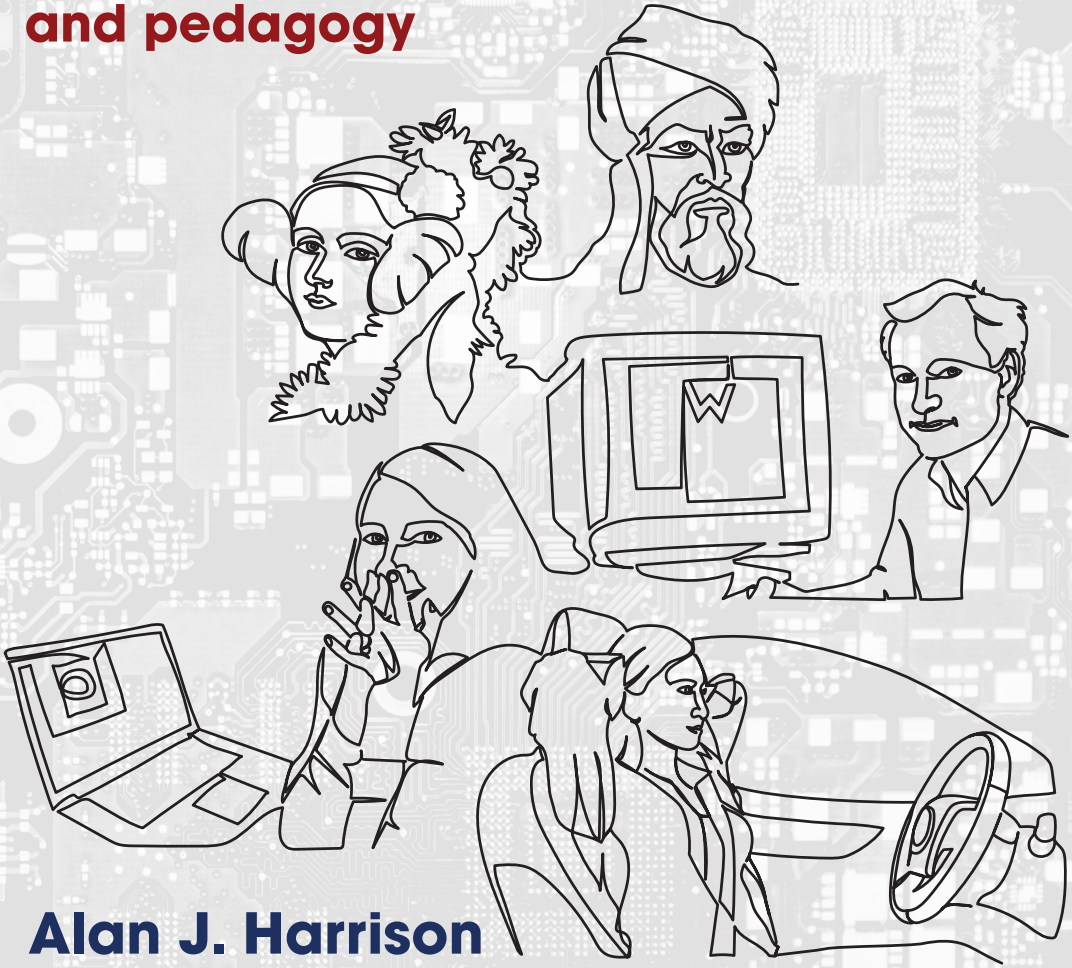
A JOHN CATT PUBLICATION

HOW TO TEACH COMPUTER SCIENCE

Parable, practice and pedagogy



Alan J. Harrison

Alan J. Harrison is head of computing at a school in Manchester. He is a Computing at School Master Teacher and community leader, a National Centre for Computing Education training facilitator and a Raspberry Pi Foundation content author. Fascinated by computing education, Alan enjoys studying the latest research, discussing with peers, and writing blogs and books. *@mraharrisoncs*

HOW TO TEACH COMPUTER SCIENCE

Parable, practice and pedagogy

Alan J. Harrison

First published 2021

by John Catt Educational Ltd,
15 Riduna Park, Station Road,
Melton, Woodbridge IP12 1QT

Tel: +44 (0) 1394 389850
Email: enquiries@johncatt.com
Website: www.johncatt.com

Set and designed by John Catt Educational Limited

Contents

Introduction

Who is this book for?

How To Teach Computer Science is for new or aspiring teachers wishing to improve their subject knowledge and gain confidence in the classroom. It's also for experienced computer science teachers who wish to hone their practice, especially in the areas of explicit instruction, tackling misconceptions and exploring pedagogical content knowledge.

Trainee teachers, NQTs and early-career teachers will find this book invaluable. Experienced teachers will find it inspiring. And all will benefit from a fresh look at the hinterland and pedagogy that makes computer science a fascinating subject to teach.

What's in the book?

We will explore some of the backstory to our subject, the "hinterland" – those fascinating journeys into history that make computer science come alive and place it in historical context. These stories will help you to enrich your lessons, cement core knowledge, develop cultural capital and excite a lifelong love for the subject. We will go beyond the mark scheme to explore the subject knowledge behind the answers, giving you the confidence to discuss the field in greater depth and enabling you to use explicit instruction methods.

We will consider the misconceptions that arise when teaching computer science, so you can head them off at the pass. And we will look at pedagogical content knowledge (PCK) – teaching ideas that can be lifted and dropped straight into the classroom to immediately enhance your teaching.

How should I use this book?

There are three ways in which *How To Teach Computer Science* could be useful:

1. **Subject knowledge enhancement.** Read this book from start to finish if you're on an initial teacher training course, in your NQT year, or switching to computer science from another subject. The book is organised into chapters by England's typical GCSE specification, so the content should map easily on to existing textbooks and CPD courses, and enhance your learning.
2. **Pedagogy primer.** Because it takes a look at all the current research findings and what they mean for the classroom, this book should get you started in considering how to teach computer science – going beyond subject knowledge to the PCK that gets results. This content should be valuable to new and experienced teachers alike.
3. **Resource index.** Dip into the book for inspiration when planning lessons, with or without reading the whole thing. Each chapter covers a typical GCSE topic to make finding inspiration simple, while footnotes contain links to online resources and there are more at the companion website, **httcs.online**. The website will be updated regularly, so check back often!

What is PCK?

Lee Shulman defined PCK as "knowledge of the most regularly taught topics in one's subject area, the most useful forms of representation of those ideas, the most powerful analogies, illustrations, examples, explanations, and demonstrations ... [it] also includes an understanding of what makes the learning of specific topics easy or difficult".[1] Punya Mishra and Matthew Koehler extended Shulman's model to include a focus on teaching with technology and gave us "technological pedagogical content knowledge", or TPCK:

> *"TPCK is the basis of good teaching with technology and requires an understanding of the representation of concepts using technologies; pedagogical techniques that use technologies in constructive ways to teach content; knowledge of what makes concepts difficult or easy to learn and how technology can help redress some of the problems that students face."*[2]

---

1 Shulman, L.S. (1986) "Those who understand: knowledge growth in teaching", *Educational Researcher*, 15, 4-14

2 Mishra, P. & Koehler, M.J. (2006) "Technological pedagogical content knowledge: a framework for teacher knowledge", *Teachers College Record*, 108(6), 1017-1054

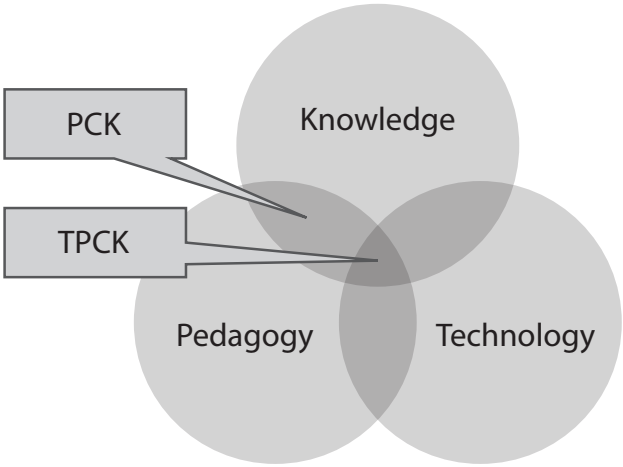


Figure 1.1: Venn diagram illustrating the intersection between PCK and TPCK.

This book explores both PCK and TPCK through the lens of a typical GCSE computer science curriculum. The National Centre for Computing Education (NCCE) has published a series of "quick reads" on the Teach Computing blog,[3] explaining some of the pedagogical techniques that work, and I have included examples of these in the PCK section of the relevant chapters. The PCK section of each chapter is organised under the following nine headings.

1. Core concepts

It seems obvious, but you can't teach a subject without first deciding what the learners need to know. Unfortunately, exam board specifications are often assessment-focused, not learning-focused, so aren't organised around concepts. This section will summarise the core concepts that learners need to grasp. For more clarity and an idea of how to sequence your teaching of these concepts, you may wish to refer to the NCCE "concept maps" on the Teach Computing curriculum website.[4]

2. Fertile questions

Fertile questions – you may know them as "big questions" or "enquiry questions" – are intriguing questions that the teaching tries to answer, and they help to tie all the lessons in a topic together. Mark Enser explains further in an article for *TES*:

3 blog.teachcomputing.org/tag/quickread

4 teachcomputing.org/curriculum

> *"This sense of intrigue sparks my pupils' natural curiosity. The subject itself becomes engaging rather than an activity designed to hook them. By phrasing each topic as a fertile question to be answered, I have been able to think more carefully about the disciplinary knowledge that a geographer would need in order to answer it. I find myself asking, what propositional and procedural knowledge will they have to bring to the question? Rather than, what can I teach to fill up the lessons this half-term?"*[5]

To work towards answering the fertile questions, you may devise lesson objectives or, better still, more specific "lesson questions". William Lau, in his book *Teaching Computing in Secondary Schools*, gives an excellent explanation of lesson questions, relating them to the computing curriculum:

> *"Linked to the Fertile Question should be a lesson question that provides a clearly defined context and encourages higher-order thinking. By having a clearly defined context we reduce demand on working memory."*[6]

Lau gives a detailed example of a set of fertile questions and example lesson questions in his book, and I've paraphrased the first set in the table below to illustrate the idea. This would be an excellent fertile question for the topic of architecture (see chapter 6).

Fertile question	Lesson questions
How can we design the fastest computer system in the world?	1. Why does my phone get hot, and why does cooling it speed it up? 2. Why do my phone and tablet boot in seconds, while my desktop takes a minute? 3. Why are some manufacturers' computers more expensive than others? 4. Why did chip manufacturers stop increasing clock speeds and instead add extra cores? 5. Why does magnetic storage still exist if solid state drives are quicker?

3. Higher-order thinking

We all know about Bloom's taxonomy of thinking skills: remembering, understanding, applying, analysing, evaluating, creating. As teachers, our job is first to ensure learners are secure in the topic: they can remember, understand and

5 Enser, M. (2020) "Are you asking fertile questions? If not, you should be", *TES*, link.httcs.online/fertile

6 Lau, W. (2017) *Teaching Computing in Secondary Schools: a practical handbook*, Routledge

apply the new knowledge. Then we should aim to increase a learner's time spent thinking at the higher levels of the taxonomy: analysing, evaluating and creating. Once learners are ready for the higher-order thinking skills, you can use some of my suggested activities for the topic.

4. Analogy and concrete examples

Analogies help to explain abstract ideas using a similar idea in a familiar concept. Concrete examples exist in the real world and put the learning into context, connecting new computing ideas to other subjects, which helps pupils to assimilate them into their existing understanding.

Analogies can be used as part of a "semantic wave". Also described as "unplug-unpack-repack", a semantic wave is explained best by James Robinson on the Teach Computing blog:

> *"You take the terminology used by experts, which is often rich in meanings, and you unpack those meanings. You make them simpler, more relevant to the learners, and more concrete. Pupils explore this new concept using a familiar context and simple terminology, making it easier for them to understand and apply. Following this, the pupils can repack those simple meanings into the expert terminology, ensuring that they understand its nuances and can use it appropriately. By following this wave from the original meaning down to something familiar and then back up, you can build pupils' understanding and prevent them misunderstanding key terms or being limited to overly simplistic language."*[7]

5. Cross-topic and cross-curricular learning

Computer science should not be taught as a series of discrete units that exist independently of each other. Learners understand the subject much better if the links between concepts are made explicit, and they are encouraged to make their own links either within the subject or across the curriculum. This section will suggest some links that you can make, and activities that make these links explicit.

6. Unplugged

Getting away from the computer is a very useful strategy for making abstract concepts clear. We lose any distractions or technical issues, and can teach computing even without computers. We can make topics real for students using analogies, similes, metaphors, role play, games, puzzles, magic tricks and storytelling.

---

7 Robinson, J. (2020) "How we teach computing", Teach Computing, blog.teachcomputing.org/how-we-teach-computing

7. Physical

Getting hands-on can make a computing lesson engaging and inclusive, with sensory and creative experiences. There is also some evidence that girls engage more with physical computing, as a physical project may have more immediate real-world applications.[8] In this section we will discuss ideas for use with Raspberry Pi, micro:bit and other equipment, and if you don't have these to hand you can borrow them from your local Computing at School hub (England only).

8. Project work

The upper levels of Bloom's taxonomy can also be accessed through project work, as pupils apply their knowledge and create products. Projects give pupils a goal, an audience and a brief to fulfil, for which they need to make autonomous decisions about the skills, knowledge and tools they will need.

9. Misconceptions

Misconceptions can seriously hinder learners' progress, and studies have shown that teachers who are aware of common misconceptions and actively seek to address them are more effective.[9] Each chapter in this book ends with a table of common misconceptions, either from computer science education research, crowdsourced from my personal learning network, or taken from Project Quantum.[10] It's not meant to be an exhaustive list, but those I've chosen should help you become misconception-aware.

General pedagogical principles

While the PCK section of each chapter covers topic-specific pedagogy, there are some teaching and learning principles that cut across topics. The following three sections detail these general principles. I recommend you adopt some or all of these in the computing classroom to improve your practice.

1. Flipped classroom

A flipped classroom or flipped learning approach allows for more powerful learning experiences in the classroom. In a flipped classroom, the students do

8 Franks, R. (2021) "A journey into physical computing", *Hello World*, link.httcs.online/hw15physical

9 Sadler, P.M., Sonnert, G., Coyle, H.P., Cook-Smith, N. & Miller J.M. (2013) "The influence of teachers' knowledge on student learning in middle school physical science classrooms", *American Education Research Journal*, 50(5)

10 diagnosticquestions.com/quantum

some learning before the lesson, usually set as the previous lesson's homework. This frees the teacher from delivering some of the core knowledge, so they can instead focus the lesson on practical application of that knowledge.

The Learning Foundation website explains the flipped classroom approach,[11] while Alan O'Donohoe, long-running host of Preston's "Raspberry Jam" and provider of computer science teacher training, explains how to use the exa.foundation's GCSE computer science MOOC[12] to facilitate flipped learning on his YouTube channel.[13]

The exa MOOC contains many links to online sources that can be used in a flipped approach, including the Craig'n'Dave video playlists. I often set an exa MOOC topic or a Craig'n'Dave video as flipped homework, requiring my students to make notes on the content before the lesson, either in Sketchnote form or in Cornell style for older students. You can read more about Craig'n'Dave's flipped approach on their website.[14]

2. Using specialist language

This book's clear descriptions of each topic and deep exploration of the hinterland should enable teachers to explain topics in great depth, suitable for a direct-instruction approach to teaching. Pedagogical approaches such as Talk for Writing from Pie Corbett[15] have shown that children internalise the key terms and language structures needed to write knowledgeably about the subject when teachers "talk the text" confidently, model their thought processes and explicitly teach the specialist language of the subject.

3. Cognitive science approach

I use the techniques described in this book as part of a wider strategy of research-informed teaching. I take an approach that follows Barak Rosenshine's Principles of Instruction,[16] explored by Tom Sherrington in his book *Rosenshine's Principles in Action*.[17] You can read more on the introduction of Rosenshine's principles to my classroom on my blog.[18]

---

11 learningfoundation.org.uk/the-flipped-classroom

12 courses.exa.foundation

13 O'Donohoe, A. (2016) "Basic introduction to J276 MOOC and flipped learning" (video), YouTube, link.httcs.online/exaflipped

14 craigndave.org/our-pedagogy

15 talk4writing.com

16 Rosenshine, B. (2010) "Principles of Instruction", International Academy of Education, link.httcs.online/rosenshine

17 Sherrington, T. (2019) *Rosenshine's Principles in Action*, John Catt

18 httcs.online/rosenshine

Much of the PCK advice in each chapter is aimed at increasing what Doug Lemov calls ratios, explained in a blog post by Adam Boxer:

- *"Participation ratio: how many of your students are participating and how often?*
- *Think ratio: when they are participating, how hard are they thinking?"*[19]

Increasing these ratios is important because "memory is the residue of thought", as Daniel Willingham explains in his book *Why Don't Students Like School?*[20] The PCK ideas in each chapter offer many ways to get students thinking hard about the subject knowledge that matters.

Every teacher should have at least a passing understanding of cognitive load and how to ensure learners' working memory is not overloaded. We will see in this book how techniques like PRIMM, pair programming, worked examples and Parsons problems can all reduce cognitive load. You can read more about cognitive load theory in the computing classroom in Phil Bagge's article for *Hello World* magazine.[21]

4. Retrieval practice

Another key finding from psychology is the Ebbinghaus forgetting curve and the role of retrieval practice in overcoming it. From 1880 to 1885, the German psychologist Hermann Ebbinghaus tested his own recall of nonsense three-letter words such as "wid" and "zof". He found that memory declines rapidly at first, then more slowly, but crucially can be boosted by what Ebbinghaus called "overlearning".[22] We can achieve this boost through retrieval practice, using techniques such as self-quizzing and frequent, low-stakes review quizzes.

Doug Lemov describes the forgetting curve and how to defeat it in *Teach Like a Champion*[23] and you can read more on his blog.[24] Lemov recommends a "do now" activity to review previous learning at the start of every lesson. William Lau describes on his blog how he runs a "quick fire five" low-stakes quiz at the start of every lesson,[25] and he's published a large set of suitable questions at mrlaulearning.com.

---

19 Boxer, A. (2020) "Ratio", *A Chemical Orthodoxy* (blog), link.httcs.online/ratio

20 Willingham, D.T. (2010) *Why Don't Students Like School?*, Jossey-Bass

21 Bagge, P. (2020) "Cognitive load theory in the computing classroom", *Hello World*, 8, link.httcs.online/hw8cog

22 Ebbinghaus, H. (1885) *Memory: a contribution to experimental psychology*

23 Lemov, D. (2014) *Teach Like a Champion 2.0*, John Wiley & Sons

24 Lemov, D. (2021) "An annotated forgetting curve", *Doug Lemov's Field Notes* (blog), link.httcs.online/tlacforget

25 Lau, W. (2019) "Quick fire five", *Look Who's Learning Too* (blog), link.httcs.online/lauqff

Learners should be encouraged to do their own retrieval practice, using techniques such as look-say-cover-write-check[26] and digital retrieval practice tools such as Quizlet, Memrise, Cram and GoConqr. The founder of the CogSciSci discussion group,[27] Adam Boxer, has created a website dedicated to retrieval practice called Carousel Learning.[28]

Inclusion

There is no subheading for inclusion in each chapter. That's because adopting some or all of the suggested PCK techniques would inherently make your classroom more inclusive. The suggested activities mostly have "a low floor, wide walls and a high ceiling" – a phrase coined by Seymour Papert that guided the development of Scratch at MIT.[29]

Reducing cognitive load can also support SEND learners. As Catherine Elliott explains in her discussion of PRIMM for *Hello World* magazine, "A young person with SEND can thus learn about the same computer science concepts as their peers, without the fear of failure, or the demand on working memory and recall that writing a program from first principles involves."[30]

As Beverly Clarke writes, also for *Hello World*, equitable computing would mean "experiences that are high quality in terms of pedagogy and robust in terms of nature and scope of learning goals, taking students beyond the curriculum".[31] And unplugged activities, physical computing and project work offer multiple means of engagement, representation, action and expression, as recommended in the Universal Design for Learning (UDL) framework discussed in *Hello World*.[32]

Universal Design for Learning

In *Computer Science in K-12: an A to Z handbook on teaching programming*, Maya Israel and Todd Lash explain that the UDL framework "maximises students'

26 link.httcs.online/bbclook

27 link.httcs.online/cogscisci

28 link.httcs.online/carousel

29 Resnick, M. (2008) "Mindstorms over time: reflections on Seymour Papert's contributions to education research", presented at a special session of the 2008 American Educational Research Association annual meeting, link.httcs.online/lowfloor

30 Elliott, C. (2020) "The inclusive computing classroom", *Hello World*, 12, link.httcs.online/hw12inc

31 Clarke, B. (2020) "Encouraging an inclusive computer science environment", *Hello World*, 11, link.httcs.online/hw11inc

32 Leonard, H. (2021) "Universal Design for Learning in computing", *Hello World*, 15, helloworld.raspberrypi.org/issues/15

strengths and reduces instructional barriers".[33] These might include accessibility barriers such as visual impairment, which might suggest a screen-reader-compatible programming environment. Cognitive load barriers would benefit from explicit instruction, modelling and scaffolding, using the techniques described in this book.

Israel and Lash explain the UDL Instructional Planning Process, which has four steps posed as questions:

1. What are my instructional goals?
2. What barriers could interfere with students achieving those goals?
3. What methods and materials can I use to address the instructional barriers in this activity?
4. How might I assess learning in a flexible manner?

When answering question 3, we might decide to scaffold a learning activity. Scaffolding might take the form of several stages of explicit instruction: modelling, guided practice, then independent practice with support. This is what P. David Pearson and Margaret Gallagher called "gradual release of responsibility", or the "I do, we do, you do" model.[34] This model aims to keep the learner in what Lev Vygotsky called the "zone of proximal development".[35] Thus we reduce what John Sweller called "extraneous cognitive load"[36] so the learners can focus just on what they need to learn at each stage.

You can read more about UDL at udlguidelines.cast.org.

> As an aside, once I understood cognitive load theory, I realised that one problem my learners faced was extraneous load from the plethora of languages and platforms in my curriculum. I inherited a KS3 plan that included Scratch, Small Basic, Code Lab (JavaScript), App Inventor (Java) and Python. With just one lesson per week, my poor students spent most of their time learning new coding environments and syntaxes, so there was very little room for "germane cognitive load", i.e. computational thinking.

33 Israel, M. & Lash, T. (2020) "Universal Design: reaching all students" in Grover, S. (ed.) *Computer Science in K-12: an A to Z handbook on teaching programming*, Edfinity

34 Pearson, P.D. & Gallagher, M.C. (1983) "The instruction of reading comprehension", *Contemporary Educational Psychology*, 8(3), 317-344

35 Vygotsky, L.S. (1978) *Mind in Society: the development of higher psychological processes*, Harvard University Press

36 Sweller, J., van Merrienboer, J.J.G. & Paas, F.G.W.C. (1998) "Cognitive architecture and instructional design", *Educational Psychology Review*, 10, 251-296

Gender balance

Teachers should also be aware of current research around gender balance in computing, with women accounting for just 17% of IT specialists in the UK.[37] The advice in this book can be augmented by an approach that encourages girls to study the subject at GCSE and beyond. Computing at School (CAS), in collaboration with the University of Manchester, produced a helpful booklet that includes advice such as these tips:

- *"**Tip 1: women can change the world**. Make links to the big picture and real-world computing roles that have an ethical remit – for example, designing assistive technologies to enable people to overcome a disability, or highlight technology's role in medicine, humanitarian work, science, fashion, communications, art, journalism or sport. Mainstream media representations of people working in computing are largely white males writing computer code. It's important to provide a more realistic, balanced and aspirational viewpoint."*
- *"**Tip 2: role models**. Discuss female role models and put up displays of women in technology. Many young women do not identify STEM careers as interesting, relevant or appropriate for their gender."*
- *"**Tip 3: out and about**. Plan trips where girls can see female role models working in technology that isn't just coding, such as Jodrell Bank, Cadbury World or local employers."*
- *"**Tip 8: encourage and praise**. Ensure that praise addresses all aspects of learning computing, including creative solutions, planning and conceptual understanding, as well as technical knowledge and skills."*[38]

Many of the other tips in the CAS booklet reinforce the need for a range of activities in computing lessons, including unplugged, physical and project work that allows for collaboration and freedom of expression.

By employing some of these strategies, I have aimed to foster a powerful, inclusive learning environment in my computing classroom. The results so far have been promising, with the proportion of girls in my Year 10 classes rising from 5% in 2016-7 to 30% in 2020-21. But like the IT industry and society more generally, I still have a long way to go!

---

37 Little, J. (2020) "Ten years on, why are there still so few women in tech?", *The Guardian*, link.httcs.online/womeninIT

38 Rydeheard, D. et al. (2018) *Girls Into Computing: top tips for schools*, University of Manchester. PDF available at: link.httcs.online/uomgirls (free CAS registration required)

Why did I write this book?

The germ of an idea for this book was planted by a blog post by Tom Sherrington called "Signposting the hinterland: practical ways to enrich your core curriculum". Sherrington explains that curriculum can be divided into "core" and "hinterland", where the hinterland is as important as the core and serves the purpose of:

- *"Increasing depth: niche details about a particular area of study that deepen and enrich the core.*
- *Increasing breath: wider surveys across the domain of any curriculum area that help to locate any specific core element within a wider frame."*[39]

Sherrington quotes from an earlier blog post by Christine Counsell that explains why the hinterland is important:

> *"The core is like a residue – the things that stay, the things that can be captured as proposition. Often, such things need to be committed to memory. But if, in certain subjects, for the purposes of teaching, we reduce it to those propositions, we may make it harder to teach, and at worst, we kill it."*[40]

The original aim of this book was to assist computer science teachers in sharing some of that hinterland with their students, in order to enrich their studies, cement core knowledge in a wider context, and engender an appreciation for the subject that goes beyond what's required to pass exams and, in many learners, excites a lifelong love for the subject. Once I started to write the book, I realised that the pedagogy of our subject is underappreciated in the classroom practitioner community. So I decided to enhance the book with insights into some of the current research, making the concepts accessible to teachers with concrete, practical ideas.

Why is this book needed?

Computer science is a young subject, taught in schools only since the early 1980s and – after a hiatus in which "ICT" took over in UK schools – re-established as a core subject only in 2014, as part of the national curriculum subject of "computing". Computer science graduate teachers are scarce, and many schools employ non-specialists to teach our hugely important subject. Pedagogy specific to computing is therefore underdeveloped and – by many teachers – largely overlooked. Computing teacher forums and social media groups are awash with requests for

39 Sherrington, T. (2019) "Signposting the hinterland: practical ways to enrich your core curriculum", *Teacherhead* (blog), link.httcs.online/sherrington

40 Counsell, C. (2018) "Senior Curriculum Leadership 1: The indirect manifestation of knowledge: (A) curriculum as narrative", *The Dignity of the Thing* (blog), link.httcs.online/counsell

"lessons on network protocols" and "schemes of work for network security". Much rarer are conversations around questions such as "Should we do protocols before network security?", "What is a good analogy for teaching protocols?" and "What misconceptions do students develop when learning about networks?" Questions like these are seen regularly in online communities dedicated to more mature subjects such as English and history.

In *How To Teach Computer Science*, we will explore pedagogy, such as the best way to teach sorting algorithms. We'll see how more emphasis on reading code before writing it and working in pairs as "driver and navigator" is getting results in programming lessons. We'll look at techniques to explain networking, including "Post-it note packet switching", and we'll ponder the question "Are we spending too long making topologies out of string, and not enough on how the internet works?"

Reading this book should improve your performance as a computer science teacher. That is my primary aim. I also hope it improves your confidence and allows you to enjoy teaching our wonderful subject as much as I do.

In the introduction to his important book *A Discipline of Programming*, Edsger Dijkstra wrote: "My original idea was to publish a number of beautiful algorithms in such a way that the reader could appreciate their beauty."[41] If this book reveals only a tiny fragment of the beauty of computer science and gives you the confidence to share it with young learners, then I will have achieved my goal.

41 Dijkstra, E. (1976) *A Discipline of Programming*, Pearson

Chapter 1.
Data representation

Switzerland, June 1816

George Gordon, Lord Byron, has fled England and settled in Switzerland, with his fellow poet Percy Bysshe Shelley and Shelley's future wife, Mary. During days of incessant rain, which keeps the party indoors, Byron suggests a competition to write the best ghost story. Mary imagines a mad professor who reanimates a corpse with tragic consequences. She names the scientist Victor Frankenstein and publishes the story anonymously on her return to England in 1818.

Byron, meanwhile, having left behind scandal, a doomed marriage, his beloved homeland and his first child – his infant daughter Augusta Ada – pens these lines:

Is thy face like thy mother's, my fair child!
Ada! sole daughter of my house and heart?
When last I saw thy young blue eyes, they smiled,
And then we parted, — not as now we part,
But with a hope. — Awaking with a start,
The waters heave around me; and on high
The winds lift up their voices: I depart,
Whither I know not; but the hour's gone by,
When Albion's lessening shores could grieve or glad mine eye.

Meeting Ada

Augusta Ada, known as Ada, was raised by her mother, the highly educated mathematician Annabella Milbanke. Byron, famously described by one of his many lovers as "mad, bad and dangerous to know", had been an ill match for the devoutly religious Annabella and the Byrons had separated after only a year of marriage. Byron never saw Ada or Annabella again. He travelled on through Italy to Greece, where he helped the Greeks fight for independence from the Ottoman Empire, ultimately earning Greek national hero status. He died of a fever in 1824, when Ada was eight years old.

Fearing Ada might inherit what she called Byron's "insanity", Lady Byron schooled her daughter in science and mathematics and discouraged literary study. Ada was a diligent pupil, studying under Mary Somerville and Augustus De Morgan; her studies and social exploits brought her together with the greatest minds of the age, including Michael Faraday, Charles Dickens and one Charles Babbage. Ada went on to marry William King, who was later made Earl of Lovelace, whereupon she became Countess of Lovelace.

Figure 1.2: Ada, Countess of Lovelace, sometimes considered the first computer programmer.

In her adolescence, Ada was fascinated by the mechanics of flight, studying birds and documenting her thoughts on human flight in a book she called *Flyology*, written at the age of 12. Ada showed a knack for using mathematics to understand nature. Her maths tutor, Mary Somerville, was one of the first female members of

the Astronomical Society and the first person to be described in print as a scientist. Somerville introduced the 17-year-old Ada to the inventor and mathematician Charles Babbage at a party, where Babbage was demonstrating his Difference Engine, a mechanical calculator designed to speed up the computation of scientific tables. Ada was fascinated by the machine and became a friend, student and assistant to Babbage for the rest of her life.

Numbers for everything

Ada Lovelace was 100 years ahead of her time in her ability to imagine what might be accomplished by computers. She saw only an unfinished prototype of the Difference Engine within her lifetime; she died of cancer in 1852, at the age of 36, while Babbage worked on an improved design for the machine. Despite having only physical experience of the number-crunching machine, Lovelace was able to describe the concept of a general-purpose computer that would be able to manipulate sounds and images. In 1843, she wrote:

> *"[The Analytical Engine] might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent."*[42]

Early bitmaps

Lovelace didn't limit her imaginary computers to music. Inspired by the images woven into rich brocades, she also suggested that Babbage's machine might be capable of creating graphics. The Analytical Engine, she wrote, "weaves algebraic patterns just as the Jacquard loom weaves flowers and leaves".

The French textile merchant Joseph-Marie Jacquard had patented his automated weaving loom in 1804; by the 1840s, when Lovelace wrote that line, Jacquard looms were common across the UK. Rolls of punched cards drove the raising and lowering of warp threads – a difficult job previously performed by a "draw boy" – thus automating the production of patterned cloth. The presence of a hole meant that a warp was raised; no hole meant that it remained lowered. The cards thus carried a binary code that was interpreted by the loom as a pattern. More than 100

42 Charman-Anderson, S. (2021) "Ada Lovelace: Victorian computing visionary", Finding Ada, link.httcs.online/ada1

years before digital computer monitors, we see a binary code that represents a two-dimensional image.

Babbage would adopt the idea of punched cards in his design for the Analytical Engine, which sadly remained unfinished. But the program that Lovelace designed for it, to calculate Bernoulli numbers, survived and we will look at it in chapter 5.

Figure 1.3: The Jacquard loom was fed with binary data on punched cards.

News travels slow

On the other side of the Atlantic, the problem of transcontinental communications was to drive the invention of more codes for data representation. The Gold Rush had swelled California's population to nearly 400,000 people by 1860. But they were largely cut off from the rest of the Union, with stagecoaches taking a month to make the arduous journey from east to west. With a civil war looming and local businesses demanding faster communications, a group of Missouri-based entrepreneurs founded the original Pony Express. Using a string of 200 relief stations and lone horsemen riding in relays, the service slashed the Missouri-

California mail-delivery time; in March 1861, the inaugural address of Abraham Lincoln arrived in the California capital, Sacramento, in a record seven days. But the Pony Express was a flop, folding after 18 months without ever turning a profit.

The route ran through Native American land, of course, and the white settlers had little respect for the Paiute people. Frequent conflict peaked in the Pyramid Lake War in May-June 1860, in which untrained settler militias lost 80 men before the US Army arrived and killed at least 160 Paiute. Several Pony Express relay stations were destroyed and the service closed until Californians raised enough cash to rebuild and fortify the line. Meanwhile, the Paiute people were forced to retreat from the fertile lands around the lake, and it is believed that many starved in the subsequent months, before the Paiute were permitted back on to the land "so long as they agreed to use peaceful means to settle disputes and grievances".[43]

But it wasn't Native American resistance that killed off the Pony Express – it was technology. On 24 October 1861, just 17 years after Samuel Morse dispatched the first telegraphic message over his experimental line between Washington DC and Baltimore (see chapter 9), the first transcontinental telegraph line across the US was complete. Instantaneous communications rendered the Pony Express obsolete and the service closed just two days later.

Zip code

Morse's invention worked by sending pulses of electricity down cables, causing an electromagnet to raise and lower a pen that made marks on paper: the famous dots and dashes. Morse intended the operator to read these dots and dashes and then resend them, using a key to complete a circuit that sent pulses to the next relay station. The relay operators, however, quickly developed an ear for the clicking of the mechanism, ignoring the paper output, so Morse replaced the pen with a beeping loudspeaker. Operators would simply hear the short and long beeps and write down the English letters they represented.

43 militarymuseum.org/PyramidLake.html

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	— — ● ●

Figure 1.4: The Morse code table for the basic English alphabet.

Why short and long beeps? Morse had only a simple electrical circuit to play with. Either the circuit is complete, or it's not. A complete circuit would cause electricity to flow and the pen to lower (or the speaker to beep). He realised that he could vary the duration to give himself two symbols: a dot (short beep) or a dash (long beep). He then assigned a set of dots and dashes to each letter of the alphabet. Morse had created an effective code with which to represent all the letters of the alphabet, enabling him to use a simple device to transmit text over long distances.

Binary: yes or no?

Morse code consists of two symbols, dots and dashes, so is it a binary code? Imagine an operator needs to send the word "road". The code is dot-dash-dot pause, dash-dash-dash pause, dot-dash pause, dash-dot-dot, or • – •, – – –, • –, – • •. But if we move the last pause slightly further on in the sequence, we get "rope": • – •, – – –, • – – •, •.

There are thousands of such ambiguous Morse patterns, including base/brie/deli, with/pens/axes, real/lend/rice and even centres/cards/trench.[44] Clearly the gaps are significant: the pauses are a symbol in their own right. In fact, a short pause is used between letters and a long pause at the end of a word, so Morse has four

44 link.httcs.online/morse1

symbols in total: dot, dash, short pause and long pause. It's not a binary but a quaternary code.

We need to head over to France to find an early binary code. Louis Braille injured an eye in his father's leather workshop at the age of three; the resulting infection led to blindness in both eyes by the time he was five. At the age of 10, in 1819, he obtained a scholarship to the National Institute for Blind Children in Paris, which at the time used a system of raised letters invented by Valentin Haüy. Letter shapes are not distinct enough to be easily discerned by touch, however, and Braille found Haüy's system hard to use.

While attending the institute, Braille was shown a system of raised dots used by the military to communicate at night. He took this system and improved upon it, using just six dots to communicate all the letters of the alphabet, plus numbers and some punctuation symbols. Each dot is raised or flat, and a blank space (effectively six flat dots) separates words and sentences. In this way, the grid of six dots could represent 2 x 2 x 2 x 2 x 2 x 2 = 26 or 64 different characters.

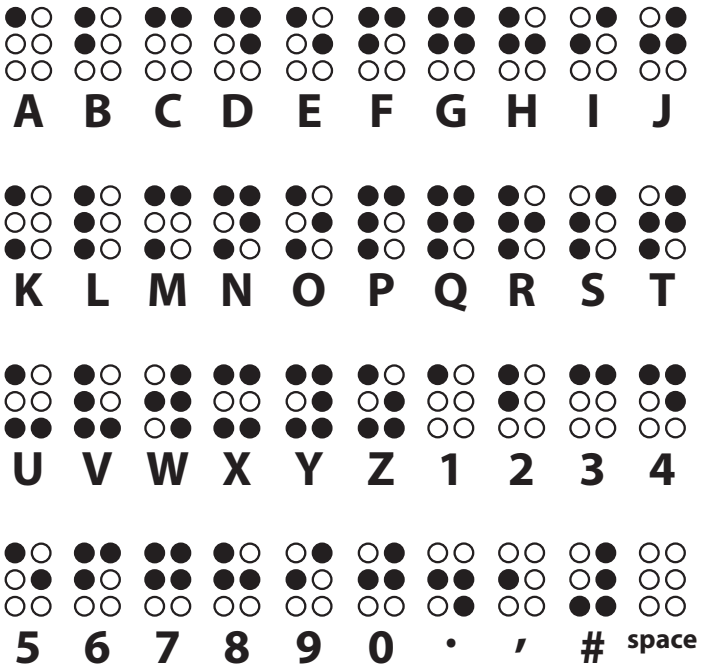


Figure 1.5: The Braille code.

Braille is therefore a binary code for representing text. If we order the dots as Braille did – from 1 to 3 on the left column, top to bottom, and then 4 to 6 on the right column – then each of the braille codes can just as easily be written out as a sequence of bumps and flats. So "A" is bump-flat-flat-flat-flat-flat and "H" is bump-bump-flat-flat-bump-flat. Replacing bump with 1 and flat with 0, we can write "A" as 100000 and "H" as 110010. We can now write any text using just two digits, 0 and 1. Louis Braille had created a binary code to represent text, when electronic computers had not yet been invented.

Number crunching

The founding fathers of the United States of America, led by Thomas Jefferson and James Madison, declared in 1787 that a census should be taken immediately and then "within every subsequent term of ten years". The first census counted around four million Americans, rising to 50 million by 1880, generating a quantity of data that took almost the whole decade to process. Experts warned that the 1890 census wouldn't be processed before the 1900 census began, prompting the government to launch a contest for a solution.

The competition was spotted by Herman Hollerith (1860-1929), who was lecturing on mechanical engineering at MIT. Despite an "inability to learn spelling easily", Hollerith had graduated with distinction from Columbia University before joining MIT. Recalling the ticket-punch machines used by train conductors, Hollerith realised that holes in card might store data that would be readable by machines. He left MIT to work for the US Patent Office in Washington DC and, in his spare time, worked on a system of machine-readable punched cards.

Hollerith's system of punches and tabulating machines won the competition easily and was adopted for the 1890 census. The "rough count" of 62 million people was announced within just six weeks, while the full reports were out in six years and answered many more questions than the previous survey. Hollerith went on to sell his invention around the world, founding the Tabulating Machine Company in 1896. The company was acquired in 1911 by Charles R. Flint, who amalgamated it with three other businesses to create the Computing-Tabulating-Recording Company (CTR). Flint later hired Thomas J. Watson from National Cash Register, and Watson became president in 1915. CTR went on to popularise the new field of data processing and selling, and the company was renamed International Business Machines (IBM) in 1924.

In the interwar period, IBM became the largest manufacturer of electromechanical data-processing machines in the world, but they were not computers. In IBM's tabulators, prongs would push through holes in card, making contact with a

brush or a mercury bath on the other side, causing a relay to close. In turn, this would make a mechanical adding machine – known as an accumulator – tick over. For example, if a hole for "marital status" were punched through, the "married" accumulator would increment, counting the married persons in the population. Cards might cause many accumulators to increment, meaning many different data points could be processed at once. Later machines added basic maths and could even compare two cards and make simple decisions.

The hole truth

Hollerith's original cards could be punched in just 24 columns, which was fine for recording numeric codes and yes/no answers. But with IBM customers wanting to process text fields such as names and addresses, more columns were needed. In 1928, IBM launched the 80-column, 12-row punch-card format, creating a new code called Binary Coded Decimal Interchange Format (BCDIC) to give the holes meaning. BCDIC was extended in 1963 to create a more general-purpose code called EBCDIC, capable of supporting many languages; this was required for IBM's flagship mainframe, the System/360 series (see chapter 8).

Figure 1.6 shows a Hollerith card punched with the first 64 characters of the EBCDIC character set. In theory, it was possible to punch up to 4,096 different hole patterns in each column. But, in practice, punching the holes too close together resulted in card failure, so only 256 different bit patterns were actually used. (A card punched in every hole, known as a "lace card", was sometimes created to prank novice operators, as it invariably jammed the card reader mechanism, requiring the poor operator to break out the "card saw" to clear the blockage.)

Figure 1.6: IBM's 80-column punched card, invented in 1928 and used right up to the 1980s.

Code of conduct

Far from being extended, BCDIC should have been dropped in 1963, superseded by an international standard. In 1961, IBM engineer Bob Bemer proposed a single code for computer communication. The American National Standards Institute (ANSI) formed a committee of all the computer and teletype makers of the day – including IBM – and two years later announced the American Standard Code for Information Interchange, or ASCII. IBM was expected to adopt ASCII in all its systems. However, the System/360 programme manager Fred Brooks realised that the cost of replacing all punch-card readers was prohibitive and went with EBCDIC instead, which persists to this day on IBM mainframes.

Both EBCDIC and ASCII are character sets: look-up tables that translate letters and punctuation marks to numeric codes. A character set thus enables the storage and processing of text by a digital computer, which also means data created on one computer can be processed by another computer. Most computers built after 1963, including the popular UNIVAC series, used ASCII.

Originally a seven-bit code representing only 128 unique symbols, ASCII's international popularity demanded more characters. Various eight-bit versions, often called Extended ASCII, were popular in the 1970s and 1980s, with an eight-bit standard emerging in 1987. Sharing a character set paved the way for direct networked communications, beginning with the ARPANET project in 1969, which had developed into the modern internet by 1981 (see chapter 9). The internet would not have been possible without ASCII, and later developments such as HTML also owe a great debt to the standard character set.

Eight bits is not enough

Computer makers standardised on eight-bit bytes in the early 1970s, so the eight-bit EBCDIC and Extended ASCII character sets made perfect sense. But the 256 different bit patterns available from eight bits were not enough for languages such as Arabic, Chinese and Japanese, and the Unicode standard was inaugurated in 1991. Originally a 16-bit code, giving over 65,000 characters, a later version called UTF-8 allows up to 32 bits per character, giving room for all modern languages. Unicode also supports many ancient languages, such as Egyptian hieroglyphs, as well as mathematical symbols, emojis and even mah-jong tiles. The first 128 characters of UTF-8 are identical to ASCII, making it backwards-compatible and the standard endorsed by the World Wide Web Consortium (W3C).

Unicode opened up the internet to non-English-speaking peoples who had previously been forced to work in European languages, and in that sense the universal character set was an important leveller. As Unicode Consortium lawyer

Andy Updegrove put it in a 2015 interview, "Without [Unicode] we would be stuck in an upgraded example of a colonial world, where historically first world nations continue to force their cultures and rules on emerging nations and their peoples."[45]

Just the fax, ma'am

Taking a selfie with a modern smartphone and sending it to friends has become second nature to many of us. Nokia, the Finnish mobile giant of the 1990s and 2000s, launched its first smartphone in 2001. The 7650 was a 104 MHz RISC device running the Symbian OS (see chapter 8), complete with a 0.3-megapixel camera and multimedia messaging support. Nokia sold 750,000 every month during 2002. But what these early "cameraphone" owners didn't know was that images were being transmitted more than 130 years earlier.

The Scottish inventor and clockmaker Alexander Bain saw the potential of the telegraph to transmit not just encoded text – as with Morse code – but images too. From 1843 to 1846 he worked on an experimental "facsimile" machine. A sending drum studded with electrically powered pins represented the image; a stylus would pick up the charge and transmit it, then the receiver would transfer the charge to a drum coated in electrochemically sensitive paper. As the stylus touched a pin, the charge would cause a dot to be created on the receiving drum.[46]

The results were disappointing, owing to difficulty in synchronising the drums, but the principle was sound. Building on Bain's work, the Italian physicist Giovanni Caselli successfully launched the first commercial fax service between Paris and Lyon in 1865. And in 1921 the first newspaper images were successfully transmitted by submarine cable from London to New York, using image data that had been encoded on to punched paper tape. The Bartlane system, developed by Harry Bartholomew and Maynard McFarlane, used a five-hole paper tape, each column representing the brightness of a "pixel" rendered in greyscale by the receiving printer.[47]

These early image-encoding techniques were invented for transmission of pictures over distances. But what they all had in common was, of course, the use of binary code to represent real-world data. Once the digital computer age was upon us, the challenge became simply how to store these black and white or greyscale brightness values and process them electronically. In 1948, scientists at Manchester University came up with a novel idea.

45 Updegrove, A. (2015) "The Unicode 8.0: a song of praise for unsung heroes", ConsortiumInfo.org, link.httcs.online/updegrove

46 Borth, D.E. "Fax", Encyclopædia Britannica, link.httcs.online/bainfax

47 link.httcs.online/bartlane

Family album

The Manchester Baby, also called the Small-Scale Experimental Machine, was invented by Freddie Williams and Tom Kilburn at Manchester University in the 1940s (see chapter 6). The memory store used in the machine was a cathode ray tube (CRT) wired to self-refresh every millisecond, so it didn't forget its contents. Each dot on the phosphor screen stored one bit, and there was a main memory of 32 x 32 dots, hence 1024 bits or 128 bytes of memory.

This Williams-Kilburn tube was covered by a detection plate and enclosed within a metal Faraday cage to prevent interference from electric trams running nearby, so Williams hooked up a second CRT, synchronised with the first, allowing the operator to see the contents of main memory. The Williams-Kilburn tube can therefore be considered the first frame buffer or bitmap used to drive a video display.

A bitmap – like Jacquard's punch cards, Bain's drum, the Bartlane tape or Williams and Kilburn's CRT memory – is an array of bits representing the brightness (and later the colour) of picture elements or pixels of an image. The principle is nothing new: it can be seen in ancient Greek, Roman and Persian mosaics.

In 1957, Russell A. Kirsch, an engineer at the US National Bureau of Standards, developed a scanner that sampled the brightness of an image at 30,000 points over a 2in square surface. The resulting 176 x 176 image of Kirsch's son was stored in the computer's acoustic delay line memory (see chapter 6) before becoming the first ever digital photograph to be printed. The original survives to this day in the Portland Art Museum in Oregon.

Figure 1.7: This digital image of Russell Kirsch's son, Walden, contained just 30,000 pixels.

Colouring in

Using a single bit for every pixel, we can store a pure black and white image, representing black with 0 and white with 1. However, figure 1.7 is actually a greyscale image, meaning it has several brightness levels, known as luminosities. Kirsch achieved this by multiple passes of the scanner, each set to a different scanning threshold. Later scanners would detect the luminosity of each point on the image in a single scan. If we increase the bit depth to two bits per pixel, we can store an image with four luminosities – for example, black (00), dark grey (01), light grey (10) and white (11).

We can extend this idea to give us colour. In 1972, the British engineer Michael Tompsett was working at Bell Labs in New Jersey, US. Using bit patterns to represent different colours, he created the first digital camera image in genuine colour, a picture of his wife. Tompsett used a charge-coupled device (CCD) sensor to record just eight bits per pixel, giving an image composed of 256 colours.[48]

> "Why would anyone want to look at their picture on a television set?" – Kodak's reaction when shown the first images taken with a portable digital camera invented by Steven Sasson, a Kodak engineer, in 1975[49]

For more than a decade, bitmap files created on one machine were unusable on other devices, until Microsoft created a standard bitmap format for Windows 3.1 with the file extension .bmp in 1994. This device-independent bitmap format supports one-, four-, eight- and 24-bit colour depth, but it doesn't compress well and so .bmp is rarely used on the internet.

In the late 1980s, the US internet service provider CompuServe (see chapter 9) wanted to add colour image support to its service, and created the Graphics Interchange Format (GIF) in 1989. With eight-bit colour, transparent backgrounds, good compression and support for simple animations, the format – originally pronounced with a soft "g" – was quickly adopted as an internet standard and supported by early browsers such as Netscape Navigator.

Meanwhile, camera manufacturers were working to enhance the image sensor demonstrated by Tompsett; they needed a file format for storing photographic images that could represent millions of colours for photographic realism. The Joint Photographic Experts Group created the first JPEG standard in 1992, whereupon it was adopted by all the major camera manufacturers. JPEG is a bitmap format

48 link.httcs.online/tompsett

49 Zhang, Michael. (2017) "What Kodak said about digital photography in 1975", *PetaPixel*, link.httcs.online/kodakquote

with 24 bits per pixel: eight bits each for red, green and blue luminosity. These values are calculated from the detected image using a lossy compression algorithm called DCT, first proposed in 1972 by Nasir Ahmed. Ahmed was made an IEEE Fellow in 1985 "for his contributions to engineering education and to digital signal processing".

The JPEG format is excellent for digital photography – its 24 bits of colour depth can represent 16 million colours. But unlike GIF, JPEG does not support transparency. And with a copyright dispute in the 1990s over JPEG's LZW compression algorithm, lack of support for lossless compression and GIF's limitation of 256 colours, an ad-hoc group of enthusiasts meeting on the Usenet group *comp.graphics* (see chapter 9) decided to develop a rival format called PNG, released in 1996. The new format became a W3C standard in the same year, and in 2013 it overtook GIF as the most popular lossless, bitmap image-compression format on the internet.[50] PNG, like JPEG and GIF, can be edited by all modern raster image-editing applications, such as Adobe Photoshop and the open-source GIMP.

Long and winding paths

Bitmap images are also called rasters, after the scanning process of a cathode ray tube. But dividing an image into square pixels and storing the colour of each pixel as a value is not always the best way to turn visual information into numbers. Bitmaps do not scale well, causing blurring, or "pixelation", when enlarged. And when working with solid shapes, using square pixels to represent curves and diagonal lines is hardly ideal, causing serrated edges known as "jaggies". Kirsch himself regretted limiting himself to square pixels: "It was something very foolish that everyone in the world has been suffering from ever since," he said in 2010.[51] Kirsch had experimented with using variable-shaped pixels to smooth images, but it didn't catch on. Meanwhile, a different solution was found that originated on radar screens.

Early radar operators viewed detected objects as dots or "blips" on a screen that began as a modified oscilloscope. As technology improved, these blips became shapes such as triangles, and the image could be annotated with codes, range lines and geographic features. Everything was drawn by an electron beam that travelled only over the lit areas, not touching the dark areas at all. Vector scanning is still used today in oscilloscopes and aircraft head-up displays, and was seen briefly in the 1980s arcade games *Asteroids*, *Star Wars* and *Battlezone* (see figure 1.8).

50 link.httcs.online/png

51 Associated Press. (2020) "Computer scientist, pixel inventor Russell Kirsch dead at 91", *ABC News*, link.httcs.online/kirschsquare

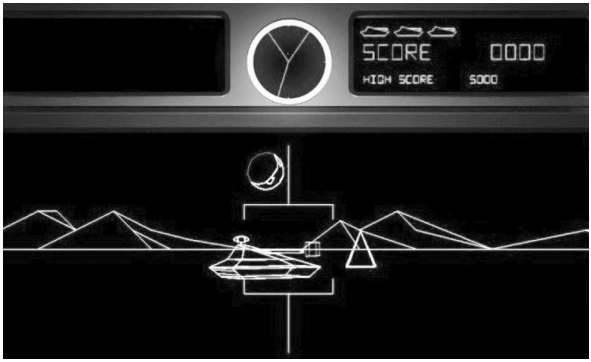


Figure 1.8: Atari's 1980 arcade game *Battlezone* featured vector-scanned white lines under a green and red overlay.

The vector-scanning method was the inspiration for vector image formats. Every shape in a vector image is a line or polygon defined by a set of x and y coordinates, or a curved path between two points defined by a mathematical function. This makes the image very precise and allows us to multiply the x and y values by a scale factor, enlarging an image while preserving quality. Defining shapes by perfect curves also ensures the image has no jagged edges – at least, none stored in the file. Of course, when rendered on a raster-scanned screen, even a vector image turns into square pixels, but at a high enough resolution the resulting edges still look smooth.

So, in the late 1990s, web users were looking for a vector file format. The desktop publishing industry, led by the fledgling Adobe Systems, had developed a vector-based file format for driving laser printers, called PostScript, back in 1984. Unfortunately, PostScript files don't scale well without becoming huge, so are unsuitable for the web. The W3C created a working group to find an alternative and Scalable Vector Graphics (SVG) was adopted as a web standard in 2001. SVG files can be edited by all modern vector image applications, including Adobe Illustrator and the open-source Inkscape.

Data about data

When an application such as a browser opens a file, it needs to know how to interpret the data. Remember that all data is just a row of bits, once stored in a file. How does the browser know the dimensions of the image and the number of colours to use? The answer is metadata. The prefix "meta" comes from the Greek for "beside" or "beyond", and "going meta" usually means going up a layer of abstraction. In this

case, we are going up above the luminosity or path data to describe the nature of the data itself, such as the dimensions. Metadata can also include other useful facts about the file, and the JPEG standard includes support for camera data such as model, f-stop and aperture in a section called "EXIF" data.

The importance of knowing the format of a file before processing it with application code is so important that there are two ways in which a file identifies itself to the operating system. Windows uses file extensions such as .gif or .jpg and associates each extension with an application. It's easy to accidentally change this extension, however, or even delete it, and Unix-derived operating systems such as Linux and macOS (see chapter 8) don't use file associations anyway, so the metadata in the file header will also identify the content. For example, all JPEG files begin with the two bytes FFD8 in hex, GIFs begin with the ASCII string GIF89a, and PDFs begin with %PDF. Because this trick originally used only numeric codes, this identifying string is still sometimes called the file's "magic number".

Sound ideas

The sensor at the back of a digital camera, called a CCD, performs an analogue-to-digital conversion, in order to store information about the real world as binary data. Analogue information is continuously variable, while digital data can have only discrete values. If you look at your face in a mirror, you will see skin of continuously varying shades and tones: no two points will have exactly the same colour. But a JPEG image must map this information to a maximum of 16 million colours. If that JPEG is converted to a GIF, the number of colours comes down to just 256. Analogue-to-digital transformation is at the heart of many data-capture processes that allow computers to process real-world information, and another is digital audio.

Just as a bitmap records the qualities of visual information as binary data by sampling squares of a two-dimensional plane, so a digital audio recording samples the sound information at regular intervals of time. While the iPod and its rivals are 21st century products, the principle behind digital audio dates back to 1937 and again we have the telecommunication industry to thank. The British telephone engineer Alec Reeves, of ITT's Paris headquarters, realised he could solve the noise problem on long-distance telephone lines by sampling the sound into digital data and sending pulses over the line, to be turned back into analogue sound at the receiving end. Pulse code modulation (PCM) required expensive valve circuitry in the 1930s, and the idea was shelved until semiconductors made it viable 30 years later.

The music industry experimented with digital recordings throughout the 1970s and the first all-digital album was Ry Cooder's *Bop Till You Drop*, released in 1979,

although the finished product was still sold on analogue vinyl records and tapes. Consumer digital audio players made their debut in 1982, with the launch of the digital audio compact disc format – CD for short. A sample rate of 44.1kHz means that CD audio stores the amplitude of the sound wave as a single binary number 44,100 times every second. That binary number has 16 bits, giving more than 65,000 possible values per sample, and CDs have both left and right channels for stereo sound, providing a total of 16 x 2 x 44,100 = 1.4 million bits for every second of sound sampled.

This quantity of data (a three-minute pop song took up more than 30MB) was just too large to share on the expanding internet in the 1990s, so compressed music formats were invented, the most popular of these by far being MP3. Short for MPEG Audio Layer III, where MPEG is the Moving Picture Experts Group, MP3 is a lossy compression algorithm that works by stripping out sounds that are largely inaudible, and then compressing the rest using a version of DCT (see page 32). MP3 achieves more than 75% reduction in file size compared with the same uncompressed WAV or CD-audio data file. With the emergence of affordable MP3 players, the format enabled a boom in digital music downloads – with associated copyright controversies – in the late 1990s.

Microsoft initially shunned the controversial format, preferring its own proprietary format, Windows Media Audio. Similarly, in 2003, Apple launched the iTunes music store, containing 200,000 songs encoded in Advanced Audio Coding (AAC) format, despite Apple's iPod range having MP3 support. Other formats sprang up, such as the open-source, lossless Ogg Vorbis and FLAC, and the Apple Lossless Audio Codec (ALAC). These formats varied in compression rates and storage techniques, but the 1937 principle of pulse code modulation remained at the heart of them all. Just like text and images, strings of binary digits are given meaning by a code. By manipulating these binary digits, we can process real-world information.

What the hex?

Computer scientists work with binary data so often that they have devised an easier way of representing it, borrowed from maths. Binary is a base-2 system because it uses only two symbols, 0 and 1. This means that the place values of digits in a binary number go up in powers of two from right to left. Denary, also called the decimal number system, uses the 10 symbols from 0 to 9, so the value of each column is 10 times the one to its right.

Converting between binary and denary is fiddly because it involves repeated dividing or multiplying by powers of two. What if there was an easier way? Denary

representation is efficient: only three or fewer digits are needed to write a byte. So we need a system of similar efficiency (or better) but with easier binary conversions.

Enter the base-16 hexadecimal system, which takes its name from the Greek for six and 10. Hexadecimal uses the numbers 0-9 plus the letters A, B, C, D, E and F. With 16 different symbols available for every column, we can encode numbers from 0 to 255 using just two symbols every time. More importantly, each symbol correlates to exactly four bits. Long binary numbers can easily be converted to short, memorable strings of hexadecimal characters, and vice versa.

Colour codes in JPEG images and HTML pages can now be represented not with a string of separate decimal numbers like RGB (255, 192, 203) but as a single "hex" code: #ffc0cb, for example, which is the code for pink. The hash symbol denotes hex, just in case it's not obvious when there are no letters in the code, such as in the code for royal blue: #002366. The same number system is used wherever binary data needs to be made human-readable, including in assembly language programming, error codes and network addresses. It's important to remember, however, that the hex code exists only for human consumption. All data remains stored in binary – the only number system that can be processed by the computer.

Codes for things

The fundamental nature of a computer as a general-purpose number-crunching machine implies that the only hurdle we must overcome before information can be processed by an algorithm is to find a way of expressing that information as numbers. In short, digitising analogue information makes it computable. Ada Lovelace would be pleased that her prediction turned out to be accurate: we did find a way to express all manner of objects by "the abstract science of operations".

TL;DR

At the heart of this topic is the idea that as long as we can turn information into binary data, we can use a computer to process it. Digital computers process binary numbers because they use two-state electrical signals. The challenge is therefore to find a transformation from real-world information to binary. This transformation is called encoding and it makes use of a code.

ASCII and Unicode are used to encode text; JPEG, GIF, PNG do the same for bitmap images; and WAV, MP3 and AAC encode digital sound. But it's important to realise that there are virtually limitless ways of encoding information and these are just the techniques that are widely used, owing to their effectiveness or official recognition, or both.

Analogue-to-digital conversion is the process of mapping the original data to the digital representation, and it's vital to understand binary in order to really grasp the importance of bit depth, resolution and their effect on file size. Metadata is "data about data" and describes the contents of the file or something about the original information.

PCK for data representation

Core concepts

- The principle of representing information as numbers.
- Why binary? Relationship between binary and two-state electrical signals.
- Number bases.
  - Base-2 or binary.
  - Base-10, decimal or denary.
  - Base-16 or hexadecimal.
  - Converting between number bases.
  - Where each number base is used and why.
- Calculations and operations on binary numbers.
  - Addition, including handling overflow.
  - Logical shifts (and how these relate to multiplying or dividing by two).
- Bits and bytes; relationship between bits and quantity via powers of two.
- Binary file sizes: bit, nibble, byte, kB, MB, GB, PB, TB.
- What is a character set?
  - ASCII, its characteristics and limitations.
  - The need for Unicode and its characteristics.
- Calculating number of bits from characters.
- Bitmap/raster and vector images; characteristics.
- How does a bitmap represent an image?
  - Pixels, bit depth (aka colour depth), dimensions, resolution.
  - Relationship between bit depth, resolution, dimensions and file size; performing calculations.
- What is digital sound?
  - Analogue-to-digital conversion; sampling.

    - Sample rate (aka sampling frequency), sample resolution (aka bit depth), duration.
    - Relationship between sample rate, sample resolution, duration and file size; performing calculations.
- Compression, the need for it and common techniques.
- Metadata, its purpose and typical usage in text, image and sound files.

Fertile questions

- Can a computer store and process anything we see or hear?
- A Word document takes up just 100KB of storage until I insert images. Then it's 12MB – why?
- Why does a 700MB CD sound the same as a 50MB MP3 album?

Higher-order thinking

Create your own character set

ASCII and Unicode are international standard character sets. But many other character sets have been used over the years. Learners may want to create their own character sets, optimised for different purposes. For example, you could ask them to create these character sets and discuss their pros and cons:

- A character set for representing only English lower-case letters and five punctuation marks. They should realise that this can fit in five bits, but the text it can store is limited.
- A character set that encodes every word in the English language separately as a single number. Learners should see there is no reason that a single number should encode a single letter, but that this character set itself would be huge. Around 18 bits would be needed to cover all 240,000 words in the *Concise Oxford English Dictionary*, and many bit patterns would almost never be used.
- A character set of just emojis. How effective would this be as an international standard?

Analogy and concrete examples

Make your own image filter

Discuss image filters: the learners are likely to have used or experienced them – for example, filters that remove blemishes or add effects such as "vignette" or "vintage" to photographs. Explain that image filtering is performed by an algorithm. Show a simple bitmap image and the result of applying an effect to it (such as "brighten"). Discuss what has happened to the data underlying the

image (all the numbers representing brightness have been increased by a similar proportion).

Use Python with an image library such as PIL. Create and demonstrate some sample code, then allow the students to edit it to make their own image filters. You can find sample code for a red filter at httcs.online – the program is easily edited to change the colour cast of a JPEG image. Martin O'Hanlon has created a whole course on using Python with image filters.[52]

Cross-topic and synoptic

Cross-topic with programming

Image filters are a great means of linking programming to data representation. You could also link programming with sound data, using a Python library such as wave.[53]

Cross-topic with architecture

Systems architecture can be linked to data representation in many ways. A Graphics Processing Unit (GPU) can improve the speed of image manipulation, because its circuitry is designed to change hundreds of binary data values in the time the CPU takes to process just one.

Cross-topic with memory and storage

Data relates closely to memory and storage, with more efficient representation methods requiring less storage capacity. Likewise, network bandwidth determines the size and quality of the images we can download, linking the topic to chapter 9.

Cross-topic with issues

Impacts and issues include body image concerns caused by "Photoshopped" images in the media; how the trend towards data-heavy multimedia websites widens the digital divide; the need for alt tags on images to improve accessibility; problems with plagiarism and protecting copyright when images are used online; and bias in algorithms – how AI struggles to recognise non-white faces in images.

Cross-curricular

Cross-curricular with other STEM subjects

The binary system pre-dates electronic computers by hundreds, possibly thousands of years. The *I Ching*, the ancient Chinese divination text that dates back to at least 750 BC, contains 64 hexagrams made up of six lines, each of which can be broken

52 link.httcs.online/fldatacourse

53 link.httcs.online/pymm

or solid, resulting in a binary code. The 17th century German mathematician Gottfried Leibniz was fascinated with the *I Ching* and his article introducing the binary system to Europe was entitled "Explanation of the binary arithmetic, which uses only the characters 1 and 0, with some remarks on its usefulness, and on the light it throws on the ancient Chinese figures of Fu Xi".[54] Like all number bases, binary is obviously a mathematical concept, so you could discuss a joint scheme of work with the maths department. Quantity multipliers kilo, mega, giga and so on are common across all STEM subjects, so discuss these in the context of the natural sciences by talking about kilometres, megawatts and gigahertz.

Cross-curricular with geography

Before the 1961 ASCII standard, computers used different character sets, making communication difficult. ASCII was thus fundamental to the ARPANET project, which evolved into the modern internet (see chapter 9). The language of the web, HTML, is built on ASCII and its later replacement, Unicode. Without standard character sets we would not have the World Wide Web, with its profound effect on human communication.

Cross-curricular with art and design

Digital images are important in art and design, and a digital photography club is a great place to really understand bitmaps, resolution, colour palettes and metadata. Likewise, the music department might already have digital music editing on the curriculum, and a digital music club could illuminate the topics of sound sampling, compression and plagiarism.

Unplugged

Paper bitmaps

Bitmap image creation is a great topic for unplugged activities. Give the learners a grid of empty pixels and a colour code and ask them to reveal the picture. They can then make their own picture, save it as numeric data and give the numbers to a partner, who reveals the image.

Link this with the system software topic by telling the story of Susan Kare, who designed the first Mac icons (see chapter 8). Ask learners to design new icons in just 16 x 16 bits![55] [56] [57]

54 Von Aue, M. (2018) "Gottfried Wilhelm Leibniz: how the I Ching inspired his binary system", *Inverse*, link.httcs.online/leibniz

55 link.httcs.online/pixelpainter

56 link.httcs.online/funpixels

57 link.httcs.online/pixelart

Physical

Micro:bit LED bitmaps

The micro:bit has a built-in 5 x 5 grid of LEDs that can be set easily with blocks or text coding – perfect for teaching one-bit bitmaps. If you have a Raspberry Pi with the camera attachment, you can take pictures and process them with Python. You can use the Minecraft API to store these images in a "photobooth" within the Minecraft world. A full tutorial is available at raspberrypi.org.[58]

Sound engineering

Using the open-source software Audacity, learners can get hands-on with sound recordings, finding out how the sample rate and bit depth affect sound quality and file size. The National Centre for Computing Education's resources include a great lesson on this (add a microphone for even more hands-on fun).[59]

Misconceptions

Misconception	Reality
Binary can only represent numbers up to 255 in denary	Binary is the base-2 number system and like any number base it can represent any number. This misconception arises due to the lack of exposure to numbers larger than 255, numbers spanning many bytes. Discuss binary in the context of number bases. Explain it exists as a number base outside of the subject of computing, and show how it can represent numbers over 255.
Binary place values start at 1 on the left and 128 on the right of an eight-bit number	Binary numbers are written right to left in order of rising place value, just like any other number base derived from the Arabic number system. The least significant bit, representing units, is on the far right. This misconception arises again through insufficient explanation of binary as a standard place-value number system analogous to denary. Plenty of comparisons of denary and binary, discussions of place value, lots of teacher demonstration, and the availability of a number grid with place values written above during early student practice, should head this off.
The number of possible colours in a bitmap image is equal to the number of bits per pixel, so four bits gives four colours	The number of possible colours is calculated as $2^{\text{bit depth}}$, so four bits gives $2^4 = 16$ colours, and eight bits gives $2^8 = 256$ colours. This misconception can be tackled by doing some student activities with low bit depths from one to four, so they can see how the number of bit patterns, therefore the number of colours, doubles each time you add a bit.

58 link.httcs.online/photobooth

59 link.httcs.online/nccerep2

Misconception	Reality
Bit rate = sample rate	In digital audio, sample rate or sampling frequency is the number of times we sample an analogue signal in a given timeframe, usually measured in samples per second, or Hertz. Bit rate is the resulting quantity of bits in a given time frame, usually a second, calculated by `bit depth * sample rate`. (Bit depth is often called sample size: number of bits per sample.)
Encoding means using a secret code to keep data secure	A code is simply a means of transforming information into numerical data. ASCII is a code, as is the RGB colour code. The resulting number can be stored and processed by the computer. Codes are not ciphers – they are designed to be transparent and completely reversible. Ciphers are used to make information secret, which is the process of encryption, not encoding.

Chapter 2. Programming

Massachusetts Institute of Technology, 10 April 2019

Late afternoon. The final stage of the pipeline of algorithms is executing. Dr Katie Bouman sits at her MacBook and watches as the picture of the black hole starts to appear. She and a team of computer scientists, astrophysicists and electrical engineers have been working on this project for three years. Five petabytes of data on half a tonne of hard drives from telescopes around the world arrived at MIT more than a week ago, and the algorithms have been churning it ever since.

The M87 black hole appears tiny from Earth – as big as an orange would appear on the surface of the Moon. Refraction limits what we can see with our telescopes, so the very best image of the Moon from Earth consists of 13,000 pixels, but each pixel would then contain around 1.5 million oranges. To take an image of a black hole we would need an Earth-sized telescope. We can't make one of those, but we can connect telescopes around the world, giving us lots of low-resolution images from different angles that could be processed by computers into a single image.

That's what Bouman's team did, creating an Earth-sized computational telescope called the Event Horizon Telescope (EHT). Just as several different low-res images of the same face can be used to generate an accurate prediction of the real face, we can use these sparse, noisy images and combine them to create a more detailed picture. Bouman has spent the last three years building a computational "pipeline" to do just that, feeding images from radio telescopes around the globe into the algorithm to eventually produce an image.[60]

60 Bouman, K.L. (2019) Written testimony before the Committee on Science, Space, and Technology, United States House of Representatives. link.httcs.online/bouman

The full story can be heard in Bouman's TED talk,[61] but what excites me is that the programming language chosen for all this computation is Python. At around 6.45pm on 10 April 2019, a researcher takes a picture of Bouman at her computer. We can see a code window on the right of her screen, which looks like the Matplotlib Python library. We can see the now famous image of the M87 black hole. But, most important of all, we are privileged to witness the joy of discovery. Bouman presses her hands to her mouth, eyes full of wonder. An algorithm, her algorithm, has unlocked one of the secrets of the universe.

Figure 2.1: The moment of discovery: the world's first image of a black hole appears on Katie Bouman's laptop screen.

Code book

Katie Bouman knew, just like Ada Lovelace almost two centuries earlier, that programming could be fun and rewarding. In the monograph *The Art of Computer Programming*, the first volume of which was originally published in 1968, Donald Knuth writes:

61 Bouman, K. (2016) "How to take a picture of a black hole" (video), TEDxBeaconStreet, link.httcs.online/boumanted

> *"The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music."*[62]

The first volume of *The Art of Computer Programming*, subtitled *Fundamental Algorithms*, was the authoritative work on the techniques we're likely to meet in school: basic input and output, mathematics, subroutines, searching and sorting. But it's not for the faint-hearted. The foreword from Bill Gates reads: "If you think you're a really good programmer, read this. You should definitely send me a résumé if you can read the whole thing."

Knuth's mammoth "programmer's bible" was included on *American Scientist*'s list of "100 or so books that shaped a century of science", alongside *The Autobiography of Charles Darwin, A Brief History of Time* by Stephen Hawking and *In the Shadow of Man* by Jane Goodall. That a discipline only decades old shares such illustrious company says a lot about the importance of programming in our modern world.

Construction time again

Sequencing instructions is obvious – it's just one thing after another. But selection, iteration and subprograms are no accidents. The early assembly-language programmers had only a conditional jump instruction to play with, yet used this to build selection, iteration and subprograms because these constructs were so useful, proving they are not imposed from above, but natural features of algorithms.

As an aside, this process would repeat itself constantly as high-level languages developed: a useful abstraction created from simpler code would later make its way into the language, and that feature would be used to build higher abstractions, and so on forever. In this way, high-level languages gained arrays, records, string handling functions, linked lists, sort and search routines, classes and much more.

These constructs were scientifically proven to be a complete set of tools for any program back in 1966. The Böhm-Jacopini theorem was welcomed by fans of structured programming, which has many benefits including readability and ease of maintenance. Hence, Niklaus Wirth's highly structured language Pascal (see chapter 4) was used by many universities to teach programming in the 1970s and 1980s.

Undergraduate courses in computer science would also delve into the other paradigms, exploring functional programming with LISP and logic programming with Prolog. Degrees often included a sprinkling of COBOL, despite its lack of local

62 Knuth, D. (1997) *The Art of Computer Programming, Volume 1: fundamental algorithms* (third edition), Addison-Wesley

variables and simple iteration (which was simulated by repeating subprograms through a PERFORM … TIMES/UNTIL statement). Many academics agreed with Edsger Dijkstra's controversial statement: "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence."[63]

One thing after another

Computer science was hardly taught at all in UK schools before 1982. A handful of pioneering maths teachers would have a Commodore PET, RM 380Z or Apple II with which to teach a few high-flyers. The BBC's Computer Literacy Project, which ran from 1982 to 1989, changed all that. Schools could access central funding to purchase BBC Micros preloaded with the BASIC language; they were supplied with a tutorial and a selection of demonstration programs on cassette, and supported by a series of television programmes. A team at Acorn Computers in Cambridge, led by Steve Furber and Sophie Wilson, designed and built the machines. They even came with a proprietary networking protocol called Econet, which allowed Micros to be connected in a bus topology to a file server sporting two 5¼-inch floppy disks of 100MB each.

The choice of BASIC was controversial, with the more structured Pascal common in higher education. But Apple, Sinclair, Commodore and Tandy were all selling home computers preloaded with a dialect of BASIC, which swung the argument. Acorn's Wilson wrote a version of BASIC for the new Micro, and the BBC commissioned the National Extension College in Cambridge to write a tutorial called "30-Hour Basic". BBC Basic included IF-THEN-ELSE and the much-despised GOTO statement (see chapter 4), but also features from structured programming including DEF PROC, DEF FN and REPEAT-UNTIL. Schools were encouraged to teach "computer studies" and the supporting TV series – *The Computer Programme*, broadcast in 1982 – explored the real-world applications of computers while teaching beginners how to code.

The first ever programming tutorial on UK TV was called "Just One Thing After Another" (the second episode in *The Computer Programme* series).[64] It demonstrated a music player, a home automation program, the sorting algorithms bubble sort and quick sort, and how to code a simple quiz using PRINT and INPUT. Many schools recorded the programmes and showed them to classes on their VCRs or set them as homework. The Royal Society's *Shut Down or Restart?* report, published in 2012, credited the Computer Literacy Project with

63 Dijkstra, E.W. (1975) "How do we tell truths that might hurt?", link.httcs.online/ewd498

64 link.httcs.online/tcp

"establishing the UK's strengths in the computer games industry, and clearly led to the establishment of ARM Ltd, the world's leading supplier of microprocessors for mobile consumer electronics".[65]

The ICT years

The 1990s saw the rise of the PC (see chapter 6). By the turn of the century, a strong understanding of digital technology and skilled use of application software were important for school-leavers. The UK government responded by putting information and communications technology (ICT) on the curriculum, although there was no mention of programming. The level 7 descriptor[66] stated merely: "They develop, test and refine sequences of instructions as part of an ICT system to solve problems."[67]

The *Shut Down or Restart?* report was scathing about the teaching of ICT in schools:

> *"We have found examples of imaginative and inspiring teaching under the ICT heading. Sadly, however, these positive examples are in a minority, and we have found far too many examples of demotivating and routine ICT activity, and a widespread perspective among pupils that 'ICT is boring'."*[68]

The report made several recommendations, including a clear statement that programming should be back on the curriculum:

> *"Pupils should be exposed to, and should have the option to take further, topics such as: … computer programming, data organisation and the design of computers; and the underlying principles of computing."*[69]

The report extolled the virtues of computational thinking and explained that not only was programming important if you wanted a job in technology, but it was also a useful skill in its own right:

> *"Programming is the quintessential craft: the principles of quality, workmanship, fitness for purpose, and considerations of project management (time, quality and cost), are all learned here in their purest form."*[70]

---

65 Furber, S. (2012) *Shut Down or Restart? The way forward for computing in UK schools*, Royal Society, link.httcs.online/shutdown

66 National curriculum level descriptors were abolished in 2014.

67 link.httcs.online/curric

68 Furber, S. (2012) *Shut Down or Restart? The way forward for computing in UK schools*, Royal Society, link.httcs.online/curric

69 Ibid.

70 Ibid.

For these reasons, the UK's 2014 national curriculum explicitly stated that computational thinking and programming should be taught in all schools, and programming was back on the syllabus.

It's not about code…

Dijkstra said: "Computer science is no more about computers than astronomy is about telescopes."[71] However you learn to code and whatever language you choose to do it in, it's important to remember that programming is not about keywords, punctuation and indentation. It's about problem-solving.

When Peter Samson learned the opcodes of the TX-0, he did so to solve problems (see chapter 4), and he wrote his music player to solve a problem, namely "How can I make this machine play Bach?" The team that created the Fortran compiler wanted a quicker way to code the mathematical problems involved in designing aircraft; then everyone used the compiler and wrote high-level programs to solve the mathematical problems. When Katie Bouman wanted to take a picture of a black hole by combining images from several telescopes, she wrote a program to solve that problem.

Programming exists to solve problems using a machine. To do this, you must find a way to state the problem computationally, then get a machine to perform the computation. The first part is what we now call computational thinking. It's easily the largest part of the process, but often overlooked by novice programmers and expert instructors alike.

71 link.httcs.online/dijkstrabio

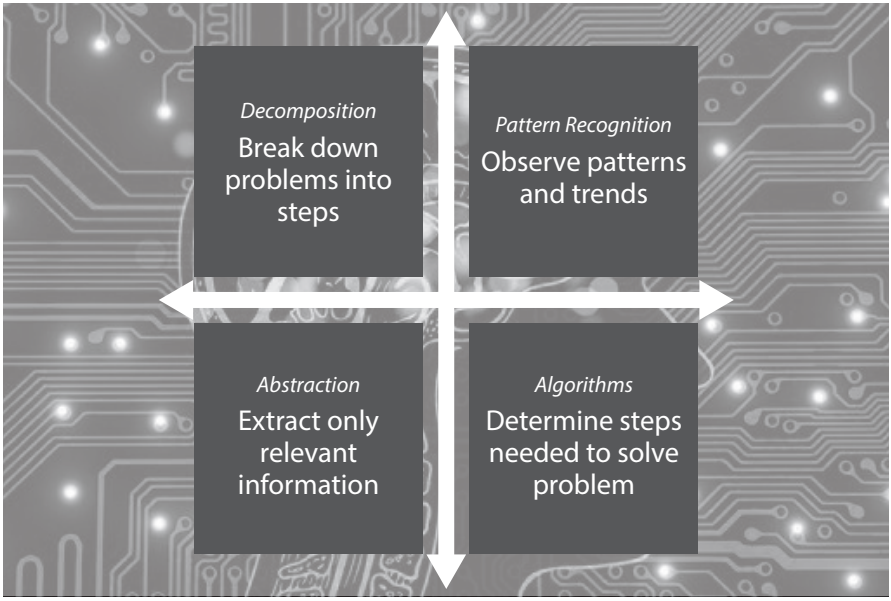


Figure 2.2: Computational thinking is the largest part of the programming process.

It's all about computational thinking

In 2017, Jeannette Wing, then head of computer science at Carnegie Mellon University in Pennsylvania, published a groundbreaking paper explaining that computational thinking (CT) "is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer – human or machine – can effectively carry out".[72]

CT is not "thinking like a computer", but creating an algorithm to solve a problem with a computer. The Barefoot programme, which supports computing education in UK primary schools, explains further:

> *"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."*[73]

72 Wing, J.M. (2017) "Computational thinking's influence on research and education for all", *Italian Journal of Educational Technology*, 25(2), 7-14, link.httcs.online/wing17

73 link.httcs.online/casct

For a full explanation of CT, visit the Barefoot website,[74] the Teaching London Computing website[75] or attend a CPD session delivered by the National Centre for Computing Education (NCCE).[76] The key skills of abstraction, decomposition, pattern-matching and algorithmic thinking are vital for problem-solving with programs.

No taxation without…

Writing an algorithm down in some semi-formal way helps to clarify it, and we need a clear, precise algorithm before we can code it. The same algorithm can be written in many ways, and it's helpful to practise several ways of representing algorithms. Flowcharts can work for young learners as they are visual and easy to construct. Software such as Flowol and Flowgorithm helps learners to construct executable flowcharts, illustrating the link between visual representations and programs.

We can use pseudocode to express an algorithm more precisely. Pseudocode should not have a formal syntax, but unfortunately several "formal pseudocodes" now exist and cause confusion. I like the TechTarget definition:

> *"Pseudocode (pronounced SOO-doh-kohd) is a detailed yet readable description of what a computer program or algorithm must do, expressed in a formally-styled natural language rather than in a programming language."*[77]

Flowcharts and pseudocode are used in the software industry to specify programs. As they are language-independent, the same specification can be used to code programs in multiple languages.

Concepts, not constructs

Despite all this research telling us that programming is really problem-solving with code, and that the major skill learners need to acquire is not "coding" but "computational thinking", picking up an introductory programming textbook or using an online programming tutorial for a few minutes reveals a depressing fact: programming tutorials focus almost exclusively on syntax, not the process of developing a program to solve a problem. This issue has been around for almost 50 years, as this 1974 quote from David Gries attests:

---

74 barefootcomputing.org

75 teachinglondoncomputing.org/resources/developing-computational-thinking

76 teachcomputing.org

77 whatis.techtarget.com/definition/pseudocode

> *"Suppose you attend a course in cabinet making. The instructor briefly shows you a saw, a plane, a hammer and a few other tools, letting you use each one for a few minutes. He next shows you a beautifully finished cabinet. Finally, he tells you to design and build your own cabinet and bring him the finished product in a few weeks. You would think he was crazy!"*[78]

Most programming books are structured according to the language constructs – for example, the while loop and the IF statement. These are the tools of code, but the process of choosing the right tools, using them in the right order, putting together the pieces, then testing and refining the product is all too often "swept under the carpet as an embarrassing secret".[79]

Instead, we should focus on the process of programming, progressing through concepts in order of complexity of tasks, not complexity of language constructs. We should teach the programming process explicitly, encourage the development of a notional machine in the minds of the learners, and follow a use-modify-create model where learners progress "from consumer to producer".[80]

PCK for programming

We are lucky in that there has been a great deal of research in this area in recent decades. I am indebted to those researchers whose works are referenced in this chapter. Because computational thinking and programming are such unique skills, this section does not follow the layout of the PCK sections in the other chapters. Instead, I explain my thoughts in a more narrative style.

Developing CT skills through problem-solving

Computational thinking is a large part of programming, and we can develop learners' CT skills without asking them to write programs. The CS4FN,[81] CS Unplugged[82] and Bebras[83] websites all offer fun challenges to stretch CT skills. Teachers should identify and explain the CT skills being used, and encourage

78 Gries, D. (1974) "What should we teach in an introductory programming course?", *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education*, 81-89

79 Caspersen, M.E. (2018) "Teaching programming" in Sentance, S., Barendsen, E. & Schulte, C. (eds) *Computer Science Education: perspectives on teaching and learning in school*, Bloomsbury

80 Ibid.

81 cs4fn.org

82 csunplugged.org

83 bebras.uk

learners to explicitly name and describe the algorithms they create when solving the puzzles.

For example, in a CS4FN exercise such as "Searching to speak",[84] make sure you identify the processes of decomposition, pattern-matching and evaluation. When learners have a complete algorithm, ask them to write it down in a flowchart, pseudocode or natural English. Explain that the process they have gone through is computational thinking and describe its importance to programming.

Hacking the Curriculum by Ian Livingstone and Shahneila Saeed[85] is a goldmine of ideas to help learners master computational thinking, with and without computers. On page 149 of *Hacking the Curriculum* you will find the "Gamebook computing" activity, which describes the creation of an algorithm for solving a "choose your own adventure" book, all without computers.

Simon Johnson's *100 Ideas for Secondary Teachers: outstanding computing lessons*[86] includes 10 great computational thinking exercises, such as "Teaching with magic", "Human robot" and "Origami algorithms".

One algorithm, many representations

Each representation method (natural language, flowchart, pseudocode, program code) has strengths and weaknesses. Natural language is often easiest to express, so has a "low floor", while flowcharts help learners to visualise the algorithm, so that loops and selection become obvious paths through the algorithm chosen by evaluating conditions.

Pseudocode can free learners from worrying about syntax such as punctuation and how to indent, getting them close to the programmed solution. Practising different algorithm representations for the same algorithm can help learners see and understand the concepts we wish them to learn: sequence, selection and iteration.

The notional machine

When Peter Samson wrote his music player program for the TX-0 in 1959 (see chapter 4), his "mind had merged into the environment of the computer".[87] With the advent of high-level languages, however, the capabilities of the computer have been extended far beyond basic arithmetic and logic, to include abstract data

84 link.httcs.online/speak

85 Livingstone, I. & Saeed, S. (2017) *Hacking the Curriculum: creative computing and the power of play*, John Catt

86 Johnson, S. (2021) *100 Ideas for Secondary Teachers: outstanding computing lessons*, Bloomsbury

87 Levy, S. (1984) *Hackers: heroes of the computer revolution*, Doubleday

structures and complex algorithms to process them. It's no longer possible to hold in one's head the entire computer.

But to interpret, correct and write programs in a given language, we must have a good understanding of the capabilities of that language running on the machine in front of us. We say that learners must construct in their heads a "notional machine", described here by Juha Sorva:

> *"A notional machine is the set of capabilities that a programming environment affords to the programmer. Understanding a notional machine enables a programmer to answer questions such as: What can this programming system do for me? What are the things it can't or won't do? … What changes in the system does each of my instructions bring about as my program runs? How do I reason about what my program does?"*[88]

Without a notional machine in the mind of the learner, they don't have a viable model of program execution. They can't reliably predict what the computer will do with their code. This leads to misconceptions.

Misconception	Reality
Several lines of a program can be simultaneously active, especially a set of assignment statements	Programs are executed sequentially, top to bottom. Only one line of code is executed at a time, and it is not revisited (except through explicit control flow changes, such as loops or branches).

The computer can deduce the intention of the programmer from incomplete statements such as:
`score + 1`
 	
`if age >= 18`
`  "come in"`
`  else "not allowed"`
	Statements must be complete and syntactically correct, so the examples given (assuming Python is the teaching language) should be:
	`score = score + 1`
	
	`if age >= 18:`
	`  print("come in")`
	`else:`
	`  print("not allowed")`

Assignment statements, such as a = b, work in both directions, and either swap variables or make them always equal throughout the program execution	Assignment statements are made of two parts: the right-hand side of the assignment operator (=) is an expression that is evaluated, and the result stored in the variable on the left. Teaching this explicitly will prevent this misconception.

88 Sorva, J. (2018) "Misconceptions and the beginner programmer" in Sentance, S., Barendsen, E. & Schulte, C. (eds) *Computer Science Education: perspectives on teaching and learning in school*, Bloomsbury

Misconception	Reality
Meaningful variable names tell the computer what can be stored in them, so that, for example, smallest won't ever contain a value bigger than largest	Variable names are meaningless to the computer, so smallest can be given a value greater than largest.
IF statements are constantly active and will trigger whenever their condition is met	An IF statement is executed once; if the condition is true at the time of execution, the program branches. The statement then has no further effect on execution.
While loops terminate the instant the condition turns false, whatever statement causes this to happen	A while loop condition is evaluated on entry to the loop, and every time the code block inside the loop ends, not constantly during that code block execution.

Helping the learner construct a notional machine, and planning for misconceptions, will make teaching programming more effective.

Exposure to many examples

Often a learner may see a limited number of working examples of code before being asked to write their own. They may make assumptions about the language that are false and limit their ability. Some examples of misconceptions caused by limited exposure to variation are detailed here:

Misconception	Reality

Function arguments must be literals or constants (except where learners have seen special examples, such as the first line here)
 	
`r = int(input("radius?"))`
`a = area(r)`
`print(a)`
	The arguments of a function call can usually be any expression, including another function call. Therefore, this is valid for Python:
	
	`print(area(int(input(`
	`  "radius?"))))`

Comparisons must appear within a conditional expression only. They cannot appear elsewhere – for example, in return or assignment statements
 	
`if age < 18 or child == True:`
`  discount = True`
`else:`
`  discount = False`
 	
`if password == "summer" then:`
`  return True`
`else:`
`  return False`
	Booleans are computable values just like numbers, so they can be used in expressions on the right-hand side of assignment statements and in return statements just the same.
	
	`discount = age < 18 or child`
	
	`return password == "summer"`

Explicit live coding

Although daunting at first, this teaching method can be hugely effective. But what is live coding?

> *"Live-coding is an approach to teaching programming by writing actual code during class as part of the lectures. In a live-coding session, the instructor thinks aloud while writing code and the students are able to understand the process of programming by observing the thought processes of the instructor. … We found that live-coding (1) makes the process of programming easy to understand for novice programmers, (2) helps students learn the process of debugging, and (3) exposes students to good programming practices."*[89]

Live coding as a technique to teach programming is a good example of technological pedagogical content knowledge (TPCK) in action.

Technology	Using an IDE to code, run, debug and test a program. Projecting the results on to the whiteboard or display screen. Using a laser pen or cursor to draw learners' attention to the elements you wish to discuss. Using classroom management software to monitor their progress.
Content	Computational thinking and programming techniques you wish to impart to the students during the lesson, such as abstraction, decomposition, sequence, selection, iteration, subprograms, tracing and so on.
Pedagogy	Understanding the learners' prior knowledge and how to build on that. Checking for understanding. Responding to learners' questions and preventing misconceptions as they make progress.

You can watch a video of me live-coding a solution during remote teaching at httcs.online/prog.

Analogy and concrete examples

Analogies are valuable teaching tools. The building blocks of algorithms – sequence, selection, iteration and subprograms – can be taught with reference to other fields. For example, a recipe book includes algorithms that cover all these building blocks. Song lyrics and dance moves can be described algorithmically. We can go cross-curricular and write an algorithm for composing a haiku or a limerick, or one for "parsing" a poem to see if it fits a certain lyrical form.

89 Raj, A.G.S., Patel, J.M., Halverson, R. & Rosenfeld Halverson, E. (2018) "Role of live-coding in learning introductory programming", *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, 1-8

To make learning memorable, explaining new material in story form is often successful, as explained by Mark Enser in his book *Teach Like Nobody's Watching.*[90] A rich source of stories that explain algorithm concepts can be found in Jeremy Kubica's books, beginning with *Computational Fairy Tales*,[91] and you can find more on the Teaching London Computing website.[92]

Eliciting explanations

Encouraging learners to explain example code, and their own code, helps them understand it. Research shows that learners who self-explain outperform students who do not.[93] [94] Put simply, self-explanation is a "mental dialogue that learners have when studying a worked example that helps them understand the example and build a schema from it".[95] Have your learners explain sample code, either individually on paper, in pairs or in small groups verbally, or as part of a class discussion. There is even some evidence that encouraging learners to explain code to themselves, or to a proxy such as a plastic toy, aids understanding, hence the migration of "rubber duck debugging" from the software industry into the classroom.[96]

Cheat sheets

We want our learners to think hard about the new input we've given them and apply this to the problems in front of them. To be successful, we need to reduce their extraneous cognitive load, and this means freeing them from having to think about unnecessary things. For this reason, learners should have a "cheat sheet" of syntax in the language of choice, or a link to a website where they can look it up, such as W3schools.com. We do not wish them to be wasting time and cognition trying to remember the exact syntax of the while loop; it's far better that they think about the conditional expression that makes the loop perform properly and hence solves the problem.

---

90 Enser, M. (2019) *Teach Like Nobody's Watching: the essential guide to effective and efficient teaching*, Crown House

91 Kubica, J. (2012) *Computational Fairy Tales*, CreateSpace

92 teachinglondoncomputing.org

93 Chi, M.T.H., Bassok, M., Lewis, M.W., Reimann, P. & Glaser, R. (1989) "Self-explanations: how students study and use examples in learning to solve problems", *Cognitive Science*, 13(2), 145-182

94 VanLehn, K. (1996) "Cognitive skill acquisition", *Annual Review of Psychology*, 47, 513-539

95 Clark, R., Nguyen, F. & Sweller, J. (2006) *Efficiency in Learning: evidence-based guidelines to manage cognitive load*, Wiley

96 link.httcs.online/duck

Pair programming

Having one learner "drive" and another "navigate" is a way to share the cognitive load between two learners and add some fun to the process. This is another example of a practice borrowed from the software industry – this time from the agile development approach. Pair programming in the classroom is explained in a "short read" blog post from the NCCE.[97]

Chunking (aka subgoal labelling)

Make good use of chunking in application tasks, where the learner is applying the knowledge they have just learned (this should be the next most complex task, not just the next code construct). Chunking involves breaking tasks down and explicitly stating the next success criterion; Mark Guzdial calls this process "subgoal labelling".[98]

For example, if coding a quiz game, instead of simply asking learners to write the complete game, the first subgoal could be "ask a single question and check the answer". The next subgoal might be "add a second question". The third could be "code a scoring mechanism", and the fourth could be "add a loop so the player gets three tries". Celebrate success at the completion of each subgoal.

Parsons problems

Writing a piece of code, breaking it into pieces and asking the learners to reconstruct the program is known as a Parsons problem. This is another way of limiting extraneous cognitive load. It tests learners' understanding of statements or constructs, and how they fit together, while removing any requirement to worry about syntax.

97 Robinson, J. (2019) "Engaging learners through pair programming", Teach Computing, link.httcs.online/pair

98 Guzdial, M. (2020) "Subgoal labelling influences student success and retention in CS", Computing Education Research Blog, link.httcs.online/subgoal

```
username = initial + last
```

```
first = input("firstname?")
last = input("lastname?")
```

```
print(username)
```

```
initial = first[0].upper()
```

Figure 2.3: A Parsons problem.

An example can be seen in figure 2.3, where the learner is required to put the statements in the correct order to assemble a program that creates a username from the inputs of first and last names.

Parsons problems are described in detail on Mark Guzdial's blog[99] and there are lots of Parsons problems in the online programming courses at Runestone Academy.[100]

Block coding

Another way to release working memory for solid computational thinking is by freeing learners from syntax altogether. Block-coding environments can do this, like Scratch, Blockly, Alice and App Lab. It's important that you use this opportunity to teach structured programming, however. Too many learners claim to "know Scratch", but a 2018 analysis discovered that three-quarters of all Scratch projects contained no selection or iteration, making them effectively just animations.[101]

A project by UK teenager Joshua Lowe, called EduBlocks,[102] provides a drag-and-drop block-coding environment for Python, which can be used to ease the transition from blocks to text.

Physical computing

Physical computing – combining software and hardware to build interactive physical systems that sense and respond to the real world – has been shown to

99 Guzdial, M. (2020) "Proposal #1 to change CS education to reduce inequity", Computing Education Research Blog, link.httcs.online/parsons

100 runestone.academy

101 Grover, S. (ed.) (2020) *Computer Science in K-12: an A to Z handbook on teaching programming*, Edfinity

102 edublocks.org

result in broad engagement across a spectrum of users.[103] However, to be successful, the teacher needs to be familiar with the technology; they also need to accept that some time will be invested in teaching the learners to use it, and some time will be lost to setting up before any meaningful programming can be done. For that reason, unless you have the space to leave your Raspberry Pis set up, or can afford integrated screen and keyboard cases like the Pi-top, you might wish to save the Pis for an after-school club.

Some products are easier to learn than others. The top five in order of complexity are the Makey Makey, Crumble, micro:bit, Arduino and Raspberry Pi. A wealth of resources is available online for all these, including on raspberrypi.org and codeclub.org. The BBC micro:bit offers a good trade-off between complexity and opportunity, and the supporting Microsoft MakeCode website[104] offers a natural transition from blocks to text-based programming.

PRIMM

It seems obvious, but we didn't learn to write before we could read. It's odd, then, that we ask children to write program code almost as soon as we introduce it to them. A better approach is to ensure plenty of exposure to working code, as part of the use-modify-create methodology (see page 51). Students should start with code that someone else has written for them, which reduces the emotional strain caused by failing; it also ties in with Seymour Papert's "low floor" approach (see page 13) and Barak Rosenshine's seventh principle of instruction: "obtain a high success rate".[105]

Research by a team including Sue Sentance at King's College London found success with a structure called PRIMM.[106] Taking use-modify-create, adding the prior stage "predict" and expanding the later stages gives us predict-run-investigate-modify-make.

---

103 Hodges, S., Sentance, S., Finney, J. and Ball, T. (2020) "Physical computing: a key element of modern computer science education", *Computer*, 53(4), 20-30, link.httcs.online/physical

104 makecode.microbit.org

105 Rosenshine, B. (2010) "Principles of Instruction", International Academy of Education, link.httcs.online/rosenshine

106 Sentance, S. And Waite, J. (2017) "PRIMM: Exploring pedagogical approaches for teaching text-based programming in school", *WiPSCE '17: Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, 113-114

Stage	Activities	Why
Predict	In pairs, look at a piece of code and ask students what they think it will do.	This activity encourages students to look for clues in the program that suggest what its function is.
Run	Run and check against your prediction. NB: have code in a shared area, don't make students waste time copying the code.	Opening prepared code moves the weight of ownership of any errors from the student to the teacher, increasing confidence and avoiding a challenging exercise to students who struggle with literacy at any level.
Investigate	There are lots of different activities you can do at this stage: trace the code, comment it, answer questions about it, label concepts, draw the flow of control, etc.	It takes many activities of this type, repeated in different forms in different lessons, for students to grasp the concepts in a secure way. We may think that writing one selection statement correctly means they have a good understanding of selection, but really "getting" this takes some time.
Modify	Starting with working code, students are challenged to add modifications, beginning simply and increasing in difficulty.	The transfer of ownership moves from the code being "not mine" to "partly mine" as students gain confidence. This provides the scaffolding that students need to add small snippets of code and see their effect within a bigger program.
Make	Once confident, students can create their own program from scratch, like the example but their own design.	Design of a new program is an important skill and should start with planning and trying to construct a suitable algorithm. This is difficult, but gives students an opportunity to be creative and have the satisfaction of making their own program.

Read more about PRIMM at link.httcs.online/primm and primming.wordpress.com.

The block model

During the "I" phase of PRIMM, while investigating the code, students should be encouraged to ask questions about it to deepen their understanding. You can prompt them with questions such as:

- What would happen if you swapped lines 2 and 3?
- What would happen if you gave it input of ___?
- What if you change the symbol on line 5 from > to < ?
- Line 5 shows a condition-controlled loop – why do we call it this?
- What will make the loop end?

We can check we're encouraging valuable thought across the whole range of programming skills using an approach called the block model. Devised by Carsten Schulte in 2008, the block model is described by Sue Sentance in an article for *Hello World* magazine:

> *"The Block Model is a grid with two axes — one showing the size of the programming element under consideration, and the other the distinction between the structure of the program, the execution of the program, and the function of the program."*[107]

If we map our questions and activities on to the block model, we can identify any gaps. Adding more tasks in those gaps will ensure that we cover the whole grid. In this way, we make sure students are thinking hard about the full range of skills required to thoroughly understand a program.

THE BLOCK MODEL

(M) Macro structure	Understanding the overall structure of the program text	Understanding the algorithm underlying a program	Understanding the goal/ purpose of the program in the current context
(R) Relationships	Relationships between blocks	Sequence of function calls, object sequence diagrams	Understanding how subgoals are related to goals
(B) Blocks	Regions of interest that build a unit (syntactically or semantically)	Operation of a block or function	Understanding of the function of a block of code
(A) Atoms	Language elements	Operation of a statement	Function of a statement
	(T) Text surface	(P) Program execution	(F) Function
	Architecture/Structure		Relevance/Intention

Figure 2.4: The block model provides a framework for program comprehension tasks.

Assessment

It's tempting to set a programming challenge and judge the final result; this is the assessment method of choice in some published schemes of work. However, novice programmers have a lot to learn, and each learning objective is a valuable step on the way to producing a whole working program of any size independently. Also, the goal should not be to judge progress and detect misconceptions at the end of the course, when it's too late to rectify them.

107 Sentance, S. (2020) "The I in PRIMM", *Hello World*, 14, link.httcs.online/hw14block

Ongoing formative assessment using misconception-sensitive methods should be our goal. Diagnostic questions are a valuable tool. A diagnostic question is a multiple-choice question that includes "distractors" (wrong answers) that match potential misconceptions.

Using our example from the table on page 53 – "Assignment statements, such as a = b, work in both directions, and either swap variables or make them always equal throughout the program execution" – we could ask this question:

> What would be the value of apples after this sequence of statements?
>
> ```
> apples = 3
> lemons = 6
> apples = lemons
> lemons = 8
> ```
>
> Answer:
>
> A) 3 B) 6 C) 8

The correct answer is, of course, B. The value of apples is set to 6 by the third line and is unaffected by the fourth line. If the learner answers C, they may be harbouring the misconception we were testing for: that this fourth line affects apples because of the "equation" in line 3 that "made apples and lemons equal".

If many learners answer C, we can address this misconception with the whole class. We can also elicit explanations from the class of why they chose their answers, which can be very illuminating. At time of writing, more than 7,000 diagnostic questions are freely available at diagnosticquestions.com/quantum, which were crowdsourced by Project Quantum, a collaboration between Computing at School and several assessment bodies. The website can be used to set formative and summative tests, and it even provides a field called "explain your answer" to elicit those explanations.

Conclusion

Programming is not about code, it's about solving problems. The process of designing a program is the hard part, often called computational thinking, and developing CT skill is where we teachers should spend our time. In all things we should consider cognitive load. We should make sure learners are thinking hard about what matters – getting better at designing programs using CT – and not about working out where the punctuation goes. Teaching programming is a tough gig, but there is a wealth of PCK at our disposal already – engaging with it should make you a more successful teacher.

Remember, you are engaged on a Grand Challenge. Programming is akin to learning a new language, yet typically we see our students for only an hour a week, so turning out programmers of any ability is an amazing achievement. Engaging with the latest research can maximise our success and produce programmers who will go on to use computation to unlock yet more of the secrets of the universe.

Chapter 3.
Robust programs

Summer 1968, Massachusetts Institute of Technology

Margaret Hamilton heads to MIT's Lincoln Laboratory to continue work on the flight software ultimately used for the Apollo space missions. This evening, as she often does because of scarce childcare, Hamilton brings her young daughter, Lauren, to work. While Lauren plays with the test rigs, Hamilton is programming. She and her colleagues are writing code for the Apollo Guidance Computer (AGC), inventing new ideas in computer programming as they go.

Lauren is "playing astronaut" with the test computer keyboard, hitting random keys until, to her delight, the simulation starts. But within minutes it has crashed: Lauren has selected a pre-launch program when she was already "in space", and the computer has wiped the navigation data. Hamilton is shocked, but is denied permission to alter the program to prevent this scenario from occurring on a real mission.[108] When the "Lauren bug" happens for real during the Apollo 8 mission in December 1968, Hamilton is authorised to add validation code to prevent this type of crash in future.

20 July 1969, six miles above the moon

The Apollo 11 mission is 102 hours old. The launch site at Cape Canaveral, Florida, is a quarter of a million miles behind. Michael Collins remains in the command module, 20 miles above Neil Armstrong and Buzz Aldrin, who are

108 Corbyn, Z. (2019) "Margaret Hamilton: 'They worried that the men might rebel. They didn't'", *The Guardian*, link.httcs.online/hamilton

descending to the lunar surface. Armstrong must control the braking thrust, avoid surface boulders and ensure a soft landing for the lunar module. But just seven minutes before touchdown, with fuel running low, an alarm sounds on the AGC. Four further alarms sound over the next few minutes, but Armstrong safely pilots the module past a boulder field. He declares, "Houston, Tranquillity base here. The Eagle has landed."

The software team at MIT can't relax – a malfunction during the return ascent to the command module could be catastrophic. The cause of the alarms must be located before the scheduled lift-off in 12 hours' time. Working through the night, they discover that the "rendezvous radar" was flooding the AGC with data, causing the computer to repeatedly restart. They realise the alarms are a feature, not a bug. The 1201 and 1202 alarms that sounded during descent were actually advice to the astronauts fired by Hamilton's recovery routine, named BAILOUT1, clearing an overloaded CPU and reloading only the top-priority programs, thus saving the mission.

Hamilton's innovations would later be described as "robust programming". Her work not only saved the first manned lunar mission, but launched a whole new discipline called "software engineering".

Figure 3.1: Margaret Hamilton pioneered the discipline of software engineering while working on the Apollo moon missions.

The software crisis

Computing power grew rapidly throughout the 1960s, as did demand for computer systems. Academic researchers, manufacturing, retail and finance were crying out for the new "ultimate labour-saving device", but with precious few programmers, the industry was in trouble. Here is Edsger Dijkstra's view in 1972:

> *"The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."*[109]

At the first NATO Software Engineering Conference, which took place in Germany in 1968, Friedrich L. "Fritz" Bauer, co-creator of the ALGOL language (see chapter 4), described the dire situation as a "software crisis". The complexity of new systems and lack of experienced developers meant software projects now routinely ran over time and over budget; programs were inefficient and of poor quality, and in some cases never delivered at all. Some companies were brought to the brink of collapse by the failure of their computing projects, for want of an effective software division.

Mother of invention

The crisis drove the computing industry to develop novel solutions to the "software quality" problem. New programming languages were invented, such as Pascal and Ada, which encouraged structured programming. Whole new paradigms appeared (see chapter 4). Functional programming was designed to reduce coding errors called "side-effects", and object-oriented programming encapsulated data and functions inside objects to protect them from unexpected changes.

The Chapel Hill Symposium at the University of North Carolina in 1972 resulted in the book *Program Test Methods*,[110] the first book to focus on problems in testing and validation, rather than the development of programs. And at the end of the decade, *The Art of Software Testing* by Glenford Myers[111] set the stage for "modern" software testing, which Myers defined for the first time as a process "with the intent of finding errors". No longer was testing an attempt to declare a program bug-free.

Early programmers simply debugged their own programs, but this informal practice was not scalable to the huge projects that got under way in the 1970s. The development process was formalised and rigorous testing replaced informal

109 Dijkstra, E.W. (1972) "The humble programmer", *Communications of the ACM*, 15(10)

110 Hetzel, W.C. (ed.) (1973) *Program Test Methods*, Prentice-Hall

111 Myers, G.J. (1979) *The Art of Software Testing*, Wiley

debugging. Programmers were encouraged to write quality code, plan for contingencies and make their programs foolproof – a technique we sometimes call "defensive design". Programmers wrote extensive test plans during development, not after. Testing was spun off as a separate discipline and, by the mid-1980s, formal software development methods were published that hugely raised the profile of the planning and testing phases.

Waterfall goes viral

The "waterfall" methodology split a software project into five key phases: requirements gathering, design, implementation, testing and maintenance.

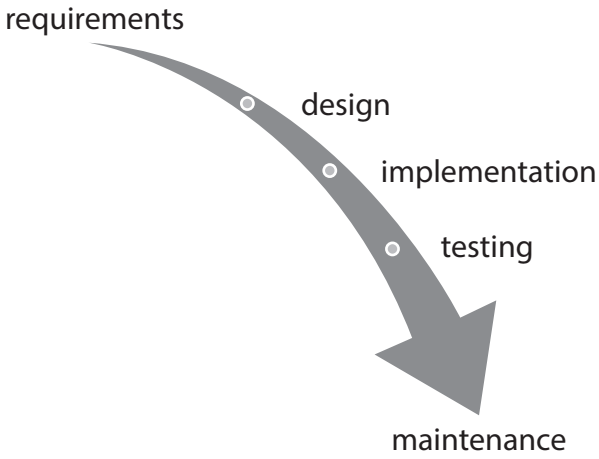


Figure 3.2: The waterfall method of software development was rigorous but inflexible.

The US Department of Defense, having selected Ada as a common structured programming language (see chapter 4), formally mandated the waterfall methodology through the 1988 military standard DOD-STD-2167A, bringing it to global attention. Waterfall went viral. Test plans would be drawn up during the design phase and often carried out by a separate team, resulting in the new job description of "software tester", a role that now demands more than £40,000 p.a. in the UK.

Dedicated testing teams would perform black-box testing, meaning they would treat the software as a black box into which they could not see. Testing would determine if the product behaved as expected: did it produce the expected output for a given input? This is distinct from white-box testing, a term later coined to

describe testing with detailed knowledge of the code – a method known to early programmers simply as debugging.

More agility needed

By the 1990s, some companies were finding the waterfall methodology too cumbersome and expensive for managing the software development life cycle (SDLC). Some projects had nothing to show for years of investment, never reaching the bottom of the waterfall, because the method could not support rapidly changing customer requirements. According to a *TechRepublic* blog post, "Very often, customers don't really know what they want up-front; rather, what they want emerges out of repeated two-way interactions over the course of the project."[112]

New iterative methods were needed that gave the developers more flexibility and made a quick return on investment more likely. Various methods evolved, including rapid application development (RAD) and extreme programming (XP), which are known collectively as agile methodologies. In agile, a project is broken into short bursts, delivering working software at the end of each burst. Bugs are inevitable, and the development and testing teams work together closely to resolve them. Agile can accommodate changing requirements more easily and it always delivers a working product.

Testing times

As programs grew in size and complexity, their test plans grew too. The cost of testing a complex program soon overshadowed its development costs, reaching on average 40% of total IT spend by 2018.[113] Even testing a small change would result in a huge testing bill, as all affected features were retested to ensure they still worked properly – a process called "regression testing". Of course, people were writing code to test other code: in 1985, Linda Hayes's company AutoTester, based in Texas, released the first commercial test tool for PCs running MS-DOS. Today, the automated testing software market is estimated to be worth $8.5 billion.[114]

---

112 Contributor Melonfire. (2006) "Understanding the pros and cons of the Waterfall Model of software development", *TechRepublic*, link.httcs.online/waterfall

113 Saran, C. (2015) "Application testing costs set to rise to 40% of IT budget", *Computer Weekly*, link.httcs.online/testcost

114 *DEVOPSdigest*. (2018) "Automation testing market worth 19.27 billion USD by 2023", link.httcs.online/testingmarket

Inevitable bugs

Despite all these efforts to improve code quality, defects in program code are inevitable. In the critically acclaimed developer's manual *Code Complete*,[115] Steve McConnell estimates that, in the average project, up to 500 errors are made in every 10,000 lines of delivered code. Even the highly formal and costly "cleanroom development" method results in one bug every 10,000 lines. The Android operating system contains more than 10 million lines of code, while Windows has well over 50 million.

Thus, our objective of robust programming is not to eliminate but to reduce the number of bugs, while making sure those that remain cannot cause catastrophe. Indeed, back in 1972, Dijkstra said: "If you want more effective programmers, they should not waste their time debugging, they should not introduce the bugs to start with."[116] The Dutch pioneer was actually building on research undertaken by John von Neumann and Alan Turing in the 1940s into formal specification and verification of computer programs.

Sadly, the history of software is too often the history of software-caused catastrophes. To illustrate the importance of robust programming, it's worth taking a look at some of the most high-profile software failures in history.

Software catastrophes

Therac-25 radiotherapy accidents

Between 1985 and 1987, the Canadian-built Therac-25 radiotherapy machine malfunctioned at least six times, giving patients massive overdoses of radiation – sometimes hundreds of times greater than planned. Three patients died and three experienced life-changing injuries. The manufacturers initially blamed the hospitals, but eventually admitted there were serious flaws in the software.

Therac-25 was the first machine in the series to have full software control of the dosage; earlier machines had hardware dials. In the Therac-25, concurrent programming errors, also known as "race conditions", made it possible to select the wrong mode of radiation. If the operator typed "x" for X-ray treatment on the control terminal, then used the cursor-up key and corrected this to "e" for electron-beam therapy within eight seconds, the machine would report "Malfunction 54". Unfortunately, malfunctions happened so frequently that the operator would simply reply "p" to proceed after every warning. In the case of Malfunction 54, this

115 McConnell, S. (2004) *Code Complete: a practical handbook of software construction* (second edition), Microsoft Press

116 Dijkstra, E.W. (1972) "The humble programmer", *Communications of the ACM*, 15(10)

left the machine set for a much stronger X-ray dose, while the electron-beam gun was in fact pointed at the patient.

The accidents starkly illustrated the dangers of software-controlled safety-critical systems and became a case study in software engineering. Partly because of the Therac-25 accidents, in 1995 the International Electrotechnical Commission recommended software safety standards for medical equipment.

Patriot missile rounding error

One of the most serious failures of robust programming resulted in the deaths of 28 American soldiers in the First Gulf War. An Iraqi Scud missile – usually an easy intercept for the Patriot missile defence system – evaded US defences owing to poor coding. The Patriot targeting computer attempted to calculate the time in tenths of a second by multiplying the system clock by 0.1. However, binary cannot exactly represent the decimal 0.1, so the computer makes an approximation, and the fewer bits you have to play with, the less accurate the approximation.

> As an aside, you can try this yourself – it makes for a superb classroom discussion. In the Python Shell, or in any Python online IDE (Integrated Development Environment), just type 0.1 + 0.1 + 0.1 and watch your students' eyes light up.

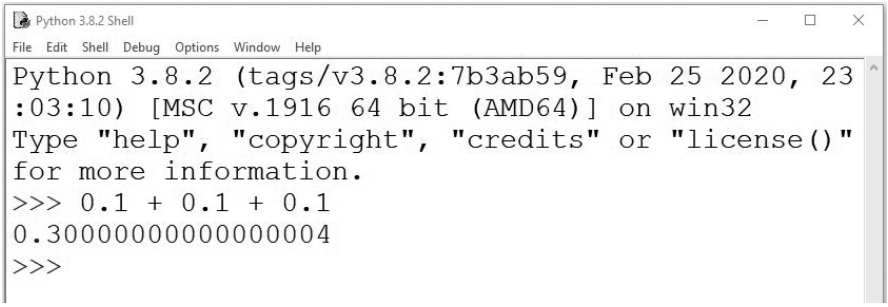


Figure 3.3: Decimals must be approximated in binary, leading to rounding errors.

The Patriot targeting computer used just 24 bits to represent 0.1, which meant every calculation was out by 0.000000095 seconds. These errors were cumulative, so after 100 hours of uptime the targeting clock was out by a third of a second. That might not seem a lot, but a Scud missile can travel 600 metres in that time, so when the targeting computer saw empty sky where a missile should be, it failed to launch, and 28 souls perished in February 1991.

143 million customer records are out there

In 2017, a data breach at credit-scoring agency Equifax leaked personal information on more than 143 million Americans and 400,000 UK citizens, putting them at risk of identity theft for years to come. The attacker used SQL Injection, first documented by "white hat" hacker Jeff Forristal in 1998.[117] In this attack, often abbreviated to SQLI, specially crafted input is executed as code by the web server, with the goal of damaging or stealing data from the backend database. We will look at SQLI in more detail in chapter 10.

We defend against SQLI by sanitising our inputs to remove any unexpected characters. This doesn't sound difficult, but breaches similar to the Equifax break-in have occurred with alarming regularity in the two decades since the flaw was documented, including data theft from Adobe, eBay, LinkedIn and Yahoo. Despite its age, SQLI remains the most common security flaw reported by the Open Web Application Security Project (OWASP).[118] The ease with which SQLI can be defeated by input sanitisation makes its frequency soul-destroying to cybersecurity managers.

What could go wrong?

Freddie Williams and Tom Kilburn's 1948 Small-Scale Experimental Machine, also known as the Manchester Baby (see chapters 1 and 6), had just one kilobyte of storage. The average small-to-medium business now stores around 48TB of data.[119] Algorithms churn this data, making decisions that are important to their customers, including credit checks, medical supply orders and insurance claims. If the data is inaccurate, customer service is affected, people may be disappointed or even harmed, and the reputational damage can be enormous. Preventing dirty data from entering your systems in the first place should be the primary aim. Keeping your data accurate and consistent is called data integrity; it's achieved via sanitisation, authentication and validation.

Validation means checking that the input is reasonable, not that it's correct: we can't know a user's age, but they can't be more than 150 years old. Validating inputs can prevent "downstream" errors that could be catastrophic, such as charging someone 100 years' worth of interest or prescribing the wrong medicine.

---

117 Kerner, S.M. (2013) "How was SQL Injection discovered?", *eSecurity Planet*, link.httcs.online/forristal

118 link.httcs.online/owasp

119 Guta, M. (2020) "67% of small businesses spend more than $10K a year on analytics", *Small Business Trends*, link.httcs.online/bizdata

Authentication means verifying someone's identity before granting access to your system, and access controls ensure they have the correct authority before allowing data to be accessed or changed. For example, counter staff in a bank might be able to raise transactions of up to £10,000 but be prevented from moving higher amounts. And they certainly should not be able to run raw SQL commands that break the system, such as "DROP TABLE Customers". These checks would not only prevent data-integrity errors by rogue staff, but also by hackers taking control of the account or the terminal.

Implementing sanitisation, validation, authentication and access control is often described as "anticipating misuse". This is a key feature of defensive design – writing programs that won't fail despite user error or malicious intent.

Move fast and break things

While the story of computing is sometimes called the "third industrial revolution" (after steam-powered machines and the invention of the production line), it's also the history of spectacular mistakes. Some were down to poor program design or inadequate testing; most were driven by a desire to deliver the product quickly and cheaply. Over time, however, the computing industry has learned at great cost the value of building robust systems through defensive design techniques and rigorous testing. Margaret Hamilton's BAILOUT1 routine, which saved the Apollo 11 mission, remains a lesson from history that resonates today.

TL;DR

Early programmers designed and debugged their own programs. Building in code to prevent failures due to user error or hardware failure was pioneered by Margaret Hamilton for the Apollo space programme. Her work led to the creation of a new discipline: software engineering, popularised by a NATO conference in 1968. At the same conference, Friedrich L. "Fritz" Bauer coined the term "software crisis" to describe a critical shortage of programmers. New techniques and tools were created throughout the 1970s to address the crisis and improve software quality, including new languages that encouraged structured programming, and new paradigms such as functional programming and object-oriented programming.

Testing began to be recognised as separate from debugging in the 1970s, with Glenford Myers publishing *The Art of Software Testing* in 1979.[120] The software development life cycle (SDLC) was formalised in the 1980s and the waterfall

120 Myers, G.J. (1979) *The Art of Software Testing*, Wiley

model became commonplace after the US Department of Defense adopted it in 1988. The waterfall model described several distinct project phases: requirements gathering, design, implementation, testing and maintenance. Software testing became a separate discipline performed by a different team to the developers.

Industry found the waterfall model unresponsive to changing user requirements, and iterative techniques, often known as "agile", grew popular in the 1990s. Many companies began to employ test automation software as testing consumed a larger part of the IT budget.

Modern robust programming includes anticipating misuse through authentication, sanitisation and validation, plus a formal development methodology such as agile, structured programming techniques focused on modular, maintainable code, and a rigorous testing regime.

PCK for robust programs

Core concepts

- Defensive design that anticipates misuse, including:
  - Validation – checking the input is reasonable, e.g. not allowing letters in a phone number.
  - Verification – checking the input is what the user intended, e.g. "Are you sure you want to delete your account?"
  - Sanitisation – stripping invalid characters from an input field to prevent injection attacks.
  - Authentication – forcing a user to prove their identity before allowing them access or elevated privileges, usually with a password.
- Maintainability of code: comments, indentation, meaningful identifiers.
- Code reviews.
- Testing:
  - Types of test: unit, functional, system.
  - Phases of test: iterative and final.
  - Test methods: black-box and white-box.
  - Test plan including purpose of test, test data and expected outcome.
  - Trace tables.
- Syntax errors and logic errors.

Fertile questions

- Can we make our programs foolproof?
- What proportion of a computing project should be testing?
- What are all the ways in which computer systems can fail? Consider:
  - Self-driving cars.
  - The Mars rovers, such as Perseverance and Curiosity.
  - A social media app.
- Which is more effective: black-box or white-box testing? Why?

Higher-order thinking

Why did it fail?

Share the software error stories from this chapter with your learners. Ask them to discuss what might have gone wrong. Explain the SDLC and ask learners to decide in which phase the errors crept in (there is more than one right answer). If they had been in charge of the software, what would they have done differently?

Was Dijkstra right?

Discuss Dijkstra's statement that we "should not introduce the bugs to start with". Ask learners how this can be achieved. Is the choice of programming language and paradigm important? What about separating programming from testing?

Peer testing

Ask the learners to code a program and write a test plan for it, then swap seats and test their neighbour's program. Is this more effective than testing their own program? Try this again, but deliberately introduce errors into the program before testing to see if the tester finds them.

Robotic verification: yes or no?

Specify a program in a flowchart or pseudocode, then write some code for it. Without running the code, have the learners check it against the specification to prove that the program will work. Ask the learners if they think this can be automated, and how. If not, why not?

Validate everything?

Discuss the "Lauren bug" introduced at the start of this chapter. Explain that, initially, NASA was reluctant to allow Margaret Hamilton to add the code to prevent this bug, claiming that no highly trained astronaut would make that mistake.[121] Of

121 Corbyn, Z. (2019) "Margaret Hamilton: 'They worried that the men might rebel. They didn't'", *The Guardian*, link.httcs.online/hamilton

course, Jim Lovell went on to press the wrong key on the Apollo 8 mission. Should we validate every input, even those we are not expecting?

Cross-topic and synoptic

Cross-topic with architecture

Computer systems are made of hardware and software. We learn in the architecture topic that hardware has a limited reliable life and it eventually breaks down. When we move controls from hardware to software, as in the design of the Therac-25, we overcome potential hardware issues, but at what cost? Ask the students if they would prefer a manual dial or a computer program to control their radiotherapy dose, and why. What would make them trust the software?

Cross-topic with languages

High-level code was designed to help programmers create complex code more easily. But we learn in the languages topic that low-level code is often used for mission-critical programs because it can be made more robust. Why is this?

Cross-topic with issues and impacts

What are the ethical issues of poor-quality software? Clearly it can kill, but what is the impact of poor-quality code on the end user of the following products:

- Medical equipment.
- Computer games.
- A shopping website.
- A social media app.
- A bank.

Industry experts suggest that 25-40% of a project should be spent on testing. Is this reasonable? What kind of projects can get away with less than this? What would require more?

Translation programs are now widely used to help everyone from travellers to researchers understand text in a foreign language. What would be the impact of translation errors in:

- A piece of school homework.
- An official document such as a trade agreement.
- An instruction document for a piece of medical hardware (like a radiotherapy machine).

Cross-curricular

Cross-curricular with design and technology

Discuss the design process that learners meet in design and technology. How does "iterative design" relate to software development?

Cross-curricular with science

Some branches of science rely heavily on computer models. These can be enormously complex – for example, a weather model or a model of particles in a nuclear reactor. How can we ensure these are robust?

Analogy and concrete

Vending machine woes

Consider a vending machine that dispenses drinks and snacks. Have the learners think of all the ways that the system could fail, and how we could design it defensively to mitigate failures. Learners could create a table listing possible failures and their mitigations:

Failure	Mitigation
Mechanical failure while vending	Sensors recognise the failure and refund the money.
Customer walks away mid-vend	Machine times out and resets for the next customer after two minutes of inactivity.
Insufficient coins in the machine to give accurate change	Warns customer that no change is available before vending, and ensures they are OK with this.
Power failure while vending	Sets a non-volatile flag when beginning to vend. Saves the values of credit and choice of product in non-volatile storage. On power-up, checks the state of this register and resumes vend if flag set.

Impenetrable code

Show the learners some code that is poorly written, without meaningful variable names, indentation or comments. Ask them to explain what the code does. When they have attempted this, show them the same code written in a maintainable way and ask the same question. Draw out conclusions from this exercise about the importance of maintainable code. Craig'n'Dave provide such an activity in their premium resources (see figure 3.4, on the next page).

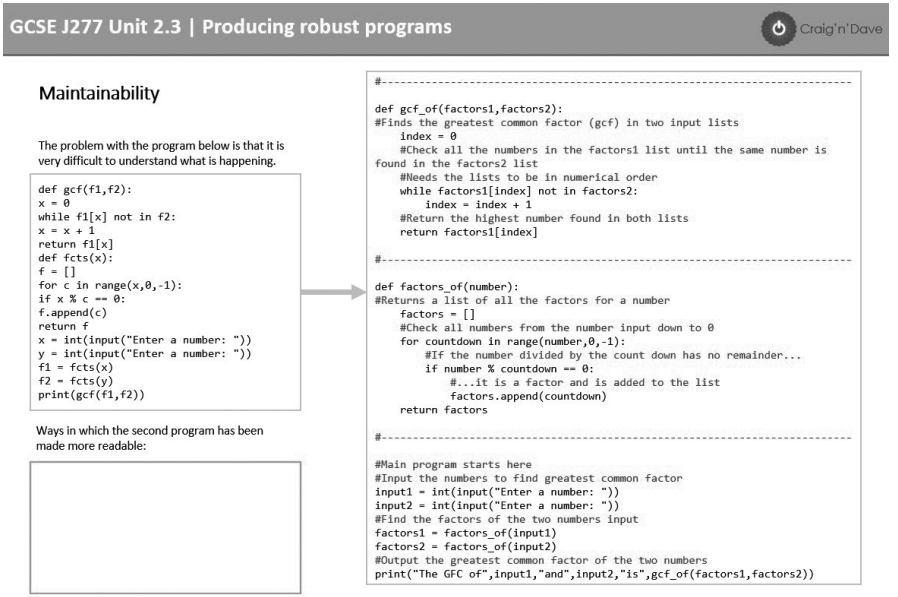


Figure 3.4: An exercise in code maintainability from the Craig'n'Dave premium resources.

Unplugged

Computer programs can be debugged and "tested" through a "dry run" walkthrough: the programmer or tester looks over the code and determines what will happen at each line. Sometimes they will use a trace table to keep track of the contents of variables. Performing walkthroughs of algorithms while unplugged helps the learner develop their understanding of the code, and builds a mental model called the "notional machine" (see chapter 2).

Physical

Using the Raspberry Pi, micro:bit or similar, learners can develop products that use physical devices. Many projects can be found at projects.raspberrypi.org. Testing is a vital part of any project and should be made explicit in the process.

Project work

Robust design and effective testing should be part of all the project work that learners undertake. Iterative testing and final testing should be built into projects such as large "makes" and substantial programming problems.

Misconceptions

Misconception	Reality
Validation is "checking the input is correct"	Validation is "checking the input is reasonable". Learners confuse validation with verification, sanitisation and authentication. These processes must be clearly explained and contrasted.
We only need to test the code we have changed	When adding or changing code, learners will often test the new section of code, without considering the rest of the program. Testing the unchanged code to ensure that its behaviour has not changed is called regression testing and it should be encouraged.

We test for expected results only. For example, after writing this code to test for vowels
`l = input("letter?")`
`if l in "aeiou":`
`  print("vowel")`
`else:`
`  print("consonant")`
novices may test with single vowels only, or single lowercase letters only, or single mixed-case letters only
	Testing should cover a sample of all possible input values including normal, boundary and erroneous data (also known as valid, extreme or invalid).
	Learners often assume that the point of testing is to check that the program performs as expected when expected data is entered. They often don't consider the possibility of erroneous data being entered – for example, letters in the wrong case, input having the wrong data type or a value out of range.

Black-box testing is performed by hackers: confusion with black-hat hacking	Black-box and white-box testing are often confused with black-hat and white-hat hacking, with students believing that black-box testers are malicious. In fact, black-box testing simply means treating the program as a black box into which we cannot see, therefore we check outputs against those expected for given inputs.
Debugging and testing are only necessary because we are novice programmers	Learners often assume that they need to debug and test their code because they are novices, and when they become more experienced they will write flawless code first time. It's important to stress that debugging and testing is a vital part of developing new programs, and experienced programmers do a lot of it! The teacher live-coding in front of the class, then debugging and testing, is very valuable here.

Chapter 4.
Languages and translators

Massachusetts Institute of Technology, 1959

A computer is playing Bach. The programmer, Peter Samson (he likes to be called a hacker), is pleased with the result and is now working on a general-purpose music decoder that will read in codes from paper tape and play the corresponding notes on the computer's speaker. These are groundbreaking programs: the first use of a musical code to produce sounds in real-time, bringing to life Ada Lovelace's idea from 1843 (see chapter 1). But how did we get here?

The story starts in the spring of 1959, when MIT offers a new course called programming taught by John McCarthy. McCarthy has recently established a controversial new field of study, naming it artificial intelligence, and caused a stir by programming MIT's million-dollar IBM 704 mainframe to play chess.

With around 18 kilobytes of RAM and able to perform 12,000 calculations per second, the valve-built 704 is impressive but working with it is frustrating. Programs are loaded from punched cards and processed in batch from start to finish. Output appears on the printer if all goes well, while any bugs require the programmer to start all over again. This "batch processing" concept makes working on the 704 slow and tedious. But things are about to change.

MIT's military research department, known as the Lincoln Lab, has donated its $3 million TX-0 to McCarthy's faculty. The "Tixo" has less memory than the 704 but several huge advantages: transistors, a display, a speaker, and a new typewriter-like input device called a Flexowriter. Thanks to its silicon innards,

the new machine can execute 100,000 calculations per second. Most importantly to Samson, a hacker can sit at the terminal and see and hear the effects of their program – and even modify it while sitting there.

The speaker is designed not as an output device, but to assist the operator. It simply clicks to indicate the contents of the 14th bit of the accumulator (see chapter 5), so if it stops clicking, that means the program has finished or, if you're unlucky, entered an infinite loop. But Samson sees a way to exploit this feature to make music. A simple program feeding the accumulator with the right sequence of data will cause the speaker bit to turn on and off at the correct frequency to play a note, albeit a simple, square-waved note. Push lots of these numbers into the register in the right sequence and you have a tune. Samson teaches the Tixo to play classical music in just 4K of memory, and the hackers of MIT are impressed.

Figure 4.1: Peter Samson and Dan Edwards in 1962 playing *Spacewar!*, often considered the first computer game.

Assembly line

Like all programs written for the Tixo, Samson wrote his music player in assembly code.[122] An assembly code instruction is written in short keywords called mnemonics, and each one consists of an opcode and an operand. The opcode is the task and the operand the thing being operated on. Every CPU has a finite instruction set of opcodes that it can process. Assembly code gives a simple one-to-one mapping from opcode to binary: one assembly code instruction represents one machine code instruction. A simple program called an assembler turns the code into binary using this one-to-one mapping.

A small program to add two numbers using the LMC Instruction Set (see chapter 5) might look like this:

```
INP     # input number into accumulator
STA 99  # store contents in RAM location 99
INP     # input number into accumulator
ADD 99  # add contents of RAM location 99
OUT     # output contents of accumulator
```

This program is simple, but the programmer must decide where in memory to store the data. Anything more complex, like a program to play a tune, would run to hundreds of lines and require a good understanding of the computer's internal workings: its registers, buses and memory. As Steven Levy explains in *Hackers*:

> *"When you programmed a computer you had to be aware of where the thousands of bits of information were going from one instruction to the next, and be able to predict – and exploit – the effect of all that movement. When you had all that information glued to your cerebral being, it was almost as if your mind had merged into the environment of the computer."*[123]

Clearly, this hacker-level immersion in the bits and bytes of a computer is beyond most mortals, but fortunately abstraction has an answer. To add two numbers, instead of writing INP, STA 99, INP, ADD 99, OUT, what if we could just write English instructions and let the computer decide what binary codes are required?

One floor below the Tixo, McCarthy is doing just that on the 704. His chess program is coming along nicely in a mixture of assembly code and Fortran. This

122 link.httcs.online/tixocodes

123 Levy, S. (1984) *Hackers: heroes of the computer revolution*, Doubleday

early high-level language was developed by a team at IBM, led by John Backus, with the aim of making mathematical computation easier.

Coding in English

Here's that "add two numbers" program in Fortran:

```
read *, a, b
s = a + b
print *, s
```

This should look familiar if you've seen Python or JavaScript. But, as we'll see in chapter 6, the computer doesn't understand anything; it can't read this code, much less comprehend it, so how can we run this program? The answer is translation. The high-level source code passes through a translator program called a compiler, which generates the binary machine code, also known as object code.

Fortran was not the first high-level language, but it was the first commercial success. The 1960s saw an explosion of high-level languages including ALGOL, COBOL and LISP.

Compilers were treated with suspicion in the early days, partly because the code they produced was not well optimised: it contained more instructions than an expert would write when "hand-coding" the program. This happens because we no longer have a one-to-one relationship between an instruction written by the programmer and the binary code that the CPU can process. The compiler must generate multiple machine code operations for each line of high-level code, and it may not choose the exact sequence of operations that an expert low-level programmer would choose. This drawback is known as the abstraction penalty.

The first high-level programming language was Plankalkül, created by Konrad Zuse, a German civil engineer who worked on his pioneering computer designs almost entirely alone in his parents' flat during the Second World War. He designed the world's first programmable computer, the Z3, which became operational in May 1941.

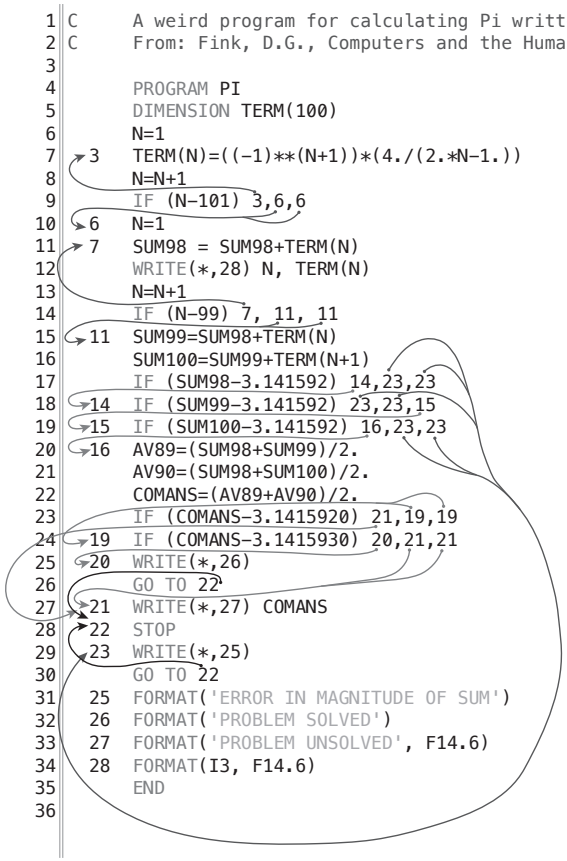
Fortunately, because compilers are themselves programs, they can be improved. Compiler optimisation has been the subject of millions of hours of research since Zuse's Plankalkül. Modern compilers create highly optimised code, making high-level programming the obvious choice for all but a handful of specialist applications. Embedded computers in military hardware, medical devices and transport systems require optimised, ultra-reliable code, which makes them candidates for low-level programming.

ALGOL, short for algorithmic language, was designed between 1958 and 1968 by a committee including Backus and McCarthy, plus other giants of computer science mentioned in this book, including Peter Naur, Fritz Bauer and Alan Perlis. In addition, many variants of ALGOL were created, including versions by Edsger Dijkstra and Niklaus Wirth.

ALGOL was designed from the beginning to promote structured programming; it was the first high-level language to include code blocks delimited by BEGIN and END, selection with IF-THEN-ELSE, conditional iteration with WHILE and control structures for procedures. These features made their way into all subsequent imperative languages.

The GOTO heresy

Low-level programmers achieved selection or branching – causing the program to make a choice between two paths – through JUMP instructions. A JUMP instruction would tell the computer to start executing instructions from a different memory location, by changing the program counter directly. Loops would simply be created by conditionally jumping back to an earlier instruction. Many high-level languages, such as Fortran, replicated this feature through the GOTO statement. But large programs written with many GOTO statements can be hard to follow and even harder to debug, as figure 4.2 illustrates on the next page.



```
C     A weird program for calculating Pi written in Fortran.
C     From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.

      PROGRAM PI
      DIMENSION TERM(100)
      N=1
3     TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
6     N=1
7     SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
11    SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
14    IF (SUM99-3.141592) 23,23,15
15    IF (SUM100-3.141592) 16,23,23
16    AV89=(SUM98+SUM99)/2.
      AV90=(SUM98+SUM100)/2.
      COMANS=(AV89+AV90)/2.
      IF (COMANS-3.1415920) 21,19,19
19    IF (COMANS-3.1415930) 20,21,21
20    WRITE(*,26)
      GO TO 22
21    WRITE(*,27) COMANS
22    STOP
23    WRITE(*,25)
      GO TO 22
25    FORMAT('ERROR IN MAGNITUDE OF SUM')
26    FORMAT('PROBLEM SOLVED')
27    FORMAT('PROBLEM UNSOLVED', F14.6)
28    FORMAT(I3, F14.6)
      END
```

Figure 4.2: Using GOTO indiscriminately leads to "spaghetti code".

In 1968, Dijkstra wrote an influential essay, published as a letter to the editor of *Communications of the ACM* and entitled "Go to statement considered harmful". He said: "The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."[124]

Dijkstra had many supporters who despised the GOTO statement, and the phrase "spaghetti code" emerged in the 1970s to describe the overuse of such one-way branches. However, Donald Knuth, author of the authoritative work on structured

124 Dijkstra, E. (1968). "Go to statement considered harmful", *Communications of the ACM*, 11(3), 147-148

programming, *The Art of Computer Programming*, disagreed. In 1974, he noted that in a small number of cases, "structured code often is twenty to thirty percent less efficient than equivalent code with gotos".[125]

The GOTO debate raged for two decades, with the statement appearing in some later high-level languages including the C family, but not Java or Python. The Usenet discussion forum for the C programming language, *comp.lang.c*, notes in an FAQ document published in 1990, "Many programmers adopt a moderate stance: GOTOs are usually to be avoided, but are acceptable in a few well-constrained situations, if necessary."[126] The GOTO statement remains in the C family of languages.

Structured code – it's the future!

The standard control structures pioneered in ALGOL (code blocks, IF-THEN-ELSE, WHILE-DO and procedures) should be familiar to even novice programmers today. We use them for program flow control using sequence, selection, iteration and subroutines. It is possible for a programmer in an ALGOL-like procedural language, such as C or Python, to easily pick up how to code in another, such as Ruby or C#, because the basic structure of the language – the building blocks with which programs are constructed – remains the same. Only the syntax changes.

Syntax is the grammar of a language. It defines what is legal code: the keywords, punctuation and structure of programs. Code can have correct syntax, but still not compile or run correctly if it contains semantic or logic errors. These words have been borrowed from natural language, where the meanings are similar – there is a common syntax.

Wordy number cruncher

In parallel to academic strands of language development was the business-oriented field. In 1959, the computer scientist and naval officer Grace Hopper attended a meeting of commercial computer users called CODASYL, set up to create a programming language for business that could process large files of data quickly and required little computing knowledge. The result was the Common Business-Oriented Language, or COBOL.

125 Knuth, D.E. (1974). "Structured programming with go to statements", *Computing Surveys*, 6(4)

126 link.httcs.online/cfaq17.10

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. hello.
PROCEDURE DIVISION.
  DISPLAY "Hello World!".
  STOP RUN.
```

The ubiquitous "Hello World" program in COBOL.

COBOL borrowed heavily from Hopper's previous language, FLOW-MATIC, which she had created in 1952 for the UNIVAC mainframes at the manufacturing company Remington Rand, inventing the compiler at the same time. COBOL was a notoriously wordy language, designed for batch processing of financial data for tasks like billing or insurance quotes. But it was easy to learn and had powerful input/output capabilities. Defining the data structure separately to the procedural code was a masterstroke, making COBOL portable – the same code could run on many different machines. COBOL compilers were written for all the major mainframes of the day, including IBM, Burroughs, UNIVAC and Honeywell. COBOL was to dominate the business software market for more than three decades, and code written in COBOL still runs on many finance and utility company mainframes today.

More importantly, Hopper did the heavy lifting in the field of compiler design, giving us the many stages of compilation: lexical analysis, syntax analysis, code generation and linking.[127]

The academic's choice

```
begin
  if a>=b then
    max := a
  else
    max := b
end;
```

A fragment of code in Pascal (1970) to find the higher of two numbers. This syntax should look familiar to Python programmers.

127 Read more about Grace Hopper at Isaac Computer Science: isaaccomputerscience.org/pages/grace_hopper

ALGOL committee member Niklaus Wirth wrote a variant in the 1960s called ALGOL W, but when this version was rejected by the committee, he forked it and added strong typing and complex data types, such as linked lists, graphs and trees. He released this as Pascal in 1970. Wirth intended Pascal to be a general-purpose language for commercial, scientific and educational purposes. The language was popular in education and the mini-computer software market throughout the 1970s, but was supplanted by C when Unix became popular in the 1980s (see chapter 8). An object-oriented version called Delphi was commercially successful well into the 21st century.

By 1975, the number of programming languages in use in embedded systems worried the US Department of Defense enough that it formed a working group to find a new all-purpose language. The ambitious project resulted in the creation of Ada, first released in 1980 and still in use today in transport systems, military hardware and space technology.

BASIC instinct

Beginners' All-Purpose Symbolic Instruction Code (BASIC) was first developed in 1964 by John Kemeny and Thomas Kurtz at Dartmouth College in New Hampshire, US, to allow students to write code for mathematics. Versions were produced for the new mini- and micro-computers that sprung up throughout the 1970s, with Bill Gates's version for the hobbyist's Altair computer at the centre of the first big software piracy dispute.[128]

The BBC's Computer Literacy Project, which ran from 1982 to 1989 (see chapter 2), was centred around BBC Basic running on a machine designed by Acorn Computers in Cambridge, UK. BASIC influenced the design of Microsoft's Visual Basic, which was widely used to build Windows applications in the 1990s; a version called VBA is still bundled with Office for writing macros.

C for miles

Dennis Ritchie wanted to create utilities for the Unix operating system he had created with Ken Thompson (see chapter 8), and cast around for a high-level language to code them in. Fortran was not up to the task, so Ritchie took and simplified BCPL, software designed by Martin Richards at the University of Cambridge for developing compilers.

Ritchie called his simplified version "B", but it was slow and limited, so he upgraded it and C was born. The obvious next step was to rewrite the whole Unix

128 Barton, M. and Stedman, C. (2008) "Timeline: the Gates era at Microsoft", *Computerworld*, link.httcs.online/gatesletter

operating system in C, and in 1973 Version 4 of Unix was compiled entirely from C source code.

C provides structured programming features borrowed from ALGOL, but is a hackers' programming language providing low-level access to memory. C programmers can allocate, modify, and free memory directly. This makes C powerful, but also dangerous: buggy or malicious code can have disastrous effects if the wrong portion of memory is altered. More forgiving languages like Python manage memory on behalf of the programmer.

C is a "static-typed" language, meaning that variables and functions must be declared with a data type that may not change. This ensures that programs are internally consistent, and the compiler will return an error if code has a data-type mismatch. As a result, logic errors are less likely in C than in loosely typed languages like Python. Descendants of C include object-oriented versions Objective-C and C++, and a proprietary Microsoft version called C# designed for use on the web and mobile platforms.

Cookie-cutter coding

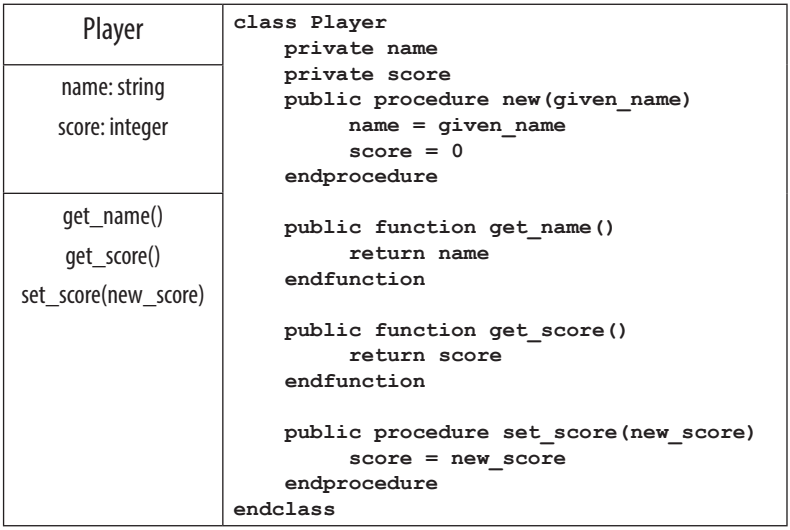
In 1962, the Norwegian developers Ole-Johan Dahl and Kristen Nygaard released Simula, a programming language developed from ALGOL 60 to run computer simulations. The integrated circuit design technique, called Very Large Scale Integration (VLSI), required extensive use of simulations and Simula was written for this purpose. Dahl and Nygaard later added features to make the language more useful and, in 1967, Simula 67 was the world's first object-oriented language. Object-oriented programming (OOP) is concerned with the definition of classes and objects.

A class is like a blueprint, sometimes called a "cookie-cutter" for objects, which are instances of a class. The object behaves according to its attributes and methods. A programmer simply defines all the classes of object and how they behave, rather than writing long chains of instructions. Object-oriented programming is well-suited to games development, simulation and neural networks.

Pure class

Inspired by Simula, a team funded by the Advanced Research Projects Agency (ARPA) at Xerox PARC created Smalltalk, an early OOP language that was highly influential. The team, led by Alan Kay and Adele Goldberg, didn't just create a new programming language. They broke new ground in "human-computer symbiosis", with interactive time-shared computers, graphics screens with overlapping windows and a pointing device we would recognise as a mouse. The

influence of Smalltalk can be seen in modern OOP languages including C++, C#, Java and Python (yes, Python can be used as a simple procedural language, but it also supports OOP!). Below is an example of a class definition, shown both in Unified Modelling Language (UML) and in the OCR exam reference language (formerly known as pseudocode). This class could be used to define player objects in a game.[129]



Well, I declare!

While ALGOL was the model for all subsequent imperative languages, LISP was the foundation for a family of languages in a different paradigm. In imperative languages we write commands for the computer to perform. Just as phrases in the English imperative mood, such as "come here", give an instruction, imperative code is a sequence of instructions for the computer. The focus is on control flow, and programs consist of sequence, selection and iteration. Fortran, ALGOL, Pascal, C++, Python and JavaScript are all imperative languages.

A subset of declarative programming, called functional programming, describes functions that evaluate expressions. Functional languages include LISP, Erlang,

129 See Isaac Computer Science for a tutorial on OOP: isaaccomputerscience.org/concepts/prog_oop_fundamentals

Clojure, Scheme and Haskell. They have no variables, only identifiers for values contained within expressions. Programming without variables is disconcerting at first, but learners soon get used to it, and Haskell is the language of choice for advanced learners.

```
fac :: (Integral a) => a -> a
fac 0 = 1
fac n = n * fac (n - 1)
```

A Haskell program to calculate factorial.

Functional programming avoids side-effects. Each function always returns the same value for the same parameters passed to it, and nothing else changes. This makes behaviour highly predictable and less prone to error. Because functions are so versatile, there is always an elegant way to get the job done, meaning functional programs tend to be shorter, with the solution much closer to the problem.

Learn you a Haskell

After a few functional languages showed promise, a conference was held in Portland, Oregon, in 1987 to agree on an open standard, and three years later Haskell was born. It gained popularity with the creation of the Glasgow Haskell Compiler (GHC) in 1991 by Cordelia Hall, Will Partain and Simon Peyton Jones (who was appointed chair of the UK's newly created National Centre for Computing Education in 2019). Haskell is used widely in academia, but also for analysing financial risk at the Dutch bank ABN AMRO, image-processing at *The New York Times* and transforming digital music into musical notation in the Chordify app. Programmers who have tried a bit of Haskell find that it helps them write more efficient code even when they return to imperative languages.[130]

130 link.httcs.online/haskell

Prolog

In another family of declarative languages, we simply make statements about relationships between objects. In 1972, two researchers at Aix-Marseille University in France, Alain Colmerauer and Philippe Roussel, wanted to formally describe the French language. They collaborated with Robert Kowalski of the University of Edinburgh to create the first logic programming language, Prolog.

In a logic language we create logic statements or predicates to describe facts about the world. Querying the relationship between these facts produces useful output (see above). The language lends itself to AI applications such as voice response or expert systems.

```
likes(ryan, cheese).
likes(ryan, wine).
not(likes(lila, wine)).
likes(inez, wine).
likes(inez, X) :- likes(X, wine).

?- likes(ryan, wine).
 yes.
?- likes(inez, ryan).
 yes.
?- likes(inez, lila).
 no.
```

A Prolog program is a set of predicates and queries.

Code once, run anywhere

Back in the imperative world, and before 1995, for your program to work on multiple devices you would need multiple compilers, creating separate object code for each machine. Java turned this concept upside down. Initially planned for the digital cable television industry, Java was perfect for the growing web application market in the late 1990s.

Named for designer James Gosling's favourite coffee bean, Java is compiled only once, to an intermediate code called Java bytecode. This portable bytecode is then interpreted at runtime directly on the platform, by a Java virtual machine (JVM). Each device needs the JVM pre-installed; the code itself would be written just once, and compiled just once, but be runnable anywhere. This gave Java a portability

advantage over C++. It quickly became the language of choice for web apps, and later smartphone apps.

The web goes interactive

Not to be confused with Java, JavaScript was developed to add interactive elements to Netscape Navigator, the most popular browser on the World Wide Web in the late 1990s. Originally called LiveScript, the name was changed shortly after release, possibly to jump on the Java bandwagon, and the similar names have been causing confusion ever since.

Microsoft initially reverse-engineered JavaScript for its Internet Explorer (IE) browser, calling its version Jscript, but added full JavaScript support later, after the Firefox and Chrome browsers hit IE market share. JavaScript is a core technology of the World Wide Web and supported by all major web browsers, but it's also a programming language in its own right. It's used in web and mobile apps, and as an instruction language in many schools and universities.

Although JavaScript is the language of choice for the browser-side code, Perl has grabbed a huge share of the server-side market since its specification in 1993. Server-side code is executed on the web server in response to an HTTP request from the browser, and is used to serve up dynamic web pages –for example, when retrieving data from a database. Perl is an easy-to-learn, well-supported, interpreted scripting language. Python creator Guido van Rossum once admitted that if Perl had been compatible with the Amoeba system he was working on at the time, he would never have created Python.

Joyful coding

The Zen of Python is a collection of guiding 19 principles published on the Python mailing list in 1999 by software engineer and major Python contributor Tim Peters. It includes these principles:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Readability counts.
- There should be one – and preferably only one – obvious way to do it.

Van Rossum launched Python in 1991 (he was a big fan of the British sketch show *Monty Python's Flying Circus*). He wanted his new language to be fun as well as simple, flexible and highly extensible. Extensibility means anyone can write code libraries to add features to the language. For simplicity, Python has very little punctuation

compared with its contemporaries, such as C++, Java and JavaScript. Python forces readability by using indentation to show where code blocks begin and end. Gone are the BEGIN and END statements of ALGOL, which had already morphed into curly brackets { and } in C++, Java and JavaScript. In Python, we just begin a code block by indenting it, and we end it by no longer indenting. This implicit delimiting of code blocks can cause problems for novice programmers, but it makes for neat code.

Python is an interpreted language, which means each line is translated to machine code just before execution. Interpreted languages are slower than compiled languages, but by the late 1990s this didn't matter much, as computers were so powerful. The advantage of C++'s small and efficient object code is outweighed by the simplicity and shorter development time of an equivalent Python program. By the turn of the century, the cost of a programmer's time was greater than the cost of spinning up a few more servers to cope with the extra CPU cycles. And because Python is interpreted, with a free interpreter available for every platform, it is highly portable. Just like Java, Python is "code once, run anywhere".

UK schools using C++, Java or VB gradually moved over to Python, and one exam board, Edexcel, now offers a Python-only programming exam. The official Raspberry Pi OS image[131] includes Python (indeed, the "Pi" in the name is short for Python[132]), and most of the coding resources listed on the Computing at School website[133] are Python-related. This is no accident, as Python is fun and easy to learn. As Van Rossum noted:

> *"The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code – not in reams of trivial code that bores the reader to death."*[134]

High-level languages were invented to be more English-like, yet many developed a very non-English syntax full of odd punctuation and structure. Python remains close to English and therefore easy to learn. The last word goes to Peter Norvig, director of research at Google and AI expert:

> *"In Perl you have to be an expert to correctly make a [complex program]. In Python, you have to be an idiot not to be able to do it, because you just write it down."*[135]

131 raspberrypi.org/software

132 Vilches, J. (2012) "Interview with Raspberry's founder Eben Upton", *TechSpot*, link.httcs.online/eben

133 computingatschool.org.uk

134 Costa, C.D. (2021) "Top 13 resources to learn Python programming", *Towards Data Science*, link.httcs.online/joypython

135 Norvig, P. (2000) Usenet group *comp.lang.functional*, link.httcs.online/norvig

TL;DR

Early programming meant manually entering binary codes. Assembly language gave us mnemonics, but we still needed to understand the architecture. The high-level languages Fortran, ALGOL and COBOL abstract away the code from the architecture, letting us use the English keywords IF, WHILE and FOR. A compiler translates this code into machine code, but we need one for every computer system, and because each high-level statement generates multiple machine code instructions, the object code might not be optimal.

Sequence, selection, iteration and subprograms are no accident, but a natural feature of algorithms, and are used in all imperative languages descended from ALGOL, including Pascal, C and Python. Structured code is preferable to "spaghetti code" because it's readable and easy to maintain. C was written by the creator of Unix, and the "hackers' language"evolved into C++ and C#, but its fussy syntax means it's tricky to learn. Universities first taught using Fortran, then Pascal, but in 1981 UK schools went with BASIC as it was already popular on home computers. BASIC inspired Visual Basic, popular in the 1990s and still used today.

Object-oriented programming, beginning with Simula and Smalltalk, gave us classes that are blueprints for objects; each object has attributes and methods and can interact with other objects. This programming style is useful for games programming, but also for modelling, simulation and AI applications. Java was the most popular language from around 2000-2015 because it was object-oriented and portable; it could run anywhere thanks to a Java virtual machine created for every popular platform.

Imperative languages like Java, C and Python execute a sequence of instructions. Another paradigm is declarative programming, which comes in two types: functional languages like LISP, Scala and Haskell, and logic languages like Prolog. A functional language describes merely how data is processed by functions. This often results in more elegant code with fewer bugs, and is suited to scientific analysis and AI applications.

Code in an interpreted language is translated to machine code line by line, instead of compiled all at once. Perl, JavaScript and Python are all interpreted, allowing rapid coding and testing, which meets the demands of the modern software market. JavaScript is built into browsers to make web pages interactive. Python was launched in 1991 and designed to be fun, simple and flexible. Its clean syntax made it a popular choice for teaching programming. According

to Paul Dubois, co-creator of the NumPy library, Python is the "most powerful language you can still read".[136]

An understanding of this topic begins with the knowledge that, at its heart, a computer is just a collection of logic circuits that process digital signals of high and low voltages, representing zeros and ones. The circuits are able to decode certain patterns of zeros and ones, and we call these bit patterns "instructions". Each CPU responds to a finite set of these "low-level instructions" – its machine code instruction set.

Coding in binary is difficult and error-prone, so each binary code is given a short, memorable name or mnemonic, such as **`LDA`**, **`SUB`** or **`BRA`**. These mnemonics are known as assembly language, as they were first used to assemble the binary codes needed to instruct the computer. Assembly language is still difficult to code and contains no useful constructs, such as loops or arrays, so high-level languages were invented. High-level languages are more English-like and give us lots of features to allow us to write complex programs very quickly. Python, Java, JavaScript, VB.NET, C, C++ and C# are popular high-level languages. Assembly language may still be used where compact code is essential, or for small, mission-critical programs, because the code it produces is extremely robust.

High-level code must be translated into machine code before it can be run on the CPU. For this we need a translator, and these come in two types: compiler and interpreter. Compilers translate the whole high-level source-code program into machine code, creating an executable file of object code. Interpreters translate the program one line at a time, which allows for rapid coding and debugging but slower execution than compiled code.

Software containing lots of useful developer features, called an Integrated Development Environment (IDE), is usually used to develop code. IDLE, PyCharm and Mu are popular IDEs for Python; Visual Studio supports C, C++, C#, VB.NET and JavaScript; and developers might use Eclipse, NetBeans or IntelliJ IDEA for Java. An IDE provides many features to speed up coding, such as syntax-checking, autocomplete, stepping, breakpoints and variable-tracing.

136 link.httcs.online/dubois

PCK for languages and translators

Core concepts

- CPU instruction set.
- Low-level languages:
  - Machine code.
  - Assembly.
- High-level languages.
- Choice of high- or low-level language for a purpose.
- Translators:
  - Compiler – advantages and disadvantages.
  - Interpreter – advantages and disadvantages.
- Purpose and features of an IDE.

Fertile questions

- Why do interpreted languages run slower than compiled languages, and why does this matter less today?
- Why do video games have a format, such as PC, Xbox, PlayStation?
- Why are there so many programming languages?
- Can a spreadsheet full of formulas be considered a computer program?
- Why did Java become so popular?

Higher-order thinking

Choice of language for a purpose

Ask learners to choose a language with which to code an application for a given purpose. Purposes could include weapons guidance, an online shop, a mobile game or a new school information system. This exercise tests whether students have mastered the relative merits of compiled versus interpreted languages, high-level versus low-level coding, and the development and support models of open-source versus proprietary systems.

Specifying syntax, or coding a parser

Ask learners to describe the syntax of their most familiar programming language. This can be done informally, either by using English descriptions such as "an assignment statement is written <variable-identifier> = <expression>", or as a block diagram. More advanced learners could be given formal tools with which

to express syntax, such as Backus-Naur form (BNF), and could even write the beginnings of a parser to check the syntax of a statement in a given language.

Comparing paradigms

Even if paradigms are not on the specification, looking briefly at a functional program and asking how it differs from a similar imperative version can help learners understand how programs are constructed. Compare a functional implementation of factorial (see page 92) with a typical imperative one. What are the advantages of each? How might errors creep into the imperative version that could not be coded in the functional one?

Cross-topic and synoptic

Cross-topic with system software, programming, issues

Explore Python's history with the class. Python is an open-source language; its runtime, interpreter and all libraries are freely available online and can be used without charge. Contrast with closed-source compilers such as C#.

Cross-topic with architecture, memory, networks, issues

Discuss what makes a programming language successful. Look at trends in programming languages driven by new platforms such as web and mobile. Consider the effect of Moore's law on processor power and cost of memory. How have these trends influenced language popularity over time?

Cross-curricular and analogy

Cross-topic with languages

The terms "syntax" and "semantics" were borrowed from the study of natural languages. We can help our learners by explaining their wider meanings using analogy.

- "Wistful purples: when the staring ;wild' imaginative, attentively fabrics!" consists entirely of valid features of English but is gibberish. It does not follow the grammar rules, so we say it has invalid syntax.
- "Colorless green ideas sleep furiously" is grammatically correct but has no meaning. This sentence was devised by the linguist and cognitive scientist Noam Chomsky in 1955 to illustrate the difference between syntax and semantics.
- "Let's eat Grandma" and "Let's eat, Grandma" are both syntactically correct but have different semantic senses. One is clearly not the author's intended meaning! Novice programmers often misuse punctuation, operators or delimiters, changing the semantics of a program.

Here's a Python example of this issue:

`if age > 18:`
`  print("come in")`
`else`
`  print("no, sorry!)`
	`if age >= 18:`
	`  print("come in")`
	`else`
	`  print("no, sorry")`

This code will output "no, sorry!" if the age entered is exactly 18, because of a semantic or logic error.	This code will behave as expected, with the addition of a single equals sign. Both code samples are syntactically correct.

After explaining the terms using analogy, ask learners to write English sentences with syntax errors and with semantic errors. Then they can create program code examples and compare the two, to better understand the meanings of "syntax" and "semantics".

Misconceptions

Misconception	Reality
Computers just "understand" code	To run a high-level language, you need a translator program, either an interpreter or compiler, which creates machine code for that computer. The binary machine code is still not "understood" but rather processed by unintelligent logic circuits.
A program is always a set of instructions	This is true of imperative languages, but not true of declarative languages such as LISP, Prolog and Haskell. Unfortunately, we usually teach this misconception as a "necessary evil" from primary school, as the programming paradigms concept is too abstract for that level of learning. In preparation for A-level, we might "unteach" it with brief exposure to functional programming in KS4 (Years 9 and 10).
Python is the only language, or all computers understand Python	Many languages are available. Lack of exposure to different languages causes this. It's a good idea to look briefly at other languages in KS4, e.g. VBA within Excel if your school has Microsoft Office, JavaScript within App Lab at code.org, or the JavaScript W3Schools tutorial.
Java = JavaScript	Explaining the history of both languages should clear up this misconception.
Translators reside in ROM	Translators are system software utilities. Learners often don't realise that compilers and interpreters are system software, and usually not supplied with the operating system but downloaded and installed on secondary storage when needed.

Chapter 5. Algorithms

Too clever by half

"… bad fortune dogged them from that day forward, the village was destroyed seven times by fire, and visited seven times by the king's vengeance. So in time it came to pass that the people fell into a wretched plight. They concluded that there must be some breeder of misfortune among them, and resolved to divide into two bands. This they did; and there were then two bands of five hundred families each. Thence-forward, ruin dogged the band which included the parents of the future Losaka, whilst the other five hundred families throve apace. So the former resolved to go on halving their numbers, and did so, until this one family was parted from all the rest. Then they knew that the breeder of misfortune was in that family, and with blows drove them away."

An extract from the Jataka, an ancient Buddhist text.[137]

Abstractions all the way down

In her excellent book *Hello World*, Hannah Fry explains that all algorithms fall into four main categories: prioritisation, classification, association and filtering.[138] Prioritisation algorithms choose the best series of chess moves from the billions of possibilities, suggest the most likely pages you'll want when you search the web, or even determine a "breeder of misfortune" in your community. Classification

137 Chalmers, R. (translator). (1895) *The Jataka: volume I*. link.httcs.online/jataka

138 Fry, H. (2019) *Hello World: how to be human in the age of the machine*, Transworld

algorithms label and remove inappropriate content on YouTube, or recognise obstacles in front of a self-driving car. Association algorithms match people in dating apps or recommend what to buy next based on previous shopping habits. Filtering algorithms remove noise from sound recordings or stop spam from filling up your inbox.

Finding a route using a mapping service such as Google Maps is an example of prioritisation. This is sometimes called optimisation: the optimal selection of roads from a vast number of choices. Google Maps owes its success to the programming pioneer Edsger Dijkstra. For A-level computer science we study the algorithm that bears his name: a pathfinding algorithm guaranteed to find the shortest route between two points.

Dijkstra's algorithm works by first checking all the possible roads from the start position to the next choice of road, which can be a junction or exit. The length of each of these road segments is recorded and then the process is repeated, keeping a running total for each route so far. If any two routes meet on the way, only the shortest route to that waypoint will be retained – all longer routes will be discarded. Once the destination has been reached, only one "winning" route remains.

Of course, Dijkstra's algorithm works on geographical information stored in a digital form. There is no fleet of drones flying around measuring the routes for us when we input our query. The algorithm works on an abstraction of the road network, a simplified diagram that mathematicians call a graph. The graph consists of nodes, edges and edge weights, all stored as numbers in a data structure. Dijkstra's algorithm is designed perfectly to process this abstract data structure, working through the numbers and prioritising some routes over others to find the optimal path.

Abstracting the data from the real world into a suitable data structure, and then writing a matching algorithm to process it, are the key skills at the heart of programming. You want a program to detect cancer cells in biopsy samples? Get the image data into a suitable abstraction and then write a program to process the data, detecting the patterns of numbers that match indicators for cancer in the image. You want a voice assistant to respond every time you say "Alexa"? Encode the sound data in a suitable abstraction and use an algorithm to detect patterns that match the "Alexa" wave. In each case, the first part of the process is data abstraction, which is not just "removing unnecessary detail". Specifically, it involves retaining *just the details required to answer questions about the data with a suitably matched algorithm*.

Algorithmic thinking is required for the second part of the process, which again begins with abstraction. When humans choose a route between two cities, we

tend to carry out lots of extraneous processing, such as "Is the route scenic?", "Am I familiar with it?" and so on. Dijkstra simply abstracted away all these human elements and wrote an algorithm that focuses only on the important parts of the process: checking the distances between nodes and keeping a running total for each route so far. Abstraction has left us with *just the processes required to solve the problem using the abstract data structure*.

One bite at a time

"There is only one way to eat an elephant: a bite at a time" – proverb

This proverb, popularised by Desmond Tutu, describes another computational thinking skill: decomposition. A routing algorithm finds and joins together multiple short sections to create the complete route. Any sufficiently large problem can be broken down into smaller sub-problems to make it easier to solve. When planning a holiday, for example, you might choose a hotel, book a flight and arrange an airport transfer. Tackling these three sub-problems solves the big problem of getting a holiday booked.

Nobody taught the German mathematician Carl Friedrich Gauss decomposition, but he was able to use it to great effect from a young age. Gauss (1777-1855) is sometimes referred to as the "Prince of Mathematics" for his prolific contributions to the subject. The process of destroying data on disks or tapes with strong magnetic fields is still called "degaussing" because of his work on magnetism.

According to anecdote, Gauss was punished by his primary school teacher with a task the teacher thought would take the boy some time: add the numbers from 1 to 100. The young Gauss was able to compute the sum in a matter of seconds. How?

Gauss had spotted a pattern. Instead of performing 1 + 2 + 3 + … + 98 + 99 + 100, if he added together the smallest and the largest number, then added together the next numbers inwards, and so on, he would get identical results. For example:

$$1 + 100 = 101$$

$$2 + 99 = 101$$

$$3 + 98 = 101$$

There are 50 such sums, so the answer is just 50 x 101 = 5050. Gauss had decomposed a large problem into 50 trivial ones, making the whole problem simple to solve.

Sum of its parts

Charles Babbage's 1820s Difference Engine (see page 21) relied on decomposition to calculate polynomials (mathematical expressions containing x raised to various powers). To illustrate how this worked, let's take a quick look at the simplest such

function, $f(x) = x^2$. The results of this function for each value from 1 to 7 are (1,4,9,16,25,36,49).

We now calculate the difference between each pair of consecutive terms, calling this the first-order difference. If we then calculate the difference between each difference, which we call the second-order difference, we see that this is now the constant value 2.

x	$f(x) = x^2$	First-order difference (this term minus the previous term) d(x)	Second-order difference (this difference minus the previous difference) constant 2
0	0		
1	1	1	
2	4	3	2
3	9	5	2
4	16	7	2
5	25	9	2
6	36	11	2
7	49	13	2

So, to calculate any term f(x), we can just take the previous term, f(x–1), add the first difference for that term, d(x–1), and then the constant difference of 2:

$$f(x) = f(x–1) + d(x–1) + 2$$

Now calculating 52 is a simple matter of addition. The previous term is 42=16. Then we add the previous first difference of 7, making 23, and then the constant difference of 2, making 25.

$$\begin{aligned} f(5) &= f(4) + d(4) + 2 \\ &= 16 + 7 + 2 = 25 \end{aligned}$$

Babbage had used pattern-matching to see that the difference of squares followed a regular sequence. In fact, a polynomial of any degree, when treated in this way, will eventually show a constant difference. For example, a function containing x^3 will have a constant third-order difference, and more generally x^n will return a constant nth-order difference.

Babbage used this finding to decompose the problem of calculating polynomials into a simple series of additions. He designed the Difference Engine to perform these calculations using a physical column of gears for each column. Each cog held a decimal digit and one whole column stored a 20-digit decimal number. Babbage

planned eight columns of gears, which would have tabulated a polynomial up to the seventh degree, or equations containing up to x^7.

Unfortunately, the precision metalwork needed for the gears to perform reliably was beyond the late Georgian engineers. This scuppered Babbage's attempts to scale the machine to work with powers of seven, so only a version capable of calculating second-order differences was ever built. Babbage's Difference Engine was displayed at the 1862 International Exhibition in London, alongside the first electric telegraph, an early plastic called Parkesine, and work by Arts and Crafts innovator William Morris.

Tiling with Euclid

Often called the founder of geometry, Euclid of Alexandria published his magnum opus, *Elements*, in around 300 BC. Within this epic work was his algorithm to calculate the highest common factor, also known as the greatest common divisor of two numbers. We can illustrate this algorithm by asking the question "What's the largest square tile I could use to tile this rectangular floor?"

Figure 5.1: Euclid was a master of algorithmic thinking. The procedure that bears his name finds the highest common factor.

Euclid says we should first choose square tiles that fit exactly in the width of the rectangle, and attempt to fill the space with these (see the largest squares in figure 5.1). When we can no longer fit a tile in the remaining space, we repeat the process within the resulting vertical rectangle, using a smaller square tile (the medium squares). We continue doing this until, eventually, a square tile will completely fill the remaining space (the small squares). The length of that tile's sides is the highest common factor of the lengths of the sides of the large rectangle.

Euclid was a master of algorithmic thinking; indeed, this procedure is known as Euclid's algorithm. In Euclid's time, however, it would have been known simply as

a procedure or method, because the man whose name gave us the word "algorithm" was not born for another thousand years…

The House of Wisdom

The Islamic Golden Age, from around the 8th century to the 14th century, was a period of cultural, economic and scientific flourishing. The caliph Hārūn al-Rashīd (circa 786-809) founded the House of Wisdom in Baghdad, which was the largest city in the world at that time. Islamic polymaths from all over the globe gathered there to translate the world's classical knowledge into Arabic and Persian. Among them was Muḥammad ibn Mūsā al-Khwārizmī (circa 780-850), a Persian scholar of mathematics, astronomy and geography. In around 820, al-Khwārizmī was appointed astronomer and head of the library at the House of Wisdom. He produced maps of the world, determined the circumference of the Earth and compiled astronomical tables for navigation.

Figure 5.2: Muḥammad ibn Mūsā al-Khwārizmī gave the world algorithms and algebra.

Al-Khwārizmī's work on elementary algebra, *Al-Kitāb al-mukhtaṣar fī ḥisāb al-jabr wa'l-muqābala* (The Compendious Book on Calculation by Completion and Balancing), was translated into Latin in the 12th century, whereupon the Arabic word "al-jabr" gave the world the term "algebra". Another work, *Al-Khwārizmī Concerning the Hindu Art of Reckoning*, is preserved only in a Latin translation, with the title *Algoritmi de numero Indorum*. From al-Khwārizmī's name came the word "algorithm".

The first programmer

Even before Babbage had finished designing his Difference Engine to run an algorithm to solve polynomials, he had begun work on a more ambitious project. The Analytical Engine was designed to be a general-purpose calculating machine, and with components we would recognise today: the "Mill" as the arithmetic logic unit (ALU) and the "Store" as storage. Although it was never built, Ada Lovelace (see chapter 1) continued to be fascinated by the Analytical Engine's possibilities. She first translated notes on the proposed machine taken by an Italian engineer called Luigi Menabrea, who had heard Babbage's talk on the engine in Turin, and later added her own notes.

The most famous of all Lovelace's notes on the Analytical Engine became known as "Note G", which contains detailed instructions on using the engine to calculate "Bernoulli numbers". The Swiss physicist Daniel Bernoulli (1700-1782) had devised a shortcut to calculate the sum of powers of integers, but it relied on first knowing a series of coefficients that now bear his name. Lovelace's "program" for the machine contains 25 steps and a loop, and, alongside the operations, Lovelace helpfully included a trace table for the eighth Bernoulli number, showing the contents of "variables" at each stage of execution.

Lovelace's version contains a bug on line 4 – the fraction is upside down – but when fixed the algorithm works flawlessly on a modern computer. It took Lovelace not a little head-scratching to devise this algorithm, but the result cements her position in history as the first computer programmer.

> *"My Dear Babbage. I am in much dismay at having got into so amazing a quagmire & botheration with these Numbers, that I cannot possibly get the thing done today. … I am now going out on horseback. Tant mieux"* – Ada Lovelace to Charles Babbage, 1843[139]

Drawing an algorithm

Babbage had experimented with ways to express machine functions in written notes, which he called "mechanical notation". He first wrote about it in 1826 under the title *On a Method of Expressing by Signs the Action of Machinery*. Meanwhile, Lovelace's program simply used the language of mathematical expressions. Early digital computers in the 1940s would be programmed in binary that could be represented by short codes called "mnemonics", in a

139 King, G. (2015) "How Ada Lovelace solved problems", *Solvitas Perambulum* (blog), link.httcs.online/tantmieux

solution called assembly language. But a husband-and-wife team in New Jersey, US, changed all that.

Lillian Gilbreth was once described as "a genius in the art of living".[140] Gilbreth, one of the first female engineers to earn a Ph.D, and her husband, Frank, were efficiency experts. They presented a paper called *Process Charts: first steps in finding the one best way to do work*, to the American Society of Mechanical Engineers in 1921. An early flowchart using the Gilbreths' notation is seen in figure 5.3. Their tools quickly became popular in industrial engineering and later in computer programming.

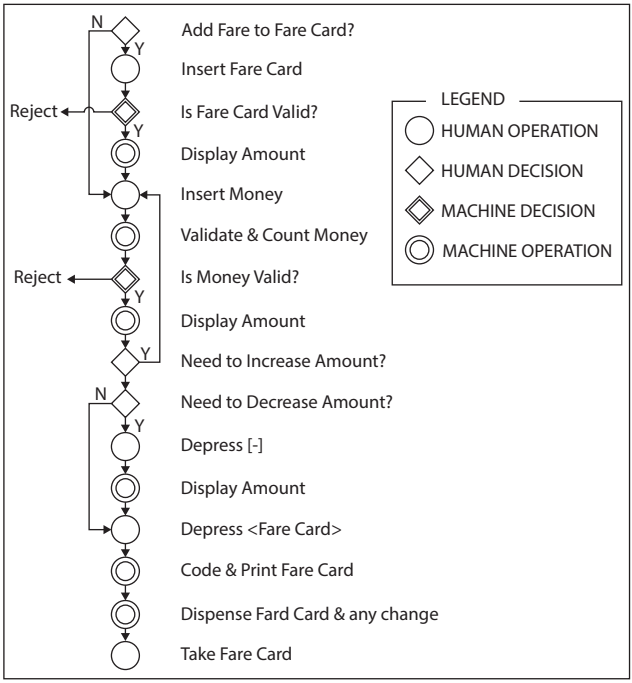


Figure 5.3: An early flowchart created by Lillian and Frank Gilbreth.

The American National Standards Institute set standards for flowcharts and their symbols in the 1960s. The rise of high-level languages and interactive

140 Graham, L.D. (1994) "Critical biography without subjects and objects: an encounter with Dr. Lillian Moller Gilbreth", *The Sociological Quarterly*, 35(4), 621-643, link.httcs.online/gilbreth

computer terminals later caused flowcharts to fall out of favour, with pseudocode replacing the flowchart when it was still necessary to write an algorithm in a standard syntax.

Mock code

If you want to precisely express an algorithm, but make it easy to read, you might use pseudocode, a plain language description of the algorithm. Pseudocode follows the structure of a programming language, but because it's intended for humans, we skip things like variable declarations, colons, brackets and braces, which would make the algorithm machine-readable. We're left with all that is necessary in order for a human to take the algorithm and code it in any language. There is no standard pseudocode syntax, as a program in pseudocode is not an executable program, but there exist some limited standards and the UK exam boards all publish their own syntax for use in assessment.

The last place you look

Some algorithms are so useful that they crop up everywhere. The simplest of filtering algorithms, linear search, is easy to understand because we do it often. To find a book on an unsorted shelf, we might look at each book in turn, read the title and compare with the book we need, until we find the one we're looking for. Linear search is sometimes called serial search or sequential search, because it looks at all the items in series or sequence.

The description "linear" comes from the behaviour of the algorithm in response to a changing number of items to be searched: as this number grows, the time taken grows in a straight line or linear fashion. In other words, if a linear search implementation takes one second to search 100 items, then it will take 10 seconds to search 1000 items, and 100 seconds to search 10,000. We say that it has "linear time complexity"; time is proportional to the number of items n, which we describe in "Big O notation" as O(n). Programs with nested loops, such as bubble sort, have "polynomial time complexity" of $O(n^2)$, meaning they quickly become slow as the number of items to process increases.

Real-world applications, like web search, need to return results in a fraction of a second from billions of items, and linear search is too slow for this purpose. If we can sort the data first (see the discussion of sorting algorithms on the next page) then we can use another algorithm called binary search. For example: check the midpoint of the array. If this matches our target, we stop. If the target is lower than the midpoint, discard the top half of the array; if higher, discard the bottom half.

Repeat this process until the target is found. This is called a binary search because we split the dataset into two parts each time; we bisect it.

Binary search can be demonstrated by playing "Twenty Questions" or the board game *Guess Who?* Because it's a "divide and conquer" algorithm that halves the search space with each comparison, binary search has logarithmic time complexity. This means it scales well to a large dataset, as long as it's sorted first.

Donald Knuth's *The Art of Computer Programming* credits the first computer binary search to John Mauchly in 1946. However, dividing in two repeatedly until the target is found is clearly not a modern idea, as illustrated by the extract from the Jataka that opened this chapter.

Order, order

As noted above, binary search requires its dataset to be sorted. Likewise, finding, adding or deleting data is much more efficient if the array or database table is in sorted order. Sorting is so important in computer science that there are at least a dozen basic algorithms in common use. Typical UK GCSE courses look at three of these: bubble, insertion and merge.

Bubble sort is simple to implement but very inefficient; according to Knuth, it "seems to have nothing to recommend it".[141] It is so called because each pass over the data, swapping adjacent items, causes the largest item to "bubble to the top". The two nested loops cause a time complexity of n^2, meaning this algorithm does not scale well to large datasets. Where bubble sort wins, however, is in the small amount of memory needed to run it. Because it holds nothing more in memory than a single temporary slot to store the current swap value, bubble sort has a low space complexity.

Rather than simply swapping adjacent items, insertion sort passes over the data, looking at each item in turn, moving it leftwards and inserting it into the correct place in the sorted sublist on its left. On the face of it, insertion sort seems like it should be more efficient, but with a worst case of n^2 it is little better than bubble sort. Both bubble and insertion sort are implementations of intuitive sorting methods that existed before computers, so they cannot be attributed to an inventor.

Merge sort, however, was an invention of one of the most brilliant minds in computing history, John von Neumann (1903-1957). He took inspiration from the card-collating machines he had been using in his work on the Manhattan Project (which produced the first nuclear weapons) and implemented the algorithm on the

141 Knuth, D. (1998) *The Art of Computer Programming, Volume 3: sorting and searching* (third edition), Addison-Wesley

EDVAC machine (one of the earliest electronic computers) in 1945. After splitting the dataset into sublists of one item, these sublists are combined into ordered pairs, then fours, and so on until we have a single sorted list. This "divide and conquer" algorithm, similar to binary search, is highly efficient. Because a list of n items can only be halved $\log_2(n)$ times, there are $\log_2(n)$ stages to a merge sort. Each stage consists of n/2 comparisons, so a merge sort consists of n/2 x $\log_2(n)$ operations. We simplify this expression to O(n.log(n)). This time complexity makes merge sort one of the most efficient sorting algorithms.

All modern programming languages come with sorting built in – for example, Python lists can be sorted just by typing .sort() at the end of the list name, running the built-in sort method. This begs the question: what sort algorithm does Python run? The answer is a hybrid of merge and insertion sort called "TimSort" after Tim Peters, major Python contributor and author of The Zen of Python (see page 94).

Lost without a trace

You will recall that Ada Lovelace helpfully included a table of variable contents during execution in her specification for the Bernoulli program. This is an early example of a technique we use to prove the correctness of an algorithm, called a trace table. Some logic errors in our algorithm are not obvious either to the programmer's eye or even after execution. This is where tracing can help.

Certain logic errors occur so frequently that they have names. A common error when coding iteration is the "off-by-one error" or "off-by-one bug", known as OBOE or OBOB for short. This is caused by looping one too many or one too few times. Consider an example: if we want to loop five times, we could do this with the Java, C++ or C# statement:

```
for (i = 0; i < 5; i++)
```

This says, "loop from zero, incrementing i each time as long as it remains below 5". The same statement in Python would be:

```
for i in range(0,5)
```

Note that the apprentice programmer might think the above statement loops six times, from 0 to 5 inclusive, as there is no "less than" symbol to suggest otherwise. Therefore, to achieve five loops with i being assigned values from 0 to 4, instead of the above we sometimes see novice Python programmers wrongly code this instead:

```
for i in range(0,4)
```

A trace table would drive out this type of error, and is often used as part of a "dry run" or walkthrough of the code before execution or during verification. This can be done as part of a code review to drive out security flaws in a program (see chapter 10).

TL;DR

In chapter 2 we saw that programming is not about using the correct keywords IF, WHILE and so on. Rather it's the process of solving a problem with the building blocks of code: sequence, selection and iteration. In this chapter, we've seen how algorithms pre-date computer science by thousands of years, and derive largely from mathematics and the natural sciences. Indeed, the word "algorithm" comes from the name of a Persian scholar, Muḥammad ibn Mūsā al-Khwārizmī (circa 780-850). It's important to understand that an algorithm exists as a concept in its own right, and that a program is merely the implementation of an algorithm on a computer. The key processes involved in creating an algorithm are abstraction, decomposition and logical thinking.

Some algorithms are so useful they crop up again and again, so an understanding of searching and sorting algorithms is necessary in computer science. Two or more algorithms can be created to solve the same problem, and they will perform differently given the same inputs, so it's important to choose the right algorithm for a task. Learners should be able to identify algorithms, interpret their purpose from flowcharts and pseudocode, correct errors and complete unfinished algorithms. To help with all this, they should be able to trace an algorithm that also drives out logic errors.

PCK for algorithms

Core concepts

- An algorithm is a precise, ordered series of steps to solve a problem constructed of sequence, selection and iterations.
- Learners should know how to choose an algorithm for a task.
- Algorithm performance can vary depending on the input data.
- Multiple algorithms exist to solve the same problem, and they will perform differently given the same input data.
- Algorithms can be represented in many ways, including flowcharts, pseudocode and program code.

- Some algorithms are very common – learners should know their characteristics and recognise their representations:
  - Bubble sort, insertion sort, merge sort.
  - Linear search and binary search.
- Learners must be able to interpret, complete, correct and trace algorithms using a trace table.

Fertile questions

- How does a satnav find a route?
- Can we write an algorithm for anything?
- How do we choose the right algorithm for the task?
- For what data is linear search quicker than bubble sort, followed by binary search?

Higher-order thinking

Making computational thinking explicit

Learners should be encouraged to "label" their work with the computational thinking (CT) skills used, as they use them. For example, if you set a challenge to write an algorithm to calculate train fares, based on number of stations travelled, learners should solve the problem using abstraction, decomposition and algorithmic thinking, and then explain how they are using those skills to solve the problem.

Matching the algorithm to the data structure

Information in the real world can be abstracted into a data structure in many ways. Discuss the ways in which routing data for a satnav or train station data can be abstracted for use by an algorithm. In how many ways could we store this data? For example, an array, a list, records in a table, a linked list. Explore the algorithms needed to process the different data structures and see how a different algorithm has to be devised for each data structure.

Seeing the complexity in the code

Bubble sort has "polynomial" time complexity, $O(n^2)$. Learners should look at the pseudocode for the algorithm and determine why this is. They can explore the relationship between the code and the time complexity and discuss tweaks that can be made to bring down the average execution time. A discussion of sorting algorithms, including use of a Boolean flag to control the outer loop to achieve this, is available at GeeksforGeeks.[142]

142 link.httcs.online/sortalgos

Complexity, exponents and logarithms

Binary search and merge sort are "divide and conquer" algorithms. Learners could explore what this means and why it causes them to have logarithmic complexity. Discuss logarithms and their relationship to number bases. Binary search is $O(\log_2(n))$ because $\log_2(n)$ answers the question "How many times can the number n be halved until we reach 1?" Encourage learners to realise that this happens exactly because $2^a = n$ implies $a = \log_2(n)$. Explore the relationship between exponent, logarithm and number base.

Analogy and concrete examples

Visualisation websites

The excellent VisuAlgo was created by a professor at the National University of Singapore. It demonstrates the major sorting algorithms in a visual way.[143]

YouTube resources

You may already be familiar with the "Hungarian folk dancers" sorting videos,[144] while Tom Scott from the Computerphile YouTube channel explains Big O notation on his own channel.[145]

Cross-topic and synoptic

Cross topic with architecture

Explore computer benchmarking, the process of determining how well a computer performs when given different algorithms to run. If you can, run the same program on different computers and compare the results. Discuss what this means, with respect to the CPU cores, clock and cache size, and the memory available to the different machines. Pseudocode samples for various algorithms are available at Isaac Computer Science.[146]

Cross-topic with data and networks

Discuss compression algorithms and why they are important in streaming services. Explore run-length encoding and discuss why it's not suitable for video-streaming. Explore JPEG compression and dictionary techniques, and compare their effectiveness on different data. Some learners may be able to determine the

143 visualgo.net/en/sorting

144 Yassin, K. (2013) "Bubble sort with Hungarian, folk dance" (video), YouTube, link.httcs.online/bubbledance

145 Scott, T. (2020) "Why my teenage code was terrible: sorting algorithms and Big O notation" (video), YouTube, link.httcs.online/bigo

146 isaaccomputerscience.org/topics/searching

time complexities of different solutions. Higher prior-attainers may be able to improve on the algorithms or even devise their own compression algorithm for images or text.

Cross-topic with issues and impacts
Algorithms such as cryptocurrency miners are in the news because they use huge amounts of energy to carry out their complex calculations. It's said that Bitcoin alone now has the energy consumption of Argentina. Discuss the ethical implications of computationally expensive algorithms.

Cross-curricular

Cross-curricular with science
Once familiar with the skills, learners should be encouraged to identify where they might use CT in problem-solving outside computer science, perhaps in other school subjects or daily life. For example, in biology they might be asked to devise an experiment to test for starch. They would abstract from all the properties of starch just the important property: iodine attaches to starch and dyes it blue. Learners would then decompose the problem into steps: set up apparatus, run experiment, gather data, analyse, write conclusion. Then they would write an algorithm for the step-by-step process. Making CT explicit in those subjects would help make it familiar.

Unplugged

Linear search is very common, so it would be easy to act this out in the classroom with playing cards, books or people. Binary search can be demonstrated with a simple "high-low" guessing game: the teacher chooses a number from 1 to 63, and the students compete to identify it in the lowest number of guesses, guided only by "higher" or "lower". They should be able to guess in six attempts every time if they operate the binary search algorithm correctly. CS Unplugged has lots of binary search activities.[147]

Physical

Writing algorithms for maze-solving robots is an excellent physical computing activity to practise CT and explore optimisation algorithms. These activities are perfect for cross-curricular learning with science and design and technology, and for project work. Many tutorials are available online.[148]

147 csunplugged.org

148 link.httcs.online/arduinomaze, link.httcs.online/microbitmaze, link.httcs.online/magpi51

Even if you can't get physical, you can write maze-solving programs in Scratch. A project you can remix to get started is described at the blog Pops' Scratch.[149]

Misconceptions

Misconception	Reality
Pseudocode has a specific syntax	Pseudocode follows the basic structure of a programming language but has no specific syntax. UK assessment authorities have clouded the issue by publishing "pseudocode syntax" guides and requiring exam answers in specific formats. Always use terms like "exam reference language" to distinguish these from pseudocode.
An algorithm is a flowchart, piece of pseudocode or program	Flowcharts, pseudocode and programs are all different representations of an algorithm that exists as a concept in its own right. A program is the implementation of an algorithm on a computer as an automatic process.

Two items in an array can be swapped with two assignment statements, one to copy left, and one to copy right, such as:
`a[i] = a[i+1]`
`a[i+1] = a[i]`
	A temporary variable is needed to store one of the values, otherwise it will be lost when overwritten by the first assignment statement.
	`temp = a[i]`
	`a[i] = a[i+1]`
	`a[i+1] = temp`
	Misconceptions like this arise because of the weakness of the learner's understanding of program execution. It is helpful to develop their understanding of a "notional machine" (see chapter 2).
	NB: Python (like some other languages) does allow assignment of values to multiple variables in one line, so you can write this code, but this is not the same as the misconception noted.
	`a[i], a[i+1] = a[i+1], a[i]`
	In this statement, each of the two expressions on the right of the equals sign are evaluated and then assigned to the elements on the left. Therefore, this code will successfully swap two array elements in one line.

149 link.httcs.online/scratchmaze

Misconception	Reality
When running a sort algorithm, the computer can "see" all the elements at once and therefore place an element in the correct place in a single operation	A program can only compare two values, so, for example, in insertion sort, the current item must be compared with each item on its left until correctly placed. This can mean many comparisons. This misconception arises during teaching of sort algorithms, when the whole array is visible to the learners at one time. We can head this off by using animation that hides all the elements, only revealing an element when the algorithm "looks" at it. The NCCE curriculum resources include just such an animation.[150]
Standard algorithms have a single, correct programmatic implementation	Standard algorithms are concepts. They describe the basic operation of a process, but the detailed implementation of an algorithm can vary. For example, when implementing binary search, it does not matter whether, on finding the mid-point in a sublist with an even number of elements, we go left or right. Neither choice is the correct one, and the implementation is usually arbitrary. Quick sort (which we meet at A-level) repeatedly sorts the data into two sublists either side of a "pivot". The choice of pivot is again arbitrary, although some books suggest the first element, while some suggest the middle element of the array. Neither choice is more correct than the other.

150 link.httcs.online/nccebubble

Chapter 6. Architecture

***Friends*, series 2, episode 8, 'The One With the List' (1995)**

Chandler: All right, check out this bad boy. Twelve megabytes of RAM, 500-megabyte hard drive. Built-in spreadsheet capabilities and a modem that transmits at over 28,000 bps.

Phoebe: Wow. What are you gonna use it for?

Chandler: Games and stuff.

We may laugh, looking back now at Chandler's "cutting-edge laptop", but this serves as a useful illustration of Moore's law. The laptop I'm writing this book on has 8GB RAM, a 500GB hard drive and 300 Mbps wireless networking, making it roughly a thousand times more powerful while costing less than a tenth of the price of Chandler's.[151]

In 1965, the 35th anniversary edition of *Electronics* magazine published this prediction from Gordon Moore, then director of research at Fairchild Semiconductor: "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year."[152] The highly competitive semiconductor industry took Moore's law as a challenge and worked hard to keep pace with the prediction, making it a self-fulfilling prophecy.

151 The *Friends* Fandom wiki claims Chandler's laptop was a 25 MHz Compaq Contura 4/25cx. link.httcs.online/chandler

152 Moore, G.E. (1965) "Cramming more components onto integrated circuits", *Electronics*, 38(8)

Moore himself co-founded the chip-making giant Intel in 1968, and his "law" remained largely reliable until recent years, when chip manufacturers began to hit limits imposed by physics. Apple's 2021 iPhone 12 sports a 5-nanometre integrated circuit, meaning the transistors inside are just 20 atoms across. Ultraviolet light traditionally used to etch the silicon wafers now has too long a wavelength, so new and expensive "extreme UV" lasers must be used. Even so, manufacturers are fast approaching the problem of "quantum tunnelling" – electrons simply "jumping the gate" when transistors become too small. But we're getting ahead of ourselves. Long before Chandler's laptop, and two decades before Gordon Moore made his famous prediction, a Hungarian-American scientist was grappling with the complex mathematics of nuclear explosions.

The 'think animal'

John von Neumann is known for the arrangement of CPU, register, buses and memory that bears his name – the von Neumann architecture – but he also made great leaps in mathematics, physics and economics. A child prodigy, von Neumann could divide eight-digit numbers in his head at the age of six, and was described by Albert Einstein as a *Denktier*, meaning "think animal".

Von Neumann founded game theory, coining the phrase "zero-sum game" and devising the "minimax" strategy – principles now fundamental to economics. His work on self-replicating machines was vital in the understanding of DNA. He worked with Alan Turing on the foundations of artificial intelligence and, as we discussed in the previous chapter, invented one of the first divide-and-conquer algorithms, merge sort, in 1945. But it was in his work on the Manhattan Project, the US effort to develop a nuclear weapon, that he made his biggest contribution to computer science.

Von Neumann worked at the US Army nuclear weapons research base in Los Alamos, New Mexico, on mathematics for the "Fat Man" bomb, the type that was dropped on the Japanese city of Nagasaki on 9 August 1945. Frustrated with the slow IBM mechanical tabulating machines (see chapter 1), he published some thoughts on a more general machine in his *First Draft of a Report on the EDVAC* in June 1945. EDVAC was the proposed successor to ENIAC and was already in use by the US Army's ballistics researchers at the University of Pennsylvania.

Sometimes called the first general purpose electronic computer, ENIAC was "programmed" by altering switches and plugboard wirings. This "setting up" was performed by a team of female operators chosen from the "human computers" in the Moore School of Electrical Engineering: Kay McNulty, Betty Jean Jennings, Betty Snyder, Marlyn Wescoff, Fran Bilas and Ruth Lichterman.

Von Neumann's *First Draft* described for the first time the internal components we would recognise today: arithmetic unit, control unit, clock, main memory, input/output manager and secondary storage. Building on Turing's seminal 1936 paper *On Computable Numbers*, which first introduced the theoretical concept of a "stored-program computer", von Neumann's *First Draft* explained how, using a cycle of "fetch, decode and execute", the same memory store could be used for both instructions and data (previous designs had loaded instructions from a separate store, usually paper tape). EDVAC, incorporating von Neumann's suggestions, was completed by John Mauchly and J. Presper Eckert in 1949. Meanwhile, von Neumann built his own machine at Princeton's Institute for Advanced Study, completed in 1951. But neither machine was the first working stored-program computer.

Baby steps

Freddie Williams had already successfully built a prototype CRT memory (see chapter 1) in 1946, when he was recruited by Max Newman to take up the chair of electrotechnics (later electrical engineering) at Manchester University, UK. Familiar with von Neumann's *First Draft*, Williams saw that his own tube memory would make an ideal random-access store for a von Neumann computer, and set about building one with the help of Tom Kilburn and Geoff Tootill. In June 1948, the Small-Scale Experimental Machine, also known as the Manchester Baby, successfully calculated the highest proper factor of the number 2^{18} (262,144) in just 52 minutes.

The Baby used more than 500 valves, was five metres long and weighed around a tonne, but had a von Neumann architecture that we would recognise today. Other contemporary machines were groundbreaking in their own ways but limited in others. The German inventor Konrad Zuse's Z3 was programmable, but it used relays rather than valves, lacked conditional branching and loaded programs from paper tape. Bletchley Park's Colossus (see chapter 10) was electronic, but ran only a few specialised programs set up by a plugboard. ENIAC was "programmed" via switches and cables on a plugboard because it had no program memory. So, despite being built with the sole intention of testing Williams' new storage device, the Manchester Baby has the honour of being the first fully electronic, stored-program, general purpose computer.

The Baby used four CRTs in total. Two small tubes functioned as registers we would recognise today: a 32-bit program counter and 32-bit accumulator. A larger 32 x 32-bit main memory was echoed to a fourth display, visible to the operator. Instructions could be fetched every 360 microseconds, then decoded and executed in around four times that. Through a three-bit binary code, just seven instructions were available, including subtraction. To save on logic circuitry, addition was

performed by negating followed by subtraction, then negating again, which works because a – b is the same as – (– a – b). Despite these limitations, the "proper factor" program required only 17 instructions – an ingenious solution to the limited instruction set issue, foreshadowing the work of the MIT hackers (see chapter 4).

A replica Baby was constructed in 1998 and is on display in the Science and Industry Museum in Manchester (see figure 6.1).

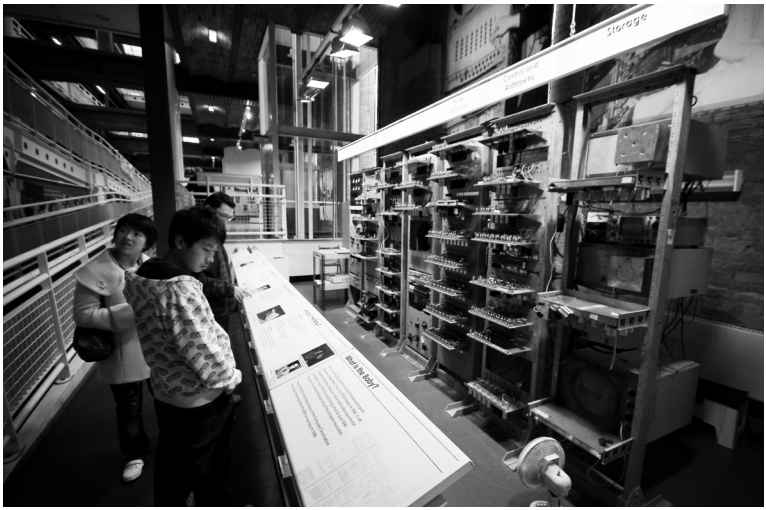


Figure 6.1: The Manchester Baby was the first fully electronic, stored-program, general purpose computer.

Mark 1

Much more than a proving ground for Williams' CRT memory, the Baby actually validated the work of Turing and von Neumann, ushering in the age of the stored-program electronic computer. After running only three different programs on the Baby, the team embarked on the more ambitious Manchester Mark 1. In 1948, Turing had taken up Max Newman's invitation to come to Manchester, frustrated with slow progress on his Automatic Computing Engine at the National Physical Laboratory in Teddington. Turing helped design the new Mark 1, which still used CRT memory, but the expanded arithmetic and logic unit (ALU) now performed addition and logical comparisons. Crucially, a newly developed magnetic drum added secondary storage for the first time. The Manchester Mark 1 successfully computed Mersenne primes in a nine-hour run in June 1949 and served as a template

for the world's first commercial computer, the Ferranti Mark 1, which went into production by the British electrical engineering firm Ferranti in February 1951.

The Ferranti Mark 1 could perform around 800 operations per second from a set of 30 simple instructions. Like the ENIAC and the EDVAC, the Mark 1 was programmed by a team of women – among them Mary Lee Berners-Lee, whose son Tim would go on to invent the World Wide Web (see chapter 9). Ferranti went on to rebrand the Mark 1 "Mercury", selling nine publicly and an unknown number to government agencies, before switching to newly invented transistors to make the Atlas supercomputer in 1962.

Evergreen design

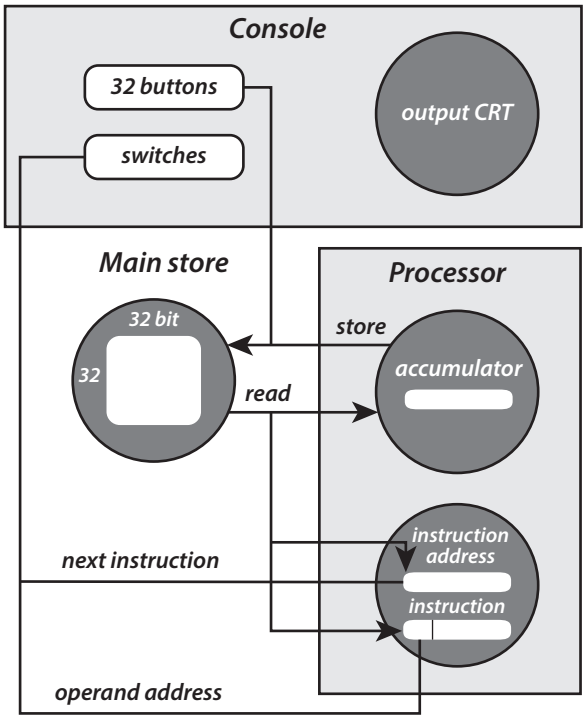


Figure 6.2: Baby's architecture is strikingly similar to that of modern computers.

From the Manchester Baby to the modern day, the design of a general-purpose computer has changed little. Figure 6.2 shows a schematic of the Baby. We can

see the processor (ALU) and three registers that are familiar to us: accumulator, instruction address (which we know as the program counter) and instruction register (or current instruction register). We see a main store (RAM) and the lines connecting the processor to the main store (buses). And at the top, we see the input devices (buttons and switches) and output device (output CRT).

Figure 6.3, created by William Lau, is the schematic of a modern computer, showing the same registers and buses. With the addition of a few more registers, this is clearly the same design.

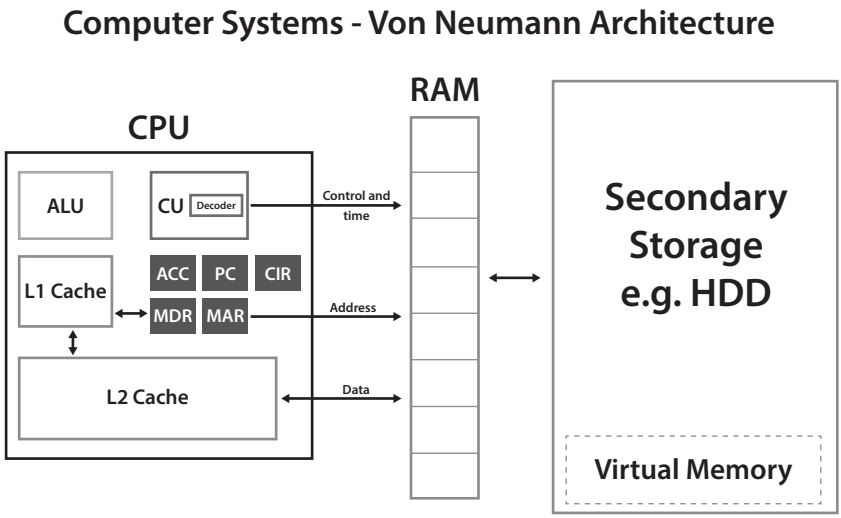


Figure 6.3: The architecture of a modern computer.

Bottle openers

Early computers were largely limited by the processing speed of the CPU. In the 1960s, fast-switching transistors changed all this: suddenly CPUs were sitting idle, waiting for instructions or data. Splitting main storage into data memory and instruction memory allowed the CPU to fetch an instruction on one bus, while fetching data on another. This twin-memory solution to the "von Neumann bottleneck" is known as the Harvard architecture. It was named, somewhat ironically, after von Neumann's own wartime Harvard Mark 1 machine, which had read instructions and data from separate paper tapes. Modern computers combine von Neumann and Harvard ideas in a "hybrid architecture" to maximise performance.

The three Cs

As the desire for more powerful computers accelerated during the 1960s and 1970s, manufacturers sought other ingenious ways to improve performance. The Manchester Baby worked at around 700 instructions per second. By the time *Friends*' Chandler bought his laptop in 1995, affordable CPUs ran at 25 MHz or 25 million instructions per second – up by a factor of 40,000 in 48 years. The CPU in your laptop probably runs at over 2 GHz, which is three million times the speed of the Baby. As valves gave way to transistors, which were then miniaturised into integrated circuits, the shrinking size and power consumption of the silicon allowed clock speeds to increase. As the electrons have shorter distances to travel, the logic gates can all do their stuff in a shorter time period, and the clock can tick over faster. The first C of performance is, therefore, clock speed.

Remember that the CPU waits idly for instructions or data a lot of the time. Harvard or hybrid architecture prevents a bus bottleneck, but this doesn't eliminate the lag of fetching from RAM. Bringing the data closer to the CPU reduces fetch times. Pulling bits across the system bus from RAM is relatively slow. Even at the speed of electricity (which is close to the speed of light), a fetch from RAM can take 100 nanoseconds – a whopping 200 clock ticks of wasted processing time.

In order to speed up execution of programs, instructions can be cached. Cache memory responds in as little as 1 nanosecond or just two CPU cycles. When cache holds both instructions and data, that program segment completes far quicker than if it were all in RAM. Cache is the second of the three Cs, which improves a computer's performance.

Why don't we make all main memory into cache? The answer is that cache is expensive and, ironically, the bigger it gets, the slower it performs. The CPU industry has therefore invested billions in predicting which instructions and data are needed next; all CPUs since Intel's 8086 (released in 1978) contain technology that prefetches contents from RAM into cache to prevent CPU idleness. Processors now have many levels of cache: a tiny level 1 cache tightly integrated with the CPU has lightning performance, while level 2 and sometimes level 3 are further away, relatively slower and cheaper.

As we discussed, physical limitations on transistor size have ended Moore's law-style increases in performance. An obvious solution is simply to put two CPUs into every computer or, more accurately, CPUs with two or more processing cores on one silicon wafer. In May 2005, AMD's Athlon 64 X2 was launched as the first commercial dual-core CPU. Multitasking operating systems like Windows are able to give CPU time to two programs at once, with each core running its own fetch-decode-execute cycle. Some programs, such as games and video-rendering

software, can even run two parts of the same program simultaneously – a trick called multithreading. Today's home computers usually have two or four cores, and Apple's iPhone 8 and X models have six. Number of cores is the third C in our discussion of computer performance characteristics.

Remembering stuff

Before Williams tubes, computer memory usually consisted of slow magnetic drums, slower magnetic tapes or cumbersome "acoustic delay lines". Developed from radar technology, a delay line carried pulses of sound representing binary data through a tube of fluid, usually mercury. A pulse reaching the end was detected and retransmitted, and this constant loop of pulses allowed von Neumann's EDSAC delay lines to each "hold" a pattern of 560 bits. Of course, mercury is heavy. In EDSAC's 1951 successor, UNIVAC, 18 mercury tubes providing just 1.5KB of memory weighed in at nearly 800lbs, or a third of a tonne.[153] The sound waves bouncing around the device sounded like human voices, earning it the nickname "mumble-tub".[154]

Figure 6.4: Pulses of sound bounced back and forth in the UNIVAC's "mumble-tub" mercury delay line memory.

153 link.httcs.online/delayline

154 Fantel, H.H. (1956) "The electronic mind – how it remembers", *Popular Electronics*, link.httcs.online/mumble

Tape and delay-line memory were both serial access devices, meaning the bits you wanted had to come back around before you could read them. UNIVAC's delay-line memory had an access time of 222 microseconds, which was faster than other memory at the time, but too slow to usefully store a program. Williams' fully electronic CRT memory allowed for random access, making Turing's stored-program computer a possibility. No longer tied to paper tapes, plugboards and cumbersome delay-line memory, electronic computers took off.

Random thoughts

"Random access" seems a strange way to describe the process of retrieving data from named memory locations. We usually define "random" as "without intent", and how can memory locations be chosen without intent? In random-access memory (RAM), however, the meaning of random is the same as in the mathematical term "random distribution" – unpredictable. The computer will return the contents of any memory location in the same timeframe, no matter where in memory you look. The pattern of retrieval locations can be without order and performance will be equal for all of them, unlike serial access, where the fetch time depends on how far down the series of bits your data resides. Mathematically speaking, when working with RAM a randomly distributed pattern of retrievals has uniform performance.

Instead of being loaded serially from tape or disk, a program stored in RAM allowed the fetch-decode-execute cycle to run quickly and include loops and branching at no extra performance cost. Therefore, along with the three Cs, RAM capacity is the fourth major characteristic affecting computer performance. We heard earlier that the Manchester Baby had a 32 x 32-bit CRT for its main memory, giving 1024 bits or 128 bytes in today's measure. But Williams tubes were highly sensitive to environmental disturbances, so the Baby's main memory was encased in steel to defend against interference from the electric trams running outside, and more robust solutions were sought.

The mid-1950s saw the arrival of ferrite core memory, thanks to the efforts of the physicists An Wang and Way-Dong Woo at Harvard, plus a team at MIT led by Jay Forrester (who had taught himself electrical engineering to bring wind power to his family's off-grid farm in Nebraska).[155] The DEC PDP-1 – on which the first computer game, *Spacewar!*, was written in 1959 by the MIT hackers – had around 9KB of core memory and IBM's flagship 360 mainframes shipped with up to 6MB of core memory in the late 1960s. The Apollo spacecraft carried around 32KB of core memory (see figure 6.5 on the next page). Core memory dominated for

155 link.httcs.online/forrester

two decades from the mid-1950s, until semiconductor "dynamic random-access memory" (DRAM) chips became affordable.

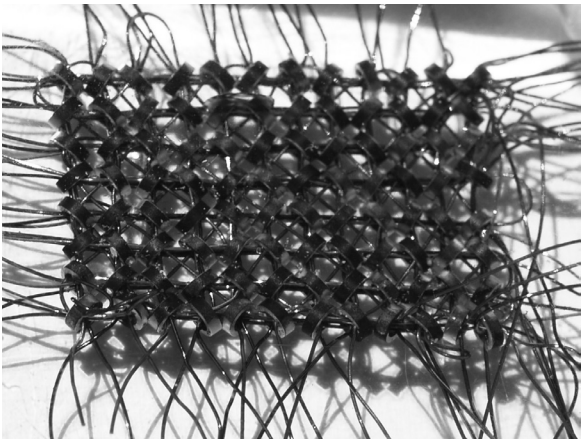


Figure 6.5: Core rope memory was used in NASA's Mariner and Apollo programmes.

Never let me go

DRAM, like the Williams tube, is volatile, which means it loses its contents when powered off. Core memory was non-volatile, but, of course, it could be altered at any time, leaving important programs at risk of being overwritten. Computers require permanent programs, including the start-up instructions for the machine, so we need memory that cannot be overwritten – or read-only memory (ROM). NASA's computers for the Apollo space missions used rope memory: tiny wires running through iron rings, threaded by a mostly female team led by Margaret Hamilton (see chapter 3).[156] IBM created a more flexible cousin of rope memory, called transformer read-only storage (TROS), for its mainframes, but these solutions required re-manufacture at great cost if a program had to be changed.[157]

The 1970s semiconductor industry made a series of breakthroughs, culminating in ROM chips that could be cheaply programmed and reprogrammed in the computer factory. These were known as "electronically erasable, programmable ROM" or EEPROM chips. By the mid-1970s, all electronic computers shipped with

156 Brock, D.C. (2017) "Software as hardware: Apollo's rope memory", *IEEE Spectrum*, link.httcs.online/apollorope

157 Shirriff, K. (2019) "TROS: How IBM mainframes stored microcode in transformers", *Ken Shirriff's Blog*, link.httcs.online/tros

semiconductor RAM and ROM. Because RAM and ROM are accessible directly by the CPU, they are known collectively as main memory or primary storage. ROM is non-volatile, but as it's read-only we can't write to it. RAM is volatile, so anything we write is lost when we power off. Another type of storage is necessary if we want to keep the data we create, and to store programs permanently.

Forget me not

Secondary storage is non-volatile, writeable storage. My computers use either a hard disk drive (HDD) or solid-state drive (SSD) for this purpose. But we look again to IBM for the first commercial magnetic hard drive. The IBM 305 RAMAC computer shipped in 1956 with a hard disk drive storing five million six-bit characters, the equivalent of around 4MB or four million bytes. Magnetic hard drives in a modern desktop PC now exceed one terabyte (TB) of storage – that's one million, million bytes. Magnetic tape drives were also used commercially from the early days of computing to the turn of the century, when cloud storage and flash memory made them largely obsolete.

Early portable and home computers shipped with their operating systems all in ROM and with just a floppy drive for storage, until small hard drives became affordable in the 1990s. A floppy disk was a thin flexible disk encased in hard plastic that stored just a couple of megabytes. Although they are now obsolete, the "save" icon in many current software programs still resembles a floppy disk.

The main drawback to magnetic storage is its reliance on moving parts. A read/write head must move over the disk surface, or the tape must be pulled past the head. This movement causes wear and tear, and all magnetic media fails eventually. Solid state drives and USB memory sticks don't have this issue, so are resilient to drops and knocks. Both contain silicon chips called flash memory, which was developed from EEPROM memory. Unlike EEPROM, where the whole chip has to be erased and rewritten, small chunks of flash memory can be edited without affecting the rest, so we can use the chip as very fast secondary storage. But beware: although flash memory is physically robust, this "program/erase" cycle degrades the silicon, causing SSDs to fail eventually.

Optical disks complete the set of three main types of secondary storage in common usage. CDs, DVDs and Blu-ray disks consist of a plastic disk pressed with indentations – or, in the case of recordable disks, a dye coating that is made reflective in places and is read as binary data by a laser. Optical disks were hugely popular for the distribution of music, video and software throughout the 1990s and 2000s, but usage has declined with the rise of online media stores and streaming services.

The right to bear ARMs

The development that made mobile phones possible came about through a series of happy accidents. Steve Furber and Sophie Wilson at Acorn Computers in Cambridge had successfully built the Micro for the BBC's 1980s Computer Literacy Project and were working on a more powerful machine for use in business. Apple had just launched the Lisa (in 1983) and Acorn was convinced it could compete. But Acorn was unimpressed with the CPUs available, so decided to design its own chip based on cutting-edge ideas on instruction sets coming out of the University of California, Berkeley.

The falling cost of RAM meant it was no longer vital to write programs in the smallest space possible, and instruction sets could shrink in size while programs grew. The resulting RISC (reduced Instruction Set Computer) chips carried far fewer logic gates, making them faster and requiring less power. Wilson was able to imagine the whole circuit for Acorn's new chip all at once, then write a simulation of it in 800 lines of BBC Basic. The Acorn co-founder Hermann Hauser recalled: "While IBM spent months simulating their instruction sets on large mainframes, Sophie did it all in her brain."[158]

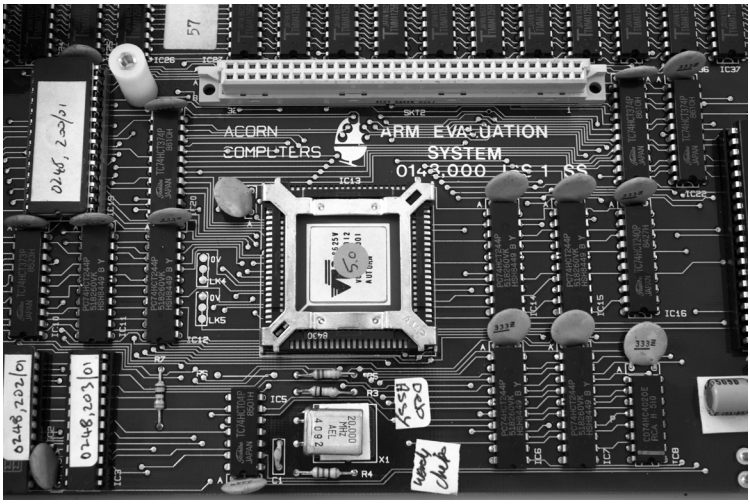


Figure 6.6: The ARM processor, seen here in an early test rig, was designed by Sophie Wilson and Steve Furber in Cambridge, UK.

158 link.httcs.online/sophie

The Acorn RISC Machine (ARM) chip was designed to improve the performance of desktop microcomputers, not to save power and space. But when Apple came looking for a low-power, cool-running chip for its ill-fated Newton handheld computer in 1987, only the ARM chip fitted the bill. Apple got behind Acorn and in 1990 the two companies spun off ARM Ltd. After the ARM-powered iPod was launched in 2001, the world went ARM-crazy. By 2010, more than 95% of smartphones contained an ARM processor and the company shipped more than 20 billion chips in 2017.

Plus ça change

What's remarkable when we look back at the history of computer architecture is not the rapid pace of change, although that is startling enough. It's that the underlying architecture has changed little, despite huge advances in technology delivering exponential growth in processor power and storage density.

Alan Turing and John von Neumann would recognise the design of today's computers; they would need but a moment in order to read the instruction set of the x86 family of processors or the ARM RISC chip and code for it. The genius of the codebreaker and the vision of the *Denktier* live on.

TL;DR

Alan Turing described the concept of a stored-program computer in 1936. John von Neumann built on Turing's work, explaining in 1945 how a cycle of fetch-decode-execute could allow the same memory to hold programs and data. Freddie Williams led a team at Manchester University that built the Small-Scale Experimental Machine, known as the Baby, to prove his CRT memory store. The Baby ran around 700 instructions per second in 1946. Its success led to the Mark 1 and then, in 1951, the Ferranti Mark 1 – the first commercial computer, for which women wrote most of the programs.

Valves gave way to much faster transistors in the 1960s and this exposed the "von Neumann bottleneck", solved by the Harvard architecture of separate memories for instructions and data. Early memory stores included paper tape, magnetic tape, magnetic drum, acoustic delay lines and core rope memory, until semiconductor RAM arrived in the 1960s.

Magnetic hard disk drives provided secondary storage from the 1950s onwards, with flash memory becoming popular in the 21st century for portable storage devices and solid-state disks (SSDs). Compact discs (CDs), invented in 1979, and DVDs and Blu-ray discs are examples of the third common storage type, the optical disk.

Computer performance is limited by the three Cs: clock speed, number of cores and size of cache. CPUs can be described as CISC (Complex Instruction Set Computer) or RISC (Reduced Instruction Set Computer). RISC processors have much simpler circuitry, reducing space and power consumption, making them suitable for mobile devices. The ARM chip is a RISC processor used in 95% of all the world's mobile devices.

From the Baby to the ARM, all CPUs still contain an ALU, registers and a control unit, and perform a fetch-decode-execute cycle first described by von Neumann in 1945.

PCK for architecture

Core concepts

- Difference between embedded and general-purpose computers.
- Architecture of a computer system.
- Parts of a CPU.
- The fetch-decode-execute cycle and the roles of the parts of a CPU during the cycle.
- Primary memory and the characteristics of RAM, ROM and cache.
- The need for secondary storage; types of secondary storage: optical, magnetic, solid state, cloud.
- Volatile versus non-volatile storage.

Fertile questions

- How can we design the fastest computer system in the world?
- Why are some computers sold as "gaming PCs"?
- What do the Manchester Baby and the iPhone still have in common?
- Why is Moore's law slowing down?
- Why don't we make the cache huge and get rid of RAM and secondary storage?
- What parts of a typical computer are upgradeable?

Higher-order thinking

Design a computer

After teaching the factors that determine the performance of a computer system, and the characteristics of different types of secondary storage, ask the learners to design computers for different tasks, such as:

- A student taking notes in lectures, studying and writing essays.
- Mining cryptocurrency.
- Navigation by soldiers on the battlefield.

Analogy and concrete examples

Window shopping
Learners could browse computers for sale on shopping websites and discuss the meaning of their characteristics.

Evaporating RAM
The meaning of "volatile" can be made clearer with references to the word's meaning in English, chemistry and politics. For example, I always say: "In chemistry, volatile liquids evaporate easily. Remember that RAM loses its contents, so you could say the data evaporates."

Remember Venn
Categorising storage types and locations with a Venn diagram can help learners grasp the difference between internal/external and primary/secondary.

Little Minion
CPU simulators such as the Little Man Computer[159] are a great way to visualise the fetch-decode-execute cycle.

Cross-topic and synoptic

Cross-topic with languages and programming
Giving learners some experience of low-level programming, using a simulator such as the Little Man Computer, is an excellent way of teaching the architecture of a computer and how it relates to the languages and programming topics.

Cross-topic with issues and impacts
Look at the performance of a computer and discuss Moore's law in the context of the ethical issue of customers perpetually wanting to upgrade their computers and smartphones.

Cross-topic with system software
Discuss how the amount of RAM and CPU determines computer performance. Then link to the process scheduler that gives programs a time slice in rotation in order to share the CPU time fairly. Also discuss the memory manager that shares out the RAM and swaps programs to virtual memory when necessary.

159 link.httcs.online/lmc, link.httcs.online/lmc101, link.httcs.online/dringcpu

Discuss defragmentation utilities and why they might be needed, particularly with a small hard drive.

Cross-topic with data representation
Talk about the size of RAM, cache and secondary storage. Discuss what this means for file sizes that can be held in memory or stored on the hard disk

Cross-curricular

Cross-curricular with history
The architecture of the computer evolved rapidly during the Second World War, out of necessity. Discuss the role of computers in the war: the UK's "Ultra" codebreakers (see chapter 10) and the US's nuclear Manhattan Project. Speculate about what might have happened without computers – would the war have ended differently?

Cross-curricular with geography
The ARM chip made mobile computing possible. Discuss the impact this has had on global communication.

Unplugged

Role-playing the components of a computer is a great way to help learners understand the fetch-decode-execute cycle. You can find an activity called "fetch-decode-execute-plot" at the STEM Learning website,[160] described as "a frantic starter activity aimed at students aged 17-18". Craig'n'Dave make a similar game available as part of their premium resource pack.[161]

Physical

The Raspberry Pi is a great way to get hands-on with a small computer. Show students an image that helps them locate the CPU, RAM and SD-card secondary storage.[162] If possible, open an old desktop computer and compare the two.

Project work

The ideal project to help students learn computer architecture is to build a desktop computer from components. If you run an after-school club, this might be a suitable activity. There are many tutorials online, including on the Instructables website.[163]

160 link.httcs.online/stemfde

161 craigndave.org/a-level-premium-resources

162 raspberrypi.org/products

163 link.httcs.online/computerbuild

Misconceptions

Misconception	Reality
Virtual memory is cloud storage	Virtual memory is a space on the hard drive used to extend the available RAM.
Primary = internal Secondary = external	Primary storage is always internal to the computer, but secondary storage can be internal (e.g. main HDD/SSD) or external (portable SSD/HDD or memory stick).
Cache = registers	Cache is general purpose memory. Registers are special purpose stores that hold only a few bytes each and include the PC, MAR, MDR and accumulator. NB: some teaching resources get the registers and cache confused.
Primary = volatile	Primary storage includes RAM, ROM and cache. What makes it primary is that the CPU can read it directly, during the FDE cycle. ROM is, of course, non-volatile.
More secondary storage improves performance	More RAM can sometimes improve performance, but not more secondary storage.
RAM always increases performance	More RAM will not increase performance if there is not a shortage of RAM in the first place. If virtual memory is rarely used, adding RAM won't help.
Secondary storage is used when the RAM is full	Secondary storage is not an extension of RAM – it is long-term non-volatile storage. It holds the operating system and application programs. Virtual memory is a special case, and learners should be clear that this does not mean secondary storage exists for when the RAM is full. Only the start-up instructions reside in ROM in modern computers; the rest of the programs are on secondary storage.
A smartphone is an embedded system	Smartphones are general purpose computers because they run many different applications. Note that a smartwatch is also commonly considered a general-purpose computer, as they usually run apps.
A 3GHz performs better than a 2GHz CPU	There are too many factors at play in the performance of a computer to make this assertion. You need to consider the program(s) it will run and their suitability for multithreading, how much input/output they perform, how much memory they need to run, and whether any processes are suitable for delegating to a co-processor such as a graphics processor unit (GPU). Other factors to consider include the number of cores, size of cache and amount of RAM.

Misconception	Reality
RISC is better than CISC (or vice versa)	RISC processors are generally smaller and consume less power, but require extra RAM and more complex software. The two CPU architectures are suitable for different applications – neither is universally better than the other.

Chapter 7.
Logic

An old joke about logic…

A mathematician, a physicist and an astronomer were travelling north by train. They had just crossed the border into Scotland when the astronomer looked out of the window and saw a single black sheep in the middle of a field. "All Scottish sheep are black," he remarked.

"No, my friend," replied the physicist. "Some Scottish sheep are black."

The mathematician looked up from his paper and glanced out of the window. After a few seconds' thought, he said blandly, "In Scotland, there exists at least one field, in which there exists at least one sheep, at least one side of which is black."

Little lightbulbs

Charles Babbage's 1820s Difference Engine used gears driven by a crank handle and his planned Analytical Engine was to be steam-powered, as these were the cutting-edge technologies in the first half of the 19th century. IBM's census-tabulating machines of the late 19th and early 20th centuries were electromechanical, meaning they combined physical prongs, bushes and belts with electrical relays, switches and plugboards. By the 1940s, when the Second World War was driving computing innovation on both sides of the Atlantic, thermionic valves were the component of choice for computer circuitry.

When the UK's Government Code and Cipher School (GC&CS) at Bletchley Park needed to crack the German Lorenz cipher, the engineer Tommy Flowers built

a computer using 1,600 valves. But valves, also known as vacuum tubes, were notoriously unreliable. As in tungsten lightbulbs, the filament inside would fail owing to the stress of repeated expansion and contraction, or air would leak into the tube, oxidising the filament so it stopped emitting electrons. Fortunately, a team of researchers in the US provided a solution, although they were not working on computers at the time.

Not the iotatron

A team at Bell Labs in New Jersey, led by William Shockley, were trying to build an improved amplifier: a fundamental electronic circuit used in radio, radar, telegraph and telephone equipment. Amplifiers used those same temperamental valves, but Shockley believed he could exploit the quantum mechanics of electrons to build a "solid state" amplifier. By gluing contacts of gold leaf to the tip of a wedge of germanium crystal, and attaching another contact to the base, Shockley's team were able to control the current flowing between the gold contacts just by turning on and off the base voltage. Adding a small voltage at the base caused electrons to fill up the gaps in the crystal, allowing a much larger voltage to flow between the gold contacts.

Shockley demonstrated the device by using it to amplify speech on 23 December 1947. Many names were suggested, including "surface states triode" and "iotatron", but because the device transferred resistance from the base to the flow circuit, the combination of the words "transfer" and "resistor" suggested the name "transistor". Nine years later, Shockley and the engineers John Bardeen and Walter Brattain were awarded the Nobel Prize for their invention.

Tom Kilburn at Manchester University again led the way with the "Transistor Computer" in 1953. But it still used a number of valves, meaning Bell Labs' 1954 TRADIC was the first fully transistorised computer. IBM launched the IBM 608 calculator in 1955, followed by several transistorised computers – the most popular of which was the 1400 series, released in 1959. In the same year, DEC launched the PDP-1, beloved of the MIT hackers (see chapter 4), and by the late 1960s nobody was using valves.

On and off

Amplification may have driven the transistor's invention, but computer scientists quickly realised its importance as a fast switch. As we saw in chapter 1, computers store and process binary numbers, which are represented in the computer as electrical signals. A high voltage represents 1, while a low voltage represents 0. In practice, modern circuits work on around 5V, so anything over 3V is counted

as "on", while anything below 1.5V means "off".[164] The transistor allows that 5V signal to be turned on and off electronically at lightning speed, without the need for mechanical relays or temperamental valves.

A transistor takes two inputs and produces an output. Specifically, it takes a data input that arrives at the collector and a control input at the base. If the base input is 1, then a 1 will flow to the emitter. Why is this useful in computers? As we saw in chapter 6, computers carry out arithmetic and logic operations. Representing "True" or "1" as a high voltage, and "False" or "0" as a low voltage, humble transistors can be combined to make logic gates, and those logic gates can be arranged to perform arithmetic.

Making choices

Imagine you are a parent and your daughter, Ada, asks for a birthday party. You decide to reward good choices and reply, "You can have a party if you tidy your room and do all your homework." You are operating a logic gate with two inputs, which we can call "tidy_room" and "homework". If we call the output "party", we can write the decision like this:

party = tidy_room AND homework

We could use two transistors connected in series (see figure 7.1 on the next page) to make the decision for us. Applying a voltage to the base of one transistor if Ada tidies her room, and to the other if she does her homework, we will get a 5V output from the circuit only if Ada does both tasks. You may recognise that we've built the AND logic gate from two transistors. Transistors can be combined in other ways to create different logic gates, which are the building blocks of computer circuits. The theory behind logic circuits goes back thousands of years, but we can thank a schoolmaster from Lincolnshire for our modern understanding.

164 link.httcs.online/voltages

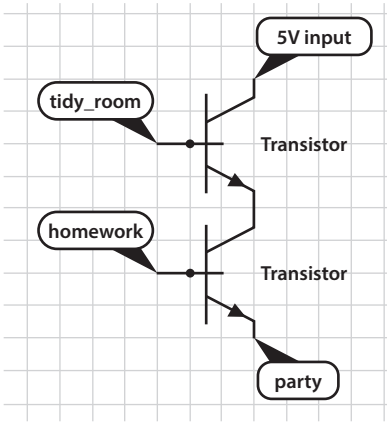


Figure 7.1: An AND logic gate made from two transistors.

The schoolmaster who classified thought

The phrase "party = tidy_room AND homework" would be easily recognised by George Boole (1815-1864) as a valid expression in what he called "symbolical algebra", now known as Boolean algebra. The young Boole was so obsessed with mathematics that he taught himself French so he could read Sylvestre François Lacroix's *Differential and Integral Calculus* in its original language. By the age of 16, Boole was teaching mathematics in a local school, and by 19 he had opened his own school in his hometown of Lincoln.

Boole was mentored by Augustus De Morgan, the first president of the London Mathematical Society, who also tutored Ada Lovelace. In 1847, while still a schoolmaster, Boole published his *Mathematical Analysis of Logic*, which led to a professorship at Queen's College in Cork, Ireland; he wrote another 50 papers in his lifetime. In 1855 he married Mary Everest, niece of George Everest, the surveyor and geographer after whom Mount Everest is named.

Mary Everest had been a child prodigy. She grew up partly in France, where she had a private tutor; she spoke fluent French and longed to study mathematics at Cambridge. But Everest had to satisfy her thirst with books, as the university didn't admit women until 1869 (and even then they could not be awarded degrees until 1948). Everest's parents were well-connected in academia, with John Herschel and Charles Babbage regular visitors to the home. Everest met Boole on a visit to Cork to see her uncle, John Ryall, a professor of Greek and vice-president at Queen's

College, after which they corresponded avidly. Everest later attended Boole's lectures and eventually worked alongside him.

Boole proposed to Everest after her father's death; they had a happy marriage and five daughters, all of whom had distinguished careers in science and literature (Lucy Everest Boole was the first female Fellow of the Royal Institute of Chemistry). After Boole's death, Mary Everest Boole continued his work and developed progressive ideas on education, publishing *Philosophy and Fun of Algebra* in 1909, which explained algebra and logic to children.

Algebra of truth

Of the many logical operations described by Boole, computer scientists are interested in just a few, usually AND, OR, NOT, NAND and XOR. A Boolean algebra expression is written with one or two Boolean variables and a Boolean operator, and when evaluated it will return a Boolean result. For example, A AND B will return True if both A AND B are True, but False otherwise (we saw this in the example of Ada's party). For every operator, its name explains what it does – for example, A OR B will evaluate to True if either A OR B is True. Boole's paper "Mathematical analysis of logic" explained how to combine operators in more complex Boolean expressions.

We can explore expressions with multiple operators using variations on our parenting example. As Ada's parent, you come home from a trip to the countryside and the car is dirty; you know Ada hates tidying her room, so you want to give her the option of washing the car instead. But homework is still mandatory. So you change the party rule to:

party = (tidy_room OR wash_car) AND homework

This will evaluate to True if either Ada tidies her room OR washes the car AND, as well as one of those tasks, does her homework. Note that we put brackets around the OR clause, to make it clear which operator is evaluated first. Without this, by convention we would evaluate AND first, and Ada could get away without doing her homework as long as she tidied her room!

In 1854, Boole published *An Investigation of the Laws of Thought*, regarded as his tour de force. In it, he showed for the first time that manipulation of symbols in equations can be used reliably for logical deduction, and even introduced mathematical probability to the world.

Boole's contributions to mathematical logic have many implications for computer science. We use Boolean expressions in selection and conditional iteration statements. For example:

```
if student = "Y" or senior = "Y" and fare > 5:
     discount = 0.1

if finished and total = 0 then
     print("no credit")

if day = 1 and not(holiday) then
     print("Go to work")

while not(found) and count < n do
     n = n + 1
```

And we can use Boolean operators in web searches, such as "cheddar AND cheese", to find only pages that contain both search terms. But this chapter is primarily concerned with Boolean logic in the design of computer circuits, and for this we owe much to the son of a judge and a schoolteacher born in Michigan in 1916.

Switch hitter

On graduating from the University of Michigan in 1936 with degrees in mathematics and engineering, Claude Shannon began graduate studies in electrical engineering at MIT. There he met Vannevar Bush, the dean of engineering, whose Differential Analyzer was designed to solve equations that were too complex for the mechanical calculating machines of the time. This massive analogue computer was hundreds of feet long and took two or three days to configure. At Bush's suggestion, Shannon analysed the machine for his master's thesis.

Shannon had noted that although the Analyzer was analogue, meaning the positions of its components varied continuously, the relay switches governed its overall behaviour and they were always in one of just two states: on or off. This led him to recall Boole's work. Shannon suggested using relays to build a digital "logic machine" that could not only calculate but perform "information processing". In 1938, he published his results in *Transactions of the American Institute of Electrical Engineers* under the title "A symbolic analysis of relay and switching circuits".[165] Shannon's 10-page article is among the most important engineering papers ever written: by linking Boolean logic to electrical circuits, it opened the door to digital electronics.

165 link.httcs.online/shannonmasters

Back to gates

We've seen how transistors can be combined to perform logical operations such as AND. Combining several transistors in this way to perform a logic operation is called a gate. When designing circuits for a computer, we use symbols for logic gates to simplify our circuit, instead of drawing the transistor symbols each time. These logic diagrams are therefore an abstraction of the actual electronic circuit beneath, hiding the detail of individual components so we can more easily understand its operation.

So, when we want to solve a logic problem, such as party = (tidy_room OR wash_car) AND homework, all we have to do is grab the necessary gates, set up the inputs of the logic circuit and read the output. Figure 7.2 shows the circuit needed.

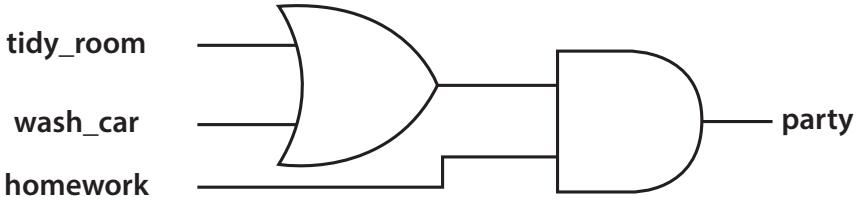


Figure 7.2. The Boolean logic circuit for "party = (tidy_room OR wash_car) AND homework".

We can see how logic gates can be used to perform logical operations with Boolean variables. But what about answering questions about numbers? Recall one of the high-level language selection statements from page 142: `if total = 0 then`. How can we perform this comparison using only logic gates? Well, checking equality to zero is easy, because NOT(0) = 1, so we just have to push each of the binary digits of the number through a NOT gate and then AND the results, which will be 1 if all the bits are zero.

Now what about comparing to other numbers? Well, we can decompose the problem IF day = 1 into two separate problems:

> subtract 1 from day.
>
> compare to zero.

We know how to compare to zero – we just did that. So all we need now is the ability to do simple arithmetic…

It all adds up

Binary addition has only four rules:

```
0 + 0 = 0
0 + 1 = 1
1 + 1 = 0  carry 1
1 + 1 + 1 = 1  carry 1
```

Figure 7.3. The four rules of binary addition.

If we treat each single column as a logic operation, it has two inputs: each of the two bits we need to add and two bits of output (the sum and carry bits). We can show what happens with a truth table that has two inputs and two outputs, like this:

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

Figure 7.4. A truth table for binary addition. Sum and Carry are the two outputs.

We can now design a logic circuit that makes the above happen. The carry output is clearly just A AND B. The sum output needs to be 1 when either A is 1 and B is NOT 1, or vice versa. We can use two AND gates and two NOT gates to do this:

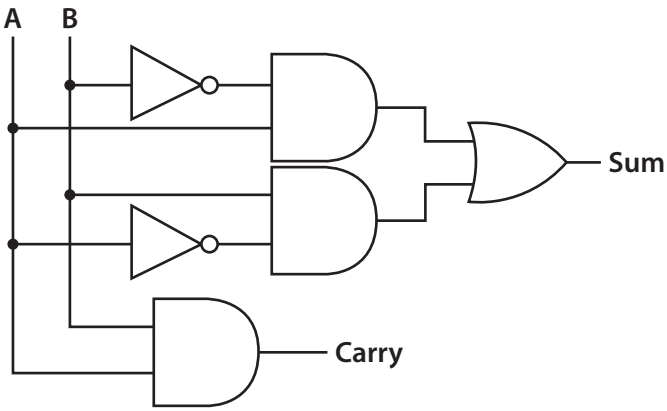


Figure 7.5. This circuit, called a "half-adder", can add two binary digits.

However, there is a special logic gate called "exclusive OR", written as XOR, that gives output 1 if either input A or B is 1, but not both.

XOR has a symbol that looks like the OR gate but with an extra line at the back. So we can use an XOR gate to tidy up our adder circuit, which now looks like this:

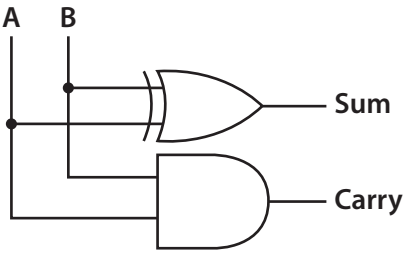


Figure 7.6. A "half-adder" using just two gates, XOR and AND.

From just two logic gates that perform simple Boolean operations on their inputs, we have an adding machine. All that remains is to string lots of these together and then we can add a big binary number.

Below zero

We can now add, but how do we subtract? We can take advantage of the fact that a – b is the same as a + (–b). Now all we need is to be able to turn a number negative!

The first attempts to use negative numbers in a computer employed a "sign bit", a single digit to indicate whether a number was positive or negative. But the logic circuit required to perform arithmetic properly with "sign and magnitude" numbers proved complicated. Computer scientists realised that a number system called "two's complement" held the solution.

The odometer that records a car's mileage holds a finite number of digits. When run in reverse, the odometer is disconnected to prevent unscrupulous car dealers running back the mileage. But imagine if it did not disconnect: running a car backwards would subtract miles. Eventually the display would read zero and then the car would flip back to 999999 miles. Every mile driven in reverse would subtract another mile, so it would read 999998, 999997 and so on.

Odometer reading	Actual miles travelled
999997	–3
999998	–2
999999	–1
000000	0
000001	1
000002	2

Figure 7.7. A car's odometer run in reverse would flip back to 999999, meaning "–1 miles driven".

If we consider backwards to be negative miles, and forwards to be positive miles, figure 7.7 shows what the odometer readings mean: 999997 means a total distance of –3 miles has been driven. We would need a rule such as "A value over 500,000 means negative miles, 500,000 or below means positive", and we would have a way of representing negative numbers without a sign.

We use the same principle in two's-complement binary. As we count down from zero to minus 1, the digits flip from 0001 back to 1111, which represents minus 1.

Two's complement	Denary
1101	–3
1110	–2
1111	–1
0000	0
0001	1
0010	2

Figure 7.8. The four-bit binary number 1111 means "–1" in two's complement.

It turns out that these numbers behave beautifully when we perform addition using our logic gate adders above, hence the popularity of two's complement.

Returning complements

In mathematics, a complement is another half of a whole – for example, the Boolean expressions A and NOT A are complements of each other. In arithmetic, a pair of numbers are complements if they add to the same value. The "one's complement" system, which was used in early computers such as the PDP-1, turned positive numbers negative by flipping all the bits, i.e. changing 0s to 1s and 1s to 0s. In this scheme, a binary digit and its negative complement always added to 1, hence one's complement.

One's complement was difficult to implement in logic, but with the slight modification of adding 1 after flipping the bits we get the two's complement system, used in Tom Kilburn's Manchester Baby and all modern computers. Two's complement solves the problem of negation, and we had addition sorted already, so now we can do subtraction.

Go forth and…

So, we have addition and subtraction, but what of multiplication and division? Well, multiplication is just repeated addition. For example, 3 x 4 is simply 4 + 4 + 4. We just need to repeatedly add 4 while counting up to three or, better still, counting down from 3 to zero. We have all the tools we need for this already: addition, subtraction and compare to zero. Likewise, division is just repeated subtraction: to perform 12 ÷ 3 we just repeatedly subtract 3 from 12, counting the number of times we can do it until we get zero. A few simple logic gates have given us the power to do really useful mathematics.

For a much fuller explanation of how computers work, I thoroughly recommend *But How Do It Know?* by J. Clark Scott[166] and *Code* by Charles Petzold.[167]

TL;DR

George Boole (1815-1864) published his paper "Mathematical analysis of logic" in 1847, describing what became known as Boolean algebra. Claude Shannon saw how Boole's work could be applied to electronics, publishing the article "A symbolic analysis of relay and switching circuits" in 1938. The first digital computers used fragile valves and slow relays. Transistor computers arrived in the 1950s, greatly improving speed and reliability.

Computers use a high voltage (around five volts) to represent either "True" or a binary "1", and a low voltage (close to zero volts) to represent "False" or a binary "0". A transistor acts like an electronic switch, turning the voltage on or off in another part of the circuit. Transistors can be combined into logic gates.

A logic gate is a collection of microscopic transistors that perform a Boolean logic operation such as AND, OR and NOT. Logic gates are combined into circuits inside a computer to perform arithmetic and logical operations.

PCK for logic

Core concepts

- How logic gates work and their role in the function of a computer.
- Logic gates AND, OR and NOT, their symbols and truth tables.
- Combining logic gates in logic circuits.
- Draw a logic circuit from a Boolean expression and vice versa.
- Constructing truth tables with three inputs.

Fertile questions

- How does a computer perform arithmetic?
- How is selection performed inside a computer?
- What's inside the ALU?
- Why don't we use analogue computers any more?

166 Clark Scott, J. (2009) *But How Do It Know? The basic principles of computers for everyone*, John C Scott

167 Petzold, C. (2000) *Code: the hidden language of computer hardware and software*, Microsoft Press

Higher-order thinking

More able learners may wish to find out more about half-adders and full-adders, and use an online logic circuit design program such as logic.ly to implement them.

Likewise, learners could explore the A-level concept of flip-flops to understand how the clock works to synchronise the operations of a computer. Some excellent videos to get them started are available from Craig'n'Dave.[168]

Learners could write truths about real-world situations, turning them into Boolean expressions, then drawing circuits and truth tables for them. For example:

- Yusuf will have a party on his birthday if he passes his exams and is not sick that day.
- You can ride the subway half-price if you're a student, but only on weekdays.
- Next year will be a leap year if it is divisible by 4, but not also divisible by 100.

Mark Mills has more of these examples and other activities on his website The Computing Café.[169]

Analogy and concrete examples

Although GCSE-level study does not require understanding of two-output circuits, the half-adder can really explain why it's important to study logic gates.

Cross-topic and synoptic

Cross-topic with programming

For the sentences in the "Higher-order thinking" section, the learners could also write Boolean expressions in various programming languages to understand the relationship between programming selection and the Boolean logic operated by the ALU. For example:

```
if passed = True and not(sick):
    print("party on Yusuf!")
```

Note that the expression "passed = True" could be simplified to "passed".

Cross-topic with networks

Boolean operators can be used in web searches to improve the quality of search results. Learners could try out Boolean expressions in searches and check the number of web pages returned.

168 link.httcs.online/cndboolean

169 link.httcs.online/cafeboolean

Cross-curricular

Cross-curricular with maths

Boolean expressions can be illustrated with Venn diagrams. Edinburgh University has a tool to generate Venns from expressions.[170]

Unplugged

The 2008 Royal Institution Christmas Lecture series included an unplugged exercise where children acted as logic gates, and offered a video showing how to make logic gates out of dominoes.[171]

Physical

The micro:bit can be used to explore Boolean logic. For example, learners could code a program that displays different symbols depending on whether one or two buttons are pressed. See httcs.online for sample code.

If you have a Raspberry Pi running Minecraft, you can make logic gates out of a special block called Redstone. Various tutorials for this are available.[172]

I am lucky enough to have a class set of Unilab Decisions boards (see figure 7.9), which were made in the 1980s to teach circuit design. You can still pick them up on auction sites and they are excellent for getting hands-on with logic gates.

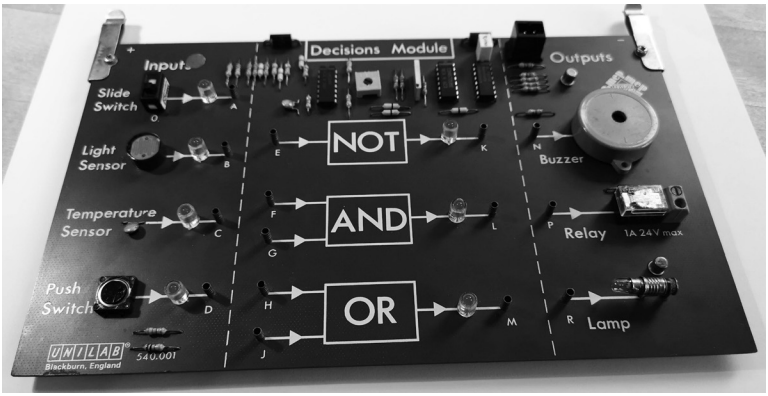


Figure 7.9: The Unilab Decisions module, part of a discontinued series of teaching equipment but still circulating on auction sites.

170 link.httcs.online/vennmaker

171 link.httcs.online/rilogic

172 link.httcs.online/minelogic, link.httcs.online/minelogictutorial

Project work

An after-school club is an excellent place to build actual logic circuits that learners have designed themselves. A real stretch would be designing an ALU; a tutorial is available on the All About Circuits website.[173]

Learners could collaborate on a website to explain logic gates to younger learners, using glitch.com or even Scratch.

Misconceptions

Misconception	Reality
NOT is a binary operator, taking two inputs – for example, we see the expression A NOT B	NOT is a unary operator – it takes only one argument – so A NOT B is invalid. NOT B is valid, as is A AND NOT B. Remind learners that NOT simply inverts the input, 0 → 1 and 1 → 0. It therefore takes only one input. Some websites make this mistake, so if learners find A NOT B online, explain it is an error.
Computers understand binary	Computers merely process electrical signals using logic gates. Careful arrangement of these gates allows binary numbers to be manipulated. This misconception can be caused by early explanations of how computers work that include use of the phrase "computers don't understand words, they only understand binary".
AND/OR confusion	AND requires both inputs to be 1 for the output to be 1. OR requires only one input to be 1 for output to be 1. Confusion between two-input logic gates can be averted by engaging in plenty of discussion of Boolean expressions. Using the words AND and OR expressively helps, such as: The AND gate requires inputs a **and** b to be True for the output to be True. The OR gate requires either A **or** B to be True for the output to be True. NB: in GCSE and A-level studies we discuss two-input logic gates only, but chips containing gates with 3, 4 and 8 inputs are available.

173 link.httcs.online/buildalu

Chapter 8.
System software

Palo Alto, California, 1982

Susan Kare is in her garage, welding a lifesize steel sculpture of a razorback hog commissioned by an Arkansas museum. Kare has been a curator at the Fine Arts Museums of San Francisco since achieving her PhD from New York University, and she is happy to be creating again. "I'd go talk to artists in their studios for exhibitions," Kare would later recall, "but I really wanted to be sitting in my studio."[174]

Her work is interrupted by a phone call from an old school friend, Andy Hertzfeld, lead software architect for the Apple Macintosh operating system. In a 2000 interview, Kare said:

> *"By remaining friendly with Andy after high school, I knew he obviously was really interested in computers. He showed me a very rudimentary Macintosh, and mentioned that he needed some graphics for it – he knew I was interested in art and graphics – and that if I got some graph paper I could make small images out of the squares [and] he could transfer those onto the computer screen. That sounded to me like a great project. I did it in exchange for an Apple II, although I didn't actually use the Apple II for Mac graphics."*[175]

174 Silberman, S. (2011) "Meet Susan Kare, the pioneer who created the Mac's original icons," *Fast Company*, link.httcs.online/karefastco

175 Pang, A.S-J. (2000) "Interview with Susan Kare", Stanford University, link.httcs.online/karestanford

Kare joins Apple and creates some of the most memorable icons for the Mac, including the scissors, finger and paintbrush, as well as the notorious "bomb" icon that was displayed alongside a fatal "error" message. Kare would later explain that her experience with needlepoint helped with the bitmap creation: "Bitmap graphics are like mosaics and needlepoint and other pseudo-digital art forms, all of which I had practiced before going to Apple. I used to say: if you like needlework, you'll love bitmap design!"[176]

As well as icons, Kare is tasked with developing fonts for Mac OS; she is determined to break away from monospaced fonts that make both a narrow "i" and a wide "m" the same width (a hangover from typewriter design). Kare designs proportional fonts, naming them after stops on the Philadelphia commuter train she had taken to school with Hertzfeld, including Overbrook, Merion, Ardmore and Rosemont.

Then Apple co-founder Steve Jobs stops by the software group. On hearing the font names, he advises: "Cities are OK, but not little cities that nobody's ever heard of. They ought to be WORLD CLASS cities!"[177] So the Mac fonts become Chicago, New York, Geneva, London, San Francisco, Toronto and Venice. When the Mac launches in 1984, Kare's proportional fonts and simple but visually appealing icons are warmly received, and help to make Mac OS the most user-friendly operating system on the market.

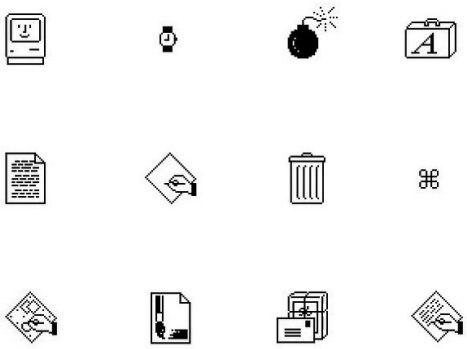


Figure 8.1: Susan Kare designed the original Macintosh icons on graph paper.

176 Campbell, O. (2018) "The story behind Susan Kare's iconic design work for Apple", The Work Behind The Work, link.httcs.online/kareicons

177 link.httcs.online/worldclass

Why's it all in hardware?

> *Random took the watch back from him and looked at it again, clearly baffled by something. Then she held it up to her ear and listened in puzzlement.*
>
> *"What's that noise?"*
>
> *"It's ticking. That's the mechanism that drives the watch. It's called clockwork. It's all kind of interlocking cogs and springs that work to turn the hands round at exactly the right speed to mark the hours and minutes and days and so on."*
>
> *Random carried on peering at it.*
>
> *"There's something puzzling you," said Arthur. "What is it?"*
>
> *"Yes," said Random, at last. "Why's it all in hardware?"*
>
> From chapter 15 of *Mostly Harmless*, book five in The Hitchhiker's Guide to the Galaxy series by Douglas Adams.

As we've seen, the early computers were programmed by switches, paper tape and plugboards. They produced output on indicator lights, dials and punched cards. If a program needed input or output, those interactions were written into the program, or requested by setting switches and plugging cables. Processing would halt while data was read from slow disks or tapes, leaving hugely expensive CPUs idle most of the time. As computers became more powerful in the early 1960s, interaction between program and hardware became very costly in lost processor time. IBM and General Motors had developed a rudimentary batch operating system for the IBM 704 in 1956, but this did little more than run the next batch program when the previous one had finished. This didn't solve the problem of a CPU sitting idle much of the time.

Battle of the North Atlantic

Two simultaneous projects taking place 3,000 miles apart were tackling this problem. In 1956, IBM had promised the US Atomic Energy Commission (AEC) a supercomputer within four years that was capable of speeds "at least 100 times the IBM 704", which meant four million instructions per second (MIPS). This goal was never achieved and the project lost IBM $20 million. But, at 1.2 MIPS, the Stretch computer (now branded the IBM 7030) was still the fastest machine on the planet when it was delivered to the AEC at Los Alamos in May 1961.

Meanwhile, across the Atlantic, Tom Kilburn was working on a supercomputer at Manchester University, which his team funded initially by renting out time on their Ferranti Mark 1 (see chapter 6) and later by a government grant. The

Atlas supercomputer was fired up in December 1962, instantly doubling the UK's computing capacity. Pioneering in many ways, Atlas was built with germanium transistors and diodes instead of valves and relays; it operated virtual memory, swapping programs from core store to magnetic drum and thus increasing the memory available for programs.

The ROM consisted of a mesh of wires woven by a local textile factory; it bore small ferrite or copper rods, with ferrite meaning "1" and copper "0". The memory resembled a hairbrush and Atlas had around 48KB of "hairbrush" ROM, with a lightning-fast access time of three microseconds.

A collection of ROM-stored procedures called "extracode" gave programmers built-in subroutines for the first time, including the mathematical operations square root and logarithm, and hardware operations for reading from tape and printing decimal numbers. But it was the supervisor program that was arguably the greatest advance. An early proto-operating system, it provided an interface between the applications and the hardware and managed the computer's resources.[178]

Application programs would call the supervisor to transfer to or from peripherals, move data to the magnetic drum, and handle overflow errors or memory/time problems. Both Atlas and Stretch's supervisor code provided multiprogramming features for the first time, switching between many application programs to keep the CPU as busy as possible. A program was put on hold if it exceeded its time slot, or when it requested a hardware operation.

While these supercomputers could not be said to have operating systems by modern standards, these innovations paved the way for modern OSs. Features of Stretch and Atlas were implemented in later OSs, such as IBM's commercially successful OS/360.

The 360-degree all-rounder

Stretch was considered a failure and the best bits went into IBM's next big project, the System/360 – an undertaking so big and complex that it nearly broke IBM. But, according to the Institute of Electrical and Electronics Engineers' *Spectrum* magazine, it "proved to be the most successful product launch ever and changed the course of computing … A short list of the most transformative products of the past century and a half would include the lightbulb, Ford's Model T—and the IBM System/360."[179]

178 Kilburn, T., Payne, R.B. and Howarth, D.J. (1962) "The Atlas Supervisor", link.httcs.online/atlas

179 Cortada, J.W. (2019) "Building the System/360 mainframe nearly destroyed IBM", *IEEE Spectrum*, link.httcs.online/ibm360

IBM had two successful machines in the early 1960s – the popular 1401 and the larger and more expensive 7000 – but they ran incompatible software. IBM needed a series of compatible machines with increasing power, providing a clear upgrade path and thus reducing the risk of lost customers. The result was the 360, built from modular hardware components and an operating system that supported a wide range of applications and peripherals.

Building a computer that was equally good at simple tabulating and complex mathematics meant devising an all-purpose operating system with more than a million lines of code – a bigger programming project than had ever been attempted. Sixty programmers grew to 1,000, the project overran by a year and the budget topped out at $5 billion, half the cost of the atomic bomb. But on 7 April 1964, with some of the modules still in development, IBM president Thomas J. Watson Jr. made what he called "the most important product announcement in the company's history".

Scenes familiar to anyone who has attended a 21st century Apple product launch greeted the arrival of the new machines, and within a month IBM had 100,000 orders. Thus began the most terrifying period of the project – delivering machines that were not yet complete. As Watson later recalled, "Not all of the equipment on display [on 7 April] was real; some units were just mock-ups made of wood. We explained that to our guests, so there was no deception. But it was … an uncomfortable reminder to me of how far we had to go before we could call the program a success."[180]

Supply outstripped demand, IBM ramped up production beyond reasonable capacity, and machines were shipped with missing parts and software bugs. In his memoirs, Watson admitted to being in a "nearly continuous panic" from 1964 to 1966. But the teething troubles were overcome: by 1971, annual sales topped $8 billion and by the mid-1970s IBM had 70% of the mainframe market.

OS/360 delivered features that are familiar to us today, including multiprogramming, a file system with indexed data files, program libraries, a job scheduler, interrupt handling and print spooling. Users could access the OS directly via CRT "dumb terminals" connected to the time-sharing system, or submit jobs via punched card readers, which remained popular into the 1980s. IBM's competitors sought to emulate the success of the 360 and the system influenced the design of rival OSs, such as General Electric's Multics and Bell Labs' Unix.

Got the power

At MIT in the mid-1960s, work had stalled on an experimental operating system called Multics for the GE mainframe. AT&T Bell Labs had pulled out of the project,

180 Ibid.

frustrated by the complexity of Multics. Bell Labs researchers Ken Thompson and Dennis Ritchie used a PDP-7 computer to create a scaled-down version of Multics, initially calling it Unics, a pun on the idea of simplifying Multics.

Bell Labs wanted to edit patent documents, so Thompson and Ritchie ported the new OS to the much larger PDP-11 and added a word processor. The Unix Programmer's Manual was published on 3 November 1971, now considered Unix's "official birthday".

Figure 8.2: Ken Thompson (seated) and Dennis Ritchie at a PDP-11 circa 1970.

As he wrote utilities for the fledgling OS, Ritchie was frustrated by the limitations of the programming language "B", so he wrote a new high-level language, calling it "C". Ritchie rewrote the whole of the Unix operating system in C code in 1973. Unix was presented to the world at the 1973 Symposium on Operating Systems Principles, the same conference where, six years earlier, Larry Roberts had announced the ARPANET (see chapter 9). Unix proved hugely popular in academia, partly because of how powerful it was.

> *"UNIX always presumes you know what you're doing. You're the human being, after all, and it is a mere operating system."*
>
> Ellen Ullman, programmer and author, 1998[181]

181 Ullman, E. (1998) "The dumbing-down of programming", *Salon*, link.httcs.online/ullman

When graduates were hired by rising technology companies, they often brought the operating system with them. Unix was written in C, so it was easy to port to other systems, and by the end of the 1980s Unix was everywhere. But there was a problem. Nobody was "in charge" of Unix, so many different standards and versions had emerged, making the choice of OS for a computing project a minefield. In August 1991, an unknown student from Finland was about to change all that.

Penguins on everything

Linus Torvalds, a student at the University of Helsinki, wanted a Unix-like operating system for his IBM 80386-based PC. A version of Unix for PCs was available, called MINIX, but it wasn't free and it couldn't be modified, so Torvalds began work on his own in 1991. Originally called Freax – from "free", "freak" and "Unix" – an administrator at the university uploaded it to an FTP server under the name "Linux", which stuck. While choosing a mascot for the product in 1996, Torvalds visited a zoo in Australia where he was bitten by a penguin, and the mascot "Tux" was born. Open-source from the beginning, Linux took the world by storm and is now running everything from Google's web servers to Tivo set-top TV boxes.

> *"I get the biggest enjoyment from the random and unexpected places. Linux on cellphones or refrigerators, just because it's so not what I envisioned it. Or on supercomputers."*
>
> Linus Torvalds

Quick and dirty

In 1980, IBM needed an operating system for its new range of personal computers (PCs). The company had planned to buy a version of Digital Research's CP/M operating system, the commercial standard for minicomputers at the time (if they weren't running a version of Unix). But the deal fell through and IBM approached a small Seattle company called Microsoft, which was selling copies of its BASIC programming language on punched tape to hobbyists. Microsoft produced PC-DOS 1.0 in July 1981, based largely on a product called 86-DOS (or QDOS, for "quick and dirty operating system") created by a local company, Seattle Computer Products. IBM demanded that 300 bugs be fixed before accepting the software from Microsoft for $430,000. MS-DOS provided a command-line interface, like all OSs before it. But a small company based in Cupertino, Silicon Valley, had designs on a more user-friendly interface.

Pronounced 'gooey'

Steve Jobs and Steve Wozniak had co-founded Apple in 1976 to sell Wozniak's Apple I personal computer, and a year later they had success with the Apple II. Jobs visited Xerox PARC in 1979, where he saw the mouse-driven Alto, which had a graphical user interface (GUI). The 1983 Apple Lisa was technologically advanced, with a mouse, GUI and twin floppy drives, but it was unreliable and, with a whopping price tag of $9,999, Apple sold only 10,000 in three years. Undeterred, Jobs threw everything at the 1984 Macintosh, which became the first mass-produced computer with a GUI. Incorporating Susan Kare's visually appealing proportional fonts and icon sets and priced at a much more affordable $2,495, the Mac was a huge hit, selling 70,000 in the first quarter of 1984.

For the first time, rather than typing commands at text prompts, users moved a mouse pointer to visually navigate folders, features and files represented by icons. The GUI model was adopted by most subsequent operating systems, most notably Windows, which launched in 1985. Initially, Jobs accused Microsoft CEO Bill Gates of ripping off his idea, but both parties had in fact seen the earlier Xerox PARC concept, and so agreed a licensing arrangement soon after.

Jobs quit Apple during a power struggle in 1985 and took several Apple staffers with him to NeXT Computer, which built the workstation on which Tim Berners-Lee hosted the world's first web server (see chapter 9). Jobs returned in 1997, when Apple bought NeXT for $427 million, bringing with him a Unix-based operating system. Apple combined the look and feel of the classic Mac OS with the innovative features of NeXTSTEP to create Mac OS X. (Nearly two decades later, in 2016, after the launch of iOS, watchOS for the Apple Watch and tvOS for the Apple TV platform, Apple would rebrand its desktop and laptop operating system macOS and include the voice assistant Siri for the first time. With Siri, Microsoft's Cortana, Amazon's Alexa and the Google Assistant, the world's most popular operating systems would all have voice interfaces.)

Wow, it confirms DOS!

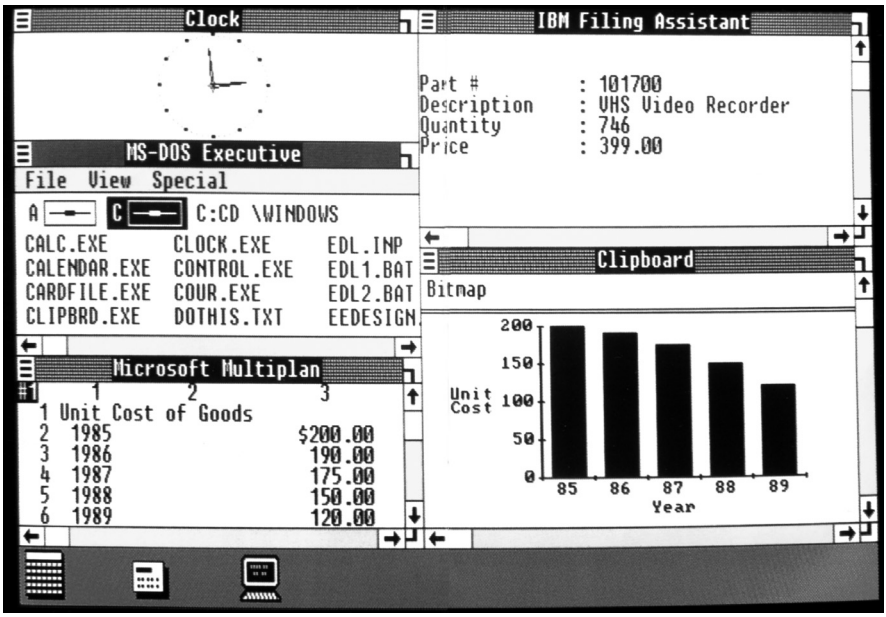


Figure 8.3: Windows 1.0 added GUI features to DOS in 1985 but didn't yet allow overlapping windows or true multitasking.

Apple had beaten Microsoft to the GUI punch with the Lisa and Mac, but Windows 1.0 wasn't far behind, launching in 1985.

Little more than a GUI add-on for MS-DOS, version 1.0 had tiled windows plus a calculator, a calendar, the image editor Paint, a strategy game called Reversi, and a simple word-processor called Write.

Although the user interface was very basic, even early versions of Windows included device drivers and cooperative multitasking, meaning it could run many programs at once, unlike DOS. Windows 3.1, released in 1993, shipped with support for peer-to-peer file sharing, virtual memory and TrueType fonts, at last allowing the OS to compete with Mac OS for desktop publishing customers.

Windows 95 brought full pre-emptive multitasking, meaning the process scheduler had full control over program execution and could kick a long-running task off the CPU to allow others to run. The release included the Start menu and taskbar for the first time, the My Computer shortcut and the recycle bin, utilising a "desktop metaphor" interface borrowed from Mac OS. Windows 95 started up with a short

musical sequence that became familiar in offices around the globe, composed by the music legend Brian Eno, ironically on an Apple Mac.[182]

Since Windows 95, the user interface has changed little, apart from an ill-judged redesign for Windows 8 that was reversed in Windows 10. Successive versions of the OS have generally added compatibility, security and versatility; and, like Apple with iOS, Microsoft spun-off a portable version called Windows CE, which became Windows Phone. In 2015, to compete with Apple's Siri, Microsoft added the voice assistant Cortana to Windows 10 and Windows Phone, based on a character from the popular Xbox game franchise *Halo* (Microsoft retired Cortana from its mobile products in 2021). Microsoft's Xbox games console runs a version of Windows, and in 2010 gained gesture support with the Kinect input devices.

Pinch me

While the 2001 iPod used a simple embedded operating system, Apple's seminal 2007 iPhone used an operating system based on Mac OS X, called iPhone OS and later just iOS. Steve Jobs had been undecided on whether to build on the iPod's minimal software or shrink the desktop operating system to fit, so he ran an internal competition won by Scott Forstall's Mac team. It was a crucial decision, allowing Mac developers to produce software for the iPhone with little modification. iOS has a direct manipulation interface. In human-computer interaction (HCI) terms, this means objects on the screen can be tapped, pressed, dragged or pinched to cause intuitive effects. But, like macOS, Apple would not license its operating systems, so rival manufacturers had to look elsewhere.

A group of developers led by Andy Rubin founded Android Inc. in Palo Alto, California, in 2003. Based on Linux, Android OS was originally meant for digital cameras, but it quickly became clear to Rubin that mobile phones were a bigger target. Google bought the company in 2005 but kept the software open-source, believing this would encourage app developers to get on board. The famous Android logo was created by Irina Blok, a Silicon Valley designer who also did creative work for Yahoo, HP and Adobe.

182 Higgins, C. (2013) "Creating the Windows 95 startup sound", *Mental Floss*, link.httcs.online/95sound

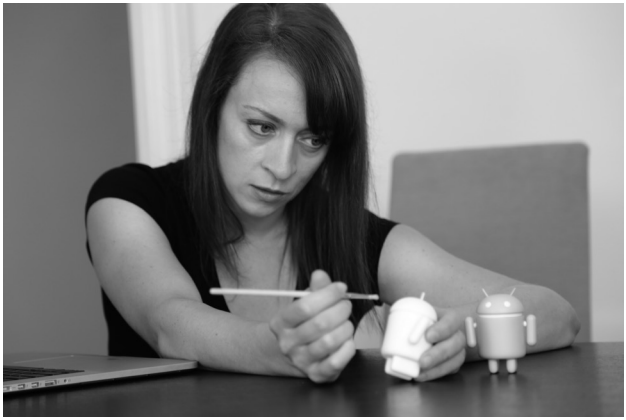


Figure 8.4: Irina Blok, designer of the Android logo.

Everyone expected Google to answer the iPhone with a phone of its own, but instead it announced in 2007 the creation of the Open Handset Alliance of several manufacturers and mobile networks. Google's then chief executive, Eric Schmidt, said: "Today's announcement is more ambitious than any single 'Google Phone' that the press has been speculating about over the past few weeks. Our vision is that the powerful platform we're unveiling will power thousands of different phone models."[183] The public beta of Android version 1.0 launched for developers on 5 November 2007. Subsequent versions were named after items of confectionery: Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwich, Jelly Bean, KitKat, Lollipop, Marshmallow, Nougat, Oreo and Pie.[184]

Initially, Google's rivals were sceptical. Nokia said: "We don't see this as a threat." A member of Microsoft's Windows Mobile team stated: "I don't understand the impact that they are going to have."[185] But Android took 71% of the mobile OS market in 2021, compared with Apple's 27%. In 2013, Nokia abandoned its Symbian OS in favour of Windows Phone and later jumped again to Android; Microsoft ceased support for its Windows Phone OS in 2019. Analysts attributed Android's runaway success to the huge number of apps available thanks to its open-source licencing.

183 Sparkes, M. (2007) "Google announces Android software platform for phones", *ITPro*, link.httcs.online/androidlaunch

184 Callaham, J. (2021) "The history of Android: the evolution of the biggest mobile OS in the world", *Android Authority*, link.httcs.online/androidversions

185 Popa, B. (2018) "Did you know? Microsoft, Apple and Nokia all mocked Android when it launched", *Softpedia News*, link.httcs.online/androidskeptics

Amazon chose to "fork" (make a spin-off version of) Android for its Fire tablets, but with tight integration with its own cloud shopping and media stores, adding the Alexa voice assistant to its products beginning in 2014.

Type, click, wave or talk?

The first operating systems had a command-line interface (CLI): users would type commands at a cursor and the operating system would interpret and process them immediately. A typical Unix series of commands to create a directory and edit a file might look like this:

```
$ pwd
/Users/mraharrison/Desktop/docs
$ ls -F
data/  notes.txt  book/
$ mkdir essay
$ ls -F
data/  notes.txt  book/ essay/
$ cd essay
$ cat chapter1.txt
```

A CLI is powerful but not suited to the average user. Commands are rarely intuitive and novice users need a reference guide to hand at all times. However, if you know what you're doing, a CLI can be hugely powerful. The downside is that it's also highly dangerous: a single command can change or delete all your data. As Ellen Ullman put it, "I came of technical age with UNIX, where I learned with power-greedy pleasure that you could kill a system right out from under yourself with a single command."[186]

A CLI, of course, is far simpler to implement than a GUI and requires far less memory. MS-DOS ran happily in 64KB, while Windows 3.1 required at least 1MB and Windows 95 would not run well in less than 8MB. The more recent platforms Windows 10 and Mac OS X both require at least 4GB. CLIs are now found largely on embedded devices or used for programming.

A GUI needs to keep track of all the pixels on a screen and redraw them when anything moves. (To reduce memory usage, early versions of Windows didn't

186 Ullman, E. (1998) "The dumbing-down of programming", *Salon*, link.httcs.online/ullman

animate the window contents when dragging, only showing the outline until the window was released.) Support for mouse, pointers, touchscreens, audio and video all adds up, and GUIs need more CPU cycles and more RAM, but the trade-off is a much more intuitive user experience suitable for the mass consumer market.

With the advent of Siri, Cortana, Google Assistant and Alexa, users have a new way of interacting with the computer – the voice command. Adoption has been rapid, with half of all internet searches voice-activated in 2020.[187] Voice user interfaces (VUIs) are accessible by disabled people who find traditional keyboard and mouse operation difficult. Gesture UIs are also here, with the Windows Kinect software development kit (SDK) allowing application developers to build gesture-activated apps. Combined with predictive technology, this means the way we interact with our computers is set to change dramatically in the coming years.

In real life

Most of the operating systems we're familiar with are general-purpose, multitasking OSs, able to run many different applications simultaneously. Other types of OS include real-time, embedded, distributed and multi-user. Unix and the IBM 360 and its successors are multi-user operating systems, meaning one single instance can be accessed by multiple users at the same time, through time-sharing features pioneered by Christopher Strachey at the UK's National Research Development Corporation (see chapter 9). A distributed operating system controls multiple computers and presents them as a single computer to the user. The advantage of this is that work can be shared across many devices without the user needing to handle the configuration.

An embedded operating system performs a single purpose. We heard how the original iPod was built on a minimal operating system. If a device does only one thing, there's no point installing a hefty, general-purpose OS at great cost, size and power consumption. The Apollo Guidance Computer system (see chapter 3) is often considered the first embedded system: designed for a single purpose, responding to events in real time, and unable to run arbitrary programs. One of the most popular embedded OSs – Wind River's VxWorks, launched in 1987 – has been used in a vast range of products from industrial robots to NASA's Mars Perseverance rover.

Tuning up

Chandler's laptop (see chapter 6) had a 250MB hard drive. After a period of heavy use, the hard drive would probably get quite full and fragmented. Disk fragmentation

187 Olson, C. (2016) "Just say it: the future of search is voice and personal digital assistants", *Campaign*, link.httcs.online/voice

happens because files are stored in chunks, usually 4KB in size, so when a file grows the OS grabs the next free 4KB chunk to write more data to. After a while, this fragmentation slows down the computer, as it causes excessive movement of the read/write head to load an entire file. A utility called a "defragger" puts the blocks back together, decreasing the file access time. Other utilities such as disk clean-up, backup, compression, encryption, antivirus and firewall are common on modern operating systems, along with a selection of device drivers.

When an application needs input or output from a peripheral, it requests the operating system to handle the data transfer. The OS in turn talks to a device driver, a small program that "speaks the language" of the hardware device. The device manufacturer writes drivers for all popular operating systems. Both Windows 10 and macOS come with many popular drivers pre-installed and can download many more automatically. Linux tends to delegate this responsibility to the user, making it slightly less idiot-proof, but distributions such as Ubuntu come with Windows-level driver support that helps flatten the learning curve.

TL;DR

Early computers were hardwired to perform a single program. Running a different program required extensive manual intervention. IBM created a simple operating system for the 704 in 1956 to speed-up batch processing, but a more ambitious IBM project called Stretch (1961) and Manchester's Atlas computer (1962) provided multiprogramming features for the first time, and abstracted hardware operation from the applications.

In 1964, IBM's OS/360 delivered indexed data files, program libraries, a job scheduler, interrupt handling and print spooling – the modern operating system was born. Two Bell Labs researchers, Ken Thompson and Dennis Ritchie, created Unix in 1971, which had become the most popular OS on the planet by 1980.

Apple's 1984 Macintosh was the world's first successful home computer with a graphical user interface, based on GUI prototypes seen by Steve Wozniak and Steve Jobs years earlier at Xerox PARC. Mac OS had a user-friendly interface navigated by a mouse. A year later, in 1985, Bill Gates's Microsoft released its first GUI, called Windows. Each subsequent version of Mac OS and Windows added more functionality, compatibility with additional hardware and accessibility features; mobile versions were spun off in the 21st century, including iOS and the Windows Phone OS.

The Finnish student Linus Torvalds released the first version of Linux in 1991, writing from scratch the features he most liked in Unix. Linux now runs hundreds of millions of devices, from home internet routers to Amazon's data centre servers. Linux is open-source, meaning anyone can see, copy, amend and contribute to the source code.

Operating systems come in multitasking, distributed, embedded or real-time versions, with each type suited to a different use. OSs are a type of system software that exists to manage the hardware and to allow applications and users to interact with and control the system. Utilities and drivers are also system software: utilities help keep the computer running smoothly, while drivers communicate with the hardware. Anything that is not an application is probably system software.

PCK for system software

Core concepts

- The difference between applications and system software.
- The role of system software: to control the hardware of a computer system.
- The role of an operating system as an interface between the applications and the hardware.
- Key operating system features:
  - Process scheduling.
  - Memory management.
  - File management.
  - I/O.
  - User interface.
  - Security.
- The purpose of utility software and example utilities:
  - Defragmentation or file reorganisation.
  - Encryption.
  - Compression.
  - Backups.
  - Diagnostics.

Fertile questions

- What happens when we click "print"?
- How can I run a game, browser and calculator all at once?
- How does my computer know what to do when I turn it on?
- How can I keep my computer running smoothly?
- How does the same program run on different computers without changes?

Higher-order thinking

System software use in context

Learners could be asked to list all the interactions between an application and the system software when using a computer to perform a task. For example: writing, saving and printing a Word document on Windows. The exercise could begin with learners attempting to list all the interactions in the table below. Once they've tried, you can reveal the suggested answer and compare. Lastly, learners could highlight all the system software activities in the table.

Task	System software response
Click Start icon	The mouse driver sets an interrupt flag to say there is a click event. The process scheduler puts one of the running processes on hold and executes the Interrupt Service Routine (ISR) for a mouse click. The ISR checks the location of the pointer, finds it is on the Start icon and so calls the GUI routine to display the Start menu, then returns control to the previously running process.
Click Word shortcut on Start menu	As before, for the mouse click. Then the OS finds the Word executable program in the file system, locates it on the hard drive, asks the hard disk driver to bring back the data, loads it into RAM and executes it.
Write content	The keyboard driver sets an interrupt flag for every keypress. The ISR for a keypress sends the data in the keyboard buffer to the current foreground application, which is Word. Word inserts those characters into the document at the cursor and then sets an interrupt flag to refresh the display. Display driver refreshes the display.
Save file	Word makes a request to the file manager to open the file directory. The user types a filename, the file manager checks this is valid and not a duplicate, then saves the file to disk. The hard disk driver writes the actual binary codes to the disk.

Comparing Atlas to modern systems

Learners could research the Atlas computer supervisor program described earlier in this chapter and compare the features to a modern operating system. What is the same? What is different? What is missing? Tom Kilburn et al.'s 1962 paper describing the Atlas supervisor is available online.[188]

Exploring human-computer interaction

OS user interfaces vary greatly in their usability and power. Modern OSs are designed using a branch of computer science known as human-computer interaction (HCI). Learners should research the 13 principles of HCI,[189] as described by Christopher Wickens et al., and decide which are best met by which OSs.

Analogy and concrete examples

Pizza metaphor for OS role

Various food metaphors suggest themselves for the relationship between applications, the OS and the hardware – for example, the pizza, burger or cupcake. Using the pizza analogy, the hardware is the base, the OS the sauce and the applications the toppings.

Queueing metaphors for process scheduling

The scheduler can be explained with reference to ordering fast food at a drive-through restaurant. Fast-food outlets serve meals fairly quickly to a single queue of drivers. But if a driver requests something that is made to order, they wait in a marked bay while the drivers behind are served. These bays are a holding queue for slower, bespoke orders, allowing the main queue to be processed quickly. This resembles a scheduler that places jobs on a low-priority queue if they request slow hardware resources, allowing jobs that require only CPU time to continue.

Hands-on with Linux in the browser

Experiencing different operating systems aids understanding of the topic and can dispel some misconceptions, such as confusion between applications, OSs and manufacturers (see the misconceptions table at the end of this chapter). The Raspberry Pi offers one way of doing this, with many operating systems available for free download.[190] The website distrotest.net allows learners to try many flavours of Linux, plus others such as FreeDOS, for that 1990s experience. You can try basic Linux commands in the browser at masswerk.at/jsuix, where you will also find a Commodore PET emulator and the world's first video game, *Spacewar!*

188 Kilburn, T., Payne, R.B. and Howarth, D.J. (1962) "The Atlas Supervisor", link.httcs.online/atlas

189 link.httcs.online/hci

190 link.httcs.online/pisoftware

Another browser option is webminal.org. And if learners want to find out what computing lessons were like in the 1980s (see chapter 2), they can access a BBC Micro emulator at bbc.godbolt.org. More operating system emulators and other useful links can be found at httcs.online/sys.

Exploring Android versions
Alan O'Donohoe of exa.foundation has made a 15-minute video exploring the "Android Playground" at Google's HQ.[191]

Cross-topic and synoptic

System software is closely linked to the topics of application software, translators, programming, networks and architecture.

Cross-topic with architecture, system software, languages and more
Tasking a learner with designing a computer for a purpose draws on their knowledge of architecture, memory, storage, system software and more. For example, you could set the following task:

> Design a computer for each of these users. Choose a form factor,[192] CPU, RAM, secondary storage, operating system, application software and network connection.
>
> - School student studying for GCSEs.
> - Games developer working in the office.
> - Web designer who travels a lot for work.
> - A self-driving car's onboard computer.
> - A cloud service provider's web servers.
> - A lawyer working in a home office.
> - The check-in kiosk in a clinic waiting room.
>
> See httcs.online/sys for suggested answers.

Cross-topic with security
The role of some system software is to improve security. Anti-malware includes any software that prevents or detects viruses, trojans, worms, spyware and adware. A firewall prevents unauthorised connections and network traffic. Both Windows

191 exafoundation. (2020) "#exapedition: Silicon Valley | 01 Android Playground" (video), YouTube, link.httcs.online/exapedition

192 Form factor means the size and shape of the computer, e.g. desktop, laptop, tablet.

and macOS contain built-in security software, but you could ask learners to research alternatives and compare them.

Cross-topic with networks

The speed at which we can transfer files over a network (see chapter 9) depends on the bandwidth and the file size. We can transfer files more quickly by using compression – for example, by uploading them to cloud storage or a file-sharing site. If the network allows, try copying a large file from one network location to another, or uploading to a cloud service. Then compress the file and check the transfer time again.

Discuss what has happened and ask the class how this relates to using cloud storage and streaming services. Windows and macOS both come with the ZIP utility, but advanced learners may explore the different compression utilities available as open-source or freeware sites, such as sourceforge.net.

Cross-curricular

Cross-curricular with design

Look back at the first Apple Macintosh icons designed by Susan Kare. Discuss with your learners how the Mac's GUI interface revolutionised the home computer. Ask these questions:

- Why is a GUI necessary for widespread adoption of an OS?
- What features of macOS and Windows make these operating systems popular?

Think of several stakeholders, including older people, disabled people and young children. Consider what makes a GUI important to them.

Using graph paper or a website such as makepixelart.com or piskelapp.com, ask learners to design icons for new features or apps. Try designing meaningful black and white icons in just a 16 x 16 grid, as Susan Kare did. How hard is it to make an icon that gets a message across in just 256 pixels?

Learners will appreciate the importance of user interface (UI) design and also how a very restrictive design brief can drive innovation.

Unplugged

The roles of some of the features of system software are ripe for acting out in an unplugged activity. The process scheduler, memory manager and disk defragmenter can all be acted out using a little preparation. See httcs.online for more advice.

Physical

The Raspberry Pi offers an affordable means of experiencing a Linux-based OS. Learners can experiment with the standard Raspbian interface, which offers

a Windows-like GUI, as well as a command line interface with which to get to grips with the Linux file system. Alternatively, you can install the popular Ubuntu Linux, or turn the Pi into a single-purpose machine such as a Kodi entertainment centre or the RetroPie arcade (see raspberrypi.org/software).

Project work

An after-school club where learners build or repair computers is an excellent experience that can cement understanding across the curriculum, including the role of system software.

Misconceptions

Misconception	Reality
Confusion between applications and utilities, e.g. "the word processor is a utility"	Word processors, spreadsheet programs and databases may be quite dull to some learners, but they are applications, not utilities. Learners should be clear that utilities keep the computer running smoothly, but beyond that they have no real-world usage.
Confusion between OS features and utilities, e.g. "the file manager is a utility"	File management is a feature of the operating system. The key features of an operating system include process scheduling, memory management, file management, I/O, a user interface and security. Utilities include defragmentation, encryption, backups and diagnostics.
OS is stored in ROM, or confusion of OS with BIOS	In desktop and laptop computers, the OS resides on secondary storage in modern computers. The BIOS is stored in ROM, and this includes instructions that cause the computer to load the OS from secondary storage. NB: mobile devices usually have their OS stored in flash memory, in an area known as the ROM, and this can be updated. Vendors usually upgrade the OS with an over-the-air software update.
Confusion between device manufacturer and OS, e.g. Samsung, Google and Apple all named as OSs	Android is an operating system published by Google. Google also manufactures phones. Samsung is a phone manufacturer that makes phones preloaded with the Android OS. Apple is a phone manufacturer that makes the iPhone, which runs the iOS operating system. Learners have experience of smartphones but sometimes confuse the manufacturer, the model, the brand or even the browser software.

Chapter 9.
Networks

Los Angeles, California, 29 October 1969

Student programmer Charley Kline sits nervously at a computer terminal at the University of California, Los Angeles (UCLA), supervised by Professor Leonard Kleinrock, and begins to type. Attached to his terminal is a Sigma 7 host computer, which Kline has been working on for a while now, but tonight is different. Network technicians at UCLA and at the Stanford Research Institute (SRI) have today finished installing the IMPs (interface message processors) establishing UCLA and SRI as nodes #1 and #2 on the Advanced Research Projects Agency Network, known as the ARPANET. Kline's job is to send the first message across the network.

It's 10.30pm. UCLA is ready to transmit, SRI is ready to receive, and Kline presses the first key. Just over 125 years earlier, when Samuel Morse had sent the first telegraphic message, the significance of the moment caused him to choose a biblical phrase. The message sent by Morse from Washington DC in 1844, which clacked out on paper tape in Baltimore, read "What hath God wrought?"

In 1969, however, Professor Kleinrock just wants to log on remotely to the machine at Stanford, so Kline is typing a rather more prosaic instruction: "login". But the universe has other ideas: the link fails after two letters are typed, and the first message transmitted across this new medium is once again suitably biblical. The first message ever sent over a wide-area network is simply "lo".

Pearl Harbor II

Kline's first "internet" message might never have happened had it not been for the Cold War. Twelve years before "lo" arrived in Stanford, the Soviet Union had won the race to launch the first satellite. Sputnik emitted short-wave radio pulses for three weeks in 1957 until its battery died, but the shock felt in Washington DC was seismic. US senator Lyndon B. Johnson (who became president in 1963 after the assassination of John F. Kennedy) experienced "the profound shock of realizing that it might be possible for another nation to achieve technological superiority over this great country of ours".[193] Americans were stunned, with some likening the moment to a second Pearl Harbor. President Dwight D. Eisenhower created the Advanced Research Projects Agency (ARPA) in 1958 to boost US technological innovation.

Jack Ruina, ARPA's third director (1961-63), realised that communication between the nation's increasingly technological military forces was becoming cumbersome and formed ARPA's Information Processing Techniques Office (IPTO) under the direction of Joseph Licklider. An expert in human-computer interaction and vice-president of technology company BBN (see box, below), Licklider was head of IPTO from 1962 to 1964. He initiated work on two vital technologies that would underpin the ARPANET: time-sharing and networking.

> Bolt Beranek and Newman (BBN) occupies a special place in computing history, boasting several computing pioneers among its staff, including John McCarthy and Marvin Minsky, the "founding fathers" of artificial intelligence.
>
> Originally specialising in acoustics, BBN was hired to analyse an audio recording of JFK's assassination, and the famous 18 missing minutes of the Nixon Watergate tapes. Modelling the acoustic properties of buildings and transport systems required lots of number-crunching, which drove BBN's computing innovations, including time-sharing and networking.

Lick's vision

Licklider, known as "Lick", had learned about time-sharing from the British computing pioneer Christopher Strachey, who had worked with Alan Turing at King's College, Cambridge. Strachey had patented time-sharing in 1959 for the UK's National Research Development Corporation (NRDC), and Licklider was determined to get it working in the US. Time-sharing allowed expensive computers

193 darpa.mil/about-us/timeline/creation-of-darpa

at big universities to be shared between many users at once. But Licklider's legacy as an internet pioneer was cemented by his work on networking.

Computers were already joined up within buildings and across campuses. Such local area networks (LANs) existed in many US universities in the 1960s. But Licklider was thinking bigger:

> *"It seems reasonable to envision, for a time 10 or 15 years hence, a 'thinking center' that will incorporate the functions of present-day libraries together with anticipated advances in information storage and retrieval … The picture readily enlarges itself into a network of such centers, connected to one another by wide-band communication lines and to individual users by leased-wire services. In such a system, the speed of the computers would be balanced, and the cost of the gigantic memories and the sophisticated programs would be divided by the number of users."*[194]

Licklider set out his vision in a series of memos addressed to "Members and Affiliates of the Intergalactic Computer Network",[195] which included his colleagues at BBN, plus many of the leading computer pioneers of the day at Stanford, MIT, UCLA and UC Berkeley. His imagination powered much of the early development of the ARPANET. Licklider saw the limitations of current networking technology and advanced a new technique called "store and forward packet switching".

Getting the message

Charley Kline's first ARPANET message travelled from UCLA to Stanford in "packets" of data, rather than a continuous signal. Traditional telephone lines are circuit-switched, requiring a complete electrical circuit from end to end. A circuit-switched connection – just like a child's tin-can-and-string telephone – can carry only one conversation at a time. If any part of the message is garbled, the whole thing must be resent. Breaking up data and sending it in chunks over a network, known as packet-switching, is more reliable and makes more efficient use of the network. Each node stores packets temporarily until they can be forwarded to the next node; the recipient rebuilds the message from the packets, and if any packet is missing that packet alone can be resent. Connections aren't tied up for long periods like they would be in a circuit-switched network. If a node is congested, packets can be routed around it, just as your satnav gets you around traffic jams.

---

194 Licklider, J.C.R. (1960) "Man-computer Symbiosis", *IRE Transactions on Human Factors in Electronics*, HFE-1, 4-11, link.httcs.online/licklider

195 link.httcs.online/intergalactic

Packet switching was developed almost simultaneously on either side of the Atlantic in the early 1960s. Paul Baran of RAND Corporation in the US and Donald Davies at the UK's National Physical Laboratory can both lay claim to its invention. Davies had earlier worked with Turing at the NPL on one of the first electronic computers, the Automatic Computing Engine (ACE). Davies' high profile helped popularise packet switching, and the technique was written into the design of the IMPs.

It's good to talk

In late 1969, UCLA and SRI, together with the University of California, Santa Barbara, and the University of Utah, made up the only four permanent nodes on the ARPANET. Many other institutions were already keen to get connected, and fortunately the network's design principles allowed for rapid growth. The IMPs were independent of the host computers, handling network traffic on their behalf, so although some institutions ran DEC PDPs, others IBM 360s, and some SDS Sigma machines, this didn't matter – they could all talk to each other.

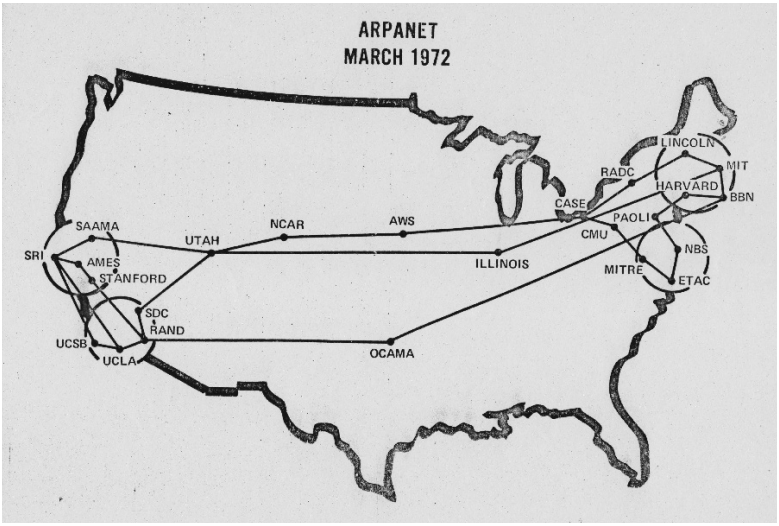


Figure 9.1: The ARPANET in March 1972 had fewer than 30 nodes.

With networking tasks abstracted from the computers into the IMPs, operating systems didn't have to change and the network could grow quickly. We take this "device-agnostic" principle for granted now: internet service providers (ISPs) don't ask you what technology you're running before sending you a router – the device

will just work with your Windows desktop, your iPhone and your games console. In the 1960s, this was a groundbreaking idea that solved the problem of incompatible hardware. By 1973 there were 40 nodes across the US, and by 1981 there were more than 200 worldwide, including in Hawaii and London. IMP design was one reason for rapid growth. Another was the ARPA team's relentless focus on protocols, one of which was to establish the ARPANET's place in internet history.

IP on everything

In 1973, computer network specialists Vint Cerf and Bob Kahn were about to turn the ARPANET into the internet – they just didn't know it yet. Cerf had been a graduate student under Leonard Kleinrock at UCLA, where he had met Kahn, an electrical engineer from BBN.

ARPA had now become DARPA, with the addition of a D for Defense. Research was progressing on packet-switched networks over radio and satellite, to enable better communications with military bases, aircraft and ships. Unfortunately, all these early networks had their own rules for communicating, with different packet sizes, naming conventions and transmission rates.

Cerf and Kahn set about designing a standard way for these networks to communicate. In September 1973, their newly formed International Networking Working Group presented a paper on the new transmission control protocol (TCP) at the University of Sussex in the UK.

In designing the new protocol, Cerf and Kahn followed the same "device-agnostic" principles underlying the IMP design, abstracting away the network protocols from the hardware. They believed that if they defined a set of rules for how devices should communicate, people could connect whatever hardware they wanted. If it spoke the same language, everything would work. With protocols but no centralised bureaucracy, Cerf guessed that the network would quickly grow organically. He was right.

> The word "protocol" comes from the Greek *prōtókollon*, which literally means "the first sheet glued on to a manuscript". This sheet would describe the contents of the document, showing readers what was to come. The word came to English via French; in late 19th century France it meant the ceremonial etiquette observed by the French head of state.

In 1981, Cerf and Kahn published a series of documents describing a stack of protocols they called TCP/IP. This included TCP to manage the device-to-device connections, and the internet protocol (IP) to route packets around the network.

On 1 January 1983, the ARPANET switched to Cerf's protocols and the internet was born. Competing protocols – different networking "languages" – fell away and TCP/IP won over the world. But this was no accident. Take this quote from a 2012 interview with Cerf by *Wired* magazine:

> "***Wired**: Are you surprised … that at this point the IP protocol seems to beat almost anything it comes up against?*
>
> ***Cerf**: I'm not surprised at all because we designed it to do that. This was very conscious. Something we did right at the very beginning, when we were writing the specifications, we wanted to make this a future-proof protocol. And so the tactic that we used to achieve that was to say that the protocol did not know how – the packets of the internet protocol layer didn't know how they were being carried. And they didn't care whether it was a satellite link or mobile radio link or an optical fiber or something else."*[196]

Cerf's big idea was so good that no matter what new technology came along, the protocol stack he designed with Kahn could still handle it. As long as the hardware speaks TCP/IP, then it can communicate across the network. (We will take a deep dive into the protocol stack using the postal service analogy in the PCK section of this chapter.)

Internet comes home

The day the ARPANET switched to Cerf and Kahn's internet protocol stack, 1 January 1983, is widely regarded as the birthday of the internet. By 1987 there were 10,000 nodes, mostly technology giants, universities and military institutions. But at the end of the decade, the first commercial services arrived on the internet and the number of users exploded. In 1989, for the first time, home users could connect their computers to the internet through internet service providers (ISPs) like CompuServe and MCI. But the internet was full of text, shared over email, file servers and bulletin boards, because the World Wide Web was yet to be invented.

As computer networks grew, a new problem surfaced. Network devices need to be able to route packets to their destination efficiently. But a complex network might have lots of routes between sender and recipient, some of them circular, causing packets to fly around the network forever without reaching their destination. Another engineer at BBN, Radia Perlman, solved this problem with her spanning tree protocol, described in a 1985 paper with a poem she called *Algorhyme*.

196 Singel, R. (2012) "We knew what we were unleashing on the world", *Wired*, link.httcs.online/cerf

I think that I shall never see
A graph more lovely than a tree.
A tree whose crucial property
Is loop-free connectivity.
A tree which must be sure to span
So packets can reach every LAN.
First the Root must be selected.
By ID it is elected.
Least cost paths from Root are traced.
In the tree these paths are placed.
A mesh is made by folks like me
Then bridges find a spanning tree.[197]

Perlman had worked with Seymour Papert in 1976 to create the Logo programming language, enabling children as young as three to program robots. She was inducted into the Internet Hall of Fame in 2014.

Remember your netiquette

The most popular service on the internet throughout the 1980s and early 1990s was an organised collection of discussion groups called Usenet, with hierarchical names that described their content, such as *rec.arts.comics* and *soc.culture.usa*. Topics that didn't fit into the "big eight" top-level hierarchies (comp, humanities, misc, news, rec, soc, sci, talk) could be discussed under the anything-goes "alt" hierarchy, where you would find groups like *alt.startrek.vs.starwars* and, yes, adult content. Usenet at this time was a fairly benign, self-regulating community of students and academics, with well-established rules of engagement called "netiquette", but all that was about to change.

Eternal September

Every September, students arriving at universities across North America would discover Usenet and be drilled in "netiquette". They would fit in with the customs or, most likely, tire of the platform and drift away, leaving the community to breathe a sigh of relief. In September 1993, however, an ISP called America Online (AOL) gave its many home subscribers, most of whom knew nothing of netiquette, access to Usenet. AOL launched an aggressive marketing campaign in the same year, distributing its software on free floppy disks and CD-ROMs and growing to one million subscribers by 1995. Users of AOL, CompuServe and many new cheap

197 Perlman, R. (1985) "An algorithm for distributed computation of a spanning tree in an extended LAN", *Association for Computing Machinery*, link.httcs.online/perlman

ISPs flooded Usenet, ignoring its "netiquette" social norms. Many long-time users lamented this development and doubted netiquette would survive. A user called Dave Fischer, writing in *alt.folklore.computers* in January 1994, said: "September 1993 will go down in net history as the September that never ended."[198]

You called it WWWhat now?

Text-based services such as Usenet, bulletin boards, file servers and email made up much of the useful content on the internet in the 1980s and early 1990s, before the invention of the World Wide Web. Home users needed a modem that used the telephone to dial an ISP, which was slow and expensive, so computer hobbyists made up much of the user community. The internet wasn't yet useful for everyone, not until a young British physics graduate made the next big leap.

In 1980, Tim Berners-Lee was working at CERN in Geneva, Switzerland, where scientists were collaborating on thousands of scientific papers at any one time and organising them was a struggle. Research meant jumping around between documents stored on lots of different servers, retrieving them using slow and clunky protocols such as file transfer protocol (FTP). Berners-Lee had an idea: what if he could just click a word to open another document? He was familiar with hypertext, first imagined by Ted Nelson in 1963 and demonstrated by Doug Engelbart of SRI in The Mother of All Demos (the name given to Engelbart's landmark 1968 computer demonstration).[199] Berners-Lee invented a coding language for hypertext, which he called hypertext markup language (HTML). He then created the first browser that could read and process HTML, which he called Enquire, initially for use only within CERN.

Of course, at this time, Cerf and Kahn had yet not written the internet protocols, so HTML was limited to CERN's local area network. Berners-Lee went off to work in the UK on computer networking, which was to be crucial experience for what happened next.

In 1984, Berners-Lee returned to CERN and continued working on HTML. By 1989, CERN had become a huge internet node. Berners-Lee had another brainwave. What if HTML worked over the internet? Then CERN scientists could more easily collaborate with other academics around the world. Talking in 2007, Berners-Lee explained that he hadn't intended to invent the web:

> *"Creating the web was really an act of desperation, because the situation without it was very difficult when I was working at CERN later. Most of the*

198 link.httcs.online/september

199 link.httcs.online/mother

> *technology … had all been designed already. I just had to put them together. It was a step of generalizing, going to a higher level of abstraction … [The web] was designed in order to make it possible to get at documentation and in order to be able to get people – students working with me, contributing to the project, for example – to be able to come in and link in their ideas, so that we wouldn't lose it all if we didn't debrief them before they left. Really, it was designed to be a collaborative workspace for people to design on a system together. That was the exciting thing about it."*[200]

Berners-Lee devised an application layer protocol for transmission of HTML, called the hypertext transfer protocol (HTTP). And he took his Enquire browser and made it better, calling it WorldWideWeb.

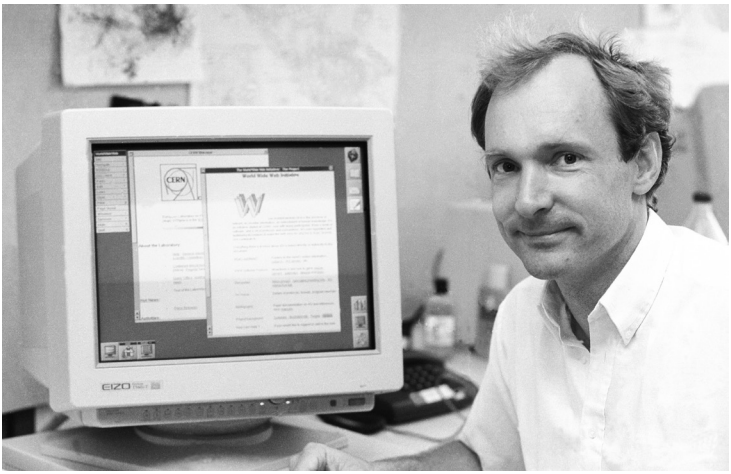


Figure 9.2: Tim Berners-Lee at his workstation at CERN, where he invented the World Wide Web.

This machine is a server

Berners-Lee also co-wrote, in the C programming language, the first web server software: HTTP Daemon, or httpd for short. In December 1990, he launched the first website – http://info.cern.ch – on a NeXT workstation complete with a sticker that said, "This machine is a server, DO NOT POWER IT DOWN!!"

200 link.httcs.online/bernerslee

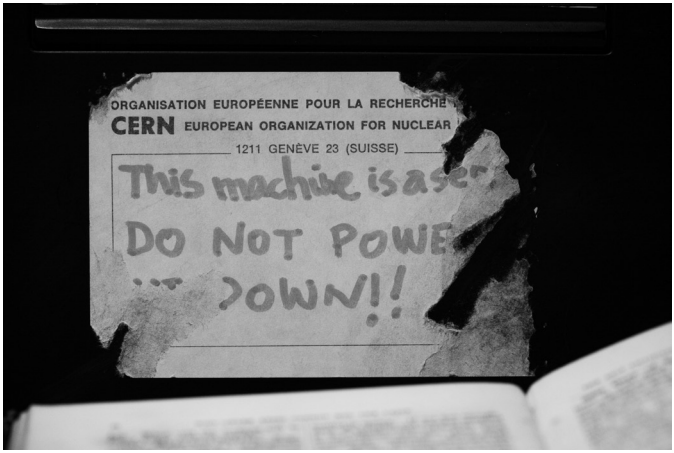


Figure 9.3: The first ever web server, complete with warning sticker.

On 30 April 1993, CERN published a statement giving up all intellectual property rights to its invention and put the source code to the World Wide Web software in the public domain for anyone to use. The user-friendly Mosaic browser was launched in the same year by the US National Center for Supercomputing Applications, and by October 1993 there were more than 500 servers on the World Wide Web. Five years later, when Google was founded, that number had grown to 2.5 million.

To understand the importance of the events described here, it's worth going back to one of the fathers of the internet, Vint Cerf. Writing for CERN in 2013 on the 20th anniversary of a free, open web, Cerf said:

> *"Although it is probably not possible to produce a precise estimate of the growth in global GDP that can be attributed to the internet and the World Wide Web, it seems likely to run into the hundreds of billions if not trillions of dollars. Some estimate that in some countries, domestic product associated with the World Wide Web may reach 8-13%. There remains a great deal of work to be done to continue to nurture and cultivate a culture of online creativity on a global scale. Internauts will be there, helping one another, inventing and sharing, as the internet and its devices and applications continue to evolve towards a future that we can only guess at today."*[201]

Cerf is right about the pace of change. Even in the few short years since he wrote those lines, we have seen internet penetration grow to around 59% of the world's

201 Cerf, V.G. (2013) "The open internet and the web", CERN, link.httcs.online/cerf2

population and witnessed the rise of voice assistants like Siri and Alexa, chatbots that help you with billing queries, home security cameras that you can view on your phone, as well as blockchain, facial recognition and AI. Although I'm not sure the term "internauts" will ever catch on.

TL;DR

In this chapter we looked back at the creation of the internet and then the World Wide Web. We learned that in the 1960s, computers in university campuses like UCLA were joined together in a local area network (LAN). Then, in 1969, the first wide area network (WAN) was created between UCLA and Stanford, as part of the ARPANET project. This network was made possible by the design of interface message processors, known as IMPs, which can be considered the first routers because they joined two LANs together to make a WAN. The principle of abstraction can be seen in the ARPANET design: the IMPs take care of networking so the computers don't have to.

These early routers implemented packet switching, the process of breaking up data into chunks and routing it across a network, with the packets potentially taking different routes and being re-assembled at the other end. This is a key strength of the ARPANET, allowing it to grow quickly and perform reliably.

To make the newly networked computers useful, ARPA's visionary director in the early 1960s, Joseph Licklider, also drove development of time-sharing – the principle of allowing multiple programs to run on one computer, which is a key feature of all modern operating systems.

In 1983, the ARPANET adopted a set of standard protocols created by Vint Cerf, called TCP/IP. Protocols are rules that enable very different computers to communicate; the protocols are arranged in layers, with each layer performing a single job. At the top is the application layer, where email sits and, later, websites displayed by the browser. Throughout the 1980s, the internet was used mostly by universities and the military to access text-only services like email, FTP and Usenet. Home users arrived on the internet in the early 1990s thanks to the first commercial ISPs, including AOL and CompuServe.

Tim Berners-Lee invented HTML in the 1980s and combined this with TCP/IP to create the World Wide Web, which was made freely available in 1993. This technology allows a browser to download and display pages from a web server anywhere in the world. The web has grown rapidly and around 59% of the world's population is now online.

It's important to draw a clear distinction between the internet and the World Wide Web. The internet is a global network of cables, satellite links, switches and routers that join computers together. The web is the collection of websites, apps and services that make use of the internet to do useful things.

In the next chapter, we will look at some of the implications of the internet and the web, including threats such as SQL Injection and malware, and developments such as cloud computing.

PCK for networks

Core concepts

- A network as a set of connected computers and devices.
- Network hardware components and how they work:
  - Hub.
  - Switch.
  - Router.
  - Network interface card (NIC).
  - Wireless access point (WAP).
  - Transmission media.
  - Unshielded twisted pair (copper wires also known as ethernet cables or Cat 5, 6 and 7 cables).
  - Fibre-optic cables.
- Wireless network connections.
- Wired vs wireless networking.
- IP address and MAC address.
- LAN vs WAN vs PAN.
- Topologies:
  - Ring.
  - Star.
  - Bus.
  - Mesh/partial mesh.
- Relationship or service model:
  - Client-server.
  - Peer-to-peer.

- The internet, the World Wide Web and the difference between them.
- Packet switching, circuit switching and the differences.
- The purpose of a protocol.
- Common protocols:
  - HTTP, HTTPS.
  - FTP.
  - TCP.
  - IP.
  - DNS.
  - SMTP.
  - POP.
  - IMAP.
- The protocol stack and the notion of layers.
- Factors affecting network performance.
- Virtual networks.
- Protecting a network from threats.

Fertile questions

- Will the internet slow down as it gets bigger and grows older?[202]
- What can I do with my phone when I have no Wi-Fi?
- How is the internet like the postal service? How is it like the telephone network?
- Why does my stream sometimes say "buffering" and how can I fix this?
- Why is the internet running out of traditional IPv4 addresses? Why is this important? What is being done about it?
- What does a small business need to do to get online?

Analogy and concrete

The postal service protocol stack

To illustrate the concept of layered protocols, I use a semantic wave approach (see page 9), first drawing an analogy with sending a birthday card to my mum. A typical postal service consists of multiple layers, like a computer network. The process begins with me writing Mum's address on the card. Then I put the card

202 Lau, W. (2017) *Teaching Computing in Secondary Schools: a practical handbook*, Routledge

in the postbox and, later, the card drops through my mum's letterbox. This is the application layer of the postal protocol stack – it's the bit we see that is useful.

Similarly, when browsing the web, I type a web address into my browser application – which might be Chrome, Edge, Firefox or Safari – and the page appears. I don't much care about what's happening in the lower layers of the stack; all I see is the application layer working and the web page being delivered by the HTTP protocol.

After I put the card in the postbox, my mum's birthday card is collected from the post office and taken to a local sorting office. We can call this the transport layer. Crucially, I don't care if this is done in a van, by bicycle, on foot, at 9am or 1pm, once or twice a day, as long as this collection service shuttles my card to the sorting office.

The mail is sorted, batched and driven in much bigger trucks to the sorting office nearest my mum. This could be called the network layer of the postal service and is another layer down in the postal protocol stack. Again, if these sorting offices move, merge or close, or if the trucks are replaced by trains or planes, I don't care. It only matters that the interface between layers – my mail moving from the transport layer to the network layer – remains intact.

From the sorting office, the post is pushed back up the protocol stack. It's sorted into batches for the postal delivery worker, who will take a bag of mail down my mum's street and pop my card through her door. Again, I don't care if the postman walks, cycles or drives; nor do I care which order the houses in Mum's street are visited. All that can change and has undoubtedly changed over time. What's important is that this layer performs its core function, which is to regularly collect mail from the previous layer and deliver it. The layer continues to interface correctly with the layer above and the layer below.

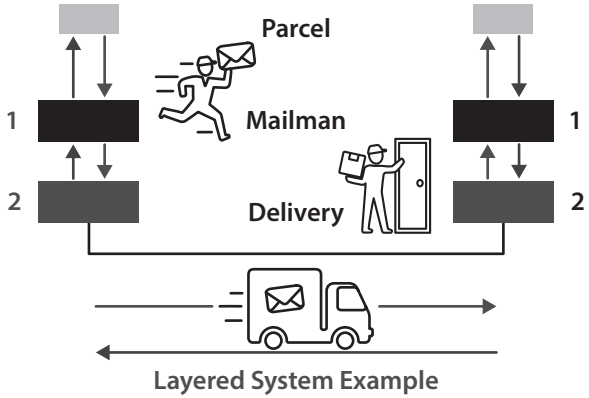


Figure 9.4: The postal system is a good analogy for the internet protocol stack.

Back up the semantic wave: the TCP/IP protocol stack in action

How does this all come together when you're surfing the web? Well, you type in a web address, then your browser uses the internet "address book", called the domain name system (DNS), to find the numeric IP address of the web server where that website lives.

The browser would now like the web page to be transmitted over application layer protocol HTTP, but we can't do that yet because we haven't established the lower layers. Currently we have a postal service with just postboxes and post offices, but no vans to move parcels around! Let's set up the lower layers now…

We need to establish the transport layer, so the browser and web server can communicate using the transmission control protocol or TCP. When we meet a new person, we say hello, introduce ourselves and get on with discussing the weather (if we're British) or our jobs (if we're American). TCP has similar rules. The protocol is so precisely defined, with so much careful error handling for unreliable 1980s internet connections, that the following joke surfaced on Usenet back in the day.

- *Hello, would you like to hear a TCP joke?*
- *Yes, I'd like to hear a TCP joke.*
- *OK, I'll tell you a TCP joke.*
- *OK, I'll hear a TCP joke.*
- *Are you ready to hear a TCP joke?*
- *Yes, I am ready to hear a TCP joke.*
- *OK, I'm about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline.*
- *OK, I'm ready to hear the TCP joke that will last 10 seconds, has two characters, does not have a setting and will end with a punchline.*
- *I'm sorry, your connection has timed out...*
- *Hello, would you like to hear a TCP joke?*

Of course, TCP is just the transport layer protocol – below that there are two more. After establishing a conversation, we exchange packets of data, which is the job of the internet layer. Here the IP ensures each packet gets to its destination using the sender address, destination address and sequential number in the packet header. If a packet is missing, the recipient will request it again. Just like in a human conversation, if we mishear a word, we will ask the speaker to repeat what they said.

But how, physically, do the data packets travel? That's the job of the data link layer. Down at the lowest layer of the stack, the packets of data are turned into electrical impulses and sent over the physical network. I guess, in our conversation model, our lips and vocal cords are doing this bit, but that would be stretching the analogy too far even for me.

Post-it packet switching

To demonstrate packet switching, we ask the learners to write messages on Post-it notes, one word per sheet, then add "To", "From" and "part x of y" on the top. They pass these notes from person to person until they reach the correct recipient. This demonstrates the principle of packet switching and its many benefits in a fun and memorable way. A detailed explanation of this activity can be found in *Hacking the Curriculum* by Ian Livingstone and Shahneila Saeed (pages 59-62).[203]

Physical

Raspberry Pi web server

With a Raspberry Pi and an SD card preinstalled with the Pi web server image, the learners can set up their own web server in the classroom, host a web page of their own making, and then connect to it from another Pi or any browser on the same network segment. This is a very powerful way of illustrating the principles of web hosting, DNS, HTML and HTTP. The National Centre for Computing Education has lesson resources for this activity.[204]

Cross-topic

Build a network for a scenario

Once learners have a clear understanding of wired and wireless technology, network devices, topologies, protocols and the client-server model, we can ask them to apply this knowledge to a real-world situation. The "build a business" exercise in the NCCE resources asks learners to do just that, drawing on their synoptic knowledge of the topic.[205]

This activity can come at the end of a series of lessons on networking to consolidate learning and drive out misconceptions before a summative test.

---

203 Livingstone, I. & Saeed, S. (2017) *Hacking the Curriculum: creative computing and the power of play*, John Catt

204 link.httcs.online/nccedns

205 link.httcs.online/nccenet

Misconceptions

Confusion between the internet and the World Wide Web is common, and this wasn't helped when the original version of Android shipped with a browser app called Internet!

It is both a blessing and a curse that learners are immersed in connected technology. Smartphones are a good example of networked computers, but the fact that they are converged devices containing many different network technologies (traditional cellular radio for calls and SMS texts, 3G/4G/5G mobile internet and Wi-Fi) can cause confusion. The ubiquity of Wi-Fi means that learners equate "internet access" with "Wi-Fi", even complaining "Sir, the Wi-Fi is down" if the wired classroom computer is offline.

Learners struggle with the distinction between browser and search engine website. They often believe that Google is the "front page" of the internet and that using the internet must begin with a Google search – this is not helped by Google's website being set to the home page on Chrome. Many students are unclear on the difference between typing a URL and typing a search term; again, browser features don't help, with the "omnibox" interface that accepts both URLs and searches via the same field.

Another misconception can arise owing to errors in textbooks and teaching materials for the network service models client-server and peer-to-peer. Some sources illustrate peer-to-peer networks as a full mesh topology, while client-server is usually illustrated with a star topology. However, there is nothing innate in the network topology that makes it a client-server or peer-to-peer network, and any topology can be set up with either service model.

Finally, learners confuse virtual networks, often called VLANs, with virtual private networks (VPNs). A virtual network is a software network running over a hardware network, entirely contained within the premises. Virtual private networks are used to create a secure "tunnel" over a public network such as the internet.

Misconception	Reality
Peer-to-peer networks are necessarily mesh or partial mesh topology, as shown in an image on this Wikipedia page: en.wikipedia.org/wiki/Peer-to-peer	Network service model (P2P or client server) is independent of the topology and star networks can be P2P, just as mesh networks can have a client-server model. The service model is an abstract concept, not related to physical connections, so using the same style of diagram for both topology and service model may lead to this confusion.
The server sits at the centre of a star network, as shown in an image on this Wikipedia page: en.wikipedia.org/wiki/Peer-to-peer	The device at the centre of a star network is a switch. The server is just another device on one of the "spokes" of the star. In the desire to illustrate an abstract concept, Wikipedia, like some textbooks, has embedded this misconception.
Internet = WWW, using the terms interchangeably or using the wrong term	The internet is a global network of networks, and the World Wide Web is the collection of websites, apps and services that uses the internet to do useful things.
Browser = search engine, such as using "Google" to mean the browser Google Chrome	The browser is a program that reads HTML and displays it on the screen. Edge, Firefox, Safari and Chrome are all browsers. Google means the Google search engine, or the company that runs it.
Internet access = Wi-Fi	Wi-Fi or wireless networking simply joins a device to a LAN via a WAP. The LAN may also have a shared internet connection, but not necessarily.
Google is the "front page" of the WWW, and all WWW usage must begin with a search	The "omnibox" interface of the most popular web browsers accepts either a URL or a search term. However, typing a well-formed URL will not cause a search to be performed and will instead open a connection to the website with that URL. But this omnibox approach has likely encouraged the misconception that all website requests begin with a search.
SMS and mobile calls use internet data	Texts via the Short Message Service (SMS) travel over standard cellular phone networks called Global System for Mobile Communications (GSM). SMS was launched in 1992. Internet Protocol (IP)-based messaging such as iMessage and WhatsApp came much later.

Misconception	Reality
Virtual networks provide secure remote access (confusion with Virtual Private Network or VPN)	A virtual network is a software network running over a hardware one. Network administrators often partition the physical network into virtual networks for security and performance reasons. Traffic that originates in one VLAN will not be visible from another VLAN even if they share physical hardware. For example, a university's physical LAN might be partitioned into student computers and staff computers, each in a different virtual network called a VLAN, despite all being attached to the same physical network. A business may divide its corporate LAN into separate VLANs for engineering, finance and HR. This is entirely different to a VPN, which creates an encrypted "tunnel" across the internet to provide secure access from a remote location to a managed network. VPNs are used by remote workers to access office LANs. The VPN is unrelated to the VLAN – they just share the word "virtual".

Chapter 10. Security

Royal Institution Lecture Theatre, London, June 1903

An expectant audience watches the physicist John Ambrose Fleming tinkering with arcane apparatus. They are waiting for a demonstration of long-range wireless messaging developed by Fleming's employer, Guglielmo Marconi (now recognised as the inventor of radio).

Marconi is 300 miles away, preparing to send a signal to London from a clifftop station in Poldhu, Cornwall. Yet, a few minutes before the official demonstration begins, the apparatus starts tapping out a message. And it's clearly not from Marconi.

RATS RATS RATS RATS …

… types the Morse code printer, set up to decode the messages arriving from Cornwall. And then, even worse, the printer begins to tap out a rude rhyme about Marconi:

THERE WAS A YOUNG FELLOW OF ITALY, WHO DIDDLED THE PUBLIC QUITE PRETTILY …

The demonstration has been hacked by the magician Nevil Maskelyne; he has been hired as a spy by the Eastern Telegraph Company, a wired telegraph provider that fears the Marconi Company will push it out of business. "I can tune my instruments so that no other instrument that is not similarly tuned can tap my messages," Marconi had boasted just a few months earlier, and Maskelyne's job today is to disprove that claim.

Eastern had no trouble recruiting Maskelyne for the hack. The magician had previously used Morse code in "mind-reading" magic tricks to communicate with a stooge. After experimenting with wireless technology, Maskelyne had hoped to make further use of it, but he was frustrated by Marconi's broad patents.

Marconi doesn't respond to the hack, but a furious Fleming writes a letter to *The Times*, asking for assistance in finding the culprit. Maskelyne happily identifies himself, saying his prank was for the good of the public, since it revealed holes in the "secure" transmission. Maskelyne has arguably become the first "white hat" hacker in history.

Figure 10.1: The stage magician Nevil Maskelyne, who hacked a demonstration by the radio pioneer Guglielmo Marconi.

Eavesdropping

Maskelyne's theatre hack was not his first attempt to expose flaws in Marconi's technology. In 1902, he had built a huge aerial and successfully intercepted radio signals from ships that used Marconi's lucrative ship-to-shore messaging system. His hacks revealed for the first time the insecurity of sending messages over distance and drove the adoption of encrypted messaging: scrambling a message so it can be understood only by the intended recipient. In precise terms, the original message, known as the plaintext, is transformed into ciphertext before being sent over an insecure network. The recipient decrypts the message to recover the plaintext. Encryption was used extensively on top of Morse code during both world wars.

Ciphers down the ages

The simplest encryption method is the substitution cipher: changing each letter for another letter or symbol. Julius Caesar is said to have shifted letters three places down the alphabet when sending messages of military importance (in cryptography, this technique is now known as a Caesar cipher). These messages could be broken at the time only by brute force: trying all possible keys until the words made sense.

Nine centuries later, a Persian scholar found a better solution. In Baghdad's House of Wisdom, Muḥammad ibn Mūsā al-Khwārizmī (see chapter 5) worked on algebra and algorithms alongside Abu Yusuf Ya'qub ibn Ishaq Al-Kindi (circa 800-870), a philosopher known for translating and expanding on the works of Aristotle and Plato. In his book on cryptography, *Risāla fī Istikhrāj al-Kutub al-Mu'ammāh* (literally *On Extracting Obscured Correspondence*), Al-Kindi described "frequency analysis" – cracking a cipher by counting how often symbols are used and comparing to typical usage in the original language.

When Mary, Queen of Scots, plotted in 1586 to assassinate her cousin, Queen Elizabeth I, she replaced letters and common words with symbols in messages to her co-conspirator Anthony Babington (see figure 10.2). Sadly for Mary, Elizabeth's spymaster Francis Walsingham and his cryptanalyst (codebreaker) Thomas Phelippes knew about frequency analysis and were able to decipher the messages.

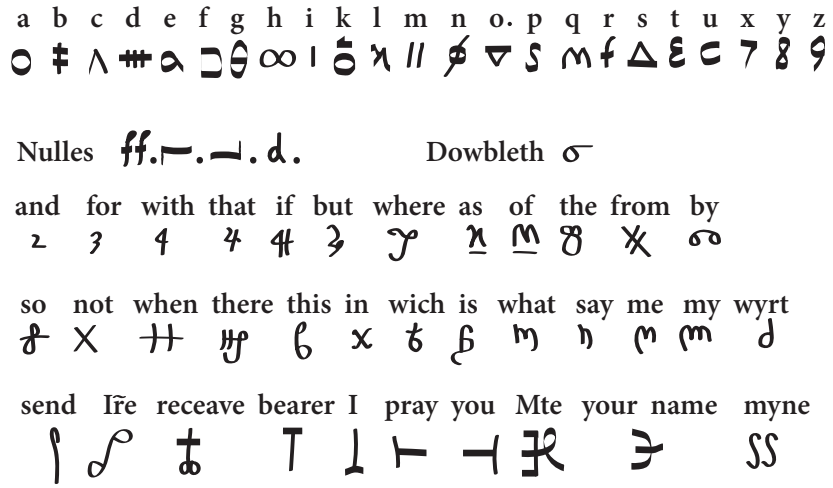


Figure 10.2: The substitution cipher used in the Babington Plot was easily broken by Queen Elizabeth I's spymaster.

At her trial, Mary proclaimed, "I would never make shipwreck of my soul by conspiring the destruction of my dearest sister." Nevertheless, the decrypted messages and Babington's confession ensured a guilty verdict and Mary was executed in 1587.

During the Crimean War (1853-56), Charles Babbage broke the Vigenère cipher, a code built on multiple Caesar ciphers that the French called "le chiffre indéchiffrable" (the indecipherable cipher). Babbage and his Victorian contemporaries were fascinated by "secret writing", which featured in many parlour games, while lovers forbidden to communicate by their families would write encrypted messages to each other via the personal columns of newspapers.

Vigenère cipher wheels and code books based on substitution methods were widely used in the First World War, but they proved clumsy and easily broken, so a "crypto arms race" ensued. Gilbert S. Vernam at Bell Labs invented the "one-time tape", in which a reel of paper tape of random letters was "added" to a plaintext message and a duplicate reel "subtracted" from the ciphertext by the receiver. This Vernam cipher, also known as a one-time pad, is equivalent to a Caesar cipher, with a different shift for each letter. The sequence of shifts is known as the "key" and if a key is used only once then the one-time pad is unbreakable.

In practice, however, the Vernam cipher was often broken because the key tapes were regularly reused or simply intercepted. To solve this problem, machines with electric rotors to do the work of the paper tape were developed almost simultaneously by the US, the Netherlands and Germany. The best-known machine was Enigma, created by the German engineer Arthur Scherbius in 1918, a full 20 years before it played its famous role in the next major conflict.

The real imitation game

In the early years of the Second World War, it was feared that German U-boats were sinking so many merchant ships bringing food, munitions and oil from North America that Britain would starve before the end of 1941. Thousands of mostly female codebreakers were working round the clock at the Government Code and Cipher School (GC&CS) at Bletchley Park in Buckinghamshire, codenamed Station X. To speed up this intelligence work, codenamed Ultra, the mathematical genius Alan Turing designed an electromechanical machine called the Bombe to crack the Enigma-encrypted German naval messages.

The enemy's ill-advised habit of including common phrases such as "weather forecast", "nothing to report" and "Heil Hitler" in their messages often gave the British codebreakers a head start. Such a greeting gifted the codebreakers a short sequence of matching plaintext and ciphertext, which they called a crib. Given this

starting sequence, Turing's Bombe would test all the possible wheel combinations that matched the crib until further German words emerged and the wheel positions were known.[206]

Thanks to German carelessness and a brilliant enhancement by fellow cryptanalyst Gordon Welchman, the Bombe (named after the earlier Polish Bomba) was able to crack Enigma messages in a matter of hours, allowing U-boats to be intercepted and protecting the shipping lanes.

It was vital to conceal the codebreakers' achievements, so British intelligence leaked false information about revolutionary long-range radar. (Ironically, radar was the subject of a previous disinformation campaign: the myth that carrots improve eyesight began with a 1939 advertising campaign to distract from innovative onboard radar being used by the RAF to repel attacks over the English Channel.)

Enigma was used by the German Navy, but from June 1941 the more complex Lorenz device was used exclusively by the German High Command. The Lorenz, nicknamed "Tunny" by the British codebreakers, was reverse-engineered by the mathematician Bill Tutte without seeing it. The head of the technological codebreaking section, Max Newman, enlisted the help of the electrical engineer Tommy Flowers to build a machine to crack Lorenz, which was nicknamed "Heath Robinson" after the illustrator of wacky contraptions. The machine proved unreliable, so Flowers set about building an electronic alternative. His Mark 1 Colossus with its 1500 valves first ran in November 1943.

Colossus could crack a Lorenz message in mere hours, which proved vital in the preparation of the D-Day landings in Normandy on 6 June 1944. Success required Hitler to believe the invasion was planned to take place hundreds of miles north-east at the Pas-de-Calais. A vast deception campaign of dummy tanks, decoy air strikes and disinformation passed by double agents proved successful.

A Colossus decrypt confirmed to the supreme commander of the Allied forces, Dwight D. Eisenhower, that Hitler wanted no additional troops sent to Normandy because he was convinced the Allies would not land there. On reading this on 5 June, Eisenhower reportedly said: "We go tomorrow." So convinced was Hitler that the Pas-de-Calais was the Allies' real target that he continued to believe the assault was a decoy invasion for several days after D-Day, refusing to send his elite Panzer tank divisions down to Normandy. As a result, the invasion was a success and turned the war decisively in the Allies' favour.

206 Cox, D. (2014) "The Imitation Game: how Alan Turing played dumb to fool US intelligence", *The Guardian*, link.httcs.online/imitation

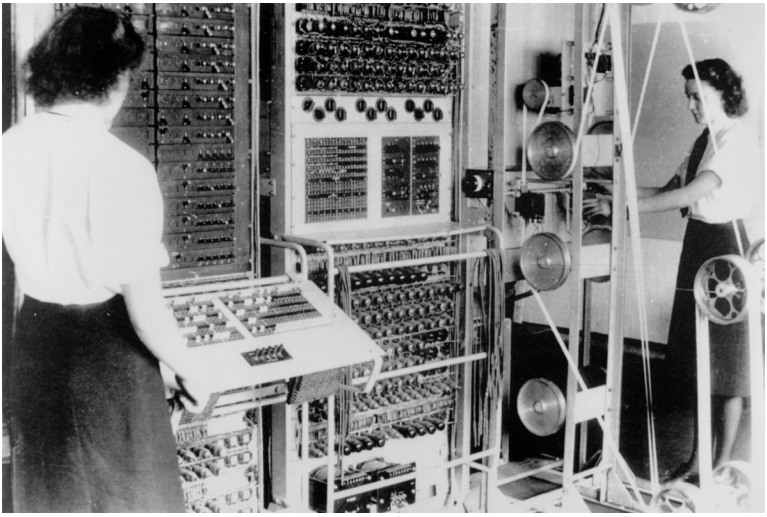


Figure 10.3: Servicewomen in the Women's Royal Naval Service, known as the Wrens, operating a Colossus Mark 2.

Ten Colossus machines were built in total, with two more under construction when VJ Day on 15 August 1945 brought the war to an end. Some analysts believe the incredible efforts of the codebreakers shaved two years off the length of the war. Most of the Colossus machines were destroyed immediately, with two moving to the new GC&CS headquarters, now known as Government Communications Headquarters (GCHQ). They were dismantled in the 1960s, and the whole Ultra project remained top secret until 1974.

Bombshell

Maskelyne's wireless hack of 1903 proved that radio communications were vulnerable to interception and jamming. In the Second World War, US radio-guided torpedoes were being disabled by noise on the same frequency as the guiding signals. A solution was proposed by an unlikely party: the Hollywood actress Hedy Lamarr and her friend George Antheil, an avant-garde composer and pianist. Lamarr suggested changing the frequency of the guidance signals rapidly, so a jamming device could not keep pace. Antheil, familiar with punched-card-driven "player pianos", suggested using piano rolls to keep the two ends in sync. Although the US Navy did not adopt the idea, "frequency hopping" is now widely used in Wi-Fi and Bluetooth implementations. In 1997, Lamarr and Antheil received the Pioneer Award from the Electronic Frontier Foundation.

Secret weapons

After the Second World War, and with computers becoming more powerful, complex mathematical encryption techniques became possible. The US adopted the Data Encryption Standard (DES) in 1976, which used a key that was 56 binary digits or bits long. DES was cracked in 1997, so a public competition was launched to find a replacement. The winners were two Belgian cryptographers, Vincent Rijmen and Joan Daemen, whose Rijndael cipher was renamed the Advanced Encryption Standard (AES). Both DES and AES are "symmetric" encryption algorithms: the same key is used for encryption and decryption, which relies on secure pre-sharing of the key.

A method of encrypting communications using a pair of keys, one of them public, was invented secretly by Clifford Cocks at GCHQ in 1973. An equivalent system was invented by three computer scientists across the Atlantic at MIT – Ron Rivest, Adi Shamir and Leonard Adleman, who described their RSA system of public key cryptography in 1977. RSA is asymmetric encryption because it uses two different keys: one for encryption and one for decryption. Furthermore, one key of the pair, named the "public key", can be shared openly, so anyone can send a private message without needing to know a shared secret. However, asymmetric encryption is much slower than its symmetric cousin, making it unsuitable for high-bandwidth applications such as streaming.

A combination of asymmetric and symmetric encryption is typically used on the World Wide Web to keep your traffic safe from hackers. When your browser connects to a secure web server using HTTPS (and you see the padlock appear next to the URL in your browser), a version of RSA called Diffie-Hellman key exchange is used to secretly create a random key known to both server and client; then the connection flips to AES using the now-shared secret key.

It's estimated that cracking an AES 128-bit key would take current computers 10^{18} years, which is millions of times the age of the universe. However, there are fears that quantum computing could successfully crack key-exchange mechanisms such as Diffie-Hellman and RSA within the next 20 years, which would cripple global communications, so the race is on to find the next big encryption advance.

Phreaks and geeks

Before digital hacking, telephone networks were under attack. In 1960s Florida, a blind student with perfect pitch called Joe Engressia worked out how to make free long-distance phone calls with a sequence of whistles. Also in the 1960s, the engineer John Draper discovered that the free whistle toys given out in Cap'n Crunch cereal boxes produced the 2600Hz tone that could be used to control the phone network.

Phone hackers such as Engressia and Draper were committing fraud, and many were arrested and some jailed. But the monopoly of the phone company AT&T meant it could charge high fees for long-distance calls, so the "phreakers" (from "phone", "free" and "freak") were lauded by their peers for their subversion.

Electronics "geeks" made phreaking devices to generate the necessary 2600Hz tones; these "blue boxes" became popular on university campuses. At the University of California, Berkeley, the top phreakers were the future founders of Apple, Steve Jobs and Steve Wozniak. Money from their blue box sales went towards buying more electronic components, with which the two Steves built their famous Apple I computer in 1976 (see chapter 6).

As the US telephone system was broken up and equipment upgraded, 2600Hz phreaking declined, but the phreaking community moved into computer hacking, trading tips on bulletin-board systems. When hackers stole passwords from naive new users of the domestic ISP America Online (AOL) by email in the mid-1990s, they called this "phishing"; they were "fishing" for gullible users, but used "ph" as a nod to phreaking.

War games

In the classic 1983 "hacker almost starts nuclear war" movie *WarGames*, the eccentric professor responsible for the rogue defence computer uses his son's name as his password. The teenage hacker just has to do a little research about Professor Falken, try "joshua" and he is in. No upper-and-lower-case rule, no numbers and definitely no confirmatory text message. Falken's poor choices would fail the most basic network security policy today. But many organisations who should know better are still vulnerable to cyberattacks on their sign-in process.

We're all familiar with the password. It's by far the most common method of authenticating yourself to a computer system, proving you are who you claim to be. A password is "something you know" – one of the three basic means of authentication. The other two are "something you have", like a mobile phone, and "something you are", like a fingerprint or retina pattern. Combining two or more of these factors massively improves security, because it is much less likely that an attacker has your password *and* your smartphone, or is able to recreate your fingerprint or retina scan accurately enough to fool the system. This is called two-factor authentication (2FA).

Without 2FA, passwords – just like encryption keys – are vulnerable to brute-force attacks that try all possible values until one works. This is likely to be successful if the password is short or matches a dictionary word. Passwords are usually encrypted before being stored, but if the hacker breaks in and steals the encrypted

password database, as happened to Adobe, eBay and LinkedIn in recent years, they can attempt to decrypt user passwords at their leisure, so strong one-way encryption and good network security is vital.

Repelling the wily hacker

"We keep hitting a damn firewall," exclaims one of the panicked engineers in *WarGames* who is trying to stop Falken's machine from launching missiles. The movie may have popularised the term "firewall", which originally referred to a barrier forming a fire break between two buildings or containing an engine in a vehicle, but evolved to mean a device to block unwanted network traffic.

As computers were being connected to the growing internet, it became important to harden them against internet-borne cyberattacks. The Digital Equipment Corporation (DEC) sold the first commercial firewall in 1992. The AT&T network analysts William Cheswick and Steven Bellovin published the book *Firewalls and Internet Security: repelling the wily hacker* in 1994.[207] Firewalls are a vital tool for securing the network perimeter – the dividing line between the equipment you own and the public network. Since the early 21st century, Windows and macOS have both had built-in software firewalls.

Worms and viruses

On 2 November 1988, much of the burgeoning internet ground to a halt. Thousands of DEC VAX machines running Unix had crashed, infected with malicious software dubbed the Great Worm. The US government put the cost at anything up to $10 million. The Worm's author, 22-year-old Robert Tappan Morris, was convicted under the 1986 Computer Fraud and Abuse Act and sentenced to three years' probation, 400 hours of community service and a $10,000 fine. Like all worms, Morris's malware replicated itself without human interaction, exploiting weaknesses in the operating system. Morris was surprised by the worm's destructive power, having intended only to highlight security weaknesses rather than cause serious damage, but his worm was too effective at using Unix's email and remote execution services to propagate.

The Morris worm wasn't the first example of malware, but it was the first to cause extensive damage. John von Neumann had imagined worms and viruses, which he called self-replicating automata, back in the 1950s, and in the early 1970s Bob Thomas at BBN (see page 174) had created a proof of concept on the ARPANET called Creeper. Thomas's worm displayed the message "I'm the creeper; catch me if

207 Cheswick, W.R. and Bellovin, S.M. (1994) *Firewalls and Internet Security: repelling the wily hacker*, Addison-Wesley

you can", before jumping to another computer and doing the same again. Thomas was forced to subsequently write the Reaper worm to eradicate Creeper.

Figure 10.4: The Creeper worm was just a proof of concept, but it was so successful on the ARPANET that the author had to write the Reaper worm to get rid of it.

Malware didn't really make the news until microcomputers started to become common in homes, schools and small businesses. In 1987, the Vienna virus caused chaos for users of IBM PCs. Vienna corrupted data and destroyed files for no apparent reason, until a German hacker named Bernd Fix wrote a program to neutralize it – the first documented antivirus software. In 1989, the first ransomware installed itself from a floppy diskette purporting to offer information about AIDS; it hid all files on the hard drive until users paid a "license fee".

Two famous email trojans, Melissa in 1999 and 2000's ILOVEYOU, used social engineering – sometimes called "hacking the human" – to spread via email. The latter arrived as an email with the subject line "ILOVEYOU"; downloading the attachment installed malware that stole passwords before emailing itself to everyone in your address book, infecting an estimated 45 million computers worldwide.

Email worms known as Klez (2001), Sobig (2003) and Mydoom (2004) are still out there replicating. These worms were written to rope computers into a "botnet" – thousands of malware-infected devices turned into zombies under the control of a "botmaster" – with the aim launching distributed denial-of-service (DDoS) attacks or sending spam emails. Mydoom is still thought to be responsible for 1% of all phishing attacks, 17 years after its creation.[208]

In 2017, the WannaCry ransomware virus spread across 150 countries, including the UK, where it caused huge disruption to the National Health Service (NHS). Around 200,000 computers worldwide were infected, mostly those running out-of-date software such as Windows XP. The virus was halted after a few days when a 22-year-old security researcher called Marcus Hutchins found the "kill switch", registering a domain that signalled the virus to stop. (In a bizarre twist, Hutchins was later arrested and convicted of writing malware himself, a crime he committed years before the WannaCry attack.)[209]

208 Gerencer, T. (2020) "The top ten worst computer viruses in history", HP, link.httcs.online/viruses

209 Greenberg, A. (2020) "The confessions of Marcus Hutchins, the hacker who saved the internet", *Wired*, link.httcs.online/hutchins

Figure 10.5: WannaCry ransomware ripped through NHS computers that were running obsolete or unpatched Windows operating systems.

These cyberattacks dramatically increased the necessity of cybersecurity, and in particular the need for spam filters, antivirus software and software updates.

Not everything is a virus

Malware comes in three main flavours. **Viruses** attach themselves to legitimate programs and replicate when launched by the user. **Worms** don't need such an interaction – they exploit features of the operating system to spread autonomously. **Trojans** disguise themselves as legitimate software or files, getting their name from the hollow wooden horse in which the Greeks concealed themselves in order to enter the city of Troy and win the Trojan War. The phrase "Trojan horse" has come to mean any trick that causes someone to invite an enemy into a protected place.

Antivirus programs work by looking for a pattern of bytes that identifies a virus, known as the "virus signature". When found, the file will be removed or quarantined so it can do no more damage. But a typical antivirus program now contains more than 250,000 signatures, with many thousands more added each

year. Coupled with the ability of modern malware to mutate, thus changing its signature to avoid detection, this means antivirus is not enough: we need to prevent the malware arriving in the first place, and fix the vulnerabilities it could exploit.

Holes everywhere

The WannaCry ransomware trojan ripped through the NHS because many computers were still running the Windows XP operating system, or an unpatched Windows 7, for want of funding for upgrades. Vendors are constantly patching holes in their software, with Microsoft shipping a slew of fixes once a month on "Patch Tuesday". After paying $3.4 billion for its software rival Macromedia, Adobe spent 15 years fixing security holes in the Flash animation software before finally declaring it dead in 2020. Any unpatched vulnerability can become a target for malware, so home users and IT managers must keep patches up to date. But sometimes technical controls alone are not enough.

On 4 April 2013, a 34-year-old man walked into a branch of Barclays Bank in North London, posing as an IT technician, and connected a small device to a computer. The man was a member of an organised crime gang and the device, a keyboard video mouse (KVM) switch attached to a 3G router, gave the gang remote control of the computer. The bank lost £1.3 million before eight gang members were arrested and later sentenced to a total of 24 years in jail.

The gang had used a £10 electronic device and some simple social engineering techniques to pull off their heist. Posing as a legitimate member of staff is known as "blagging", "pretexting" or "impersonation", and the gang also conducted "phishing" and "vishing" (voice phishing) scams when they emailed and phoned account holders seeking their credit card PINs.

Social engineering exploits human traits such as the desire to conform, be helpful or gain social status. A brilliant demonstration of a phone scam can be seen on YouTube.[210] A security researcher impersonates a reporter visiting the Defcon security conference in Las Vegas, gaining full control over the reporter's mobile phone account simply by playing the sound of a baby crying in the background and exploiting the call centre operator's desire to be helpful.

A strong network security policy and staff training are the best defences against social engineering attacks. The staff at Barclays should never have allowed "the IT guy" into the office: they should have checked his credentials and, if in doubt, called someone to verify his legitimacy. Staff vigilance against "tailgating" or "shoulder

210 Conflict International. (2017) "Hacking challenge at DEFCON" (video), YouTube, link.httcs.online/phonehack

surfing" is important, as well as shredding all documents to prevent "dumpster diving". The Barclays hackers employed these tactics for profit.

Blocking the information highway

"Christmas ruined for millions" read the headlines on Boxing Day 2014, after a second day of disruption to the Xbox Live and Sony PlayStation Network (PSN) web servers. The hacking group Lizard Squad claimed responsibility for locking out millions of gamers on the day they wanted to try out their new gifts. Using a botnet, the hackers flooded the servers with traffic, preventing legitimate connections in a DDoS attack. A Finnish Lizard Squad hacker, Julius Kivimäki, told Sky News they had caused the chaos "to amuse ourselves". (Kivimaki was convicted in July 2015 on more than 50,000 counts of computer crime and sentenced to probation.)

Four years earlier, the so-called "hacktivist" group Anonymous launched a DDoS attack against Mastercard, Visa and PayPal in support of Julian Assange, founder of WikiLeaks, whose accounts had been frozen. The attack in December 2010 prevented millions of individuals and small businesses from conducting legitimate transactions and cost PayPal alone $5.5 million. Four Anonymous members were convicted and two jailed for a total of 25 months.

DDoS is simple to perpetrate and difficult to defend against. Companies like Cloudflare make a living out of protecting web servers from DDoS attacks. They do this by placing their servers in front of the customer's web server, constantly dropping suspicious connections and blocking those IP addresses. Cloudflare then passes on the traffic to the customer's website over an encrypted connection. To ensure the encryption keys are truly random, the company has installed a bank of lava lamps in the lobby of its San Francisco office. The image from the camera trained on the lobby is used to generate a truly random number, making encryption keys unpredictable.[211]

211 Liebow-Feeser, J. (2017) "LavaRand in production: the nitty-gritty technical details", The Cloudflare Blog, link.httcs.online/lavalamps

SQL Injection

Web servers can be knocked offline by DDoS, but the no.1 attack, according to the web security organisation OWASP, is SQL Injection. To understand how this works, we need to remember that all web forms have code behind them that makes them work; when I type in my password, some code processes the data I've entered. If that code is written badly, an attacker can craft some input that causes unexpected activity, such as querying the database or even deleting data.

SQL Injection can be easily defeated by checking input data for unexpected characters – for example, nobody should have quotes or semi-colons in their password – yet attacks remain common. The most expensive cyberattack ever cost Heartland Payment Systems $140 million in compensation and 80% of its share price when attackers got hold of more than 100 million credit card accounts in 2008. And Sony was the target again when a hacker group calling itself Guardians of Peace published unreleased movies and embarrassing emails from executives in 2014 after a successful SQL Injection attack. We really must sanitise our inputs.

Bug-free by design?

The JavaScript that makes websites interactive is known as "browser-side code" because it runs in the browser. If a website does anything complex, however, it might run some code on the server instead; this is known as "server-side code" or "back-end code". If any of this code is poorly written, it could be exploited by hackers. That's why programmers often check each other's code for bugs during "code reviews", and why open-source software – with its source code visible to the whole world – is often considered more secure than proprietary code.

Designing a computer system with security built-in can reduce the number of vulnerabilities. In the 2017 action movie *The Fate of the Furious*, a car-jacker played by Charlize Theron remotely takes control of a fleet of vehicles, turning them into "zombie cars". As far-fetched as it sounds, six years earlier a team of researchers in the US revealed they had used malicious code on a CD to hack a car from the music player and shut down the engine.[212] This was only possible because the manufacturers had connected both the music player and the engine management system to the same network, called a controller area network (CAN) bus, making the car insecure by design. This illustrates the importance of defensive design.

212 McMillan, R. (2011) "With hacking, music can take control of your car", *Computerworld*, link.httcs.online/carhack

Protecting the CIA

Information security aims to protect the "CIA triad" of confidentiality, integrity and availability. Eavesdropping and data theft are attacks on **confidentiality**, damaging or changing data through malware or after an SQL Injection harms **integrity**, while DDoS reduces service **availability**. Attacks can be passive, meaning they change nothing (eavesdropping on the network traffic, for example), or active (malware and injection attacks). The key to keeping the bad guys out is to be aware of the "attack surface": the sum of all the ways, the "vectors", in which the bad guys can attack us. Just like bolting the door but leaving the windows open, it's no good having super-strong encryption, patching vulnerabilities every week and having security guards on the doors if 1% of your staff use the password "letmein".

TL;DR

Keeping secrets is as old as writing messages. Julius Caesar is said to have encrypted his messages by shifting each letter down the alphabet by a known shift key. The recipient would reverse the operation, only needing to know the key. An encryption method that changes each letter for another letter or symbol is called a substitution cipher; these are easily broken by frequency analysis, first documented by the Persian scholar Abu Yusuf Ya'qub ibn Ishaq Al-Kindi (circa 800-870).

More elaborate encryption methods were invented in the 20th century. During the Second World War, the Nazis used electromechanical machines called Enigma and Lorenz, which were cracked by expert mathematicians working with machines and early computers at the UK's Bletchley Park codebreaking centre. Modern encryption uses mathematical methods to ensure that computers cannot brute-force the key.

Before the internet was the telephone network. Students in the 1960s wishing to place free long-distance calls developed phone-hacking techniques called phreaking. The "ph" prefix persists today in terms like phishing (deceptive emails and texts) and pharming (redirecting web requests to a malicious site).

Passwords are the most common means of authentication, but a weak password can easily be brute-forced by trying all possible combinations. Passwords can also be guessed or spotted while shoulder-surfing. A second layer of protection is added by two-factor authentication or 2FA. Typically, 2FA requires a code delivered by text message or a biometric indicator such as fingerprint or face recognition.

Attacks on the network include distributed denial-of-service (DDoS) and hacking attempts. Firewalls at the network perimeter will keep out unwanted network traffic, and websites should be fortified against SQL Injection attacks.

Malicious software rose to prominence in the 1990s. Malware consists of viruses, trojans and worms. Antivirus software can protect against malware, but other security measures such as patching software, firewalls and user training are vital.

Social engineering is often called "hacking the human" and includes phishing, pretexting and shoulder-surfing. For any company, educating users is important and this should be a part of the network security policy.

Finally, defensive design means designing systems to be secure in the first place. This can include secure network design, code reviews, testing and anticipating misuse.

PCK for security

Nothing gets a class exercised like the topic of network security. "Will you teach us how to hack, Sir?" is a common refrain. Unfortunately, because of an amateur interest in hacking, the classroom can be full of misconceptions, which we will explore in the section at the end of this chapter. More advice and links to supporting documents and further reading can be found at httcs.online/sec.

Core concepts

- Definition of network security.
- Describing vulnerabilities in a networked computer system.
- Understanding cyberattacks, including automated (e.g. DDoS, malware) and manual (e.g. social engineering or hacking a website through SQL Injection).
- The anatomy of a cyberattack, including motives, vectors and impacts.
- Measures we can take to protect against cyberattacks.

Fertile questions

- If I've got antivirus software, why do I still need to patch software?
- If it's possible then it must be legal, surely?
- What should be in a network security policy?
- What would the world look like without encryption?

Higher-order thinking

Black hat or white hat?

Security experts who break into computer systems are often described as either black-hat (malicious) or white-hat (ethical) hackers. Explain these terms, then provide the following scenarios; task the learner with deciding if the hacker would be described as a black or white hat and explaining why.

- Lizard Squad's Christmas takedown of Sony PSN.
- The Anonymous attack on Mastercard, Visa and PayPal in defence of WikiLeaks.
- The Barclays KVM attack.
- Blue-box phone phreaking on campus by Steve Jobs.
- Jon Lech Johansen aka DVD Jon's cracking of DVD encryption.[213]

Analogy and concrete examples

Exploring a real-life hacking example

Many of the stories above can be used in classroom exercises. For example, you could tell the story of the Anonymous Christmas Day hack and ask questions such as:

- Why did they do it?
- How did they do it?
- What laws did they break?
- What were the impacts?
- How could we protect against these attacks?

Peer instruction: choosing countermeasures

The powerful peer-instruction technique is explained in a "quick read" blog post on the NCCE website.[214] Set a homework task before the lesson to research the WannaCry attack (see page 202). Then pose the question: which of these security measures would have been most effective against the WannaCry malware? Give these four possible answers:

1. Updating antivirus signatures.
2. Keeping operating system software up to date.

213 link.httcs.online/dvdjon

214 link.httcs.online/peer

3. Making regular backups of key files.
4. Using strong passwords and changing them regularly.

These are all valuable countermeasures against various threats. Some of them have value in this scenario, such as backups and antivirus signatures, but we know from the WannaCry story that lack of software updates was the most important factor.

Student discussion around this topic should be highly valuable as they explore the different countermeasures. After students have discussed in small groups and voted, reveal the correct answer and discuss all four, explaining how the other answers are not as good and exploring different scenarios in which they would be valuable.

Password strength checker
The tech charity SWGfL has an excellent guide to password security.[215] Show this to your class and discuss what makes a good password, then try a few values with a password strength checker such as the one from the Open University.[216]

Symmetric versus asymmetric
101computing.net has a great explainer for the two types of encryption, with an interactive tool for exploring them both.[217]

Trying out SQL Injection
SQL Injection is more easily understood if learners can try it out – obviously on a demonstration website! Some sites where students can do this are w3schools.com[218] and hacksplaining.com.[219]

Spot the phish
Learners can test their ability to spot a phishing scam via the phishing tests run by Google[220] and OpenDNS.[221]

Cross-topic, cross-curricular and synoptic

Cross topic with CT and programming
Encryption is an excellent topic to mix with computational thinking and programming. Writing a Caesar cipher is not difficult if students can code loops,

215 link.httcs.online/pwdguide
216 link.httcs.online/pwdcheckou
217 link.httcs.online/101enc
218 link.httcs.online/w3sqli
219 link.httcs.online/hacksqli
220 link.httcs.online/googlephish
221 link.httcs.online/ophish

but they may need help with the relationship between letters and numbers. In Python, you could demonstrate the **ord** and **chr** functions to move between letters and their ASCII codes and back, then set the challenge.

Here is my sample code for a Caesar cipher, also available at httcs.online.

```
plaintext=input("plaintext?").lower()   # input & make lowercase
shift = int(input("shift?"))            # input integer shift
ciphertext=""                           # initialise output
for letter in plaintext:                # iterate over string
  pos = ord(letter)                     # get the ascii code
  newpos = pos + shift                  # add the shift to it
  if newpos > ord("z"):                 # if we go beyond z...
    newpos = newpos - 26                # wrap by subtracting 26
  ciphertext = ciphertext + chr(newpos) # append new character
print("ciphertext=",ciphertext)         # output the ciphertext
```

Able coders will notice that this program does not validate its inputs and is written for positive shifts from 1 to 25 only. Fixing these issues is left as an exercise for the reader.

Another good cross-curricular coding challenge is writing a password strength checker. For example, it could check length, then count the instances of upper case, lower case and numerics and give a score. 101computing.net has an excellent tutorial.[222]

Physical

If you have access to a Raspberry Pi connected to a network, then you could create a website vulnerable to SQL Injection that the students can actually "hack". See the Raspberry Pi[223] website for how to set up a web server on a Raspberry Pi and then the Guru99 website for how to make a vulnerable web form.[224]

Unplugged

Decrypting coded messages is always popular, and I have included a couple in the Escape the Room activity on my blog.[225]

---

222 link.httcs.online/pwd101

223 link.httcs.online/rpiweb

224 link.httcs.online/sqliform

225 httcs.online/escape

Simon Johnson describes an excellent activity related to encryption in "idea 49" in his book *100 Ideas for Secondary Teachers: outstanding computing lessons*.[226] Learners must work together to decrypt a message, which turns out to be the password to unlock a laptop billed as a biological weapon!

Project work

Many competitions run each year that can be entered by teams of pupils, including CyberFirst Girls,[227] Cyber Centurion,[228] CyberDiscovery[229] and more.

Misconceptions

Misconception	Reality
A DDoS attack steals data	DDoS is an attack on availability only; a web server is disabled so genuine users cannot connect to it. No data is stolen, changed or deleted.
Virus, worm and trojan confusion, or "everything is a virus" misconception	Because the term "computer virus" captured the public imagination decades ago, that term is usually used for all malware. It gave its name to the protection software – anti-virus programs – although these programs defend against many types of software, including worms and trojans.
If your motives are good then hacking is acceptable	Often described as the "hackers' fallacy", learners may believe that probing a website's defences or attempting to guess a password is not actually unethical or illegal. They should know that just because something is possible, or no harm is intended, that doesn't make it right.
Confusion between penetration testing, vulnerability scanning and network forensics	Penetration testing is performed by an ethical hacker, who attempts to break into the system to determine weaknesses to be addressed. Vulnerability scanning means running an automated tool to check for missing software patches or other security holes. This is usually done regularly by the IT admin team. Network forensics is carried out during an attack and means analysing the data packets being transmitted on a network to identify where the hack comes from, how it is done and what data may have been stolen.

226 Johnson, S. (2021) *100 Ideas for Secondary Teachers: outstanding computing lessons*, Bloomsbury

227 link.httcs.online/cyberfirstgirls

228 link.httcs.online/cybercenturion

229 link.httcs.online/cyberdisc

Misconception	Reality
The "padlock" declares a website is trustworthy	The padlock next to the address bar in a browser denotes that a secure connection has been established between the browser and the web server using HTTPS. This means only that traffic is secured end-to-end against passive attack (eavesdropping) by third parties. It says little about the trustworthiness of the website. Any web host can purchase a digital certificate, thus giving them a "padlock" symbol. Websites serving malware use HTTPS just as often as legitimate websites. The importance of the "padlock" has been historically over-emphasised, probably because 20 years ago HTTPS was rare and malicious websites didn't bother with it, so looking for the padlock was desirable. Teachers should explain the very narrow implications of the padlock symbol and advise that it is not a reliable indicator of trustworthiness.
Encryption prevents interception or reading of messages	Encryption merely prevents the understanding of messages. When using a wireless hotspot, for example, all the data packets are visible to anyone within range and a hacker could intercept these easily. However, if HTTPS is being used, the packets will be incomprehensible because the data is scrambled.
Encryption backdoors "for the good guys only" are possible	An encryption backdoor will be found and exploited by criminals if one is ever implemented. Encryption is vital for e-commerce and general privacy. Law enforcement has argued for backdoors – a supposedly easy way to decrypt messages known only to the police and intelligence agencies – to aid the fight against crime and terrorism. But technical experts agree that there can never be a "good-guys-only backdoor".
All ciphers are substitution ciphers	Modern encryption relies on complicated mathematical models that avoid simple substitution, which is easily broken by frequency analysis. The Caesar cipher is easy to explain and even to code. But it's important to discuss other ciphers so the learners have an appreciation of strong encryption.
Confusion between anti-virus and firewall purposes	An anti-virus program checks a file for virus signatures when it is downloaded or opened. A firewall will block suspicious traffic but will not check for viruses, so if a user legitimately downloads a malware-infected file then the firewall will allow this. But hopefully the anti-virus software will detect it and prevent it from being opened.

Misconception	Reality
The Data Protection Act or GDPR criminalises hacking (confusion with Computer Misuse Act)	The Data Protection Act describes how personal data may be collected, used and stored. While it contains the principle that data must be kept securely, it makes no mention of hacking. The Computer Misuse Act describes the offences of hacking and virus distribution and associated penalties. Learners often confuse the many pieces of legislation. A number of scenario-based exercises, where they are asked "What law has been broken here?", can help.

Chapter 11.
Issues and impacts

Beitar Illit, Israeli-occupied West Bank, October 2017

The Palestinian construction worker Halawim Halawi poses by his bulldozer for a selfie. After posting it on Facebook with the Arabic caption "يصبحهم" or "yusbihuhum", which means "good morning", Halawi heads off to work. He is oblivious to the danger he is in, caused by Facebook's AI software.

That afternoon, the police arrest Halawi and question him for hours. He is utterly bewildered as the police accuse him of making threats of terrorism. "Are you planning a vehicle attack with your bulldozer?" they ask him. The police show Halawi the Facebook post and eventually the penny drops. Facebook's auto-translate algorithm has rendered his caption as "attack them" in Hebrew. Halawi explains that his words simply meant "good morning" and, after consulting an Arabic speaker, the police let him go.

Facebook has since apologised, saying: "Even though our translations are getting better each day, mistakes like these might happen from time to time."[230] With so many dialects in use around the world, Arabic is considered particularly difficult for machine translation services. Halawi deleted the Facebook post and declined to comment to the media; with Israeli police routinely monitoring the social media posts of Palestinians in order to "prevent terrorism", he didn't comment on Facebook either.

230 Hern, A. (2017) "Facebook translates 'good morning' into 'attack them', leading to arrest", *The Guardian*, link.httcs.online/halawi

Racist algorithms

Arabic is not the only language that is difficult for AI translation algorithms. Chinese and other logographic languages – those that use pictograms instead of an alphabet – are fiendishly difficult for software. In 2017, the Chinese messaging app WeChat translated *hei laowai*, a neutral phrase that literally means "black foreigner", into the n-word.[231]

But translation algorithms are not alone in being accused of racism. Several US police forces use an AI system called Correctional Offender Management Profiling for Alternative Sanctions (COMPAS) to predict the risk of an offender committing further crimes. A 2016 study showed that "black defendants were 77% more likely to be assigned higher risk scores than white defendants".[232]

A separate AI system called PredPol, used in California to predictively direct police resources to where crimes might take place, repeatedly sent officers into ethnic minority communities, regardless of the true crime rate in those areas. Researchers from the University of Utah suggested that because the software learns from reports recorded by the police rather than actual crime rates, PredPol creates a "feedback loop" that can exacerbate racial biases.[233]

In the UK, the human rights charity Liberty has criticised Durham Constabulary's use of a COMPAS-like AI system called the Harm Assessment Risk Tool, which uses data on age, gender and postcode to predict whether a person is likely to commit a crime, as well as factors like "cramped houses" and "jobs with high turnover". As Liberty points out, "The predictive programs aren't neutral. They are trained by people and rely on existing police data, and so they reflect patterns of discrimination and further embed them into police practice."[234]

In June 2020 – amid worldwide fury at the murder of George Floyd, a 46-year-old Black man, by a White police officer in Minneapolis – IBM, Amazon and Microsoft announced they would pause sales of their AI-powered facial recognition technology to police in the US.[235] A 2018 study by researchers at MIT had found that

231 Hern, A. (2017) "Chinese messaging app error sees n-word used in translation", *The Guardian*, link.httcs.online/nword

232 Larson, J., Mattu, S., Kirchner, L. and Angwin, J. (2016) "How we analyzed the COMPAS recidivism algorithm", *ProPublica*, link.httcs.online/compas

233 Reynolds, M. (2017) "Biased policing is made worse by errors in pre-crime algorithms", *New Scientist*, link.httcs.online/predpol

234 link.httcs.online/liberty

235 Heilweil, R. (2020) "Big tech companies back away from selling facial recognition to police. That's progress", *Vox Recode*, link.httcs.online/facepause

the algorithms misidentified dark-skinned women nearly 35% of the time, while nearly always getting it right for white men.[236] Alison Powell, a data ethicist at the Ada Lovelace Institute in the UK, told *New Scientist* magazine: "Face-recognition systems have internal biases because they are primarily trained on libraries of white faces. They don't recognise black and brown faces well."[237]

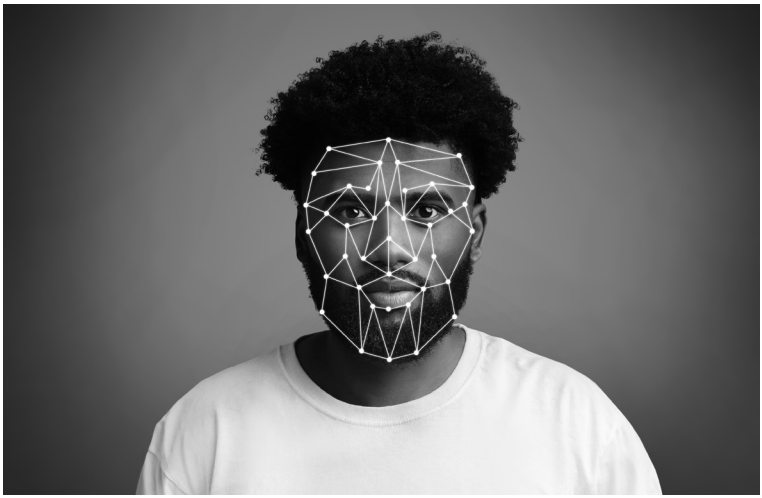


Figure 11.1: Big technology companies have paused sales of facial recognition software, citing the technology's potential for abuse or misuse.

Privacy

Databases fed by facial recognition data are a new way in which technology might infringe on our privacy and civil liberties. But tech companies already know plenty about us, and what they know is growing rapidly. I downloaded my Google data recently and it came to over 2GB, not including tens of GB of photographs that I happily share with the data giant because I like to backup and share my memories.

On Facebook I have another several GB of data, including family photos, groups and pages that I like, and a friends list. I'm happy to trade a piece of my privacy for the convenience of chatting with friends and colleagues, but I admit I came close to deleting my account in the wake of the Cambridge Analytica data scandal. Like

236 link.httcs.online/faceerrors

237 Stokel-Walker, C. (2020) "IBM gives up on face-recognition business – will other firms follow?", *New Scientist*, link.httcs.online/ibmface

many, I decided to stay for the convenience of sharing family pictures with my relatives and chatting with computing teachers in various groups.

To live in the modern world means to choose where to draw that dividing line between private and shared data; to choose which technology companies you trust and how much of your life you trust them with. As IBM puts it in a 2018 report, "Personal data is the new currency of the digital economy."[238] But all that personal data swilling around carries some risk, with more than 223,000 people in the UK reporting identity fraud to the fraud prevention organisation Cifas in 2019.[239] And as Halawim Halawi found out, where you draw that privacy line can sometimes have catastrophic results.

Old boys' (neural) network

In 2014, Amazon began developing an AI recruiting tool designed to identify the characteristics of existing high-performing employees and score incoming CVs against those standards. "Everyone wanted this holy grail," an insider told Reuters. "They literally wanted it to be an engine where I'm going to give you 100 resumes, it will spit out the top five, and we'll hire those."[240]

The computer models were trained on CVs submitted to the company over a 10-year period. But most were from men – a reflection of male dominance across the tech industry that caused Amazon's algorithm to disproportionately recommend men. In effect, because of historic human bias in the tech industry, the system taught itself that male candidates were preferable. Amazon initially tried to correct for this bias, but abandoned the project in 2018.

On 7 November 2019, the programmer David Heinemeier Hansson, creator of the Ruby on Rails web development software, complained on Twitter about gender bias in credit scoring by Apple:

> *"The @AppleCard is such a f***ing sexist program. My wife and I filed joint tax returns, live in a community-property state, and have been married for a long time. Yet Apple's black box algorithm thinks I deserve 20x the credit limit she does. No appeals work."*[241]

238 Fox, B., Gurney, N., Cavestany, M. & van den Dam, R. (2018) *The Trust Factor in the Cognitive Era*, IBM Institute for Business Value, link.httcs.online/ibmtrust

239 Cifas newsroom. (2020) "Cifas reveals cases of identity fraud up by nearly a third over last five years", link.httcs.online/cifas

240 Dastin, J. (2018) "Amazon scraps secret AI recruiting tool that showed bias against women", Reuters, link.httcs.online/amazonai

241 link.httcs.online/dhhrant

His complaint caught the attention of the authorities. The New York Department of Financial Services launched an investigation that concluded no laws had been broken, but that Apple should improve the transparency of its credit scoring process and laws should be reviewed.

One of Hansson's biggest frustrations was the company's inability to "look inside the black box" and explain the algorithm's decision. Increasingly, machine learning algorithms are spitting out results that their human operators cannot understand, much less explain. Trust in the algorithm makes us loath to question its authority. When the computer says no, humans are reluctant to overrule it and quick to believe the algorithm's decision is superior to their own.

Sexism and racism in algorithms will be one of the IT industry's defining challenges in the coming decade. Caroline Criado Perez, in her book *Invisible Women: data bias in a world designed for men*, writes, "The presumption that what is male is universal is a direct consequence of the gender data gap."[242] And in *Algorithms of Oppression: how search engines reinforce racism*, Safiya Umoja Noble predicts that "artificial intelligence will become a major human rights issue in the twenty-first century".[243]

With great power…

Google performs 90% of the searches made from desktop computers. We trust the search giant to return what we're looking for 3.5 billion times per day.[244] But, as the Swedish technology journalist Andreas Ekström explained in a 2015 TED talk,[245] the results may be more biased than we think.

In 2009, Michelle Obama was the victim of a racist campaign. An image of the US first lady with monkey features was distributed widely over the internet; the offensive picture was posted online many times with the caption "Michelle Obama" and the filename "MichelleObama.jpeg" in order to game Google's PageRank algorithm. Google acted, writing some filtering code, and within a few days the image no longer appeared on the first page of results.

In 2011, Norway's worst ever terrorist attacks resulted in the tragic deaths of 77 people, many of whom were teenagers attending a summer camp. The terrorist, Anders Behring Breivik, had emailed his far-right anti-Muslim "manifesto" to 1000

242 Criado Perez, C. (2019) *Invisible Women: data bias in a world designed for men*, Abrams

243 Umoja Noble, S. (2018) *Algorithms of Oppression: how search engines reinforce racism*, NYU Press

244 link.httcs.online/googleusage

245 Ekström, A. (2015) "The moral bias behind your search results", TEDxOslo, link.httcs.online/ekstrom

people just before the attacks, hoping for it to go viral. But a Swedish web developer named Nikke Lindqvist had other ideas. According to Ekström, Lindqvist told his followers to "go out there on the Internet, find pictures of dog poop on sidewalks, publish them in your feeds, on your websites, on your blogs. Make sure to write the terrorist's name in the caption. Make sure to name the picture file breivik.jpeg. Let's teach Google that that's the face of a terrorist."[246] And it worked. If you googled "Breivik" for weeks after the attacks on 22 July, images of dog poo came high up in the search results. And this time Google did not intervene.

Few people would criticise Google's conduct: most would judge Michelle Obama an honourable person and Anders Behring Breivik a despicable person. But who are Google to decide who's honourable and who's despicable? We see here the huge influence of Google. As Ekström says, "There's only one power-player in the world with the authority to say who's who. 'We like you, we dislike you. We believe in you, we don't believe in you. You're right, you're wrong. You're true, you're false. You're Obama, and you're Breivik.' That's power if I ever saw it."[247]

Autopilot overpromises

Biased search results may be described as censorship but are unlikely to be a life-or-death matter. Not so for high-tech carmaker Tesla. For a century or more, technology's promise has been to take the drudgery out of life, and few sectors are more suitable for automation than transport. But the roads are notoriously unpredictable, as early adopters of driver-assist technology are finding to their cost.

In March 2018, 38-year-old Apple employee Walter Huang was playing a game on his company-issued iPhone when his Tesla drove straight into a barrier. Huang was taken to hospital but later died of his injuries. He had been playing the game while his car was driving in Autopilot mode, which critics say should be renamed. As of March 2021, US regulators are investigating 23 separate accidents involving Tesla's driver-assist technology.

Bryant Walker Smith, a professor at the University of South Carolina, told *The New York Times* in 2021 he was concerned about the name Autopilot and about how Tesla's marketing – which now includes the phrase "full self-driving capability" – suggests drivers can safely turn their attention away from the road. "There is an incredible disconnect between what the company and its founder are saying and letting people believe, and what their system is actually capable of," Walker Smith said.[248]

246 Ibid.

247 Ibid.

248 Boudette, N.E. (2021) "Tesla's Autopilot technology faces fresh scrutiny", *The New York Times*, link.httcs.online/teslasafety

Inside the black box

In 2015, at New York City's Mount Sinai Hospital, Deep Patient software was trained on data from 700,000 patients, after which its accuracy in predicting diseases such as severe diabetes, schizophrenia and various cancers was exceptional. But, unfortunately, the software was unable to give doctors any clue as to how it came to its decisions. Without a rationale with which to reassure patients, doctors were unable to justify any changes to a patient's medical prescriptions. The inability to see into the black box rendered the overall system next to useless.

So, if algorithms have all these problems – perceived bias, lack of transparency, and an unhealthy air of authority even when offering spurious results – why are they so popular? The answer is: because they often work spectacularly well. In 2016, the IEEE International Symposium on Biomedical Imaging in Prague ran an AI competition to detect cancer cells. The winning system, Harvard's deep-learning algorithm PathAI, achieved 92% accuracy, compared with human pathologists' 96%. However, when the algorithm paired up with the doctors, performance rose to a staggering 99.5%.[249]

"Our guiding hypothesis is that 'AI plus pathologist' will be superior to pathologist alone," said Dr Andrew Beck, who led the creation of Harvard's system.[250] The power of AI to look for patterns in images, medical records or DNA data – investigations that would take humans aeons and traditional programs centuries – is driving a revolution in healthcare, with spending on AI tools set to top $34 billion by 2025.[251]

The social dilemma

During the early 2010s, the "Arab Spring" uprisings across North Africa and the Middle East led to regime change in many countries. Journalists latched on to the protesters' social media use and dubbed the movement a "Twitter uprising" or "Facebook revolution". In the words of one Egyptian protester, Fawaz Rashed, "We use Facebook to schedule the protests, Twitter to coordinate, and YouTube to tell the world."[252]

In 2012, Facebook added an option for users to share their organ-donor status on their timelines. Nearly 60,000 users did so on the first day. Social host-finding

249 Wanjek, C. (2016) "AI boosts cancer screens to nearly 100 percent accuracy", *Live Science*, link.httcs.online/pathai

250 Ibid.

251 Bresnick, J. (2018) "Healthcare artificial intelligence market to top $34B by 2025", *HealthITAnalytics*, link.httcs.online/healthai

252 Shearlaw, M. (2016) "Egypt five years on: was it ever a 'social media revolution'?", *The Guardian*, link.httcs.online/arabspring

site Couchsurfing was enabling low-budget travelling years before Airbnb made a billion-dollar business out of the idea. Social media is now routinely used around the world in the search for missing people, with some success. While social media can undoubtedly be a force for good, raising awareness of health campaigns, connecting divided people and amplifying the voice of the individual, its vast reach can also be exploited by governments and corporations in order to manipulate public opinion.

In the run-up to the UK's Brexit referendum on EU membership, the official Vote Leave campaign paid £2.7 million for targeted ads on Facebook. The ads included emotive but fact-free claims including "The European Union wants to kill our cuppa" and "The EU blocks our ability to speak out and protect polar bears!"[253] The ads were created by Aggregate AIQ, a Canadian company with links to Cambridge Analytica, the consulting firm that harvested private data from Facebook users.[254]

The QAnon conspiracy theory in the US, which began in 2017 on the imageboard 4chan, has swelled to hundreds of thousands of people who believe a completely unfounded narrative that Donald Trump is waging a secret war against elite Satan-worshipping paedophiles in government, business and the media. In 2019, QAnon followers shared a bizarre interpretation of an innocuous tweet by the former FBI director James Comey, claiming that Comey was planning a "false flag" attack on a California school fundraiser. The claims went viral, causing hundreds of people to "warn" the school. Despite the lack of any credible threat, the school cancelled the event out of "an abundance of caution".[255]

Many QAnon followers came straight from the 2016 alt-right Pizzagate conspiracy theory, which falsely claimed that there were coded messages in the hacked emails of John Podesta, who was Hillary Clinton's campaign manager for the 2016 presidential election. Pizzagate proponents claimed that Podesta's "coded messages" connected high-ranking Democratic Party officials with an alleged child sex ring operating from the basement of a pizzeria in Washington DC. Edgar Welch, a 28-year-old man from North Carolina, decided to self-investigate the theory, arriving at the pizzeria with an semi-automatic rifle and firing three shots.[256]

253 BBC News. (2018) "Vote Leave's targeted Brexit ads released by Facebook", link.httcs.online/leaveads

254 Davies, R. and Rushe, D. (2019) "Facebook to pay $5bn fine as regulator settles Cambridge Analytica complaint", *The Guardian*, link.httcs.online/facebookfine

255 BBC News. (2019) "QAnon conspiracy theory on James Comey shuts school festival", link.httcs.online/qanonschool

256 Yuhas, A. (2017) "'Pizzagate' gunman pleads guilty as conspiracy theorist apologizes over case", *The Guardian*, link.httcs.online/pizzagate

In 2019, the FBI issued a bulletin warning that conspiracy theory-driven domestic extremism was a growing threat.[257] The US Combating Terrorism Center has described QAnon as a "novel challenge to public security".[258] And when rioters stormed the US Capitol on 6 January 2021, in an effort to overturn Donald Trump's election defeat, all eyes turned to social media, where Trump supporters had been discussing the protest on QAnon-related pages for weeks.

Back in the UK, as Covid-19 spread during 2020, social media conspiracy theorists had a field day, blaming the virus on 5G mobile masts, calling it a deliberate attempt to control the world's population, and claiming the vaccine program was a secret plot to "chip" us all.[259] A "troll farm" in the Russian city of St Petersburg allegedly employs young people to spread misinformation and sow division using social media, with the intention of destabilising society in the US and Europe.[260]

Growing concerns over the prevalence of misinformation and other harmful online content – such as pornography, child abuse images, gambling promotion, suicide promotion and extremist material – have forced governments to act. The UK's online harms bill, still being debated at the time of writing, will give the communications regulator Ofcom the power to levy unprecedented fines of up to 10% of global turnover, which would cost Facebook, for example, £5 billion if it failed to limit the spread of harmful content. Germany already has similar tough legislation, as does Australia. With the legislation not expected to come into force in the UK until 2024, it remains to be seen whether anything is changed by redefining social media as a quasi-public space where user safety must be paramount.

Energy

Elon Musk's Tesla very publicly bought $1.5 billion of the cryptocurrency Bitcoin in February 2021, making a profit of more than $900 million when Musk's own high-profile interest pushed the price of Bitcoin sky-high. The decentralised currency – based on solving cryptographic problems whose solutions are rare – was launched in 2009 and has created many "Bitcoin millionaires", with a single Bitcoin rising in value from 30 cents in 2011 to $60,000 in 2021. But Bitcoin has a dirty secret (quite apart from its notorious popularity with drug dealers and money launderers).

257 Wilson, J. (2019) "Conspiracy theories like QAnon could fuel 'extremist' violence, FBI says", *The Guardian*, link.httcs.online/fbi

258 Amarasingam, A. and Argentino, M-A. (2020) "The QAnon conspiracy theory: a security threat in the making?", *CTC Sentinel*, 13(7), link.httcs.online/ctcqanon

259 Goodman, J. and Carmichael, F. (2020) "Coronavirus: 5G and microchip conspiracies around the world", BBC News, link.httcs.online/5gconspiracy

260 Lee, D. (2018) "The tactics of a Russian troll farm", BBC News, link.httcs.online/russiantrolls

The computing power needed to "mine" the currency, coupled with that needed to add transactions to the encrypted "blockchain" that acts as a secure ledger, means Bitcoin now exceeds the electricity consumption of Argentina.[261] Meanwhile, data centres run by Google, Facebook, Amazon, Microsoft and Apple are thought to account for 2% of all US electricity consumption; Sweden's Anders Andrae, sustainable ICT expert at Huawei, predicts that data centres are likely to gobble up 8% of the world's electricity by 2030.[262] And that new iPhone in your pocket? It has a carbon footprint of around 78kg of CO_2.[263]

Our rare earth

Energy is not the only environmental issue. The entire continent of Africa currently produces about 2.9 million tonnes (Mt) of electronic waste annually, according to the UN, amounting to just 5.4% of the world's e-waste mountain.[264] But Nigeria alone is the destination for more than 1Mt of e-waste every year, much of which is imported illegally from North America and Europe. Once there, valuable metals are recovered in hazardous conditions, often by children with their bare hands, using furnaces and chemicals. Burning insulated cables and circuit boards to extract copper and aluminium, these young people are exposed to highly poisonous dioxins and heavy metals on a daily basis. Meanwhile, the value of rare elements needed for phones and tablets encourages exploitative mining practices. In 2016, Unicef estimated that 40,000 children were working in mines across the Democratic Republic of Congo, mostly digging cobalt for lithium batteries.[265]

261 Criddle, C. (2021) "Bitcoin consumes 'more electricity than Argentina'", BBC News, link.httcs.online/bitcoinenergy

262 Jones, N. (2018) "How to stop data centres from gobbling up the world's electricity", *Nature*, link.httcs.online/datacentres

263 link.httcs.online/iphoneenergy

264 Forti, V., Baldé, C.P., Kuehr, R., Bellink, G. (2020) *The Global E-waste Monitor 2020: quantities, flows and the circular economy potential*, httcs.online/ewaste

265 Wakefield, J. (2016) "Apple, Samsung and Sony face child labour claims", BBC News, link.httcs.online/congo

Figure 11.2: Low-income Ghanaians endure harsh conditions to recover metal from electronic waste in the Agbogbloshie suburb of Accra.

Lawmakers and lawbreakers

Exporting e-waste to developing nations is actually illegal under the EU's Basel Convention on hazardous waste, but a two-year investigation revealed the practice was still widespread – and the UK was Europe's worst offender.[266] Disposal of electronic waste has been regulated in the UK since 1991. The current law, the Waste Electrical and Electronic Equipment Regulations 2013, is part of the important legislation that governs the IT industry.

The Computer Misuse Act 1990 criminalised hacking for the first time, prompted by the failed prosecution of two hackers in the mid-1980s. Robert Schifreen and Stephen Gold broke into the pre-internet "Prestel" viewdata service, run by BT, and accessed the mailbox of Prince Philip. The 30-year-old act has been criticised recently for "[preventing] investigators from dealing effectively with online threats while overpunishing immature defendants".[267] US legislators are considering

266 Laville, S. (2019) "UK worst offender in Europe for electronic waste exports – report", *The Guardian*, link.httcs.online/ukwaste

267 Bowcott, O. (2020) "Cybercrime laws need urgent reform to protect UK, says report", *The Guardian*, link.httcs.online/cybercrime

legalising "hack back" laws that allow companies to retaliate to online attacks.[268] In response to the issues of bias and transparency, the UK government created the Centre for Data Ethics and Innovation in 2018; in 2020, the CDEI recommended that the developers of algorithms be obliged to evaluate their impacts on equality.[269]

The UK's Data Protection Act 2018 enacted the EU's General Data Protection Regulation (GDPR) to bring data privacy laws up to date, superseding the 1998 Data Protection Act and adding new accountability and reporting obligations to companies that hold our data. The Copyright, Designs and Patents Act 1988 gives creators of digital media the right to control how their work is used and distributed. Music, books, videos, games and software are all covered by copyright law.

The Investigatory Powers Act 2016 gives sweeping powers to UK security services, including the right to intercept communications and collect browsing histories to fight crime; critics have dubbed the act a "snooper's charter" for its broad expansion of police powers. One thing everyone agrees on is that our vital computerised services need regulation, but debates continue to rage about what that should look like, driven largely by the inevitable conflict between individual freedom and the common good.

What now?

"The Fourth Industrial Revolution" was the theme of the 2016 World Economic Forum Annual Meeting in Davos, Switzerland. After steam, electricity and computers, our new age of mobile internet, automation and AI brings boundless possibilities but also unprecedented challenge. Computer scientists must work with legislators to ensure technology is a force for good. Algorithmic bias, harmful content, disinformation and radicalisation are huge challenges that need an informed, intelligent response. As Hannah Fry writes in her book *Hello World*:

> *"Imagine that, rather than exclusively focusing our attention on designing our algorithms to adhere to some impossible standard of perfect fairness, we instead designed them to facilitate redress when they inevitably erred; that we put as much time and effort into ensuring that automatic systems were as easy to challenge as they are to implement. Perhaps the answer is to build algorithms to be contestable from the ground up. Imagine that we designed them to support humans in their decisions, rather than instruct them. To be transparent about why they came to a particular decision, rather than just inform us of the result."*[270]

268 Ibid.

269 Thomas, A. (2020) "Accountability for algorithms: a response to the CDEI review into bias in algorithmic decision-making", Ada Lovelace Institute, link.httcs.online/cdei

270 Fry, H. (2019) *Hello World: how to be human in the age of the machine*, Transworld

The tech magazine *ZDNet* has predicted the rise of AI-powered malware, which can learn how to get around our defences, and "low-code" app development platforms to allow non-programmers to create apps.[271] Virtual reality, augmented reality and extended reality are set to become big, and the Internet of Things will reach 25 billion devices by 2030. But the field of AI is a cause for growing concern. Ray Kurzweil, now Google's foremost expert on natural language processing, famously predicted the "AI singularity" in his 2005 book *The Singularity is Near*.[272] In 2009, he told *Big Think*, "By 2045, we'll have expanded the intelligence of our human machine civilization by a billionfold. That will be singularity, and we borrow this metaphor from physics to talk about an event horizon it's hard to see beyond."[273]

In 2015, alarm bells began ringing even louder, as Stephen Hawking and Elon Musk put their signatures to an open letter warning against the "pitfalls" of AI research:

> *"The potential benefits [of AI] are huge, since everything that civilization has to offer is a product of human intelligence; we cannot predict what we might achieve when this intelligence is magnified by the tools AI may provide, but the eradication of disease and poverty are not unfathomable. Because of the great potential of AI, it is important to research how to reap its benefits while avoiding potential pitfalls."*[274]

In my recruitment speeches to students choosing their options in both Year 9 and Year 11, I exhort them to choose computer science and "join the fight against the robot apocalypse". Maybe that humorous challenge is not actually so far-fetched.

Making predictions about technology has a long and chequered history. In 1903, the president of the Michigan Savings Bank advised Henry Ford's lawyer, Horace Rackham, not to invest in the Ford Motor Company, saying, "The automobile is a fad, a novelty. Horses are here to stay." In 1977, Ken Olsen, founder of Digital Equipment Corporation, said, "There is no reason an individual would ever want a computer in their home."[275] But other predictions have proved startlingly accurate.

In a 1945 article for *The Atlantic* magazine, Vannevar Bush, chief engineer to US president Harry Truman, predicted the connected desktop computer, which

---

271 Leprince-Ringuet, D. (2020) "10 tech predictions that could mean huge changes ahead", *ZDNet*, link.httcs.online/10predictions

272 Kurzweil, R. (2005) *The Singularity is Near*, Penguin

273 Big Think. (2009) "Ray Kurzweil: the coming singularity" (video), YouTube, link.httcs.online/singularity

274 link.httcs.online/hawkingletter

275 (2016) "Worst tech predictions of all time", *The Telegraph*, link.httcs.online/badpredictions

he called the "memex", plus memory cards, speech-to-text software and digital photography. The most striking passage describes a user browsing their memex and connecting articles with "trails", foreshadowing the World Wide Web:

> *"First he runs through an encyclopedia, finds an interesting but sketchy article, leaves it projected. Next, in a history, he finds another pertinent item, and ties the two together. Thus he goes, building a trail of many items … Wholly new forms of encyclopedias will appear, ready made with a mesh of associative trails running through them, ready to be dropped into the memex and there amplified."*[276]

Predictions for the coming decade published by the Pew Research Center in Washington DC recognise the huge social change brought about by the internet. Experts agree that a reckoning is coming for social media, which might include breaking up large tech companies, a greater focus on privacy and action on harmful content. Pew's report[277] warns that totalitarian countries will increasingly use technology to manipulate and control their populations. On the plus side, we can expect natural language processing to accelerate until Douglas Adams' Babel Fish[278] becomes a reality. An explosion in "telemedicine" and wearable healthcare devices will tackle conditions from diabetes to depression.

Writing for *The Guardian* in 2021, Emily Segal of the technology and culture thinktank Nemesis presented 20 predictions for the coming decade, including:

- Universal work-from-home policies, after widespread adoption during the Covid-19 pandemic.
- "Waist-up" fashion that makes an impact on Zoom calls.
- Biometric payments, or "pay with your face" technology.[279]

Whether or not the robot apocalypse comes before we can solve climate change, it's clear that the world's major issues in 2021 can either be worsened by technology or ultimately solved. It's our job as teachers to empower the next generation to be part of the solution, not part of the problem.

---

276 Bush, V. (1945) "As we may think", *The Atlantic*, link.httcs.online/aswemaythink

277 Vogels, E., Rainie, L. and Anderson, J. (2020) "Experts predict more digital innovation by 2030 aimed at enhancing democracy", Pew Research Center, link.httcs.online/innovations

278 link.httcs.online/babelfish

279 Segal, E. (2021) "What will the next decade bring? Here are 20 predictions from trend forecasters", *The Guardian*, link.httcs.online/segalpredictions

TL;DR

Information technology caused a "third industrial revolution" and analysts are calling the convergence of mobile internet, automation and AI the "fourth industrial revolution". With all new technology comes both opportunities and challenges.

We face privacy, legal, cultural, environmental and ethical questions, and many issues span two or more of those categories, such as automation, spam and viruses. Most technology decisions require the balancing of competing issues and impacts – for example, automation drives down the cost of production and eliminates hazardous occupations, but can cut jobs or worsen inequality. The internet has created opportunities for communication that were previously impossible, but has created a digital divide between those with access and those without.

Artificial intelligence is opening up huge possibilities in fields as diverse as healthcare, transport and the arts, but there are fears over bias, discrimination and lack of transparency. Cryptocurrencies such as Bitcoin have been criticised for their energy use. Electronic waste is a growing ethical, environmental and legal issue, while finite resources needed in smartphones are mined by low-paid workers in exploitative practices.

In every question about the issues and impacts of technology, we must consider all the stakeholders involved, including the creators, vendors, shareholders, consumers and wider society.

PCK for issues and impacts

Core concepts

- Key terms:
  - Privacy.
  - Legal.
  - Ethical.
  - Environmental.
  - Cultural.
  - Stakeholder.
- Relevant legislation:
  - General Data Protection Regulation (GDPR) and Data Protection Act 2018.

    - Computer Misuse Act 1990.
    - Copyright, Designs and Patents Act 1988.
    - Creative Commons licensing.
    - Investigatory Powers Act 2016.
    - Freedom of Information Act 2000.
- Distinguish between creative uses and copyright infringement.
- Digital divide.
- Positive and negative aspects of mobile technology.
- Implications of having personal data online.
- Social and environmental impacts of social media.
- Positive and negative effects of online content.
- Environmental effects of technology.
- Ethical and cultural impacts of AI and algorithms.

Fertile questions

- Is AI a force for good?
- Are there any positives to being offline?
- At what age should children have smartphones?
- Why is the Investigatory Powers Act 2016 sometimes called the "snooper's charter?"
- How can we prevent algorithms from being racist?
- What is the carbon footprint of a smartphone?
- How can we protect children from harmful online content?
- Why do some economists say data is the new oil?

Higher-order thinking

To encourage higher-order thinking, learners could attempt the following tasks.

Legislation brain dump

List all the ways that school computers are affected by legislation. After explaining the legislation, learners should create a grid of aspects of the computer systems and the relevant laws. With some encouragement the grid might look like this:

Student information system	Contains personal data so is covered by the DPA 2018. If anyone hacks, this would be covered by the CMA 1990.
Student-produced work, e.g. portfolios	Must not include any copyright material under the Copyright Act 1998. Students can use Creative Commons content.
School website	Hacking would be covered by the CMA.
School software	Must be licensed appropriately under the Copyright Act. Alternatively, open-source software could be used under a Creative Commons licence.
E-waste	Must be disposed of in accordance with the WEEE regulations.
Internet traffic	The school's ISP must keep a log of all traffic for six months under the Investigatory Powers Act 2016.

Gaps in legislation

Learners could be asked: what gaps are there in legislation that need to be filled? For example, is it right that employers are able to trawl social media and fire people for comments they made many years ago? Does social media need better policing against hate speech or radicalisation?

Analogy and concrete examples

List the issues

Consider one of the stories discussed in this chapter, such as the Palestinian construction worker who was implicated by poor automatic translation, or the mountain of e-waste illegally exported to developing nations. Discuss the privacy, legal, ethical, environmental and cultural issues raised by the story.

The Moral Machine

Show your learners the Moral Machine, developed by MIT (moralmachine.net). They have to decide, in varying scenarios, whether a self-driving car with brake failure should continue in its lane and hit a pedestrian or animal, or swerve into the other lane to hit a different pedestrian, animal or even a brick wall that would kill the car's occupants. Each decision requires the learner to make a value judgement about who to save. The activity should be used as a starter before a discussion about the ethics of self-driving cars. You could then pose these questions:

- Should a self-driving car protect its occupants at all costs?
- Should the human pilot always be responsible for a self-driving car?
- What legislation would be appropriate to reduce the potential harm of self-driving cars?

- In 20 years' time, will we consider human drivers more dangerous than AI drivers?

An interesting exercise would be to rerun the moral machine activity after the debate and see if the learners' choices change.

Which law has been broken?
Offer a range of scenarios: hacking, selling personal data, software piracy, music sampling, disposal of batteries in the domestic waste bin. Ask the learners to explain which law has been broken in each case. Offer bonus marks for multiple laws or for estimating the sentence correctly.

Live mock question
Live-answering a nine-mark question, by narrating your thoughts in front of the learners, is a very powerful teaching exercise.

Cross-topic, cross-curricular and synoptic

Cross topic with networks
The networks and security topics have a lot of overlap with legal and ethical issues. For example, you could ask learners to act as an IT manager. Design an office computer network for a given business, which could be a fast-food restaurant, a hairdressing salon or a solicitors' office. Consider all the impacts and issues of implementing a computer system in the business.

Unplugged

Debate lessons
Pick a topic such as censorship, the snooper's charter, encryption backdoors, e-waste legislation or bias in algorithms. Pose a motion such as:

- The police should be able to read everyone's direct messages.
- Children should be offline until at least the age of 13.
- A carbon tax on smartphones should be introduced to fund recycling.
- Racist algorithms should be banned.

Hold a class debate. Gentle steering by the teacher should enable the discussion to cover many objectives in this topic, such as privacy, ethical, legal, social, environmental and cultural issues. Ensure that the students consider all the relevant stakeholders both now and in the future, such as consumers, workers, vendors, shareholders and the wider society.

Unplugged AI activities

You can find an activity called "Paper AI" at the Microsoft MakeCode website for Minecraft.[280] Paul Curzon of Queen Mary, University of London, has published his excellent "Brain in a bag" activity on the Teaching London Computing website.[281] Many more AI-related activities are available at aiunplugged.org.

Physical

Machine learning for kids and micro:bit cybersecurity

The Raspberry Pi website has activities on the page "Machine learning for kids"[282] and you'll find micro:bit lessons on cybersecurity[283] and encryption[284] at microbit.org.

Project work

Google has published lots of activities to help explore machine learning through pictures, drawings, language and music.[285]

Misconceptions

Some of the misconceptions on the next page were sourced from the diagnostic question collection at eedi.co.uk.

280 link.httcs.online/paperai
281 link.httcs.online/braininabag
282 link.httcs.online/pimachinelearning
283 link.httcs.online/microbitcyber
284 link.httcs.online/microbitcrypto
285 link.httcs.online/googleai

Misconception	Reality
DPA/GDPR criminalises hacking	The Computer Misuse Act criminalises hacking (unauthorised access)
DPA criminalises piracy	The Copyright, Designs and Patents Act criminalises piracy (unlicensed copying or sharing of intellectual property)
Ethical/legal/cultural/ environmental issue confusion	Ethics are moral principles, or rules, that govern a person's attitudes and behaviour, e.g. public safety. Legal issues pertain to the legality of acts – whether they are against the law or not – e.g. compliance with the DPA. Cultural issues are those that affect the nature and culture of society, e.g. the digital divide or biased algorithms. Environmental issues are those that have an impact on the environment, e.g. teleworking reduces commuting, but e-waste pollutes and causes health issues in developing nations. Some issues overlap: for example, e-waste exports to Africa are illegal and also unethical.
Encryption backdoors "for the good guys only" are possible	Security experts believe encryption backdoors would quickly be found by hackers, rendering the encryption useless.
Cookies steal data or are used for password storage	Cookies cannot access data on hard drives, so cannot read anything; they are plaintext files so would never be used for security data.
Stakeholder/shareholder/ consumer confusion	Stakeholder: any person with a stake, including consumer, vendor, shareholder and society.
Waste phones release CO_2	While waste phones do pose an environmental risk, due to leaching of heavy metals into local water, the carbon impact comes mostly from fossil fuels burned to generate energy for manufacturing replacement phones.
FOI allows individuals to request data held about them	The DPA provides this facility. The Freedom of Information Act gives citizens the right to request information about the activities of public sector bodies.
The digital divide includes software compatibility issues	The digital divide is about disparity of access to technology, not software compatibility.
Young vs old is not digital divide	Young people generally have greater access to technology through familiarity and via the workplace. Young vs old is therefore definitely a digital divide issue.
Creative Commons licensing does not allow onward sharing or derivative works	All CC licences allow resharing. BY-ND and BY-NC-ND do not permit derivatives, i.e. the work cannot be changed. (This knowledge is required by some exam boards only.)

Conclusion

Aims for this book

The first computer science education department was founded at Purdue University, Indiana, in 1962, which gives the academic subject just 60 years of history. Almost every other subject on the curriculum has had centuries in which to develop a mature and powerful pedagogy. A quick search of online bookshops reveals thousands of books just on teaching maths. Organisations and learned societies to support the teaching of core subjects have been around for a long time, such as the London Mathematical Society (founded in 1865) and the Association for Science Education (which dates back to 1900). Yet Computing at School (founded in 2008) and the National Centre for Computing Education (2018) are much younger bodies.

Driving computer science teaching forward requires us to develop a mature and effective pedagogical repertoire. This, in turn, requires discussion, collaboration and inspiration. I have been inspired by many speakers, bloggers and authors on my journey so far, and I wish to add my voice to the mix while amplifying those I have found helpful. Hopefully, this book will inspire other computer science teachers to consider not just what to teach, but how to teach it. Perhaps those teachers will add their own voices to the pedagogy discussion, and we will continue to take our fascinating subject from strength to strength.

Further reading

This book is many things, but it is not a computer science curriculum, a teacher-training primer or a comprehensive summary of educational research. Those books exist already and I'm indebted to the authors of the following works, without whom this book would not have been possible.

Sentance, S., Barendsen, E. & Schulte, C. (eds). (2018) *Computer Science Education: perspectives on teaching and learning in school*, Bloomsbury

A goldmine while I was researching *How To Teach Computer Science*, this book brings together much of the latest research in a digestible format.

Lau, W. (2017) *Teaching Computing in Secondary Schools: a practical handbook*, Routledge

This book contains a wealth of practical advice and teaching ideas. It was invaluable as I transitioned from classroom teacher to head of computing.

Grover, S. (ed.). (2020) *Computer Science in K-12: an A to Z handbook on teaching programming*, Edfinity

Containing tons of practical advice, this book came out just as I was writing the programming chapter – it helped enormously!

Simmons, C. & Hawkins, C. (2015) *Teaching Computing* (second edition), Sage

This superb book was updated in 2015 and remains one of the best primers for trainee computing teachers out there. Essential reading.

Clarke, B. (2017) *Computer Science Teacher: insight into the computing classroom*, BCS

An excellent introduction to what it means to be a computer science teacher, exploring the curriculum, career progression and resources available.

Hey, T. and Pápay, G. (2014) *The Computing Universe: a journey through a revolution*, Cambridge University Press

A great read while I was researching the book, recommended for further developing your computing hinterland knowledge.

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

A very readable history of information encoding that joyfully demystifies the whole topic of data representation.

Clark Scott, J. (2009) *But How Do It Know? The basic principles of computers for everyone*, John C Scott

This book describes what computers are made of, explaining the link between logic gates and computation in an enjoyable read.

Levy, S. (1984) *Hackers: heroes of the computer revolution*, Doubleday

Deservedly a classic, this covers the golden age of "computer hacking" from the Tech Model Railway Club at MIT in the 1950s to the "hardware hackers" Steve Jobs and Steve Wozniak in the 1980s. A must-read for computer enthusiasts that offers great hinterland knowledge.

Acknowledgements

I started writing this book in early 2020, as the world shut down in response to the Covid-19 pandemic. Just over a year and 67,000 words later, it's ready to go to press, but it would not have been possible without these wonderful people:

My wife and my rock, Nicola, and my patient family, who have seen somewhat less of me over the past 12 months.

The strategic leadership team at William Hulme's Grammar School in Manchester, and the board of United Learning, for providing a supportive working environment and a commitment to professional development. This enabled me to grow as a teacher and a leader, and to find the voice (and the time) to write this book.

The computing experts and educators who helped during the writing process, by contributing ideas, answering queries, proofreading or generally being helpful and supportive: Ruth Ashbee, Katie Bouman, Dan Burrows, Penny Cater, Neil Chappell, Katharine Childs, Beverly Clarke, Andy Colley, Dave Cross, Mark Enser, Zoe Enser, Claire Gryspeerdt, Dave Hillyard, Simon Johnson, Paul Knighton, William Lau, Sarah Longshaw, Nicola Looker, Ben Newmark, Alan O'Donohoe, Ellie Overland, Richard Pawson, James Robinson, Marc Robinson, Laura Sach, Craig Sargent, Sue Sentance, Vivita Sequeira, Martin Sexton, Carl Simmons, Christine Swan, Adrienne Tough, Allen Tsui, Katie Vanderpere-Brown, Jane Waite, Alex Weatherall, Claire Wicher and John Woollard.

Finally, thank you to my editor, Isla McMillan, and everyone at John Catt Educational for making my first book an enjoyable and relatively painless experience. I hope this is the first of many, but that depends on you, dear reader. I look forward to your feedback.

Tweet me @MrAHarrisonCS or visit httcs.online and leave a comment. Thanks for the purchase and I hope we talk soon!

Alan Harrison, 1 May 2021, from my office and "remote teaching station" in Sale, Greater Manchester

Image credits

Figure 1.2: Emgravey/CC BY-SA 4.0

Figure 1.3: David Monniaux/CC BY-SA 3.0

Figure 1.6: Mutatis mutandis/CC BY-SA 3.0

Figure 1.7: Russell A. Kirsch/National Institute of Standards and Technology (US)

Figure 2.1: Katie Bouman.

Figure 2.4: Sue Sentance.

Figure 3.1: AP/Shutterstock.

Figure 3.4: Craig'n'Dave.

Figure 4.1: Massachusetts Institute of Technology (US)

Figure 4.2: Recreated from source: craftofcoding.wordpress.com/2013/10/07/what-is-spaghetti-code

Figure 5.2: Michel Bakni, CC BY-SA 4.0

Figure 6.1: Parrot of Doom/CC BY-SA 3.0

Figure 6.2: Malleus Fatuorum/CC BY 3.0

Figure 6.3: William Lau/CC BY-SA 4.0

Figure 6.4: Tiia Monto/CC BY-SA 3.0

Figure 6.5: Daniel Sancho/CC BY 2.0

Figure 6.6: Peter Howkins/CC BY-SA 3.0

Figure 8.1: Tara Tiger Brown/CC BY-NC-SA 2.0

Figure 8.2: Peter Hamer/Magnus Manske/CC BY-SA 2.0

Figure 8.3: Rezonansowy, Microsoft

Figure 8.4: Sashatemov/Irina Blok/CC0 1.0

Figure 9.1: UCLA/BBN/CC BY-SA 4.0

Figure 9.2: ITU Pictures/CC BY 2.0

Figure 9.3: Robert Scoble/CC BY 2.0

Figure 9.4: Recreated from source: steves-internet-guide.com/internet-protocol-suite-explained

Figure 10.3: The National Archives (UK)

Figure 11.1: Prostock-Studio

Figure 11.2: Marlenenapoli/CC0 1.0