.uk/sample-resources/decomposition/> (free, but registration required).

*Basecamp* (n.d.) Professional project management software that can be used by teachers with their class (free). Available from <https://basecamp.com/teachers>

BBC Bitesize (n.d.) *Decomposition*. Available from <http://www.bbc.co.uk/education/guides/zqqfyrd/revision>

(2011) *Ratatouille: Rats doing massively parallel computing*. Available from [www.cs4fn.org/parallelcomputing/parallelrats.php](http://www.cs4fn.org/parallelcomputing/parallelrats.php)

Gadget Teardowns (n.d.) *Teardowns*. Available from [www.ifixit.com/Teardown](http://www.ifixit.com/Teardown)

NRICH (n.d.) *Planning a school trip*. Available from <http://nrich.maths.org/6969>; *Fractals*. Available from <http://nrich.maths.org/public/leg.php?code=-384>



**In computing, we work simultaneously with at least two, usually more, layers of abstraction: the layer of interest and the layer below; or the layer of interest and the layer above. Well-defined interfaces between layers enable us to build large, complex systems. (Wing, 2008)**

Programming also makes use of abstraction – in modular software design, programmers develop procedures, functions or methods to accomplish particular goals, sometimes making these available to others in libraries. What matters is what the function does and that it does this in a safe and reliable way; the precise implementation details are much less important.

## Where does abstraction fit in the new computing curriculum?

Abstraction is part of the overarching aims for the computing curriculum, which seeks to ensure that pupils:

**can understand and apply the fundamental principles and concepts of computer science, including abstraction, logic, algorithms and data representation. (DfE, 2013)**

At primary school, pupils will have encountered the **idea** of abstractions in maths, as ‘word problems’ are represented in the more abstract language of arithmetic, algebra or geometry, or in geography where the complexity of the environment is abstracted into maps of different scales. In their computing lessons, pupils may have learnt about the process of abstraction from playing computer games, particularly those that involve interactive simulations of real world systems; most pupils will have experienced writing computer games or other simulations themselves: the Key Stage 2 programme of study expects pupils to be taught to:

**design, write and debug programs that accomplish specific goals, including controlling or simulating physical systems.**

The multi-layered nature of abstraction in computing is well worth discussing with pupils as they learn about **how** computers work, for example asking pupils to work out the **detail** of what’s

happening inside a computer when they press a key and the corresponding letter appears on screen in their word processor or text editor. Similarly, pupils’ knowledge of how search engines work or how web pages are transmitted across the internet is going to draw on their understanding of the many different **layers** of systems on which these processes depend.

In Key Stage 3, abstraction is an intrinsic part of developing computational solutions to real world problems, as pupils focus attention on the detail of the real world problem or system, deciding for themselves what they won’t need to take account of in the algorithms, data structures and programs that they develop. The programme of study includes the requirement to:

**design, use and evaluate computational abstractions**

As well as understanding the algorithms that describe systems or allow us to compute solutions to problems, designing and using computational abstractions also means that we need the right **data structure** to describe the state of the system. Niklaus Wirth famously argued that programs are made up of algorithms and data structures (Wirth, 1976) and more recently Greg Michaelson has described solutions as made up of computation plus information (Michaelson, 2015). Whilst most pupils will be familiar with the concept of algorithms from their primary school, few will have spent long considering the information or the data that they need to take into account when designing an abstraction of a problem or system and how this can be best represented in a computer.

The programme of study talks of modelling ‘the state and behaviour of real-world problems and physical systems’, and this provides one approach to thinking about the relationship between algorithms and data, with the algorithm describing the set of rules or sequence of steps that the system obeys, and the data structure describing the state in which the system is.

Take for example the process of shuffling a deck of cards. A ‘perfect’ riffle shuffle could be described by the following algorithm:

- Split the pack in two, calling these smaller packs the top and the bottom.

- Take cards sequentially, one from the top, one from the bottom to assemble a new pack.
- Repeat until there are no cards remaining in the top or bottom.

This describes the **behaviour** of our system, but any abstraction also needs to include the **state** of the system, which is simply an ordered list of the cards in the pack. Working with a deck of just eight cards, we might start with:

A, 2, 3, 4, 5, 6, 7, 8

Applying the above algorithm once would change this to:

A, 5, 2, 6, 3, 7, 4, 8

Card games are rich territory for thinking about computational abstractions: the rules of the game describe (perhaps only in part) the behaviour of the system – they are its algorithm; the cards in the pack and in individual hands describe the state of the system. Games in general, from snakes and ladders or noughts and crosses to chess and go, can be typically thought of as sets of rules (algorithms) for legally-valid moves and some data structure (often counters or pieces on a board, plus perhaps a random element such as a dice) to define the state of the game at any time. The same is true of computer games: Minecraft could be thought of as a complex system of rules governing the interaction of blocks and other game elements, as well as a three-dimensional data structure (and some random elements).<sup>(18)</sup>

Mathematician J H Conway created a simple computational abstraction, his ‘Game of Life’ (Berlekamp et al., 2004),<sup>(19)</sup> that opened up a rich field of research in mathematics, computer science and biology into the behaviour of cellular automata. In Conway’s Game of Life, the state of the world is a two-dimensional grid (or an array) where each cell is either alive or dead. The algorithm or rules of the game describe its behaviour. For each cell:

- Any live cell with fewer than two live neighbours dies.
- Any live cell with two or three live neighbours lives on to the next generation.

- Any live cell with more than three live neighbours dies.
- Any dead cell with exactly three live neighbours becomes a live cell.

Thus, for example, the pattern

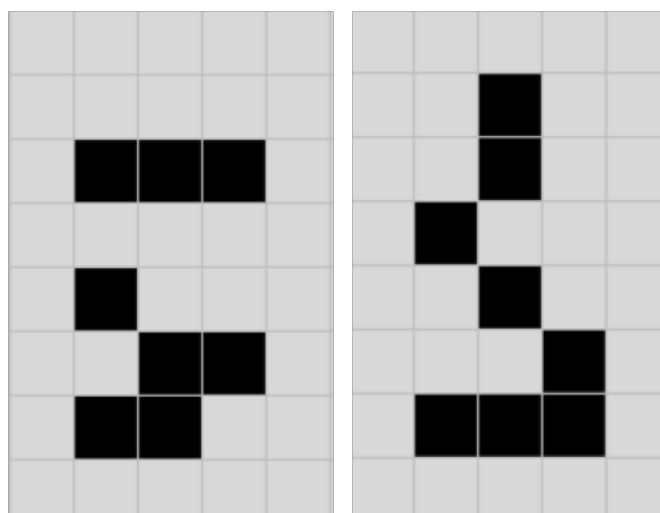


Figure 16a becomes Figure 16b

which becomes

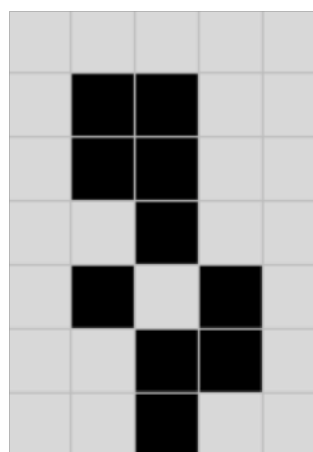


Figure 16c

and so on (Figure 16a–c).

You can play the game on a grid using counters to represent live cells, but it’s quicker on a computer and not too challenging an abstraction for Key Stage 3 pupils to be able to program for themselves. Despite this relatively simple description and ease of implementation, complex, unexpected and subtle behaviour emerges in this system: for example Life can itself be ‘programmed’ to perform any computation.

<sup>18</sup> Things in Minecraft are somewhat more sophisticated than this. See [http://minecraft.gamepedia.com/Chunk\\_format](http://minecraft.gamepedia.com/Chunk_format) for details of the data structure used, or explore this in Minecraft for the Raspberry Pi: [www.raspberrypi.org/learning/getting-started-with-minecraft-pi/worksheet/](http://www.raspberrypi.org/learning/getting-started-with-minecraft-pi/worksheet/)

<sup>19</sup> Computer implementations of Life include Golly: <http://golly.sourceforge.net/>

As well as the list that represents the state of a pack of cards and the two-dimensional array that represents the state of cells in Life, the graph is a particularly useful data structure that can be used in a wide variety of computational abstractions. A graph consists of nodes connected by edges. (A graph, in this sense, is quite different from the sort of statistical charts or diagrams which pupils might be familiar with. The same technical term ‘graph’ is used for two completely different things.)

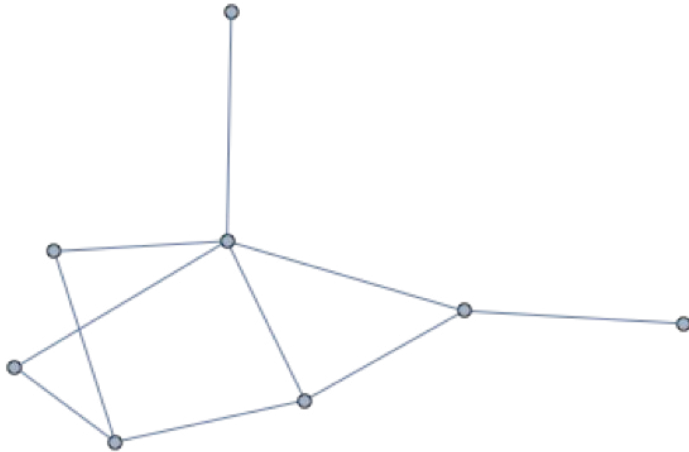


Figure I.17

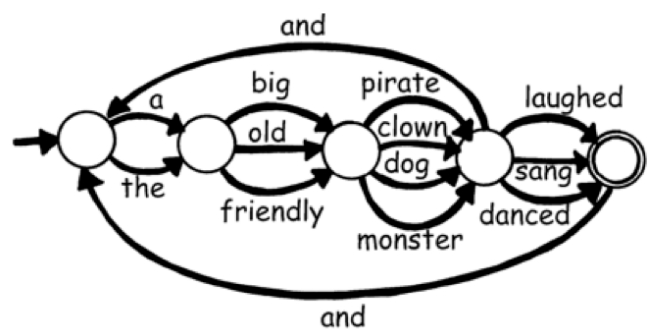
The classic example of a graph as a computational abstraction is the London Underground Map, which simply shows which stations connect with which others. This sort of abstraction allows us to work out quite easily the route to take from one station to another without concerning ourselves with the physical location of stations or the detail of underground engineering such as track gradients or curve radii. Note that different abstractions of the same underlying reality are useful for different purposes: for example, the underground map is useless for estimating how far apart on the surface two stations are or (if you were a maintenance engineer) how much track-cleaning fluid you would need to clean the track between Russell Square and Covent Garden.

Graphs like this can also represent the ‘friendship’ links in social networks such as on Facebook or links between scientists co-authoring papers (Newman, 2001) or actors who have co-starred.<sup>(20)</sup>

In most cases we attach values or labels to the nodes, as in these examples. In some cases the

edges connect nodes in one direction but not the other – for example, the links between web pages or followers on Twitter. In some cases we attach numbers or weights to the edges, for example so that we can work out the shortest path between two nodes, as in the sat nav example earlier (page 12). There are many examples of problems which appear initially very complex (such as the Knight’s Tour in chess [Curzon, 2015]) but become very much simpler to solve using standard algorithms when represented as graphs. Graphs are particularly general sorts of structures: lists, arrays and trees (such as in our binary search example on page 15) can all be thought of as special sorts of graphs.<sup>(21)</sup> Languages such as Snap!, Python and Mathematica have libraries available for working with graphs.

Another interesting, and important, way of modelling state and behaviour is through ‘finite state machine’. A graph is a very useful way to visualise this sort of abstraction, representing the state of the system as nodes and the transitions between states as the edges. It is possible to think of the grammatical structure of languages this way, for example this from CS Unplugged (Figure I.18) generates grammatically correct sentences (notice the double circle on the right, used to show the exit or ‘accept’ state of the machine).

Figure I.18<sup>(22)</sup>

This sort of abstraction is very useful when thinking about interface design – from toasters and kettles through digital watches and cash point machines to websites and sophisticated apps. Thinking of the states of a system, such as an app, and how the app moves between these states can be helpful in the design process. Notice the similarities between the state transition diagram for finite state machines and graphs – this isn’t coincidental and means that we can use the algorithms and techniques for graphs to explore the properties of finite state machines.

20 See <http://oracleofbacon.org/how.php>

21 Conversely, graphs can be represented as lists of edges or arrays of node connections.

22 From CS Unplugged: [http://csunplugged.org/wp-content/uploads/2014/12/unplugged-11-finite\\_state\\_automata.pdf](http://csunplugged.org/wp-content/uploads/2014/12/unplugged-11-finite_state_automata.pdf)

At Key Stage 3, there is the expectation that pupils will be drawing on the ideas of decomposition and abstraction by developing software with a modular architecture using procedures or functions, or drawing on procedures or functions developed by others. Procedures and functions are covered on [pages 63 - 67](#).



### Classroom activity ideas

- Without using computers to think about programming, set pupils the challenge of designing interesting playable games, thinking carefully about the state (typically, board, card deck, die) of their game and its behaviour (the rules or algorithm, according to which play takes place). Pupils might start by adapting the state and behaviour of games they are already familiar with (Noughts and Crosses, Nim, Draughts, Pontoon).
- In programming, you might ask pupils to create their own games. If these are based on real-world systems then they will need to use some abstraction to manage the complexity of that system in their game. In a simple table tennis game, for example Pong, the simulation of the system's behaviour includes the ball's motion in two dimensions and how it bounces off the bat, but it ignores factors such as air resistance, spin or even gravity: the state of the system might be modelled by coordinates to specify the position of the ball and the bats, as well as each player's score.
- Whilst developing an app is an ambitious project in Key Stage 3, designing apps is accessible to most pupils: they can take the idea of a finite state machine and apply it to designing the screens of their app and the components of the user interface which allow the app to move from one screen to another.



### Further resources

Barefoot Computing (2014) *Abstraction*. Available from <http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/abstraction/> (free, but registration required).

BBC Bitesize (n.d.) *Abstraction*. Available from [www.bbc.co.uk/education/guides/zttrcdm/revision](http://www.bbc.co.uk/education/guides/zttrcdm/revision)

BBC Cracking the Code (2013) *Simulating the experience of F1 racing through realistic computer models*. Available from [www.bbc.co.uk/programmes/p016612j](http://www.bbc.co.uk/programmes/p016612j)

Computerphile (2013) *The art of abstraction – Computerphile*. Available from [www.youtube.com/watch?v=p7nGcY73epw](http://www.youtube.com/watch?v=p7nGcY73epw)

CS Fieldguide (2016) *Formal languages*. Available from <http://csfieldguide.org.nz/en/chapters/formal-languages.html>

CS4FN (n.d.) *Download our computer science magic books!* Available from [www.cs4fn.org/magic/magicdownload.php](http://www.cs4fn.org/magic/magicdownload.php)

CS4FN (2016) *Computational thinking: Puzzling tours*. Available from <https://cs4fndownloads.files.wordpress.com/2016/02/puzzlingtours-booklet.pdf>

Google for Education (2013) *Solving problems at Google using computational thinking*. Available from [www.youtube.com/watch?v=SVVB5RQfYxk](http://www.youtube.com/watch?v=SVVB5RQfYxk)

Kafai, Y. B. (1995) *Minds in play: Computer game design as a context for children's learning*. Mahwah, NJ: Lawrence Erlbaum.

## Patterns and Generalisation

How can you make things easier for yourself?

There is a sense in which the good software engineer is a lazy software engineer, although this isn't the same as saying that all lazy software engineers are good software engineers! In this case, being lazy means looking for an easier way to do something: partly this is about efficient algorithms, for example looking for a quicker way for the computer to do something, but it's also about **not** doing unnecessary work yourself.

In the long run it can save lots of time to come up with a solution to a general class of problems and apply this to all the individual problems in that class, rather than solving each of these as an entirely separate problem. For example, in learning about area, pupils could find the area of a particular



rectangle by counting the centimetre squares on the grid on which it is drawn. It's better to realise that in each case all we need do is multiply the length by the width: not only is this quicker, it's also a method that will work **all** rectangles, including really small ones and really large ones. Although it takes a while for pupils to understand this formula, once they do it's so much faster than counting squares.

In computing, this method of looking for a general approach to a class of problems is called generalisation. It is dependent on the ability to spot patterns in the class of problems that you are working with: in the areas example, a child might well come to spot the relationship between the length and width of the rectangles and the number of centimetre squares they contain, suggesting a rule they can use for other rectangles too. In computing, in the early days of the web, directories of all the best sites were compiled to help people discover the pages they needed. The general rule for these lists might be to include those websites which lots of people used or linked to, and that's a pattern which can be used to produce a general solution to the problem automatically by search engines such as Google or Bing. Rather than individual editors compiling long lists of relevant web pages, an algorithm can apply the general rule of finding the most linked to pages with a particular phrase automatically to its indexed cache of the web.

In computer science, the field of machine learning has taken this idea of recognising patterns and made this something which computers can do. With a large enough database, it is possible for algorithms to spot patterns in the data which appear to be related to particular outcomes. For example, Amazon's algorithms can spot the patterns in the data of all their customers' buying and browsing habits to suggest which products another customer might be interested in. We are already starting to see these ideas being applied in education – with enough data available, the patterns in pupils' interactions with online questions can be used to suggest helpful activities for pupils to try next, in much the same way that teachers might make 'intuitive' judgements based on past experience of how best to teach a topic to the individuals in their class. Of course these judgements aren't really 'intuitive': because thinking every situation through logically, in such a way that we can give a rational explanation for our decision, is time consuming and difficult, we create our own 'rules of thumb'

based on the patterns we spot in our day to day professional experience.

The term 'pattern' has another meaning in software engineering: it also refers to common approaches to solving the same problem. Taking inspiration from A pattern language by Alexander et al. (1977), which sought to document good solutions to common problems in architecture and urban design, Gamma and three colleagues ('The Gang of Four') published a very influential set of 23 classic software design patterns for object oriented programming (Gamma et al., 1994). Examples of these design patterns include 'iterator', which accesses the elements of an object sequentially without exposing its underlying representation and 'memento', the 'undo' behaviour which provides the ability to restore an object to its previous state. Further design patterns have been written up since and the approach has been applied to other domains too, including teaching (Laurillard, 2012).

One particularly useful design pattern for developing software, including apps and games, is the 'model-view-controller' pattern – the model here is the part of the program that captures the computational abstraction of the state and the behaviour of the system; the view is the part of the program which displays the state of the system to the user; the controller is the part that allows the user to control the behaviour of the system. This pattern is the basis for most software that relies on user interaction via a graphical user interface (GUI) (Michaelson, 2016).

Generalisation is one of the reasons why computational thinking is so important beyond the realms of software engineering or computer science. Wing argues that computational thinking for everyone includes being able to:

**Apply or adapt a computational tool or technique to a new use,**

**Recognize an opportunity to use computation in a new way, and**

**Apply computational strategies such divide and conquer in any domain. (Wing, 2010)**

All of which are directly linked to this element of computational thinking.

## How are patterns and generalisation used in the national curriculum?

When at primary school, pupils are likely to have encountered the idea of generalising patterns in many areas of the curriculum. From an early age, they will become familiar with repeated phrases in nursery rhymes and stories; later on they will notice repeated narrative structures in traditional tales or other genres. In maths, pupils typically undertake investigations in which they spot patterns and deduce generalised results. In English, pupils might notice common rules for spellings themselves as well as being taught these and their exceptions.

Draw pupils' attention to the opportunities to use the same or similar techniques or approaches in computing, for example, highlighting where pupils can apply a 'divide and conquer' algorithm to solving a problem, or where lists, arrays or graphs would be the best way of thinking about the state of a system they wish to model, or where decomposition or abstraction provide effective strategies for dealing with a problem.

In computing at Key Stage 3, always encourage pupils to look for simpler or quicker ways to solve a problem or achieve a result, particularly where they can draw on the idea of patterns or generalisation to help them do this. One example might be developing a quiz program in Scratch or Small Basic – each question could be coded by hand but pupils might also create a general form of the question, using repetition to ask variations of this a number of times.

In turtle graphics, pupils might create programs to draw equilateral triangles, squares, regular pentagons and so on with sides of particular lengths, before generalising this pattern to create a procedure to draw any regular polygon with arbitrary length sides (Figure 1.19):

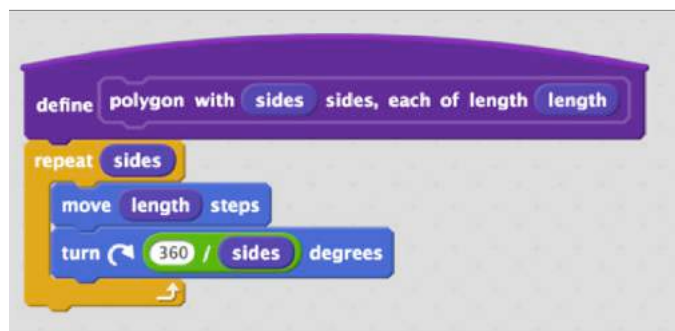


Figure 1.19 A Scratch block to draw a general regular polygon

```
def polygon (sides, length):
  for i in range (sides):
    fd (length)
    rt (360./sides)
```

*Python turtle code for drawing a general regular polygon*

As the above examples illustrate, as pupils become familiar with more programming languages, they might start to notice and draw on how patterns they've used in one programming language can be applied in another. For example when working in Scratch, pupils might often find that they need to keep count of the number of times they go round a repeating loop, using a structure like this (Figure 1.20):

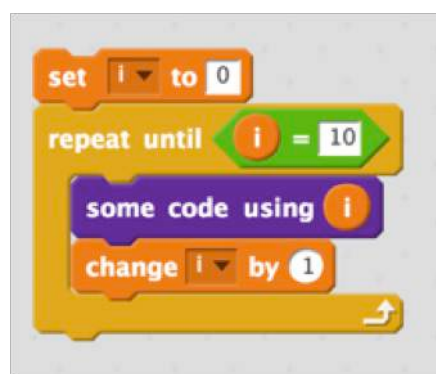


Figure 1.20

Subsequently, they will notice how the same idea is accomplished in say Snap! using the standard tools (Figure 1.21):



Figure 1.21

Or in Python:

```
for i in range(10):
    some_code_using(i)
```

Part of a pupil's learning to program is building up a portfolio of such patterns that they can draw on fluently in a range of different contexts for solving quite different problems. This fluency comes through reading and remixing code written by others (Rajlich and Wilde, 2002), as much as it might through writing a program from scratch.

Pupils also learn common ways of operating a range of different programs: one indication of pupils' developing IT capability is that they can generalise from their way of using one piece of software to working with a completely different piece of software, or from working with one computer system to a different platform.



### Classroom activity ideas

- In computing, encourage pupils to always look for simpler or quicker ways to solve a problem or achieve a result. Ask pupils to explore geometric patterns using turtle graphics commands in languages like Scratch, Logo or TouchDevelop to create 'crystal flowers' (see pages 61 - 62). Emphasise how the use of repeating blocks of code or procedures are much more efficient than writing each command separately, and allow pupils to experiment with how changing one or two of the numbers used in their program can produce different shapes.
- In programming, set pupils the challenge of completing the same task in two different programming languages, perhaps one block-based and the other text-based. Can they see the similarities between the two implementations of the same algorithm?
- When programming games or simple apps, encourage pupils to adopt, or at least to think in terms of, the model-view-controller design pattern.<sup>(23)</sup>
- Teach pupils to use graphics software to create tessellating patterns to cover the screen. As they do this, ask them to find quicker ways of completing the pattern, typically by copying and pasting groups of individual shapes, or

alternatively by writing a turtle graphics program to do this.

- Teach pupils to create rhythmic and effective music compositions using simple sequencing software in which patterns of beats are repeated; encourage them to experiment with repeating and changing the patterns of notes in their composition.



### Further resources

Barefoot Computing (n.d.) *Patterns*. Available from <http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/patterns/> (free, but registration required).

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D. and Marshall, K. (2011) **Recognizing Computational Thinking Patterns**. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (SIGCSE '11).

Hoover, D. and Oshineye, A. (2009) *Apprenticeship patterns: Guidance for the aspiring software craftsman*. Sebastopol, CA: O'Reilly.

*Isle of Tune* app (n.d.) Available from <http://isleoftune.com>

*Pattern in Islamic Art* (n.d.) Available from [www.patterninislamicart.com](http://www.patterninislamicart.com)

*Pysnap* (n.d.) Design patterns explained with Python examples. Available from [www.pysnap.com/design-patterns-explained/](http://www.pysnap.com/design-patterns-explained/)

## Evaluation

### How good is our solution?

Whereas other aspects of computational thinking are focussed on looking at problems or systems in such a way that computers can help us solve or understand them, evaluation is more concerned with checking that we have a solution and about considering qualities of that solution, from algorithmic efficiency through to design of the user interface.

<sup>23</sup> See, for example, [https://svn.python.org/projects/python/trunk/Demo/turtle/tdemo\\_nim.py](https://svn.python.org/projects/python/trunk/Demo/turtle/tdemo_nim.py) for an example of this approach to implementing the game of Nim.

In the Computing At School guide to computational thinking, the authors write:

**Evaluation is the process of ensuring that a solution, whether an algorithm, system, or process, is a good one: that it is fit for purpose. Various properties of solutions need to be evaluated. Are [they] correct? Are they fast enough? Do they use resources economically? Are they easy for people to use? Do they promote an appropriate experience? Trade-offs need to be made, as there is rarely a single ideal solution for all situations. There is ... attention to detail in evaluation based on computational thinking. (Csizmadia et al., 2015)**

Evaluation though isn't just for computer scientists or software engineers. As users of technology, it's important that everyone consider whether the software and hardware available is fit for purpose, and recognise the limits of what computers can do. Wing argues that computational thinking means that everyone should be able to:

**Evaluate the match between computational tools and techniques and a problem [and] understand the limitations and power of computational tools and techniques. (Wing, 2010)**

The multi-layered abstraction common to computational thinking provides one way of thinking through the evaluation of a computational solution.

At the most fundamental level, the algorithms that lay at the heart of the solution must be correct, and here logical reasoning can be used to provide proof that, given the correct input data, the algorithm will produce the correct output data. Alongside the algorithms governing the behaviour of the solution, the underlying abstraction must also reflect the state of the problem or system we are working with. In the process of developing a computational abstraction, some information is put to one side, reducing the complexity of the problem: the process of evaluation involves considering whether the choices in reducing complexity were well-made. Computational abstractions also involve choices over how data are to be structured and represented, and again evaluation should consider whether such choices were correct and optimal.

Evaluation also needs to consider the implementation of the algorithm and associated data structures as code. Part of this involves carefully and logically reviewing the code to ask whether it does what it should do, but also whether it's **good** code. Good code is likely to be well formatted and commented, so that it's easier for others to read and review. Variables and functions or procedures will have useful, sensible names. It's likely to make use of decomposition and abstraction through a modular approach. Good code is likely to use a simple, sensible, obvious way to get things done, drawing perhaps on some of the classic design patterns; it shouldn't make a reviewer wonder 'why did they do it like that?' Good code is also likely to make good use of the features of the language it is written in, particularly the available function libraries. Knuth argues for **literate** programming, in which programmers document the logical argument and design decisions of their programs, illustrating these with the source code (Knuth, 1992).

As well as reasoning about and understanding code, evaluation also has to involve testing code. Good practice, in programming if not always in education, is to test early and often: as each part of a program is written, it should be tested to see that it does exactly what it's supposed to do. Modular programming, typically involving functions or procedures, makes it easier to test code as it's being developed, as each function or procedure can be tested independently of the rest of the program.

Test-Driven Development (TDD) is an agile programming methodology in which the tests for features are written **first** before the new features are developed. There's a three-phase cycle here (Figure 1.22): at first the test should fail (as the new feature hasn't been implemented yet), then the test should pass (as the feature has been implemented successfully), and then there's often the need to refactor the already working code so that it's better integrated or more efficient. There are parallels here with assessment for learning in schools: check first what pupils don't know; teach them; check again that they've learnt this (providing evidence of progress); then ensure that they develop fluency and mastery in the new content. These tests are typically automated – we work out in advance what a function or procedure **should** do given different input data, and then check that it does indeed return the expected output.



## Evaluation in the national curriculum

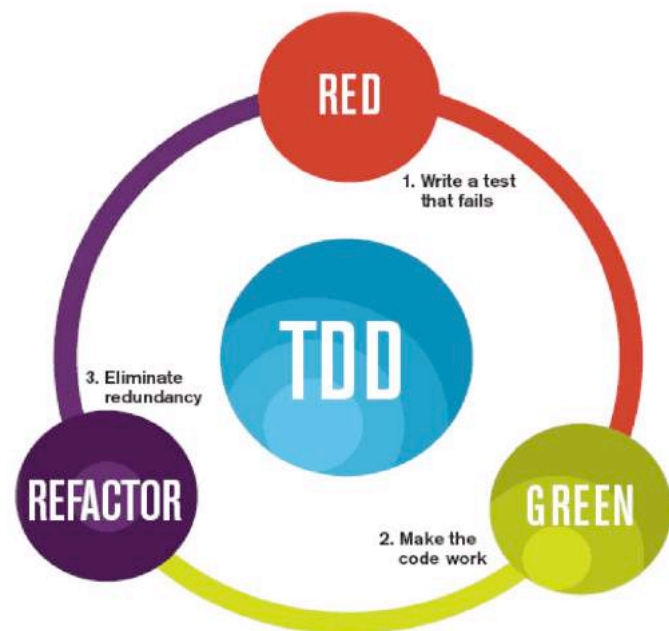


Figure 1.22 The mantra of Test-Driven Development (TDD) is ‘red, green, refactor’

Above this attention to the detail of implementing functions and procedures in code, it’s important that those evaluating computational artefacts don’t lose sight of the big picture. Typically, programs are written to solve problems and a crucial element of evaluation is checking that the program does indeed solve the problem. Evaluation might also take into account how **good** the solution is: is it an efficient one? Is it an elegant one? Is it one that meets the needs of its users? Does it meet all the overarching criteria in the design specification?

Typically software is developed with **users** in mind, and part of evaluation is about looking to see how well the software meets the needs and expectations of these users: whether it lets them get things done effectively and efficiently. This would include considerations such as user interface design, accessibility and appeal, and could draw on rigorous analytical techniques, A/B testing of different interfaces on live websites, observations of users’ interactions with the software and feedback surveys or focus groups.

There is another level of reflection here, which is important in education and software craftsmanship. Useful as it is to evaluate the artefacts produced, it’s also helpful to reflect on what has been learnt through the process of making the artefact, and indeed on how new knowledge, skills and understanding have been acquired.<sup>(24)</sup>

There is relatively little attention to formal evaluation in Key Stage 2 of the national curriculum. Pupils should have learnt to be discerning in evaluating digital content whilst at primary school: this might have involved their reflecting critically on their own work or that of their peers, but would also have involved their forming judgements about content produced by others and shared via the web. In primary school, pupils will also have learnt about evaluating data and information, and these evaluation skills can be built on in secondary school as pupils learn to evaluate algorithms, programs and other digital artefacts.

The programme of study at Key Stage 3 expects pupils to **evaluate computational abstractions**. This need not be computational abstractions of their own design, and the skills of evaluation can perhaps best be developed by looking at the abstractions made by others. Evaluating abstractions starts by considering whether the abstraction does model the original system; it then considers whether the abstraction is at the right level of detail, and then looks at whether the abstraction is actually a helpful one for solving the problem at hand or understanding the original system. Looking at different ways in which the same problem can be represented can soon make it clear that some abstractions are more useful than others (see, for example, Curzon, 2015). Finite state machines (see page 26) provide rich territory here, as pupils can consider the extent to which the abstraction does capture the relevant detail of the state and behaviour of the system it models, but they can also be used for evaluating interface design – how many transitions/clicks are needed to get back to the home page or find a company’s phone number on its website.

The curriculum also expects pupils to **use logical reasoning to compare the utility of alternative algorithms** for the same problem. An algorithm is only useful if it solves the problem it sets out to do: can pupils justify why an algorithm must work? How do they know that linear search will eventually find the right item or that bubble sort will produce a correctly ordered list?

24 See, for example, <https://educationendowmentfoundation.org.uk/evidence/teaching-learning-toolkit/meta-cognition-and-self-regulation/>

Evaluating the utility of an algorithm is also about judging the algorithm's efficiency. The 'random driver' approach to finding a path through a graph may eventually work, but it is unlikely to be much help when driving to Birmingham in time for a meeting. The examples earlier of different algorithms for searching, sorting or other problems provide ample scope for pupils to learn about this. Notice that the curriculum talks here of using logical reasoning: evaluating algorithms should start with thinking about them, rather than by rushing to implement them as code. Can pupils explain to you or to one another why a binary search will be more efficient than a linear search? Can they explain why quicksort gets its name? There's something to be gained by letting pupils implement these algorithms as code and to see for themselves the difference in how long their programs take to complete – bubble sort and quicksort programs in Snap! will have appreciably different run times if given lists of 100 random integers to sort.

As well as evaluating abstractions and algorithms, pupils should also learn to evaluate the digital artefacts that they make. When pupils **develop, create, reuse, revise and repurpose digital artefacts**, they do so **for a given audience**, and evaluation should consider the extent to which they have met the needs of these known users. These needs should feed into the design process from the start, perhaps in terms of specifications or criteria against which the eventual product or prototype might be judged, but also perhaps as the 'user stories' which play a similar role in agile development (see pages 37 - 38), often expressed in the form 'As a \_\_, I want \_\_ so that \_\_.' It's worth giving pupils a genuine experience of developing for actual users at some point in Key Stage 3 – perhaps from their peer group or for parents at the school or for younger children.<sup>(25)</sup> Evaluation in terms of meeting the needs of known users can then involve trying out the product or prototype with these users, observing their interactions and listening to their feedback.

The programme of study also expects attention to be paid to **trustworthiness, design and usability** when using or developing digital artefacts. Evaluating trustworthiness can help develop pupils' logical and critical reasoning, as they come to consider internal and external consistency, logical

flaws in arguments, unsubstantiated claims, vested interests and other forms of bias. Whilst there are perhaps inevitably some subjective elements to evaluating the design of a digital artefact, there are also principles which seem common to much, even if not all, good design: simplicity, symmetry, consistency, proportion, attention to detail, fitness for purpose, honesty, inclusion and sustainability.<sup>(26)</sup> Evaluating usability is about considering how well an artefact meets the needs of its intended users: doing this demands some empathy with those users. Encourage pupils to take an inclusive approach to usability, considering how well an artefact would meet the needs of a diverse group of users – those whose first language isn't English, those who are visually impaired, those for whom fine motor control is difficult or impossible.



## Activities

- Get pupils to do code reviews for one another. Given a problem, pupils should write programs to solve it and add comments to their code. They should then review a solution written by one of their peers, evaluating how well they have solved the problem and providing constructive, critical feedback on their solution.
- Again in programming activities, pupils should be able to create tests for the correctness of their code, determining by hand what output should follow from particular input data and then testing to see whether their code performs correctly.
- Encourage pupils to be constructively critical of websites, software and systems that they use – how might these be improved? Have they found any bugs? In many cases, particularly open source software projects, users can play an important role in software development by submitting bug reports or feature requests.
- Using keyboard-only input, using just a screen reader for output or swapping the language settings for a program into another language would give pupils an insight into the challenges of designing and developing with accessibility in mind.

<sup>25</sup> Some great examples via [www.appsforgood.org/](http://www.appsforgood.org/)

<sup>26</sup> See also [www.vitsoe.com/gb/about/good-design](http://www.vitsoe.com/gb/about/good-design) and [www.gov.uk/design-principles](http://www.gov.uk/design-principles)



Barefoot Computing (2014) *Evaluation*. Available from <http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/evaluation/> (free registration required).

BBC Bitesize (n.d.) *Evaluating solutions*. Available from [www.bbc.co.uk/education/guides/zssk87h/revision](http://www.bbc.co.uk/education/guides/zssk87h/revision)

Bragg, M. (2015) *P v NP*. BBC Radio 4: In Our Time. Available from [www.bbc.co.uk/programmes/b06mtms8](http://www.bbc.co.uk/programmes/b06mtms8)

CASTV (2016) *Simon Peyton Jones on algorithmic complexity*. YouTube. Available from [www.youtube.com/watch?v=ixmbkp0QEDM](http://www.youtube.com/watch?v=ixmbkp0QEDM)

CS Field Guide (2016) *Complexity and tractability*. Available from <http://csfieldguide.org.nz/en/chapters/complexity-tractability.html>

CS Field Guide (2016) *Human computer interaction*. Available from <http://csfieldguide.org.nz/en/chapters/human-computer-interaction.html>

Peyton Jones, S. (2010) *Getting from A to B: Fast route-finding using slow computers*. Microsoft. Available from [www.microsoft.com/en-us/research/video/getting-from-a-b-fast-route-finding-using-slow-computers/](http://www.microsoft.com/en-us/research/video/getting-from-a-b-fast-route-finding-using-slow-computers/)

Programming challenges that are particularly good at encouraging pupils to look for more efficient or elegant solutions: *Project Euler*: <https://projecteuler.net/>, *Cargo Bot* (iPad only) <https://itunes.apple.com/gb/app/cargo-bot/id519690804?mt=8>, *Code Hunt* (Java and C++) [www.codehunt.com/](http://www.codehunt.com/)

Teaching London Computing (n.d.) *Evaluation*. Available from <https://teachinglondoncomputing.org/resources/developing-computational-thinking/evaluation/>

*TryEngineering* (n.d.) Lesson plan introducing ideas of algorithms and complexity. Available from <http://tryengineering.org/lesson-plans/complexity-its-simple>

# How does Software get written?

The national curriculum for computing starts with the ambition that:

**A high-quality computing education equips pupils to use computational thinking and creativity to understand and change the world.**

Whilst the above concepts of computational thinking help with understanding the world, it would be wrong to see them as separate from the processes of computational **doing**, that have resulted in the profound changes to our world through the applications of computer science to digital technology. These approaches can be applied in computing, but, as with the concepts of computational thinking, have wide applications beyond this (see also the discussion of computational thinking practices and perspectives in Brennan and Resnick, 2012).

## Computational doing

### Tinkering

There is often a willingness to experiment and explore in computer scientists' work. Some elements of learning a new programming language or exploring a new system look quite similar to the sort of purposeful play that's seen as such an effective approach to learning in the best nursery and reception classrooms. Tinkering is also a great way to learn about elements of physical computing on platforms such as the BBC micro:bit and the Raspberry Pi.

Open source software makes it easy to take someone else's code, look at how it's been made and then adapt it to your own particular project or purpose. Platforms such as Scratch and TouchDevelop positively encourage users to look at other programmers' work and use this as a basis for their own creative coding.

In class, encourage pupils to experiment with a new piece of software, sharing what they discover about

it with one another, rather than you explaining exactly how it works. Also, look for ways in which pupils can use others' code, from you, their peers or online as a starting point for their own programming projects.

## Creating

Programming is a creative process. Creative work involves both originality and making something of value: typically something that is useful or at least fit for the purpose intended.

Encourage pupils to approach tasks with a creative spirit, and look for programming tasks that allow some scope for creative expression rather than merely arriving at the right answer.

Encourage pupils to reflect on the quality of the work they produce, critiquing their own and others' projects. The process of always looking for ways to improve on a software project is becoming common practice in software development. Look for projects in which artistic creativity is emphasised, such as working with digital music, images, animation, virtual environments or even 3D printing.<sup>(27)</sup>

Creating need not be confined to the screen: there is ample scope at Key Stage 3 to introduce pupils to electronic circuits, microcontrollers, wearable electronics and simple robotics. Platforms such as the BBC micro:bit and the Raspberry Pi make this sort of activity more accessible than ever at school, and there are many extra-curricular opportunities for pupils such as hack days, Coder Dojos<sup>(28)</sup> and Raspberry Jams.<sup>(29)</sup>

## Debugging

Because of its complexity, the code programmers' write often doesn't work as intended.

Getting pupils to take responsibility for thinking through their algorithms and code, to identify and fix errors is an important part of learning to think, and work, like a programmer. It's also something to encourage across the curriculum: get pupils to check through their working in maths or to proofread their stories in English. Ask pupils to debug one another's code (or indeed proofread one another's work), looking for mistakes and suggesting

improvements. There's evidence that learning from mistakes is a particularly effective approach and the process of pupils debugging their own or others' code is one way to do this. A positive attitude towards mistakes as learning opportunities and taking responsibility for fixing them can help develop pupils' resilience and contribute towards a 'growth mindset' (Dweck, 2006; qv Cutts et al., 2010), as Papert observed:

**The question to ask about the program is ... if it is fixable. If this way of looking at intellectual products were generalised to how the larger culture thinks about knowledge and its acquisition, we all might be less intimidated by our fears of 'being wrong'. (Papert, 1980)**

Keep an eye on the bugs that your pupils do encounter, as these can sometimes reveal particular misconceptions that you may need to address.

Debugging is discussed in more detail on [pages 77 - 79](#).

## Persevering

Computer programming is hard. This is part of its appeal – writing elegant and effective code is an intellectual challenge requiring not only an understanding of the ideas of the algorithms being coded and the programming language you're working in, but also a willingness to persevere with something that's often quite difficult and sometimes very frustrating. There's evidence that learning is more effective when there are challenges to overcome, in part because we then have to think more (Bjork and Bjork, 2011).

Carol Dweck's work on 'growth mind-sets' suggests that hard work and a willingness to persevere in the face of difficulties can be key factors in educational outcomes. Encourage pupils to look for strategies they can use when they do encounter difficulties with their programming work, such as working out exactly what the problem is, searching for the solution on Google or Bing, or on Stack Overflow or Stack Exchange, or asking a friend for help.

<sup>27</sup> For a survey of young people's digital making see Quinlan (2015).

<sup>28</sup> <https://coderdojo.com/>

<sup>29</sup> [www.raspberrypi.org/jam/](http://www.raspberrypi.org/jam/)



## Collaborating

Software is developed by teams of programmers and others working together on a shared project. Look for ways to provide pupils with this experience in computing lessons too. Collaborative group work has long had a place in education, and computing should be no different.

Many see ‘pair programming’ as a particularly effective development method, with two programmers sharing a screen and a keyboard, working together to write software (Williams and Kessler, 2000). Typically one programmer acts as the driver, dealing with the detail of the programming, whilst the other takes on a navigator role, looking at the bigger picture.

The two programmers regularly swap roles, so both have a grasp of both detail and big picture. Working in a larger group develops a number of additional skills, with each pupil contributing some of their own particular talents to a shared project. However, it’s important to remember that all pupils should develop their understanding of each part of the process, so some sharing of roles or peer-tutoring ought normally to be incorporated into such activities.

### Software engineering<sup>(30)</sup>

There’s much more to software development than algorithms and coding: the process of developing software has much in common with other engineering disciplines, and so there are some close parallels with design–make–evaluate projects in design and technology on the school curriculum, and it’s through projects such as these that the above approaches of computational doing are perhaps best developed.

The first stage of developing any software project isn’t coding, it’s planning. To plan the project as a whole, and to plan how the computer will be programmed draws on the set of computational thinking concepts discussed above.

Typically, developers need to understand problems or a system before they have any chance of being able to develop some software for this, and that’s likely to draw on processes such as **logical reasoning**,

to identify the relationships between cause and effect, **abstraction**, where they will focus on the key features of the problem or system, leaving others to one side, and **generalisation**, where they might think of other projects which have something in common with the current project, looking to see if there are aspects of the approaches or solutions to these which could be re-used. **Decomposition** is really important for breaking big projects down into manageable tasks, and **algorithmic thinking** is necessary to plan how these will be tackled.

Developers will also need to draw on computational thinking in first designing their programs before they start coding. How formal this design stage is will vary from one development methodology to another, but there’s always some thinking and planning necessary before the actual coding can begin.

## Waterfall

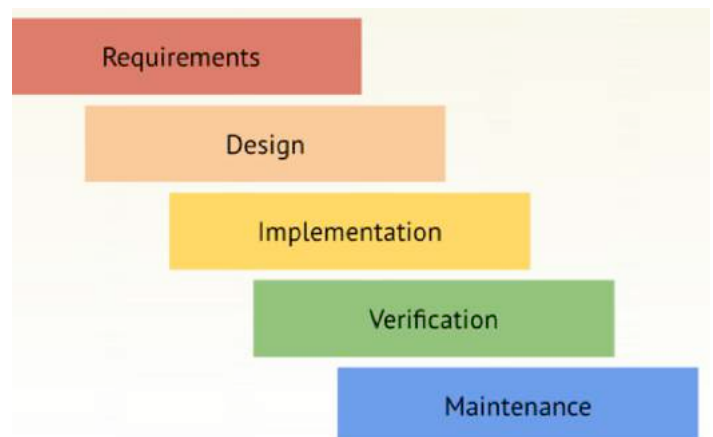


Figure 1.23

In traditional ‘waterfall’ software development, a single path is planned through the project from beginning to end (see Figure 1.23). If a bespoke solution is being developed for a particular client, the project will start with the client working with analysts to specify requirements for what the software needs to do. A more detailed specification can then be worked up, which will include much of the technical planning for how to program a solution, including consideration of systems, language, algorithms and data structures but no actual code. The specification then gets implemented as code in whatever language for whatever system has been decided – often this will be by a team of developers, each weighing in on one or more particular parts of the project in parallel with others. The next stage is to test the

<sup>30</sup> Based on an earlier blog post, <http://milesberry.net/2014/11/software-engineering-in-schools/>

code rigorously, making sure that it has no bugs and that it meets the detail outlined in the specification and the original requirements. There's usually a fifth stage in commercial waterfall development in which the software house undertakes to maintain the program, updating it if necessary to meet changing requirements. Waterfall methods are still used for some big, public sector software deployments.

This approach has something in common with curriculum development – moving from requirements that children should be taught computer science, IT and digital literacy through a detailed specification of programmes of study to implementation through schemes of work, lesson plans and activities to testing and evaluation, with further support there if necessary.

### Iterative development

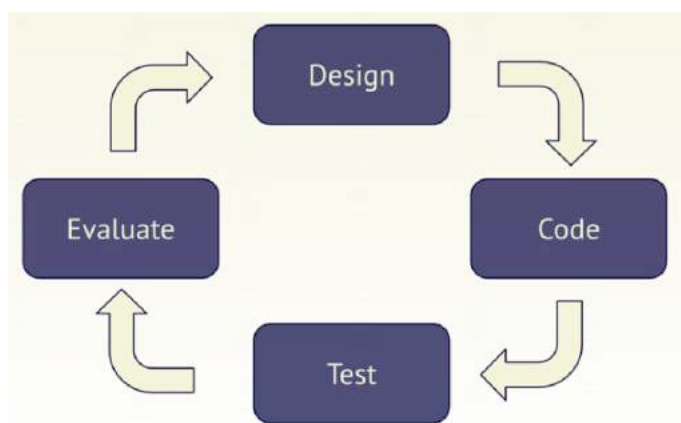


Figure 1.24

In iterative development, the process of designing, coding, testing and evaluating becomes a cycle rather than once and for all (see Figure 1.24). Most modern software development fits into this pattern or a variant of it, hence new versions of software are regularly released which fix bugs that only became apparent once the software was released, or implement new features in response to customer suggestions, technical innovations or market pressures. Often developers will release an early 'beta' version of their software, perhaps to a limited number or quite openly to get help with testing and evaluating the software before committing to an official final release. This is common practice in open source development, where the users of the software are positively encouraged to help with fixing as well as finding bugs or adding code for new features themselves.

There are parallels between the design–code–test–evaluate cycle of iterative development and the plan–teach–assess–evaluate cycle for teaching that many teachers and schools now use routinely (Figure 1.25). Similarly, just as assessment for learning has produced a tight loop between teaching and assessing, so that the results of formative assessment feed directly into how the rest of the lesson or unit of work proceeds, so in iterative development, there's a tight loop between coding and testing – as bugs become apparent in testing, they get fixed through more coding.

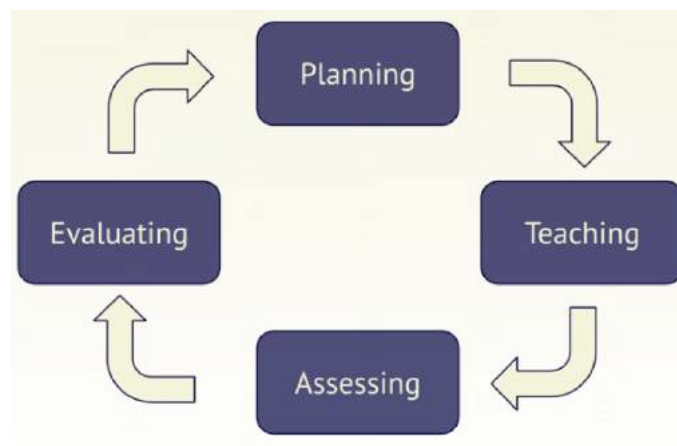


Figure 1.25

### Agile methods

Following a plan	Responding to change
Contract negotiation	Customer collaboration
Comprehensive documentation	Working software
Processes and tools	Individuals and interactions

Figure 1.26

Whilst recognising the importance of things such as planning, agreeing requirements and producing documentation, agile software development moves the focus of the effort to producing working, usable code, typically much earlier in the process (see Figure 1.26). It also emphasises the importance of collaboration with users and responsiveness to change.<sup>(31)</sup> Whilst by no means universally accepted, the effectiveness of agile methods in getting to a

31 <http://agilemanifesto.org/>

‘minimum viable product’ and then developing this further in response to changing needs and rapidly developing technologies has made this approach popular with many working in technology based start-ups, as well as for developing new online tools and apps for tablets or smartphones.

The emphasis in agile methods on individuals and interactions, collaborating with customers and responsiveness to change might put us in mind of the ‘child-centred education’ pedagogies of an earlier generation, but perhaps even today there’s scope in some computing lessons for supporting and encouraging pupils as they pursue individual, independent projects or their own lines of investigative enquiry.

### So which approach should we use in class?

One of the aims of the programme of study is that pupils:

**can analyse problems in computational terms, and have repeated practical experience of writing computer programs in order to solve such problems**

At Key Stage 2, pupils should already have had some experience of working on larger software projects than just learning the key programming concepts of sequence, selection and repetition:

**design, write and debug programs that accomplish specific goals**

At Key Stage 3, pupils are taught to:

**design and develop modular programs that use procedures or functions**

and to:

**create, reuse, revise and repurpose digital artefacts for a given audience, with attention to trustworthiness, design and usability**

Which allow plenty of scope for larger software development projects alongside shorter programming tasks.

The way you go about this though is up to you! Choose the approach which would work best with your pupils, and for the particular project you (or they) have in mind.

It’s perhaps best to let pupils have some experience of all three of these methodologies. For some programming projects, you may only have time to work through from planning to debugging and evaluation once, in which case guiding pupils through the waterfall process may make most sense. Other times, it would be worth taking a more iterative approach, getting pupils to look for ways in which they could add further features to their programs, improve the user interface or refine their algorithms, as well as emphasising repeated coding, testing, debugging as part of the programming process itself.

Pupils who find that they really enjoy coding and choose to do this independently outside of formal lessons might often adopt an approach with much in common with agile methods – there is anecdotal evidence that this is often the case for those contributing to the Scratch community or pursuing their own project ideas on the Raspberry Pi. You might like to look for ways to facilitate this approach in curriculum time too: perhaps setting very open challenges to pupils, for example ‘make an educational game’, providing support and challenge as needed as well as encouraging pupils to help support one another as they rise to meet the challenge. There is anecdotal evidence that girls seem to find programming projects where there’s a clear purpose and scope for creativity more engaging than relatively closed, abstract coding challenges such as ‘implement bubble sort’.



### Further resources

*Apps for Good* (n.d.) Available from [www.appsforgood.org/](http://www.appsforgood.org/)

Bagge, P. (2015) *Eight steps to promote problem solving and resilience and combat learnt helplessness in computing*. Available from <http://philbagge.blogspot.co.uk/2015/02/eight-steps-to-promote-problem-solving.html>

Barefoot Computing (2014) *Computational thinking approaches*. Available from <http://barefootcas.org.uk/barefoot-primary-computing-resources/computational-thinking-approaches/> (free, but registration required).

Briggs, J. (2013) *Programming with Scratch software: The benefits for year six learners*. MA dissertation.

Bath Spa University.

Brooks, F.P. (1975) *The mythical man-month*. Reading, MA: Addison-Wesley.

CS Field Guide (n.d.) *Software engineering*. Available from <http://csfieldguide.org.nz/en/chapters/software-engineering.html>

DevArt: *Art Made with Code* (n.d.) Available from <https://devart.withgoogle.com/>

Dweck, C. (2012) *Mindset: How you can fulfil your potential*. London: Hachette.

Harel, I. and Papert, S. (1991) *Constructionism*. New York, NY: Ablex Publishing Corporation.

Education Endowment Foundation (n.d.) *Teaching and learning toolkit*. Available from <http://educationendowmentfoundation.org.uk/toolkit/>

Kafai, Y. and Burke, Q. (2014) *Connected code: Why children need to learn programming*. Boston, MA: MIT Press.

Peha, S. (2011) *Agile schools: How technology saves education (just not the way we thought it would)*. InfoQ. Available from [www.infoq.com/articles/agile-schools-education](http://www.infoq.com/articles/agile-schools-education)

Philbin, C.A. (2015) *Adventures in Raspberry Pi*. Hoboken, NJ: John Wiley & Sons.

## References

Alexander, C., Ishikawa, S. and Silverstein, M. (1977) *A pattern language: Towns, buildings, construction*. Oxford: OUP.

Aristotle (350 BCE, translated by Smith, R. 1989) *Prior analytics*. Indianapolis, IN: Hackett.

Berlekamp, E.R., Conway, J.H. and Guy, R.K. (2004) *Winning ways for your mathematical plays (volume 4, 2nd edition)*. Boca Raton, FL: Taylor and Francis Group.

Bjork, E.L. and Bjork, R.A. (2011) Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. In: *Psychology and the real world: Essays illustrating fundamental contributions to society*. 56–64.

Boole, G. (2003) [1854] *An investigation of the laws of thought*. Amherst, NY: Prometheus Books.

Brennan, K. and Resnick, M., 2012, April. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada* (pp. 1-25).

Brin, S. and Page, L. (1998) The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems* 30:1-7. 107–117.

Christian, B. and Griffiths, T. (2016) *Algorithms to live by: The computer science of human decisions*. London: William Collins.

Csizmadia, A., Curzon, P., Dorling, M., et al. (2015) *Computational thinking: A guide for teachers*. Cambridge: Computing at Schools. Available from <http://community.computingatschool.org.uk/files/6695/original.pdf>

Curzon, P. (2015) *Computational thinking: Puzzling tours*. London: Queen Mary University of London.

Cutts, Q., Cutts, E., Draper, S., et al. (2010) Manipulating Mindset to Positively Influence Introductory Programming Performance. In: *SIGCSE '10 Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. Milwaukee, USA, 10–13 March 2010. 431–443.



- Dean, J. and Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51 (1), pp.107-113.
- DfE (2013) National curriculum in England: *Computing programmes of study*. London: DfE
- Dweck, C.S. (2006) *Mindset*. New York: Random House.
- Gamma, E., Helm, R., Johnson, R., et al. (1994) *Design patterns: Elements of reusable object-oriented software*. Boston, MA: Addison Wesley.
- Hoare, C.A.R. (1961) Algorithm 64: Quicksort. *Comm.ACM.*, 4 (7). 321.
- Knuth, D.E. (1992) *Literate programming*. California: Stanford University Center for the Study of Language and Information.
- Knuth, D.E. (1997) *The art of computer programming* (volume 1: Fundamental algorithms). Boston, MA: Addison Wesley
- Laurillard, D. (2012) *Teaching as a design science*. Abingdon: Routledge.
- Michaelson, G. (2015) Teaching programming with computational and informational thinking. *Journal of Pedagogic Development* 5 (1). 51–66.
- Michaelson, G. (2016) Some reflections on the state we're in. *Switched On*, Spring 2016, 22–23.
- Miller, G.L. (1976) Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences* 13 (3). 300–317.
- Newman, M.E.J. (2001) The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences of the United States of America*, 98 (2). 404–409.
- Papert, S. (1980) *Mindstorms: Children, computers and powerful ideas*. New York: Basic Books.
- Peng, D. and Dabek, F. (n.d.) *Large-scale incremental processing using distributed transactions and notifications*. Available from [www.usenix.org/legacy/event/osdi10/tech/full\\_papers/Peng.pdf](http://www.usenix.org/legacy/event/osdi10/tech/full_papers/Peng.pdf)
- Quinlan, O. (2015) *Young digital makers*. London: Nesta.
- Rajlich, V. and Wilde, N. (2002) The Role of Concepts in Program Comprehension. In: *Proceedings of the 10th International Workshop on Program Comprehension*. 271–278.
- Russell, B. (1946) *A history of western philosophy*. Crows Nest, NSW: George Allen and Unwin.
- Sorva, J. (2013) Notional machines and introductory programming education. *ACM Transactions on Computing Education* 13 (2). 8:1-8:31
- Williams, L.A. and Kessler, R.R. (2000) All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM*, 43 (5). 108–114.
- Wing, J. (2008) Computational thinking and thinking about computing. *Phil. Trans. R. Soc. A.*, 366:1881. 3717–3725.
- Wing, J. (2010) *Computational thinking: What and why?* Available from [www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf](http://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf)
- Wirth, N. (1976) *Algorithms + data structures = programs*. Upper Saddle River, NJ: Prentice-Hall.