Practical programming in computing education

Miles Berry, Paul Curzon, Quintin Cutts, Celia Hoyles, Simon Peyton Jones, and Shahneila Saeed
NCCE Academic Board[1], Feb 2022

Executive summary

This white paper argues for the fundamental importance of practical programming as a central element of our young people's computing education. Our intended audience is senior leaders, heads of computing, and teachers, especially in secondary schools.

In 2014 the new English computing curriculum re-envisioned computing as a foundational subject, rooted in a body of knowledge and ideas, rather than as a vocational one. A danger is that the pendulum swings too far: that we come to see computing as a mainly *theoretical* discipline, divorced from the practical activities that animate it, including programming. The truth is that practical programming is absolutely central to a good education in computing, for three main reasons:

- **Programming is intrinsic to computing**. Programming is both a means to an end *and* an end in itself. Could you imagine a good education in science without science labs, in music without performance, in English without creative writing (only reading)? The whole purpose of computing is to *build things* that change the way we live, work and play, so actually writing programs is key. See Section 1.
- **Programming skills in computing are exceptionally useful**. Programming is extraordinarily useful in other domains, not just the software industry. Moreover, programming projects develop many of the "soft skills" that are so highly prized by employers: teamwork, planning, logical thinking, communication skills, and resilience. See Section 2.
- **Programming work is essential pedagogy**. Designing, writing, predicting, and debugging programs brings together the theoretical knowledge that students encounter in class, consolidates and integrates theoretical understanding, and is incredibly motivating. See Section 3.

This is a particularly opportune moment to emphasise programming as part of the taught curriculum. In 2019 Ofqual removed Non-Examined Assessment from the grade of GCSE Computer Science. Many schools read this as a signal that practical work is not important. But that is completely backwards! The real message is that *programming work is too important to be conducted in 20-hour hermetically-controlled assessments* – it is hard to imagine a less authentic or motivating setting. Rather, freed from the onerous demands of formally-assessed work, schools are now able to design imaginative, creative, and ambitious programming projects in computing, incorporating collaboration and teamwork, connections to "live" data, industrial mentors, and iterative review that strongly supports learning. Section 4 suggests some ideas.

Our call to action is:

> ***Senior leaders*:** actively support your computing department to make rich, creative practical programming a priority in your computing timetable, and in your resourcing, at every Key Stage.
>
> ***Computing subject leaders and teachers*:** ensure that practical programming is deeply integrated in your scheme of work, in diverse forms ranging from program comprehensions, through short exercises, up to capstone projects (see Sections 4 and 5).

[1] The Academic Board of the National Centre for Computing Education provides oversight and guidance for the work of the NCCE, ensuring that quality and rigour, informed by evidence from research, are at the heart of the NCCE's professional development programmes. Alongside the NCCE staff, the Academic Board has five independent members, who are the authors of this white paper.

1 Programming is intrinsic to computer science

The importance of programming is explicit in one of the four Aims of the National Curriculum for computing:

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

But why? Because **programming is not just a means to an end: it is a fundamental part of the subject itself**, just as music performance is fundamental to music, and writing is as fundamental to English as reading. A computing education without programming would be a dry eviscerated husk, shorn not just of motivation and enjoyment, but of intrinsic content.

The whole purpose of computer science is to help us build things. As Fred Brooks put it: "the natural scientist *builds in order to study*; but the engineer *studies in order to build*". He goes on: "*I submit that by any reasonable criterion the discipline we call "computer science" is in fact not a science but a synthetic, an engineering, discipline. We are concerned with making things, be they computers, algorithms, or software systems.*" In reality, computer science is both (and much more too). However, studying in order to build is a fundamental part.

So building things (programming) is not an optional add-on to the subject: it is a key raison d'etre of the subject itself.

The Gatsby report on Good Practical Science makes the case for practical science in a similar way. For example, it says *"Experimentation gives science its identity. Science uses experiments to discover the realities underlying the world, and this practical approach seems to be as intrinsic to young learners as it is to professional researchers."* The Ofsted Research Review on Science makes a similar case: *"At its heart, science involves the study of the material world. Practical work therefore forms a fundamental part of learning science because it connects scientific concepts and procedures to the phenomena and methods being studied."*

Computer science can be considered as 'the silent C' in CSTEM, and it is through programming that the linkages with science, technology, engineering and mathematics are most strongly manifested.

- **Science**. Programming, and particularly debugging, embodies repeated experience of the scientific method: gathering information through observation, formulating a hypothesis, devising experiments to test it, refining the hypothesis, all in service of developing a better mental model of the system. Programming also makes possible new realms of scientific discovery ('computational science'). It also gives a new powerful way to do science that complements experimentation, that has revolutionised scientific method, for example, through simulation and virtual experiments, and through discovery via machine learning.
- **Technology**. Programming is also about acquiring mastery over technology through making the computer carry out your own instructions. In today's world much technology is digital technology, and thus owes its existence to programming. Only by understanding programming can a student deeply understand the possibilities and limitations of existing technology: how it could be better or modified to do new things, for example. Also often programming skills are needed to have mastery over technology: expert use of spreadsheets is one clear example, among many.
- **Engineering**. Programming, like engineering more generally, is concerned with making or building things, and whilst programs are less physically tangible than bridges or circuit boards, their impact on our lives is no less significant. Programmed artefacts are also often now a key part of engineered ones and are often used as a tool to do engineering. Engineering now relies heavily on programming.
- **Mathematics**. There are strong parallels between mathematical reasoning and computational thinking (the key skills underpinning programming). Polya's How to Solve It, whilst originally about mathematical problem solving, applies just as readily to computer programming today. The subject of Mathematics uses abstraction to represent real world things in mathematics, perhaps numerically, algebraically or as more complex structures. Similarly, programmers represent real world things

(including text, images and sound) digitally, as code, or in data structures with their own properties and methods. Mathematical thinking and programming skills are complementary and the practice of each supports the other in deep ways. Programming can also be seen as mathematics in practice: a form of applied mathematics, turning underpinning mathematics into artefacts that do things in the world.

It's not just STEM subjects though! Programming links to, and is a useful skill, in the arts and humanities too (in fact all other subjects). For example:

- **English**. A key and often sadly neglected part of programming is that of documentation. Clear written explanations of what code does, and how, is vital to help limit the mistakes in large software systems and the maintainability of the code.
- **Design and Creativity**. Programming and other forms of practical work are inherently creative. Good design is an important part of good programming. It is good design of the user interface, for example, that makes a program usable (or not).
- **Social sciences**. A key step in developing software is to understand the problem you are solving first. As problems are often problems being solved for people, it is often the contexts of people and society that need to be understood. Social sciences methods such as interviewing people to understand their needs is key to real software development, for example.
- **Art**. The virtual world provides whole new mediums for creating art and whole new techniques for doing so. An artist who can program has a vast new toolset to work with, whether explicitly programming the artwork, developing programs that support doing so, or writing artificial intelligence programs that create the artworks themselves.

2 Programming teaches exceptionally useful skills

In school computing, we should aim to develop in learners the necessary knowledge and skills to be able to *use computing practically* in other areas of their lives, e.g. in other school subjects, in the jobs they will take up, or when studying other subjects at university. The practical programming that we argue is so essential in school lays the foundation for this practical use of the subject in all these other areas. For example:

- **Digital literacy**. Every child needs to be digitally literate, a confident and competent user of computer systems: the fourth aim of the computing national curriculum. Programming underpins that competence, by giving the learner an accurate understanding of what is really going on, rather than just guessing and trying things randomly. It also gives an understanding of what is possible and when computer systems themselves are inadequate.
- **Further study**. Programming lays the foundation for higher study, at A level and beyond. Computation over large data sets is becoming pervasive in many subjects, and programming and modelling skills are increasingly *necessary*, rather than merely *desirable*, and applied in a range of school subjects.
- **The workplace.** Practical computing skills, especially programming, are enormously valuable when students later enter the world of work. Programming skills equip students for work in the tech sector, but are also highly prized across a huge range of other professions. This will only increase in the future.
- **Life.** As learners progress with their lives, they will ever more need to be able to both *do* and *understand* computing in order to be successful citizens. Great computational thinking, of value almost everywhere, will come from great programming experience.

Programming projects develop teamwork, communication, logical thinking, and problem solving skills. These are precisely the "soft skills" that are so highly prized by employers and others. Practical work in computing offers repeated opportunities to develop, exemplify and practise these skills.

Programming informs critical judgement. The experience of how hard it is to get a program right, and how subtle bugs can be, may help students to have more accurate and well-founded critical judgement about the application of computing. Knowing that the systems they are using may have bugs, or may be poorly designed and unnecessarily hard to use, engenders a very different attitude than the blind faith one sometimes sees,

and avoids the common refrain “I can’t use computers” when the problem is that they are unusable. There have been a series of high profile cases in recent years, exemplified by the post office Horizon software scandal, where a computer system was blindly trusted over humans, leading to many innocent sub-postmasters and mistresses being jailed with thousands of lives ruined. Such scandals would be far less likely if more people understood how easily programs can be fallible, and so were more critical in them rather than investing blind faith.

Looking to the future. In the professional world, computing is already changing not only *how we do science*, but *what science we do*; not only *how we do art*, to *what art we do*; and similarly in almost all subject disciplines. These profound changes in subject disciplines will increasingly appear at school level (as they have already in geography and design and technology, for example), and success will depend critically on students' digital competence.

3 Programming work is essential pedagogy

Programming gives an enormous boost to learning, both of computing itself, and other aspects besides. Indeed, it is indispensable in learning the theory of computing, both in terms of better understanding and getting better grades. Why?

- **Programming brings theory to life**. While some will love the subject for its intellectual beauty alone, academic computer science divorced from learning-by-doing can become for many a dry, unmotivated subject. The experience of deploying knowledge to solve problems illustrates and motivates that knowledge, and also consolidates it into a form that is actually retained. Practical building work is thus one very strong (if not the only) way to avoid the trap of turning an exciting subject into a boring, difficult one.
- **Programming is enormously motivating for some**. Learning is hard work, and learning to program can be very hard work, so it is incredibly helpful that programming is so rewarding. Getting their program to work powerfully motivates many students to learn, and gives them a sense of *ownership and agency* through solving a programming task they set themselves.
- **Programming can build confidence**. If they manage to develop strategies to solve new and unfamiliar problems, students can gain confidence, especially when the initial solutions may not work. To gain such benefits universally requires careful scaffolding.
- **Programming environments offer immediate, non-judgemental feedback.** If your program is wrong, it won’t work, and the computer will tell you so, remorselessly, but non-judgmentally (and for certain kinds of mistakes immediately). It will give you the opportunity to form a hypothesis about what is wrong, devise experiments to verify that hypothesis, leading to understanding the cause of the bug and a path to fixing it.
- **Programming consolidates learning**. When doing practical programming work students repeatedly draw on, and give concrete form to, knowledge and ideas learned in class. It involves them actively processing the material (in semantic wave terms doing unpacking and repacking activity themselves, Maton, 2013). Learning by doing in general, and practical work (so programming) in particular, is a vital part of this activity. This unpacking/repacking process consolidates learning, both through working with concrete examples linking theory and practice, repetition and through giving the ideas a practical context. It leads to deep rather than shallow understanding.
- **Programming develops a learner’s initiative and creativity**. Much of school work is closed-form: learn this knowledge. Well-scaffolded practical work can put the student in the driving seat, inviting them to create systems or programs that have never before existed. Their solutions can be eye-opening.

Although practical programming is not formally assessed outside exams as a part of the grade for GCSE Computer Science, students who have the opportunity to engage in well-scaffolded practical work are likely to get much better grades in written exams than students who do not:

- Firstly, practical programming promotes good learning, as discussed earlier, and good learning leads directly to good grades in written exams.
- Secondly, it is an explicit goal of the awarding organisations to design *written exams that are hard to do well in, without extensive experience of programming*. This is much more possible in computing than it is to do the equivalent in (say) natural science. For example, if a question asks a student to predict what a particular code fragment will do, they are much more likely to succeed if they have had plenty of experience of reading and writing code themselves.

We are not arguing that computing should be taught only, or even primarily, through practical programming work. Theory is important as well as practice. Deep conceptual understanding underpins practical skills as well as vice versa. Program comprehension is important as well as program development. A balance is essential.

Nor are we arguing that all practical work in computing should take the form of programming: computing can also be enriched by many other forms of practical work across all areas, such as unplugged activities, physical computing, debates, competitions, digital media, and the like. Rather, we have chosen to address programming specifically in this paper, both for its singular importance, and to maintain a sharp focus.

4 What good practical work looks like

Practical programming should be an engaging, motivating and authentic experience. For example, in addition to core tasks concerned with learning the basics:

- Pupils can engage in **tasks that they find personally relevant**, perhaps basing these on their interests outside of school or on topics from other subjects they are studying.
- They can **collaborate** with one another or with others, including adult mentors both within and outside the school.
- They can do **independent research** using the web, incorporate others' code or ideas into their own programs and get advice or feedback from external mentors.
- They can develop programs or systems to solve **real-world problems** relevant to others in their communities. They may even in doing so form the foundations of a start-up company or career.
- They can **share their work with a wide audience**, both to others in the school, but also to a global audience, so gaining prestige and confidence.
- They can **develop their knowledge** of a particular programming language beyond the narrow scope of an exam specification.
- For those who have already achieved competence in a first programming language, they can go on to develop knowledge of multiple programming languages and paradigms, so **deepening and generalising their understanding** of constructs.

A rich experience of practical programming takes time, and requires support from teachers and school leaders. However, freed from the requirement for controlled conditions, project work can now be done in the pupil's own time, perhaps as a regular homework task or as open ended tasks during school holidays. It can also be done just for fun as a hobby at home or in clubs in ways that contribute directly in terms of skills and knowledge to the ultimate assessments.

4.1 What is "practical programming"?

Programming, or coding, is often thought of as "starting from a blank sheet of paper, write a program to do this task". But actually there is a much richer space of practical work in programming, including

- **Simply experimenting with the medium**. Programming environments like Scratch make it easy to try things out in a playful, exploratory way: "I wonder what happens if I press that button/drag that shape?". At this stage the goal is to experiment, gain confidence that nothing bad will happen, and to gain intuition about what happens.

- **Run a program, and work out what causes it to do what it does**. A good way to start learning is to teach yourself. Have pupils take small programs you provide and run them, see what they do and then try to work out what line causes certain effects to happen. For example, when you run the program, it prints "Hello World". Which specific line actually causes that to happen? Why print that rather than something else? Another program asks for the pupil's name, then greets them by name. Which line made it stop and wait for them to type something in. How did what they typed then end up being printed in the greeting? What causes a monster to appear? What makes it a red monster? This is a good precursor to making changes …
- **Copy an existing program, run it, and then start *making small changes* to it**. The program solves the "blank sheet of paper" problem. Some changes are limited but fun (e.g. change the colour of the monster). As confidence builds, pupils will become more ambitious (e.g. can we have more than one monster?).
- ***Predict* what a program does**. One important aspect of computational thinking is to be able to predict what a program will do. For simple, straight-line programs (i.e. a simple sequence of instructions) this is pretty easy; the more complicated the program, the harder it gets. But at every level the ability to reason logically about the program is key. After making the prediction run it to see if you were right. If you were then it confirms in part your understanding. More importantly, if not, then this tells you something important about things you do not understand, such as identifying misconceptions about the way concepts work. As with science it is when predictions are wrong that most is learnt!
- ***Predict* what a change will do**. A variation of the above is to predict what effect a *change* to the program will have. This needs to be founded on a good understanding of what it does originally. Uncaught bugs often arise from changes where the consequences were not understood by the programmer (for example much of the US telephone network crashed in 1990 with 75 million calls unanswered because of a change made to 6 lines of code out of millions to improve it). Thinking up small changes and predicting what they will do is a powerful way for pupils to *actively* experiment with their understanding of new concepts.
- ***Test* a program thoroughly to check it is working properly.** Take a program and ensure it always works. Programs often appear to work when run but bugs are hidden waiting to be triggered when certain things are input. Creating and following a test plan (e.g. ensuring all branches are tested, checking what happens with extreme values or unexpected values is a skill in itself).
- ***Debug* a program that is not working properly.** For example, if you want to draw a square with a floor turtle, you might forget to put the pen down, so the turtle crawls around but doesn't draw anything. Debugging always involves coming up with a guess (or hypothesis) about what is going wrong, performing experiments to confirm the guess, and making a change that you predict will fix it.
- ***Explain to someone else* how/why your program works**. The simple act of explaining your program often reveals latent bugs in it, potential simplifications to your code, and reveals misconceptions or misunderstandings. It also deepens understanding of concepts. A variation of this (part of 'pair programming') is to write a program in pairs, with one person (the 'driver') at any time doing the typing with the other (the 'navigator') deciding and explaining what to write and why. The navigator must convince the driver that what they are suggesting is both sensible and correct. This gives the driver the chance to point out issues and ask for better explanations where things are unclear. Requiring the navigator to do this helps both gain a deeper understanding and improve their skills.
- ***Read* a program and figure out its purpose**. For example

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

  You could talk about loops and variables, but an experienced programmer would say "oh, that just adds up the numbers between 1 and N, and puts the total in T". That is, she has worked out the *purpose* of the code, rather than just following the individual steps it takes.
- **Starting from an idea of what you want your program to do, *write a program from scratch* to do it**. This involves specification, design, abstraction, implementation, testing, and debugging.
- ***Iteratively develop a large program: modify a large program to do more***. This involves repeatedly understanding the existing program and making changes. It may involve adding new functionality, so you extend what it does while ensuring the old behaviour still works correctly. It could involve modifying that existing behaviour to do some more complex and useful variation (eg so that it does

the same thing faster). This is also the basis of iterative development - a key and critical way to develop large programs, where a series of versions are created that each are ensured to work before moving to the next. In this case it may be the programmers own work that is being developed. It is an important programming skill to develop in its own right.

4.2 Reviewing your provision for practical programming

In reviewing your approach to practical programming, at every key stage, the following questions may be useful. Different things may work for different schools; there are no silver bullets!

Responding to the individual learner

How are you adapting programming work to the needs and interests of individual pupils?

- Do pupils have a choice over the context for practical work?
- Does practical work allow some opportunities for creative expression?
- How are you, or teachers in your department, helping pupils who would otherwise struggle?
- What support is in place to allow students to work at their own pace, repeatedly building solid foundations before moving on to concepts that build on them?
- How are you/ your teachers challenging pupils who would otherwise find typical projects trivial?
- What support is offered for those with special educational needs and disability (SEN/D)?

Working with others

- Does your practical work involve teamwork?
- Do stronger students have the opportunity (and the incentive) to help weaker ones? Is their support constructive to learning (eg teaching to debug) rather than destructive (eg giving code answers or taking over the keyboard)?
- Can pupils explain (to you and to others) what they've done, and how they did it? Are they documenting their code so that others could read and understand it?
- Are you making use of mentors from local employers / universities?
- Is there an audience for pupils' work?

Process

- Is the practical work well scaffolded? Are you offering guidance about what to do, in what order, to help your pupils past the "blank sheet of paper" problem? Are they using standard algorithms, design patterns and development approaches?
- Are students working in a way so they do not overreach themselves, so that they master individual concepts in practice before moving on to incorporating more complex concepts in their work. Are they creating complex artefacts that they have little hope of getting to work, before doing any work to see if they do? If they are, how are you ensuring they step back and take smaller steps?
- Do you have a timely way to identify and fix basic misconceptions that act as a barrier to progress?
- What happens when pupils' programs don't work? Have they been provided with a good understanding of different kinds of errors, and strategies for testing and debugging their code from the outset? Do they show perseverance when faced with challenges? How do you deal with learned helplessness?
- Do you offer in-flight formative feedback to your students about their practical work, to help them over obstacles?
- Are pupils thinking about the design of their program, how to solve problems or develop their projects before they start programming?
- How can pupils tell if they have succeeded with their practical work? What feedback will they receive? Who will provide this? Is such feedback available only at the end or in stages throughout?

- Are there opportunities for pupils to learn new programming ideas independently? How do you avoid their developing misconceptions if they do? How do you ensure they do not use code or techniques found on the Internet that they do not understand?
- How do you support pupils to manage their time in an extended project? Are you using any approaches from software engineering, such as iterative development or sprints?
- Is practical programming helping pupils to acquire fluency in a particular programming language and a particular paradigm?

5 Going deeper

If you want more ideas to dip into, for how to turn a general intention to offer rich practical work into a practical reality, there are many resources to help. Here are some suggestions, many written by or contributed to by the authors of this white paper, and upon which much of the argument here is based.

5.1 Tips and principles available on the web

The questions in Section 4 may feel daunting, depending on your experience, in teaching. However, there is lots of advice in the form of more detailed tips and principles with respect to answering these questions both specifically and in general. Here are some places to start, whatever your level of experience.

- How we teach computing, a set of twelve principles for teaching computing from the UK National Centre for Computing Education (NCCE). Points range from the importance of program comprehension to the need for students to do unpacking and repacking activity. Most are backed up with quick read sheets that go into the points made in much more detail, but in a very practical way.
- Predict, Run, Investigate, Modify and Make (PRIMM). This NCCE quick read is a short overview of a popular approach to structuring lessons in programming, backed by research. The separate steps represent different stages of one or more lessons. It promotes discussion about how programs work, and encourages reading of code before writing, so combines several of the suggestions outlined above.
- Teaching London Computing's "Learning to Learn to Program". These web pages provide practical tips for both teachers and students over what matters most when learning (so teaching) programming. Advice ranges from the importance of believing your students can get better to the importance of setting up a learning environment with quick and frequent feedback, all with specific practical tips.
- CS Teaching Tips. The goal of this project, led by Colleen Lewis was to document and disseminate effective computer science teaching practices. They offer 1300+ "tips", many backed with videos, and categorised by topic, such as Scratch, or classroom management.
- Miles Berry's 14 principles of practical programming, focussed on upper secondary projects. This includes suggestions ranging from how to approach debugging positively to the importance of explanations.
- Ten quick tips for teaching practical programming, Brown & Wilson 2018. Research from educational psychology suggests that teaching and learning are subject-specific activities: learning programming has a different set of challenges and techniques than learning physics or learning to read and write. Computing is a younger discipline than mathematics, physics, or biology, and while there have been correspondingly fewer studies of how best to teach it, there is a growing body of evidence about what works and what doesn't. This paper presents 10 quick tips that should be the foundation of any teaching of programming, whether formal or informal.
- Why we should teach our children to code, Peyton Jones, Hello World Oct 2019 (page 62ff). This article was a direct response to Andreas Schleicher (Director of Education at OECD)'s question "Should schools teach coding?".

5.2 Books

There are also a range of recent books that explore how practically to teach computing including programming that give much more detailed advice, when you are ready, and have the time, to explore how to teach programming, and computing more generally, in more depth.

- Computer Science in K12: An A to Z handbook on teaching programming, edited by Suchi Grover, Edfinity, 2020. This is a practical guide focussing on teaching introductory programming effectively based on research (and written by leading researchers). It covers a wide variety of approaches to teaching through practical work.
- Computer science education: a perspective on teaching and learning in school, edited by Sue Sentance, Bloomsbury Academic, 2021. This has a broader focus than Grover's book, but very much based on research, with chapters written by experts and also with a practical focus.
- 100 ideas for secondary teachers: outstanding computing lessons, Simon Johnson, Bloomsbury Education, 2021. This is again wider than just programming though focusses on very practical classroom approaches - lots of ideas of what an outstanding lesson looks like.
- Creating the Coding Generation in Primary Schools: A Practical Guide for Cross-Curricular Teaching, Steve Humble, Routledge, 2017
- Hacking the Curriculum: Creative Computing and the Power of Play, Ian Livingstone and Shahneila Saeed. John Catt Educational, 2017. Whilst not specifically about programming this argues for the importance of play-based learning with a strong emphasis on creativity. It focuses on more general computing practical activity, but the overarching ideas certainly apply to teaching programming too.
- Code-It: How To Teach Primary Programming Using Scratch, Phil Bagge, University of Buckingham Press, 2015. This focuses specifically on teaching programming in Key Stage 2, with classroom-tested approaches. It gradually introduces important concepts and techniques through a wide variety of different programming projects.
- Mindstorms: Children, Computers and Powerful Ideas, Seymour Papert, New York: Basic Books, 1980. The classic text on teaching programming in elementary and middle school, using Logo. Papert makes a strong argument for learning through making, with programming being a tool to think with, rather than the end goal.
- Learner-centred design of computing education: research on computing for everyone, Mark Guzdial, Morgan and Claypool, 2015. Students have different reasons for wanting to learn to program; Guzdial's book takes this as a starting point to explore what different approaches might be appropriate for the different goals that students have in mind.
- Lifelong Kindergarten: cultivating creativity through projects, passion, peers and play, Mitch Resnick, MIT Press, 2018. Scratch's creator argues for a more creative approach to computing education, in which learners' own projects form the focus.
- How to Raise a Tech Genius, Shahneila Saeed, 2020. This has a broad focus and is primarily aimed at parents learning Computer Science alongside their children. However, the breadth of practical unplugged activities can provide some good ideas for engaging classroom activities; with several chapters being devoted specifically to algorithmic thinking and logical reasoning.

There are many more books and resources available on teaching programming, with more being published all the time, so consider these as places to start.