COMPUTING AT SCHOOL SCOTLAND

EDUCATE · ENGAGE · ENCOURAGE

Part of BCS, The Chartered Institute for IT

QuickStart Computing Scotland

Subject Knowledge covering the transition from Primary to Secondary

SUPPORTED BY

Foreword

In 2017, Education Scotland revised the computing science content within the Broad General Education, with significant input from Computing At School Scotland (CAS Scotland), to reflect the developing worldwide understanding of the importance of the subject.

We recognise that there is a large increase in the computing science content to be covered in both primary and early secondary compared to the old Experiences and Outcomes (Es & Os) for computing science. In secondary schools, with our current computing teacher numbers, many teachers from related subjects may be asked to deliver teaching at the lower levels. It is important, then, that materials are available for such teachers to enable them to prepare for these classes.

This *QuickStart* resource is the first to support early secondary teaching, and it particularly focusses on the necessary subject knowledge for teachers. It is not full of teaching activities and pedagogical advice – we will shortly be extending the primary teacher guide available at http://teachcs.scot, developed by CAS Scotland, to include these. That guide explains well the particular structure of the new Es & Os, and so is essential reading too.

This resource is an adjusted version of one originally developed by CAS for the English curriculum. Such sharing between the nations is valuable, and should be effective since the subject knowledge required for both the English and Scottish curricula is similar. We have largely removed direct reference to the English curriculum, although connections may still be apparent. Most importantly, the structuring of the two curricula is significantly different, and so the chapter structure in this resource does not readily follow the Scottish curriculum. For this reason, we have provided a mapping from the Es & Os, and the Benchmarks, to the relevant sections of the resource, which will help you to find the descriptions and explanations you need. This appears right after the main Contents page.

We are acutely aware of the huge effort it has taken in recent years to teach the new computing science qualifications, and also how much it will take to teach to the new Es & Os. Be assured that the best international research evidence in computing science education has been incorporated into these Es & Os. As such, we are confident that the combination of foundational work in the early years and primary, as well as the overall structure and extended time in secondary, will enable all pupils to succeed in the subject, correcting the imbalance in background and gender to which our subject has long been prone.

A word about sponsorship. The *QuickStart* project was funded by Microsoft, with matched funding from the English Department for Education, and further funding from Education Scotland, and it is heartening to see such tangible support for teachers, both from business and from government. I would like to thank them warmly and to emphasise that the *QuickStart* resources were developed for teachers by a Computing At School working group, without the direct influence of the sponsors.

It's an exciting time for computing science! We are a subject coming of age, with great relevance to every one of our pupils. We hope very much that you find this *QuickStart* resource of value in underpinning your delivery of great experiences for your pupils.

Quintin Cutts
Chair, Computing At School Scotland

Acknowledgements

Every effort has been made to trace copyright holders and obtain their permission for the use of copyright materials. The author and publisher will gladly receive information enabling them to rectify any error or omission in subsequent editions.

Although every effort has been made to ensure that website addresses are correct at time of going to press, Computing At School cannot be held responsible for the content of any website mentioned. It is sometimes possible to find a relocated web page by typing in the address of the home page for a website in the URL window of your browser.

© Crown Copyright 2017. Published 2017

Author: Miles Berry. Adjusted for Scotland: Bill Sargent

To reference this work, please use the following citation: Berry, M. (2017) QuickStart Computing –Subject Knowledge Enhancement for Scottish Secondary Teachers. Swindon: BCS.

This content is free to use under the Open Government Licence v3.0.
A catalogue record for this title is available from the British Library. ISBN: 978-1-78017-441-9

We would like to thank the following people for their invaluable advice as members of the Academic Standards Committee:

- Professor Jeff Magee FREng, Dean of the Faculty of Engineering, Imperial College
- Professor Simon Peyton-Jones FRS, Principal Researcher, Microsoft Research Cambridge, and Chair of CAS
- Professor Paul Curzon, School of Electronic Engineering and Computer Science, Queen Mary University of London
- Professor Michael Kölling, School of Computing, King's College London
- Dr Nick Cook, School of Computing Science, the University of Newcastle
- Roger Davies, CAS Master Teacher, Queen Elizabeth School, Kirkby Lonsdale
- Chidi Iweha, CAS Master Teacher, Ernest Bevin College, London
- Peter Marshman, CAS Master Teacher, Leighton Park School, Reading
- Dr Bill Mitchell, Director of Education, BCS, the Chartered Institute for IT

Computing At School is grateful to the following reviewers and contributors:

CAS Barefoot Computing, CAS Computational Thinking Working Group, Sunrise Setting Ltd, Burville Riley Partnership, Tomcat Design, Julia Briggs, Lee Goss, Miles Ellison, Department for Education, Microsoft and all of the parties that have kindly made their work available to share within this document.

Introduction

Digital technology has come to play a huge part in the lives of individuals and our society: there seems hardly any sphere of work, leisure or study that has not been transformed by computers, the internet or smartphones. The changes we've seen in our lifetimes show no signs of slowing down. If we take seriously our responsibilities to prepare the next generation for the opportunities, responsibilities and experiences of their later lives, then that must involve empowering them not only to use the technologies that will be part of their future, but also to understand these technologies, and perhaps develop them further still.

Computing is about more than using or understanding technologies though. The principles of computer science which lie at the foundation of digital technology have wide applications beyond this. An understanding of these principles (things like logic, computability and the properties of information) certainly helps to make sense of current technology and looks likely to be necessary for a grasp of future technology, but it also offers unique insights into the nature of the world.

In thinking about the purpose of education, surely part of this is that school should help young people develop an understanding of the world in which they find themselves, and equip them to make a difference to that world. This vision is reflected in the opening sentence of the 2014 English computing curriculum:

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

Reflecting the advice of The Royal Society's *Shut down or restart?* report (Furber, 2012), the Scottish curriculum includes two distinct but inter-related aspects of the subject: computing science, and digital literacy. These are perhaps best thought of as the **foundations** of the discipline, and its **applications** to getting useful things done, and the wider personal, cultural, ethical and societal **implications** of the subject. Computing science provides a unique way of looking at the world, like mathematics and the natural sciences; digital literacy is a creative discipline with much in common with expressive arts, but also might include links with the personal and social aspects of health and wellbeing, social studies and even religious and moral education.

The computing science and digital literacy experiences and outcomes are unashamedly ambitious. For a school to deliver on the promise of these experiences and outcomes and do the best that it can for its pupils, there needs to be enough time to teach the programmes of study, plus good resources, robust connectivity, effective pedagogy and crucially, knowledgeable, enthusiastic teachers.

With primary teachers, CAS (Computing At School) developed CPD (continuing professional development) programmes such as Barefoot Computing[1] and Primary QuickStart Computing[2] to help address the subject knowledge gap, and alongside initiatives such as PLAN C, these have generally been regarded as effective. At secondary level, subject knowledge professional development has unsurprisingly often focussed on those new to teaching exam courses with relatively little in place to equip teachers to rise to the challenge of the Level 3 programme of study.

This new *QuickStart* guide has therefore been developed with Level 3 teachers particularly in mind. I have taken as a starting point the structure of the primary guide and have included much of the content from that to provide a starting point for getting to grips with computing in lower secondary schools. But this guide goes much further, covering, in more or less depth, the subject knowledge needed to teach the whole of the Level 3 programme of study.

I have broken the curriculum up into six, unequal but inter-related parts: computational thinking, programming, computer systems, networks, productive and creative IT, and the safe and responsible use of technology. In each chapter, I discuss the ideas that the curriculum is built on, weaving in some thoughts on how these can best be taught. I have added in a good scattering of footnotes pointing to original sources and further examples, and each section includes a few suggestions for relevant teaching activities and a list

1 http://barefootcas.org.uk/
2 http://primary.quickstartcomputing.org/

of further resources for those who want to learn more or draw on alternative perspectives.

This **is not** a guide on how to plan, teach or assess computing, important as these elements are. There's good advice on these things in the original secondary *QuickStart* guide,[3] as well as CAS's guidance notes on the secondary computing curriculum[4] and the more-recent guide to computational thinking.[5] Those looking to go rather further in their understanding of computing than an introduction like this can cover, might be interested in pursuing the BCS Certificate in Computer Science teaching,[6] enrolling in one or more of the excellent online computing MOOCs (massive, open, online courses)[7] or attending CAS's new Tenderfoot Computing CPD programme.[8]

Miles Berry,
Roehampton,
November 2017.

References

Furber, S. (2012) *Shut down or restart?* The way forward for computing in UK schools. London: The Royal Society.

3 www.quickstartcomputing.org/secondary/
4 www.computingatschool.org.uk/data/uploads/cas_secondary.pdf
5 https://community.computingatschool.org.uk/files/6695/original.pdf
6 www.computingatschool.org.uk/certificate
7 For example, Harvard's CS50: www.edx.org/course/introduction-computer-science-harvardx-cs50x
8 www.computingatschool.org.uk/custom_pages/56-tenderfoot

Contents

Introduction 1
References 2

Computational Thinking 9

Logical Reasoning 12
Algorithms 16
Decomposition 24
Abstraction 27
Patterns and Generalisation 31
Evaluation 34
How does Software get written? 38
References 43

Programming 45

How do you program a Computer? 48
Visual Programming Languages 51
Text-based Programming Languages 53
What's inside a Program? 59
Data Structures 73
Can we fix the Code? 81
References 84

Systems 85

Binary 86
Logic Circuits 100
Hardware Components 104
Software Components 108
Physical Computing 110
References 116

Computer Networks 117

How does the Internet work? 117
What can you do with the Internet? 120
What is the World Wide Web? 122
How do you make a Web Page? 124
How does a Search Engine work? 127
References 128

Productivity and Creativity 129

What can pupils do with Data? 135
How can we best support Collaboration? 137
References 142

Safe and Responsible Use 143

Privacy, Security and Identity 148
References 157

Benchmarks – Third Level Technologies				
Curriculum Organisers		Experiences and Outcomes for planning learning, teaching and assessment	Benchmarks to support practitioners' professional judgement	Page reference
Digital Literacy	Using digital products and services in a variety of contexts to achieve a purposeful outcome	I can explore and use the features of a range of digital technologies, integrated software and online resources to determine the most appropriate to solve problems. **TCH 3-01a**	• Uses the most appropriate applications and software tools to capture, create and modify text, images, sound, and video to present and collaborate. • Demonstrates an understanding of file handling, for example, uploading, downloading, sharing and permission-setting, for example within Glow or other platforms.	129 - 135 154 - 157
	Searching, processing and managing information responsibly	Having used digital technologies to search, access and retrieve information, I can justify my selection in terms of validity, reliability, and have an awareness of plagiarism. **TCH 3-02a**	• Gathers and combines data and information from a range of sources to create a publication, presentation or information resource. • Uses applications to analyse data and identify trends / make predictions based on source data. • Demonstrates efficient searching techniques, for example using 'and', 'or', 'not'.	129 - 135 135 - 137 127 - 128
	Cyber resilience and internet safety	I can keep myself safe and secure in online environments and I am aware of the importance and consequences of doing this for myself and others. **TCH 3-03a**	• Demonstrates an understanding of the legal implications and importance of protecting their own and others' privacy when communicating online. • Evaluates online presence and identifies safeguards. • Presents relevant ideas and information to explain risks to safety and security of their personal devices and networks, including encryption. • Applies appropriate online safety features when becoming involved with online communities such as online gaming, chat rooms, forums and social media. • Demonstrate an understanding of different cyber threats, for example, viruses, phishing, identity theft, extortion and sextortion. • Demonstrates understanding of device security including personal and domestic devices.	143 - 157 for all aspects here

Benchmarks highlighted in red are covered in the materials on the pages indicated. Benchmarks in blue are not covered in any detail by the materials.

Benchmarks – Third Level Technologies				
Curriculum Organisers		Experiences and Outcomes for planning learning, teaching and assessment	Benchmarks to support practitioners' professional judgement	Page reference
Computing Science	Understanding the world through computational thinking	I can describe different fundamental information processes, and how they communicate and can identify their use in solving different problems. **TCH 3-13a** I am developing my understanding of information and can use an information model to describe particular aspects of a real-world system. **TCH 3-13b**	• Recognises and describes information systems with communicating processes which occur in the world around them. • Explains the difference between parallel processes and those that communicate with each other. • Demonstrates an understanding of the basic principles of compression and encryption of information. • Identifies a set of characteristics describing a collection of related items that enable each item to be individually identified. • Identifies the use of common algorithms such as sorting and searching as part of larger processes.	106 - 108 96 - 100 (compression) 151 - 153 (encryption) 75 - 81 19 - 21
	Understanding and analysing computing technology	I understand language constructs for representing structured information. **TCH 3-14a** I can describe the structure and operation of computing systems which have multiple software and hardware levels that interact with each other. **TCH 3-14b**	• Understands that the same information could be represented in more than one representational system. • Understands that different information could be represented in exactly the same representation. • Demonstrates an understanding of structured information in programs, databases or web pages. • Describes the effect of mark-up language on the appearance of a webpage, and understands that this may be different on different devices. • Demonstrates an understanding of the von Neumann architecture and how machine code instructions are stored and executed within a computer system. • Reads and explains code extracts including those with variables and data structures. • Demonstrates an understanding of how computers communicate and share information over networks, including the concepts of sender, receiver, address and packets. • Understands simple compression and encryption techniques used in computing technology.	90 - 100 90 - 100 75 - 81 123 - 126 48 - 49 73 - 81 117 - 121 96 - 100 (compression) 151 - 153 (encryption)
	Designing, building and testing computing solutions	I can select appropriate development tools to design, build, evaluate and refine computing solutions based on requirements. **TCH 3-15a**	• Designs and builds a program using a visual language combining constructs and using multiple variables. • Represents and manipulates structured information in programs, or databases; for example, works with a list data structure in a visual language, or a flat file database. • Interprets a problem statement, and identifies processes and information to create a physical computing and/or software solution. • Can find and correct errors in program logic. • Groups related instructions into named subprograms (in a visual language). • Writes code in which there is communication between parallel processes (in a visual language). • Writes code which receives and responds to real-world inputs (in a visual language). • Designs and builds web pages using appropriate mark-up languages.	59 - 73 75 - 78 38 - 43 81 - 83 68 - 73 110 - 116 123 - 126

Benchmarks – Fourth Level Technologies				
	Curriculum Organisers	Experiences and Outcomes for planning learning, teaching and assessment	Benchmarks to support practitioners' professional judgement	Page reference
Digital Literacy	Using digital products and services in a variety of contexts to achieve a purposeful outcome	I can select and use digital technologies to access, select relevant information and solve real-world problems. **TCH 4-01a**	• Demonstrates an understanding of how digital literacy will impact on their future learning and career pathways. • Consistently uses a range of devices and digital software and applications and services to share, create, collaborate effectively and publish digital content online.	137 - 141
	Searching, processing and managing information responsibly	I can use digital technologies to process and manage information responsibly and can reference sources accordingly. **TCH 4-02a**	• Gathers, evaluates and combines data and information from a range of sources to create a publication, presentation or information resource. • Evaluates applications to analyse data and identify trends / make predictions based on source data. • Evaluates efficient searching techniques, for example using 'and', 'or','not'.	129 - 135 135 - 137 127 - 128
	Cyber resilience and internet safety	I can explore the impact of cyber-crime for business and industry and the consequences this can have on me. **TCH 4-03a**	• Demonstrates understanding of how industry collects and uses personal data ethically and how this relates to data security legislation. • Demonstrates understanding of how cyber security breaches in industry can impact on individuals. • Evaluates the digital footprint of industry and identifies good practice. • Identifies the main causes of security breaches in industry. • Demonstrates understanding of safe disposal of data and devices.	148 - 157 148 - 157

Benchmarks – Fourth Level Technologies

	Curriculum Organisers	Experiences and Outcomes for planning learning, teaching and assessment	Benchmarks to support practitioners' professional judgement	Page reference
Computing Science	Understanding the world through computational thinking	I can describe in detail the processes used in real-world solutions, compare these processes against alternative solutions and justify which is the most appropriate. **TCH 4-13a** I can informally compare algorithms for correctness and efficiency. **TCH 4-13b**	• Identifies the transfer of information through complex systems involving both computers and physical artefacts, for example, airline check-in, parcel tracking and delivery.	
			• Describes instances of human decision making as an information process, for example, deciding which check-out queue to pick, which route to take to school, how to prepare family dinner / a school event.	
			• Compares alternative algorithms for the same problem and understands that there are different ways of defining "better" solutions depending on the problem context, for example, is speed or space more valuable in this context?	36 - 38
	Understanding and analysing computing technology	I understand constructs and data structures in a textual programming language. **TCH 4-14a** I can explain the overall operation and architecture of a digitally created solution. **TCH 4-14b** I understand the relationship between high level language and the operation of computer. **TCH 4-14c**	• Understands basic control constructs such as sequence, selection, repetition, variables and numerical calculations in a textual language.	59 - 73
			• Demonstrates an understanding of how visual instructions and textual instructions for the same construct are related.	59 - 73
			• Identifies and explains syntax errors in a program written in a textual language.	81 - 83
			• Demonstrates an understanding of representations of data structures in a textual language.	73 - 78
			• Demonstrates an understanding of how computers represent and manipulate information in a range of formats.	78 - 81
			• Demonstrates an understanding of program plans expressed in accepted design representations, for example pseudocode, storyboarding, structure diagram, data flow diagram, flow chart.	17
			• Demonstrates an understanding of the underlying technical concepts of some specific facets of modern complex technologies, for example, online payment systems and sat-nav.	
			• Demonstrates an understanding that computers translate information processes between different levels of abstraction.	108 - 110
	Designing, building and testing computing solutions	I can select appropriate development tools to design, build, evaluate and refine computing solutions to process and present information whilst making reasoned arguments to justify my decisions. **TCH 4-15a**	• Analyses problem specifications across a range of contexts, identifying key requirements.	
			• Writes a program in a textual language which uses variables and constructs such as sequence, selection and repetition.	59 - 73
			• Creates a design using accepted design notations, for example, pseudocode storyboarding, structure diagram, data flow diagram, flow chart.	17
			• Develops a relational database to represent structured information.	
			• Debugs code and can distinguish between the nature of identified errors e.g. syntax and logic.	81 - 83
			• Writes test and evaluation reports.	
			• Can make use of logical operators – AND, OR, NOT.	13 - 16
			• Writes a program in a textual language which uses variables within instructions instead of specific values, where appropriate.	59 - 73
			• Designs appropriate data structures to represent information in a textual language.	73 - 77
			• Selects an appropriate platform on which to develop a physical and/or software solution from a requirements specification.	
			• Compares common algorithms, for example, those for sorting and searching, and justify which would be most appropriate for a given problem.	18 - 21
			• Design and build web pages which include interactivity.	

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).

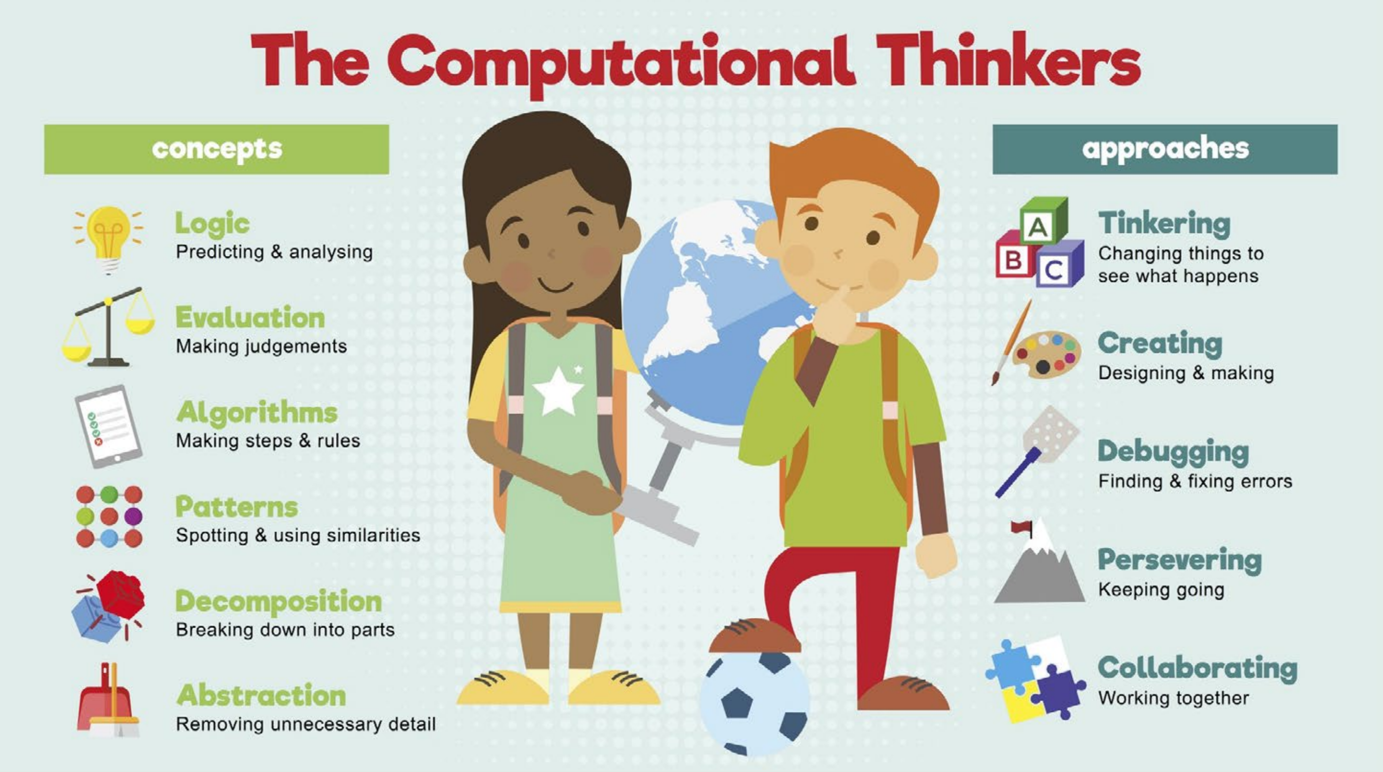
Computational thinking is not thinking **about** computers or **like** computers. Computers don't think for themselves. Not yet, at least!

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

- logical reasoning: predicting, analysing and explaining (see pages 12 - 16);
- algorithms: making steps and rules (see pages 16 - 24);
- decomposition: breaking problems or systems down into parts (see pages 24 - 27);
- abstraction: managing complexity, sometimes through removing unnecessary detail (see pages 27 - 31);
- generalisation:[1] spotting and using patterns and similarities (see pages 31 - 34);
- evaluation: making judgements (see pages 34 - 38).

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):

1 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).



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 goes 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 focussing 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 Benchmarks for Technologies have computational thinking as one of the key curricular organisers from Early Years to Level 4:

> **Understanding the world through computational thinking (Education Scotland 2017)**

At all levels, one element of very good practice is:

> **As their digital literacy becomes more sophisticated they embed computation to solve problems.**

Whilst programming (see pages 45 - 84) 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

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.computingatschool.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.computingatschool.org.uk/files/3517/original.pdf

Google for Education (n.d.) *Exploring computational thinking.* Available from 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 is the intersection of the two sets is where both condition X and condition Y are satisfied (Figure 1.2):

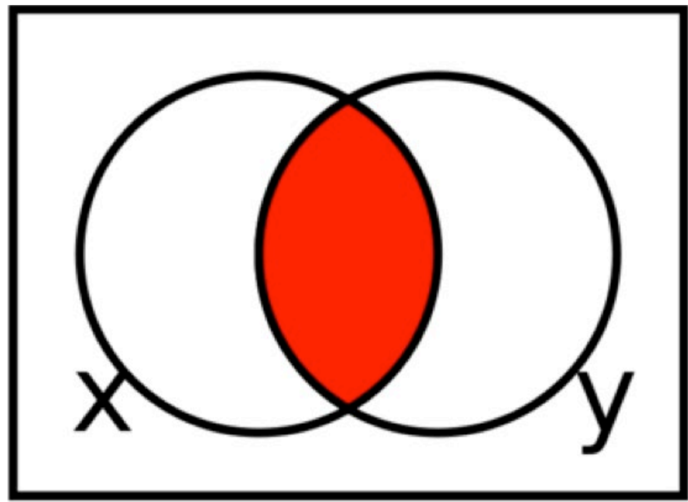


Figure 1.2

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

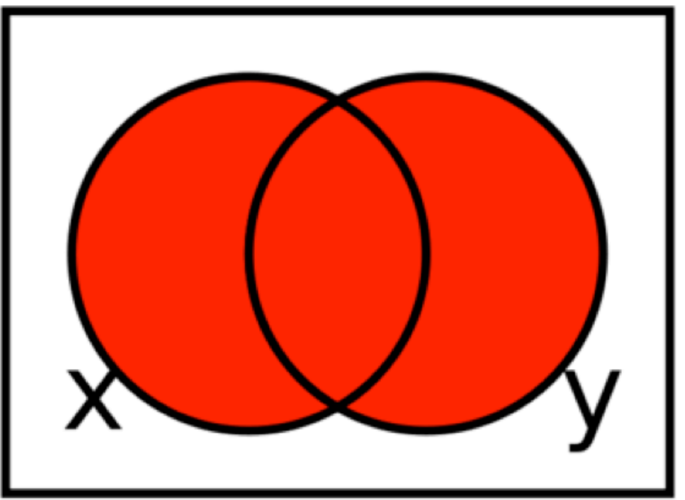


Figure 1.3

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

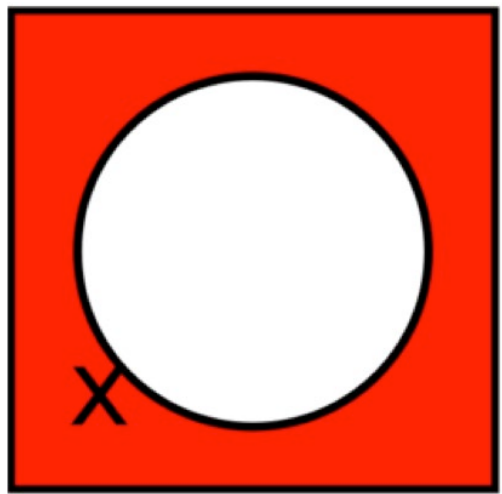


Figure 1.4

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 100 - 103). 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.

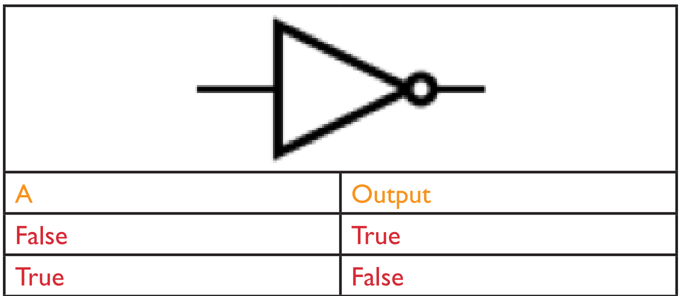
AND:

A	B	Output Q
False	False	False
False	True	False
True	False	False
True	True	True

OR:

A	B	Output Q
False	False	False
False	True	True
True	False	True
True	True	True

NOT:



A	Output
False	True
True	False

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 used in quickly isolating particular parts of a byte or word.

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 chosen programming languag 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 39 & 81 - 83) or in 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, pupils at Level 1 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. Pupils at Level 2 are expected to 'use logical reasoning to explain how some simple algorithms work and to detect and correct errors in algorithms and programs'.

At Levels 3 and 4, 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]'.

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.

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

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/

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

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

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

PhiloComp.net (n.d.) Website highlighting the strong links between philosophy and computing. Available from 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.

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.[2]

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 between places on a road network, underground stations, states of a Rubik's Cube or film stars who have acted alongside one another.[3]

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 out the steps or rules of an algorithm 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 flow chart, such as in Figure 1.5:

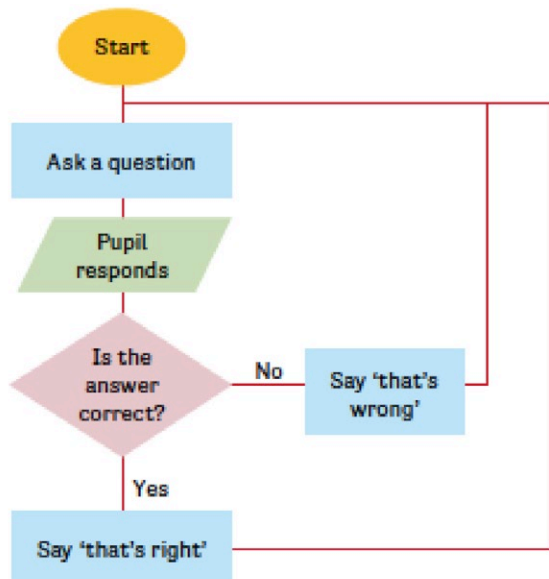


Figure 1.5 An example of a flow chart

Algorithms can also be expressed in pseudocode, which is perhaps best thought of as a halfway 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.

2 See https://en.wikipedia.org/wiki/Shortest_path_problem
3 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 at Level 1 to construct an algorithm to solve a task. 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

Level 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 being able to use 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 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 for themselves, perhaps expressed as flow charts or pseudocode, 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 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 being like 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 is 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 a binary search.[4]

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):

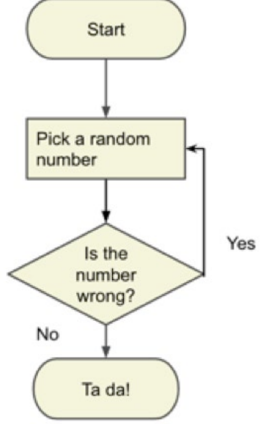


Figure 1.7 Random search

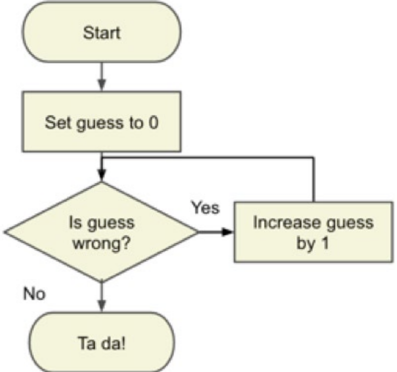


Figure 1.8 Linear search

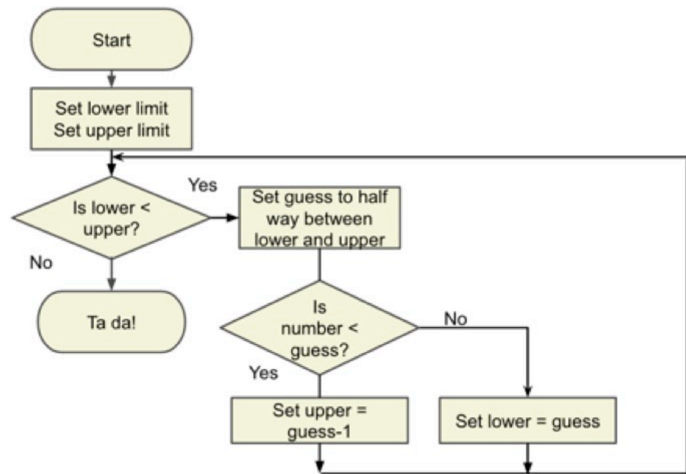


Figure 1.9 Binary search

After only a few goes at the game, or perhaps just by reasoning logically about the flow charts here, it should be evident that the first two 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 an average of 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: 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 between 64 and 127. We 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/zFenJJtAEzE?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 68); sorting a list of numbers (see quicksort and merge sort on page 21); and parsing algebraic expressions, computer code or even natural language.[5]

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 unsorted 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 page 127 - 128) 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![6] 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 ten things?[7]

5 See https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms
6 See the Harvard CS50 demonstration at 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 items before the end, to get the third largest; then three items before the end, and so on until there's only one thing left.
- The list is now sorted.

Try it yourself![8] 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 ten things?[9]

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 70). 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![10] 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 ten things into order?[11]

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: (4x1 comparisons)	1 3 – 1 4 – 5 9 – 2 7
Sorting quads: (2x3 comparisons)	1 1 3 4 – 2 5 7 9
Sorting all eight: (1x7 comparisons)	1 1 2 3 4 5 7 9

Again, to really get a feel for this, you need to try it for yourself![12] 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
9 Also 45.
10 See the Harvard CS50 demonstration at www.youtube.com/watch?v=aQiWF4E8flQ
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

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 50x20=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).[13] 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 it 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. It's a bit slow, but it would get there, although we would have 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 its multiples – 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.

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	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~~	~~100~~

This is a really nice programming challenge too.[14]

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

13 qv Simon Peyton Jones for CAS TV: https://youtu.be/ixmbkp0QEDM?t=12m30s
14 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.[15]

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 x 2r board;
- see if it lands in the inscribed circle radius r.

The proportion of darts thrown, that land in the circle, should be equivalent to the area of the circle divided by the area of the board, which allows you to estimate pi! So, in programmatic terms:

- Choose two random numbers, x,y in -r to r.
- See if x2 + y2 is less than r2. If so, the dart landed in the circle.
- Keep doing this, keeping track of how many landed inside and how many outside.

```
set goes to 0
set hits to 0
repeat 1000000
    go to x: (pick random -180 to 180) y: (pick random -180 to 180)
    if (distance to Centre) < 180 then
        change hits by 1
    change goes by 1
set pi to (4 * (hits / goes))
```

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 34).
- There's much scope here for using 'unplugged' activities, i.e. 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 34).

15 See 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

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/

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

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 is 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, which are assembled to make the finished product, each under the control of the operating system and applications.

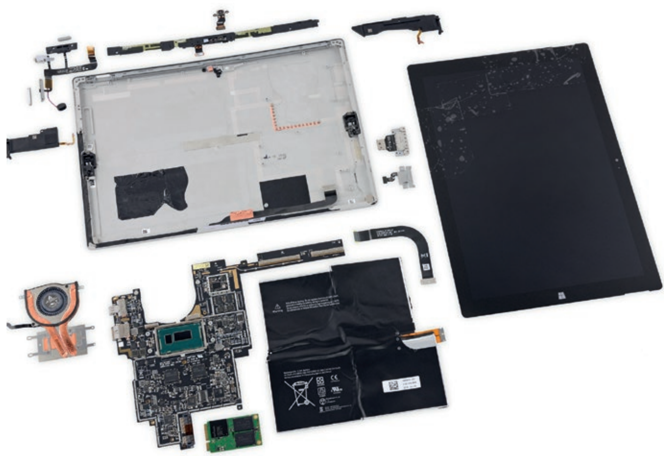


Figure 1.11 A tablet can be broken down (decomposed) into smaller components

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 into years and subjects, and 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 'Simplify problems by breaking them down into smaller more manageable parts' (Education Scotland 2017).

At Level 3, there is an expectation that pupils will use modularity in their programming, using subroutines, procedures or functions (see page 67). 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.

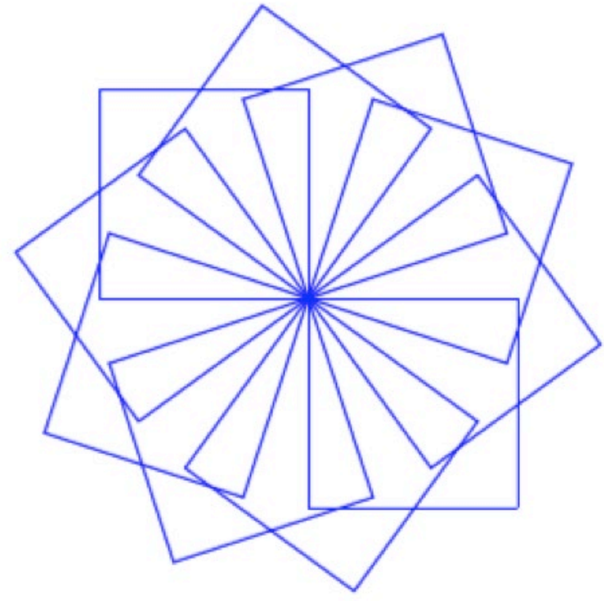


Figure 1.12

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.

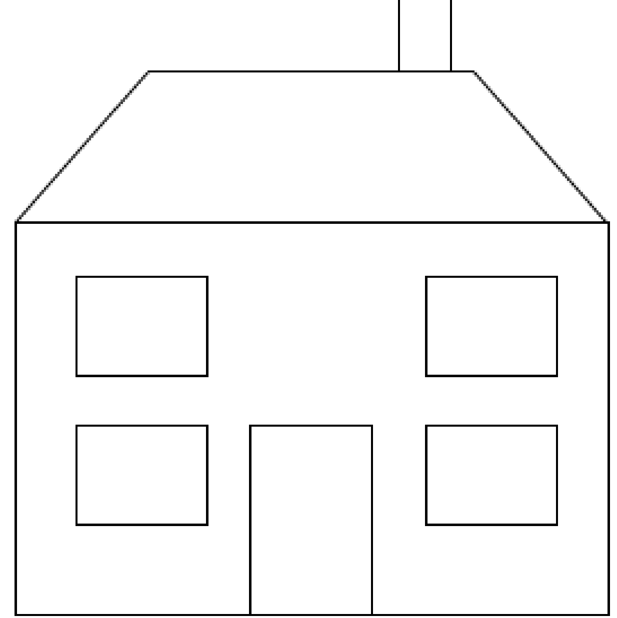


Figure 1.13

'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):

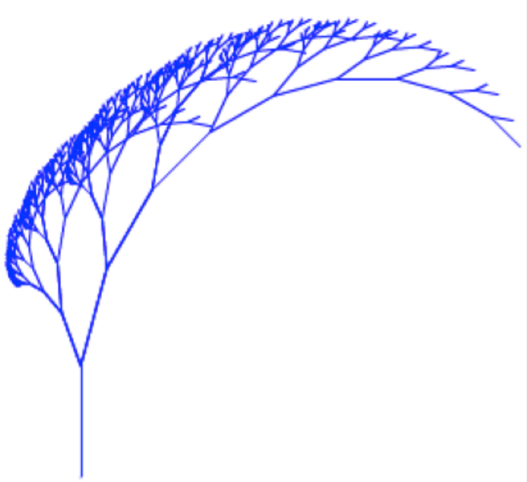


Figure 1.14

```
define tree size level
if level > 0 then
  move size steps
  turn ↺ 20 degrees
  tree size * 0.5 level -
  turn ↻ 35 degrees
  tree size * 0.9 level -
  turn ↺ 15 degrees
  move ( ) - size steps
```

Figure 1.15

Recursive procedure in Scratch to draw Figure 1.15.[16]

```
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.[17]

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 answers are 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.

16 See https://scratch.mit.edu/projects/72176670/
17 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, developing a multi-page wiki site. 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, as well as 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.

Further resources

Apps for Good (n.d.) Available from 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

Gadget Teardowns (n.d.) *Teardowns.* Available from 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

Project Management Institute Educational Foundation (2011) *Project management toolkit for youth.* Available from https://pmief.org/library/resources/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 it:

> **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 which class 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 unconcerned 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 yet another, in that machine code but less so in how this is executed on the CPU or stored in physical memory. Wing puts it well:

> **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 are capable of:

> **Understanding the world through computational thinking (Education Scotland 2017)**

At primary school, pupils will have encountered the **idea** of abstractions in maths as 'word problems' 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 multi-layered nature of abstraction in computing is well worth discussing with pupils as they learn about **how** computers work. For example, ask 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.

At Level 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.

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 consisting 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).[18]

Mathematician J H Conway created a simple computational abstraction, his 'Game of Life' (Berlekamp et al., 2004),[19] 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

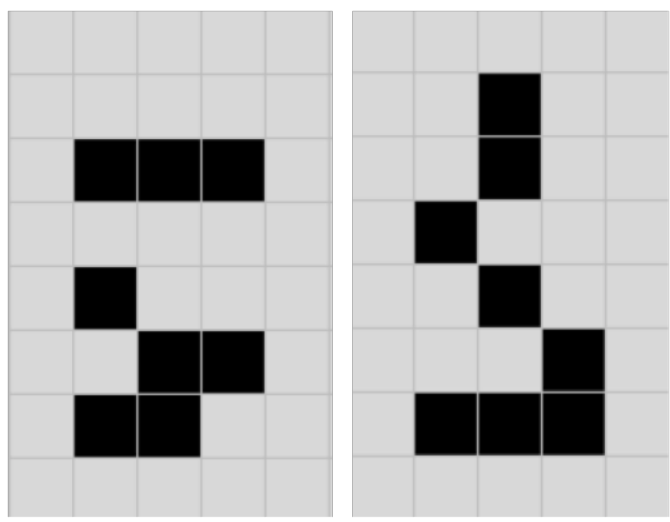


Figure 16a becomes Figure 16b

which becomes

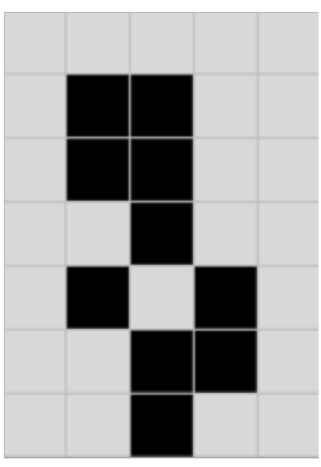


Figure 16c

and so on.

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 Level 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.

18 Things in Minecraft are somewhat more-sophisticated than this. See 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/

19 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, a 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.)

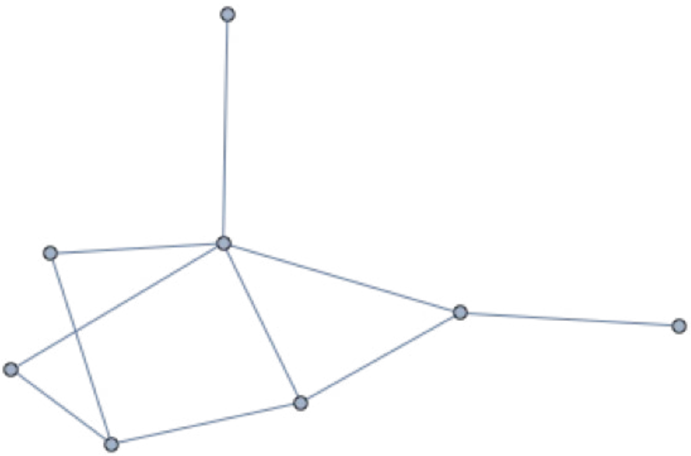


Figure 1.17

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 Facebook, or links between scientists co-authoring papers (Newman, 2001), or actors who have co-starred.[20]

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 16). 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 19) can all be thought of as special sorts of graphs.[21] 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 machines'. A graph is a very useful way to visualise this sort of abstraction, representing the states 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 1.18) generates grammatically correct sentences (notice the double circle on the right, used to show the exit or 'accept' state of the machine).

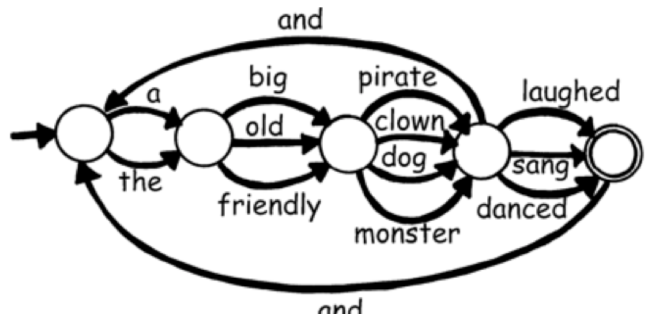


Figure 1.18[22]

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 a graph and the state transition diagram for a finite-state machine – 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

At Level 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 67 - 69.

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 at Level 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

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

Computerphile (2013) *The art of abstraction – Computerphile.* Available from 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

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

Google for Education (2013) S*olving problems at Google using computational thinking.* Available from 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 for **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 use or link 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, automatically, the general rule of finding pages most linked, with a particular phrase, 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 it's time-consuming to think every situation through logically, in such a way that we can give a rational explanation for our decision, 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 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 as 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 Scottish 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, 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' Emphasise how the use of repeating blocks of code or procedures is 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.[23]
- 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

Pysnap (n.d.) Design patterns explained with Python examples. Available from 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.

23 See, for example, 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 considers whether the software and hardware available is fit for purpose, and recognises 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 lie 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 have been well made. Computational abstractions also involve choices over how data are to be structured and represented, and again evaluation should consider whether such choices are correct and optimal.

Evaluation 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 it (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.

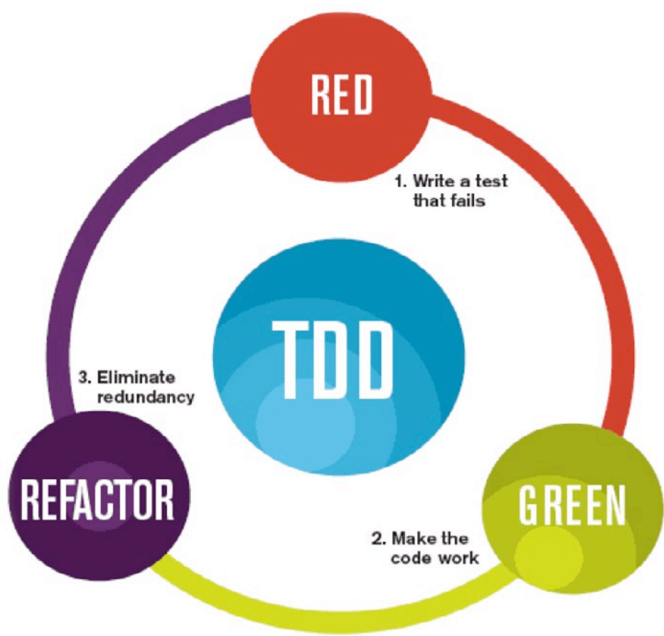


Figure 1.22 The mantra of test-driven development (TDD) is 'red, green, refactor'

Above this attention to the detail in 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.[24]

Evaluation in the curriculum

There is relatively little attention to formal evaluation at Level 2. 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 Benchmarks at Level 3 expect pupils to **evaluate computing solutions.** 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 an abstraction does model the original system; it then considers whether the abstraction is at the right level of detail, and then 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 30) 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 – e.g. how many transitions/clicks are needed to get back to the home page or to 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 page 41- 43), 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 – perhaps for their peer group, for parents at the school or for younger children.[25] 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 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, if not all, good design: simplicity, symmetry, consistency, proportion, attention to detail, fitness for purpose, honesty, inclusion and sustainability.[26] 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 – e.g. those whose first language isn't English, those who are visually impaired, or 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 a program's language settings into another language would give pupils an insight into the challenges of designing and developing with accessibility in mind.

25 Some great examples via www.appsforgood.org/
26 See also www.vitsoe.com/gb/about/good-design and www.gov.uk/design-principles

Further resources

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

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

CAS TV (2016) *Simon Peyton Jones on algorithmic complexity.* YouTube. Available from 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/

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/

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?

Features of a highly effective curriculum include:

> **All children and young people have the opportunity to develop and apply more-sophisticated computational thinking skills.**
>
> **Learners are able to challenge the status quo constructively and generate ideas, including, if appropriate, digital solutions to improve it.**

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 arrival 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.[27]

Creating need not be confined to the screen: there is ample scope 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[28] and Raspberry Jams.[29]

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 81 - 83.

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 mindsets' 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 seeking help from a friend.

27 For a survey of young people's digital making see Quinlan (2015).
28 https://coderdojo.com/
29 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 each gain 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[30]

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 those which could be reused. **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

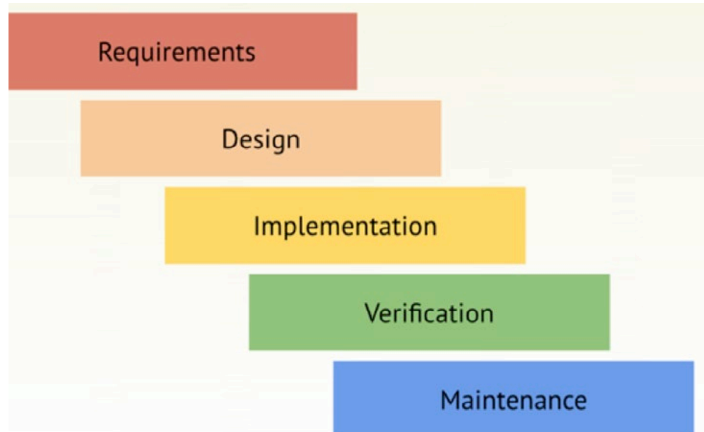


Figure 1.23

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

30 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, and on to testing and evaluation, with further support there if necessary.

Iterative development

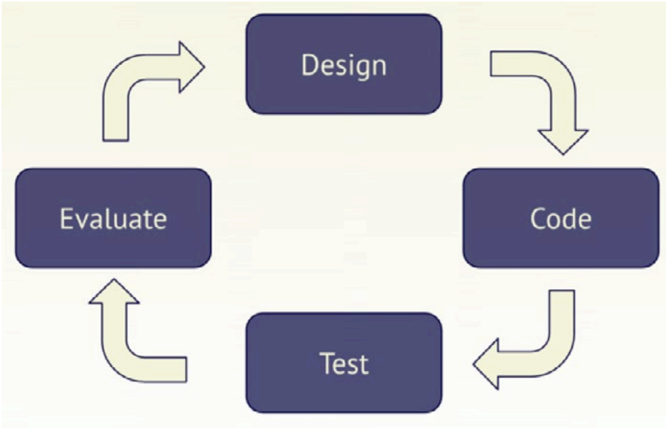


Figure 1.24

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 which 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.

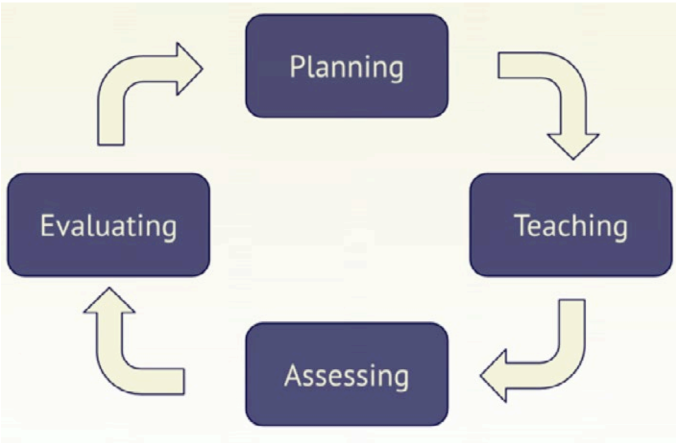


Figure 1.25

Agile methods

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.[31] 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 on 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 design, build and test computing solutions**

At Level 2, pupils should already have had some experience of working on larger software projects, rather than just learning the key programming concepts of sequence, selection and repetition, so that they can:

> **create, develop and evaluate computing solutions in response to a design challenge**

At Level 3, pupils are taught to:

> **group related instructions into named subprograms**

and to:

> **identify processes and information to create a physical computing and/or software solution.**

These points 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 and 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 having 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: you could perhaps set 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'.

Apps for Good (n.d.) Available from 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

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

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

Wirth, N. (1976) *Algorithms + data structures = programs.* Upper Saddle River, NJ: Prentice-Hall.

Programming

What is programming?

Programming

WHAT IS PROGRAMMING?

Programming is the process of designing and writing a set of instructions (a program) for a computer in a language it can understand.

This can be really simple, such as the program to make a robot toy trace out a square; or it can be incredibly sophisticated, such as the software used to forecast the weather or to generate a set of ranked search results.

Programming is a two-step process:

- First, you need to analyse the problem and design a solution. This process will use logical reasoning, decomposition and generalisation to develop computational abstractions which capture the right level of detail about the state and behaviour of the system, as well as algorithms that can solve the problem correctly and efficiently.
- Second, you need to express these ideas in a particular programming language on a computer, making use of data structures so the program can manipulate information. This might sometimes be called coding, and we can refer to the set of instructions that make up the program as 'code'.

Coding provides the motivation for learning computer science – there's a great sense of achievement when a computer does just what you ask it, because you've written the precise set of instructions necessary to make something happen. Coding also provides the opportunity to test out ideas and get immediate feedback on whether something works or not.

What should programming be like in schools?

It is possible to teach computational thinking without coding and vice versa, but the two seem to work best hand-in-hand.

Teaching computational thinking without giving pupils the opportunity to try out their ideas as code on a computer is like teaching science without doing any experiments. Similarly, teaching coding without helping pupils to understand the underlying processes of computational thinking is like doing experiments in science without any attempt to teach pupils the theories which underpin them.

This relationship is reflected in the new computing curriculum, which states that pupils should not only know the principles of information and computation, but should also be able to put this knowledge to use through programming. One of the aims of the Scottish curriculum for computing is for pupils to be able to:

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

At primary school, pupils will have been taught how simple algorithms are implemented as programs on digital devices, from floor turtles to distant web servers. They will have had the opportunity to create and debug their own programs, as well as to predict what a program will do. They will have been taught to design and write programs that accomplish specific goals, which should include controlling or simulating physical systems. They should have learnt to use sequence, selection and repetition in their programs, as well as variables to store data. They should have learnt to use logical reasoning to detect and fix the errors in their programs. All this is likely to have been in the context of device-specific languages for programmable toys such as the Bee Bot and then visual, block-based programming languages such as Scratch.

The Benchmarks at Level 3 build progressively on those from Levels 1 and 2. There will be initial challenges as the Benchmarks are fully implemented at primary level, but these will decrease over time.

Programming at Level 3 should include working with real-world problems and physical systems, with an emphasis on teaching pupils how to develop computational abstractions which model the state and behaviour of such systems. Pupils are also expected to solve a variety of computational problems: look to provide as diverse a range as possible here of contexts for pupils' programming, including cross-curricular opportunities arising out of the other subjects pupils are studying.

Moving beyond the visual programming pupils will have studied at primary school, at Level 4 they are taught at least two programming languages, at least one of which should be text based (see pages 49 - 58 for some thoughts on the choice of language).

It's fine to continue to work in Scratch for Level 3, although anecdotal evidence suggests that some pupils have become somewhat bored with Scratch after much focus on this in primary computing lessons. Other visual languages are available which extend Scratch's functionality, such as Snap! or the semi-visual TouchDevelop.

There seems evidence to suggest that it is effective to introduce a text-based language *alongside* a visual language during Level 3 (Dorling and White, 2015), using the familiar blocks to scaffold pupils' understanding of the semantics of their programming whilst they become increasingly fluent in the syntax of the text-based language (Shneiderman and Mayer, 1979). There are very few things in the expectations for programming or computational thinking at Level 3 that cannot be accomplished in a visual programming language, so a sensible approach would be to avoid rushing to take on the additional cognitive load of programming in a text-based language, until pupils' understanding of the underpinning ideas is very sound (qv Robins, 2010).

Strange as it may sound, teaching pupils to program need not always involve them programming. It is vital that pupils think about their solution before they start coding it, perhaps documenting their solution as pseudocode or a flow chart. Evidence from higher education, and emerging models of effective practice in schools, suggest that pupils need to be able to understand code before they can write code – give them programs in block- or text-based languages and ask them to trace them through, using logical reasoning to predict what would happen when the code is run. For pupils, it is often less daunting for to be given skeleton code to edit, or buggy code to fix, than to have to start from a blank screen. Pair programming is a proven, effective development methodology in the software industry and is likely to have its place in the classroom too. Similarly, reviewing code written by peers helps develop evaluation skills and gives pupils more experience of reasoning about code (see Grover, 2016).

In his guide to *Decoding the programmes of study for computing* document, Simon Peyton Jones suggests the following approaches to teaching programming (Peyton Jones, 2014):

- **Simply experiment with the medium.** Programming environments like Scratch and Kodu make it easy to try things out in a playful, exploratory way: 'I wonder what happens if I press that button/drag that shape?' At this stage the goal is to experiment, gain confidence that nothing bad will happen, and gain intuition about what happens. It's rather like a toddler playing with building bricks.
- **Simply copy an existing program, run it and then start making small changes to it.** The program solves the 'blank sheet of paper' problem. Some changes are limited but fun (for example, change the colour of the monster). As confidence builds, pupils will become more ambitious (for example, can we have more than one monster?).
- **Start to predict what a change will do.** One important aspect of computational thinking is to be able to predict what a program will do or what effect a change to the program will have. For simple, straight-line programs (that is, a simple sequence of instructions) this is pretty easy; the more complicated the program, the harder it gets. But at every level the ability to reason logically about the program is key.
- **Debug a program that is not working properly.** For example, if you want to draw a square with a floor turtle, you might forget to put the pen down, so the turtle crawls around but doesn't draw anything. Debugging always involves coming up with a guess (or hypothesis) about what is going wrong, performing experiments to confirm the guess and making a change that you predict will fix it.[1]
- **Explain to someone else how/why your program works.** The simple act of explaining often reveals latent bugs in your program or potential simplifications to your code.

1 Pupils often try to debug programs by making random changes, unsupported by any reasoning, and running the changed program to see if it behaves better. This isn't computational thinking; it's simply guesswork.

- **Read a program and figure out its purpose.** For example

```
T := 0
for I = 1..N { T := T+I }
```

- You could talk about loops and variables, but an experienced programmer would say 'oh, that just adds up the numbers between I and N, and puts the total in T'. That is, she has worked out the **purpose** of the code, rather than just following the individual steps it takes.
- **Starting from an idea of what you want your program to do, write a program from scratch to do it.**

Classroom activity ideas

- Get pupils to 'reverse engineer' some of the programs they use. For example, they might think through the different states in which a simple system, such as a smart TV remote or a digital microwave oven, can be, and how one links to another (an example of a finite-state machine, see page 30). They could think through what algorithms have been coded into simple or more-complex games they play. Pupils could think about how complicated the code would have to be for familiar application software such as Word or PowerPoint. Draw their attention to how the model-view-controller design pattern has been applied to many of these examples.
- Look for (or create) some simple programming exercises which focus on particular learning objectives. For example, when teaching pupils how a sequence of steps in an algorithm can be translated into code, give pupils a simple algorithm (for example, to draw a regular pentagon) and set them the challenge of implementing this as code.
- Set some extended programming projects in which pupils can work through the process of software development, from original design through writing code to testing and debugging their programs.
- Here are some ideas for extended programming projects:
  - Year 7: turtle graphics in two languages; creating an animated dance routine; developing a game for the BBC micro:bit; developing a maths quiz for primary pupils in a feeder school.
  - Year 8: write a program to encrypt and decrypt text using a shared key; program a robot to find a path through a maze; create a simple chat bot; investigating recursion (for example, fractals using turtle graphics).
  - Year 9: composing music; storing and retrieving information in a database; analysing a large public data set; mobile phone app development (Dorling and Rouse, 2014).

Further resources

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

BBC Bitesize (n.d.) *Controlling physical systems.* Available from www.bbc.co.uk/guides/zxjsfg8

BBC Cracking the Code (n.d.) For examples of source code for complex software systems such as robot footballers and a racing car simulator. available from www.bbc.co.uk/programmes/p01661pj

Computing at School (2016.) CAS Chair, Prof. Simon Peyton Jones' explanation of some of the computer science that forms the basis for the computing curriculum. Available from http://community.computingatschool.org.uk/resources/2936

Code.org (n.d.) For activities and resources. Available from http://code.org/educate

Ford, P. (2015) What is code? *Business Week, Bloomberg.* Available from www.bloomberg.com/graphics/2015-paul-ford-what-is-code/

Norvig, P. (2014) *Teach yourself programming in ten years.* Available from http://norvig.com/21-days.html

Rushkoff, D. (2010) *Program or be programmed: Ten commands for a digital age.* New York, NY: OR Books.

How do you program a Computer?

Programming a computer involves writing code

The code is the set of instructions needed to make the computer do what you want, and this has to be written in a programming language which the computer understands. There are **many** languages to choose from. Some of these will work on only particular devices, or are only intended for a small set of particular purposes. Some languages have been developed specifically with children or other new programmers in mind, whereas others would be best left to professional software developers working on complex projects (although some more-confident and motivated pupils might relish the challenge of mastering such a language). In some languages, you write a program as a sequence of commands for the computer to execute; in others, you might create classes of objects with particular properties that can interact with one another, or sets of functions, each of which will produce certain output from given input.

Programming languages are formal and have to be used in precise ways. Programs are made up of precise, unambiguous instructions – there's no room for interpretation or debate about the meaning of a particular line of computer code. You are only able to write code using the clearly defined vocabulary and grammar of the language, but typically you do so using words taken from English, so code is something which people can write and understand but the computer can also follow. Each programming language will have its own compiler or interpreter, written, or at least customised, for the particular system on which it runs. This takes the code written in the programming language and converts that, either in one go or a bit at a time, into the sort of instructions which the computer's central processing unit (CPU) can follow. We call these instructions 'machine code': the commands here are very simple ones but modern processors can execute these at very, very high speed.

At Level 3, the details of compilers and interpreters are unlikely to feature significantly, and often other layers of abstraction are present between the programming language itself and the CPU – Scratch programs run inside a Flash runtime environment within the web browser; and Snap!, TouchDevelop and the Trinket version of Python are all interpreted as JavaScript, itself being executed within the web browser that runs on the CPU itself.

Programming the BBC micro:bit perhaps gives pupils more of an insight into the detail here, with programs in TouchDevelop, Code Kingdoms and the blocks editor being converted to runtime machine code (the .hex file) by a JavaScript compiler running in the web browser.[2] Python on the micro:bit works a little differently: it flashes a micro-Python **interpreter** onto the micro:bit, which then reads and interprets the Python source code that's flashed to the micro:bit.

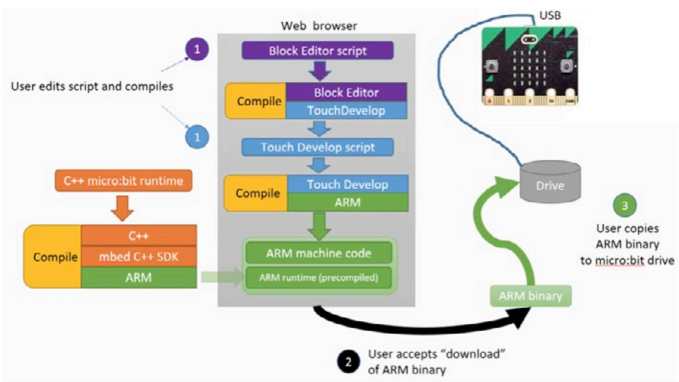


Figure 2.1[3]

How are instructions stored and executed?

You can get a feel for what machine code is like through emulators such as the Little Man Computer (LMC). This abstraction captures the fundamental architecture of modern computers well, in which data and instructions are stored side by side in main memory, with a central processor fetching instructions, executing them and then receiving or outputting data depending on what those instructions are – called Von Neumann architecture, after the designer of ENIAC (Electronic Numerical Integrator And Computer), one of the first electronic stored-program computers which, like most computers after it, adopted this approach.

2 www.touchdevelop.com/docs/touch-develop-in-208-bits
3 From www.microbit.co.uk/offline

The LMC abstraction models a computer as if it were a 'little man' in a closed room with just 100 mailboxes (the memory) at one end of the room and two further mailboxes, input and output, at the other end. In the middle of the room, a calculator (the accumulator) allows addition and subtraction to be done, and there's a resettable counter which points to the mailbox where the next instruction will come from. Normally, the counter increases by one each time, so instructions are worked through in sequence, although branch and repeat instructions can change this. Programming the LMC involves putting a sequence of instructions, plus the data into the mailboxes, setting the counter to zero and letting the LMC work through the instructions given.[4] A number of online LMC implementations are available.[5]

Modern processors are somewhat more complex than this and obviously run significantly faster, but the essential ideas of data and instructions being stored in memory, and simple instructions being executed one after another, remain the same. Between fetching and executing instructions, the instructions are decoded, routing the data that follows to one of the many logic circuits which make up the CPU. Neil Brown puts it like this:

> **The processor runs a fetch – execute cycle. It fetches a single instruction from memory, which is then executed. For example, a LOAD instruction loads a value from memory into a processor register, an ADD instruction adds two registers, and a STORE instruction stores a value from a register back into memory. Once an instruction has been executed, the next instruction is fetched and executed. The number of instructions that can be executed in a second is known as the clock speed, so 1 MHz is one million instructions per second.**[6]

However, he goes on to explain that modern processors are in practice somewhat more complex. Apart from the simplest microprocessor-controlled devices, modern computers are multi-core devices, with a number of CPUs each able to execute instructions independently of and, in parallel with, the others. Parallel computing, using many CPU cores simultaneously, can be used for applications such as graphics rendering, search and even simulating the brain.[7]

What programming languages should you use?

There are many languages to choose from. The majority are more complex than necessary for those still getting to grips with the ideas of programming, but there are plenty of simple, well supported, general-purpose languages that can be used very effectively in the lower secondary classroom. Try to pick a language that you will find easy to learn or, better still, know already.

Consider these points when choosing a programming language:

- Not all languages run on all computer systems.
- Choose a language that is suitable for your pupils. There are computer languages that are readily accessible to lower-secondary pupils – in some cases this will mean one that has been written with pupils in mind, or at least adapted to make it easier to learn, but well constructed general-purpose languages should not be ruled out.
- Choose a language supported by a good range of learning resources. It's better still if it has online support communities available, both for those who are teaching the language and those who are learning it.
- It is beneficial to the pupils if they can continue working in the language on their home computer or, even better, if they can easily continue to work on the same project via the internet.
- Think carefully about primary to secondary transition. If many of your pupils have been learning, for example, Scratch whilst at primary school, then they can hit the ground running with some ambitious projects for S1 without having to learn the commands and interface of a new language. On the other hand, they might perhaps have become somewhat jaded with Scratch and be ready for something different, now they are at 'big school'.

4 https://en.wikipedia.org/wiki/Little_man_computer
5 For example www.peterhigginson.co.uk/LMC/
6 https://academiccomputing.wordpress.com/2012/04/29/the-computer-is-a-lie/
7 Prof. Steve Furber discussing the SpiNNaker spiking neural net computer at www.youtube.com/watch?v=wnSjR04qang

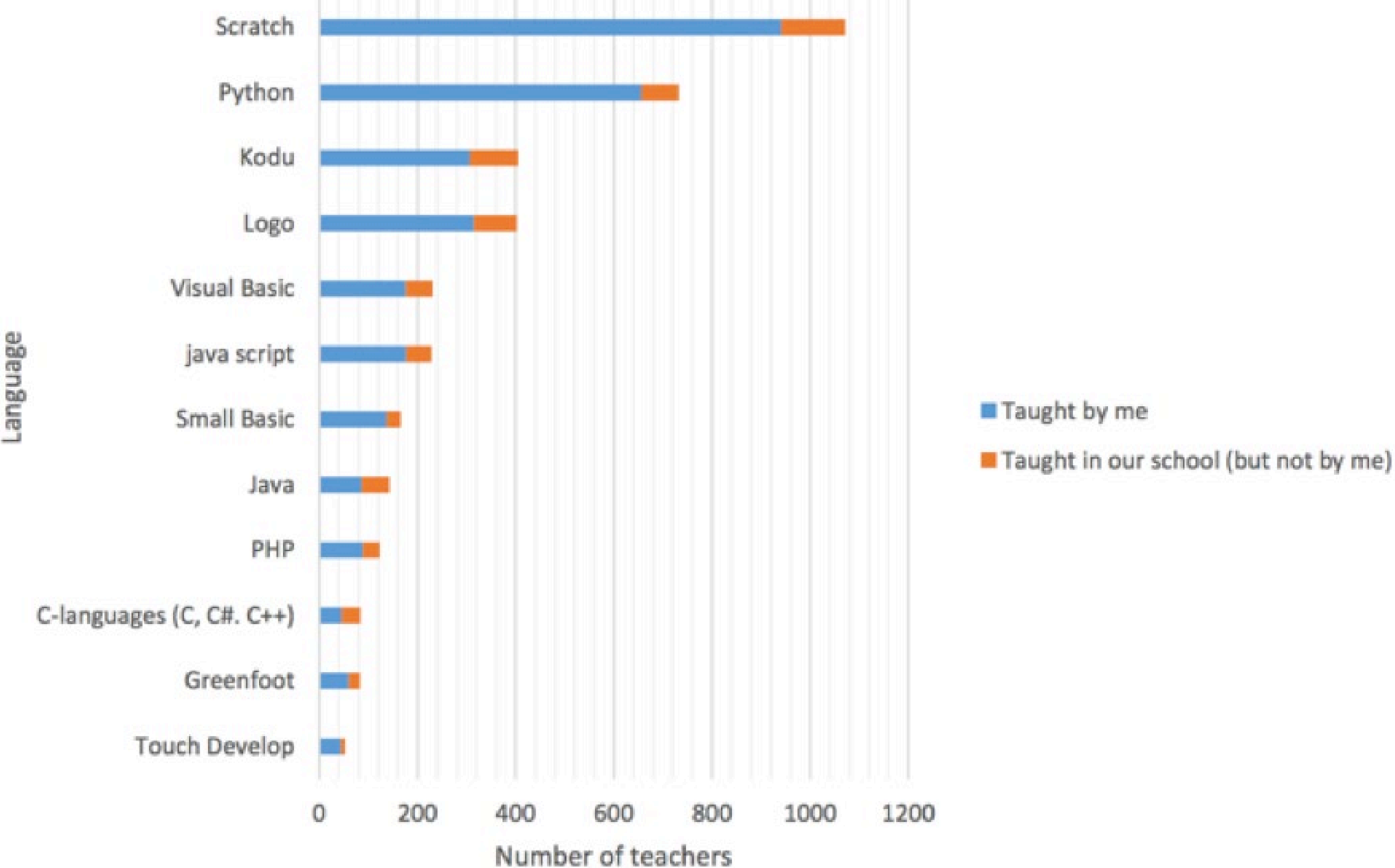


Figure 2.2 CAS survey of 1,159 teachers of computing, 2015[8]

- Aim for depth rather than breadth – the computing curriculum is about learning the principles of computer science through practical programming, rather than learning lots of different languages. Mastering one or two languages will mean pupils can start to tackle authentic problems, perhaps from elsewhere in the curriculum, with a degree of independence. Fluency in a couple of languages is far more useful than a vague familiarity with a dozen. There **is** value in learning multiple languages and, particularly, different language paradigms, but there's no need to rush into this at Level 3 or Level 4. Few of your pupils will become professional software developers but all ought to understand the basics of programming.

Mark Guzdial recently blogged about his own set of principles for choosing a programming language for teaching:[9]

(1) Connect to what learners know.
(2) Keep cognitive load low.
(3) Be honest.
(4) Be generative and productive.
(5) Test, don't trust.

There's a view that some languages are better at developing good programming 'habits' than others. Good teaching, in which computational thinking is stressed alongside coding, through an emphasis on planning and reasoning about code, should help to prevent pupils developing bad coding habits at this stage.

8 Sentence, S (2016) Computing At School Annual Survey 2016. Available at http://community.computingatschool.org.uk/files/8106/original.pdf (accessed 28/12/16)
http://ishallteach.org/index.php/2016/05/18/survey-results-what-programming-languages-are-being-taught-in-classroom-edtech-ict-computing-education/
9 Online at http://cacm.acm.org/blogs/blog-cacm/203554-five-principles-for-programming-languages-for-learners/fulltext; qv https://computinged.wordpress.com/2016/06/20/how-to-choose-programming-languages-for-learners/

Further resources

CS Field Guide (2016) *Programming languages.* Available from http://csfieldguide.org.nz/en/chapters/programming-languages.html

Iry, J. (2009) *A brief, incomplete, and mostly wrong history of programming languages.* Available from http://james-iry.blogspot.mx/2009/05/brief-incomplete-and-mostly-wrong.html

Rosettacode.org (n.d.) Many programming languages compared for different problems and algorithms. Available from http://rosettacode.org/wiki/Rosetta_Code

Utting, I., Cooper, S., Kolling, M., et al. (2010) *Alice, Greenfoot, and Scratch – A discussion.* University of Kent. Available from http://kar.kent.ac.uk/30617/2/2010-11-TOCE-discussion.pdf

Visual Programming Languages

There is a number of graphical programming toolkits available; these make learning to code easier than it's ever been. In most of these, programs are developed by dragging or selecting on-screen blocks or icons which represent particular instructions in the programming language. These can normally only fit together in ways that make sense, and the amount of typing – and thus the potential for spelling or punctuation (syntax) errors – is kept to an absolute minimum.

With toolkits like these it's easy to experiment with creating code. By letting the programmer focus on the ideas of their algorithm, rather than the particular vocabulary and grammar of the programming language, programming and learning to program become easier and often need less teacher input.

Kodu

Microsoft's Kodu (Figure 2.3) is a rich, graphical toolkit for developing simple, interactive 3D games.

Each object in the Kodu game world can have its own program. These programs are 'event driven': they are made up of sets of 'when [this happens], do [that]' conditions, so that particular actions are triggered when certain things happen, such as a key being pressed, one object hitting another, or the score reaching a certain level.

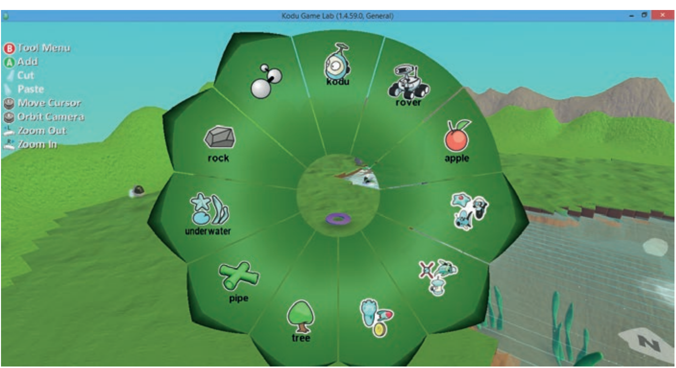


Figure 2.3 Kodu interface

Programmers can share their games with others in the Kodu community, which facilitates informal and independent learning. There's also plenty of scope for pupils to download and modify games developed by others, which many find quite an effective way to learn the craft of programming. This can also offer pupils a sense of creating games with an audience and purpose in mind.

Scratch

In Massachusetts Institute of Technology's Scratch (Figure 2.4), the programmer can create their own graphical objects, including the stage background on which the action of a Scratch program happens, and a number of moving objects ('sprites'), such as the characters in an animation or game.

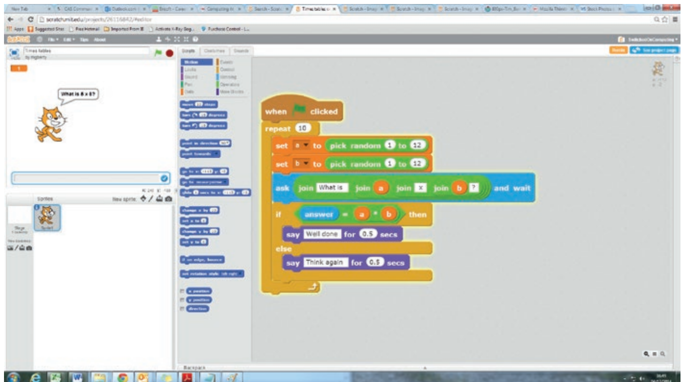


Figure 2.4 Screenshot of a Scratch program

Each object can have one or more scripts, built up using the building blocks of the Scratch language. To program an object in Scratch, you drag the colour-coded block you want from the different palettes of blocks, and snap this into place with other blocks to form a script. Scripts can run in parallel with one another or be triggered by particular events, as in Kodu.

A number of other projects use Scratch as a starting point for their own platforms. For example, ScratchJr is a tablet app designed for very young programmers. There's a great online community for Scratch developers to download and share projects globally, making it easier for pupils to pursue programming in Scratch far beyond what's needed for the Scottish curriculum. There's also a supportive educator community, which has developed and shared high-quality curriculum materials.

The current version of Scratch (2.0) allows users to create their own new blocks built from Scratch's commands, and allows parameters to be passed to these blocks. These custom blocks can display text on screen, change sprite or global variables and so on, but they cannot return values – that is, they are procedures rather than functions, and this can be a limiting factor with Scratch.

Scratch is available as a free web-based editor or as a standalone desktop application. Files can be moved between online and offline versions. There's some support for interfacing with hardware components, including webcams, and a way to extend Scratch functionality via an application programming interface (API).[10]

Snap!

Berkeley's Snap![11] started life as Build Your Own Blocks (BYOB) – a fork of the Scratch code which, unsurprisingly, allowed users to create their own blocks. Scratch now has this functionality, albeit in a somewhat limited way, but Snap! has continued development, focussing on implementing some more-sophisticated computer science ideas in a block-based language: perhaps the most immediate offers proper functions that can return values to the program or function that called them (see Figure 2.5).

Snap!'s functions, though, are 'first class' citizens; that is, functions can be passed as arguments to other functions. It's relatively easy in Snap! to implement new control structures (such as 'for' loops) or functional programming ideas (such as a map function which applies the same user-specified function to each of a list of elements). Snap! also supports anonymous 'lambda' functions. Snap! has better support for lists too, which can be useful when teaching data structures, including the ability to have lists of lists.

Snap! is implemented in JavaScript so runs in any browser, including those without Flash support, such as tablet computers. Like Scratch, there's a number of extensions and modifications available, including tools to import (but not export) Scratch projects, an exporter to create standalone applications, and Edgy,[12] a version of Snap! designed for programming with graphs. On the downside, there is nothing like the vibrant, global user community of Scratch, and far fewer teaching resources are available. However, it is used as the teaching language for Berkeley's Beauty and Joy of Computing course, which is also offered to US high-school students as an APCS (advance placement computer science) Principles course.[13]

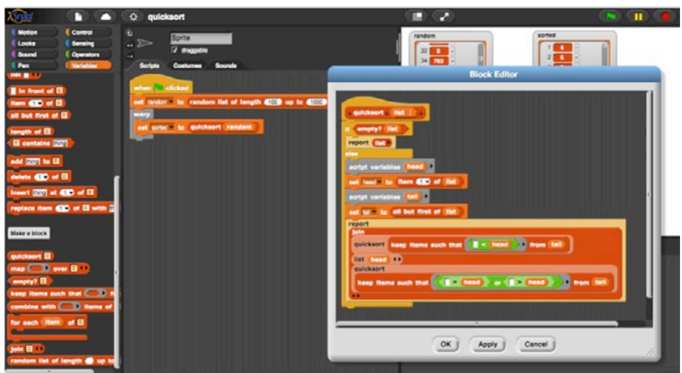


Figure 2.5 Recursive implementation of quicksort in Snap!

10 http://scratchx.org/
11 Introduction by John Stout for CAS TV at www.youtube.com/watch?v=7tjNnF4fAgI
12 http://snapapps.github.io/
13 http://bjc.berkeley.edu/

Classroom activity ideas

- Pupils could develop a game in Kodu, taking inspiration from some of the games on the Kodu community site. As a starting point, tell them to create a game in which Kodu (the player's avatar in the game) is guided around the landscape bumping into (or shooting) enemies.
- Pupils could take photographs of one another in a variety of dance poses, then use Scratch to create a program which animates these to choreograph a simple (or complex) dance routine. They could add music to their program, either by importing an MP3 or by composing music in Scratch.
- Scratch lends itself to game programming and it can be a good platform for pupils to work on projects like this quite independently. Start by asking pairs or groups to plan their games very carefully, thinking through the **rules** of their games, which will be the algorithms used as the basis for their programs. As well as working creatively to design the media used in their games, pupils will need to think through how the user's interaction with the game will work, tweaking their game to provide just the right level of challenge to the player. Games based on classic arcade names such as Pong, Pacman and Duck Shoot can be programmed in Scratch without too much difficulty.
- Pupils could implement fractions arithmetic in Snap!, treating fractions as lists with just two members. They would need to write helper functions to find highest common factors and simplify fractions, which could then be used in other functions to implement addition, subtraction, multiplication and division.

Further resources

Armoni, M. and Ben-Ari, M. (2013) *Computer science concepts in Scratch.* Available from http://docplayer.net/14600408-Computer-science-concepts-in-scratch.html.

Bagge, P. (2015) *How to teach primary programming using Scratch.* Buckingham: The University of Buckingham Press.

Brennan, K., Balch, C., Chung, M. (2014) *An introductory computing curriculum using Scratch.* Harvard, MA: Harvard Graduate School of Education. Available from http://scratched.gse.harvard.edu/guide/ [29/12/16]

Harvey, B. and Mönig, J. (2011) *Snap! reference manual.* Available from http://snap.berkeley.edu/SnapManual.pdf

Kelly, J. (2013) *Kodu for kids.* Indianapolis, IN: Que Publishing.

Kodu Game Lab Community (n.d.) Available from www.kodugamelab.com/

Malan, D. (2007) *Scratch for budding computer scientists.* Available from http://cs.harvard.edu/malan/scratch/printer.php

Scratch (n.d.) Available from http://scratch.mit.edu/

ScratchEd (n.d.) Online community for educators. Available from http://scratch.mit.edu/educators/

Snap! (n.d.) Available from http://snap.berkeley.edu/

Text-based Programming Languages

Most software development in academia and industry takes place using text-based languages, where programs are constructed by typing the commands from the programming language at a keyboard.

Historically, text-based programming has been a real barrier for children when learning to code, and there's no need to rush into text-based programming for Level 3. Python is by far the most common text-based programming language in secondary schools at the moment, and this offers many advantages, as it is relatively easy to learn and sufficiently flexible to be used for general-purpose, real-world development. However, it's worth looking at some of the alternatives as well.

Logo

Logo was developed by Seymour Papert and others at MIT as an introductory programming language for children. It's probably best known for its use of

'turtle graphics' – an approach to creating images in which a 'turtle' (either a robot or a representation on screen) is given instructions for drawing a shape, such as:

```
REPEAT 4 [
        FORWARD 100
        RIGHT 90 ]
```

Repetition can be nested, allowing relatively complex figures to be programmed quite easily:

```
REPEAT 10 [
  REPEAT 5 [
    FD 100
    RT 72
  ]
  RT 36
]
```

Figure 2.6

Papert saw Logo as a tool for children to think with, just as programming is both the means to and motivation for computational thinking.

In Logo programming, more-complex programs are built up by 'teaching' the computer new words. These are called procedures. For example, you could define a procedure to draw a square of a certain size using the key words of the language. Once this is defined, typing it in will then result in the turtle drawing a square.

```
TO SQUARE :SIDE
        REPEAT 4 [
                FORWARD :SIDE
                RIGHT 90 ]
        END
SQUARE 50
```

Many associate Logo with these sorts of turtle graphics programs. Turtle graphics are supported by most programming languages, including Scratch, Snap!, TouchDevelop, Small Basic and Python. Logo is, however, capable of more-general programming so, for example, factorials (the product of the integers up to and including a number, for example 5!=1x2x3x4x5=120) can be calculated using a recursive function in Logo:

```
TO FACTORIAL :NUMBER
  IF :NUMBER = 1 [OUTPUT 1]
  OUTPUT :NUMBER * FACTORIAL :NUMBER - 1
END
```

Logo's original development grew out of Lisp, and thus it also has good support for lists.[14] Whilst not a popular choice at present, the text-based programming requirements of the Scottish curriculum could be met with Logo.

Microsoft Small Basic

Microsoft Small Basic is a simplified version of Microsoft's Visual Basic programming language and associated environments, and indeed Small Basic programs can be exported to form the basis of more-complex Visual Basic code. The language is text based, but has a development studio designed to help with many of the difficulties of text-based programming: thus, there's a built-in visual environment in which programs can be run, and 'IntelliSense' is used to help suggest and complete the keywords of the language as you type. The language is kept deliberately small (just fourteen keywords), although these are supplemented through an extensive standard library, with support for turtle graphics, as in Logo, as well as external resources such as Flickr. For example, the program to draw a square in Small Basic would look something like:

```
Turtle.Show()
For i = 1 to 4
  Turtle.Move(100)
  Turtle.TurnRight(90)
EndFor
```

As with Kodu and Scratch, there's an online gallery in which programmers can share the source code for their programs, with others using these as a starting point for their own work.

14 The classic introduction to programming in Logo beyond the realm of turtle graphics is Harvey (1997).

TouchDevelop

Typing code on a tablet computer or a smartphone is not easy, and this can be problematic for schools that use these devices extensively.

Developed by Microsoft Research, TouchDevelop is a programming language and environment which takes into account both the challenges posed and the opportunities offered by touch-based interfaces such as those on tablets and smartphones.

TouchDevelop makes it quite easy to develop an app for a smartphone or tablet on the smartphone or tablet itself.

Although TouchDevelop is a text-based language, programmes aren't typed but are created by choosing commands from the options displayed in a menu system. In this way, TouchDevelop is a halfway house between graphical and text-based programming. Those who have become familiar with drag-and-drop or keyboard-based programming sometimes find it hard to adapt to the touch-optimised interface of TouchDevelop.

As with Logo, turtle graphics commands are available as standard. On some platforms TouchDevelop can also access some of the additional hardware built into the device, such as the accelerometer or global positioning system (GPS) receiver, allowing more-complex apps to be developed: these can be hosted online as web-based apps, or installed directly on the device if it's a Windows phone. TouchDevelop is one of the program editors provided for the BBC micro:bit (Figure 2.7).

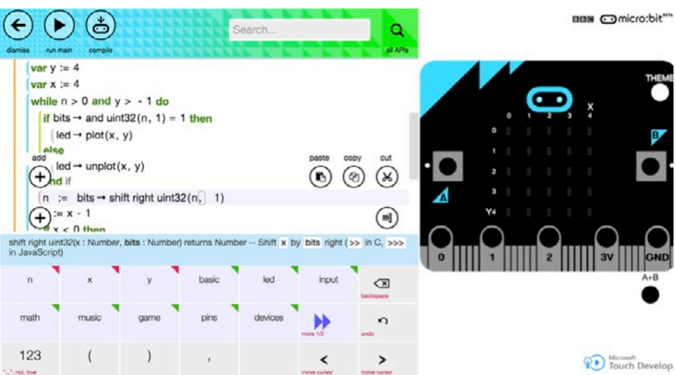


Figure 2.7 TouchDevelop for the BBC micro:bit – the function shown implements a decimal-to-binary converter

A particularly nice feature of TouchDevelop is the use of interactive tutorials to scaffold pupils' learning of the language.

Microsoft Excel

Whilst few would immediately think of it as a programming language, the Excel spreadsheet package is a text-based programming language, although admittedly a rather strange one (Peyton Jones et al., 2003). In Excel, rather than creating a sequence of instructions, you write code (Excel formulae) to create a system of interlinked functions, which take values in the spreadsheet cells, and return the results of performing computation on those functions. This provides some introduction to **functional programming**, as well as being really useful in all sorts of situations where a large amount of numerical data needs to be processed, or where mathematical functions provide a good way to model a complex real-world problem. The way in which an Excel spreadsheet shows the values in each cell can help pupils visualise how this sort of computation is performed.

Computational thinking processes such as logical reasoning, abstraction, decomposition and generalisation apply just as much to developing a spreadsheet in Excel as they do to writing imperative programs in Scratch, Small Basic or the other languages discussed here. There are many real-world problems for which a spreadsheet may be the most efficient solution. It would be a shame for pupils to miss out on developing some fluency with this approach to solving computational problems.

Python

For many secondary teachers and their pupils, Python seems a great introductory text-based programming language. Papert argued that a good teaching language should have 'low floors, wide walls and high ceilings', and Python seems to offer all three:

- Whilst text-based programming inevitably introduces additional cognitive load over graphical languages, Python allows pupils to write programs with a similar **structure** to that which they would write in Scratch, at least in the case

of Scratch programs made up of just one script and some custom blocks – support for multi-threading, which is easy in Scratch, is much less straightforward in Python.

- Python has a good set of standard libraries to extend the functionality of the language, including a great implementation of turtle graphics,[15] game libraries[16] and support for developing programs with graphical user interfaces. There are many other libraries available publicly via the internet, together with tools to install these without too much difficulty. There is very good support for science, maths and statistics, plus libraries for natural-language processing, working with graphs and many, many other specialised areas. Python is installed as standard on the Raspberry Pi and Macs, and it can be run on Android phones and the BBC micro:bit. There's also a good culture of folk sharing Python programs for a wide range of applications via Github.
- Python is a proper, grown-up programming language used for real-world software development in a range of domains. Whilst not an entirely functional programming language, Python supports functional programming., and whilst not an entirely object-oriented language, it supports object-oriented programming. There is lots of interest in it in academic computing, including as a teaching language for computer science degrees as well as for science and humanities. It can be used for developing server-driven web applications, and is used by Google, Facebook, Yahoo, EventBrite, Reddit and NASA. Python programming skills are in demand for jobs in software development.

Here's an example of a simple Python program to do a drill-and-practice tables test, which illustrates some of the features of the language:

```
import random
for i in range(10):
    a = random.randint(1,12)
    b = random.randint(1,12)
    question="What is "+str(a)+" x "+str(b)+"? "
    answer = int(input(question))
    if answer == a*b:
        print("Well done!")
    else:
        print("No.")
```

If you've never seen Python code before, this might be a bit daunting, but spend a couple of minutes reading through the code and you should get a reasonable feel of what's going on here.

A few things to mention:

- The nice, indented layout here is a feature of the language – repeated code in the 'for' loop is indented, as are the different bits of code that get executed in the 'if … else' selection statement. Similarly, the ':' which precedes these indented blocks is part of the language.
- The **randint** command (which picks a random integer from the given range) isn't part of the Python language itself, and so needs to be imported as part of the **random** library.
- Variables in Python have implied types – 'a' and 'b' are integers, 'question' is a string. Functions allow variables to be converted between one type and another.
- In the **print** command, the text to be printed is inside (brackets). This was one of the significant changes from Python 2 to Python 3, so it's worth double-checking which version of Python is running on your computer.

Don't be too ambitious as you introduce pupils to programming in Python: learning any text-based language demands concentration and attention to detail, and this makes it hard for pupils to give lots of attention to mastering complex algorithmic ideas at the same time.

Starting with something familiar, such as turtle graphics, offers a nice way in. The Logo program on page 54 can be implemented in Python very easily, using the standard Turtle library:

```
from turtle import *

for i in range(10):
   for j in range(5):
      forward(100)
      right(72)
   right(36)
done()
```

This is very similar to the same program in Scratch (Figure 2.8):

15 https://docs.python.org/3/library/turtle.html
16 http://www.pygame.org/docs/ref/pygame.html and (more usefully) https://pygame-zero.readthedocs.io/en/latest/

```
pen down
repeat 10
    repeat 5
        move 100 steps
        turn ↻ 72 degrees
    turn ↻ 36 degrees
```

Figure 2.8

Many pupils will be able to see the connections between the algorithm for their pattern and its expression as code in the two languages.

Another good introductory project is to get pupils to recreate the 'choose your own adventure' games of old (Jackson and Livingstone, 1982), perhaps providing pupils with the skeleton code of a procedure for an individual 'room' and allowing them to adapt and expand on this:

```
def room0():
    print ("""You are in room 0.
There are exits here to room 1 and room 2.""")
    choice = input ("Choose 1 or 2: ")
    if choice == "1" :
        room1()
    elif choice == "2":
        room2()
    else:
        print("That's not one of the choices!")
        room0()
```

Adventure games are based on a binary tree graph as the computational abstraction, with the nodes being the rooms (the states), and the edges the choices between them (the behaviour) at each point. With minor adjustments the same program could be turned into a 'branching database' classification key for a group of plants or animals.

Some practicalities

Python is a free download and can be installed on Windows and Linux systems; Python is pre-installed on OS X. Python itself does not introduce any security risks to a properly configured system or network – it doesn't need to run with administrator or root permission and thus can only modify files or directories to which the user already has write-access.[17] An alternative to installing Python locally is to access a Python interpreter and editor via the web:[18] this is useful for learning the language, but can make it difficult to access particular libraries or develop more-complex software.

Downloading and installing Python brings you the Python interpreter itself and a simple integrated development environment (IDE) called IDLE. You can use IDLE to write, save and edit Python programs and to run them – the output of the code appears in another window (Figure 2.9).

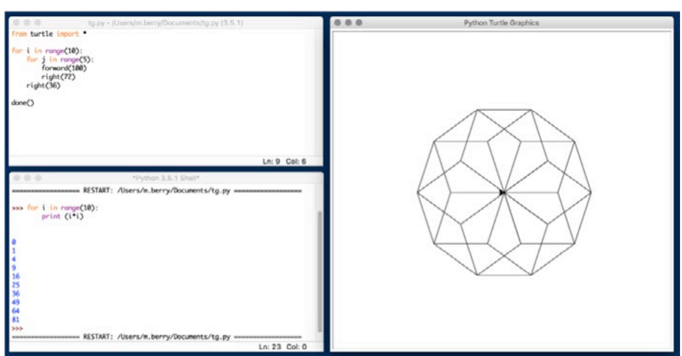


Figure 2.9 IDLE showing program editor, console and turtle graphics output on a Mac

You don't have to use IDLE to use Python. You can write Python code in any text editor,[19] running the program you save at the command prompt or shell, or you can use more sophisticated IDEs.[20] You can also use Python in interactive mode, either in IDLE's console or at the command prompt/shell after just typing 'python'.[21] This can be useful for just experimenting with the syntax of the language rather than for writing programs.

17 See http://community.computingatschool.org.uk/resources/446 for a discussion of some of the school infrastructure issues associated with providing pupils with access to programming tools.
18 For example https://trinket.io/, https://codio.com/ and https://c9.io/
19 For example Notepad ++, Atom or Mu.
20 Integrated development environments, for example PyCharm Edu or Visual Studio Python Tools for Windows.
21 Or using the Jupyter/iPython interactive notebook package.

Other languages

Other programming languages are available, which could be used as introductory text-based languages in secondary schools, for example: JavaScript, Ruby, Pyret, Visual Basic, Swift (OS X only) and Java (perhaps using Greenfoot).

Classroom activity ideas

- Revisit the turtle graphics activities you might have been using for programming in the old information and communication technology (ICT) curriculum. Whilst these can be accomplished using the motion and pen commands in Scratch, projects such as drawing regular polygons, a simple house or complex repeating patterns like 'crystal flowers' are usually well enough understood for pupils to get to grips with the additional challenges of text-based languages – there's evidence that it can be effective for pupils to work in a visual- and a text-based language, side by side, for this (Dorling and White, 2015).
- Explore some of the commands and functions available in these languages for working with text. For example, can pupils write a program which takes any sentence and converts it into capital letters, or reverses the sentence, or removes all the vowels from the sentence, or reverses each word in the sentence?
- Explore how one or more of these programming languages could be used to simulate dice being rolled. In Excel, could pupils simulate rolling 100 dice at the same time and then draw a bar chart of the results? Ask them to think how they would do that in Scratch. Can pupils create an app in TouchDevelop which simulates rolling a dice when the phone, tablet or micro:bit is shaken? Ask pupils to think about how deterministic computers can simulate random events such as these.
- On the Raspberry Pi, Python can be used as a scripting language for Minecraft or for simple physical computing activities, using the Raspberry Pi's general-purpose input/output (GPIO) pins.

Further resources

Code Club (n.d.) Python projects. Available from www.codeclubprojects.org/en-GB/python/

DfES (1998) *Archived lesson plan for creating crystal flowers.* Available from http://webarchive.nationalarchives.gov.uk/20090608182316/http://standards.dfes.gov.uk/pdf/primaryschemes/itx4e.pdf

Downey, A. (2012) *Think Python.* Sebastopol, CA: O'Reilly Media (qv http://greenteapress.com/thinkpython/html/index.html).

Horspool, N. and Ball, T. (2013) *TouchDevelop: Programming on the go.* New York, NY: Apress. Available from www.touchdevelop.com/docs/book

Logo (n.d.) Available from www.calormen.com/jslogo/ and elsewhere.

Papert, S. (1980) Mindstorms: *Children, computers, and powerful ideas.* New York, NY: Basic Books Inc.

Python (n.d.) Available from www.python.org/ and online via trinket.io

Raspberry Pi (n.d.) Teaching and learning resources, many of which include Python programming. Available from www.raspberrypi.org/resources/ for example www.raspberrypi.org/learning/python-intro/

Shaw, Z.A. (2013) *Learn Python the hard way: A very simple introduction to the terrifyingly beautiful world of computers and code.* Boston, MA: Addison-Wesley (qv http://learnpythonthehardway.org/book/).

Small Basic (n.d.) Available from www.smallbasic.com/

TouchDevelop from Microsoft Research (n.d.) Available from www.Touchdevelop.com

Tranter, M. (2014) *Ten Python lessons.* CAS. Available from http://community.computingatschool.org.uk/resources/2155

What's inside a Program?

Whilst the detail will vary from one language to another, there are some common structures and ideas which programmers use over and over again, from one language to another and from one problem to another:

- Sequence: running instructions in order (see below).
- Selection: running one set of instructions or another, depending on what happens (see page 61).
- Repetition: running some instructions several times (see page 63).
- Modularity: building programs from smaller, independent blocks of code that return values or do specific things (see page 67).
- Data structures: organising data so that it can be stored and retrieved from the computer's memory (see page 73).

These are so useful that it's important to make sure all pupils learn these. Sequence, selection and repetition are introduced at Level 2 of the computing curriculum, where pupils also learn about variables, simple data structures that handle just one piece of information.

This Scratch script (Figure 2.10) shows sequence, selection, repetition and variables. Can you work out which bit is which before we look at these ideas in detail?

```
when [flag] clicked
set score to 0
repeat 10
  set a to pick random 1 to 12
  set b to pick random 1 to 12
  ask join What is join a join x join b ? and wait
  if answer = a * b then
    say Well done for 0.5 secs
    change score by 1
  else
    say Think again for 0.5 secs
```

Figure 2.10

This program does the same thing in Python. Can you see the similarities and differences between the two? Can you work out how Python deals with sequence, selection, repetition and variables?

```
import random
for i in range(10):
    a = random.randint(1,12)
    b = random.randint(1,12)
    question="What is "+str(a)+" x "+str(b)+"? "
    answer = int(input(question))
    if answer == a*b:
      print("Well done")
    else:
      print("Think again")
```

Further resources

BBC Bitesize (n.d.) *How do we get computers to do what we want?* (Covering sequence, selection and repetition). Available from www.bbc.co.uk/guides/z23q7ty

Berry, M. (2014) *Tables Test.* Scratch program, available from http://scratch.mit.edu/projects/26116842/#editor [29/12/16].

Berry, M. (2016) *Tables test.* Python program, available from https://trinket.io/library/trinkets/d15c8f972b [29/12/16].

Bitesize programming materials at www.bbc.co.uk/education/topics/zhy39j6

Cracking the Code clip (n.d.) Available from www.bbc.co.uk/programmes/p016j4g5

Sequence

Programs are built up of sequences of instructions.[22] At Early Years Level and Level 1, when pupils start programming with floor turtles, their programs consist entirely of sequences of instructions, built up as the stored sequence of button presses for what the floor turtle should do. As with any program, these instructions are precise and unambiguous, and the floor turtle will simply take each instruction (each button press) and turn that into signals for the motors driving its wheels.

22 In imperative programming languages such as those discussed here. Declarative languages such as Haskell, F# and Excel work rather differently.

Pupils' first Scratch or Python programs are also likely to be made up of simple sequences of instructions. Again, these need to be precise and unambiguous and, of course, the order of the instructions matters. In developing their algorithms, pupils have to work out exactly what order to put the steps in to complete a task. In more complex programs involving variables or other data structures, they will need to think through how the steps in their programs change the data stored.

```
when [flag] clicked
set volume to 100 %
set tempo to 72 bpm
set instrument to 1
play note 64 for 0.5 beats
play note 64 for 0.5 beats
play note 65 for 0.5 beats
play note 67 for 0.5 beats
play note 67 for 0.5 beats
play note 65 for 0.5 beats
play note 64 for 0.5 beats
play note 62 for 0.5 beats
play note 60 for 0.5 beats
play note 60 for 0.5 beats
play note 62 for 0.5 beats
play note 64 for 0.5 beats
play note 64 for 0.75 beats
play note 62 for 0.25 beats
```

Figure 2.11 A simple music program in Scratch

```
print("Hello!")
name = input("What is your name?")
print("It's a pleasure to meet you, "+name+".")
print("What odd weather it's been of late.")
today = input("What have you been doing today?")
print("What a coincidence! I've been "+today.lower()+" too.")
```

A simple chat bot in Python.

Classroom activity ideas

- Ask pupils to experiment with programming Scratch to play music, as in Figure 2.11 above. Take a simple, familiar melody, perhaps in score notation or just as a list of notes, and have pupils translate this into a sequence of Scratch commands. Can pupils tell by ear where there are mistakes in their code? Pupils could do a similar exercise using Python's winsound library for Windows, or Sonic Pi.
- Ask pupils to design, plan and code scripted animations in Scratch, perhaps using a timeline or storyboard to work out their algorithm before converting this into instructions for sprites in Scratch. Animations could be based on historical events, scenes from a reading book or dialogue in a foreign language pupils are studying. Scratch has support for recording and playing back audio. Pupils might enter their animation for the UK Schools Computer Animation Competition.
- Pupils could take the chat bot idea above and develop this further, either in Python (as shown) or Scratch, or perhaps using both languages side by side.

Further resources

Animation 16 (2015) *UK schools computer animation competition.* University of Manchester. Available from http://animation16.cs.manchester.ac.uk/. YouTube channel for winning entries www.youtube.com/user/AnimationComp

Barefoot Computing (2014) *Sequence.* Available from http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/programming/sequence/ (free, but registration required).

Barefoot Computing (2014) *Viking invasion animation in Scratch (for upper KS2).* Available from http://barefootcas.org.uk/programme-of-study/use-sequence-in-programs/upper-ks2-viking-raid-animation-activity/ (free, but registration required).

Code Club (n.d.) Python activities involving sequence. Available from https://codeclubprojects.org/en-GB/python/about-me/ and http://projects.codeclubworld.org/en-GB/09_python/04/Turtle%20Power.html

Cracking the Code (2013) *Clip on programming a robotic toy car.* Available from www.bbc.co.uk/programmes/p01661yg

Raspberry Pi (n.d.) *Python chat bot activity.* Available from www.raspberrypi.org/learning/turing-test-lessons/lessons/

Selection

Selection is the programming structure through which a computer executes one or other set of instructions according to whether a particular condition is met or not. This ability to do different things, depending on what happens in the computer as the program is run – or out in the real world – lies at the heart of what makes programming such a powerful tool.

Selection is an important part of creating a game in Kodu. An object's behaviour in a game is determined by a set of conditions, for example: WHEN the left arrow is pressed, the object will move left. Similarly, interactions with other objects, variables and environments in Kodu are programmed as sets of WHEN … DO … conditions. For example, WHEN I bump the apple DO eat it AND add 2 points to score. Scratch can also be programmed in this event-driven way (Figure 2.12):

Figure 2.12

Many apps and other programs include this sort of event-driven programming for implementing the user interface: tapping this button or clicking that icon causes the program to respond in a particular way, perhaps changing the stored data and the user's view of it.

In Scratch, Python and other languages, you can build selection into a sequence of instructions, allowing the computer to run different instructions depending on whether a condition is met. For example, this program tests whether a word has more than five letters (Figure 2.13):

Figure 2.13

```
answer = input("Type a word ")
if len(answer)>5:
   print("That's a long word!")
```

Word length tests in Scratch and Python

Notice that the thing which determines whether 'That's a long word!' gets displayed is a test (a 'condition') which is either true or false in the Boolean sense. If it's true then the next bit of code (here say… or print…) gets executed; otherwise it doesn't.

We can use more complex Boolean conditions, for example the somewhat contrived (Figure 2.14):

Figure 2.14

```
answer = int(input("Give me a number! "))
if (answer % 3 == 0) and (not (answer < 1000)):
   print("That's a multiple of three that's not less than a thousand!")
```

Boolean selection in Scratch and Python

Selection statements such as these are at the core of most game programs too, for example (Figure 2.15):

Figure 2.15

It's worth noting that selection statements can be nested inside one another, allowing more-complex sets of conditions to be used to determine what happens in a program. Look at the way some 'if' blocks are inside others in the following script to model a clock in Scratch, which also uses repetition and three variables for the seconds, minutes and hours of the time (see Figure 2.16):

```
forever
  wait 1 secs
  change secs by 1
  if secs = 60 then
    set secs to 0
    change mins by 1
    if mins = 60 then
      set mins to 0
      change hours by 1
      if hours = 24 then
        set hours to 0
```

Figure 2.16

```
from time import sleep

hours=7
minutes=50
seconds=48

while True:
    sleep(1)
    seconds=seconds+1
    if seconds==60:
        seconds=0
        minutes=minutes+1
        if minutes==60:
            minutes=0
            hours=hours+1
            if hours==24:
                hours=0
        print (hours,minutes,seconds)
```

Simple clock programs in Scratch and Python

Notice that in the Python code here and above we use a double == to check for equality; a single = is used to assign a value to a variable.

Selection statements in programming languages typically also include the ability to say what should happen if the condition is false. The usual structure for this is:

```
if <some condition> then:
      <do something>
else:
      <do a different thing>
```

At the core of many educational games are selection statements like this: if the answer is right then give a reward, else say the answer is wrong (see Figure 2.17).

```
ask What is 7 x 8? and wait
if answer = 56 then
  say Well done! for 2 secs
else
  say Think again! for 2 secs
```

Figure 2.17

```
answer=input("what is 7x8? ")
if answer=="56":
    print("Well done!")
else:
    print("Think again!")
```

Tables question in Scratch and Python

See also the Scratch and Python programs for the times tables game on page 75.

Some programming languages, including Python, allow multiple conditions to be combined into a single selection statement, with only the code for the first condition that's true being executed:

```
answer = int(input("What was your mark? "))
if answer >= 70:
   print ("You get an A")
elif answer >= 60:
    print ("You get a B")
elif answer >= 50:
    print ("You get a C")
else:
    print ("You fail!")
```

Grading program in Python

Other languages, such as the functional programming language Haskell, implement something similar through pattern matching:

```
grade :: Integer->String
grade mark
 | mark >=70 = "A"
 | mark >=60 = "B"
 | mark >=50 = "C"
 | otherwise = "fail"
```

Grading function in Haskell

Classroom activity ideas

- Encourage pupils to explore the different conditions which the character in Kodu can respond to in its event-driven programming. Get pupils to think creatively about how they might use these when developing a game of their own. Give them time to design their game, thinking carefully about the algorithm – that is, the rules – they are using.
- Ask pupils to design simple question-and-answer games in Scratch. Encourage them to first think about the overall algorithm for their game before coding this, and then to work to develop the user interface, making this more engaging than just a cat asking lots of questions. It's helpful if pupils have a target audience in mind for software like this.
- Selection can also be used to design computer simulations for real-world systems. Pupils could use Scratch to model the outbreak of a disease, using colours to represent whether a sprite is infected or not, and then nested selection statements to determine if a sprite becomes infected when it touches another carrying the disease. The simulation can be made more sophisticated by adding in further selection criteria, such as natural immunity or whether a sprite has been vaccinated. This approach to simulating complex systems is called agent-based modelling. A more sophisticated approach could use Python's Mesa library[23] or Star Logo TNG.[24]

Further resources

Barefoot Computing (2014) *Selection.* Available from http://barefootcas.org.uk/programme-of-study/use-selection-programs/selection/ (free, but registration required).

Papert, S. (1998) *Does easy do it? Children, games, and learning.* Available from www.papert.org/articles/Doeseasydoit.html

Raspberry Pi (n.d.) *Sorting hat lesson.* Available from www.raspberrypi.org/learning/sorting-hat-lesson/

Berry, M. (2014) *Analogue clock.* Scratch program, available from http://scratch.mit.edu/projects/28742256/#editor [29/12/16].

Berry, M. (2013) *Addition race.* Scratch program, available from http://scratch.mit.edu/projects/15905989/#editor [29/12/16].

Repetition

Repetition in programming means to repeat the execution of certain instructions. This can make a long sequence of instructions much shorter, and typically easier to understand.

Using repetition in programming usually involves spotting that some of the instructions you want the computer to follow are the same, or very similar, and therefore draws on the computational thinking process of pattern recognition/generalisation. You will sometimes hear the repeating block of code referred to as a 'loop', that is, the computer keeps looping through the commands one at a time as they are executed (carried out).

Think about a simple turtle graphics program for a square (Figure 2.18):

23 https://pypi.python.org/pypi/Mesa/
24 http://education.mit.edu/portfolio_page/starlogo-tng/

```
pen down
move 100 steps
turn ↻ 90 degrees
move 100 steps
turn ↻ 90 degrees
move 100 steps
turn ↻ 90 degrees
move 100 steps
turn ↻ 90 degrees
```

Figure 2.18

```
from turtle import *

forward(100)
right(90)
forward(100)
right(90)
forward(100)
right(90)
forward(100)
right(90)
```

Squares (without repetition) in Scratch and Python

Notice how for each side we move forward and then turn right. In Scratch or Python, you could use repetition to simplify the coding for this by using the built-in repeat command, replacing this code with, for example (Figure 2.19):

```
pen down
repeat 4
    move 100 steps
    turn ↻ 90 degrees
```

Figure 2.19

```
from turtle import *

for i in range(4):
    forward(100)
    right(90)
```

Squares in Scratch and Python

Using repetition reduces the amount of typing and makes the program reflect the underlying algorithm more clearly.

Notice that in Python we use a variable to keep track of how many times we've been round the loop. The Python function 'range(4)' is shorthand for the list of numbers 0, 1, 2, 3. The iterator variable 'i' takes each of these values in turn. Thus the program:

```
for i in range(12):
    print(7*i)
```

prints 0, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77 on screen.

To do something similar in Scratch, we'd need to keep track of this ourselves (Figure 2.20):

Figure 2.20

In Snap! this is easier as we have a **for** loop available in the standard tools library (Figure 2.21):

```
for i = 0 to 11
    say 7 × i for 2 secs
```

Figure 2.21

In Python and Snap! the list for the iteration doesn't need to be a sequence of numbers: any list will do. For example (Figure 2.22):

```
for day in ["Monday","Tuesday","Wednesday"]:
    print day
```

```
for each day of list Monday Tuesday Wednesday
  say day for 2 secs
```

Figure 2.22 Snap! using standard tools

In the examples above, the repeated code is run a fixed number of times, which is the best way to introduce the idea. You can also repeat code forever (Figure 2.23):

```
pen down
forever
  move 10 steps
  turn ↻ pick random 0 to 3 * 90 degrees
```

Figure 2.23

```
from turtle import *
from random import randint

while True:
    forward(10)
    right(randint(0,3) * 90)
```

Random walks in Scratch and Python

Notice that the Python code here is a particular version of a 'while' loop (see below), where the condition is always true, so the code inside the loop runs forever.

This can be useful in real-world systems, such as a control program for a digital thermostat, which would continually check the temperature of a room, sending a signal to turn the heating on when this dropped below a certain value. This is a common technique in game programming. For example, the following Scratch code (Figure 2.24) would make a sprite continually chase another around the screen:

Figure 2.24

In event-driven applications, such as a game programmed in Kodu, you can think of all the different event conditions as sitting inside one big 'repeat forever' loop. It's easy to program the same idea in Scratch, as in the following example (Figure 2.25) which uses the W, A, S and D keys to move a sprite around the screen.

```
forever
  if key w pressed? then
    change y by 10
  if key s pressed? then
    change y by -10
  if key d pressed? then
    change x by 10
  if key a pressed? then
    change x by -10
```

Figure 2.25

Much the same thing happens in PyGame Zero's main game loop in Python:[25]

```
while game_has_not_ended():
    process_input()
    update()
    draw()
```

You can nest one repeating block of code inside another. The 'crystal flower' programs in Logo use this idea. For example (Figure 2.26):

25 http://pygame-zero.readthedocs.io/en/latest/hooks.html

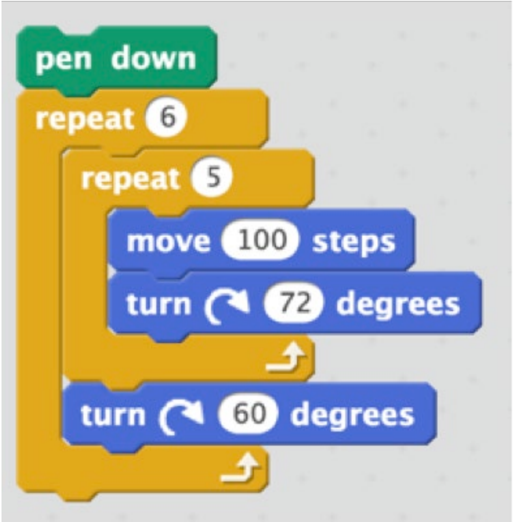


Figure 2.26

```
from turtle import *

for i in range(6):
    for j in range(5):
        forward(100)
        right(72)
    right(60)
```

Repetition can be combined with selection, so that a repeating block of code is repeated as many times as necessary, until or while a certain condition is true. There's a subtle but important distinction here. The <code> in a 'repeat until <condition> <code>' loop is executed when the <condition> is **false**, but the code in the 'while <condition> <code>' loop is executed when the <condition> is **true**.

Compare (Figure 2.27):

set guess to 0
repeat until answer = yes
change guess by 1
ask join join Is your number guess ? (type yes or no) and wait
say join join Your number is guess ! for 2 secs

Figure 2.27[26]

and:

```
guess = 0
answer="no"

while answer == "no":
    guess = guess+1
    answer=input("Is your number " + str(guess)
+"? (type yes or no)")

print("Your number is " + str(guess) +"!")
```

Linear search in Scratch and Python

Scratch and Snap! only provide 'repeat until' loops, whilst Python only offers 'while' loops, which is an important teaching point as pupils move from programming in one language to the other.

Sometimes it can be useful to break out of a loop before the end, and Python provides **break** and 'continue' commands to allow this. The break jumps straight out of the loop, running whatever code comes next:

```
sentence = input('Give me a sentence: ')
firstword = ''
for letter in sentence:
    if letter==' ':
        break
    firstword = firstword + letter
print ('The first word was ' + firstword)
```

Finding the first word of a sentence in Python using break

The **continue**, on the other hand, skips the rest of the code inside the loop but then goes back to beginning of the repeating loop, with the iterator moved on one point:

```
sentence = input('Give me a sentence: ')
nospaces = ''
for letter in sentence:
    if letter==' ':
        continue
    nospaces = nospaces + letter
print ('Without spaces, you get ' + nospaces)
```

Stripping the spaces from a sentence in Python using continue

26 Note that this only works correctly the first time it is run. Can you work out why?

Classroom activity ideas

- Ask pupils to use simple repetition commands to produce a 'fish tank' animation in Scratch, with a number of different sprites each running their own set of repeating motion instructions. This can be made more complex by including some selection commands to change the behaviour of sprites as they touch one another.
- Encourage pupils to experiment with 'crystal flower' programs in Scratch, Logo, Python or other languages that support turtle graphics, and investigate the effect of changing the number of times a loop repeats, as well as the parameters for the commands inside the loop. What combination of numbers produces complete, symmetrical 'flowers'? What numbers produce particularly aesthetically pleasing images? There are some great opportunities to link computing with spiritual, social and cultural education here, noting that traditional Islamic art uses repeated geometric patterns.
- Simple game programming in Scratch, Kodu or PyGame Zero will often use a combination of repetition and selection. Pupils could program a simple, one-player squash game by writing scripts for the ball which make it move repeatedly around the court until hitting the racquet or the back wall. The racquet script could use the event-driven loop above, but restrict movement to just up and down. Scratch has a built-in (**if on edge bounce**) command, so the trickiest thing here is determining what should happen to the ball when it hits the bat. Once pupils have a game like this working, they could adapt it to make a two-player version like the classic Pong video game.

Further resources

Barefoot Computing (2014) *Repetition.* Available from http://barefootcas.org.uk/programme-of-study/use-repetition-programs/repetition (free, but registration required).

Berry, M. (2014) *Scratch 2.0 Fishtank Game* (tutorial). YouTube. Available from www.youtube.com/watch?v=-qTZ5bFEdC8

Code Club World (n.d.) *The Monty Hall problem in Python.* Available from http://projects.codeclubworld.org/en-GB/09_python_archive/05/Gameshow.html qv Krauss and Wang, 2003.

Digital Schoolhouse (n.d.) *Dance scripts.* Available from www.digitalschoolhouse.org.uk/workshops/get-algo-rhythm

Raspberry Pi Learning Resources (n.d.) *Drawing snowflakes with Python Turtle.* Available from www.raspberrypi.org/learning/turtle-snowflakes/ (qv www.youtube.com/watch?v=JO2BTc7s38I).

Modularity

The above ideas of sequence, selection and repetition are covered at Level 2 in the curriculum, but remain conceptually and practically important as pupils continue programming at Level 3 and beyond. At Level 3, pupils should be introduced to modularity in their programming, making use of ideas such as procedures and functions to bring computational thinking concepts like decomposition and abstraction into their coding and planning.

Procedures and functions (and other modular ideas such as classes) allow programs to be written with a far clearer structure, better reflecting the decomposition and abstraction that went into their design: just as we use decomposition to break a problem down into smaller problems, so modularity allows us to build programs up out of smaller parts. Similarly, as abstraction allows us to set to one side details, so modularity means that we can hide the details of specific implementation within procedure, function or class definitions.

Modularity allows for better generalisation too: often someone else might have written a function that solves part of a problem – typically we can simply call that function, perhaps part of a standard library, from our program without generally concerning ourselves with how they implemented this. Useful as it is to know algorithms for search and sort, most software developers will just take those as given. Most of the time, sorting a list in Python involves simply calling the built-in sort function, rather than writing your own code to implement bubble sort, quicksort or one of the other algorithms.

```
sorted ([31, 41, 59, 26, 53, 58, 97])
```

To check if a number is prime or not, you **can** write your own function, or you can just use the 'isprime' function from the number theory module in SymPy:[27]

```
from sympy.ntheory import
isprimeprint(isprime(1301))
```

Modularity also makes it easy for a programmer to **reuse** her own code between different projects, as well-constructed functions and classes can be moved between programs quite easily.

Modularity is important for collaborative software development, as it makes it easy to share out across a team the work of writing software with individuals (or pairs) taking responsibility for implementing the detail of particular elements, according to agreed specifications.

Modularity helps with testing and debugging too, as each function, procedure or class can be tested independently of the others, making sure it does exactly what it's supposed to do. For this to work, it's important that the modular code doesn't introduce unpredictable side effects which affect how other procedures or the main program itself operates.

Modularity also makes it easy to maintain a program, gradually or dramatically improving efficiency. For example, a function which returns the highest common factor of a couple of numbers can be replaced by a more efficient one – the outside program calling this function will work, irrespective of which version of the function is used, but one version is (very much) faster than the other.

Figure 2.28

```
def hcf(a,b):
    bestsofar=1
    test=1
    while test <= a:
        if (a % test == 0) and (b % test == 0):
            bestsofar = test
        test = test + 1
    return bestsofar
```

Inefficient algorithm to find highest common factors in Snap! and Python

Figure 2.29

```
def hcf(a,b):
    if a == 0:
        return b
    else:
        return hcf (b % a, a)
```

Euclid's (recursive) algorithm to find highest common factor in Snap! and Python

Procedures

Procedures are a simple way of using modularity. We group together code with a particular purpose and give it a name. Then, rather than having to type the code each time we want to use it, we simply call the name we've given it.

To create a procedure in Scratch we use the 'Make a Block' button; in Python, we write code to define the procedure:

```
def procedure:
    <procedure code goes here>
```

27 A Python library for symbolic mathematics, www.sympy.org/

Take for example a procedure to draw a square (Figure 2.30):

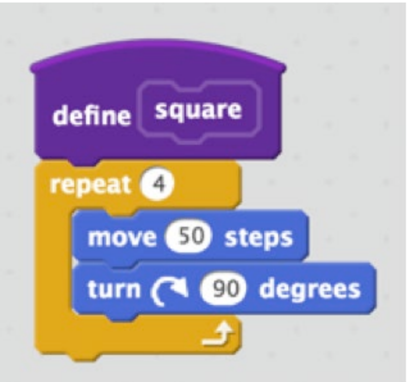


Figure 2.30

```
from turtle import *

def square():
    for i in range(4):
        forward(50)
        right(90)
```

Procedure for a turtle graphics square in Scratch and Python

Then, code to draw a pattern of squares becomes easier to write, and to understand (Figure 2.31):

Figure 2.31

```
setx(-220)
clear()
for i in range(5):
    pendown()
    square()
    penup()
    forward(100)
```

Scratch and Python code for drawing five squares (Figure 2.32) using the above procedure

Figure 2.32

Parameters

You can pass values (numbers or other data such as strings and lists) to procedures, using generalisation to make procedures much more flexible. We call these values 'parameters'. For example, we can generalise our square procedure to make another procedure which draws a regular polygon with sides of any given length (Figure 2.33):

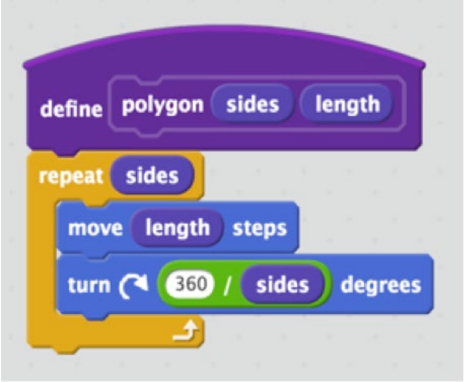


Figure 2.33

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

Procedure for a turtle graphics regular polygon in Scratch and Python

We can then use this procedure to make more-complex patterns (Figure 2.34):

Figure 2.34

```
clear()
edge = 20
for i in range(5):
    polygon(6, edge)
    edge = edge + 20
```

Scratch and Python code to draw a nest of hexagons (Figure 2.35) using the above procedure

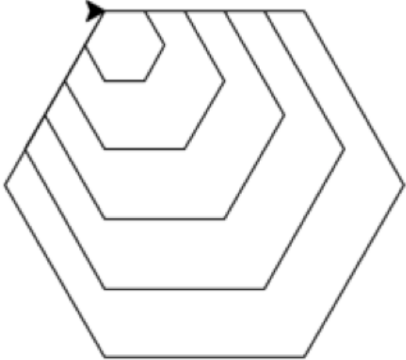


Figure 2.35

Functions

Simply, functions are procedures that return values to the code that called them, which might include another function. Whereas procedures help with structuring what a program does, functions come into their own for decomposing and abstracting the computation that the program performs.

Thus, to print the average (mean) of a list of numbers with a procedure, we might do:

```
def mean(list):
    total = 0.
    for item in list:
        total = total + item
    average = total / len(list)
    print('the mean is ' + str(average))

mean([3,4,5,9])
```

... whereas a function would simply work out the average and then let the outside program decide what to do with that; it could be printing, or it could be updating a record, or using it in another calculation or whatever. This provides much more flexibility.

```
def mean(list):
    total = 0.
    for item in list:
        total = total + item
    return total / len(list)
```

The inability to create functions that return values is one of the limitations of Scratch but Snap! allows this, which makes it useful for a much broader range of computation. Thus, for example, we can write a function which takes a decimal number and returns the binary equivalent (Figure 2.36):

```
+convert+ decimal = 0 +to+binary+
script variables binary
set binary to
repeat until decimal = 0
    set binary to join (decimal mod 2) binary
    set decimal to floor of (decimal / 2)
report binary
```

Figure 2.36

```
def dec2bin(decimal=0):
    if decimal == 0:
        return '0'
    binary = ''
    while decimal > 0:
        binary = str(decimal % 2) + binary
        decimal = decimal // 2
    return (binary)
```

Snap! and Python functions for converting decimal numbers to a string of bits as their binary representation. Note that Python has a built-in function for this: bin()

Snap! and Python also allow **functions** to be passed as arguments to other, higher order, functions. For example, converting a list of numbers into binary can be done using the higher order map function, which takes a function and applies it to every element of a list, returning a list as a result (Figure 2.37):

```
map (convert [ ] to binary) over list 67 65 83
```

Figure 2.37

```
list(map(dec2bin,[65,67,83]))
```

Mapping the decimal-to-binary conversion function above over a list of three numbers

Functions become important in computer science and software engineering later on, partly because it's easier to reason logically and mathematically about what a function does, and also because functions, at least in strict, functional programming languages such as Haskell, **cannot** have side effects, which is important when safety and security are of primary concern.

Recursion

Procedures and functions can refer back to themselves.

Fractals are geometrical figures where each part of the figure is a smaller version of the whole. We can draw these in Scratch or Python by defining a procedure which calls itself (Figure 2.38).

```
define tree (size)
if <(size) > 1> then
  move (size) steps
  turn ↺ 20 degrees
  tree ((size) * 0.5)
  turn ↻ 35 degrees
  tree ((size) * 0.7)
  turn ↺ 15 degrees
  move (( ) - (size)) steps
```

Figure 2.38

```
from turtle import *

def tree(size):
   if size > 1:
     forward(size)
     left(20)
     tree(size*0.5)
     right(35)
     tree(size*0.7)
     left(15)
     forward(-size)
  else:
     return
```

Procedures to draw fractal 'trees' in Scratch and Python

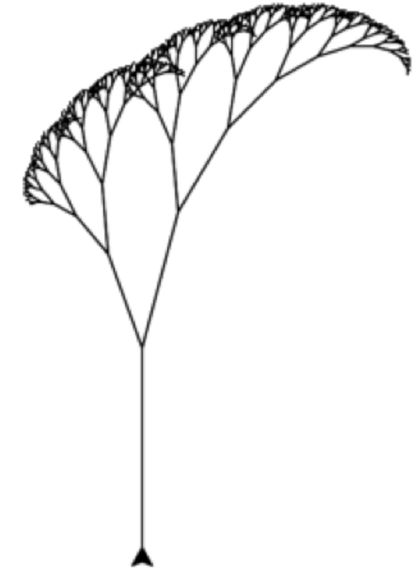


Figure 2.39

We can define functions recursively, for example, to work out the factorial of a number (that is, the product of all the integers up to and including it, represented mathematically as ! – i.e. 4! = 1 x 2 x 3 x 4) we could code (Figure 2.40):

```
+ (n) + ! +
if <(n) = 0>
  report 1
else
  report ((n) × (((n) − (1)) !))
```

Figure 2.40

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial (n-1)
```

Functions (or procedures) that call themselves allow the sort of recursive decomposition of a problem that divide-and-conquer algorithms are built on – you could easily implement binary search this way.

In both of the examples above, note that there's an exit condition inside the recursive function or procedure; otherwise the code could keep calling itself forever.

Whilst recursion is quite a subtle idea, and not one which all pupils are likely to 'get' immediately, once pupils do understand it and can apply it, it becomes a very powerful way of thinking about problems and systems, and can offer a far more elegant way of expressing computational solutions, both as algorithms and as code, than iterating over lists or around loops.

Class

Another example of modularity is the idea of the 'class'. One way of thinking about a class is as setting up a new data structure to store the state or properties of a particular category of things, together with functions or methods that describe the behaviours of things in that category, including how their states might change. A member of the class is called an object, and this approach to programming is described as 'object oriented'.

We might think of a class of cars, and a particular car as an object in that class. The properties of any particular car might include things like its position, its speed, its direction, its engine capacity, its fuel level, its fuel consumption and so on. We could define methods which operate on all objects in the class, such as 'accelerate', 'turn left' or 'put petrol in', which would change some of the properties of any object to which they were applied. We're using abstraction here as we define the class of objects, its properties and methods, to implement those that are relevant for our problem, but also to hide within the definition of the class the details of the implementation.

Whilst most teachers are unlikely to want to teach about classes and objects at Levels 3 and 4, the combination of abstract data structure and associated methods can be useful. For example, we can implement fractions arithmetic by defining a new class in Python and overloading the standard arithmetic operators so that they become methods to operate on objects in this class. The properties of objects in the class are simply the numerator and denominator (in their simplest terms); methods could include creating a new fraction, printing the value of a fraction, adding two fractions together, finding the difference between two fractions and so on.

```
def hcf(a,b):
    if a == 0:
        return b
    else:
        return hcf (b % a, a)

class fraction:

    def __init__(self,top,bottom):
        gcd=hcf(top,bottom)
        self.numerator=top//gcd
        self.denominator=bottom//gcd

    def __str__(self):
        return str(self.numerator)+"/"+str(self.
denominator)

    def __add__(self,other):
        newnum=self.numerator * other.denominator + \
                    other.numerator * self.denominator
        newden=self.denominator * other.denominator
        return fraction(newnum,newden)

    def __sub__(self, other):
        return(self + fraction(-
other.numerator,other.denominator))
```

This implementation of a fraction class in Python always stores each in its lowest terms, using the helper hcf (highest common factor) function to simplify by dividing by the highest common factor of numerator and denominator. Note also how subtraction is defined using the addition method

Notice that a person – perhaps another programmer – using our fraction class can type things like 'print(fraction(2,3)+fraction(1,8))' and get back the correct response without needing to know how this calculation is performed.

Activities

Turtle graphics are a great way to introduce pupils to some of the ideas of procedures. Get pupils to create custom blocks to draw simple shapes (rectangles, squares, trapeziums) and then use these to create a drawing of a house. Get pupils to generalise their procedure for drawing squares so that it draws regular polygons of any size, and explore the repeating patterns which they can use with this as a building block.

Cryptography is a great way to introduce pupils to some of the ideas of functions, creating functions which take plain text (and a key) and encrypt it, or take cipher text (and a key) and then decrypt it. Pupils could extend these ideas further by writing a function which takes a message and converts it to Morse code, or vice versa. Pupils could write

a function to do frequency analysis on a piece of text, counting how many times each letter of the alphabet occurs: one approach would be to write a function to count how many times a particular character comes up, and then to use 'map' to apply this to all 26 letters, producing a list as a result.

Pupils could extend the idea of the fractions arithmetic class to include multiplication, division and comparisons, or even mixed number arithmetic.

Further resources

BBC Bitesize (n.d.) *Procedures and functions.* Available from www.bbc.co.uk/education/guides/zqh49j6/revision; and *Functions, procedures and modules.* Available from www.bbc.co.uk/education/guides/z9hykqt/revision

Berry, M. (2016) *Fractral tree.* Scratch program, available from https://scratch.mit.edu/projects/107864391/#editor [29/12/16].

Berry, M. (2016) *Trees.* Python program, available from https://trinket.io/python/6ffbb68412 [29/12/16].

Berry, M. (2016) *Fractions.* Python class fragment, available https://trinket.io/python/231b9080f3 [29/12/16].

Berry, M. (2016) *Fractions2.* Snap! functions, available from http://snap.berkeley.edu/snapsource/snap.html#present:Username=mgberry&ProjectName=fractions2 [29/12/16].

Berry, M. (2016) *Bindec.* Snap! functions, available from http://snap.berkeley.edu/snapsource/snap.html#present:Username=mgberry&ProjectName=bindec [29/12/16].

Berry, M. (2016) *Binary and decimal conversion.* Python functions, available from https://trinket.io/python/a529b90900 [29/12/16].

Digital Schoolhouse (n.d.) *Generating art: Creating a shape calculator in Scratch.* Available from www.digitalschoolhouse.org.uk/workshops/generating-art-creating-shape-calculator-scratch

Hofstadter, D. (1999) *Gödel, Escher, Bach: An eternal golden braid.* New York, NY: Basic Books Inc.

Papert, S. (1980) *Mindstorms: Children, computers and powerful ideas.* New York, NY: Basic Books Inc.

Raspberry Pi Learning Resources (n.d.) Materials on cryptography in Python. Available from www.raspberrypi.org/learning/secret-agent-chat/

Raspberry Pi Learning Resources (n.d.) *Morse Code decoder.* Available from www.raspberrypi.org/learning/morse-code-virtual-radio/; qv an implementation in microPython for the BBC micro:bit. Available from http://microbit-micropython.readthedocs.io/en/latest/tutorials/network.html

Raspberry Pi Learning Resources (n.d.) Resources on visualising sorting algorithms in Python. Available from www.raspberrypi.org/learning/visualising-sorting-with-python/

Data Structures

Alongside the programming control structures of sequence, selection, repetition and modularity, implementing any algorithm or computational abstraction involves deciding how the computer is going to manage the information to be processed – how the data on which the program draws are to be stored and organised.

Not all data structures are available in all programming languages, although often more-complex data structures can be built up from simpler ones. In object-oriented languages, classes can be created to implement specific data structures out of more primitive ones, as in the example of fractions earlier where we implement a simple fraction data type using Python's tuples (essentially, ordered lists of two elements).

Variables

Pupils are introduced to variables at primary school: a variable is a simple data structure. It is a way of storing one piece of information somewhere in the computer's memory whilst the program is running, and getting that information back later. There's a degree of abstraction involved here – the detail of how the programming language, operating system and hardware manage the storing and retrieving of data from the memory chips inside the computer isn't important to us as programmers, just as these

details aren't important when we're using the clipboard for copying and pasting text. One way of thinking of variables is as labelled shoeboxes, with the difference that the contents don't get removed when they are used.

The concept of a variable is one that many pupils struggle with, and it's worth showing them lots of examples to ensure they grasp this. A classic example which pupils are likely to be familiar with, particularly from computer games, is that of score.

You can use variables to store data input by the person using your program, and then refer to this data later on.

```
ask Hello! What's your name? and wait
set name to answer
say join Hello name for 2 secs
say join name is a very nice name. for 2 secs
```

Figure 2.41

```
name = input ('Hello, what is your name?')
print ('Hello, ' + name)
print (name + ' is a very nice name.')
```

Storing user input in a variable and referring to it in Scratch and Python

Here (Figure 2.41), 'name' is a variable in which we store whatever the user types in; it is then used a couple of times in Scratch's or Python's response. In the case of Scratch, 'answer' is a special temporary variable used to store for the time being whatever the user types in. Notice that variables can store text as well as numbers. Other types of data can be stored in variables too, depending on the particular programming language you are working in.

Variables can also be created by the program, perhaps to store a constant value so that we can refer to it by name (Pi below), or the result of a computation (Circumference in the code below), or random numbers generated by the computer (for example Radius below) (Figure 2.42):

```
set Pi to 3.14
set Radius to pick random 1 to 10
set Circumference to 2 * Pi * Radius
say join If a circle has a radius of Radius for 2 secs
say join Its circumference would be Circumference for 2 secs
```

Figure 2.42

```
pi = 3.14
radius = randint (1,10)
circumference = 2 * pi * radius
print ('If a circle has a radius of ' + str(radius) + 'cm.')
print ('Its circumference would be '' + str(circumference) + 'cm.')
```

Random circumference calculator in Scratch and Python

The idea that the contents of the 'box' are still there after the variable is used is sometimes a confusing one for those learning to program. Have a look at the following code and decide what will be displayed on the screen (Figure 2.43):

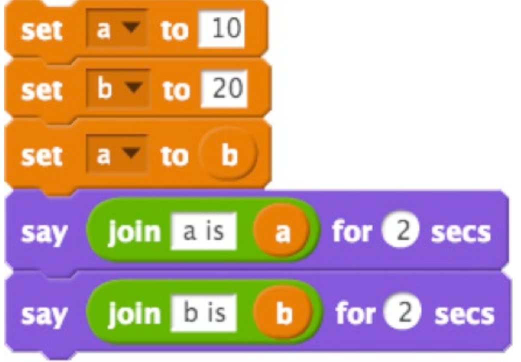


Figure 2.43

```
a = 10
b = 20
a = b
print ('a is ' + str(a))
print ('b is ' + str(b))
```

Variable assignment in Scratch and Python

You should see 'a is 20' followed by 'b is 20'. Try it! Was it easier to understand in Scratch than Python? (Dehnadi and Bornat, 2006; but qv Bornat, 2014).

In Kodu and other game programming, variables are useful for keeping track of rewards, such as a score, and for introducing some sort of limit, such as a time limit or health points that reduce each time you're hit. Kodu's event-driven approach allows particular actions to be done when variables reach a predetermined level.

One particularly useful example of variables in programming is as an iterator – this is a way of keeping track of how many times you've been round a repeating loop and of doing something different each time you do. To do this in Scratch we initialise a counter to zero or one at the beginning of the loop and then add one to it each time we go round the loop. In Python, we can iterate across values in a list. For example, the following program displays the eight times table (Figure 2.44):

```
set i to 1
repeat until i > 12
  say join join i x 8 = i * 8 for .5 secs
  change i by 1
```

Figure 2.44

```
for i in range (1,13):
  print (str(i) + ' x 8 = ' + str(i*8))
```

Code for the eight times table in Scratch and Python. Note that range (1,13) means the integers from 1 up to but not including 13

As we have seen, you can also use an iterator like this to work with strings (words and sentences) one letter at a time, or through lists of data one item at a time. Take care with the beginning and end, as it's all too easy with iterators to start or end too soon or too late.

Classroom activity ideas

- Get pupils to create a mystery function machine in Scratch or Python, which accepts an input, stores this in a variable and then uses mathematical operators to produce an output shown on screen. Setting the display to full screen in Scratch, or running at the command line in Python, pupils can challenge one another (and you) to work out what the program does by trying different inputs.
- Pupils can use variables in their games programs, in say Scratch or Kodu, using a score to reward the player for achieving particular objectives (such as collecting apples) and imposing a time limit.

Further resources

Barefoot Computing (n.d.) *Variables.* Available from http://barefootcas.org.uk/programme-of-study/work-variables/variables/ (free, but registration required).

BBC Bitesize (n.d.) *How do computer programs use variables?* Available from www.bbc.co.uk/guides/zw3dwmn

Berry, M. (2014) *How to program a Scratch 2.0 times table test.* YouTube. Available from www.youtube.com/watch?v=YHGyPfGg1x8

Khan Academy Computing (2013) *Teaching variables: Analogies and approaches.* Available from http://cs-blog.khanacademy.org/2013/09/teaching-variables-analogies-and.html

Notes and tutorial on variables in *Scratch.* Available from http://wiki.scratch.mit.edu/wiki/Variable and https://wiki.scratch.mit.edu/wiki/Variables_Tutorial

Lists

A list is an ordered collection of data, each element of which is of the same type (normally), and where we can reference each by its position in the list. Remember that this is simply an abstraction that allows us to store and retrieve data in the computer's memory, but often it's a very helpful abstraction when we are dealing with lots of related data, such as marks for pupils in a test, words in a sentence, scores in a game, notes in a tune, locations on a route, etc. Just as we might think of variables as a special sort of shoebox in which a single piece of information can be kept, so we could think of a list as a deck of cards, on each of which a piece of information can be recorded.

In order to work with variables, we need only a few basic operations – creating the variable, retrieving information from the variable and storing new information to it. For lists, things are more complex. Scratch, which has a relatively basic implementation of lists as a data structure, allows the following operations (Figure 2.45):

Figure 2.45

These allow data to be added to the end of the list, items to be deleted from the list, items to be inserted at any position in the list, the shifting of items within the list, and the replacement of an item with something else. We can retrieve the data stored at any item in the list, find out how long the list is and check whether a list contains a particular value or not.

The equivalent commands in Python are as follows. Note that Python numbers list elements from zero.

```
list.append('thing')
list.pop(0)
list.insert(0,'thing')
list[0]='thing'
list[0]
len(list)
'thing' in list
```

Both Snap! and Python extend much further the range of commands that can be used to operate on a list. For example, Python includes a function to sort a list into order ('sorted'). Both Snap! and Python allow lists to be made up of other lists, which is one way of providing a multi-dimensional data structure, if that's needed.

To illustrate lists, let's take the example of a shopping list:

- We start with an empty list: [].
- We add milk: ['milk'].
- We add bread: ['milk', 'bread'].
- We add butter: ['milk', 'bread', 'butter'].
- We add eggs: ['milk', 'bread', 'butter', 'eggs'].
- We can sort the list into alphabetical order: ['bread', 'butter', 'eggs', 'milk'].
- We can check if we need to buy quinoa: no, not on our list, this time.
- We buy some eggs, removing that from our list: ['bread, 'butter', 'milk'].
- We remember that we should buy low fat spread rather than butter: ['bread', 'low fat spread', 'milk'].
- We count up how many things we need to buy: three.

Figure 2.46

```
shopping=[]
shopping.append('milk')
shopping.append('bread')
shopping.append('butter')
shopping.append('eggs')
shopping.sort()
print('quinoa' in shopping)
shopping.remove('eggs')
shopping[shopping.index('butter')]='low fat spread'
print (len(shopping))
```

Implementing the above operations on a shopping list in Scratch (see Figure 2.46) and Python. Note that Python allows us to reference elements of the list by their value, whereas Scratch only references by position

Python provides a useful mechanism for 'slicing' a list to extract particular elements or lists of elements. For example, take a list of the first ten prime numbers – primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]. We can get the element primes[0], the last element (primes[-1]) the first three elements (primes[:3]) the last three elements (primes[-3]) or the fourth, fifth and sixth (primes[3:6]). Primes[a,b] is from the a+1 the position (remembering we start at zero) up to – but not including – the b-th position. Negative numbers count from the last item back. It's worth practising with this, as it's simpler than it sounds.

Python also provides 'list comprehensions', which allow new lists with particular properties to be created. For example, we can create a list of the first ten square numbers with the code:

```
squares = [n * n for n in range(1,11)]
```

We can then filter this list to produce another of just the even square numbers from the first ten:

```
evensquares = [s for s in squares if s % 2 == 0 ]
```

Returning to the deck of cards analogy for lists, we might plan how to implement a random shuffle on a list.

One algorithm would be to swap the last card with one of the cards up to and including it, chosen at random. Then, to swap the last but one card with one of the ones up to and including it; then the last but two with one of the ones up to and including it, and so on until we get back to the first card in the pack (Fisher and Yates, 1948 [1938]: 26–27; qv Knuth, 1969).

In Scratch we would code (Figure 2.47):

```
set current place to length of pack
repeat until current place = 1
    set swap with to pick random 1 to current place
    set temp to item current place of pack
    replace item current place of pack with item swap with of pack
    replace item swap with of pack with temp
    change current place by -1
```

Figure 2.47

In Python this would be:

```
from random import randint

for currentplace in range(len(pack)-1,-1,-1):
    swapwith = randint(0,currentplace)

pack[currentplace],pack[swapwith]=pack[swap-
with],pack[currentplace]
```

After shuffling our cards, we might then want to sort them. Combining lists with recursive functions allows us to implement divide-and-conquer algorithms such as quicksort quite elegantly. We can implement quicksort as a function, which calls itself on shorter and shorter lists above and below the pivot for the previous list, until only an empty list is left, which is trivially already sorted (Figure 2.48).

```
quicksort list
if empty? list
    report list
else
    script variables head
    set head to item 1 of list
    script variables tail
    set tail to all but first of list
    report join
        quicksort keep items such that ( < head) from tail
        list head
        quicksort
            keep items such that (( = head) or ( > head)) from tail
```

Figure 2.48

```
def quicksort(list):
    if len(list)==0:
        return list
    else:
        head = list[0]
        tail = list[1:]
        lower = [x for x in tail if x <  head]
        upper = [x for x in tail if x >= head]
        return quicksort(lower) + [head] + quicksort(upper)
```

Recursive quicksort in Snap! and Python. Notice the escape clause common to the conquer stage of recursive divide-and-conquer algorithms

Classroom activity ideas

- Pupils could create lists of notes, perhaps with a paired list of durations, to play in Scratch or Sonic Pi.
- Pupils could explore implementing the 'perfect' riffle shuffle on a deck of cards using list manipulations, splitting the pack in two equal halves and taking cards alternately from top and bottom halves (Diaconis et al., 1983).
- Pupils could implement one or more of the sort algorithms discussed on pages 20 - 21 using list manipulations.
- Can pupils write programs to compute descriptive statistics for lists of numerical values (for example, to find the mean, median and mode, the minimum or maximum, the total, the standard deviation)?
- Can pupils write a program which would generate valid National Lottery results giving results as ordered lists, noting that no number may occur more than once)?

Further resources

Code Club World (n.d.) *Comment generator.* Available from http://projects.codeclubworld.org/en-GB/09_python_archive/06/Compliment%20Generator.html

Computing at School (n.d.) *Fun with lists* by Mark Tranter, looking at how Python's built in list functions could be implemented. Available from http://community.computingatschool.org.uk/resources/2683 (free registration).

Harvey, B. (1997) *Computer science logo style: Symbolic computing* (volume 1). Boston, MA: MIT Press.

Raspberry Pi Learning Resources (n.d.) *Magic 8 ball.* Available from www.raspberrypi.org/learning/magic-8-ball/

Raspberry Pi Learning Resources (n.d.) *Sonic Pi lesson on data structures.* Available from https://www.raspberrypi.org/learning/sonic-pi-lessons/lesson-4/plan/

Other data structures

Other data structures are available.

Whilst Scratch does a good job of hiding this from the programmer, variables themselves are more complex than they might appear, as a variable might store data of one of several different **data types:** perhaps a number, but also possibly a Boolean value (true or false), or text, each of which might be represented quite differently inside the computer, and for each of which only certain operations would make sense.

Some programming languages are much more demanding ('strongly typed') in their treatment of data types, demanding that these be declared explicitly before the variable is ever used. Python is relatively easy-going about data types, but even in Python it is sometimes necessary explicitly to change (or 'cast') a variable from one type to another. In some of the examples above we use 'str(x)' to convert a number, 'x', into a string so that we can join it to other strings; or 'int(y)' to take a string of user input and convert it into a number so that we can compare it with other numbers.

Strings of text are quite different from numbers, and can be thought of as simply lists of letters or other characters. Thus, some of the operations we might perform on a list also make sense when working with strings – it makes sense to ask how long a string is, to be able to reference particular characters directly, to be able to replace one character with another (such as converting a string between different cases), or to be able to join two strings together (concatenation).

Scratch provides a few blocks for working with strings (Figure 2.49):

Figure 2.49

The equivalent commands in Python are:

```
'hello ' + 'world'
'world'[0]
len('world')
```

Python provides the same slicing tools for strings as it does for lists, thus if:

```
cas='Computing At School'
```

then:

- cas[0] is C
- cas[-1] is l
- cas[:9] is Computing
- cas[-6:] is School
- cas[10:12] is At

But as with lists, Python's string-handling capabilities extend far beyond this.[28]

A variable can be thought of as a single number and a list as an ordered, one-dimensional set of numbers. We could also have two-dimensional

28 See https://docs.python.org/3.5/tutorial/introduction.html#strings and https://docs.python.org/3.5/library/stdtypes.html#string-methods. Python's Natural Language Toolkit provides powerful libraries for working with larger bodies of text: www.nltk.org/

(or greater) collections of data. These are called **arrays**. Python doesn't support arrays as standard (although it offers good support through the NumPy library[29]), nor does Scratch. It is possible in Python and Snap! to construct higher-dimensional collections of data using nested lists, but this is unlikely to be particularly appealing or accessible to pupils at this stage.

Rather than working in Python or Snap! for two-dimensional arrays of data, revisiting Excel may be much more useful and accessible, given the direct and immediate view of all the contents of the array. Excel (and other spreadsheet software such as Google Sheets) can be used for genuinely two-dimensional data, such as heights, temperatures or rainfall at locations on a grid; or the presence or absence of a cell in a Life simulation; or greyscale values for a monochrome pixel bitmap. It allows calculations to be done to and with these data.

Many pupils might be familiar with Minecraft, in which the world is represented as a 3D array of data about the block at each location. Strategy or construction games, such as Sim City or Civilisation, represent the world as a 2D array. Pupils can develop an understanding of arrays, and practise their Python programming skills, using a Python API for Minecraft, which is provided as standard for the Raspberry Pi and is also available, with a little ingenuity, on other platforms. In the Raspberry Pi version of Minecraft, we can create a floating cube of 1000 stone blocks using this code:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
stone = 1
x, y, z = mc.player.getPos()
mc.setBlocks(x+1, y+1, z+1, x+11, y+11, z+11, stone)
```

Notice the three parameters necessary to specify locations in this 3D virtual world (see Figure 2.50).

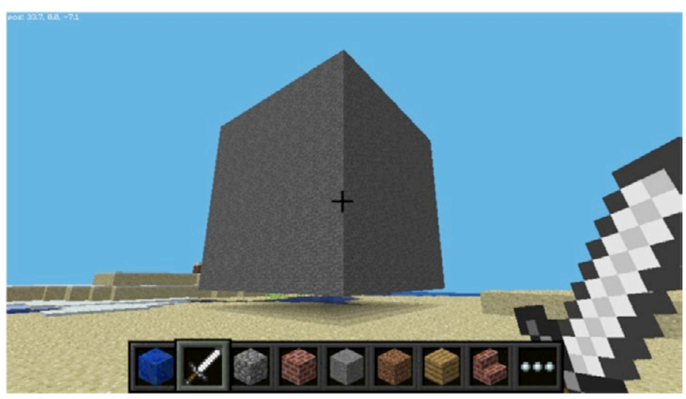


Figure 2.50[30]

Whilst a **table** in a spreadsheet could be a 2D array of data, more often we might think of a table of values in a spreadsheet or a database as providing a structured collection of data about a number of different things – each row of the table becomes a record of an individual case, each column the different fields of those records. For example, a spreadsheet might be used by a teacher to track assessment data for pupils in her class, with each row storing data on an individual pupil, and each column recording attainment on particular tasks or tests, as well as additional personal information such as name, roll number and date of birth.

	A	B	C	D	E	F	G	H	I	J	K
1	Name	Target		percentage	mark	grade	feedback	percentage	mark	grade	feedback
2				Read throug	Read throug	Read throug	Read throug	Use two of t	Use two of t	Use two of t	Use two of t
3	Sean Abbey	C		71	25	B	be pleased	54	19	C	understood
4	Jeremy Also	B		40	14	C	good	69	24	B	be pleased
5	Laura Barrist	A		29	10	D	have been	54	19	C	understood
6	Emma Brigg	A		71	25	B	some good	94	33	A	like the way
7	Jonathan Ca	A		71	25	B	be pleased	66	23	B	very
8	Pedro Cerve	B		74	26	B	some good	86	30	A	You have
9	Kerry Crosby	B		54	19	B	some good	43	15	C	understood
10	Emma Darar	C		60	21	B	be pleased	71	25	B	be pleased
11	Sandra Dubr	B		60	21	B	be pleased	71	25	B	very
12	Andrew Ebb	B		74	26	B	very	74	26	B	some good
13	Carl Effen	B		63	22	B	some good	80	28	B	some good
14	Joseph Goul	A		29	10	D	have been	34	12	D	really good
15	Isha Hussain	C		49	17	C	good	71	25	B	be pleased
16	Paula Jake	B		40	14	C	understood	60	21	C	good
17	Jonny James	A		29	10	D	too many	34	12	D	good
18	John Karlos	A		37	13	C	understood	77	27	B	some good
19	Nicholas Loc	A		74	26	B	There are so	86	30	A	Some really
20	Hannah Mar	B		74	26	B	very	89	31	A	You have
21	Yvonne McD	C		34	12	C	understood				
22	Penny Parke	B		57	20	B	some good	57	20	C	really good
23	Polly Pocket	B		46	16	C	understood	63	22	B	very
24	Ryan Queen	B		54	19	B	very	66	23	B	be pleased
25	Deborah Qu	C		49	17	C	really good	66	23	B	some good
26	Nicola Stava	A		37	13	C	understood	77	27	B	some good
27	Sally Studen	A		49	17	C	understood	71	25	B	very
28	Ian Westerly	A		57	20	B	some good	57	20	C	good

	L	M	N	O	P	Q	R	S	T	U	V	W
1	percentage	mark	grade	feedback	percentage	mark	grade	feedback	percentage	mark	grade	feedback
2	Look at the	Look at the	Look at the	Look at the	Read up to p	Read up to p	Read up to p	Read up to p	Produce a su	Produce a su	Produce a su	Produce a su
3	57	57	C	This is a real	69	69	A	You have			B	some good
4	51	51	C	You have un	74	74	A	You have			D	have been
5	43	43	C	You have un	71	71	A	good work			D	too many
6	71	71	B	You should t	91	91	A*	good work			B	some good
7	57	57	C	This is a goo	66	66	B	be pleased			C	understood
8	57	57	C	This is a goo	94	94	A	like the way			B	be pleased
9	49	49	C	This is a real	74	74	A	good work			B	very
10					63	63	B	be pleased			C	understood
11	46	46	B	You have un	63	63	B	some good			C	understood
12	54	54	C	This is a goo	74	74	A	You have			C	understood
13	54	54	C	You have un	80	80	A	good work			B	very
14	37	37	C	This is a goo	66	66	B	very			C	really good
15	57	57	C	This is a real	83	83	A	You have			B	very
16	34	34	C	This is a goo	69	69	A	like the way			C	really good
17	37	37	C	This is a goo	66	66	B	be pleased			C	really good
18	57	57	C	This is a real	60	60	B	very			B	be pleased
19	57	57	C	This is a goo	94	94	A*	Excellent! I like the way you have disc			B	This is a very
20	51	51	C	This is a real	83	83	A	like the way			B	be pleased
21	49	49	C	You have un	77	77	A	like the way				
22	60	60	C	This is a goo	74	74	A	You have			C	really good
23	46	46	C	You have un	66	66	B	some good			C	understood
24	46	46	C	You have un	80	80	A	like the way			B	very
25	46	46	C	You have un	80	80	A	You have			B	some good
26	40	40	C	You have un	60	60	B	some good			B	be pleased
27					89	89	A	like the way			B	very
28	60	60	C	This is a real	74	74	A	You have			C	understood

(Anonymised example of teacher's mark book, with thanks to Firefly Learning)

29 www.numpy.org/

30 From www.raspberrypi.org/learning/getting-started-with-minecraft-pi/worksheet/, CC by-sa Raspberry Pi Foundation.

The spreadsheet is really then being used as a single-table database, and our focus moves somewhat from performing computation to managing and processing this structured information.

Further tables, typically in a **database** now rather than a spreadsheet, could link to these data, perhaps providing further personal details, or information about the objectives of each assessment. Relating tables of data in this way means that we only need to store information once, in one place, but can use it in many different ways – typically we would use other software to manage this database (a 'relational database management system' [RDBMS], such as SQLite, MySQL or Microsoft Access).

Pupils learn to 'structure related items of information' at Level 2. At Level 3, pupils should 'represent and manipulate structured information in programs, or databases'. Pupils certainly don't need to create database management programs themselves in order to do this, but can write programs using APIs to work with data stored in a standard database.

Whilst not supporting arrays or databases without additional libraries, Python does include another very useful data structure, the **dictionary**. Like a normal dictionary, this makes it easy to look up one value (the key) and get back a bit of associated information: the value. Unlike normal dictionaries, Python dictionaries aren't in any particular order – they are unordered collections of key:value pairs, where the value is stored or retrieved using the associated key.

For example, the command:

```
languages=dict([('Alex', 'Python'), \
                ('Bobbie', 'Snap!'), \
                ('Chris', 'Python'), \
                ('Drew', 'Scratch'), \
                ('Elliott', 'Visual Basic')])
```

creates a new dict, 'languages', in which we might store the preferred language of each of five students. Notice that no two students have the same name, but a couple of them like the same language – which is fine.

We can type:

```
print(languages['Alex'])
```

and get back the response 'Python', as expected.

We can change an entry too:

```
languages['Drew']='Snap!'
```

We can remove an entry from the dict:

```
del languages['Bobbie']
```

And we can add a new entry very simply:

```
languages['Frankie']='Kodu'
```

Graphs were discussed on page 30 as a particularly useful form of computational abstraction. Graphs can be stored and manipulated programmatically as lists of edges, plus perhaps associated 'weights', or as an array showing which nodes are connected to which, with the weight of the edge given in the array. Python has a number of libraries for working with graphs, including NetworkX,[31] and a variant of Snap!, Edgy, provides tools for dealing with graphs using a block-based language.

As mentioned earlier (page 72), object orientation allows programmers to define their own **classes** of abstract data types – in which different properties of objects can be drawn together – as well as the methods which operate on objects in those classes.

Classroom activity ideas

- You can set pupils many challenges with string handling – can they remove all the spaces or all the vowels from a sentence? Can they reverse the order of letters in a word? Can they make a list of all the words in a sentence and order these alphabetically? Can they count how many times each letter occurs in a piece of text? And so on.
- Can pupils use a dictionary to convert text into Morse code or vice versa? Can they then play the Morse code?

31 https://networkx.github.io/

- If you have access to Raspberry Pis (or can install Python scripting for Minecraft on another platform) encourage pupils to experiment building things, or changing things, in Minecraft using Python programming. Can they create charts in Minecraft to visualise data? Can they import a low-res photo into Minecraft?
- If you have a school weather station, could pupils use their programming to interface with this, adding readings to an external database, or analysing or visualising data from a database of weather records?

Further resources

BBC Bitesize (n.d.) *Arrays and lists.* Available from www.bbc.co.uk/education/guides/zy9thyc/revision

BBC Bitesize (n.d.) *Databases.* Available from www.bbc.co.uk/education/topics/zwm6fg8

Code Club World (n.d.) Introductory Python game inspired by Minecraft, but treating the world as a 2D array. Available from http://projects.codeclubworld.org/en-GB/08_python_02/06/CodeCraft.html

Code Club World (n.d.) Project using a dictionary to convert 'text' speak into English. Available from http://projects.codeclubworld.org/en-GB/09_python_archive/08/Text-speak%20Converter.html

Raspberry Pi Learning Resources (n.d.) Activity covering lists, dictionaries and comprehensions. Available from www.raspberrypi.org/learning/n-days-of-christmas/

Raspberry Pi Learning Resources (n.d.) *Morse Code decoder.* Available from www.raspberrypi.org/learning/morse-code-virtual-radio/; qv an implementation in microPython for the BBC micro:bit. Available from http://microbit-micropython.readthedocs.io/en/latest/tutorials/network.html

Raspberry Pi Learning Resources (n.d.) Resources on *getting started with Minecraft Pi.* Available from www.raspberrypi.org/learning/getting-started-with-minecraft-pi/

Richardson, C. (2015) *Learn to program with Minecraft.* San Francisco, CA: No Starch Press.

Whale, D. and O'Hanlan, M. (2014) *Adventures in Minecraft.* Hoboken, NJ: Wiley.

Can we fix the Code?

Back in the days when there were very few computers, which took up a whole room and used electro-mechanical relays rather than transistors, there was a story of one machine that just wouldn't work as it should – careful investigations revealed that a moth, a literal 'bug', had become lodged between the blades of a relay switch, stopping the switch from closing and thus the computer from operating.

Errors in algorithms and code are still called 'bugs', and the process of finding and fixing these is called 'debugging'. Debugging can often take much longer than writing the code in the first place, and whilst fixing a program so that it does work can bring a great buzz, staring at code that still won't work, apparently no matter what you do, can be the cause of great frustration too: this can be tricky to manage in class. It is worth spending time getting the code right in the first place, through careful planning, logical reasoning and a good command of the language, rather than having to spend time fixing things later. Not all bugs get spotted, and those that don't can have profound consequences.[32]

Bugs fall into two main categories – those in the algorithms, which sometimes are called logic bugs, and are often due to not quite understanding the problem properly; and those in the code.

In text-based languages, many of the bugs in the code are 'syntax' errors, where the formal rules of the language's vocabulary and grammar haven't been adhered to, and so the computer isn't able to turn the code you've written into machine code that its CPU can execute. Not all software engineers see these as 'bugs', merely as relatively easy-to-fix syntax errors which a good IDE or text editor should prevent from getting made in the first place. Seemingly cryptic error messages about the syntax

32 See, for example, www.theregister.co.uk/2014/04/09/heartbleed_explained

error are generated, which are at least a starting point for identifying exactly where indentation or a colon has been missed out or similar. This can be a good teaching opportunity for emphasising the importance of spelling, punctuation and grammar in **all** pupils' work.

In graphical languages like Kodu and Scratch it's almost impossible to make syntax errors, so as pupils make the transition from graphical- to text-based languages, much of their time might be spent on getting the syntax right.

Much more important than these syntax errors are the logical or semantic errors, where the code runs or compiles perfectly but doesn't quite do what is intended. These errors are more likely to be about having the wrong algorithm, of not translating the ideas of the algorithm into code quite correctly, or sometimes misunderstanding the semantics – the meaning – of the commands of the language or even of how the computer itself operates. Because it's normally clear when a program isn't working properly (particularly if we test programs, procedures and functions carefully), it is often easier to address misconceptions like these in computing than it is in other subjects, where feedback is less immediate.

Sometimes bugs in algorithms or code only become apparent in certain circumstances – a program might function perfectly well most of the time and then crash (suddenly stop working) very occasionally. For example, normally a program could find the mean of a list of numbers by adding them up and dividing by how many there are, but if given an empty list, it might attempt to divide by zero, which in some programming languages would cause the code to crash. Creating a good, comprehensive set of test data with known outcomes is important for tracking down these sorts of bugs in a systematic way, but even working out from the input what caused the crash is a great chance for pupils to put logical reasoning to work.

From primary school onwards, pupils should be taught to **use logical reasoning** to detect and correct errors in algorithms and programs, so it's not really enough for pupils to fix their code without being able to give an explanation for what went wrong and how they fixed this. In programming classes, pupils focussed on the task of writing a program for a particular goal might want help from you or others to fix their programs. Tempting as this may be, it's worth you and they remembering that the objective in class is not to get a working program but to learn how to program, and their being able to debug their own code is a big part of that.

One way that you can, and should, help is to provide a reasonably robust, general set of debugging strategies which they can use for any programming, or indeed more-general strategies which they can use when they encounter problems elsewhere.

The Barefoot Computing team suggests a simple set of four points, emphasising the importance of logical reasoning:

1. Predict what should happen.
2. Find out exactly what actually happens.
3. Work out where something has gone wrong.
4. Fix it.

One way to help predict what should happen is to get pupils to explain their algorithm and code to someone else (or even an inanimate object such as a rubber duck) – in doing so, it's quite likely that they will spot where there's an error in the way they are thinking about the problem or in the way they've coded the solution.

In finding out exactly what happens, it can be useful to work through the code, line by line. Seymour Papert described this as 'playing turtle': in a turtle graphics program, pupils could act out the role of the turtle, walking and turning as they follow the commands in the language themselves, or following the instructions with a pencil on paper. Away from the easily visualised world of turtle graphics, pupils could maintain a trace table, keeping a record of the values of variables and lists as they step through their code one line at a time. Some IDEs include debuggers, allowing this process to be automated. However, the careful thought involved in doing this by hand might still make it easier to spot, and learn from, what's going wrong.

In working out where something has gone wrong, encourage pupils to look back at their algorithms before they look at their code: before they can get started with fixing bugs, they will need to establish whether it was an issue with their **thinking** or with the way they've implemented it. Another technique is to use something like the 'divide and conquer' algorithm (for guessing a hidden number) to find a bug – working out whether the bug is in the first or second half of the code, or in the first or second

quarter, and so on. Sometimes this is called 'wolf-fencing' – i.e. to find the wolf, build a fence and listen whether the howl comes from one side or another; repeat with smaller and smaller areas until you find the wolf.

Debugging is a great opportunity for pupils to learn from their mistakes and to get better at programming. Encourage your pupils to adopt a 'growth mindset', making the most of the opportunities their bugs present them to learn more about how to program.

Classroom activity ideas

- Pupils are likely to make many authentic errors in their own code, which they will want to fix. You might find that it's worth spending some time giving pupils some bugs to find and fix in other programs, both as a way to help develop strategies for debugging and to help with assessment of logical reasoning and programming knowledge. Create some programs with deliberate mistakes in, perhaps using a range of logical or semantic errors, and set pupils the challenge of finding and fixing these. For example, can pupils find all the errors in the following Python program, designed to ask ten different multiplication-test questions?

```
import Random
a = random.randint(1,12)
b = random.randint(1,12)
for i in range(l0):
  question = "What is "+a+" x "+b+"? "
  answer = input(question)
  if answer = a*b
    print (Well done!)
  else:
    print("No.")
```
[33]

- Encourage pupils to debug one another's code. One approach is for pupils to work on their own program for the first part of the lesson and then to take over their partner's project, completing and then debugging it for their friend.
- A similar paired activity is for pupils to write code with deliberate mistakes, setting a challenge to their partner to find and then fix the known errors in the code.[34]

Further resources

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

BBC Bitesize (n.d.) *What is debugging?* Available from www.bbc.co.uk/guides/ztkx6sg; *Writing error-free code.* Available from www.bbc.co.uk/education/guides/zcjfyrd/revision

Brennan, K. (2010) *Debug it!* ScratchEd. Available from http://scratched.gse.harvard.edu/resources/debug-it

Cutts, Q. (2007) *Cribsheet.* Available from http://level1wiki.wikidot.com/cribsheet

Jonassen, D.H. (2004) *Learning to solve problems: An instructional design guide* (volume 6). Hoboken, NJ: John Wiley & Sons.

Berry, M (2014) *Switched on Computing Scratch Projects.* Available from http://scratch.mit.edu/studios/306100/ [29/12/16].

Wikipedia (n.d.) *Rubber duck debugging.* Available from http://en.wikipedia.org/wiki/Rubber_duck_debugging

33 Try online at https://trinket.io/library/trinkets/2da63b4823
34 See https://teachcomputing.wordpress.com/2013/11/23/sabotage-teach-debugging-by-stealth/

References

Bornat, R. (2014) *Camels and humps: A retraction.* Middlesex University. Available from www.eis.mdx.ac.uk/staffpages/r_bornat/papers/camel_hump_retraction.pdf

Dehnadi, S. and Bornat, R. (2006) *The camel has two humps* (working title). Middlesex University. Available at http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf (accessed 28/12/16).

Diaconis, P., Graham, R.L. and Kantor, W.M. (1983) The mathematics of perfect shuffles. *Advances in Applied Mathematics* 4 (2). 175–196.

Dorling, M. and White, D. (2015) Scratch: A Way to Logo and Python. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education.* ACM. 191–196.

Dorling, M. and Rouse, G. (eds) (2014) *Compute-IT.* London: Hodder.

Fisher, R.A. and Yates, F. (1948) [1938]. *Statistical tables for biological, agricultural and medical research* (3rd edition). London: Oliver & Boyd.

Grover, S. (2016) Classroom strategies. *CSTA Voice* 12 (1). 7-8.

Harvey, B. (1997) *Computer science logo style: Symbolic computing* (volume 1). Boston, MA: MIT Press.

Jackson, S. and Livingstone, I. (1982) *The warlock of firetop mountain.* London: Puffin.

Knuth, D.E. (1969) *Seminumerical algorithms. The art of computer programming 2.* Reading, MA: Addison–Wesley.

Krauss, S. and Wang, X.T. (2003) The psychology of the Monty Hall problem: Discovering psychological mechanisms for solving a tenacious brain teaser. *Journal of Experimental Psychology: General,* 132 (1). 3-22.

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

Peyton Jones, S., Blackwell, A. and Burnett, M. (2003) A user-centred approach to functions in Excel. *ACM SIGPLAN Notices,* 38 (9). 165–176.

Robins, A. (2010) Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education,* 20 (1). 37–71.

Shneiderman, B. and Mayer, R. (1979) Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences,* 8 (3). 219–238.

Systems

What is a computer?

Systems

WHAT IS A COMPUTER?

The term 'computer' originally referred to **people** whose job it was to perform repeated numerical calculations according to some pre-determined set of instructions; that is, an algorithm. At the beginning of modern computing, Alan Turing captured the essence of what human computers did – of what calculation or computation were: that all this could be understood as making or changing marks on paper according to some set of rules, and that those rules could be determined by the marks on the paper. This model became known as the Turing machine, and it still forms one of the foundations of theoretical computer science.

Since the 1940s the term 'computer' has been used pretty much exclusively to refer to digital machines which accept some sort of input data, process this according to some set of stored instructions (that is, a program) and output some sort of information.

The power of digital computers comes from their ability to run through these stored instructions incredibly quickly: the chip at the heart of a modern smartphone might execute up to a couple of billion instructions per second! On the other hand, without programming, a computer can do nothing – it needs to be given instructions to follow.

You can think of digital technology as baing made up of two inter-related systems: the **hardware**, being the physical components, from processor and memory to power supply and screen; and the **software**, being the core operating system, embedded control programs, compilers or interpreters for high-level programming languages, and all the many application programs used by or written by the computer's user.

Computers now seem almost ubiquitous, with an incredible variety of electronic devices each having some sort of digital computer controlling how they operate, according to stored programs. It's worth distinguishing between devices that contain computers, where the computer controls the operation of the device for one specific purpose, and more-general programmable computers, where one computer can do many different things.

In the former category, the computer-controlled device, we might count digital watches, digital radios, digital televisions, computerised central heating controllers, digital cameras, the engine management system of a car and many, many other devices now commonplace. Even in these categories, convergence of technologies and the Internet Of Things[(1)] has meant that previously 'dumb' digital devices such as watches, televisions, cameras and cars can now connect to the internet, have apps installed on them and, in some cases, be reprogrammed by their users.

When thinking about general-purpose computers (see Doctorow, 2012), it can sometimes be helpful to distinguish between those which the user themselves can program and those which can only run software written specifically for the device. For example, a smart TV or a games console could be thought of as a general-purpose computer, capable of doing many different things, and whilst it's possible to create smart TV apps or write a video game, you would normally need to use another computer to do that. Originally, smartphones and tablet computers fell into this category too: they would only be able to run programs written and licensed for them on other systems, but tools such as TouchDevelop and Codea now mean that programs for devices like these can be written on the smartphone or tablet itself. Indeed, even some games consoles can be directly programmed, to a limited extent, using programs such as Kodu and Project Spark.

General-purpose programmable computers, from laptops to the large and fast computers running data centres and 'cloud computing', are capable of running many, many different types of program, including the compilers or interpreters necessary to write and run programs in many different programming languages. At a theoretical level, if something can be computed by one system that meets certain basic conditions,[(2)] it can be computed by any system that meets those conditions.

1 Physical objects that collect, share and access data via the internet; see, for example, https://en.wikipedia.org/wiki/Internet_of_Things
2 We call these conditions Turing completeness – systems that can simulate Turing's theoretical computing machine can, in theory, simulate one another; see, for example, https://en.wikipedia.org/wiki/Turing_completeness

It's not **quite** true that programming lets us solve any problem we might imagine.[3] However, by using computational thinking processes to understand a problem and develop algorithms for solving it, and then to write the computer code which implements that algorithm as a program, on hardware that accepts input, produces output and connects to other machines, computers can be used to solve many, many interesting and difficult real-world problems, as well as allowing us to watch videos of cats playing the piano.

Binary

All the data that computers work with, and all the instructions they follow, have to be represented as numbers. There are particular conventions or codes for particular types of data (or instructions). An understanding of the ways in which information can be represented, organised and processed by computers can be seen as of comparable importance to the concepts of computational thinking (Michaelson, 2015).

Binary Numbers

Whereas we think of numbers as expressed in base 10 (decimal) notation, expressing any (whole) number using our digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, it's much, much easier for computers to work with just two symbols, a 0 and 1, as each 'switch' in the computer can be simply set as off or on[4] to represent these – this is the case with modern integrated circuits as well as the relays, valves and discreet transistors that preceded them. Binary representation isn't really fundamental to the ideas of computing (Brown, 2012), but the numerical representation of information and instructions is, and binary is important in the low level **implementation** of digital computing on current and past hardware.

The programme of study expects pupils to learn about binary:

> **Demonstrates an understanding that all computer data is represented in binary, for example, numbers, text, black and white graphics**

In base 10 (decimal or 'denary' notation), we use place value so that the same digit can represent different numbers:

Thousands	Hundreds	Tens	Units
2	3	9	5

Thus, 2395 is interpreted as two thousands, three hundreds, nine tens and five units, 2000+300+90+5. Note how each place is ten times larger than the one that follows it.

A similar place value system works in binary, but the places carry twice the value of the following one:

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

So 101010 is interpreted as one thirty-two, one eight and one two, 32+8+2, that is, 42 in base ten. Note that you **never** get more than one in each place in binary, so converting a binary number to a decimal one is simply the process of adding up the respective place values. Given the number 10011101, simply write out the place values of each 'bit' (binary digit), starting at the right (the least significant bit) and doubling each time:

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

Then, add up the values of the places where you have a 1: 128+16+8+4+1=157.

This approach gives us an algorithm for converting from binary to decimal, which, of course, we can implement as code (Figure 3.1):

3 For example, computers cannot solve the halting problem – that is, they cannot determine if any arbitrary code would terminate or not (Turing, 1936).

4 More strictly, low or high voltages.

```
convert binary = 0 to decimal
script variables i total place value
set i to length of binary
set total to 0
set place value to 1
repeat length of binary
    set total to total + letter i of binary × place value
    set place value to 2 × place value
    change i by -1
report total
```

Figure 3.1

```
def bin2dec(binary='0'):
    decimal = 0
    place = 1
    for i in range(len(binary)-1,-1,-1):
        decimal = decimal + int(binary[i])*place
        place=place*2
    return (decimal)
```

Binary to decimal conversions in Snap! and Python. Python allows numbers to be input in binary, they are stored internally in binary, as all numbers are, but are printed in decimal. Thus print(0b101010) produces the output 42

Going in the other direction is easy enough too. The easiest approach is to start with writing down place value headings, again starting at the right (the least significant bit) and doubling until you get to a column that would mean the next one would be bigger than the number. So with 150, we would have column headings:

128 64 32 16 8 4 2 1

We then start at the left, including any place we can, and keeping track of how much is left. Taking 150 as our example:

1 (and 22 left)

1 0 0 1 (and 6 left)

1 0 0 1 0 1 (and 2 left)

1 0 0 1 0 1 1 (and 0 left)

1 0 0 1 0 1 1 0

There are quicker approaches, for example we could use repeated division by two, keeping track of the remainders.

150 / 2 = 75 r 0
75 / 2 = 37 r 1
37 / 2 = 18 r 1
18 / 2 = 9 r 0
9 / 2 = 4 r 1
4 / 2 = 2 r 0
2 / 2 = 1 r 0
1/ 2 = 0 r 1

The remainders then give the binary, in reverse order, so reading from the bottom up, we get 10010110 which is the binary representation for 150 as above.

Because of the repetition here, it's relatively easy to code this algorithm (Figure 3.2):

```
convert decimal = 0 to binary
script variables binary
set binary to
repeat until decimal = 0
    set binary to join decimal mod 2 binary
    set decimal to floor of decimal / 2
report binary
```

Figure 3.2

```
def dec2bin(decimal=0):
    if decimal == 0:
        return '0'
    binary = ''
    while decimal > 0:
        binary = str(decimal % 2) + binary
        decimal = decimal // 2
    return (binary)
```

Decimal to binary conversion functions in Snap! and Python. Python has a built-in bin command to do this too

It's possible to represent numbers less than one in binary too, using the binary equivalent of decimal numbers, sometimes called 'bicimal'. So, just as in decimal we extend place value to the right as tenths, hundredths, thousandths and so on, each place being a tenth the size of the previous one, so in binary we halve each time: one-half, one-quarter, one-eighth and so on.

Thus, 3/8 would be 0.011 in binary, as three-eighths = one-quarter + one-eighth. In binary, recurring 'bicimal' numbers are quite common; so, for example, one-tenth would be represented as

0.00011001100011… which will leave a rounding error wherever it's truncated.

Given that computers only have limited memory, for very large or very small numbers it's inefficient to store all the bits in a simple place value representation, so we use a floating point form, equivalent to scientific notation, storing both a mantissa and exponent. So, just as 1.14×10^2 is another representation of 114, so we could represent 1110010 as 1.110010×2^{110} where the 'bicimal' (radix) point has been shifted six places (six being 110 in binary), storing the binary mantissa (1.110010) and exponent (110) to represent decimal 114.

It's also possible to store negative numbers in a binary representation. Given a fixed word length (the number of bits set aside for the number), the usual method is called 'two's compliment': for a negative number we reverse the bits (0 becomes 1, 1 becomes 0) and add one. For example, 75 is 1001011 in binary:

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

but with two's compliment and an eight-bit word (a 'byte'), –75 would be 10110100+1, that is, 10110101.

–128	64	32	16	8	4	2	1
1	0	1	1	0	1	0	1

Note though that the first bit no longer represents 128s; it now shows 0 for positive and 1 for negative; thus, instead of using eight bits to store numbers from 0 to 255, we instead would store –128 to 127.

It's worth bearing in mind that binary and decimal are just different ways of representing the same number: forty-two is still forty-two, whether we write it as 'forty-two', 42, XLII or in binary as 101010. Don't let pupils confuse the thing itself with the way the thing is represented.

Arithmetic

Counting in binary is easy, and a really nice pattern quickly emerges. Start with one, changing the rightmost bit by one each time, carrying into the next column when you would get to two:

1
10
11
100
101
110
111
1000
1001
1010
1011
1100
1101
1110
1111
10000
…

Like counting, binary arithmetic is surprisingly easy to master, and it's a good way to revisit the standard algorithms for the four rules of arithmetic that pupils will have learnt in primary school. The key to this is to remember that we carry when we get to 2, not 10, as we only ever have 1s and 0s in any place.

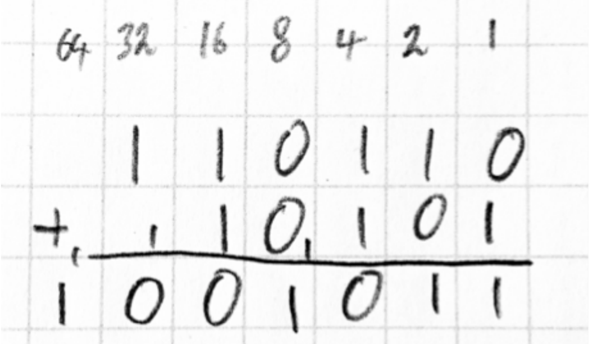


Figure 3.3

Look at the addition example in Figure 3.3, starting from the units column on the right. 0 plus 1 is 1. In the twos, 1 plus 0 is 1. In the fours, 1 plus 1 is two, so we put 0 down and carry the 1. In the eights we have 0 plus 0 plus 1 which is 1, and so on. The sum and carry operations in binary addition here can be carried out using a relatively simple combination of logic gates – see page 103.

Subtraction can be done easily by hand too, in the example below (Figure 3.4) using the decomposition method that most schools use for decimal subtraction.

Figure 3.4

Again, starting from the right: 1 minus 1 is 0. In the twos column, 1 minus 0 is 1. In the fours we have a problem, as we can't do 0 minus 1, so we decompose the eight into two fours, then 10 minus 1 is 1 in binary. In the eights we have 0 minus 0, which is 0. In the sixteens we have a problem again, as we can't do 0 minus 1. We can't decompose the thirty-twos, so we decompose one sixty-four into two thirty-twos, then decompose one of them into two sixteens, and 10 minus 1 is 1 as before. Finally in the thirty-twos, 10 minus 1 is 1.

The times tables for binary are quite easy to learn:

- 0 x 0 = 0
- 0 x 1 = 0
- 1 x 0 = 0
- 1 x 1 = 1

which is the same as the truth table for Boolean AND, if we represent True as 1 and False as 0. Again, we can apply the usual decimal long multiplication to multiplication in binary (Figure 3.5):

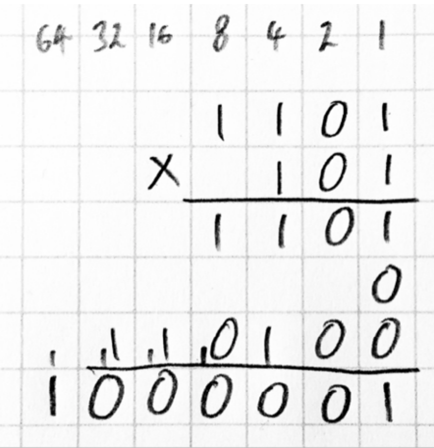


Figure 3.5

Again, starting from the right of the number we're multiplying by, in the units column, 1101 x 1 is 1101; in the twos column, 1101 x 0 is 0; in the fours column, 1101 x 1 is 1101, but we shift across into the fours column (that is, a couple of places) to write this down (1101 x 100 = 110100). Adding these answers up using binary addition, we get 1000001. Notice that multiplication is simply repeated shifts of the original number together with binary addition, both of which are easy to accomplish in digital circuits.

The process here matches the 'Egyptian' or 'Russian Peasant' method for long multiplication:[5] start by writing down the two numbers to be multiplied, the larger on the left, the smaller on the right. Double the numbers on the right, halve the numbers on the left, discarding any remainders:

13	5
26	2
52	1

Now, discard any lines with an even number on the right:

13	5
52	1

and then just add the numbers on the left:

13+52 = 65

The first step here, doubling the numbers on the left, is simply a left shift in binary. Halving and ignoring the even values is the equivalent of binary conversion, so we only add up the shifted values corresponding to a 1 in the binary representation.

Pupils can also do division in binary, again using the same algorithm as for decimal long division,[6] but here with the advantage that the divisor either divides or does not divide into the dividend at each stage. There's an argument that binary is a far better base for learning the mechanics of this algorithm, as the additional cognitive load of estimating how many times the divisor goes into the dividend at each step is removed.

5 See, for example, www.cut-the-knot.org/Curriculum/Algebra/PeasantMultiplication.shtml
6 See https://en.wikipedia.org/wiki/Division_algorithm#Integer_division_.28unsigned.29_with_remainder for a version expressed as pseudocode.

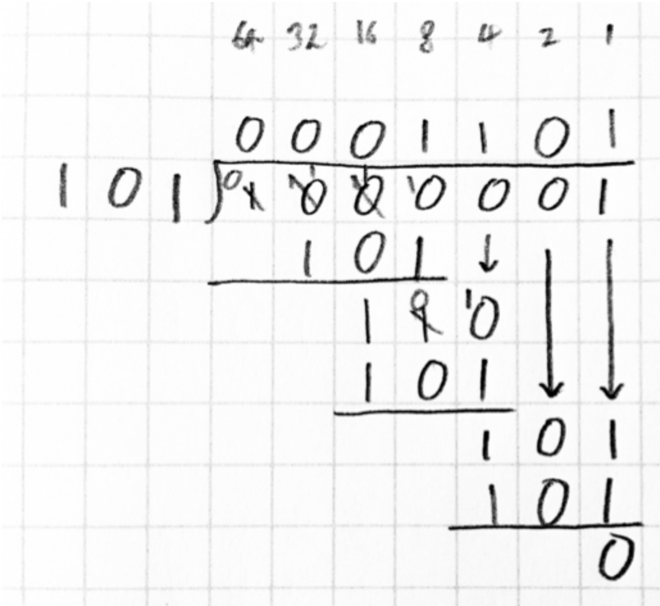


Figure 3.6

Working this time from the most significant bit of the dividend, on the left (Figure 3.6):

101 doesn't go into 1. It doesn't go into 10. It doesn't go into 100. It does go into 1000, once, so we now work out the remainder in the eights column, using binary subtraction, 1000–101=11. Bringing down the next bit, a 0, 101 does go into 110, once, with a remainder of 110–101, that is, 1. Bringing down the next bit, 0, 101 doesn't go into 10, so we write 0 in the twos column of the quotient. Bringing down the next bit, 1, 101 goes into 101 once, with a remainder of 101–101, that is, 0.

Text

As well as being able to work with numbers in a binary form, the programme of study also expects pupils to understand how other forms of information are represented in a computer numerically, requiring that the pupil

Understands that different information could be represented in exactly the same representation

In order for computers to store, process or transmit information as text, it's necessary for this to be coded as numbers (with the numbers themselves stored as binary).

From Victorian times, way before the advent of digital computers and the internet, electrical circuits were used to transmit information, originally using simple, binary state, on-off switches (the telegraph), before the use of analogue sound signals (the telephone). Rather than converting text to numbers, a different form of representation was soon agreed on, in which each letter of the alphabet (plus punctuation, digits and other symbols) had an agreed sequence of short or long pulses associated with it – this was Morse code, named after its inventor (Figure 3.7).

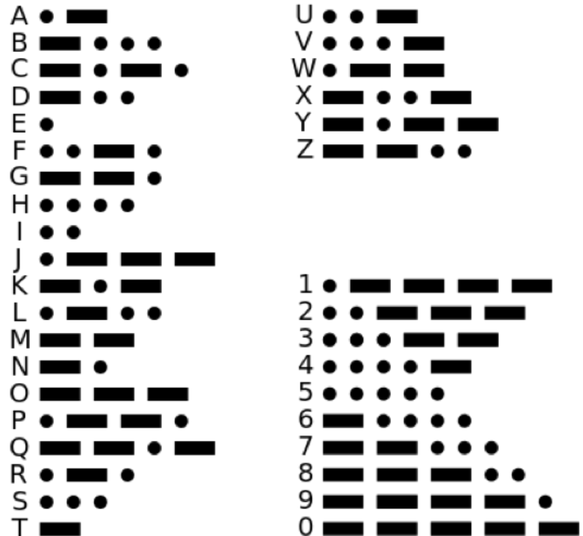


Figure 3.7

The key thing here is not the details of the representation, but the idea that there needed to be a single, agreed system for communicating via the telegraph's infrastructure: the same held true with the adoption of digital computers and, particularly, their connection via the internet. Morse's system took account of the relative frequency of letters in English, thus e and t, the most frequently occurring letters in typical English prose, have very short symbols, (. and -) respectively, but other letters, such as q and j, which occur far less frequently, have longer pulse patterns, and thus take longer to transmit.

Another system, Baudot code,[7] represented each letter as a pattern of five on or off signals, which subsequently became the standard for telegraph communications and the teleprinters used as terminals to communicate with the first computers. Five bits allowed only 32 different symbols to be represented, but shift codes were used to swap between letters

7 https://en.wikipedia.org/wiki/Baudot_code

and symbols (including numbers). A version of this, the International Telegraph Alphabet (ITA2) remains in use for some applications even today.

For a long time, the most widely adopted code was US-ASCII (United States-American Standard Code for Information Interchange; American Standards Association, 1963). In US-ASCII, numbers from 0 to 127 are used to represent upper- and lower-case letters of the Latin alphabet, the digits 0–9, commonly used punctuation, and necessary control characters (such as new lines and the backspace). By this point, eight bits (enough for the numbers 0–255) had become a standard unit of memory, the **byte,** and one byte was thus more than enough to store or transmit any single character of standard English text, with room to spare if needed.

The spare capacity in US-ASCII (it only took seven bits of an eight-bit byte) allowed other alphabets to be represented by the numbers 128–255, swapping in and out different code pages depending on the particular alphabet and language to be represented – thus Russian Cyrillic characters could be represented using the same numbers as would be used to represent Arabic characters, depending on the particular code page added on for the 128–255 range above standard US-ASCII.

Back in the days when memory was expensive and scarce, such a swappable code page system would work well enough with single, alphabet-based languages, but what hope would it have of representing the characters of a language such as Chinese?

Number	Binary	Character	Number	Binary	Character	Number	Binary	Character
32	00100000	(space)	64	01000000	@	96	01100000	`
33	00100001	!	65	01000001	A	97	01100001	a
34	00100010	“	66	01000010	B	98	01100010	b
35	00100011	#	67	01000011	C	99	01100011	c
36	00100100	$	68	01000100	D	100	01100100	d
37	00100101	%	69	01000101	E	101	01100101	e
38	00100110	&	70	01000110	F	102	01100110	f
39	00100111	‘	71	01000111	G	103	01100111	g
40	00101000	(	72	01001000	H	104	01101000	h
41	00101001	)	73	01001001	I	105	01101001	i
42	00101010	*	74	01001010	J	106	01101010	j
43	00101011	+	75	01001011	K	107	01101011	k
44	00101100	,	76	01001100	L	108	01101100	l
45	00101101	-	77	01001101	M	109	01101101	m
46	00101110	.	78	01001110	N	110	01101110	n
47	00101111	/	79	01001111	O	111	01101111	o
48	00110000	0	80	01010000	P	112	01110000	p
49	00110001	1	81	01010001	Q	113	01110001	q
50	00110010	2	82	01010010	R	114	01110010	r
51	00110011	3	83	01010011	S	115	01110011	s
52	00110100	4	84	01010100	T	116	01110100	t
53	00110101	5	85	01010101	U	117	01110101	u
54	00110110	6	86	01010110	V	118	01110110	v
55	00110111	7	87	01010111	W	119	01110111	w
56	00111000	8	88	01011000	X	120	01111000	x
57	00111001	9	89	01011001	Y	121	01111001	y
58	00111010	:	90	01011010	Z	122	01111010	z
59	00111011	;	91	01011011	[	123	01111011	{
60	00111100	<	92	01011100	\	124	01111100	|
61	00111101	=	93	01011101	]	125	01111101	}
62	00111110	>	94	01011110	^	126	01111110	~
63	00111111	?	95	01011111	_	127	01111111	⌧ (delete)

Table 3.1

Subsequent development saw the extension or perhaps, more accurately, the replacement of ASCII with Unicode, which uses (up to) 32 bits, that is, four bytes, to store each character, representing linguistic (and other) symbols with numbers between 0 and 4,294,967,295, although thus far only 120,000 or so characters from 129 scripts are encoded. The characters coded with 0–127 in Unicode (UTF-8) match exactly the characters given these numbers in US-ASCII, so for those working with standard English there are few practical differences between the two systems. It's fascinating to explore the full Unicode table[8] to see the diversity of symbols used to write human languages.

A couple of lines of code allow text to be converted to its numeric representation and vice-versa.

```
map (unicode ( ) as letter) over (list 72 101 108 108 111)
map (unicode of [ ]) over (word → list Hello)
```

Figure 3.8

```
list(map(chr,[72,101,108,108,111]))
list(map(ord,"Hello"))
```

Snap! (Figure 3.8) and Python code to convert between character codes and text. Note that Snap! works in Unicode by default, whereas Python defaults to ASCII

Editing text is, in essence, simply about making changes to the sequence of numbers which represent any particular string of characters. Familiar operations such as cut, copy and paste involve manipulating sequences of numbers: thus, cut involves removing some numbers from the sequence, making a copy in another memory location (essentially a variable); copy involves duplicating part of the sequence; paste would be inserting one sequence within another. Whilst strings of characters and lists are typically thought of as different data structures, their internal representation as sequences of numbers will have much in common: and cut, copy and paste for text directly parallel common operations on lists.

One interesting development has been the way in which the US-ASCII punctuation and other characters have been combined in ways previously absent from language to succinctly convey emotions which would otherwise be cumbersome to write: thus emoticons such as the following are commonly used in online communication:

- :-)
- :-(
- ;-)
- 8-()

More recently, this has extended into the novel linguistic form of the 'emoji', in which pictorial representations of words can take the place of more conventional, character-based forms. Emojis[9] too are represented as numbers, with many now being included in the Unicode table.

The way in which a particular character is shown on screen or when printed out is different from its internal representation. Part of the job of the operating system and application software is to take the internal numerical representation of the character and display or print it as a specific **glyph** using a particular font, converting the code for the character into patterns in pixels, lines and curves, or ink that can be read on screen or paper respectively. Similarly, text-to-speech interfaces must take the character-by-character numerical representation, process this according to the grapheme/phoneme correspondence for the language, and produce appropriate audio using one of perhaps several 'voices'.

Images

There are two main ways to represent images digitally: the most common involves imposing some form of grid on the image, and then allocating numbers to the colour of each cell (square) in the grid – we call this a 'bitmap' representation; an alternative is to describe the shapes (lines, curves, polygons and so on) from which the image is made, essentially writing a program to reproduce the image from its components.

8 https://unicode-table.com/en/ [2/1/17]
9 http://unicode.org/emoji/charts/full-emoji-list.html

A colour bitmap, then, is typically, made up of a rectangular grid of small squares, called pixels, each of which is thought of as having one of a fixed number of possible values for red, green and blue components. Digital cameras take this approach for input, using a lens to focus light onto an array of light-sensitive receptors, usually with red, green and blue filters in front. LCD (and similar) screens take the same approach to output, shining light through a semi-transparent grid through which brightness can be controlled to a particular level, again with red, green and blue filters in front.

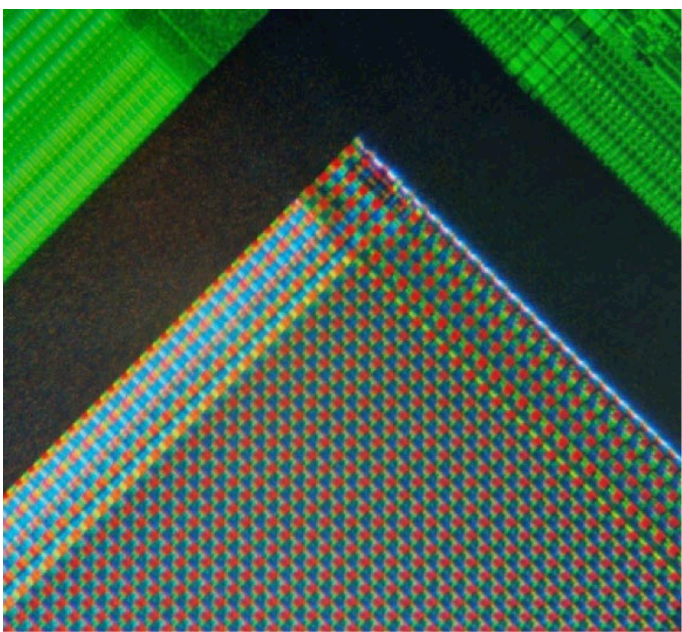


Figure 3.9 A microphotograph of a webcam image sensor (CC by-sa Natural Philo)

In storing bitmaps, as with any digitisation process, there's a trade-off here between the amount of storage (memory) needed for the image and the resolution and colour fidelity stored.

Using just one bit per pixel reduces the representation to simply black or white, but even with only a few pixels it's often possible to recognise the image (Figure 3.10–3.11):

Figure 3.10 3,550 pixel bitmap, 1 bit per pixel, so 419 bytes of memory

Figure 3.11 120,000 pixel bitmap, 1 bit per pixel, so 15KB memory

Eight bits (one byte) per pixel allows 256 shades of grey to be represented, greatly improving the quality of the representation, but taking eight times as much memory as a black-and-white image

(Figure 3.12–3.13).

Figure 3.12 3,550 pixel bitmap, 8 bits per pixel, so 3.55KB of memory

Figure 3.13 120,000 pixel bitmap, 8 bits per pixel, so 120KB of memory

With three times the memory, using one byte (eight bits, 256 levels) for each red, green and blue colour channel, we have the full 16 million-colour representation that we are used to in digital media (Figure 3.14–3.15).

Figure 3.14 3,550 pixel image, 24 bits per pixel, so 10.65KB of memory

Figure 3.15 120,000 pixel image, 24 bits per pixel, so 360KB of memory

Image manipulation software[10] allows pupils to experiment with the effect of reducing the resolution of an image and the number of bits used to store the colour or brightness information. Pupils can also explore creating 'pixel' art, choosing the colour for each pixel of the image, typically at a very low resolution. They can do this by hand on gridded paper or using a spreadsheet, perhaps using conditional-formatting tools to shade cells according to the number entered.

Mathematician and comedian Matt Parker makes an online tool available to convert images in standard file formats into suitably coloured-in Excel spreadsheets.[11] Once the pixel colour values of the image are in a spreadsheet format, it's easy enough to apply formulae to cells and groups of cells, to see how simple image manipulation can be accomplished: increasing the brightness of the image means increasing the values in each cell; reducing an image to greyscale involves replacing each colour value with the average of the red, green and blue values for a pixel; blurring an image can be accomplished by replacing each red, green or blue pixel value with the average of the corresponding values for the nine or 25 surrounding cells; and so on.

Similar effects, and indeed much more, can be accomplished in programming languages, for example using Python's pillow or scikit-image libraries, or using the tools built into standard image manipulation packages.

The other approach to working with images, **vector graphics**, in which we give the instructions for the lines and curves that make up an image, has a number of advantages: the files here tend to be more compact, there's no fundamental limit to the resolution at which images can be displayed, and there's no 'pixelation'. However, this approach is much more suitable for working with drawings created originally on the computer than for digitising images of the real world.

Audio

In the case of sound, again there are a couple of options: the first involves storing a sequence of numbers to represent the volume of sound at different points in time; the other is closer to composing music, creating a set of instructions for sound to be played.

Let's take the case of recording some sound on a computer. A microphone takes the pressure waves in the air that we hear and converts these into an analogue electrical signal. The analogue signal is then **sampled** lots of times a second and each of the sampled voltage values is then simply converted to a number. This is called pulse code modulation (PCM) and is used in the .WAV file format. 'CD-quality' audio is sampled 44,100 times a second (that is, a sampling frequency of 44.1KHz), and sixteen bits are used to store the different sound or voltage values (that is, from 0 to 65,535) for both left and right channels. Thus, one minute of CD-quality stereo audio takes just over 10MB of storage.[12]

As with image representation, there's a trade-off between the storage capacity needed and the veracity of the digital representation. It's possible to store reasonable audio with a lower quality than this, and for spoken-word recording this may suffice – mono recordings at 11KHz storing just eight bits for each sample are usually acceptable, with quality comparable to long wave radio or analogue telephone calls. It's also possible to store at a higher quality – so called 'high definition' audio uses 24 bits for each sample, allowing 8,388,608 different audio intensity values to be stored, and sample rates as high as 192KHz.[13] Even here though, a digital representation can never be a perfect match to the even-finer-grained analogue signal.

10 For example, Photoshop, The Gimp or Pixlr.
11 www.think-maths.co.uk/spreadsheet, qv www.youtube.com/watch?v=UBX2QQHIQ_I
12 Note: 2 channels * 2 bytes per sample * 44,100 samples a second * 60 seconds in a minute.
13 Note that 32bit, 384KHz audio is also available.

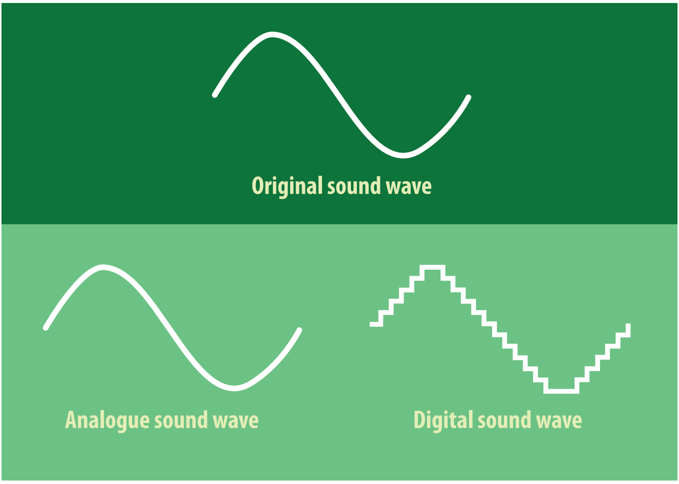


Figure 3.16

Editing digital audio essentially means manipulating the sequence of numbers that represents the audio signal – making a recording louder can be accomplished by multiplying all the numbers by a number bigger than one; making the recording quieter involves multiplying by a number less than one; silencing part of a recording means replacing the audio signal values with 0; cutting a section of a recording involves deleting the numbers corresponding to that section from the sequence of numbers in the file.

It is possible to view and edit the contents of an audio file (for example the .WAV format) using a hexadecimal editor for binary files,[14] although, given the very high number of samples per second, it's hard work to do so in any meaningful way. With some ingenuity, .WAV audio files can be imported and exported from Excel or other spreadsheets,[15] and the data manipulated directly using cell-based formulae. Audio files can also be manipulated as data in Python[16] or other programming languages. Audio editing software such as Audacity provides a graphical user interface and simple, intuitive tools for working with audio files at a higher level of abstraction, hiding the numerical representation of the audio from the user.

```
import wave
import random
import struct

noise_output = wave.open('noise.wav', 'w')
noise_output.setparams((2, 2, 44100, 0, 'NONE', 'not compressed'))

for i in range(0, 44100):
        value = random.randint(-32767, 32767)
        packed_value = struct.pack('h', value)
        noise_output.writeframes(packed_value)
        noise_output.writeframes(packed_value)


noise_output.close()
```

Python program to generate 1 second of stereo random noise in 16 bit 44.1KHz PCM format[17]

For music, it's possible to think in terms of representing the composition digitally, rather than the sound that's heard, essentially writing a sequence of instructions (a program) which when executed would play the music – the MIDI file format does this, storing the order of note values and durations that make up the music, and then using software to play these back, typically using short samples of audio from recorded instruments or digitally-generated (synthesized) tones.

It's possible to create midi format files using a wide range of applications, including sequencing and traditional stave-notation composition software, and pupils can get a feel for this more programmatic approach to music using Scratch or Sonic Pi, both of which use MIDI note values as a starting point.

Compression

Often it is useful to take large text, image or sound files and store them in a more compact form, using fewer bytes to store the same, or almost the same, information. Although computer storage capacities have increased exponentially for unit cost (Walter, 2005), ever increasing amounts of data at higher and higher resolutions are stored. Moreover, a more-slowly increasing internet bandwidth is used to transmit these files, with many applications – from text, audio and video chat or conferencing to the streaming of audio and video content – requiring low-latency, real-time transmission.

14 https://mh-nexus.de/en/hxd/ [2/1/17]
15 For example, using the .DAT text data format for SoX, http://sox.sourceforge.net/
16 Using the standard wave library: https://docs.python.org/3/library/wave.html
17 Adapted from https://soledadpenades.com/2009/10/29/fastest-way-to-generate-wav-files-in-python-using-the-wave-module/

Pioneering work by Claude Shannon in 1948 (Shannon and Weaver, 1949) established the theoretical foundation for information theory, including the idea of information entropy, which measures the degree of uncertainty in any message. This uncertainty is determined by the nature of the message: if an English message includes the letter q, the next letter is very likely to be u; if a message in English contains ee, the next character cannot be another e; t is more likely to be followed by h than any other letter, and so on. For example, given a sentence in English presented without vowels, it's possible to recover much, if not all, of the original using our knowledge of English vocabulary, permitted syllables, and - in this case - English literature to help:

> **t s trth nvrslly cknwldgd, tht sngl mn n pssssn f gd frtn, mst b n wnt f wf.**[18]

Note that whilst we have saved space here, this is at the expense of some fairly intensive processing to uncompress this text. Shannon's insight was to recognise that the limits of a communication system were determined, not by the message communicated, but by the **possible messages** that could be communicated, and their relative likelihood. As Morse had recognised earlier, some letters, such as e and t, are far more frequent than others, and hence had shorter signals; in contrast, not all US-ASCII and Unicode characters are equally likely to occur in a message, and yet each takes the same number of bits to transmit. Shannon's information entropy uses probability to determine the minimum number of bits needed to communicate a message in a particular system. His original estimate, based on the patterns from looking at groups of eight characters, was about 2.3 bits per character for English, but subsequent work, using longer-range characteristics of the language, suggested an entropy of as little as one bit per character (Shannon, 1951). Users of predictive text systems will be familiar with how quickly smartphones are able to guess the word that is being typed: these systems draw directly on Shannon's ideas.

Compression techniques draw on Shannon's work too, finding clever ways to represent the same (or in some cases, similar) information in a smaller number of bytes.

Huffman coding is closely related to Shannon's information entropy idea. Rather than using the same number of bits for each symbol, Huffman formalised the idea of assigning shorter codes to more-frequently occurring symbols (Huffman, 1952) in such a way that there would be no ambiguity in decoding.

One simple approach to compression is run-length encoding. If we wish to communicate a message about coin tosses:

HHHTTHHTTTHHTTTTTTHHTHHTTTT

run-length encoding shortens the message by simply encoding how many instances of each symbol there are in each run of it:

H3T2H2T3H2T6H2T1H2T4

Similarly, our black-and-white image of Ada Lovelace (Figure 3.17)

Figure 3.17

has very long runs of black pixels with short runs of white pixels: run-length encoding would compress this very efficiently.

Another technique involves looking for patterns in the information. In the case of English text, we

18 It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.

might simply choose to replace common words (such as 'the', 'is', 'to', 'of', 'and', etc) with short, one-byte codes to represent them. A more-sophisticated system would be to recursively look for longer and longer patterns in the text, image or audio, replacing these with codes and storing the pattern against the code. LZW (Lempel-Ziv-Welch) compression (Welch, 1984) takes this approach:

1. Initialize the dictionary to contain all strings of length one.
2. Find the longest string in the dictionary that matches the current input.
3. Emit the dictionary index for the string to output and remove the string from the input.
4. Add the string followed by the next symbol in the input to the dictionary.[19]
5. Go to Step 2.

For English text, this typically achieves a 50 percent compression saving. The algorithm most commonly used with the .ZIP compression format is a combination of Huffman coding and an earlier version of LZW compression, which might save 65 percent of the space for a text file.

In the case of text, scientific data and, particularly, program source or binary files, any compression has to be **lossless:** it is essential that we can recover an exact copy of the original information. Huffman codes, run-length encoding, LZW and .ZIP compression all achieve this, and can be applied to any form of data. Other media formats have particular compression algorithms that can be even more efficient, taking account of the particular properties of the medium. For images, .TIFF and .PNG formats both support lossless compression. For audio, .FLAC supports lossless compression.

However, in many cases, it is not absolutely essential to be able to uncompress a file to recover all the information originally present: close enough is often good enough. In these circumstances, we can use **lossy** compression. Very high compression ratios can be obtained but only at the expense of discarding some of the information contained in the original data.

For images, the .JPEG format (Austin, 2009) offers high compression ratios, saving up to 90 percent of the space required for a full bitmap representation, with little loss of image quality. .JPEG makes use of Huffman coding, but it also takes account of how we perceive images – that changes in brightness are noticed more than subtle changes in colour, and that low-frequency changes are more noticeable than high-frequency ones.

Similar ideas are used for audio compression using the .MP3 format (Sellars, 2000): we notice relatively loud noises more than relatively quiet ones, so the data from the relatively quiet noise can have fewer bits devoted to storing it without significantly impacting our perception of the sound. Transient high- or mid-frequency sounds capture our attention more than repetitive low-frequency ones, and thus deserve more bits for their storage.

We mentioned the idea of creating images and audio using vector graphics or midi notation: these formats are far more compact than high-resolution bitmap or PCM audio respectively, as in both cases we store instructions for making the image rather than the image itself. This is related to the idea of Kolmogorov complexity (Kolmogorov, 1998), in which the information of a file or message is measured not by the bits needed to store it but by the bits needed for a program to reproduce it. Take for example the sequence:

3141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117 0679

This appears to take 100 bytes to represent as text. An alternative form, using just 35 characters, would be 'the first 100 decimal digits of Pi'. A genuinely-random sequence (such as is needed for a 'one-time pad' in cryptography; see page 152) requires as many bytes to describe it as it contains. However, a pseudorandom sequence from a computer random number generator (that is, one which has similar properties to a genuinely-random sequence but is generated by an entirely deterministic system), can be described fully by the program for that generator and its initial seed state.

19 See https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch#Example for a worked example, qv www.cs4fn.org/internet/crushed.php

Video

Video is a particular challenge for storing and transmitting: if we simply store a single bitmap at full HD resolution (1920 x 1080 pixels), it needs over 6MB assuming three bytes per pixel. Video might typically show 25 frames per second, so one minute of video without any compression would require over 9GB, plus a further 10MB for uncompressed PCM audio. Obviously such figures are impractical for storing, processing or transmitting video and necessitate some clever uses of lossy compression, including those discussed earlier for images and audio.

Video compression can also make use of the generally-static nature of most of what's seen on screen. Between one frame and the next in a video, relatively little changes and, furthermore, our perception generally tunes out things that don't change much. Thus, video standards such as H.264 (Wiegand et al., 2003) need only pay attention to the changing bit of a video signal, perhaps using keyframes a couple of times a second (or, for video conferencing, much less frequently) in which all pixels are captured; the other frames need only store the changes from the previous frame or from the keyframe. For H.264 compression, these techniques reduce the file size for our one minute of 1080p video from over 9GB to around 400MB; with the later HEVC (H.265) standard (Sullivan et al., 2012), this figure drops to around 200MB. Streaming at a lower resolution would obviously reduce the file size further still.

To give pupils some feel for video compression, they could compare the file size of all the frames in a stop-motion animation to that of the H.264 compressed video exported from a video editor. Telling the same story using scripted animation, such as in Scratch or Blender, would be a useful exercise. Whilst screencast output from Scratch or rendered output from Blender would produce files of a comparable size, the Scratch program files or Blender project files would be significantly smaller, reflecting the Kolmogorov complexity.

Classroom activity ideas

- Spend time helping pupils develop fluency in converting between binary and decimal numbers, and in doing arithmetic in binary. Counting games, arithmetic drill-and-practice, worksheets and having pupils record screencast tutorials are all likely to be useful here. Using functions to convert between binary and decimal, pupils could write their own drill-and-practice programs in Snap!, Python or other programming languages.
- Introduce pupils to the idea of encoding and decoding text through Morse code activities, perhaps using torches or electrical circuits to transmit messages, or using the automatic Morse decoders for the BBC micro:bit[20] or Raspberry Pi.[21] Pupils could go on to experiment with converting between text and ASCII or Unicode representations, or develop programs to do so.
- Pupils can explore bitmap images using the tools available in image editing software.[22] They can create small images pixel by pixel in Excel or pixel art editors.[23] Matt Parker's tool[24] to create Excel spreadsheets from image files is highly recommended. Pupils should construct formulae in the spreadsheet to manipulate the image to desired effects.
- It's harder to work with audio files, but Audacity provides a simple editor for files in this format and, once pupils' programming has reached a level of some fluency, they can explore creating or editing PCM-encoded audio in Python using the wave library.[25]

Further resources

BBC Bitesize (n.d.) *Data representation.* Available from www.bbc.co.uk/education/topics/zxnfr82

CIMT (n.d.) *MEP exercises on binary conversion and arithmetic.* Available from www.cimt.org.uk/projects/mepres/book9/bk9_1.pdf [2/1/17]; qv www.cimt.org.uk/projects/mepres/book9/book9.htm#unit1 [2/1/17] and www.cimt.org.uk/projects/mepres/book9/book9int.htm [2/1/17].

20 http://microbit-micropython.readthedocs.io/en/latest/tutorials/network.html
21 www.raspberrypi.org/learning/morse-code-virtual-radio/
22 For example Photoshop, The Gimp or Pixlr.com.
23 For example http://www.pixilart.net/
24 www.think-maths.co.uk/spreadsheet
25 See also https://people.csail.mit.edu/hubert/pyaudio/ for basic Python audio input/output handling.

CS Unplugged (n.d.) Resources on counting in binary. Available from http://csunplugged.org/binary-numbers/; image representation: http://csunplugged.org/image-representation/; text compression: http://csunplugged.org/text-compression/; sound representation: http://csunplugged.org/modems-unplugged-2/; and information theory more generally: http://csunplugged.org/information-theory/

CS4FN (n.d.) Resources on run-length encoding. Available from www.cs4fn.org/compression/burrowswheeler.php; mp3 audio compression: www.cs4fn.org/mathemagic/sonic.html; LZW compression applied to Vicky Pollard: www.cs4fn.org/internet/crushed.php; and bitmaps: www.cs4fn.org/pixels/pixels.html

CS Field Guide (2016) *Data representation.* Available from http://csfieldguide.org.nz/en/chapters/data-representation.html; Coding: http://csfieldguide.org.nz/en/chapters/coding-introduction.html; and Coding-Compression: http://csfieldguide.org.nz/en/chapters/coding-compression.html

Digital Schoolhouse (n.d.) *Crazy graphics.* Available from www.digitalschoolhouse.org.uk/workshops/crazy-graphics

Gleick, J. (2012) *The information: A history, a theory, a flood.* New York, NY: Harper Collins.

Guzdial, M. and Ericson, B. (2015) *Introduction to computing and programming in Python: A multimedia approach.* Harlow: Pearson Education. Also online resources for Mark Guzdial's highly regarded media computation course (http://coweb.cc.gatech.edu/mediaComp-teach) and the Jython development environment for media computation (https://github.com/gatech-csl/jes)

Kolas, O. (2005) *Image processing with gluas.* Available from http://pippin.gimp.org/image_processing/chap_dir.html

Petzold, C. (2000) *Code: The hidden language of computer hardware and software.* Redmond, WA: Microsoft Press.

Shannon, C. and Weaver, W. (1949) *The mathematical theory of communication* (PDF). Urbana, IL: University of Illinois Press.

Logic Circuits

We discussed the principles of Boolean logic on pages 13 - 14. Inside the central processing unit (CPU) that controls the computer, all operations are implemented by using logic gates to switch the bits that make up digital data between different sets of logic circuits.

We can introduce pupils to the AND, OR and NOT gates through simple electrical circuits (Figures 3.18–3.20):

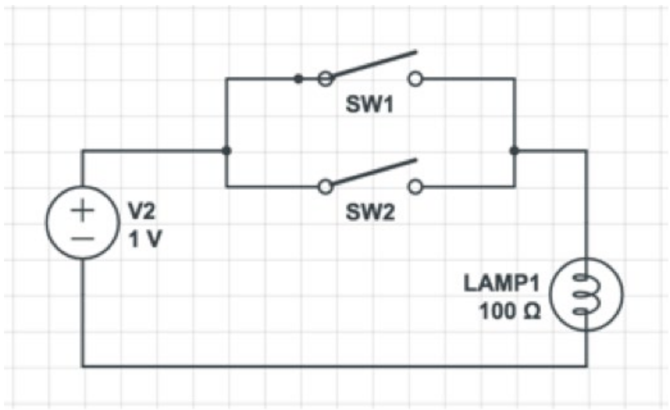


Figure 3.18 Circuit to illustrate an OR gate – LAMP1 lights if SW1 OR SW2 is closed

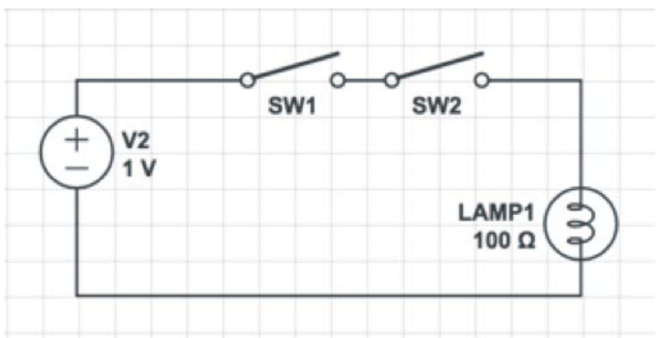


Figure 3.19 Circuit to illustrate an AND gate – LAMP1 lights if SW1 AND SW2 are closed

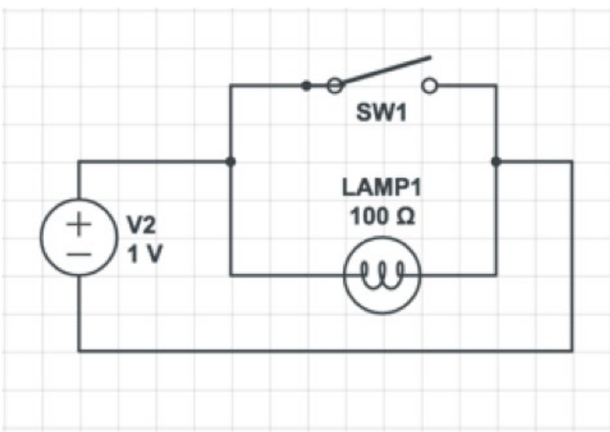


Figure 3.20 Circuit to illustrate a NOT gate – LAMP1 lights if SW1 is open

With somewhat more authenticity, these gates can be built from individual transistors, perhaps on a breadboard (Figures 3.21–3.23):

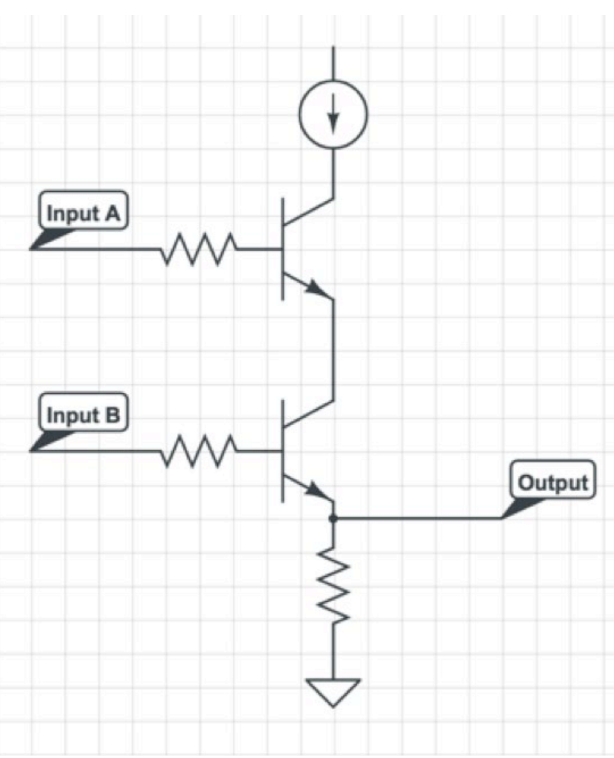


Figure 3.21 Electronic circuit for AND

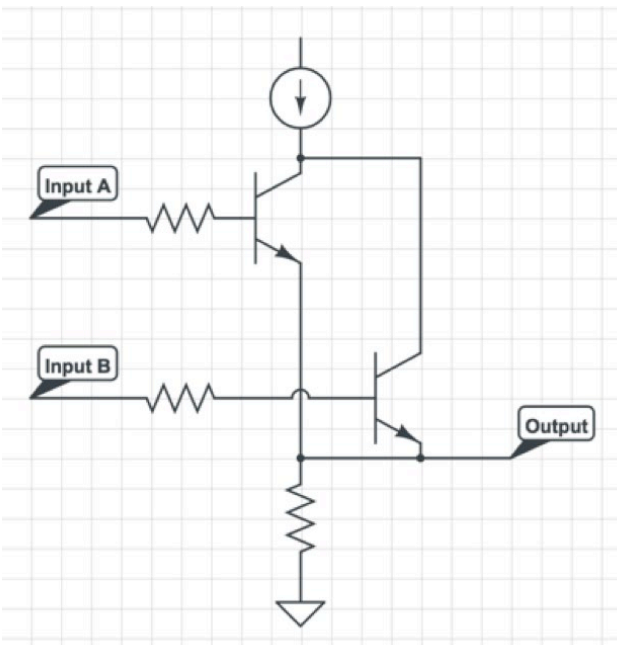


Figure 3.22 Electronic circuit for an OR gate

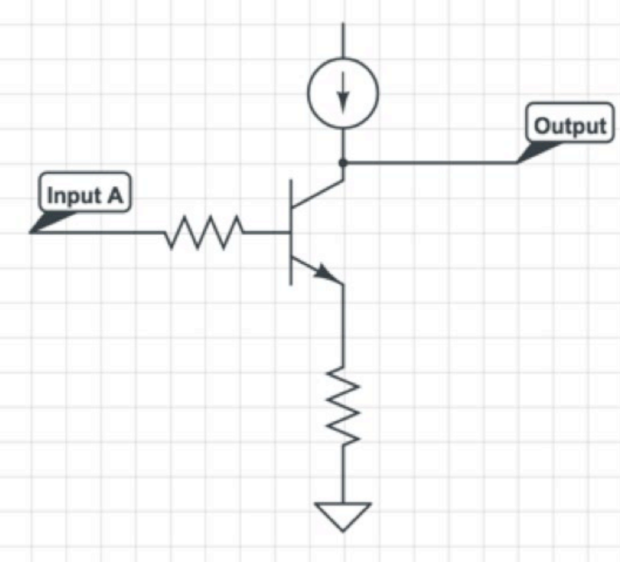


Figure 3.23 Simple electronic circuit for a NOT gate

Before the invention of the integrated circuit, digital computers would be made using logic gates composed of surface-mounted, transistor-based circuits such as these.

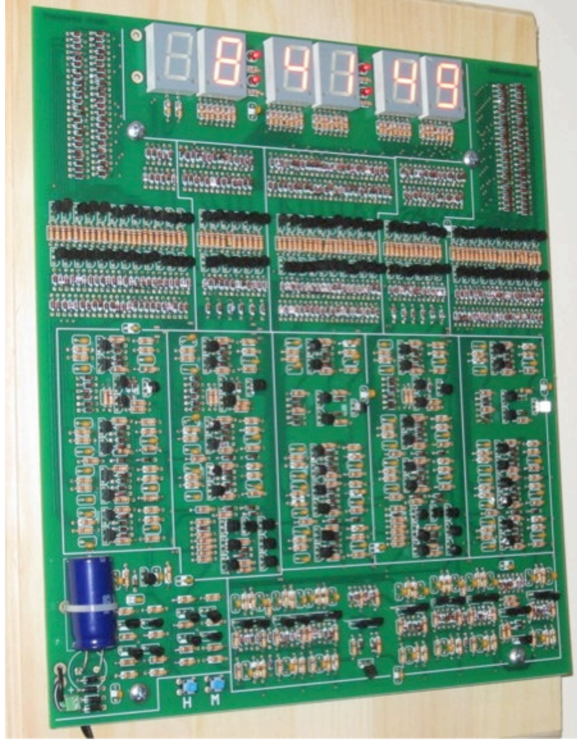


Figure 3.24 Digital clock using only transistors and other surface-mounted components. CC by-sa Wtshymanski. See also http://monster6502.com/ for a 6502 CPU implementation using surface-mounted components

The beauty of abstraction though is that we don't need to worry about the internal operation of logic gates; we can treat them as 'black boxes' which produce certain output for certain input, according to their truth tables.

Thus we can create simple logic circuits using individual logic gates as the components, rather than thinking about switches or transistors.

For example, we can build (or simulate) the circuits (Figures 3.25–3.26):

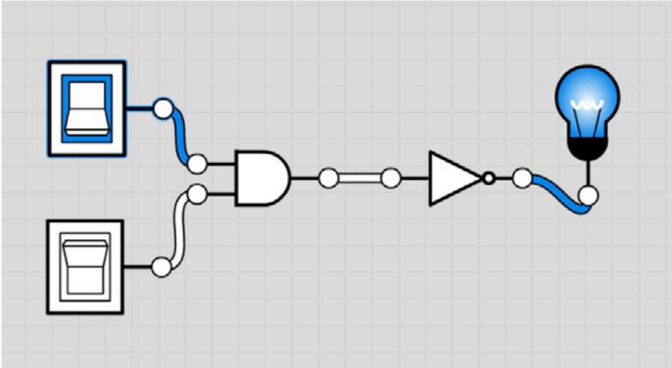


Figure 3.25 NOT (A AND B)

and:

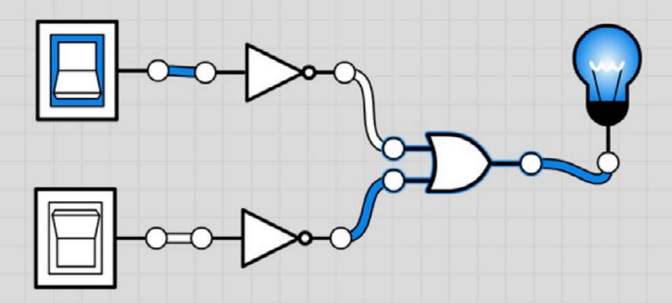


Figure 3.26 (NOT A) OR (NOT B)

We can use this symbolic representation of the gate, rather than needing to show its internal structure. Note that the two circuits above have the same truth tables:

A	B	Output
False	False	True
False	True	True
True	False	True
True	True	False

Thus the two are functionally equivalent. A gate with this truth table is called NAND (NOT AND).

More complex circuits can be constructed. Thus, for the truth table:

A	B	Carry	Sum
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	False

we have Carry = A AND B and Sum = (A OR B) AND NOT (A AND B) (Figure 3.27):

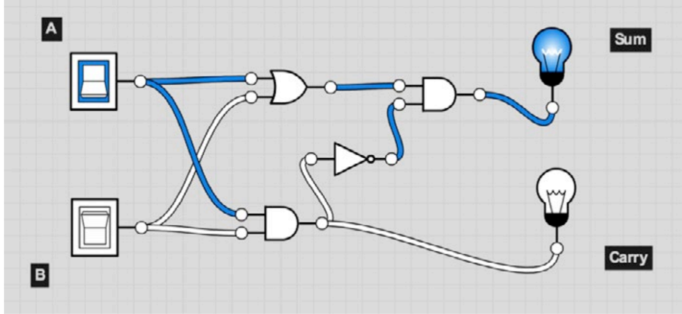


Figure 3.27

If we replace False with 0 and True with 1, the table becomes:

A + B	Carry	Sum
0 + 0	0	0
0 + 1	0	1
1 + 0	0	1
1 + 1	1	0

This allows binary addition, at least for just one bit, to be implemented using nothing more than logic gates. This circuit is known as a half adder.

Two half adders can be combined using an OR gate to allow three inputs, one from a previous carry, plus two bits of data, producing a sum and carry for bitwise addition (Figure 3.28):

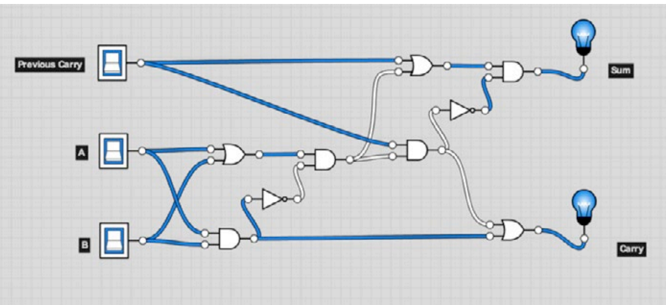


Figure 3.28 Full adder

A set of eight of these, connecting the Carry output of each to the Previous Carry input of the next would allow two bytes to be added together.

More complex logic circuits still can be designed and built. Given any truth table, it's possible to construct a logic circuit that will produce the desired output using just a combination of NOT, OR and AND gates.[26] All logic circuits, including NOT, OR and AND gates, can be built using NAND gates.[27]

Classroom activity ideas

- Provide pupils with increasingly sophisticated logic circuit diagrams, asking them to work out the truth table by tracing the output at each gate for all the possible combinations of TRUE and FALSE inputs to the circuit. A more challenging problem is to create the logic circuit for a given truth table.
- Circuit simulators such as logic.ly, circuitlab.com and logic lab[28] allow pupils to experiment with electrical, electronic and logic circuits on screen. It's well worth giving pupils some experience of creating simple logic circuits using switches, transistors or integrated circuits if the resources are available.
- You can get pupils themselves to act as a logic circuit, some taking on the role of gates, others the part of bits. See this BBC clip of implementing a two-bit adder: www.bbc.co.uk/programmes/p01m5xfs
- Give pupils particular scenarios to create logic circuits: home security systems and traffic management are popular contexts.
- Pupils can build simple logical circuits out of redstone in Minecraft.

Further resources

BBC Bitesize (n.d.) *Boolean logic.* Available from www.bbc.co.uk/education/guides/zc4bb9q/revision

BBC Joy of Logic (2014) Available from www.youtube.com/watch?v=ZO__UZ6iV0A

Gregg, J. (1998) *Ones and zeros: Understanding boolean algebra, digital circuits, and the logic of sets.* New York, NY: Wiley-IEEE Press.

Intel (n.d.) Lesson resources on circuits and switches. Available from www.intel.com/content/www/us/en/education/k12/the-journey-inside/explore-the-curriculum/circuits-and-switches.html

Minecraft (n.d.) *Logic circuit.* Available from http://minecraft.gamepedia.com/Logic_circuit; and introductory tutorial. Available from http://minecraft.gamepedia.com/Tutorials/Basic_logic_gates

PyroEdu (n.d.) Course on digital electronics. Available from www.pyroelectro.com/edu/digital/

26 Conjunctive normal form: https://en.wikipedia.org/wiki/Conjunctive_normal_form
27 https://en.wikipedia.org/wiki/NAND_logic
28 www.neuroproductions.be/logic-lab/

Hardware Components

One quite intuitive and generally helpful way of thinking about computers is as machines which accept input, process this according to some stored set of instructions and produce output (Figure 3.29).

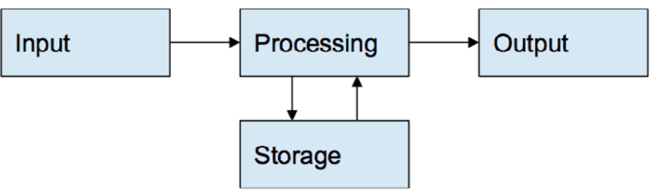


Figure 3.29

In thinking about hardware components, it's worth considering each function here separately.

Input

The form of input will vary, depending on the type of computer and the uses to which it is put. On a laptop, you might typically find a keyboard, a trackpad, a microphone and a webcam, as well as ports which other input devices, such as a USB mouse, could be plugged into. On a smartphone, you would probably find a touch-sensitive screen, some other buttons, a microphone or two, a couple of cameras, a global positioning system (GPS) receiver, an accelerometer, perhaps a barometer and again one or more ports for additional input devices.

Both Raspberry Pi and BBC micro:bit have support for a broader range of sensor input, including simple switches and the digitisation of analogue signals connected to their general-purpose input/output (GPIO) pins or connector. Sensors can be connected to these pins which can, for example, be used to capture temperature levels over time, or proximity in some robot-control applications. With other computers, an interface can be used to provide similar functionality.

For trackpads, touch screens, microphones and cameras it's necessary for the computer to convert the continuous, analogue real-world data into a digital format before it can be processed, stored or transmitted by the computer. As discussed on pages 92 - 96 in the context of images and sound, digitisation inevitably involves throwing away some of the fine detail of the real-world information.

Some pre-processing can be applied to the raw input signals, for example voice recognition provides an alternative form of input using the microphone, or positions of objects in three dimensions can be determined using two cameras or a linked laser and sensor for establishing depth, as in Microsoft's Kinect.[29] A basic form of brain–computer interface is possible using current technology, typically through sensing electrical activity in the brain.

Processing

The fetch–decode–execute cycle in which processors execute machine code instructions is described on pages 48 - 49. Programs written in high-level languages are converted into machine code using interpreters or compilers – most of the programs that run on the computer are already compiled as machine code binaries, and so this process is typically hidden from the user. The machine code instructions and the data on which the programs operate are all stored together in memory (see later), and the computer provides a fast way of moving both program instructions and data out between the processor and memory: the internal (or 'front side') bus.[30]

Processors work very, very quickly. Modern operating systems are efficient at managing the load on the processor so that, despite giving the appearance of always being ready for whatever instruction it is next required to execute, it can also run instructions from many of the other programs on the computer almost simultaneously. This is called multi-tasking.

The processors inside modern computers are typically multicore chips, containing perhaps four (or more) CPU cores and cache memory (see later), each operating independently and being capable of running instructions from quite separate

29 http://techtalks.tv/talks/54443/
30 Neil Brown explains that modern processors use optimisations such as caches, pipelining, out-of-order execution, speculative execution and microcode, in addition to the basic fetch–decode–execute cycle: https://academiccomputing.wordpress.com/2012/04/29/the-computer-is-a-lie/

programs literally at the same time. Multicore processing isn't confined to the desktop or laptop form factor either: some current smartphone models have eight-core processors.

As well as the main, typically multicore, processor, modern computers often have other processors – most notably graphical processor units (GPUs), now often designed specifically for the parallel processing needed for physics simulation and fast 2D and particularly 3D rendering – most commonly deployed for video games (Sony's PlayStation 4 has some 1,152 parallel cores for its GPU). For the iPhone 5s, Apple introduced a 'motion' coprocessor (the M7) alongside the smartphone's main CPU, dedicated to processing sensor data and running on low power even with the phone asleep.

Current developments include massively parallel processing, in which complex computing problems are shared between many processors with results being subsequently combined;[31] and processing 'in the cloud', in which complex tasks are handled not by the user's own computer but by running programs on computers to which it communicates via the internet.[32] Speech-to-text processing of services, such as Apple's Siri, is accomplished in this way.

Storage

Both the programs a computer needs to operate, and the data that it processes, are stored in the computer's memory. In modern computer architecture, the same memory is used to store both the programs and the data, side by side.

There are different types of computer memory, and usually there's a trade-off between speed and cost or capacity: the fastest, most expensive memory is used for the data that's needed most immediately and most frequently; the cheapest and slowest storage is used for data that may or may not be needed again at some distant point in the future. This structuring of memory, from small and fast to vast and slow, has proven a particularly powerful way of organising data for processing and programs for execution. Moving data between one level of memory and another takes time.

The fastest memory available will be the registers and 'level 1 cache' memory built into each CPU core itself but, even with modern processors, the amount of data that can be stored here is very limited. Elsewhere on the CPU chip, and connected to the rest of the computer via the bus, is the 'level 2 cache' – larger than the level 1 cache, and somewhat slower, but still much faster to access than the main memory.

On the internal bus, and connected directly to the main circuit board of the computer (the 'motherboard') is fast, high-capacity memory called 'RAM', although this is typically 'volatile', meaning it loses all the data stored in it when the computer is switched off.

Rather slower, but again of much larger capacity, will be the computer's main drive: until recently, this would typically contain magnetic disks able to retain data with the power turned off, but these are increasingly being replaced by faster, non-volatile 'flash' memory or solid-state drives[33] (SSDs), similar to what you might also find in USB sticks or the memory cards used for digital photography.

Optical storage, such as CD-ROMs, DVDs and Blu-ray disks, is slower still, but costs for these media are low, making them suitable for long-term storage of data rarely needed for processing. These media are not, by modern standards, of particularly high capacity, and thus many find the need to connect high-capacity external hard drives or SSD drives to the computer, perhaps via USB or a higher-bandwidth external bus interface.

Storage capacity has grown at an even quicker pace than the increase in processing speeds (Walter, 2005), and thus, now, data centres connected via the internet can provide very high capacity, but relatively slow storage, for data, at a very low cost. Whilst once upon a time, long-term archival storage might have made use of magnetic tape, there are interesting developments using low-cost, high-capacity hard drives for this sort of 'cold' storage.

31 See, for example, www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/
32 For example Google Compute, Microsoft Azure and Amazon EC2.
33 www.extremetech.com/extreme/210492-extremetech-explains-how-do-ssds-work

Internet pioneer Vint Cerf has, perhaps surprisingly, argued[34] that the best approach to **very** long-term (for example, millennium-long) archival storage might more reliably use paper than digital media, given that no special systems need to be maintained to ensure that paper remains readable.

Power and cooling

Other internal hardware components are needed too.

Processing requires power, and so one of the most noticeable components inside a desktop computer will be the power supply unit, providing regulated current at the various voltage levels needed for the computer's components. In the case of business-critical machines some, form of alternative power supply, such as an uninterruptible power supply (UPS – essentially a large battery and associated monitoring sensors and software), or perhaps even an emergency generator, may be necessary. Battery life remains a problem for portable technology including laptops, tablets and smartphones, although battery technologies have advanced significantly in recent years, alongside advances in processor and storage efficiency, and better power management software at operating-system level. A relatively recent development has been the use of additional, external batteries to provide a top-up charge for the smartphones on which many have come to rely. Convenient access to a cheap source of power is one of the principal considerations when siting a data centre.

The processing that computers do generates lots of heat (and thus the second law of thermodynamics holds, as entropy increases overall). Processors stop working if they get too hot and thus attention must be given to keeping processors cool. In a traditional desktop computer, CPUs are mounted with passive heatsinks and active fan cooling, and a smaller version of these components will also feature on laptop computers. Tablet computers and smartphones typically do not include active cooling with a fan although much care is taken in their design to ensure effective passive cooling. Bareboard computers such as the Raspberry Pi and BBC micro:bit achieve sufficient cooling in normal use, as air can freely circulate over the processor.

Cooling becomes a particularly important consideration in building data centres, where a large number of processors and other components are packed together in a small space.

Output

Computers are able to produce many different forms of output. On a laptop or desktop computer, these are likely to be the screen and speakers, together with connections for external peripherals such as printers or headphones.

On a smartphone, tablet or games console controller, outputs might also include a small motor to produce vibrations. Smartphones typically include bright light-emitting diodes (LEDs) used as a flash for photography, or to provide extra light when recording video.

Other output devices can be connected too, for example the computer can be used to control motors, such as in a robot. In addition to traditional 2D display technologies, computers can also power virtual- or augmented-reality headsets: the former replaces the wearer's view of the world with a 3D computer generated image (shown as different images to each of the wearer's eyes), the latter adds an additional layer of information to the wearer's view of the world.

Similar to the way in which a computer can print information on paper through sending instructions to a printer, computers can send instructions to 3D printers, producing 3D objects through detailed layering of a plastic resin (or other material) according to the instructions received.

Connectivity

The traditional model of input – processing – output has evolved to include network connections. The data a computer processes need not be provided directly through input devices attached to the machine; it could easily come via one or more network connections. Similarly, the information the computer outputs could be transmitted via a network connection.

34 www.theguardian.com/technology/2015/feb/13/google-boss-warns-forgotten-century-email-photos-vint-cerf

For example, a typical web server is unlikely to have a keyboard or screen connected to it directly – it accepts requests for web pages or commands via its internet connection, and responds with the HTML for the page or other output via its internet connection.

A smartphone includes a number of different network connections, including: near-field communication (NFC) for passing small packets of data, such as cashless payments over very short distances; Bluetooth for external keyboards and hands-free audio; WiFi for high-speed internet access and longer-range connection to the phone network for voice and data, possibly at high speed using 3G or 4G systems. WiFi and Bluetooth connectivity is typically provided as standard on laptop and tablet computers, and is also built into the Raspberry Pi 3. The BBC micro:bit includes Bluetooth connectivity. Traditional cabled network connections provide greater communication bandwidth, and are less likely to suffer from contention issues than the WiFi equivalents.

Many devices now contain microprocessors and should be thought of as small, dedicated-use computers with their own input/output systems: for example, a digital thermostat includes a heat sensor, some form of control interface for user input, processing capabilities and a stored program, a display and a control interface for the heating system itself. Increasing numbers of these devices now provide network connectivity too, most often via WiFi, but perhaps via Bluetooth or the mobile phone system, becoming part of the 'Internet Of Things'. Such connectivity provides greater convenience, such as the ability to turn on central heating via smartphone on the journey home, or to upload photos directly to the internet from a digital camera, but also allows integration with other internet-based systems and cloud-based processing. For example, a smart thermostat could 'learn' typical use patterns and adjust these predictively to take account of weather forecasts, or a smart, internet-connected refrigerator could autonomously submit orders to an online supermarket as staple supplies run low. Many are concerned over the privacy and security issues associated with such systems.

Classroom activity ideas

- Show pupils how input and output devices and the network are connected to a computer. Disassemble a computer as pupils watch, explaining the purpose of each of the components in turn. Explain that the components pupils see are made up of smaller components, to illustrate the multi-layered nature of abstraction in computing. Some teachers make displays of computer components.
- Compare the components of different types of computer system, showing how the internal components of desktop, laptop, smartphone and Raspberry Pi computers must all accomplish the same function, but that these different form factors mean that their physical forms can be quite different.
- Demonstrate CPU usage using operating system tools to monitor activity (Performance Monitor for Windows, Activity Monitor on OS X and the top command on Linux).
- Pupils in an extracurricular computer club could perhaps build their own computer from component parts, or fix a broken computer.

Further resources

Barefoot Computing (2014) *Computer systems.* Available from http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/computer-systems/ (free registration required).

BBC Bitesize (n.d.) *Digital devices.* Available from www.bbc.co.uk/education/guides/zxb72hv/revision; and the CPU: www.bbc.co.uk/education/guides/zws8d2p/revision

Bishop, C. (2008). Royal Institution Christmas Lecture on *'chips with everything'.* Available from www.richannel.org/christmas-lectures-2008-chris-bishop--chips-with-everything

Google Data Centres (n.d.) Available from www.google.co.uk/about/datacenters/

IFixit (n.d.) Repair and 'tear down' guides to common devices, including some great photos of internal components. Available from https://www.ifixit.com

Intel (n.d.) Teaching resources on microprocessors. Available from www.intel.com/content/www/us/en/education/k12/the-journey-inside/explore-the-curriculum/microprocessors.html

Lockwood, B. and Cornell, R. (2013) *School ICT Infrastructure Requirements for Teaching Computing:* A Computing at School (CAS) Whitepaper. Available from http://community.computingatschool.org.uk/resources/446

SpiNNaker (n.d.) *Novel, massively parallel computer architecture modelled on how the brain works.* Available from http://apt.cs.manchester.ac.uk/projects/SpiNNaker/. See also Steve Furber on this for CAS TV: www.youtube.com/watch?v=wnSjR04qang

Software Components

None of this computer hardware would do anything if it had no software: the programs that make it work, that make it useful. The many **layers** of software that make up a computer system are each abstractions of the software systems, and ultimately the hardware, beneath them. At each layer of the system, assumptions are made about how the layer lower down behaves, without needing to know how it works, which means that:

- There's no need to deal with the complexity of the system at that layer since this has already been addressed in a reliable way.
- There's no duplication of the functionality provided by the lower layer, as this is done once for all the different programs running at the upper layers.
- The internal operations of the lower layer are generally something with which we don't need to concern ourselves; indeed, in many systems the internal operation of lower layers is deliberately hidden from those working at upper layers, sometimes for reasons of security, but also for proprietary, commercial reasons.

Working from the bottom of the software stack up, and thinking in terms of a general-purpose computer such as a laptop, desktop, smartphone, tablet or Raspberry Pi, we typically begin with a very small set of **firmware** instructions, compiled into machine code, providing just enough functionality to load and run (to 'bootstrap') an operating system. Many of the internal components of the computer, such as disk drives, will have their own firmware too, and modern CPUs often include a layer of 'microcode', which is essentially an interpreter for machine code which sits between the hardware itself and the abstraction of the internal architecture that's presented to the rest of the system.

Operating systems (OS) such as Windows, OS X, Linux, iOS and Android sit between the user's experience of the computer and the computer hardware itself. Operating systems are themselves multi-layered. One level deals with fundamental operations, including: managing multi-tasking for the different system and user programs to be run on the CPU; main memory and longer-term storage; input; output; network connectivity; power and cooling; and presenting abstractions (such as a file system, virtual memory and inter-process communication) of all these complex sub-systems in a simple, consistent and reliable way to other programs on the system.

The various hardware components typically have **device driver** software, running at operating system level, (although not formally part of the operating system), which allows components from different manufacturers and with different specifications to be used by the operating system without the OS having to deal with the specifics of implementation.

Outside of the operating system itself there's often a number of **utility programs** that are useful, if perhaps not absolutely essential, to the operation of the computer, such as programs to find files, to protect against viruses and other malware and to manage the installation of other programs.

The operating system also provides one or more user interfaces – a way for the user to interact with the computer's core functions and with other software that might be running on it. The most basic of these is a **command line interface** (CLI), in which commands can be typed on screen and responses displayed on screen. Some computers (for example Linux servers) start up in this mode, but more often you can access this interface through launching a shell or terminal program. Whilst seeming to lack the functionality, familiarity and ease of use of more familiar graphical user interfaces (GUIs; see below), this stripped-down way of interacting with computers offers power and flexibility, and can sometimes be the only, or more often the fastest, way to get things done.

The other layer of a modern operating system is the **graphical user interface (GUI)**, which provides a consistent look and feel for the user's interaction with the computer in a way that's more intuitive and easier to learn through exploration than the CLI. The GUI provides a way to interact with the computer through keyboard, mouse and now voice input, task switching, an interface to manage files and folders, control interfaces for sound, networking, dates and times, and so on, as well as managing windows for programs, displaying text and graphics, looking after the printer and providing accessibility support such as voice synthesis. It's typically the GUI that people think of when talking about Windows or Mac (OS X) operating systems. All of this of course needs processing capacity in order to operate, and so computers running as servers or in data centres would typically not run, or even have, a GUI.

Whilst the CLI or GUI allow us to interact with the computer and run utility programs if needed, they don't, in themselves, provide software for getting useful work done. For this, we use **application software,** and most of the programs which we think of as running on a computer fall into this broad category. Within this category are, for example:

- Office productivity programs: such as a word-processor, spreadsheet and presentation software, desktop publishing software, calendars, task management, contact management.
- Media production tools: such as image editors, drawing programs, audio editors, music composition software, video-editing software, 3D animation packages.
- Media management tools: for example music players, video players, image galleries, photo browsers.
- Communication software: a web browser, email client, video conferencing tools, instant messaging.
- Technical software: computer algebra systems, bibliography management tools.
- Games and educational software.

Software in all of these categories can run on traditional Windows, OS X or Linux desktop computers or laptops, but these categories of software are also available as 'apps' (short for application programs) on iOS or Android smartphones and tablets, and can be run on remote web servers and accessed using just a web browser via the internet (as is the case with Google's Chrome operating system).

A few special categories of application software deserve particular mention:

- **Server software**: via the internet (or perhaps just a local network), one computer can run a program to provide services to many other computers, for example managing user accounts and password authentication on a school network; saving files centrally; storing, forwarding and providing access to email; serving static or dynamically-generated web pages when requested; managing a database of records so that connected computers can each update or interrogate a single consistent version of the information contained in it; and so on.
- **Programming language software** and associated tools: converting a program written in a human-readable, high-level language such as Scratch or Python requires an interpreter or compiler although, once compiled, the program could be treated as any other application or system program. Programmers also use a range of tools to support the process of writing programs, including text editors or integrated development environments (such as the Scratch web and offline editors, or Python's IDLE).
- **Virtualisation**: theoretically, one computer system can simulate any other; in practical terms, this allows computers to run virtual emulations of other computer systems. Programs such as Virtual Box or VMWare provide a simulation of computer hardware, running as software, onto which other operating systems can be installed. This is a great way to explore other operating systems safely, cheaply and securely, and also offers a useful way to deploy preconfigured combinations of operating systems and application software for particular purposes.

As with other application software, these categories can be used on Windows, Mac (OS X) and Linux operating systems, or accessed remotely via the web. With the exception of providing some simple web-based services to the local network, support for these categories is at present largely absent from iOS or Android platforms.

Classroom activity ideas

- Use the operating system tools to monitor activity (Performance Monitor for Windows, Activity Monitor on OS X and the top command on Linux) to show pupils all the programs which run on the CPU, to demonstrate how the operating system manages multi-tasking.
- Provide pupils with the opportunity to try out several operating systems, perhaps setting them the same task (sending an email, making a presentation, writing a short program) to accomplish using the Windows GUI, Linux at the command line, a smartphone and just a web browser.
- Give pupils the opportunity to install an operating system from scratch, perhaps using virtual hardware. If using open-source software such as Linux, pupils could then assemble and test a suite of application programs selected with a particular user in mind.
- Work with your network manager to ensure pupils get some experience of working with a command line interface, perhaps using the command prompt on Windows, or a Linux shell via terminal access, or virtualisation.

Further resources

BBC Bitesize (n.d.) *Software.* Available from www.bbc.co.uk/education/guides/zcxgr82/revision and www.bbc.co.uk/education/guides/z6r86sg/revision, also operating systems: www.bbc.co.uk/education/guides/ztcdtfr/revision

Bishop, C. (2008). *The ghost in the machine*. Royal Institution Christmas Lecture from 2008. Available from www.richannel.org/christmas-lectures-2008-chris-bishop--the-ghost-in-the-machine

Moody, G. (2002) *Rebel code: The inside story of Linux and the open source revolution.* New York, NY: Basic Books Inc.

Raymond, E.S. (2001) *The cathedral & the bazaar: Musings on linux and open source by an accidental revolutionary.* Sebastopol, CA: O'Reilly Media, Inc.

Smedley, R. (2016) *Conquer the command line.* Raspberry Pi. www.raspberrypi.org/magpi-issues/Essentials_Bash_v1.pdf

Stephenson, N. (1999) *In the beginning... was the command line.* New York, NY: Avon Books.

Physical Computing

In order for a computer to be able to do anything with the real world, it needs some form of input to get data in, and some form of output to put information back out.

Traditional 'control' activities certainly have their place in the new Computing Science and Digital Literacy programmes of study.

Teaching control technology is perhaps implied by the requirement that pupils be taught to:

> **Interpret a problem statement, and identify processes and information to create a physical computing and/or software solution**

and that he or she:

> **Writes code which receives and responds to real world inputs (in a visual language).**

It is also a requirement that pupils can

> **Identify the transfer of information through complex systems involving both computers and physical artefacts**

Perhaps the easiest way into the realm of physical computing is through using computers to monitor activity in the real world. A very simple introduction might involve recording or plotting the level of noise in the classroom using Scratch's microphone input (Figure 3.30):

```
when [flag] clicked
forever
  clear
  set x to -240
  pen down
  repeat 480
    set y to ((loudness * 3.6) - 180)
    change x by 1
  pen up
```

Figure 3.30 Scratch sound level monitor

The Makey Makey interface board[35] plugs into a computer's USB port, allowing other conductive objects to function as a replacement for some of the keys on the computer's keyboard. This can be combined with pupils' own Scratch programs to produce, for example, a maze game controlled by jumping in buckets of water, or a piano with keys made from bananas. The PicoBoard[36] provides an alternative input interface for Scratch.

Integrating monitoring software with web-based services allows pupils to explore some aspects of the 'Internet Of Things', for example, setting up a bird box camera which uploads a photograph to the web when a bird enters or leaves the box. Data could be tweeted from a school weather station in response to particular queries.

There are some great cross-curricular opportunities here, for example, the use of data loggers in science experiments, activity monitors in PE or weather station data[37] for science and geography. Working with real-world data such as this provides a very motivating context for visualisation and exploratory analysis, and perhaps some scope for introducing ideas of machine learning.

Beyond monitoring activities, many teachers report success with having pupils write software which controls real-world components. The 'Hello, world' of this sort of programming is typically flashing an LED on and off (see Figure 3.31).

```
when [flag] clicked
broadcast gpioserveron
broadcast config17out
forever
  broadcast gpio17on
  wait 1 secs
  broadcast gpio17off
  wait 1 secs
```

Figure 3.31 Scratch code to flash an LED connected to GPIO pin 17 on a Raspberry Pi. CC by-sa Raspberry Pi Foundation

```
from gpiozero import LED
from time import sleep

led = LED(17)

while True:
    led.on()
    sleep(1)
    led.off()
    sleep(1)
```

The equivalent Python code using Raspberry Pi's GPIO Zero library. CC by-sa Raspberry Pi Foundation

From here, it's a relatively simple step to producing a set of working traffic light signals using the correct sequence.

It's when monitoring (input) and control (output) aspects of physical computing are combined using a single computer program that the most exciting and creative elements of physical computing open up. For example, pupils could 'hack' a soft toy so that its eyes light up and a noise plays when the toy in tapped,[38] or write a program to play a game of noughts and crosses on a small LED screen such as

35 www.makeymakey.com/
36 www.picocricket.com/picoboard.html
37 See, for example, www.raspberrypi.org/blog/school-weather-station-project/
38 https://vimeo.com/4313755

on Raspberry Pi's Sense HAT,[39] or create a digital musical instrument.[40]

Pupils might take these ideas of monitoring and control and apply them to projects involving robotics. For example, pupils can build a model or robot out of Lego NXT or EV3 Mindstorms kit, incorporating sensors and motors, and then write code in Enchanting Scratch or EV3's own programming language to control how their model moves in response to the input signals received from the sensors.

One way of thinking about a robot is as a computer which can move, perhaps as a single, integrated system following a sequence of instructions, such as the Pro-Bot, Bigtrak or a floor turtle, or a flying drone under the direct control of its operator, or a largely stationary device with one or more motors controlling moving parts – e.g. a robotic arm under computer control, used in industrial manufacturing, or a surgical robot under the remote control of a human operator.

Robotics has long had wide applications in industry, where repetitive tasks can be performed effectively and efficiently by machines, but as better or 'smarter' algorithms have been developed by computer scientists, more and more decision-making capabilities have been built into the robot, so that the robot is able to autonomously react to changes in its environment. Autonomous, self-driving cars such as those pioneered by Google are an example, even if we may think of them as cars rather than robots. The long communication delays between Earth and Mars mean that the robotic Mars rover[41] must use lots of event-driven, 'when...do' programming to be able to respond to what happens in its environment without waiting for control instructions from Earth. One application of machine learning will be programming robots to respond to input data to improve their own operation over time, perhaps particularly in such far-flung settings.

Whilst few might attempt such projects with a whole class, an advanced group of pupils or an extracurricular club might combine Design & Technology and computing skills, knowledge and understanding to build and program its own robot, perhaps entering a competition with the work or focussing on developing a solution to a real-world problem.

There are many platforms available on which pupils can develop their understanding of physical computing:

- Interface boards such as **Makey Makey** provide an easy way to connect the 'real world' beyond keyboard and mouse to a computer, and can be used directly with lots of different programs.
- Small microcontroller-based boards such as **CodeBug** and **Crumble** allow pupils to write programs on screen and then flash (download) them to the board to run independently; each of these examples can be used for both monitoring and control.
- Lego **WeDo** hardware provides simple sensors and a motor, and is connected to a computer or tablet via Bluetooth; it can be programmed using Scratch as well as its own tile-based language.
- Lego **Mindstorms** is a more sophisticated system, with a wider range of sensors and motors. Programs are downloaded to the Mindstorm control brick and can then run independently of the computer on which they were written.
- **Arduino** microcontrollers provide a range of boards able to work with a variety of different components. Again, programs are written on one computer and then downloaded to the Arduino board.

The two most common platforms for physical computing activities at present are the BBC micro:bit and the Raspberry Pi, both of which have been mentioned elsewhere in this guide, but both merit further discussion here.

Raspberry Pi

The Raspberry Pi is a small, cheap and high-powered bare-circuit-board general-purpose computer. It was created by a small, Cambridge-based team who had noticed the decline in undergraduate admissions to university computer science courses, and who believed that easy access to a computing platform which positively encouraged tinkering, programming and making would be helpful in supporting computing education. The first model went on sale to the general public in 2012.

39 www.raspberrypi.org/products/sense-hat/
40 For example a digital theramin: www.raspberrypi.org/magpi/ultrasonic-theremin/
41 http://mars.nasa.gov/mer/overview/

The Raspberry Pi foundation sees its mission as:

> **to put the power of digital making into the hands of people all over the world, so they are capable of understanding and shaping our increasingly digital world, able to solve the problems that matter to them, and equipped for the jobs of the future.**

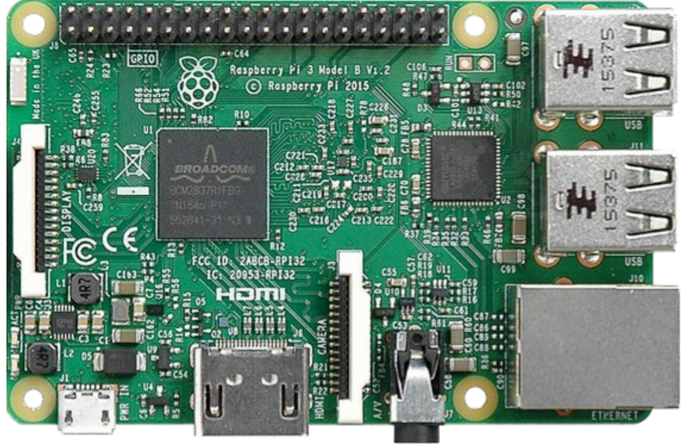


Figure 3.32 Raspberry Pi 3 (CC by-sa Herbfargus)

Version 3 of the Raspberry Pi (Figure 3.32) is a capable machine: the board provides four USB inputs, such as for mouse and keyboard, HDMI video and audio output to connect to a monitor or television screen, Bluetooth, WiFi and wired networking, a connector for a camera module (sold separately), and a set of general-purpose input/output pins used for the control and monitoring needed for physical computing. The processor is a four-core ARM chip running at 1.2GHz, similar to that used in smartphones, and it is powered by a micro-USB connector. It has 1GB of RAM but has no built-in permanent storage; instead, the operating system, application software and any data are stored on a removable Micro SD card, from which it boots.

The Raspberry Pi can use a number of different operating systems, including a version of Windows 10, as well as Acorn's RISC OS first released in 1987 and popular at the time in UK schools. Most users, however, use a version of Debian Linux, Raspbian, that has been tailored to both the Raspberry Pi hardware platform and the foundation's educational mission. Raspbian has an easy-to-navigate desktop GUI, using a similar system of icons and menus to other operating system GUIs.

The bundled software focusses, unsurprisingly, on programming but includes a broader range of application software too: there's the Libre Office suite, a web browser and an email client, as well as the high-end computer algebra system Mathematica. On the programming side, we have Python with the IDLE IDE (integrated development environment), a bespoke version of Scratch with support for the GPIO pins, and Sonic Pi, a Ruby-like language for composing (and performing) music. Raspbian includes Minecraft as standard, with the API (application programming interface) to allow Minecraft to be controlled through programming in Python. Many additional programs can be installed very easily: for example, the command (typed in the shell):

```
sudo apt-get install tree
```

is all it takes to install the tree utility.

The Raspberry Pi Foundation has assembled a collection of curriculum resources, many of which are directly relevant to the Level 3 computing curriculum, with others providing much that might inspire pupils in extracurricular computing clubs within or beyond school.[42] The Foundation merged with Code Club towards the end of 2015. Whilst the Code Club activities[43] have been written with primary pupils in mind, many, particularly those on HTML and Python, would be appropriate for Level 3.

The Raspberry Pi Foundation has developed two-day face-to-face PiCademy workshops. Day 1 is spent learning about computing and the Raspberry Pi, including physical computing, Minecraft and Sonic Pi. On day 2, participants work in teams, with contributions from the Raspberry Pi team, to develop their own project ideas. The Raspberry Pi community also hosts frequent local 'Raspberry Jam' events, in which community members meet to share their knowledge, learn new things and show off what they've done with their Raspberry Pi.

BBC micro:bit

Back in the 1980s the BBC played a pivotal role in the early days of computing education. The BBC sponsored the development of a home computer by Acorn, the BBC Micro, and developed television content to promote computer literacy in the home.

42 www.raspberrypi.org/resources/
43 http://projects.codeclubworld.org/en-GB/index.html

The BBC Micro was chosen as one of the computer models that would be supplied to every school.

Building on this legacy, and as part of a year-long 'Make it digital' initiative across the BBC's media, the BBC drew together a consortium of some 29 partner organisations (including ARM, Samsung, Microsoft and Lancaster University) to develop the BBC micro:bit, with the aim of inspiring 'young people to get creative with digital and develop core skills in science, technology and engineering'.

In 2016, close on 1 million micro:bits were distributed to the entire Year 7 national cohort. Significantly, the micro:bits, whilst distributed via schools, are intended to be given to the pupils themselves. Schools, pupils and others are able to buy further micro:bits if they desire.

The micro:bit (Figures 3.33–3.34) is smaller than the Raspberry Pi and, unlike the Raspberry Pi, it cannot be programmed directly, but rather must have programs downloaded to it from a connected computer, tablet or smartphone.

Figure 3.33 BBC micro:bit showing buttons, 25 pixel display and IO connector. CC by-sa Gareth Halfacree

Figure 3.34 The other side of the BBC micro:bit showing power and data connectors, processor, accelerometer, compass and reset button

The hardware includes an ARM microcontroller, running at 16MHz, 16KB of RAM and 256KB of flash memory, in which programs are stored. Input is through two buttons, an accelerometer, a magnetometer and the GPIO edge connector. Output is through a 25-pixel display or via the GPIO connector (for example, a speaker can be wired between connectors 0 and GND to produce simple audio). Connectivity is via USB and Bluetooth, as well as, somewhat unexpectedly, the GPIO pins.[44]

Programming the micro:bit is done via a web-based interface at www.microbit.co.uk/create-code which provides access to four different editors: a block-based javascript-like editor from Code Kingdoms, a blockly-based block editor, Microsoft's TouchDevelop, and an online version of the Mu educational IDE for microPython (see Figures 3.35–3.36).[45]

```
on shake
do  set choice to pick random 0 to 5
    change choice by 1
    show number choice
    pause (ms) 1000
    clear screen
```

Figure 3.35 Simple micro:bit dice program in Blocks

```
script amazing script (converted)
function main ()
  var choice := 0
  input → on shake do
    choice := math → random(6)
    choice := choice + 1
    basic → show number(choice, 150)
    basic → pause(1000)
    basic → clear screen
  end
end function
```

Figure 3.36 The same program converted to TouchDevelop

44 https://microbit-micropython.readthedocs.io/en/latest/tutorials/network.html
45 Alternative programming platforms are available such as Microsoft's www.pxt.io/

```
// When the microbit runs.
function onStart( ) {

}

function onShake( ) {
      microbit.say(Random.number(1, 6));
      wait(1000);
      microbit.clear();
}
```

A similar program in CodeKingdoms' Javascript editor (code view)

```
from microbit import *
import random

while True:
    if accelerometer.is_gesture("shake"):
        display.show(str(random.randint(1, 6)))
        sleep(1000)
```

A similar program in microPython

There are a couple of particularly nice features of the micro:bit code editors. Firstly, there's an on-screen emulator, so (with the exception of microPython) it is possible to test your code on screen without downloading it to the micro:bit (Figure 3.37).

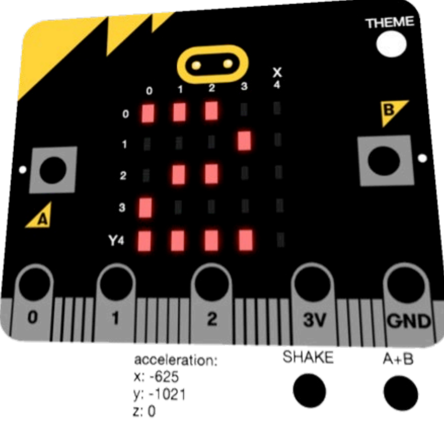


Figure 3.37 On-screen micro:bit emulator running the above program

Secondly, although these editors work in the web browser, they don't need any server side-processing to work: compiling the program you write happens inside the browser itself so, once they have been accessed online, they can be subsequently used offline. Micro:bit source code files can be uploaded or downloaded to the editor from the desktop.

Once you have written your program, to run it on the micro:bit itself you first compile the program, which produces a .hex machine code file. This contains a pre-compiled run-time environment, a device abstraction layer that sits between the hardware and your program, and your program compiled into ARM mbed machine code. Connecting the micro:bit to your computer via a USB cable, it shows up as if it were a USB memory stick, so you can simply drag the compiled .hex file across onto the micro:bit. A quick press of the reset button and your code should run very happily on the micro:bit itself. You can now plug in the micro:bit's own battery pack and disconnect the USB cable.

As you would expect, there's a good range of support materials available from the BBC itself and many of the other partner organisations. The project site at www.microbit.org/ is the best place to get started.

CAS (Computing At School) regional centres and Master Teachers have been active in supporting teachers with introductory CPD (continuing professional development) courses for the micro:bit, as have many of the partner organisations.

Classroom activity ideas

- Pupils could use the micro:bit, Raspberry Pi Sense HAT or GPIO pins, or a Makey Makey to control a simple game.
- Pupils might use sensors to collect weather data for school and then analyse for patterns, relationships and interesting exceptions.
- Ask pupils to write a program which could control a set of traffic lights in the correct sequence, perhaps on screen initially. Can they connect suitable LEDs to the computer and control these directly?
- Pupils could build and program a robot that could find its way out of a maze.

Further resources

BBC Bitesize (n.d.) *Raspberry Pi and Arduino.* Available from www.bbc.co.uk/education/guides/zdsbwmn/revision

BBC Cracking the Code clips (2013) Robots. Available from www.bbc.co.uk/programmes/p01661tn; and aerial photography using the Raspberry Pi: www.bbc.co.uk/programmes/p01661f7

BBC micro:bit site (n.d.) Available from www.microbit.org/

Berry, M. and Chambers, R. (2015) *Quick start guide to the BBC micro:bit.* London: Hodder Education.

IBM (n.d.) *Workshop on robotics.* Available from www.ibm.com/ibm/responsibility/initiatives/activitykits/robotics/

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

PiCademy (n.d.) Available from www.raspberrypi.org/picademy/

Raspberry Pi (n.d.) Education resources. Available from www.raspberrypi.org/education/ including *a teacher's guide to using Raspberry Pi* in the classroom: www.raspberrypi.org/guides/teachers/

Royal Academy of Engineering (2015) *Applying computing in D&T at KS2 and KS3: The 2014 national curriculum requirements.* Available from http://community.computingatschool.org.uk/files/6994/original.pdf

Technology will Save Us (n.d.) Available from www.techwillsaveus.com/

References

American Standards Association (1963) *American standard code for information interchange.* ASA X3, 4.

Austin, D. (2009) Image compression: *Seeing what's not there.* AMS. Available from www.ams.org/samplings/feature-column/fcarc-image-compression [2/1/17].

Brown, N. (2012) *Binary is an implementation detail.* Academic Computing. Available from https://academiccomputing.wordpress.com/2012/04/13/binary-is-an-implementation-detail/

DfE (2013) *National curriculum in England:* Design and technology programmes of study. London: DfE

Doctorow, C. (2012) *Lockdown.* BoingBoing. Available from http://boingboing.net/2012/01/10/lockdown.html

Huffman, D. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40 (9). 1098–1101.

Kolmogorov, A. (1998) On tables of random numbers. *Theoretical Computer Science* 207 (2). 387–395.

Michaelson, G. (2015) Teaching programming with computational and informational thinking. *Journal of Pedagogic Development,* 5 (1). 51-66.

Sellars, P. (2000) Perceptual coding: How Mp3 compression works. Sound on Sound (May).

Shannon, C.E. (1951) Prediction and entropy of printed English. *Bell System Technical Journal,* 30 (1). 50–64.

Shannon, C.E. and Weaver, W. (1949) *The mathematical theory of communication.* Champaign, IL: University of Illinois Press.

Sullivan, G.J. Ohm, J.-R, Han, W.-J, et al. (2012). Overview of the High Efficiency Video Coding (HEVC) Standard (PDF). *IEEE Transactions on Circuits and Systems for Video Technology (IEEE)* 22 (12). 1649-1668.

Turing, A.M. (1936) On computable numbers, with an application to the Entscheidungsproblem. *J. of Math,* 58. 345–363.

Walter, C. (2005). Kryder's Law. *Scientific American.* August, 25/7/2005.

Welch, T. (1984). A technique for high-performance data compression (PDF). *Computer* 17 (6). 8–19.

Wiegand, T., Sullivan, G.J., Bjøntegaard, G. et al. (2003) Overview of the H. 264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions,* 13 (7). 560–576.

Computer Networks

How do computers communicate?

Computer Networks

HOW DO COMPUTERS COMMUNICATE?

Useful as individual computers are for running programs, such as games, calendars and spreadsheets, to perform calculations and help manage information, it has really been through connecting computers together to form networks, and particularly through the internet – a network of networks – that they have had the most immediate impact on our lives. Consider how limited our use of technology in school would be if we had no access to the local network or the internet. Consider how frustrating many find it when, even temporarily, we have no data signal for smartphones, or no WiFi for a tablet or laptop.

The internet has made possible communication and collaboration with a diversity and immediacy as never before, and yet, perhaps like writing, printing and the telephone before it, it's something that most of us take for granted, and possibly have little understanding of. The computing curriculum sets out to change this: alongside developing pupils' computational thinking through practical programming, it includes requirements that primary pupils be taught to 'access websites and use navigation skills to retrieve information for a specific task' and 'use search engines to search the internet for specific or relevant information'. At Level 3, pupils have to 'demonstrate an understanding of how computers communicate and share information over networks, including the concepts of sender, receiver, address and packets.'

How does the Internet work?

The internet is a physical thing: it is the cables, fibre, transmitters, receivers, switches, routers and all the rest of the hardware that connect computers, or networks of computers, to one another.

The internet has been designed to do one job: to transport data from one computer to another. This information might be an email, the content of a web page or the audio and video for a video call.

The data that travels via the internet is digital: this means it is expressed as numbers. All information on the internet is expressed this way, including text, images and audio. These numbers are communicated using binary code, which is made up of 1s and 0s, using on/off (or low and high) electrical or optical signals. Binary code is similar to the Morse code used for the telegraph in Victorian times, but it's much, much faster. A good telegraph operator could work at maybe 70 characters (letters) a minute, but even a basic school network can pass data at 100 million on/off pulses a second, enough for some 750 million characters per minute. One transatlantic fibre connection has the capacity for up to 24 **trillion** characters per minute.

Digitised information needs to be broken down into small chunks by the computer, before it can be sent efficiently. These smaller chunks of data are known as 'packets'.

The small packets can be passed quickly through the internet to the receiving computer where they are re-assembled into the original data. The process happens so quickly that high-definition video can be watched this way, normally without any glitches.

The packets don't all have to travel the same way through the internet: they can take any route from sender to recipient. However, there is generally a most efficient route, which all the packets would take (Figure 4.1).

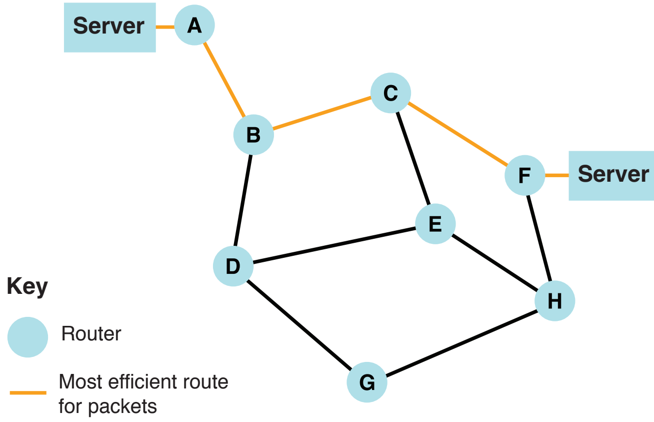


Figure 4.1 A sample network: Note there is more than one route for packets to travel

It is perhaps easier to understand how the internet works nowadays by looking at a picture of how it worked in 1969 when it started (Figure 4.2):

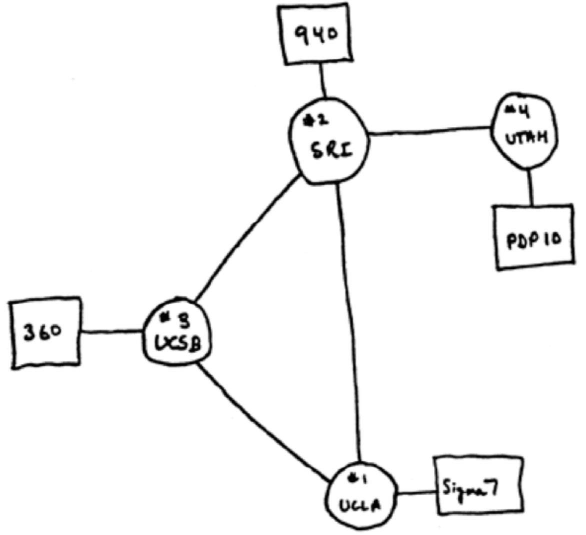


Figure 4.2[1]

Here you see the internet made up of just four routers: UCLA, SRI, UCSB and UTAH. Each router is a piece of hardware that passes packets of data from the computer it is connected to (in the case of UTAH, that's PDP10; in the case of UCLA, it's SIGMA 7) – and perhaps any terminals connected to those computers – to any of the other three computers and their terminals.

So, if you were using the PDP10 computer at the University of Utah and sent a message to someone at UCLA, your message would be passed first to your router at Utah, then on to the router at Stanford Research Institute (SRI), then (normally) to UCLA's router, where it would be passed on to the intended recipient on their SIGMA 7.

The internet is obviously much, much bigger than this example. In real life, the journey of a packet of data from your home computer to one of Microsoft's server farms might look something like this:

- your home WiFi access point;
- your home switch and router (usually all in the same black box);
- switches in your nearest BT green cabinet;
- more switches in your local telephone exchange;
- London internet exchange;
- routers near Porthcurno in Cornwall;
- fibre optics under the Atlantic;
- further switches and routers in the USA. until Microsoft's internet connection at whichever of its data centres you are communicating with.

When you type a URL (such as www.bbc.co.uk or www.computingatschool.org.uk) into your browser, you send a packet of data requesting the content of these pages to be returned to you. But, before this can happen, the domain name first needs to be converted into numbers. This is the job of the Domain Name Service (DNS), which converts these familiar web addresses into numbers known as Internet Protocol (IP) addresses. The DNS itself uses the internet to look up (in the equivalent of huge phone books) the numeric address corresponding to the domain names, but it keeps a local record (cache) of these, so that the next time the domain name is requested, the IP address can be returned more quickly.[2]

Each packet has a destination IP address on it. With it the router can easily look up which way to pass the packet on.

1 From www.computerhistory.org/internet_history/
2 Simple Scratch-based simulation of DNS lookups at https://scratch.mit.edu/projects/105316834/#editor

Who can see the data we transmit?

There's nothing to stop routers from looking at the data in the packet before they pass it on (just as there was nothing to stop telegraph clerks reading the messages they passed on in Morse code).

To be able to send information, such as passwords or bank account details, secretly via the internet, it's important to encrypt the data first. This happens automatically when using the 'https' version of websites. In these situations, you will see a little green padlock displayed in your browser's address bar. The data are decrypted when they reach their destination – see pages 151 - 154 for more on cryptography.

Classroom activity ideas

- Ask pupils to draw a picture of the internet. This will allow you to spot any misconceptions they have, and provide an opportunity for pupils to share their understanding.
- Carry out this 'unplugged' activity to model how the internet passes packets of data.
  - Organise all but four of your pupils into groups.
  - Tell the pupils to choose one pupil in their group to be the 'group router'. The rest of the group will be 'computers'.
  - Ask the remaining four pupils to take on the role of 'internet routers', which connect the group routers together.
  - Give each 'computer' a numerical address, comprising a group number and a computer number (for example 1.1, 1.2, 1.3; 2.1, 2.2, 2.3, and so on; Figure 4.3).
  - Ask each 'computer' to write a short message to another 'computer' in a different group, splitting their message over three different slips of paper and marking their slips '1 of 3', '2 of 3' and '3 of 3'. Tell them to write their numerical address and the numerical address of the recipient, for example 'To: 2.2; From: 3.4; 2 of 3'. This is the 'packet header'.
  - Ask the 'computers' to pass their slips to their 'group router', who can pass these on one at a time to the 'internet routers'. They in turn pass them to the correct 'group router' who passes them to the recipient themselves, who can reassemble the message as their other packets arrive.

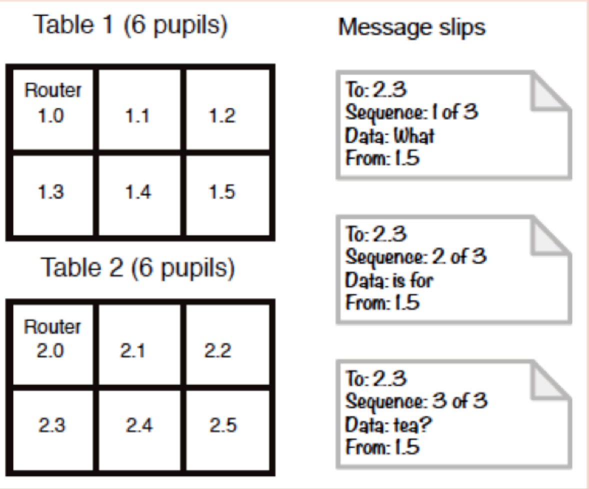


Figure 4.3 Role-playing a computer network in class

- Investigate the physical infrastructure of the school network. Tell the pupils to walk from their laptop to the local WiFi point or to follow the network cable from the computer to the classroom switch. Next, walk together to the school's main network switch, firewall and router. If you can, then walk down to the nearest BT green cabinet, and perhaps to your local telephone exchange, depending on how close this is to you.
- Explore the steps on the journey of a packet using the 'tracert' command at the Windows command prompt, if you have access to this. Also see the Visual traceroute reference in Further resources.
- Ask your school network manager to talk pupils through how the school network connects their computers to the rest of the internet.

Further resources

Bagge, P. (n.d.) *Network, internet and web search planning.* Code-it. Available from http://code-it.co.uk/netintsearch

Barefoot Computing (2014) *Internet services.* Available from http://barefootcas.org.uk/programme-of-study/multiple-services-provided-networks-internet/internet-services/ (free, but registration required).

Barefoot Computing (2014) *Modelling the internet activity.* Available from http://barefootcas.org.uk/programme- of-study/understand-computer-networks-including- internet/ks2-activity-modelling-the-internet/ (free, but registration required).

BBC Bitesize (n.d.) *Introduction to networks.* Available from www.bbc.co.uk/education/guides/zc6rcdm/revision and Internet and communication: www.bbc.co.uk/education/guides/z8nk87h/revision

Blum, A. (2012. *Discover the physical side of the internet.* TED. Available from www.ted.com/talks/andrew_blum_what_is_the_Internet_really

Blum, A. (2012) *Tubes: Behind the scenes at the internet.* London: Penguin Books.

Digital Schoolhouse (n.d.) Mark Dorling and others: *Networks unplugged.* Available from www.digitalschoolhouse.org.uk/documents/networks-unplugged-workshop-pack

Naughton, J. (2012) *From Gutenberg to Zuckerberg:* What you really need to know about the internet. London: Quercus.

Raspberry Pi Learning Resources (n.d.) *Networking lessons.* Available from www.raspberrypi.org/learning/networking-lessons/

Visual traceroute to find the path from their web server to an internet address. Available from www.yougetsignal.com/tools/visual-tracert/

What can you do with the Internet?

One way to think of the internet is as the train network, efficiently routing trains of all kinds from one point to another, irrespective of what those trains contain: some will have passengers, others freight, others are perhaps maintenance stock. Similarly the infrastructure of the internet can be used for lots of different things. At present, we are most familiar with the web as the main application of the internet, but the internet pre-dates the web by a couple of decades and there are many who think we will be using the internet, or something very like it, long after the web becomes a historical curiosity.

The services which run on computer networks, including the internet, fall into roughly two groups (Figure 4.4):

(1) client–server: one computer (the client) accesses services or content running or stored on another, typically larger, computer (the server);

(2) peer-to-peer: two computers communicate directly as equals, passing data directly to and from each other.

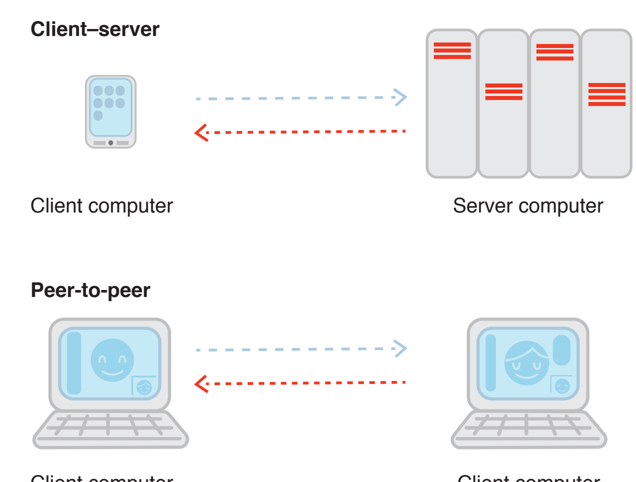


Figure 4.4

The World Wide Web (see page 122) fits into the client–server model, but so do lots of other services which use computer networks and the internet as a means of communicating.

A school network will often have one or more computers acting as servers, responding to requests from the desktop, laptop and tablet computers which act as clients. On a local area network (LAN) like this, the servers might provide central storage and backup for files, access to documents and so on, from any computer on the network; a management information system (such as SIMS – the Student Information Management System); local email accounts; access to printers; username and password authentication; filtering and logging of access to the web; and even locally-stored copies of frequently visited web pages.

Email is a good example of a client–server system using the internet (although many people's experience of email is as webmail accessed through a browser like Internet Explorer or Chrome). The journey of an email might be something like this:

- Alice opens up Outlook and starts typing in her email to Bob. She includes Bob's email address, bob@builders.com, in the 'To' line of the email and clicks 'send'.
- The email is transmitted via the internet (or the local network) to her outgoing mail server. If the email is intended for another domain (builders.com here) rather than Alice's own (lookingglass.org) then Exchange will forward the email as packets of data via the internet, which routes these through to the incoming mail server for builders.com as discussed above.
- The inbound mail server at builders.com (again, perhaps running Exchange) re-assembles the message from the packets of data, accepts this and stores it ready for Bob to collect.
- Later on, Bob's email client (perhaps also Outlook) connects to his mail server and asks if there are any messages for him. The one from Alice gets transmitted to Bob's computer via the local network or the internet, where Bob can read it in his email software.

Although it might look to Alice and Bob as though they are communicating directly with each other, all their emails are going via the outbound and inbound mail servers. Notice that the contents of their emails are not encrypted, so the organisations running the two mail servers can read the contents of these messages if they wish.

Not all communication on the internet uses a client–server model. For example, peer-to-peer communication is a model used for Skype and a number of other video conferencing or voice-over-internet systems. Although Skype uses a server to maintain a list of logged-in users and the IP addresses of their computers, when a call is connected, the packets of data that make up the digitised video and audio for the call are routed directly through the internet between the two parties.

Some online gaming websites use a similar peer-to-peer system, as does BitTorrent (Cohen, 2003), a protocol which allows large files to be shared between many computers by allowing direct peer-to-peer connections, and blockchain (Nakamoto, 2008), a distributed ledger system for transactions in cryptographically-generated ('mined') currency. Because peer-to-peer connections are harder for large organisations to monitor, they are favoured by those using the internet for criminal purposes, for example the use of the BitTorrent protocol for illegally sharing copyrighted material, or the blockchain-based Bitcoin for purchasing illegal goods.

Classroom activity ideas

- Role-play can be used very effectively to teach how email works and issues with email security. Explain to pupils that email addresses can be 'spoofed' or accounts hacked – So, not all emails are from whom they appear to be. Warn pupils that files attached to emails can contain viruses. Also explain that links in emails can sometimes point to websites that are set up to capture personal information such as passwords. You might like to run this as part of a larger topic looking at the effective and safe use of email, perhaps in a twinning project with a class in this or another country.
- Use a video conferencing system to allow experts to talk to the class or to allow two classes to communicate. As you set up the computer, talk through the technical aspects of the call with your pupils. Note: Skype and most other video conferencing systems don't allow children to register for accounts, so you will need to run this as a whole-class activity.
- Encourage pupils to talk about how they and their families use the internet to communicate, highlighting any services they use in addition to the World Wide Web.

Further resources

Google Green (2012) *Story of send on Google Green* (a short cartoon about the journey of a Gmail-based e-mail). Available from www.youtube.com/watch?v=5Be2YnlRIg8

Guha, S., Daswani, N. and Jain, R. (2006) *An experimental study of the Skype peer-to-peer VoIP system.* Available from http://saikat.guha.cc/pub/iptps06-skype.pdf

The journey of a letter. Available from www.anpost.ie/anpost/schoolbag/primary/our+people/the+journey+of+your+mail/

What is the World Wide Web?

In 1989, British computer scientist Tim Berners-Lee decided to combine the capabilities of the internet with the functions of hypertext (documents that include hyperlinks that allow connections to be made between different files; see Figure 4.5) to manage information systems at CERN where he was working (Berners-Lee, 1989).

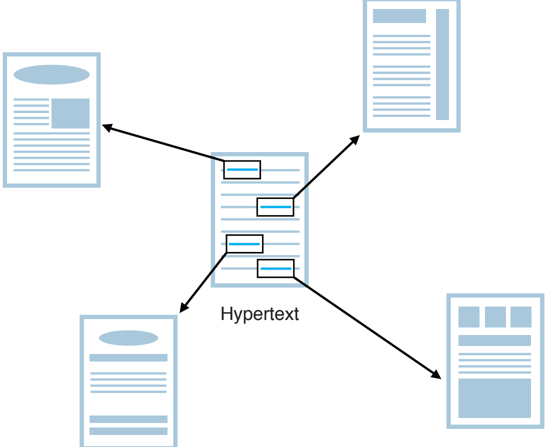


Figure 4.5 The links in the hypertext take the reader to different documents which extend or support the information in the original document

Berners-Lee developed a specification for how an internet-based version of hypertext would work, and then wrote the software for the first web servers and web browsers. The result was the World Wide Web.

The internet is about connecting computers together, but the World Wide Web is about the connections between documents (Figure 4.6). When you click on a web link, another web page is requested from (typically) a different web server somewhere else on the internet.

The content of this web page is then delivered to your web browser.

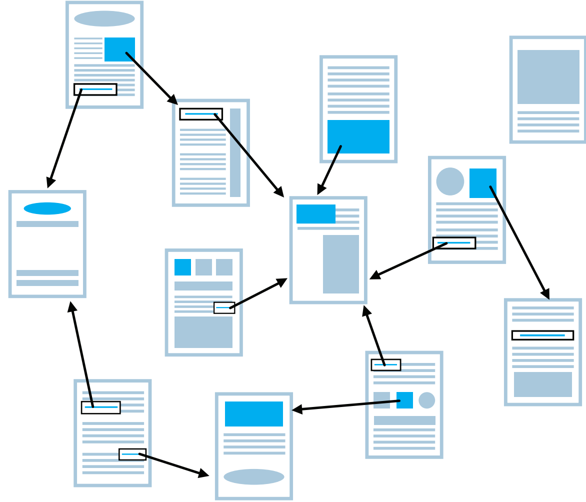


Figure 4.6 The World Wide Web is about the connection (the links) between documents

What standards does the World Wide Web use?

To ensure that all computers could communicate with one another, Berners-Lee developed a set of standards (called protocols) for the web. Versions of these are all still used today.

HTTP (HyperText Transfer Protocol)

This is the process that computers use to request and transfer hypertext to one another.

The web is a client–server system: we use a web browser on our computer to request a web page from one of the many, many web servers connected to the internet. The request travels as a packet of data via switches and routers until it reaches the intended web server. The server responds by sending back the content of the page, together with any images and formatting instructions and mini programs (typically in JavaScript) needed for the page. If the page isn't there, it sends back a '404: Not found' error message – sometimes you will see other error messages too.

Remember that the internet doesn't encrypt packets of data: there's another version of HTTP, called HTTPS, where the request for a page, the contents of the page and any information entered into a form (such as a password) are sent over the internet in an encrypted form. This encryption can sometimes be bypassed by network managers and government agencies.

URL (Uniform Resource Locator)

URLs are the precise location on the web where web pages or their components are stored. They are what you type in to your browser's address bar to request a page.

Each bit of a URL means something. Let's look at the URL of one of the first web pages – Berners-Lee's home page for the World Wide Web project itself – to work out what each bit means:

http://info.cern.ch/hypertext/WWW/TheProject.html

- 'http' is the protocol we are using to request hypertext and the content that comes back – see above.
- '://' is just punctuation – Berners-Lee now thinks it would have been better if he'd skipped the // bit!.
- 'info' is the name of the web server we are connecting to. Often this will be 'www' these days, or it's just omitted, as the main web server for the organisation will be assumed.
- 'cern' is the name of the organisation, in this case the European Centre for Nuclear Research.
- 'ch' is an abbreviation for the country where the organisation has registered its domain name, in this case Switzerland. Some countries also show what sort of organisation is registered; for example, 'co.uk' for a commercial site and '.sch.uk' for a school site in the UK. If no country is shown, then it will be registered in the USA: '.com' for commercial sites, '.edu' for university sites, and so on.
- 'hypertext' is a directory (folder) on the web server.
- 'WWW' is a directory inside the 'hypertext' directory on the web server.
- 'TheProject' is the name of the actual file we are requesting, in this case a web page about the World Wide Web project. Sometimes you don't see a file name at the end of a URL, in which case the web server will send back the default file for the directory, often an index page such as 'index.html'.
- '.html' is the file extension, which shows what format the page is written in, in this case HTML. This is like '.doc' or '.docx' for a Word file or '.jpg' or '.jpeg' for an image.

Although it is often convenient to use search engines like Google or Bing to find pages rather than typing in URLs, the URL is a good way to check that you're connecting to the intended web server (rather than a spoof website). URLs are also needed when acknowledging sources of information and for creating links between pages (and so building more of the connections that make the web so useful).

HTML (HyperText Mark-up Language)

HTML is the computer language (code) in which the content and structure of a web page are described, or 'marked up'.

The content of web pages is stored in HTML format on web servers. Creating a web page involves writing (or getting a computer to generate) the HTML that describes the page. HTML can be read, and written, by humans as well as computers. You can view the HTML source code for any web page using tools built into your web browser. (There's a menu command to do this, or you can press 'ctrl-u' in Internet Explorer.)

These days, the HTML for a web page might not be stored as a file on the web server: in content management systems, when a page is requested, it will be generated automatically using a database of content, a template and some programs running on the web server, perhaps written in Python or PHP. For example, every time you visit www.bbc.co.uk/newsround/, the page will be generated using the latest news in the database.

More recently, a couple of other languages have come to play an important part in developing the web.

CSS (Cascading Style Sheets)

CSS provides formatting information alongside the content and structure of HTML, allowing designers and developers to specify exactly how the content of the page should be displayed in the web browser on a computer, tablet, smartphone or printer.

JavaScript

JavaScript is a programming language that can be interpreted by the web browser itself, allowing interaction with the content of a page to be handled by the user's computer (the client) rather than on the server itself. The web-based version of Office 365 relies heavily on JavaScript.

What's the most amazing thing about the web?

The amazing thing about the web isn't really these technologies though. It's that, from its early days as the preserve of academic scientists, so many organisations and individuals have connected their own web servers to the internet and added their own content to the web. In part, this was because Berners-Lee created a system that was accessible, scalable and extensible, capturing the imagination of many, but it's also because he and CERN gave it to the world for free – the standards and the technology were entirely open, without any central authority or commercial company licensing or charging for their use.

Classroom activity ideas

- Encourage pupils to look at the different parts of the URLs for the web pages they visit, asking them to explain what each part of the URL means. Make a display showing the different parts of some interesting or common URLs.
- Ask pupils to talk to their parents, grandparents or carers about the difference the World Wide Web has made in their lives.
- Tell pupils to keep a diary of the different ways they use the web over a week.
- It's not too tricky to set up a web server in school, although providing access to this from the rest of the internet may be harder. With access to a web server, either on the school network or via the internet, pupils could create their own web pages either in HTML or using a content management system, for others to view. They could install a number of open-source applications such as Moodle or Wordpress, configuring these as they wish. They might also watch and analyse the data logged by the web server as it responds to page requests across the network.[3]

Further resources

BBC Bitesize (n.d.) *What is the world wide web?* Available from www.bbc.co.uk/guides/z2nbgk7

Berners-Lee, T. (n.d.) *Answers for young people.* Available from www.w3.org/People/Berners-Lee/Kids.html

CERN (n.d.) The *original CERN home page* for the web. Available from http://info.cern.ch/hypertext/WWW/TheProject.html

Mozilla Webmaker (n.d.) *Web literacy whitepaper.* Available from http://mozilla.github.io/webmaker-whitepaper/

Raspberry Pi Learning Resources (n.d.) *Build a Python webserver with Flask.* Available from www.raspberrypi.org/learning/python-web-server-with-flask/

Wayback Machine (n.d.) To search for historic web pages. Available from http://archive.org/web/

How do you make a Web Page?

There are plenty of tools available for you and your pupils to create your own content for the web.

Your school's learning platform, or Virtual Learning Environment (VLE), provides one way to get content online, as do blogging platforms like WordPress. These platforms usually include a 'WYSIWYG' ('what you see is what you get') editor. This makes writing content for the web similar to using Microsoft Word, with a range of formatting controls built in. In most of these editors, you can swap into code (or source view), seeing and editing the HTML itself. This can be a good introduction to working directly in HTML, as you can always swap back to the WYSIWYG view to see the effects of editing the code.

Giving pupils some experience of writing content for the web through editing HTML 'by hand' is well worth doing, although it isn't, strictly speaking, programming. It adds to their understanding of networks, including the internet, that the curriculum

3 GitHub offers free hosting of static webpages: https://pages.github.com/

expects, and is one more way of using software on a range of devices to create content. It is also a good way to get pupils used to working in a formal, text-based computer language. As with other text-based languages, working in HTML helps reinforce the importance of spelling, punctuation and grammar: mistakes in the mark-up of the page usually become quite apparent in the way the browser displays the page.

Many pupils are likely to find these skills useful in the long term too, both at secondary school and beyond: developing content for the web is part of many jobs, teaching included.

What does HTML look like?

Let's compare the HTML code for a simple web page and the page itself.

```
<!doctype html>
<html>
   <head>
        <meta charset="utf-8">
      <title>A simple webpage</title>
   </head>
   <body>
      <h1>Origins of the Web</h1>
      <p>Tim Berners-Lee started working on
the world-wide web project in 1989.</p>
      <p>He was working at <a href="http://
home.web.cern.ch/">CERN</a> in Switzerland
at the time.</p>
      <img src="http://www.w3.org/Press/Stock/
Berners-Lee/2001-europaeum-eighth.jpg">
  </body>
</html>
```

Origins of the Web

Tim Berners-Lee started working on the world-wide web project in 1989.

He was working at CERN in Switzerland at the time.

Figure 4.7

Can you see where the content for the page (Figure 4.7) comes from in the code? Can you see what effect some of the HTML tags (the bits in the <...> angle brackets, like <h1> and <p>) have on how the content is structured?

Notice how most of the tags come in matched pairs, for example:

- <html> and ending </html> for the whole page;
- <head> to </head> for the information about the page, such as its character set and title;
- <body> to </body> for the content of the page;
- <h1> to </h1> around the main heading for the page;
- <p> to </p> around each paragraph.

Compare the underlined link in the web page with the corresponding code. In the code, <a> to shows where the link should be, and href="http://home.web.cern.ch/" inside the <a> tag details where the link should point to.

An image is inserted from elsewhere on the web, using a single <img> tag, this time without a matched closing tag, and again giving the location of the image, using src=" http://www.w3.org/Press/Stock/Berners-Lee/2001-europaeum-eighth.jpg" inside the <img> tag.

How do I get started with HTML?

Mozilla's Thimble tool for creating websites (available at: https://thimble.webmaker.org/) makes it easy to get started with coding in HTML, as it displays the source code alongside the resulting web page, as does Trinket.io in HTML (rather than Python) mode.

Instead of starting from a blank page, pupils can try editing other web pages, exploring the structure and HTML code of these pages, and seeing what effect changing the code has on how the page is displayed in the browser.

On Internet Explorer or Chrome, you can use the Developer Tools (hit F12 or launch via the menu) to view and edit the source code (the HTML code which describes the content and structure) for a page. Alternatively, you can install Mozilla's X-Ray Goggles as an active bookmarklet (see Further resources) to remix and share edited web pages.

Classroom activity ideas

- When using their learning platform, VLE or class blog, encourage pupils to swap from the normal WYSIWYG (what you see is what you get) mode of the built-in editor, into the code, source or HTML mode and try writing their post or page in that. Remind them that they can swap back and forth to see how the code relates to the page that's displayed. Give pupils a list of some common HTML tags to try out for themselves.
- Set pupils the challenge of making a parody of a web page by using either the Developer Tools in Internet Explorer, or X-Ray Goggles, to edit the code for the page. It's wise to decide some ground rules for this activity in advance. Show pupils how easily a spoof page can be created this way, and explain why it's so important to check the address of the page they are visiting, to confirm it is authentic rather than merely one which looks convincing.
- Rather than asking pupils to write up a story or a report using Word, challenge them to do this using HTML code to make a web page. Emphasise that they need to concentrate on the content and structure of their page, which is what HTML is designed for. Encourage them to add in links to supporting material using the <a> tag if they are creating a non-fiction account, and perhaps to add in some images from elsewhere on the web using the <img> tag.

Further resources

Codecademy (n.d.) *Curriculum materials.* Available from www.codecademy.com/schools/curriculum (registration required). See also: www.codecademy.com/learn/make-a-website

CodeClub World (n.d.) Resources. Available from http://projects.codeclubworld.org/en-GB/05_html_01/index.html and http://projects.codeclubworld.org/en-GB/06_html_02/index.html

Howe, S. (n.d.) *Learn to code HTML and CSS* (tutorials). Available from http://learn.shayhowe.com/

Mozilla X-Ray Goggles (n.d.) See the source code behind web pages using X-Ray Goggles. Available from https://goggles.webmaker.org/

Playto (n.d.) *App design basics. Learn to code using HTML and CSS.* Available from https://learn.playto.io/html-css/lesson/0

Raspberry Pi Learning Resources (n.d.) Google Coder resources. Available from www.raspberrypi.org/learning/coder-html-css-lessons/

Thimble (n.d.) Available from https://thimble.webmaker.org/

W3schools.com (n.d.) Tutorials on a wide range of computer languages. Available from www.w3schools.com/

How does a Search Engine work?

Search engines like Google and Bing have transformed the way we use the web. Instead of having to remember URLs for the pages we want, or following the links from one page to another, we can normally rely on these web-based programs to give us the most relevant results for our query.

Given how much we use search engines, it's important to use them effectively and efficiently, to show some discernment in deciding how far a particular page can be trusted, and to have some grasp of the algorithms that underpin them.

In order for Google or Bing to be able to respond to a search query, they use their index of the web. A search engine builds its index by using specially written programs called 'web crawlers'. The web crawlers create a huge copy of the publicly accessible bits of the web (called a cache) which is stored on the search engine's servers.

When a new or updated copy of a web page is added to the cache, an entry for the page will be added to, or updated in, the search engine's index of the web for each of the words on the page (typically ignoring small, common words like 'and', 'the' and so on). The web crawlers continue to build and update the cache by following all the hyperlinks in the page, requesting and making copies of those pages too, adding or updating index entries for them and following the links on those pages too. And so on.

So, when we type a keyword such as 'dog' into a search engine, it consults the index and returns a list of all the web pages on which that keyword appears. Typing in several keywords, for example 'dog' and 'bowl' would only return pages with both of these keywords, which helps to narrow down the set of results.

How are search results ranked?

The really clever bit about web searches is not the list of results but the rank order the results are put into. How do the search engine algorithms decide what to put top of the list?

Google's founders, Larry Page and Sergei Brin, recognised that the key to determining the relevance of a particular result was likely to lie in the links between other pages and the result. They realised that a high-quality page is a one that has lots of links pointing to it from other web pages, particularly if they too are high-quality results (Page et al., 1999). This is shown in Figure 4.8, where the larger the circle is, the higher the quality of the web page.

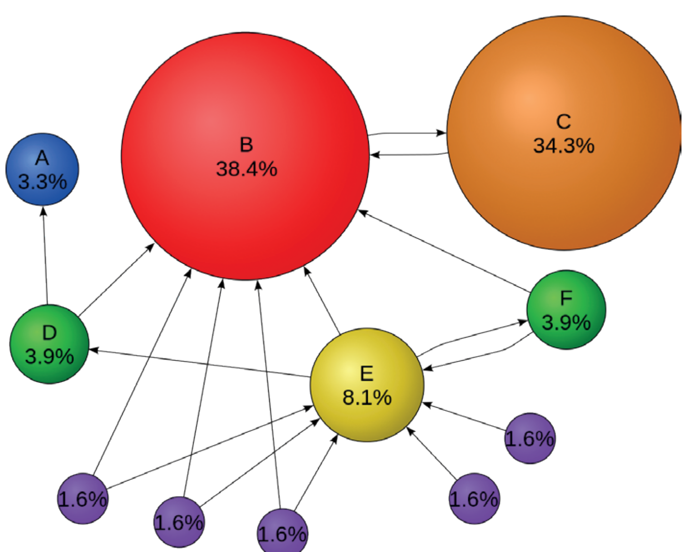


Figure 4.8

The cached and indexed copy of the (publicly accessible) web on the servers of search engines also includes the links between them. This allows Page and Brin's PageRank algorithm to work out which pages are considered the highest quality to other web developers (as they add links to those into their own content). Thus, for many queries, the Wikipedia entry will often be at the top of, or at least high up, the results list, not because of its accuracy or authority, or even because people click on this more than other results, but because lots of the other high-quality search results link to it.

The actual algorithms that search engines use can be very complicated and are frequently tweaked to keep one step ahead of the 'search engine optimisation' (SEO) industry, that tries to improve

the ranking for its clients' pages. These days, the ranking of results is typically personalised, based on location, the history of what the user has searched for and clicked on before, and close to 200 other factors or 'signals'.

When teaching pupils about how search engines work, point out the sponsored results which are shown above or to the side of those generated using this relevance algorithm. The sponsored results are also algorithmically generated, based on the keyword, some quality measure for the advert, the page it points to and often your search history. They are placed on a 'pay per click' basis: the search engine doesn't charge for showing the advert, but the advertiser pays when you click on it, so it's in the interests of the search engine to only show the most relevant adverts here.

The mechanics will vary from one search engine to another, but a good search engine should also filter out explicit content automatically, allow you to search within a particular site, and allow results to be filtered by their location (for example, just the UK) and by date range (for example, just pages created or edited in the last year).

Classroom activity ideas

- Encourage pupils to use search engines for independent or guided research projects. Get pupils to experiment with the effect that adding further keywords, or searching for phrases, (by putting quotation marks around the phrases) has on a set of results.
- Demonstrate, and ask pupils to use, some of the more advanced search features, such as filtering by date. Show pupils how they can view the cached copy of a web page (for both Google and Bing this is hidden under the green drop-down next to the URL on the results page).
- Read through the Digital Schoolhouse notes on a simulation of how a search engine works, based on Google engineer Doug Aberdeen's presentation at the 2012 CAS Conference (see Further resources). Print off the resources and run this as an activity with your class.

Further resources

Aberdeen, D. (2011) *Simulation from the CAS conference.* Available from https://youtu.be/bNp4ZP5CDcA?t=20m35s; qv www.computingatschool.org.uk/data/uploads/conf2011/real-life.pdf

Bing (n.d.) *Useful list of advanced search keywords in Bing.* Available from http://onlinehelp.microsoft.com/en-gb/bing/ff808421.aspx [2/1/17].

Cutts, M. (2010). *How search works.* YouTube. Available from www.youtube.com/watch?v=BNHR6IQJGZs

Dickman, P. (2012) *How Google search works.* YouTube. Available from www.youtube.com/watch?v=C8v7AM1o7uM

Digital Schoolhouse (n.d.) *Simulation of how a search engine works.* Available from http://community.computingatschool.org.uk/files/3874/original.pdf

Pariser, E. (2011) *Beware online 'filter bubbles'* (how individually focussed our search results are). TED. Available from www.ted.com/talks/eli_pariser_beware_online_filter_bubbles?language=en

References

Berners-Lee, T. (1989) *Information management: A proposal.* Available from https://www.w3.org/History/1989/proposal.html [2/1/17].

Cohen, B. (2003) Incentives Build Robustness in BitTorrent. In: *Workshop on Economics of Peer-to-Peer Systems.* Volume 6. 68–72.

DfE (2013) *National curriculum in England:* Computing programmes of study. London: DfE.

Nakamoto, S. (2008) *Bitcoin: A peer-to-peer electronic cash system.* Available from https://bitcoin.org/bitcoin.pdf [2/1/17].

Page, L., Brin, S., Motwani, R., et al. (1999) *The PageRank citation ranking: Bringing order to the web.* Available from http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf [2/1/17].

Productivity and Creativity

Can we carry on teaching our old topics?

Productivity and Creativity

CAN WE CARRY ON TEACHING OUR OLD TOPICS?

The 2017 Technologies Benchmarks make a clear distinction between Digital Literacy and Computing Science. The organisers for Digital Literacy:

> **Using digital products and services in a variety of contexts to achieve a purposeful outcome. Searching, processing and managing information responsibly. Cyber resilience and internet safety**

These encompass the older concept of 'ICT across the curriculum' and, as such, are the responsibility for all subject areas but in many establishments much specific teaching of these elements will be done alongside the Computing Science delivery.

The topics which encompassed the broad areas of productivity and creativity will remain an important part of the new curriculum.

Level 1 requires that the pupil 'uses digital technology to collect, capture, combine and share text, sound, video and images'. At Level 2 this extends to selecting the most appropriate applications and techniques and to 'share and collaborate'.

At Level 3, it extends to pupils being able to 'gather and combine data and information from a range of sources to create a publication, presentation or information resource, and that the pupil 'uses the most appropriate applications and software tools to capture, create and modify text, images, sound, and video to present and collaborate'.

In planning what to include, think back to the units that worked particularly well in your old scheme of work. Avoid having too many units which do little more than allow pupils to practise or reinforce skills they already have. Do include any opportunities to make connections between Digital Literacy and Computing Science – for example, video-editing work provides a great opportunity to develop the ideas of sequencing, which can be linked to sequencing in programming, as well as a chance to consider compression algorithms and file formats. Also it can link to digital literacy: have pupils consider the privacy implications of videoing one another and potentially sharing this with an audience beyond the class or school.

How can we make IT activities more meaningful for pupils?

Back in 2008, David Jonassen (Jonassen, 2008) coined the term 'meaningful learning' to describe learning that met a number of criteria: he and his colleagues were thinking particularly about learning activities that involved using technology, but the principles can be applied more broadly. Jonassen's list was:

- **Active:** a good IT activity should be one in which learners are **doing** something, not merely reading about something, watching someone else do something or listening to someone talking about something.
- **Constructive:** a good IT activity should be both constructive in the sense of pupils **making** something – that is, working creatively – but also in the sense of 'making meaning', of developing their mental model of how a particular technology works.
- **Intentional:** ideally, IT activities should allow pupils some element of choice in what they do, or how they accomplish something: this can often be done through specifying some required outcomes in functional terms, rather than particular tools that should be used to accomplish these.
- **Authentic:** where possible, IT activities should be embedded in pupils' experience, including, perhaps particularly, that of school: look for connections with other areas of the curriculum, for example, embedding the teaching of IT skills in work which develops pupils' understanding of other topics.
- **Co-operative:** again, where possible, look for activities where pupils can learn with and from one another, ensuring that they have the chance to talk purposefully and productively with one another, to share their ideas and insights with each other.

These ideas can be applied directly to projects in IT: for example, pupils could work together to create an online survey of other pupils about their views on the breadth of the school's curriculum, choosing for themselves how they might analyse and present the results. It is just as easy though to

apply these to topics in computer science, perhaps setting pupils the challenge of working together to develop a simple phonics game for younger pupils, leaving many of the decisions over implementation to pupils themselves.

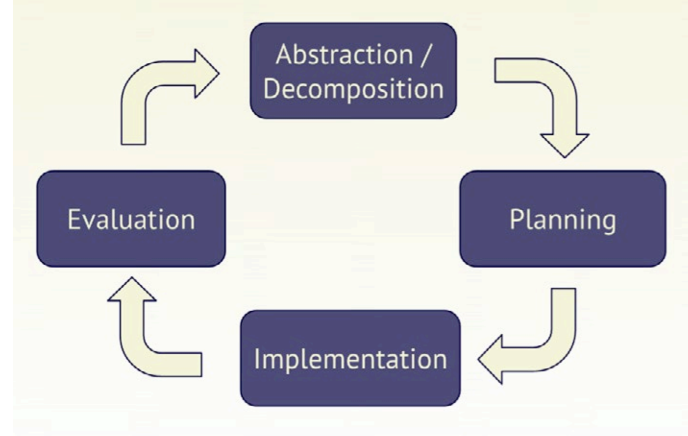
There's some quantitative evidence to support some of Jonassen's ideas. In his survey of meta-analyses of education research, John Hattie (2008) considers the evidence for the most effective use of technology in education. He argues:

> The use of computers is more effective when:
>
> - There is a diversity of teaching strategies.
> - There is teacher pre-training in the use of computers as a teaching and learning tool.
> - There are multiple opportunities for learning.
> - The student, not the teacher, is in control of the learning.
> - Peer learning is optimised.
> - Feedback is optimised.

In terms of IT, notice Hattie's emphasis on the student rather than the teacher being in control of the learning, and compare this with Jonassen's expectation of intentionality, and indeed the Level 2 curriculum requirement that pupils should be able to **select** as well as use and combine applications. Similarly, Hattie's emphasis on peer learning mirrors Jonassen's focus on co-operation. An emphasis on the collaborative use of technology is also supported in Steven Higgins and colleagues' (Higgins et al., 2012) synthesis of meta-analyses of technology in education; they state: 'collaborative use of technology (in pairs or small groups) is usually more effective than individual use.'

How should pupils go about project work?

It is important to find a balance between getting things done, adopting an agile approach of producing a 'minimum viable product' in the limited time available, and developing good working habits for more extended projects. One way to achieve this balance is to include a mix of short activities, in which pupils simply roll up their sleeves and create a spreadsheet, make a presentation or shoot and edit some digital photos, and more extended projects, in which the processes of planning, implementing, revising and evaluating the project are fully explored, including some occasions when these become part of a cycle of iterative development. Note that these phases mirror the computational thinking concepts and approaches of algorithms and decomposition, creating, debugging and evaluation. Working through the stages of a project in detail, sometimes repeatedly, is good experience for project work elsewhere in the curriculum and beyond school although, obviously, extended projects will take longer to complete than short, focussed tasks.



Where possible, look for ways to get pupils themselves involved in the work of managing the projects, including deciding what particular programs, and what equipment, they will need to use, and even in managing their time and the work of others in the team, if they are working collaboratively. The sort of project management skills involved in creative IT or digital media work are very similar to those required in developing computer software, so a similar approach can be applied to project work in both the IT and computer science strands of the curriculum.

What applications should pupils work with?

The 2017 Benchmarks are quite careful not to specify particular digital media, partly in recognition that new forms of digital content are likely to develop over time, and partly through a desire not to limit the forms in which pupils could explore ideas and express their creativity.

Technology currently available in most schools can be used for work across a very wide range of media, including: text (in both print and digital formats), images (both as vector illustrations and bitmap photographs), sound (as both recorded audio and composed music), animations (as stop-frame and scripted), video, and 3D (typically virtual representations, but some schools are starting to explore 3D printing).

Pupils need not be limited to working in just one medium – creative work in digital media will often combine multiple forms: a simple PowerPoint presentation or a website is likely to include text and images and perhaps video, audio and animations. Notice that the programme of study expects pupils to be able to combine applications across a range of devices. Pupils might begin by shooting video and recording audio using a tablet computer, importing this to a laptop to use video-editing software, combining this live footage with appropriately licensed images sourced from the web, before uploading their final edited film to an online video-hosting site.

Digital media can also be interactive – perhaps using little more than hyperlinks to allow non-linear navigation, but potentially drawing on more-complex scripting or programming.

What are digital artefacts?

A digital artefact[(1)] is a thing made using digital technology. The thing may not have a physical existence – it might be a virtual or a transient thing, but it must be something that has been made, something which provides evidence of its maker's creative work.

There's a very wide variety of digital artefacts which pupils might create. Oliver Quinlan identifies a number of categories in his landscape review of young people's digital making for Nesta (Quinlan, 2015):

- digital pictures;
- edited videos or visual effects;
- music;
- animation;
- games;
- websites;
- remixes and mashups;
- apps;
- software;
- robots;
- 3D-printed objects;
- edited photos.

Note that the list includes a number of artefacts which are made with code: games, apps and software, but perhaps also animation, visual effects, websites and robots. To this list we might add more conventional categories of artefacts created using digital tools: text, word-processed documents and desktop publishing, web content below the level of a site (such as a blog post), fan fiction or a forum contribution, spreadsheets, presentations, audio other than music, and 3D virtual objects (apart from in the context of games and animation).

Sir Ken Robinson defines creativity as 'the process of having original ideas that have value' (Robinson, 2011): both aspects here matter. Creative work should be original: in school, this should at least mean that it's a pupil's own work, not something where they have simply filled in a blank or copied

1 Confusingly, the term also means 'any undesired or unintended alteration in data introduced in a digital process by an involved technique and/or technology'. The programme of study uses it in its anthropological sense of 'an artifact that is of a digital nature or creation'.

something made by their teacher. Creative work should also be of value: at the very least, to the pupil herself, but perhaps also to her teacher and a wider audience. This means that pupils should aim to produce the best work they possibly can, and that their teachers and peers should be unafraid to offer constructive critical feedback on work so that it can be improved, further developing the computational thinking concept of evaluation.

As well as originality and value, creative work also implies that the pupil has **made** something. An emphasis on creativity recognises how powerful the process of making things for others is as a means to learning, as Seymour Papert did in the very early days of Logo programming in schools, coining the term 'constructionism' for this as a theory of how people learn (Papert and Harel, 1991). Pupils seem far more likely to develop an understanding of how software works, as well as becoming more skilful in using it, if they have the chance to use it creatively to make something original and valuable for others.

Related to this are ideas of craftsmanship. In describing craftsmanship values, Hoover and Oshineye (2009) discuss much that should have a place in creative computing lessons, such as the idea of a growth mindset, recognising the importance of hard work to the mastery of any craft, a willingness to experiment and be proven wrong, and the need for craftspeople to have some control over and responsibility for their work. Richard Sennett (2008) discusses the relationship of the craftsperson to their tools, recognising the importance of mastering the tools of a trade, and that tool use can be bound up with creative expression: 'tools used in certain ways organize this imaginative experience ... with productive results'. In the classroom, help pupils to become masters of the software tools and digital devices they use, helping them to develop confidence, competence and independence as they do so, and then encourage them to employ these, playfully or experimentally, as a means towards the expression of their own insights and ideas.

How can pupils learn to reuse, revise and repurpose digital artefacts?

Pupils do not have to work from scratch in creating digital artefacts. It is entirely legitimate for them to start with someone else's work, adapting this for their own particular purpose and audience, or using others' work as components within their own.

The Creative Commons licensing scheme[2] enables artists to license their work for others to reuse or develop without the need for further permission, and current copyright legislation permits some reuse, for purposes of parody[3] and for private study, subject to reasonable fair dealing limits. There are extensive online repositories of text, images and audio that can be reused, revised and repurposed, either in the public domain or licensed for reuse under the Creative Commons scheme. Public-domain or Creative Commons video resources are somewhat harder to obtain in general, in part due to the restrictions on downloading imposed by YouTube, however some downloadable, remixable content is available via Vimeo and the Internet Archive.

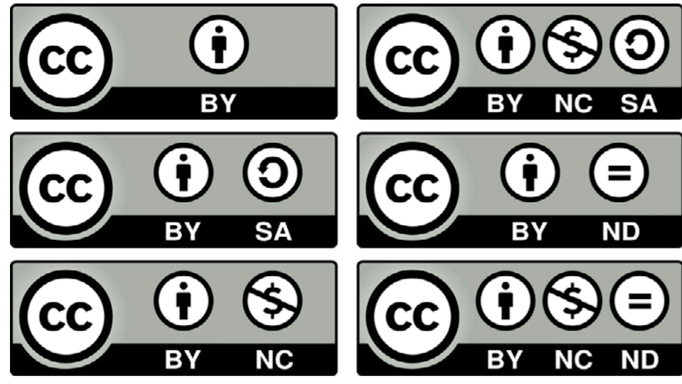


Figure 5.2 Creative Commons scheme

Content shared on the Scratch community site automatically carries a Creative Commons licence and can thus be freely remixed by any other Scratch users. Much other software is licensed in a way that permits reuse and further development, and there's a number of different licence terms available.[4] In general open-source software will permit free reuse and further development of source code, although

2 https://creativecommons.org/
3 www.gov.uk/guidance/exceptions-to-copyright#parody-caricature-and-pastiche
4 See https://opensource.org/licenses

the specific conditions vary according to the licence that applies. Projects such as Moodle, Wordpress, Firefox, Scratch, Android and Python have freely downloadable and editable source code, although it would be a brave pupil who decided to develop their own version of Android. To help manage the process of access to, remixing of and contributing back to the main development process for open-source projects, GitHub seems to be the platform of choice now.

The negative side of reuse is plagiarism. Reusing another's work as a basis for your own creative work is acceptable only if the original work is duly acknowledged. Artists sharing their work under a Creative Commons licence, or developers releasing code under open-source licences, are entitled to have their work, and their contribution to other's creative work, properly recognised; such recognition may be the only reward they have for freely sharing their work in this way. Passing off another's creative work as one's own is unethical, in breach of most forms of the Creative Commons licence and in contravention of academic discipline codes for higher education and public exams. With code, it is okay to look for others' solutions on StackOverflow or other sites, and it's (usually) okay to make use of their solutions in your own work, but only if you acknowledge that you have done so.

Are there principles for good design?

Whilst there are purely artistic, creative projects accomplished in digital media, very often in computing, pupils are likely to have a sense of audience and purpose for their work. Their creative work is closer to architecture than to sculpture – in most cases it has to be something which its audience will find useful as well as beautiful. That's not to say that its beauty is unimportant but, the design of a product should be led by the needs of its users, rather than just the desire for creative expression of its developer.

Any approach to user-centred design must acknowledge that functionality is fundamental. As Steve Jobs put it:

It's not just what it looks like and feels like. Design is how it works. (Walker, 2003)

User-centred design doesn't in itself specify principles of good design, but it does suggest a process that helps ensure that any design is fit for its audience and purpose:

- Specify who will be the users of a product and what they will use it for – audience.
- Identify the goals that must be met for the product to be a success – purpose.
- Plan and implement a solution.
- Evaluate the solution.[5]

A common design approach is to plan, and perhaps partially implement or prototype, many possible solutions, evaluating each against the audience and purpose criteria, before taking one solution through to full implementation.

Many lists of design principles have been drawn up. Dieter Rams, the designer of many of Braun's iconic 20th-century products, had the following list:

- Good design is innovative.
- Good design makes a product useful.
- Good design is aesthetic.
- Good design makes a product understandable.
- Good design is unobtrusive.
- Good design is honest.
- Good design is long-lasting.
- Good design is thorough down to the last detail.
- Good design is environmentally friendly.
- Good design is as little design as possible.[6]

5 See www.usability.gov/what-and-why/user-centered-design.html
6 www.vitsoe.com/gb/about/good-design

Figure 5.3 Some of Dieter Rams's designs for Braun. From https://www.flickr.com/photos/42035325@N00/15538137829/ CC by-nc-nd albyantoniazzi

More recently, the Government Digital Service has adopted the following principles for its work developing and refining the UK Government's online presence and interaction:

- Start with needs.
- Do less.
- Design with data.
- Do the hard work to make it simple.
- Iterate. Then iterate again.
- This is for everyone.
- Understand context.
- Build digital services, not websites.
- Be consistent, not uniform.
- Make things open: it makes things better.[7]

Comparing these and similar lists, there seems some agreement over some core principles of good design, such as utility, inclusion, honesty and simplicity. Simplicity seems particularly striking in the context of Digital Literacy where, until recently, many of us might have encouraged pupils to include all the possible 'bangs and whistles', clipart and animation in their media work to demonstrate that they could use every last aspect of the application software. Perhaps, when developing digital artefacts as part of the curriculum, we should encourage pupils to strive for a simpler design aesthetic, prioritising function over form, and placing the needs of users before the need to demonstrate prowess with particular tools.

As Apple's lead designer, Sir Jonathan (Jony) Ive, explains:

> **Simplicity is not the absence of clutter; that's a consequence of simplicity. Simplicity is somehow essentially describing the purpose and place of an object and product... The quest for simplicity has to pervade every part of the process. It really is fundamental. (Richmond, 2012)**

Classroom activity ideas

- Take a topic in which pupils are interested, perhaps from computing or from another area of the curriculum, and ask them to document it in a rigorous, critical way using the digital medium and tools of their choice.
- Working in a medium such as digital images or audio, provide pupils with some Creative Commons licensed content and set them the challenge of remixing this in the most original way that they can.
- Extend pupils' skills, knowledge and understanding in digital media work beyond what they covered in primary school, for example introducing them to the techniques of 3D animation using the open-source Blender platform.

Further resources

Design thinking for educators (n.d.) Available from www.designthinkingforeducators.com/

Duarte, N. (2008) Slide:ology: *The art and science of creating great presentations.* Sebastopol, CA: O'Reilly Media, 417–421.

Government Digital Service (n.d.) *Design principles.* Available from www.gov.uk/design-principles

7 www.gov.uk/design-principles

Kemp, P. (n.d.) *Introduction to 3D animation using Blender.* Available from https://goo.gl/5F9esz

Martinez, S.L. and Stager, G. (2013) *Invent to learn: Making, tinkering, and engineering in the classroom.* Torrance, CA: Constructing Modern Knowledge Press.

Nielsen, J. (2003) *Usability 101: Introduction to usability.* Available at https://www.nngroup.com/articles/usability-101-introduction-to-usability/ [2/1/17].

Norman, D.A. (2013) *The design of everyday things: Revised and expanded edition.* New York, NY: Basic Books Inc.

Quinlan, O. (2015) *Young digital makers surveying attitudes and opportunities for digital creativity across the UK.* London: Nesta.

Reynolds, G. (2011) *Presentation Zen: Simple ideas on presentation design and delivery.* Vancouver: New Riders.

Sefton-Green, J. (2013) *Mapping digital makers: A review exploring everyday creativity, learning lives and the digital.* Oxford: Nominet Trust.

Williams, R. (2014) *The non-designer's design book: Design and topographic principles for the visual novice.* Upper Saddle River, NJ: Pearson Education.

What can pupils do with Data?

The new computing curriculum places an emphasis on pupils working with numerical, quantitative data. This is a hugely important application of computer systems, and seems likely to become even more so in the future. There's much that you can do to provide pupils with a meaningful, authentic experience of working with both small and large datasets, and the skills and insights this work provides can be applied immediately in studying other subjects, as well as being useful for pupils as they work with datasets in the future. Given the ready access to tools with which pupils can generate interesting sets of data or access large open-data repositories on the web, the rather artificial database activities that many teachers and pupils found understandably dull should now be a thing of the past.

Online survey tools, such as Google Forms, allow pupils to design and deploy quick opinion polls or surveys, and then analyse, evaluate and present the results. Choosing topics of genuine interest to pupils, perhaps concerned with aspects of school life, can make activities like this much more engaging; or pupils could use these to survey opinion more broadly on local or national issues about which they have become concerned. In this case, care needs to be taken to avoid written-in, free-text responses, to avoid potential e-safety issues. Pupils should think about privacy and ethical aspects of such surveys – good practice includes principles of informed consent and anonymity; the latter is particularly important as, otherwise, data protection legislation might apply to the processing of personal data.

Data activities can also draw on automatically-generated data, perhaps using sensors to record environmental information (for example, a Scratch script to record the level of sound in class over a school day, or a Raspberry Pi-based weather station, or the data generated automatically in the log files of the school's website or its VLE (virtual learning environment) if you have access to these. Often there will be a 'dashboard' interface available to explore summaries of these data, but you might also be able to provide pupils with the raw data, so they gain some insight into how they are structured and so they can experiment with

analysing them in Excel or other software. You or your pupils could create random simulations to generate large datasets, for example using Excel to simulate rolling two dice 1,000 times. Pupils could then analyse these data to learn more about what was being modelled in the simulation – this is called the 'Monte Carlo method' and is an important application of computer modelling.

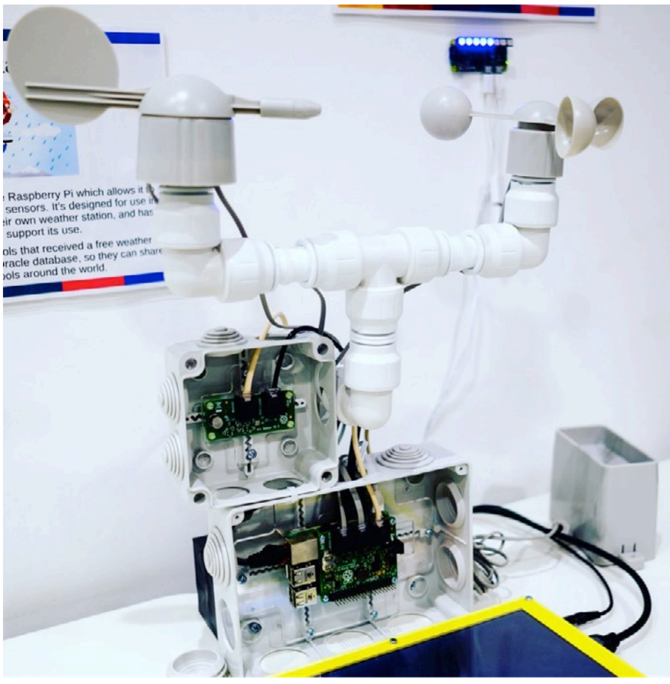


Figure 5.4 Raspberry Pi weather station,

(picture, Miles Berry)

Pupils can also analyse some genuinely **big** datasets made publicly available on the internet: for example, Google makes it easy to run searches for the occurrence of words or phrases in the vast number of books it has digitised, seeing how this changes over time.[8] Google also allows the trends in search-term popularity over time to be explored[9] – for example looking at the relative popularity of searches for 'ICT' and 'Computing' over time in the UK. The DfE provides detailed data on performance and other measures for all English schools,[10] and pupils could use Excel to analyse these data, for example exploring for themselves whether there's any relationship between proportions of pupils receiving free school meals and a school's GCSE results.

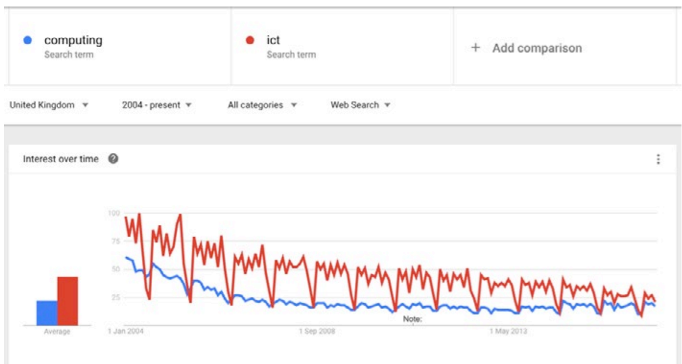


Figure 5.5 Screenshot from Google Trends, e.g. https://www.google.co.uk/trends/explore?date=all&geo=GB&q=computing,ict

There is an opportunity here to touch on some of the ethical implications of data processing. Pupils might think about the data which the school routinely stores about their activities, particularly that which becomes part of the DfE's national pupil database.

Classroom activity ideas

- Carry out activities that draw on automatically-generated data, perhaps using sensors (for example a Scratch script to record the level of sound in class; see Further resources).
- Organise your pupils to analyse some big datasets made publicly available on the internet. Help them to use n-gram viewer to search for the occurrence of words or phrases in the vast number of books that Google has digitised, and to see how this changes over time (see Further resources). Analyse how search-term popularity has changed over time, for example look at the relative popularity of searches for 'Britain's Got Talent' and 'The X Factor' over time in the UK, using Google Trends (see below).
- Discuss the ethical implications of data processing (that is, what others do with our data). Ask pupils to think about the detailed profile which internet, email or search-engine providers might build up through analysing each user's activity, as well as to what uses this information might be put.

8 https://books.google.com/ngrams
9 www.google.co.uk/trends
10 www.compare-school-performance.service.gov.uk/download-data

Further resources

BBC Two: *The Code* (2011) Using Google searches to predict flu. Available from www.youtube.com/watch?v=uEt8NuqBvPQ; see also Google Trends: www.google.com/trends/

Google (n.d.) *Google forms.* Available from www.google.co.uk/forms/about; or *Excel Surveys:* http://blogs.office.com/2012/11/16/excel-surveys/ for creating online surveys.

Mayer-Schönberger, V. and Cukier, K. (2013) *Big data: A revolution that will transform how we live, work, and think.* Boston, MA: Houghton Mifflin Harcourt.

Nelson, S.L. and Nelson, E.C. (2014) *Excel data analysis for dummies.* Hoboken, NJ: John Wiley & Sons.

Raspberry Pi Learning Resources (n.d.) *Weather station guide.* Available from www.raspberrypi.org/learning/weather-station-guide/

TED (n.d.) *Data talks.* Available from www.ted.com/topics/data

TED (2011) *A picture is worth a thousand words: what we learned from 5 million books.* Available from www.youtube.com/watch?v=5l4cA8zSreQ; see also n-gram viewer: https://books.google.com/ngrams

Wikipedia (n.d.) *Monte Carlo method.* Available from http://en.wikipedia.org/wiki/Monte_Carlo_method

How can we best support Collaboration?

The research cited earlier by Jonassen, Hattie and Higgins et al. all attests to the benefit of collaboration when using technology in education, but how can such collaborative use of technology be best developed?

The use of digital technology such as smartphones and the internet for communication has had a huge impact on the personal and professional lives of many: over 3 billion are connected to the internet worldwide, and the number of iPhones sold per day has exceeded the number of children born. It's hard to think of any sphere of life, including secondary education, which hasn't been changed by the near-ubiquitous nature of communication technology.

There's substantial evidence that young people are comfortable making use of a range of digital technologies to communicate with one another, although the extent to which they act safely and responsibly, or show discernment or wisdom when doing so, cannot be taken for granted. There's rather less evidence that young people are skilled in using technologies to work collaboratively on shared projects. Whilst the fit is far from perfect, one way of thinking about communication technology is to look at the size of the groups sending and receiving information, for example:

- one to one: email, skype, instant messaging;
- one to many: blogging, personal website, publishing on YouTube, podcasting, posting to social media, uploading projects to the Scratch community site;
- many to one: searching the web, watching YouTube, browsing social media, downloading and remixing projects from the Scratch or Kodu community sites;
- many to many: discussion forums, Wikipedia.

Irrespective of their access to or familiarity with technology outside of school, Level 2 pupils learn to collaborate using an online cloud-based service, for example, Glow or other platforms.

In the best primary schools, this will have been about developing children's understanding of these technologies, and some critical discernment about their use, rather than merely a set of skills in using one platform or another.

Can communication technology be embedded across the whole curriculum?

Yes! Many schools are now routinely using digital communication and collaboration technologies as part of their day-to-day work. Learning platforms, VLEs and systems such as Glow provide a reasonably convenient way for teachers to share resources and activities with a class, groups of pupils or even parents. They also offer one way in which pupils' work can be completed, submitted and sometimes marked online. Whilst the take-up of such technologies has been far lower than originally expected, the digital domain has become the default place for teachers' and pupils' work in at least some schools. In schools that have gone down this route, pupils can continue to access content, complete exercises, take part in discussion forums and contribute to collaborative projects from their home computers as well as within school. The enthusiastic take-up of tablet technology by many schools must almost assume ubiquitous connectivity, so that resources and outcomes can be stored, shared and, in the latter case, assessed.

Are pupils able to communicate with pupils in other schools?

Again, yes! Looking beyond the confines of an individual school's network, the internet can provide many opportunities for pupils in one class to communicate with or work collaboratively with pupils in another class, elsewhere in the local authority, the country or internationally. There's so much that can be gained through even a simple, email-based e-Twinning project, in which pupils, either collectively as a class or one-to-one as individual pupils, share opinions and experiences: think of the scope for exploring 'contrasting localities', for practising modern languages or looking at a period in history from a global perspective. As well as email, a shared discussion forum, perhaps hosted in one or other school's virtual learning environment, can make it easier to see the multiple perspectives on a topic, as well as allaying some of the e-safety concerns raised by providing pupils with individual email accounts. Digital media has a role here too, with pupils perhaps recording videos, taking photos or making presentations to share with the other class.

What sort of audience can pupils reach with their work?

For a previous generation, those who would read a pupil's work were perhaps just their teacher and maybe their parents; even if the work was put on display, the audience would rarely reach beyond the class itself. These days, it's easy enough for a school, an individual teacher or even particularly keen pupils to set up a blog, with the option of open access to all those connected to the internet worldwide, so that a child's work can reach an audience, potentially, of close on 3 billion others.

There's been a great deal of interest and enthusiasm of late in blogging in education: for many, this has been about sharing their educational insights with a community of fellow professionals, but it's also a great way to provide an authentic audience for pupils' work. Blogs can be used as a basis for partnership projects, as described above, with another class or group of classes.

There are obviously aspects to the safe and responsible use of blogging which teachers and pupils need to be aware of. Pupils should be taught to keep personal information private, so they will need to think carefully about what sort of content is suitable to post to a public blog. It's really important that comments posted to a class or school blog are moderated by a teacher before pupils get to see these – the workload here isn't too bad, but this needs doing to keep a blog free of unwanted advertising, inappropriate links or hurtful comments.

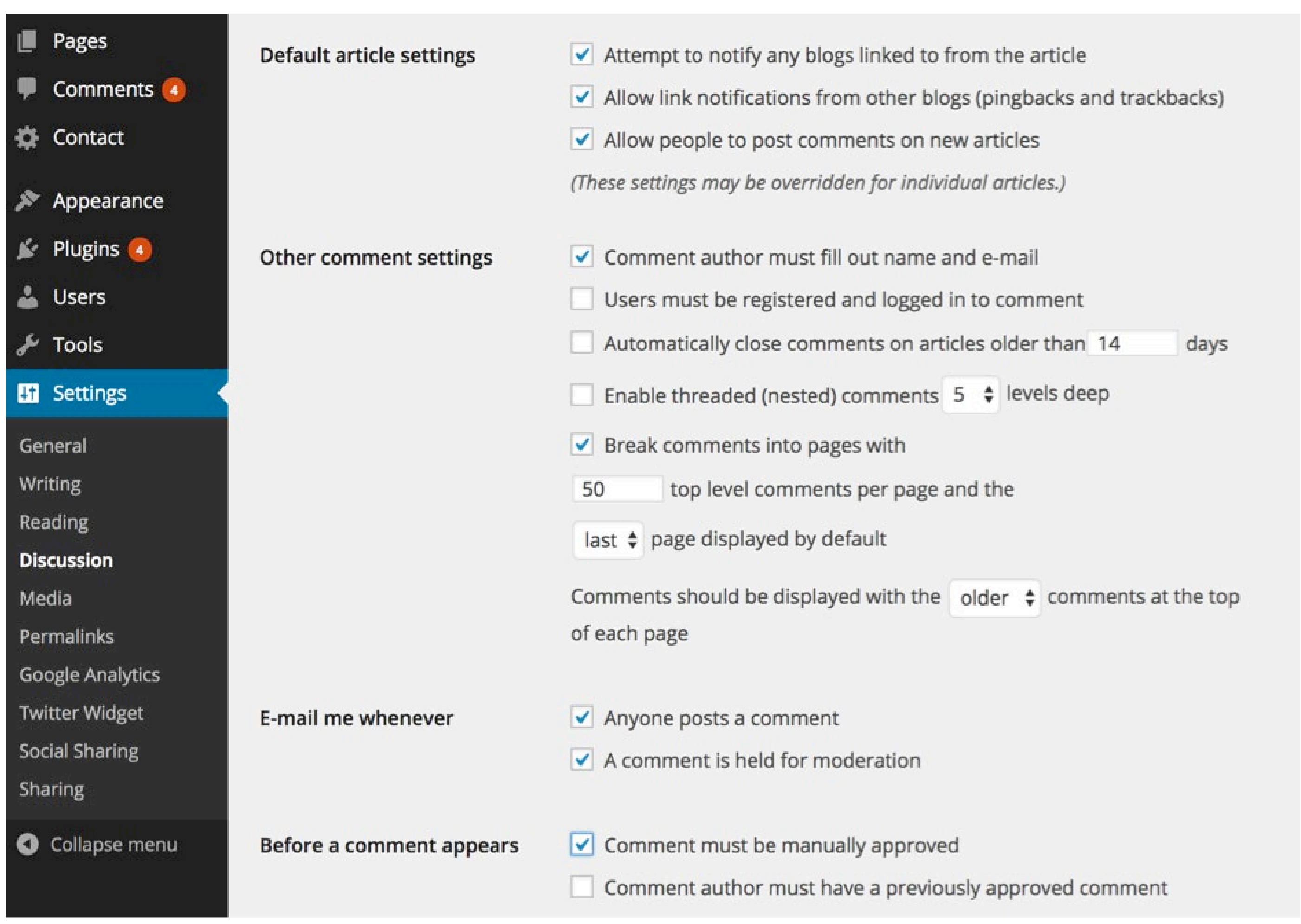


Figure 5.6 Screenshot of setting for moderating comments

Even without blogging, pupils could share their programming work through community sites for tools such as Scratch, Trinket.io and GitHub, although take care that you and they observe the terms and conditions which apply to these platforms.

What opportunities are there for pupils to work collaboratively?

The web and the internet make it easy for pupils to work collaboratively online, just as they have always been able to do in class, working together to research a topic, to draft, revise and complete documents or to make original, creative artefacts of their own, drawing on one another's skills and ideas.

Web-based platforms such as Google Apps for Education and Office 365 mean that pupils can work on documents, spreadsheets and presentation files together, either inviting comment and review from others, in much the same way as professional writers might, or through real-time collaboration in which several pupils edit the same document at the same time, seeing the changes made by others as they happen. Although this takes a little getting used to for some, the efficiency with which joint projects can be undertaken and reviewed can make this a very appealing, and exciting, mode of work.

On a broader canvas, teachers and pupils alike will be aware of the collaborative nature of Wikipedia, and that the contents of Wikipedia pages can be edited by anyone who has access to them. Whilst many in education steer clear of Wikipedia as a result, worrying about its reliability, this actually provides a good opportunity for pupils to become more discerning in evaluating digital content, and indeed to correct errors or add content to Wikipedia when they can.

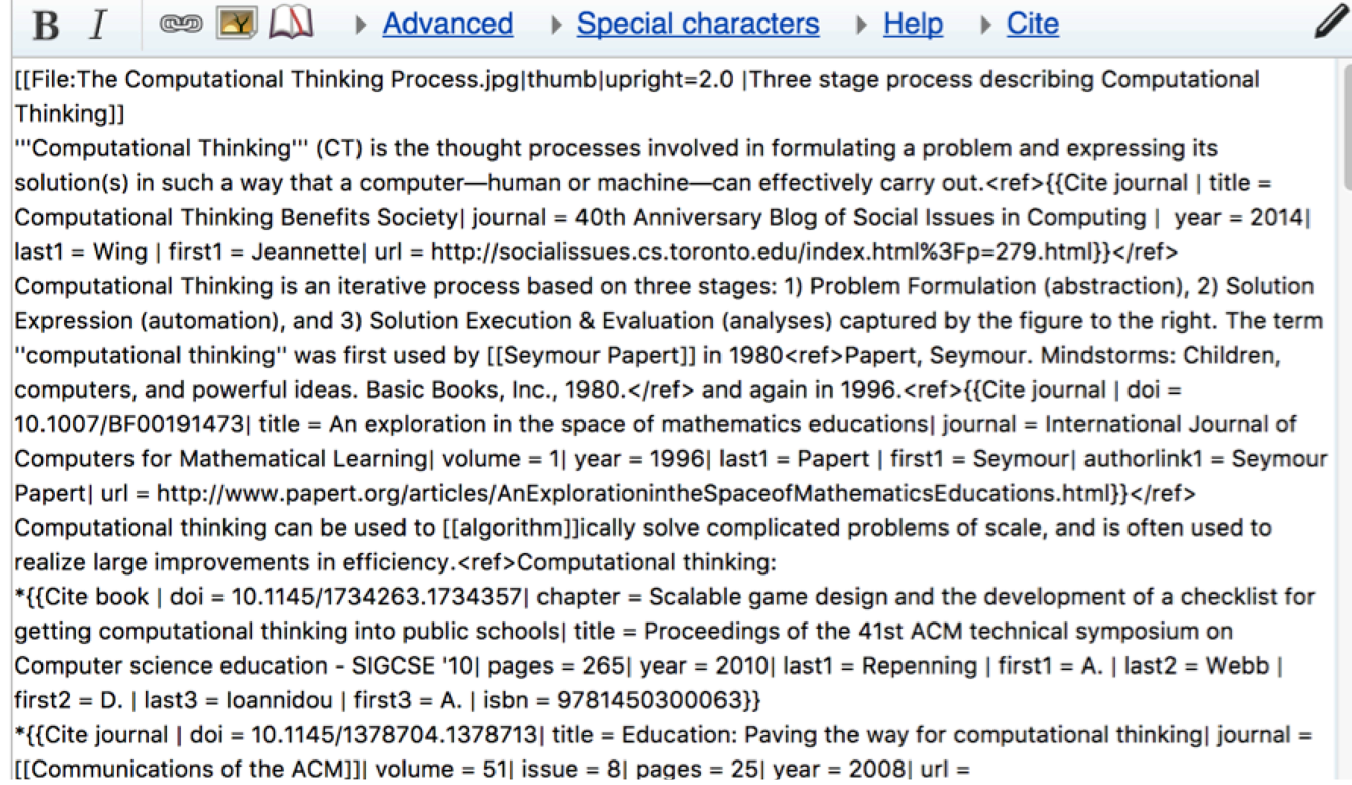


Figure 5.7 Screenshot of editing a Wikipedia entry

Online collaborative working is a very important part of software development – whilst much of the industry will keep these processes behind closed doors, those interested can get some flavour of this through open-source software projects such as Wordpress, Moodle and Firefox, and pupils themselves can get some experience in collaborative software development through the re-mix feature built into the Scratch community site.

GitHub supports version control and collaboration on software projects and online text. Whilst the learning curve can be quite steep, and the interface and associated vocabulary are not particularly intuitive, it has become the platform of choice for open-source software development. GitHub offers unlimited private repositories for students and those wishing to use it in school.

What ground rules should we establish?

It's important to establish an agreed set of rules for any online communication or collaboration activities, as can be seen from the importance given to specifying acceptable-use policies, and terms and conditions of use for computer networks and online platforms. Whilst pupils need to be aware that these conditions do apply to them, these are rarely written in a language which pupils will find particularly accessible, so teachers ought to spend time briefing pupils on what is expected of them.

It's helpful to have a simple set of guiding principles here: for example, pupils should behave online just as they would offline; this would include not being

deliberately hurtful, taking care of shared resources, leaving things as they would hope to find them, being prepared to stand up for doing the right thing, even if it's unpopular, and not talking to strangers.

Pupils ought to be aware that most online systems automatically log the activities that take place in them: that is, that someone (or something) is watching what they do online.

The Wikipedia community has established a set of community guidelines, its 'five pillars', which outline how contributors can work together and what constitutes acceptable behaviour in their online system, just as you are likely to have a code of conduct as a school or a class. These principles include ideas such as assuming good faith, being bold but not reckless, and striving for a neutral point of view which recognises both sides where there is disagreement.

Classroom activity ideas

- Set up a blog for computing in your school or for each class you teach, and ask pupils to share the best examples of their work, and comments on computing stories in the news, on the blog, reading and commenting positively on one another's posts.
- Request a school account from GitHub Education and use this as a way of sharing notes on lessons with your pupils, allowing them to 'fork' any handouts, notes or plans, to annotate these with their own notes from lessons.
- Pupils could work collaboratively to create a multi-page website, perhaps using GitHub Pages or a wiki platform, to present an informed, balanced review of issues around online safety, responsibility, privacy and security.

Further resources

Blog platforms for education. Available from http://kidblog.org/home/, http://creativeblogs.net/ and http://primaryblogger.co.uk/

Davies, J. and Merchant, G. (2009) *Web 2.0 for schools. Learning and social participation.* New York, NY: Peter Lang Publishing.

eTwinning (n.d.) Connect with classes across Europe. Available from *www.eTwinning.net*

GitHub Education (n.d.) Available from https://education.github.com/

Richardson, W. (2010) *Blogs, wikis, podcasts, and other powerful web tools for classrooms.* Thousand Oaks, CA: Corwin Press.

Wikipedia (n.d.) *Five pillars* (the guiding principles behind Wikipedia). Available from http://en.wikipedia.org/wiki/Wikipedia:Five_pillars

Wikispaces (n.d.) *Wikispaces classroom* (creating wikis in school). Available from www.wikispaces.com/content/classroom

References

Hattie, J. (2008) *Visible learning: A synthesis of over 800 meta-analyses relating to achievement.* Abingdon: Routledge.

Higgins, S., Xiao, Z. and Katsipataki, M. (2012) *The impact of digital technology on learning: A summary for the education endowment foundation.* Durham, UK: Education Endowment Foundation and Durham University.

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

Jonassen, D.H. (2008) *Meaningful learning with technology.* Harlow: Pearson Education.

Papert, S. and Harel, I. (1991) Situating constructionism. *Constructionism,* 36. 1–11.

Quinlan, O. (2015) *Young digital makers.* London: Nesta.

Richmond, S. (2012) *Jonathan Ive interview: Simplicity isn't simple.* Daily Telegraph, 23 May 2012.

Robinson, K. (2011) *Out of our minds: Learning to be creative.* Hoboken, NJ: John Wiley & Sons.

Sennett, R. (2008) *The craftsman.* New Haven, CT: Yale University Press.

Walker, R. (2003) *The guts of a new machine.* The New York Times, 30/11/2003.

Safe and Responsible Use

How can we best keep young people safe online?

Safe and Responsible Use

HOW CAN WE BEST KEEP YOUNG PEOPLE SAFE ONLINE?

Schools have long had a responsibility to keep pupils safe, and the recommendations from Tanya Byron (2008), Ofsted (2010) and others have emphasised that the best way to do this is through teaching pupils how they can best keep **themselves** safe. This is perhaps akin to cycling: pupils cycling to school are exposed to risks which could otherwise be avoided, but many see the benefits (for independence, health, the environment, the easing of road congestion, and so on) as being worth the additional risk, so we then do all we can to mitigate the risks through teaching pupils to cycle well and safely.

The computing curriculum includes the requirement that pupils are taught to keep themselves safe, and indeed goes beyond just teaching 'online safety', teaching pupils how to act respectfully, responsibly and securely when using technology, to know what constitutes inappropriate content, contact or conduct, and how to report concerns that they may have.

Including these requirements in the computing programmes of study does not mean that these should only be taught in computing lessons, or that the computing head of department becomes responsible for these things: good practice is to see these as a whole-school responsibility, and to embed their teaching across the curriculum and the life of the school. Within computing, these matters can be addressed very effectively, emphasising them as you teach other topics. A few online-safety lessons and an assembly for Safer Internet Day seem less effective than an approach in which safe, responsible and secure practices for the use of technology are taught and followed in all aspects of the school's life and work.

Stepping back from the risk-mitigation approach to online safety, seeing the development of the responsible use of technology as just one aspect of values or character education (see, for example, Department for Education [DfE], 2014) may be particularly effective. If, over pupils' time in school, we can help develop a strong sense of moral responsibility and the 'grit' necessary to stand up for doing the right thing, they will leave us far better at coping with the challenges of adult life, and far less likely to fall prey to the more sinister aspects of the internet and other technology.

What are the risks?

In her 2008 report, Safer children in a digital world, clinical psychologist Prof. Tanya Byron (2008) outlined three broad categories of risk to which young people are exposed through their use of digital technology: content, contact and conduct (see Figure 6.1).

	Commercial	Aggressive	Sexual	Values
Content (child as recipient)	Adverts Spam Sponsorship Personal info	Violent/hateful content	Pornographic or unwelcome sexual content	Bias Racism Misleading info or advice
Contact (child as participant)	Tracking Harvesting personal info	Being bullied, harassed or stalked	Meeting strangers Being groomed	Self-harm Unwelcome persuasions
Conduct (child as actor)	Illegal downloading Hacking Gambling Financial scams Terrorism	Bullying or harassing another	Creating and uploading inappropriate material	Providing misleading info/advice

Figure 6.1 (from Byron, 2008)

Content

Young people are naturally curious and, as teachers, we would hope to nurture and develop that innate curiosity, doing what we can to establish a lifelong love of learning in our pupils. However, whilst a previous generation's curiosity might have led them to look up rude words in a dictionary or encyclopaedia, today's young people are far more likely to search the words they overhear on Google or Bing. The loss of innocence through exposure to highly graphic depictions of sex or violence is far too prevalent. Schools must have effective filters and monitoring in place to prevent access to inappropriate or harmful material (DfE, 2016), but this in itself does little to mitigate the risk to young people through access to such material outside of school, including on smartphones.

Both Google and Bing have SafeSearch settings which, whilst not infallible, will do much to prevent pupils accessing particularly inappropriate content via these; these settings can be locked in place, and a number of organisations have developed search engines targeted at children, often through using

a combination of SafeSearch and custom search tools in Google Search. Pupils might consider how algorithms can be designed to filter search results as effectively as they do.[1]

It would be wrong to think of filtering merely in terms of preventing access to inappropriate or harmful sexual content. Schools have a duty to promote fundamental British values and should prevent access to terrorist or extremist material which might lead to pupils' radicalisation (DfE, 2014, 2015).

Just as schools typically receive a filtered internet connection, in which access is blocked to content considered inappropriate, so parents can request filtered internet access at home and on mobile devices: it's worth teachers explaining to parents how to do this, and the reasons why they should. Even with filters in place, young people may still encounter content that concerns them, and establishing a 'no blame' culture, in which they can alert you or their parents to such content, can be helpful. Many schools operate a policy of teaching young people to close the laptop, switching off the monitor or turning over the iPad if ever they find content they know they shouldn't view or are otherwise concerned about.

Byron identified other risks associated with content, including commercialisation (qv Bailey, 2011). When teaching pupils about the internet, and particularly the web, it's worth helping them to become more discerning and critical about the commercial aspects. You could address the prevalence of spam in email, how this can be filtered semi-automatically, as well perhaps as what sort of algorithms might be used in doing so.[2] It's also worth helping pupils to become aware of the use of advertising on the web and how this can be avoided through the use of browser plugins such as AdBlock, and covering the difference between sponsored and other results in search engines. It's important to help pupils become aware of the difference between altruistically-created content (e.g. Wikipedia, many blogs and much of YouTube), and content created with a perhaps hidden or implicit commercial purpose, noting the absence of a 'free lunch' in many apparently free online services.

Contact

Much good work has been done to teach young people about the dangers of posting personal information online, and of contact via the internet from those they don't already know. The Child Exploitation and Online Protection Centre (CEOP) makes some excellent resources available to support teachers in effectively delivering a clear message to pupils about these risks, and what they can do to minimise them.[3]

Teachers and parents can do much to help young people become more discerning in their use of the internet or other communication technologies, thinking carefully about strange or otherwise unanticipated contact or communication, and the potential long-term consequence of sharing information online.

'Sexting', the sharing of sexually explicit text, images or video via smartphones, perhaps using apps such as Snapchat, seems increasingly prevalent amongst some groups of young people. This is in part due to peer pressure, but also short-term perspectives and some misconceptions about online privacy. There can be profound consequences for both the sender and recipient of the content. The advice from Childline, if someone keeps asking a young person for inappropriate photos, is:

- 'Ask them to stop … or just don't reply at all and hopefully they will get the hint. But if they are still bothering you or making you feel upset it's okay to block them – even if it's just for a bit'.
- 'If an adult has been making you feel uncomfortable by asking you to send them images, you can report them on the CEOP site. If an adult does this it is sometimes called online grooming'.
- 'It is wrong for anyone to be pressuring you in this way. If you are under 18, they are breaking the law'.[4]

Whilst acknowledging that it's illegal for a person under 18 to take or share indecent images of themselves, current ACPO (Association of Chief Police Officers) advice does not support

1 www.youtube.com/watch?v=H2EIN24r-3M, qv Edelman (2003).
2 See https://gmail.googleblog.com/2007/10/how-our-spam-filter-works.html and www.wired.com/2015/07/google-says-ai-catches-99-9-percent-gmail-spam/
3 www.ceop.police.uk/
4 www.childline.org.uk/explore/onlinesafety/Pages/Sexting.aspx

prosecution, and emphasises the need to put safeguarding at the heart of any intervention.[5]

As with any safeguarding issue in school, teachers have a responsibility to report concerns to the designated person, in accordance with school policies.

Conduct

The curriculum requires that pupils understand how to use technology responsibly and respectfully. Supporting young people's moral development is a vital part of secondary education, a statutory requirement for a school's curriculum and, as part of 'spiritual, moral, social and cultural development', an element of all Ofsted inspections. Kohlberg's stages of moral development (for example, Kohlberg, 1984) offers one model for thinking about this:

1. obedience and punishment orientation (how can I avoid punishment?);
2. self-interest orientation (what's in it for me?);
3. interpersonal accord and conformity (the good boy/girl attitude);
4. authority and social-order-maintaining orientation (law and order morality);
5. social contract orientation (Do unto others…);
6. universal ethical principles (principled conscience).

Under this model, we would hope to see pupils already taking responsibility for their own moral and ethical decisions and behaviour when they get to secondary school. We then support them as they learn to do the right thing out of a respect for others and, ultimately, on the basis of their personal adoption of universal ethical principles, probably including such 'fundamental British values' as 'democracy, the rule of law, individual liberty and mutual respect and tolerance of those with different faiths and beliefs'. If schools take seriously moral education, focussing on character and values, seriously, many aspects of pupils' inappropriate conduct using technology can perhaps be avoided, or their consequences reduced.

In many schools, **cyberbullying** is a common problem: a BBC/Comres survey reported 22 percent of 10–12-year-olds had experienced bullying or 'trolling'.[6] Whilst this is more likely to happen outside of school, it's common for both bully and victim to be members of the same class, year group or school, and the cause and consequences may often be connected to school. As with bullying in general, a focus on moral education might reduce the prevalence of such hurtful behaviour in the school community, but a clear zero tolerance message is essential, together with a culture in which this can be reported, safe in the knowledge that swift and effective action will follow. Alongside this, it's worth building up young people's resilience to off-hand, unintentionally hurtful remarks from others, recognising that not every online disagreement or critical comment constitutes bullying.

Copyright, and other aspects of intellectual property, is another area in which young people's (and sometimes teachers') conduct isn't all that it could be. Perhaps because the web works through automatically making copies of the content from a distant web server in the user's web browser when a page is accessed – and the ease with which digital content can be perfectly copied – it's all too easy to assume that content found online can be used wherever and however someone wants, without paying attention to the legal and ethical aspects of intellectual property. There are generous exemptions from much copyright legislation for clearly specified educational use,[7] as well as educational-use licences for a range a media purchased centrally by the DfE on behalf of state-funded schools in England.[8] However, it remains important to teach and show best practice in the use of copyrighted material, including properly acknowledging the source of content and respecting any associated licence terms.

The Creative Commons family of licences makes it easy for those who create work in any medium to license it for reuse, under a range of different conditions: you can teach pupils about this approach to sharing online, and show them how they can search for, acknowledge and reuse Creative Commons licensed content in their own work.

5 https://ceop.police.uk/Documents/ceopdocs/externaldocs/ACPO_Lead_position_on_Self_Taken_Images.pdf
6 www.bbc.co.uk/news/education-35524429
7 www.gov.uk/government/uploads/system/uploads/attachment_data/file/375951/Education_and_Teaching.pdf
8 www.copyrightandschools.org/

Both Google and Bing image searches allow results to be filtered to show just images that have been licensed in this way.

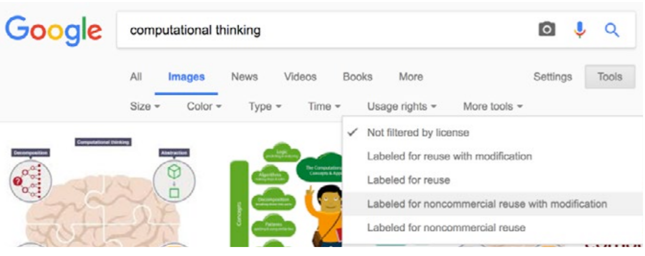


Figure 6.2 Screenshot to show filtering by licence in Google image search

Furthermore, the work uploaded to the Scratch website is covered by a Creative Commons by-share-alike licence, as are resources shared on the CAS (Computing At School) community site, except where stated otherwise. There's ample scope in the curriculum for pupils to make use of Creative Commons and public-domain[9] content: the Level 2 Benchmarks state that pupils 'Demonstrate an understanding of usage rights and can apply these within a search'.

It's worth bearing in mind that pupils automatically own the copyright in their own work, including that which they produce in school, and that we as teachers should respect this, for example checking with pupils and their parents before publishing their work online in a class or school blog. Asking parents to license their children's work for these and other uses by the school might seem unnecessarily legalistic, but it's important that pupils learn about, and have respect shown for, their rights as well as their responsibilities.

It's important that pupils be taught to respect the **terms and conditions** of any websites or other online services which they use, and indeed see you doing so yourself. The terms and conditions of most online services run to many, many pages, but when signing up for new services, or asking pupils to do so, it's well worth checking through the sections on any age restrictions as well as through those on copyright and data privacy. US-based companies are required to abide by American COPPA (Children's Online Privacy Protection Act) legislation,[10] which prevents their storing personal data on under-13s without parental consent. As a result, many US-based internet services and websites (including Facebook) prohibit under-13s from creating accounts or using the service. Pupils under 13 using these services would be doing so without the operator's permission, which could be considered an offence under the Computer Misuse Act.

A number of services, including Google Apps for Education and Office 365, allow schools to create accounts on behalf of children, with the school taking responsibility for obtaining the necessary parental consent. Other websites, such as Scratch, allow teachers to create multiple accounts in their own name and share these with pupils, but this is an exception rather than the rule: it's much better to check, and abide by, the terms and conditions, rather than making these assumptions.

Concern is sometimes expressed that young people might use their knowledge of programming and computer networks for harmful or illegal purposes, including **cybercrime**. Even in the context of school networks, it's not unheard of for pupils to attempt to obtain administrator or teacher password details, bypass filtering through proxy servers or VPNs (virtual private networks), or attempt to install keyloggers or password sniffers: all of these are likely to be prohibited under the school's acceptable-use policy. Pupils investigating the tools and techniques involved here may get drawn in further to a subculture in which circumventing computer security is seen as an acceptable intellectual challenge.

The Computer Misuse Act[11] was introduced to make hacking computer systems illegal. It covers a number of offences involving the unauthorised use of computers, with or without the intention of committing further crimes, or impairing the operation of computers. As well as account hacking, using malware (including rootkits) is against the law, as is conducting distributed denial-of-service attacks. The National Crime Agency suggests a number of warning signs that parents might watch out for, to alert them to a young person's involvement in cyber-crime:

- Are they resistant when asked what they do online?
- Do they get an income from their online activities, do you know why and how?

9 That is, content which can be used without any restriction, sometimes called Creative Commons
10 http://bit.ly/ftccoppa
11 www.legislation.gov.uk/ukpga/1990/18

- Is your child spending all of their time online?
- Do they have irregular sleeping patterns?
- Have they become more socially isolated?

Whilst parents of many teenagers not involved in cybercrime might answer yes to many or all of these, it's important that parents and teachers feel able to discuss any concerns or suspicions they have over an individual's interest in or involvement with criminal activities. A clear emphasis on character or values education in school, including in computing lessons, might do much to help prevent young people becoming involved in computer-based crime.

Finally, be aware, and make your pupils aware, of **the opportunity cost** associated with screen time – time spent using a computer, tablet or smartphone is time not spent doing other things, such as reading a (paper-based) book, learning a musical instrument, playing in a team and socialising face-to-face with family and friends. Whilst digital technology is seen by many as transformative of so many aspects of learning and life, many would count it a great shame if it came to dominate childhood, within or beyond school, to any greater extent than it already has. Helping young people to become more discerning users of technology, knowing when it would be useful, and when it might be more of a distraction, is perhaps also one of our responsibilities as teachers.

Reporting concerns

The new curriculum requires that pupils are taught how to report concerns they have over technology. In most cases, pupils should talk to their parents or their teachers about such concerns: if pupils report such concerns to you, this may be covered by your safeguarding policy, so make sure you follow this very carefully. Sometimes, pupils might be too embarrassed about something to turn to either you or their parents, so it's worth making them aware that there are others whom can they talk to, including Childline and CEOP.

Pupils need not be directly affected by something to report it: it's important to establish a classroom and school culture in which pupils feel that they can discuss any concerns openly with teachers and one another, including if they believe any of their friends are involved in risky online activity, or if they notice unusual changes in a friend's behaviour. A number of schools have established a 'digital leader' role for pupils,[12] which might include particular responsibilities around supporting other pupils in staying safe online.

Older pupils should also know how they can address concerns they have, particularly over content or conduct, with social media sites directly.[13] And reputable sites, including Facebook, Twitter, Tumblr, Instagram and YouTube, will act promptly in the case of illegal activity or where the site's terms and conditions have been infringed.

Classroom activity ideas

- Online safety and responsibility provide great topics in which pupils can develop their creative use of technology. From making and giving high-quality presentations and blogging advice for younger pupils, to creating short live-action video[14] or animations exploring these issues, there's ample scope here for pupils to collaborate and produce high-quality work across a range of digital media.
- Pupils might work together to develop a website summarising the terms and conditions, and the reporting arrangements, for popular social media platforms and apps.
- Keep abreast of media stories concerning illegal or unethical use of technologies, using these as starting points for class discussions. Encourage pupils to use a framework of ethical principles when discussing these.
- Online content, conduct and contact issues might be effectively explored through role-play and discussion, although you should be aware that some pupils might be directly affected by some of these issues, and thus a particularly sensitive approach may be necessary. Dilemma cards might be particularly useful.

12 www.childnet.com/new-for-schools/childnet-digital-leaders-programme
13 See www.thinkuknow.co.uk/14_plus/help/Contact-social-sites/ for links to policies and reporting portals for many social media providers.
14 Pupils might enter their film for Childnet's annual competition: www.childnet.com/resources/film-competition/

Further resources

Computing At School (n.d.) *Computing starters: Social, ethical, economic and philosophical issues.* Available from http://community.computingatschool.org.uk/resources/2212

Childnet (n.d.) Secondary hub. Available from www.childnet.com/young-people/secondary; see also advice for teachers: www.childnet.com/teachers-and-professionals

Creative Commons (n.d.) For information and free licences to use. Available from http://creativecommons.org/

Cutts, M. (2011). *Google's Matt Cutts on how Google Safe Search works.* YouTube. Available from www.youtube.com/watch?v=H2EIN24r-3M

Dredge, S. (2014) *How do I keep my children safe online? What the security experts tell their kids.* The Guardian. Available from www.theguardian.com/technology/2014/aug/11/how-to-keep-kids-safe-online-children-advice

Intellectual Property Office (2014) *Exceptions to copyright: Education and teaching.* Available from www.gov.uk/government/uploads/system/uploads/attachment_data/file/375951/Education_and_Teaching.pdf

Kohlberg, L. (1981) *The philosophy of moral development: Moral stages and the idea of justice (Essays on moral development, volume 1).* San Francisco: Harper & Row.

Ofsted (2015) *Ofsted on inspecting safeguarding.* Available from www.gov.uk/government/publications/inspecting-safeguarding-in-early-years-education-and-skills-from-september-2015

South West Grid for Learning (n.d.) *Digital literacy & citizenship curriculum: Teaching resources.* Available from www.digital-literacy.org.uk/Home.aspx

Thinkuknow.co.uk (CEOP) (n.d.) Resources for lower secondary pupils. Available from www.thinkuknow.co.uk/11_13/; and for teachers: www.thinkuknow.co.uk/Teachers/

UK Safer Internet Centre (n.d.) Available from www.saferinternet.org.uk/

UNICEF (n.d.) *Children's rights: United Nations Convention on the Rights of the Child.* Available from www.unicef.org.uk/Documents/Publication-pdfs/UNCRC_PRESS200910web.pdf

Privacy, Security and Identity

Online safety is linked directly with issues of privacy, security and identity, and these topics lend themselves to further exploration within computing lessons. Privacy and security are closely related ideas but are not synonymous: we put curtains at our windows to protect our privacy but fit locks to our doors to maintain security.

Privacy

Pupils at Level 1 are taught that they should keep personal information private. At Level 3 they learn how to use technology to protect their privacy.

Pupils ought to have a good idea of what's meant by personal information. The Data Protection Act defines personal data as data about a person who can be identified from the data.(15) This includes names, home addresses, personal phone numbers and email addresses, as well as photographs or videos showing the face of a person, but it might also be reasonably seen as including the Internet Protocol (IP) or Media Access Control (MAC) addresses of connections or computers used by the person, or details of their social media accounts.

There's other information which a person might reasonably expect to be kept private, such as their internet history, their search history, mobile cell or Global Positioning System (GPS) location information, personal photographs, and records of mobile phone calls or email correspondence.

Keeping all this information entirely private is, to all intents, incompatible with the use of online technology – when using a standard mobile phone, the network operator must know the cell to which

15 https://ico.org.uk/for-organisations/guide-to-data-protection/key-definitions/

any call should be routed; when communicating by email, the email provider of sender and recipient must have access to the email and its contents; search providers must know what it is that you are searching for if they are to provide results; web servers automatically maintain records of pages requested and the IP address of the computer requesting them: such information **has** to be provided through the very nature of the technology used. Similarly, without encryption there's nothing to prevent routers, gateways and switches sniffing the contents of the packets transmitted through them across the internet's infrastructure. Users particularly concerned about privacy issues might set up their own virtual private network,[16] use an anonymising routing protocol such as TOR,[17] set up their own server and domain for email and other services,[18] and avoid social media: for users outside of oppressive regimes, such steps might suggest an excessive degree of paranoia or that they have something to hide.

Rather than such an entirely paranoid approach, it's worth getting pupils to think in terms of circles of trust, thinking carefully about with whom they would choose to share information. There are some people whom a pupil should trust to a very great extent, secure in the knowledge that that person has the pupil's own best interests at heart: one would hope that for almost all children this would include their parents and their teachers. With those in this circle of trust, pupils might confidently share almost any information. Close friends and relatives might be trusted somewhat less, but we would be predisposed to consider them worthy of our trust in most matters. There's then a looser circle of friends and acquaintances, whom we are perhaps somewhat more wary of, but in whom we are still willing to invest some trust to strengthen such relationships – although we might exercise some degree of caution in doing so.

In a benign, liberal democracy, and in an education system which places its pupils' well-being as its number one concern, we might also consider the police, other government agencies and those maintaining the security of the school's information systems as meriting a high degree of trust, although not all would necessarily agree.

Many of us feel confident placing our trust in large, multinational corporations, not because we believe they act in our best interests, but because what they provide in return for what they ask seems a good deal, and many such organisations have stated policies in which they seem to take their customers' privacy very seriously, even if it's on their own terms rather than ours.

Those whom we don't know are another matter. Back in Jane Austen's day, new acquaintances would only be accepted once they had been introduced, either by letter or in person, by someone already known and trusted, and perhaps something similar operates at a lower level of trust on the web and through social media.

It seems clear that some degree of caution and discernment is needed when navigating the complex web of personal, commercial and regulatory relationships that our online life makes us part of. To share nothing of oneself denies the opportunity for participation, but to share everything seems foolhardy. A middle path seems the appropriate way to strike a balance between these extremes, sharing some things more generally whilst keeping other things to a closer circle.

In sharing information online, pupils should be aware of the long-term persistence of the information they share – what goes online typically stays online. Once a photograph, video or message is available on the open web, then anyone with access to it may be able to make a copy of it – indeed, the very act of viewing the content involves transferring that content from a web server to a computer. Google's Page Rank algorithm can only operate through indexing a cached copy of the web. The Internet Archive similarly makes regular copies of many websites to preserve this content for the future. Even within password-protected sites, it's essentially impossible to prevent what other users with access to content will do with that content. A photograph might be shared in the expectation that it would remain private, but the other person might still, in a breach of trust, share a copy of that with others or on the open web. That which a pupil thinks is worth sharing at the age of 14 might subsequently be regretted when it remains permanently associated.

16 For example using a Raspberry Pi: www.bbc.co.uk/news/technology-33548728
17 www.torproject.org/
18 Perhaps using https://owncloud.org/ or https://sandstorm.io/

As well as deciding for themselves with whom to share particular information, based on the extent to which they trust the other, pupils should be aware of the routine and almost inevitable recording or surveillance of their online activities. As mentioned above, schools now have a duty to monitor pupils' access to the internet; internet service providers maintain records of sites visited; mobile phone companies maintain records of the cell masts to which a mobile phone automatically transmits its location; search-engine providers build detailed profiles of users based on their search queries and other activities; social media sites and app providers similarly know much about whom any one of us is friends with or follows.

With the Investigatory Powers Act[19] becoming law, much of this information has to be disclosed to investigators in certain defined circumstances. Many would argue that such recording or surveillance is a not an unreasonable price to pay for the online services provided, and to ensure that individuals and society are protected from those who would wish them harm.

Pupils should have a reasonable expectation that any data held about them is kept private, as the Data Protection Act[20] requires; that personal information is not, without one of a few very good reasons, or the subject's explicit permission, shared with third parties; and that, when others are, of necessity, involved in processing data, they too have an obligation to protect the privacy of those to whom it relates. This expectation applies to schools as much as to any other organisation processing personal data, and including that data be kept securely in such a way that unauthorised users cannot access it.

Security

Security is about preventing those who shouldn't have access to data, information or systems from gaining such access.

Security is related to privacy – in general it's a necessary but not sufficient condition: that is, you cannot expect privacy without security, but privacy needs more than just security.

At one level, information can be kept secure by physical means – recording information on paper only, and storing that information in a safe or locked filing cabinet, would foil all but the most intrepid.[21] On a computer, an 'air gapped' machine[22], without network access and without support for removable media, would provide a high degree of security for any data stored on it, assuming that the physical security of the system itself could be guaranteed – although it would be a far-from-convenient system to use for most practical tasks.

Beyond the physical security of a system, some attention should be given to the security of the data stored on it, or on removable media used with it – whilst challenge-and-response passwords provide a degree of protection, encrypting the data is the best way to ensure that, even if unauthorised users were to gain access to the system or find the memory stick, it would be essentially impossible for them to read the data stored on the device without knowing the secret key with which the data had been encrypted. These days, most operating systems include the ability to encrypt all of the data on the startup disk or equivalent system, and communication via the internet can be routinely encrypted without any additional efforts. Smartphones can be set to delete any data stored on them if a wrong passcode is entered more than a set number of times, and can be wiped remotely when connected to the internet.

19 www.gov.uk/government/collections/investigatory-powers-bill
20 www.gov.uk/data-protection/the-data-protection-act
21 See, for example, www.theguardian.com/world/2014/jul/15/germany-typewriters-espionage-nsa-spying-surveillance
22 https://en.wikipedia.org/wiki/Air_gap_(networking)

Cryptography

Cryptography is central to an understanding of the security of digital data, and particularly to its communication via the internet, over what are essentially insecure, open channels. The encryption techniques used to transmit messages securely can also be used to store those messages securely.

In classical cryptography, we take a plain text message to be encrypted, some agreed protocol for encrypting the message and, crucially, a secret key that's used to encrypt the message into some 'ciphertext'. The idea is that, even if the enemy has access to the ciphertext and full knowledge of the protocol used, they cannot recover the original plain text without knowing (or guessing) the encryption key used.

Figure 6.3 Image of Caesar cipher wheel: this from https://commons.wikimedia.org/wiki/File:CipherDisk2000.jpg

The history of cryptography long and interesting. One of the earliest cryptographic systems was the Caesar cipher (Figure 6.3), in which the letters in the plain text message were simply shifted along the alphabet by an agreed number of places – thus the plain text

attack at dawn

would become the ciphertext

BUUBDL BU EBXO

Decrypting the message is simply the reverse of this process, shifting each letter of the ciphertext back along the alphabet the correct number of places.

Whilst this is an easy system to implement by hand and to code on a computer, it's very far from secure: to break the encryption, the enemy only needs to try the 25 possible shifts until something in English turns up.

A more sophisticated system might involve replacing each letter in the plain text alphabet with another, agreed letter. For example, we might swap

a b c d e f g h i j k l m n o p q r s t u v w x y z

with

E G L J H O T U P V F S X A W Q K R M Z D Y N C I B

Using this key, the message

attack at dawn

would become the ciphertext

EZZELF EZ JENA

With knowledge of the key, this is reasonably easy to decipher by hand, and easy enough to program on a computer (Figure 6.4).

```
encode from plain : to cipher : message: message
script variables cipher text letter
set cipher text to
for i = 1 to length of plain
  set letter to letter i of message
  if plain contains letter
    set cipher text to
      join cipher text item position of letter in plain of cipher
  else
    set cipher text to join cipher text
report cipher text
```

Figure 6.4

```
def encode(message, cipher,
plain=”abcdefghijklmnopqrstuvwxyz”):
    ciphertext = “”
    for i in range(len(message)):
        letter = message[i]
        position = plain.find(letter)
```

```
        if position > -1:
            ciphertext = ciphertext + cipher[position
        else:
            ciphertext = ciphertext + " "
    return ciphertext
```

Simple substitution functions in Snap! and Python

At first sight, it looks very hard to break by trying out all the possible key combinations (of which there are 26 x 25 x 24 x … x 2 x 1 possible keys, roughly 4 x 10^{26}), but such brute-force approaches are only ever the last resort of cryptanalysts or hackers. If the language of the original message is known or can be guessed, and the message is long enough for statistical techniques to work, then the enemy can use the relative frequency of letters (and pairs of letters) in the original language to start making plausible guesses at which letter had been swapped for which (see the discussion of Shannon and information entropy on pages 96 - 98). For example, E and T are the most common letters in English, T is more likely to be followed by H than any other letter, and so on. Unsurprisingly, it is possible to automate or at least semi-automate this frequency-analysis approach to breaking a simple substitution cipher. Automated or semi-automated breaking of codes has figured prominently in the history of computing, most notably in the work of Turing, Flowers, Welchman and others at Bletchley Park in the Second World War.

A more sophisticated system still is the Vigenère or polyalphabetic cipher in which different Caesar shift covers are applied to different letters of the plain text, according to some predefined system. For example, one could take the letters of a key word and use these to determine different Caesar shifts: the key word 'FAB' could suggest shifts of 6, 1 and 3 positions repeatedly, or a longer text, such as an agreed passage from a book could be used. This is far less amenable to frequency analysis.

```
def vigenere(message, key):
    ciphertext = ""
    for i in range(len(message)):
        letter = message[i]
        newletter = (ord(letter) - 96 + ord(key[i %
len(key)]) - 96) % 26
        ciphertext = ciphertext + chr(newletter +
64)
     return ciphertext
```

Simple implementation of a Viginere or polyalphabetic cipher in Python

Such a system can form the basis of an unbreakable code – a 'one-time pad' (Shannon, 1949) in which Caesar shifts or their equivalents are applied according to a genuinely random stream of values known only to sender and recipient in advance of communication. If there's **no** pattern to the shifts that are applied, and if the random stream is known only to sender and recipient, then even a brute-force attack cannot recover the original plain text, as all information about the plain text is hidden within the randomness of the key. The downside is that the key must be the same length as the plain text and cannot be used again – hence the 'one-time' name for this method. Furthermore, the security of the communication now becomes about 'marinating' the security of the one-time pad, which is at least as hard as marinating the security of the message it was designed to protect.

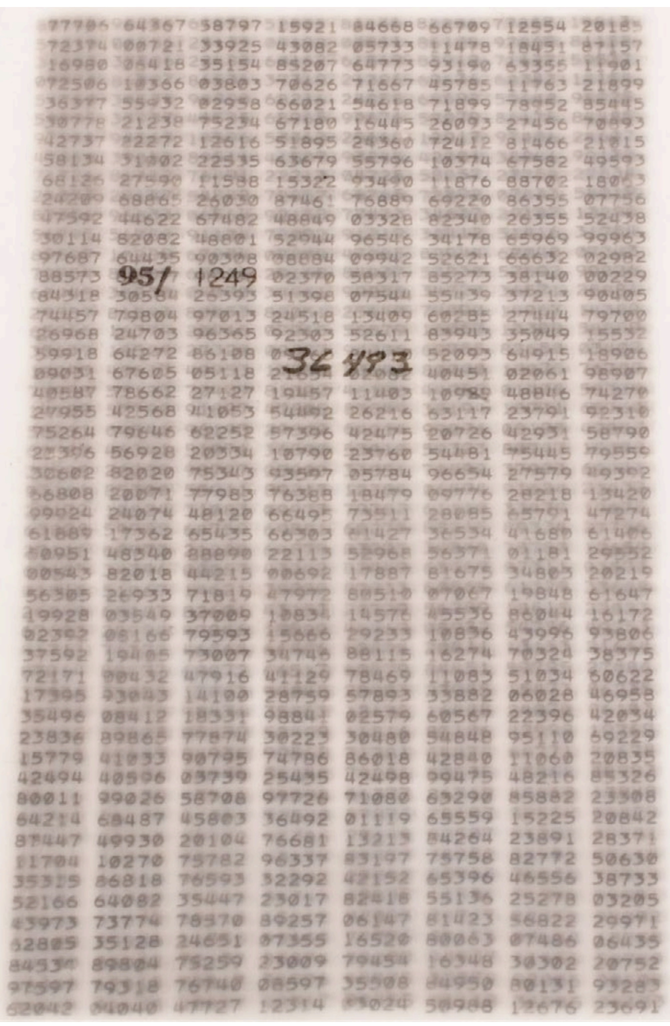


Figure 6.5 One-time pad

PD: https://upload.wikimedia.org/wikipedia/commons/thumb/6/62/One-Time_Pads_-_Flickr_-_The_Central_Intelligence_Agency.jpg/399px-One-Time_Pads_-_Flickr_-_The_Central_Intelligence_Agency.jpg

Whilst the one-time pad is secure, it is impractical for general-purpose communication on the internet, due to the key exchange problem. If Alice and Bob are to communicate securely, they need to agree in advance the key they will use, and this would need to be communicated securely – and if Alice and Bob can do this already, they have no need to establish a new secure-communication channel…

The breakthrough to the key exchange problem came with finding mathematical functions which are easy to perform but very hard to reverse. For example, if I take two very large prime numbers, it's easy to multiply them together; however, given the product of two very large prime numbers, it's very hard, even for a very fast computer, to reverse the factorisation. Without the aid of a calculator, you should be able to multiply 7,919 and 8,863, but even with a calculator it would take a long time for you to find the factors of 62,080,727. Factoring this number is relatively trivial, but finding the factor of a number some 620 digits in length seems beyond the reach of even the fastest computers, for some time to come.

There's a little more to Diffie-Hellman (see Diffie and Hellman, 1976; and also Merkle, 1978) key exchange than simply multiplying large prime numbers together, but the idea here is that it allows two people wishing to communicate to decide a secret key that could be used for encryption using another cryptographic system, in such a way that anyone eavesdropping on the communication about the key couldn't work out the key, because it relies on secret information that is never shared. Thus, when communicating over the internet, my browser and the server it is communicating with can, across an open channel, establish a secret key known only to them, and then use this for subsequent encrypted communication – without the key itself ever being transmitted, and in such a way that no third party can guess that key.

Another approach to the key exchange problem, again based on the difficulty of factoring a product of primes compared to the ease with which they can be multiplied, is the RSA public/private key algorithm.[23] Here, two different keys are needed for the cryptographic system: one which allows data to be encrypted, the public key, and another which allows a message to be decrypted, the private key. If I wish to encourage folks to communicate with me securely, I can publish my public key on the web and invite anyone to use it. Folks can then send me encrypted messages using this key, but even with access to the public key, no eavesdropper can reverse the encryption process to read the message, as this is only possible using the associated private key, which only I have access to. RSA offers another advantage in that it allows me to cryptographically 'sign' messages – I could, for example, encrypt an outbound message using my private key; anyone reading this could then decrypt it using my public key, and the fact that it can be decrypted using my public key proves that it was originally encrypted (that is, signed) using the associated private key that only I hold.

The HTTPS protocol, used for secure communication for web traffic over the internet, builds on these ideas. HTTPS does two things: it establishes that the computer you are talking to is the one you think you are talking to, by presenting (and checking) a signed cryptographic certificate; and it sets up secure communication for subsequent communication between you and the far web server. Thus, when I visit my bank's home page over HTTPS, my browser checks that the site carries a cryptographic certificate signed by someone I already trust (one of a number of supposed incorruptible certificate authorities accepted by my browser); assuming all is well, key exchange for the session takes place and all the following communication is done in secret, from my password, through my statement, to setting up new payments - even though this is over the insecure channels of the internet.

HTTPS has vulnerabilities, but it's not vulnerable to folk in the same coffee shop sniffing otherwise open or poorly-encrypted WiFi traffic. The vulnerability lies in the acceptance of certificate authorities (Callegati et al., 2009) – on a school network, you may be expected to accept, to trust, a security certificate from the school for internet access. Once you've done so, it's possible for the school's gateway computer or router to sit between you and distant web servers (a man-in-the-middle attack) routinely decrypting and re-encrypting any traffic between you whilst assuring you that all is very well, as you've trusted it to sign and to encrypt traffic on your behalf. Assuming HTTPS access is permitted, it could be possible for pupils to use HTTPS to access web-based content in school, without the school being able to monitor or filter what they were accessing.

23 www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-rsa-cryptography-standard.htm

Beyond encryption of data and communication, it's worth considering some other elements of security.

As well as ensuring that unauthorised users cannot access data, it's important for the security of data that those who should have access do have access, even if disaster strikes. In part, this involves implementing a robust approach to backing-up and archiving data. In the case of data stored locally on hard drives or removable media, a sensible approach would be to make copies of this at regular intervals, wherever possible ensuring this happens automatically, thus protecting against hardware failure of a drive or memory stick, and, in some cases, against operator error too. It's wise to store one copy of the data in another location, to protect against fire or theft. If the capacity and availability of back-up media isn't an issue, then an incremental back-up policy is wise, keeping older copies of data as well as recording subsequent changes, thus allowing older versions of files to be retrieved if subsequent changes harm their integrity.

In the case of data stored 'in the cloud' on remote servers, it's important to establish whose responsibility it is to back-up data. Google, Microsoft and other reputable providers will have robust and well-tested back-up strategies in place, and you might consider using a service like this as an additional back-up for locally-stored files. For sensitive data, you should consider encrypting such files locally before storing them remotely.

It's particularly important to protect the integrity of any admin or root accounts on a computer or, particularly, a server or domain. Such accounts should not normally be used when operating the computer at a user level, as it's possible for accounts with these privileges to make far-reaching and long-lasting changes to other users' data and the system itself. If installing software using these accounts, you should take particular care that any programs installed come from a trusted source.

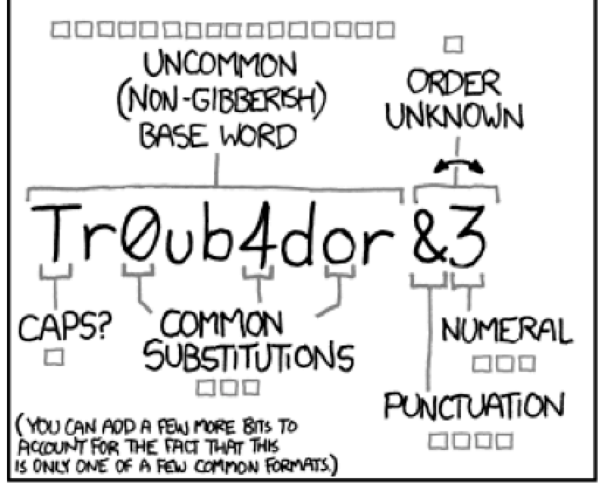
Identity

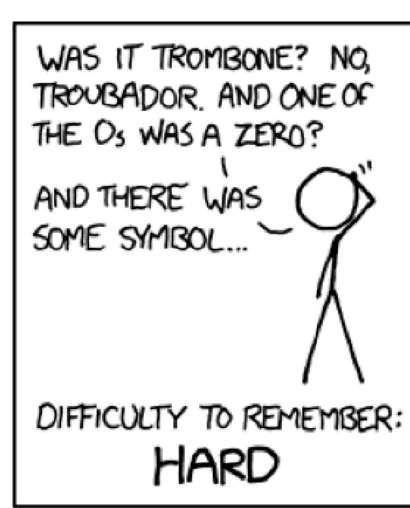
Pupils construct their identity in many ways, often presenting different persona in different contexts, behaving quite differently at school, at home and when out with their friends. Online identity can make this harder, as multiple accounts across different services are often linked, either explicitly or implicitly. Many social media sites, most notably Facebook, prohibit multiple accounts and require that one registers using a real name. Furthermore, open platforms and an eagerness to share content can make it hard for young people to maintain different persona in different online contexts. Both the deliberately shared and the automatically recorded aspects of a digital footprint become inextricably linked to a user's online identity. Young people, and their teachers, can do much to promote their best selves to the world online: and some recognition of the persistence of online data and of their responsibilities might encourage them to be so. This is less about using the web to experiment with the cyberculture (Turkle, 1995) of the past, as it is about presenting an authentic picture of one's best self.

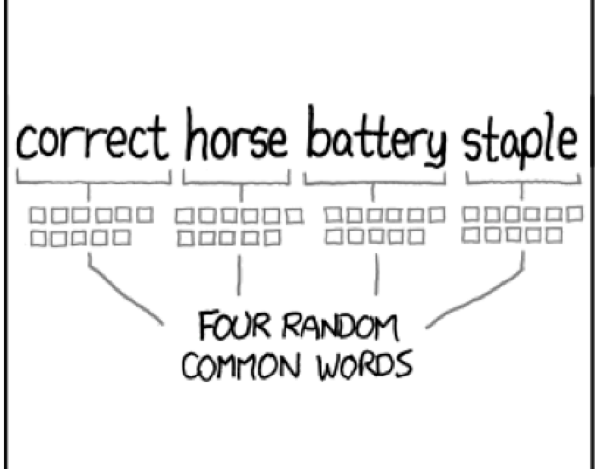
As more and more aspects of pupils' learning and life are mediated through online systems, it's important that they learn to protect their own online identity and respect the online identity of others.

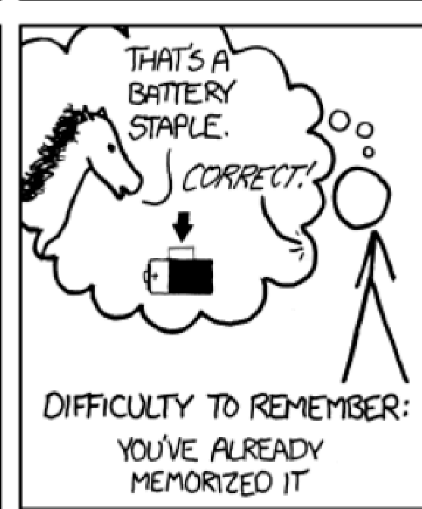
Typically, online identity is established through some form of password system. Pupils should treat passwords as they do toothbrushes: only use their own, and change them regularly! Encourage pupils to use long passwords that cannot easily be guessed (for example CorrectHorseBatteryStaple,[24] or in accordance with the rules enforced on the systems they access), to use different passwords for different sites or services, and to change passwords regularly.

24 https://xkcd.com/936/









THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Figure 6.6 From XKCD at https://xkcd.com/936/, licensed CC by-nc

Discourage pupils from sharing passwords with one another, as this is usually their only way to prove who they are in any online system, and avoid encouraging pupils to share their passwords with their parents: many difficulties could arise through one parent impersonating their son or daughter in an otherwise secure 'walled garden' environment such as a school VLE or learning platform. Remind pupils that they should log off when they have finished using a computer or website, and that they should only allow browsers to maintain logged-in status if they are the only person with access to that browser. Similarly, they should only ever use the 'remember my password' feature in a web browser if they are certain that they are the only person who will use that browser. Pupils should consider which accounts they care particularly about and ensure that passwords for these are unique and particularly secure, for example a main, personal email account should always have a unique password since it would be the principal means of requesting a password reset for other online accounts.

Encourage pupils to consider the security of their passwords on the server they are connecting to. There have been well-publicised stories of user-account databases being hacked and passwords sold on, which is particularly worrying for users who happen to have used the same password across multiple sites. Passwords should never be transmitted in an unencrypted form. A secure password system must never store the password itself in an unencrypted form, and it should never be possible for those who maintain the system to recover the original password: distrust (and stop using) any system which, when you ask for a password reminder, is able to tell you what password you set.

When setting a password online, the minimum acceptable approach is for the browser to transmit the password via an encrypted connection, then for the server to cryptographically 'hash' the password and store this hash in the database. To check the password, the user transmits their password over an encrypted connection; this is again hashed by the server and the hash compared to the stored version. If they match, the user can access the system. This system too is vulnerable if the password table is available to an attacker, as possible passwords can simply be hashed and compared to the hashed value in the table. By adding some predetermined, unique, random 'salt' to the password before it's hashed, the risk here can be mitigated.[25]

For particularly sensitive accounts, consider some form of two-factor authentication. User authentication can be based on who a person is, what they know or what they have. Usual password protocols rely on one factor only: what a person knows. Simple swipe cards, or biometric systems sometimes used for registration, catering or library access, rely on what a person has or who a person is. Two-factor systems require two distinct elements before allowing access. For example, an ATM uses a relatively insecure four-digit PIN because it only provides account access if the user has the corresponding card. Online systems which send authentication codes to mobile phones, or require codes to be generated on a previously authenticated app or device, provide a similar degree of security – even if an attacker guesses the account password, they cannot get access without also having the associated phone. If the phone used can only be unlocked with a fingerprint or biometric, then this could be considered a three-factor authentication system.

Any such authentication systems are vulnerable if users can be tricked into giving away their passwords, and it's important to teach pupils how to spot phishing or other social engineering attempts. For example, pupils should learn to distrust the links shown in emails or similar messages, as these might be spoof and point to websites other than those suggested. Pupils should never give passwords out in response to questions, whether online, face to face or by phone.

Classroom activity ideas

- Help pupils to consider the extent of their digital footprint, perhaps keeping a diary of the apps they use in one day, or reviewing their own browser or search histories for a day. Take care here as some pupils might be particularly sensitive to the associated issues around privacy and surveillance. What do pupils find if they type their name into Google? Ask pupils to review their privacy settings on any social media sites they use. If they have accounts for services they no longer use, why don't they delete them?
- A combined visit to the birthplace of computing[26] and the home of wartime codebreaking efforts[27] is highly recommended, although places for school visits tend to be booked months in advance.
- Talk through some of the issues of privacy and surveillance with pupils. Do they consider this an appropriate way to keep them and others safe? Is it acceptable to use search or browser history or tracking cookies to better target marketing information? Do they mind automated systems reading their emails?
- Cryptography is rich territory for linking pupils' computational thinking and programming skills to issues of privacy and security. Can pupils write programs to implement simple cryptography systems in Snap! or Python? Can they write programs which can crack or help crack simple encryption? Can they implement a program to securely store salted and hashed passwords and then check passwords against these?

25 See www.owasp.org/index.php/Authentication_Cheat_Sheet for more details about secure password systems.
26 www.tnmoc.org/
27 www.bletchleypark.org.uk/

Further resources

British Library (n.d.) Education resources on *'My Digital Rights'.* Available from www.bl.uk/my-digital-rights/

CESG / CPNI (n.d.) *Password guidance.* Available from www.gov.uk/government/uploads/system/uploads/attachment_data/file/458857/Password_guidance_-_simplifying_your_approach.pdf

CIMT (n.d.) *Codes and ciphers resources.* Available from http://www.cimt.org.uk/resources/codes/ [2/1/17].

Cyber Security Challenge UK (n.d.) Available from https://cybersecuritychallenge.org.uk/

Electronic Frontier Foundation (US) (n.d.) *Resources on student privacy.* Available from www.eff.org/issues/student-privacy/

Open Rights Group (n.d.) Available from www.openrightsgroup.org/

Raspberry Pi Learning Resources (n.d.) *One time pad activity (Python).* Available from www.raspberrypi.org/learning/secret-agent-chat/

Schmidt, E. and Cohen, J. (2013) *The new digital age: Reshaping the future of people, nations and business.* London, Hachette UK.

Schneier, B. (2009) *Schneier on security.* Hoboken, NJ: John Wiley & Sons.

Singh, S. (1999) *The code book: the evolution of secrecy from Mary, Queen of Scots, to quantum cryptography.* London: Harper Collins. See also interactive tools at www.simonsingh.net/The_Black_Chamber/chamberguide.html and *Singh's Science of Secrecy* programme on public key cryptography: www.youtube.com/watch?v=_ZTWLAqYf9c and www.youtube.com/watch?v=oR0_LPbWxe4

References

Bailey, R. (2011) *Letting children be children: Report of an independent review of the commercialisation and sexualisation of childhood* (CM. 8078). London: The Stationery Office.

Byron, T. (2008) *Safer children in a digital world: The report of the Byron Review.* DCFS. Available from http://webarchive.nationalarchives.gov.uk/20130401151715/http://www.education.gov.uk/publications/eOrderingDownload/DCSF-00334-2008.pdf

Callegati, F., Cerroni, W. and Ramilli, M. (2009) *Man-in-the-Middle attack to the HTTPS protocol.* IEEE Security and Privacy Magazine 7(1):78 - 81.

DfE (2014) *Promoting fundamental British values through SMSC.*

DfE (2015) *Keeping children safe in education.*

DfE (2016) *Draft: Keeping children safe in education.* London: DfE.

Diffie, W. and Hellman, M. (1976) *New directions in cryptography (PDF). IEEE Transactions on Information Theory* 22 (6). 644–654.

Edelman, B. (2003) *Empirical analysis of Google SafeSearch.* Berkman Center for Internet & Society, Harvard Law School, Available at http://cyber.harvard.edu/archived_content/people/edelman/google-safesearch/ [2/1/17].

Kohlberg, L. (1984) *Essays on moral development: Volume 2: The psychology of moral development.* New York, NY: Harper & Row.

Merkle, R.C. (April 1978). Secure communications over insecure channels. *Communications of the ACM,* 21 (4). 294–299.

Ofsted (2010) *The safe use of new technologies* London.

Shannon, C. (1949) Communication theory of secrecy systems. *Bell System Technical Journal,* 28 (4). 656–715.

Turkle, S. (1995) *Life on the screen. Identity in the age of the age of internet.* New York, NY: Touchstone.

QuickStart Computing Scotland

Subject Knowledge covering Level 3 and 4

QuickStart Computing provides the vital subject knowledge enhancement for secondary computing teachers.

It covers the content of the Computing Science and Digital Literacy curriculum at Levels 3 and 4 so that you and your colleagues can teach it confidently and successfully.

Inside you'll find:

- Detailed coverage of Computing Science and Digital Literacy topics included at Levels 3 and 4
- Guidance on developing and applying the concepts and approaches of computational thinking
- Programming examples using Scratch, Snap! and Python
- Practical classroom activities and inspiration
- A carefully curated collection of the best online resources and reference materials
- Notes on what's covered in primary computing to ensure excellent transition

All this guidance is also available online at:
computingatschool.org.uk/scotland/quickstart

COMPUTING AT SCHOOL SCOTLAND
EDUCATE · ENGAGE · ENCOURAGE
Part of BCS, The Chartered Institute for IT

SUPPORTED BY

ISBN 978-1-78017-441-9