

Report

Programming and Algorithms within the Computing Curriculum

January 2022



Raspberry Pi
Foundation

Summary

This report explores two major strands of the computing curriculum in England: programming and algorithms. It explores the importance of both strands and their relevance to learners as they represent areas that allow many ideas from across computing to be demonstrated and expressed.

The report aims to summarise the approach taken to programming and algorithms by the National Centre for Computing Education (NCCE). It presents a research-based view of programming that is broader than simply writing code, and encourages learners to consider different abstract levels and ways of thinking about programming.

The report also highlights progression within these strands from the beginning of primary school education all the way through to when learners leave school. The report includes supporting resources and further reading for those looking for more information, courses, or classroom content.

The report consists of six sections:

- Section 1 discusses the role and importance of the programming and algorithms strands of the curriculum. Programming allows all learners to apply concepts from across computing in creative and innovative ways to solve problems relevant to them. Thinking about algorithms enables learners to plan, represent, and communicate their understanding of a problem or task as well as their ideas. There is also mention of the wider skills that are associated with programming (often referred to as computational thinking skills) with particular focus on the skill of abstraction.
- Section 2 examines the current computing curriculum for England and the aims and objectives that relate to programming and algorithms. These aims are discussed and grouped into four themes, which align with the Levels of Abstraction (LOA) model.
- Section 3 details the NCCE's implementation of the programming and algorithms aspects of the computing national curriculum. The structure of the Teach Computing Curriculum (TCC) is explored alongside an overview of the experiences provided across all five key stages.

The design principles for the TCC are discussed with a particular focus on the levels of abstraction framework, which has been taken from research and applied to this curriculum. The final part of Section 3 is all about progression; each unit relating to programming and algorithms is summarised and mapped against the levels of abstraction framework. These key stage by key stage summaries are further collated and presented as a progression of skills and concepts from age 5 to 19.

- Pedagogy is the focus of Section 4, in which pedagogy principles that relate to programming and algorithms are explored and examples of specific practices provided.
- Section 5 provides a summary of relevant professional development sessions and courses provided by NCCE programmes.
- To conclude, Section 6 provides an overview of the key messages of this report.

Contents

1. Introduction	1
2. Programming and algorithms in the national curriculum and beyond	5
3. Programming and algorithms within the Teach Computing Curriculum	7
3.1. NCCE curriculum structure	7
3.2. Programming and algorithm principles	10
3.3. Levels of abstraction	11
3.4. Progression in programming and algorithms	13
Key stage 1	14
Key stage 2	17
Key stage 3	21
GCSE computer science and key stage 4	25
A level computer science	30
3.5. Progression across key stages	33
4. Pedagogical strategies for programming and algorithms	36
4.1. Pedagogy principles	36
Lead with concepts	37
Model everything	39
Make concrete	39
Unplug, unpack, repack	40
Challenge misconceptions	40
Create projects	41
Structure lessons	41
Work together	42
Read and explore code	42
Get hands on	42
Foster program comprehension	43
5. Professional development for computing teachers	44
6. Conclusion	45

1. Introduction

Programming is a practical expression of many concepts and ideas that form the subject of computing. Through programming, learners can create new tools and experiences, solve complex problems, and express ideas. Programming can be, and is, applied across a wide range of contexts to solve a diverse range of problems. This broad application of the skill alongside the increasing pervasiveness of computing in all areas of our lives makes programming an important and relevant skill for all learners.

Programming is also a multi-faceted process that extends beyond just the writing of code. Whilst coding is a big part of the programming process, programming also encompasses analysis and understanding of the task or problem being addressed, designing a solution, as well as testing and debugging.

Algorithms are an important aspect of the programming process; they provide an abstract model of a program, meaning they are independent of any specific tools or programming languages. As such, they can act as a plan prior to the construction of a program, a way to communicate ideas, and something to analyse for efficiency.

Algorithms can be represented in any number of ways but some common tools include flowcharts, pseudocode, structured English, or even simple symbols. Algorithms are precise in nature, leaving no room for ambiguity, and always result in the same outcome (given the same inputs) each time they are followed. How students understand and engage with algorithms changes over time from simple program plans rooted in design and concrete experiences to precise, structured, and formal representations.

Building a suitable programmed solution to meet a specific need or solve a particular problem involves more than just knowledge of a programming language or the ability to select or create an algorithm. The process involves being able to focus on only the important aspects of a scenario, break a problem down into smaller problems, and evaluate the success of an overall solution.

The skills that support effective programming are often referred to as computational thinking (CT). This term is widely used but with varied definitions and perspectives about the breadth of its application. The spectrum of views ranges from CT being intimately connected with programming to it being a generally applicable way of thinking.

Most characterisations of computational thinking are based upon more traditional approaches to programming and do not easily translate to emerging fields such as machine learning and AI. As these areas become more embedded in curricula, computational thinking frameworks may need to be expanded to incorporate relevant skills¹.

¹ Tedre, M., Denning, P., & Toivonen, T., CT 2.0. In: O. Seppälä, A. Petersen (eds.), *21st Koli Calling International Conference on Computing Education Research*, New York: Association of Computer Machinery. 2021. Article number: 3. <https://doi.org/10.1145/3488042.3488053>

This report focuses on programming within the current national curriculum for computing in England and therefore concerns traditional programming and related computational thinking skills.

One skill of particular interest is 'abstraction'. Abstraction is a skill that expert programmers routinely use, adjusting their focus whilst developing a programmed solution. They are able to move between the specific goals of a task, their design for a solution, the building and coding of a program, to how that program behaves when it is run. These different perspectives can be modelled as **levels of abstraction** and provide a common way to describe the routine shift in focus that is part of programming.

To summarise, the study of programming and algorithms involves:

- Solving problems and creating solutions through computer programs
- Understanding, representing, and designing the steps needed to solve a problem
- Being able to routinely shift focus to view a solution at different levels of abstraction

The National Centre for Computing Education (NCCE) was launched in 2018 to work with schools across England and support the teaching of computing. In the first two years, we have engaged with 29,500 teachers, of which 7,500 teachers have participated in professional development. The NCCE includes 34 regional Computing Hubs that take a leadership role in their localities and support schools to deliver a high-quality computing education.

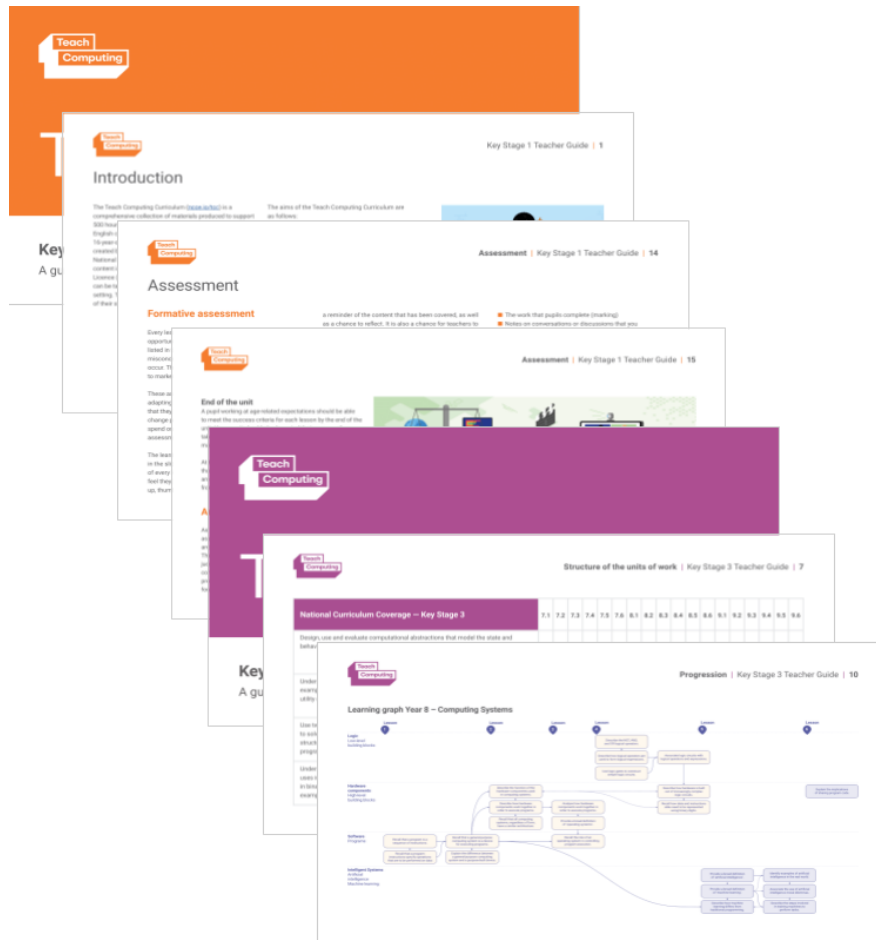


Figure 1: Teach Computing Curriculum teacher guides.

The NCCE's role has been to support the **entire computing curriculum**. A central part of this role has been the development of the Teach Computing Curriculum², which offers teaching resources for each stage of the curriculum. This groundbreaking, and freely available, curriculum supports teachers and learners alike on a journey from key stage 1 to 4; it builds upon the latest research, as well as years of expert teaching experience.

You can read more about our approach to curriculum design in our [teacher guides for all key stages](#). In each guide, we describe how units are structured, the progression within and between units, as well as emphasising appropriate pedagogical approaches.

To complement these curriculum resources is the [Isaac Computer Science](#)³ website, which provides direct support to learners studying A level (and soon GCSE) computer science. The combination of computing content and high-quality questions makes these resources ideally suited to the classroom, self study, and revision.

² National Centre for Computing Education. *Teach Computing Curriculum*. <https://teachcomputing.org/curriculum> [Accessed 17 January 2022]

³ National Centre for Computing Education. *Isaac Computer Science*. <https://isaacomputerscience.org/> [Accessed 17 January 2022]

Together, the Teach Computing Curriculum and Isaac Computer Science cover the teaching of computing and computer science from key stage 1 to 5 (5- to 19-year-olds). Both have been expertly designed with progression in mind and exemplify our approach to sequencing concepts and skills.

This report is part of a series of NCCE reports; each explores teaching and learning within a different aspect of the curriculum. The purpose of this report is to outline the ways in which the NCCE can support you with all aspects of the teaching and learning of programming and algorithms. It has been written in relation to the curriculum in England, although you may also find it interesting if you're reading this from another context. The intended audience is all serving teachers, prospective teachers, and educators involved in teaching computing, as well as those leading on remote education for their school.

2. Programming and algorithms in the national curriculum and beyond

Programming and the understanding of algorithms has become an increasingly important part of the computing national curriculum for England over the last decade. Since 2014, the expectation for schools is that all students between the ages of 5 and 16 should be able to “understand and apply the fundamental principles and concepts of computer science, including abstraction, logic, algorithms, and data representation”.

Whilst many students will go on to study computer science in greater depth through GCSE and A level qualifications, an implicit goal of the computing national curriculum is to prepare all students for an increasingly digital world in which programming, understanding of algorithms, and computational thinking can, and will, be valuable skills in many careers.

Looking across the aims and key stage specifications⁴ of the computing curriculum as well and GCSE⁵ and A level⁶ specifications, it is possible to identify broad themes in which a learner can progress their understanding and application of programming.

The first clear theme is programs themselves; in particular, the reading and writing of them. Students learn to read and write simple programs from their first year of school and over time develop their understanding of key programming concepts. Initially, they focus on sequence, repetition, selection, and variables to ensure a firm grasp of each. Later, students encounter more complex ideas such as modularisation, recursion, and data structures. This experience culminates in GCSE and A level qualifications with students learning about alternative programming paradigms including object orientation and functional programming. Throughout this journey students will use ever-more sophisticated tools and languages.

Alongside programs and programming is the theme of algorithms, and how they differ from programming. Students should understand that an algorithm is a representation of, or a plan for, a program. The two are closely related, but distinct. Whilst a program follows the rules and syntax of the language in which it's written, an algorithm exists independently of any language. The same ideas of sequence, selection, repetition, and variables are also the purview of algorithms except that here they are viewed more as concepts rather than being implemented in a particular language. Students learn what algorithms are (and what they aren't), learn to follow and explain them, and to use them as they design and create

⁴ Department for Education. *National curriculum in England: Computing programmes of study*. Gov.uk. 2013. <https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study>

⁵ Ofqual. *GCSE (9 to 1) subject-level guidance for computer science*. Gov.uk. Report number: Ofqual/19/6513. 2019. <https://www.gov.uk/government/publications/gcse-9-to-1-subject-level-guidance-for-computer-science>

⁶ Ofqual. *GCE subject-level guidance for Computer Science*. Gov.uk. Report number: Ofqual/14/5459. 2014. <https://www.gov.uk/government/publications/gce-subject-level-guidance-for-computer-science>

programs. Later, they will recognise some common algorithms and appreciate how different algorithms can solve the same problem in different ways with different levels of efficiency.

Before students can design their own solutions using algorithms and other tools, they need a clear understanding of the problem they are trying to solve. Problem solving is a clear theme in the curriculum in which students begin by creating programs to solve specific problems and decomposing problems into smaller parts or creating abstractions that represent the important features of an object or scenario.

The final theme related to programming is what happens when a program is run or 'executed'. Students are expected to learn how to debug even their very first programs, spotting and fixing simple errors. Later, they learn to trace a program, running through its execution in their head or on paper. As they begin using new tools and creating increasingly complex programs, the nature and number of potential errors increases and students learn about tools and testing approaches that enable them to be more strategic and systematic in their debugging.

3. Programming and algorithms within the Teach Computing Curriculum

3.1. NCCE curriculum structure

As already mentioned, the NCCE exists to help teachers deliver the entire computing curriculum. The Teach Computing Curriculum, as well as other content from the NCCE, is built upon a 'taxonomy' system used to classify and categorise content. This categorisation consists of ten strands that span the current national curriculum for computing in England. Each strand has a combination of skills and knowledge that feature throughout the national curriculum. Strands that are rich in knowledge form the basis of the units within the Teach Computing Curriculum. Other strands focus on skills across all units. Table 1 gives a summary of the ten strands.

Table 1: A summary of the ten strands in the NCCE content taxonomy.

Strand	Description
Algorithms	Comprehend, design, create, and evaluate algorithms
Creating media	Select and create a range of media including text, images, sounds, and video
Computing systems	What a computer is and how its constituent parts function together
Design and development	The activities involved in planning, creating, and evaluating computing artefacts
Data and information	How data is stored, organised, and used to represent real-world artefacts and scenarios
Effective use of tools	Use hardware and software tools to support computing work
Impact of technology	How individuals, systems, and society as a whole interact with computer systems
Networks	How networks can be used to retrieve and share information and the risks associated with them
Programming	Create software to allow computers to solve problems
Safety and security	Understand risks when using technology and how to protect individuals and systems

From Table 1, we can identify three relevant taxonomy strands: 'Programming', 'Algorithms', and 'Design and development'. These strands are the subject of this report, and they focus on the process of designing, building, testing, and evaluating programmed solutions to a problem or task. These strands contain a combination of skills and concepts for learners to master, and this report illustrates our approach to progression within both areas.

These three strands also intersect with other themes, such as 'Computing systems' and 'Networks', which focus on the environments in which programs run. These strands were the focus of our previous report *Computer Systems and Networking Within the Computing Curriculum*.

For more information on these themes and how they are addressed in the Teach Computing Curriculum, please refer to the [teacher guides for your key stages](#)² and the [NCCE Digital Literacy Within the Computing Curriculum](#)⁷ report.

During their journey through the Teach Computing Curriculum and when subsequently studying with Isaac Computer Science, learners will encounter programming and algorithms in a broad range of contexts. They will experience a range of tools and languages through a variety of projects and across several contexts. This breadth of experiences is intended to help engage and make computing relevant to the widest range of learners possible. It is also the intention that this range of experiences will allow learners to apply and reinforce the same knowledge and skills in multiple settings. Table 2 provides a summary of the breadth of experiences that learners following the Teach Computing Curriculum and Isaac Computer Science materials will receive.

⁷ National Centre for Computing Education. *Digital literacy within the computing curriculum*. <https://blog.teachcomputing.org/digital-literacy-within-the-computing-curriculum/> [Accessed 21 June 2021]

Table 2: The breadth of experiences available to learners using the Teach Computing Curriculum and Isaac Computer Science materials.

	Key stage 1	Key stage 2	Key stage 3	Key stage 4 & GCSE	Key stage 5 (A level)
Contexts	<ul style="list-style-type: none"> - Robots (route finding) - Animations - Interactive quizzes 	<ul style="list-style-type: none"> - Music - Drawing - Games - Quizzes - Physical computing (sensing) 	<ul style="list-style-type: none"> - Music - Story telling - Interactive chat - Games - Quizzes - App development - Physical computing 	<ul style="list-style-type: none"> - Everyday problems - Joke machine - Solving anagrams - Games and puzzles - Challenges - Physical computing 	<ul style="list-style-type: none"> - All revision and study materials presented with a range of contextualised examples
Programming Paradigms	<ul style="list-style-type: none"> - Simple imperative commands and sequences - Some event-based experience 	<ul style="list-style-type: none"> - Event based - Procedural <p>(Predominantly block based)</p>	<ul style="list-style-type: none"> - Event based - Procedural <p>(Split between text and block based)</p>	<ul style="list-style-type: none"> - Structured and procedural - Assembly language - Object oriented (optional) 	<ul style="list-style-type: none"> - Structured and procedural - Functional - Declarative - Event driven - Assembly language - Object oriented
Programming Languages	<ul style="list-style-type: none"> - Simple block based (Scratch Jr) - Bee Bot directional language 	<ul style="list-style-type: none"> - Logo - Scratch - Crumble software - Micro:bit Makecode 	<ul style="list-style-type: none"> - Scratch - Python - App Lab - MicroPython 	<ul style="list-style-type: none"> - Python - Micropython - Little Man Computer 	<ul style="list-style-type: none"> - Formal pseudocode - High-level languages including C# and Python - Haskell - SQL - Assembly language(s)

3.2. Programming and algorithm principles

In approaching the programming and algorithms strands of the Teach Computing Curriculum, four key principles were followed with the goal of supporting all learners to be capable and confident in their programming skills.

Constructs, plans, and patterns

Learners should be introduced to the constructs 'sequence', 'selection', and 'iteration' separately to a program's context. They are then supported in committing common programming 'patterns' to long-term memory, enabling them to recall and apply these patterns to new problems and scenarios. Patterns might be as simple as an assignment statement, a count-controlled loop, or validation loop. Regular retrieval of these patterns through code reading and writing activities will help reinforce them.

Context

Learners will experience programming concepts and common programming patterns across a range of contexts. Each new context is an opportunity to present and reinforce concepts in a way that resonates with the broadest range of learners. The context may vary in terms of the programming language or paradigm used, the nature of the problem being addressed, and, importantly, the cultural relevance to the learners.

Transfer ownership

Learners should first experience using existing programs and designs, which include the concepts they will be taught, in the appropriate context before they begin creating their own programs. This gives learners the opportunity to develop their notional machine relevant to the language and paradigm that is being used. As their understanding of the concepts grows, they will begin modifying existing programs and designs, and finally, making their own. This process develops ownership from code that is 'not theirs', to 'partially theirs' (as they begin to edit), to 'entirely theirs' (as they build their own code), scaffolding the accountability they feel when something does not work as they expect.

Perspective

As already discussed, programming involves moving between levels and being able to focus on the task, design, code, and its execution. Learners should develop the ability to switch perspectives as needed and understand the focus that each perspective affords. Key affordances are concepts such as choosing the correct way to represent an algorithm in the 'design' level, or an appropriate debugging strategy in the 'running the code' level. Being explicit about these different perspectives should help learners to identify, and switch between these levels.

3.3. Levels of abstraction

Whilst there are several ways to characterise the process of programming, an approach taken within the development of the Teach Computing Curriculum builds on the Levels of Abstraction (LOA) hierarchy. Whilst not conclusive, there is some evidence to suggest that this approach can support novices in learning to program^{8,9}.

This hierarchy emphasises the critical role that abstraction plays in developing programs and describes four levels, encompassing different degrees of abstraction. We adopt these four levels, adapting the language to make it more inclusive of younger learners. Throughout, we refer to these levels as 'Task', 'Design', 'Code', and 'Running (the project)'. The only additional adaptation comes during physical computing projects, where the term code is replaced with 'build' to reflect the physical elements of these projects. The four levels can be characterised as follows:

Task

The task outlines the problem to be solved or describes what the project should actually do. With younger learners, a task is often defined by a teacher. As students become more experienced, they can expand a given task or develop the task themselves. Later, they may work from formal specifications or user requirements and even define the task independently through user research.

Design

The design level includes the algorithm, which outlines the process and logic that will exist within the program. The design may also contain other aspects such as artwork, sounds, and sketches of what the project will look like or how it will be put together. This level contains more detail than the overall task, but doesn't yet refer to the code or programming languages that will be used. Learners can use a range of tools to represent their design including text, sketches, flowcharts, and diagrams.

Code

The code level represents a static program that implements the design from the level above. This could be constructed in any number of programming languages, including block-based and text programs. Learners will be limited to the languages and tools they are familiar with initially; as their confidence and repertoire increases, they can be more discerning about the best tool to implement their design.

⁸ Perrenet, J., & Kaasenbrood, E. Levels of abstraction in students' understanding of the concept of algorithm: the qualitative perspective. In: *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*. New York, Association for Computing Machinery. 2006, 270–274. <https://doi.org/10.1145/1140124.1140196>

⁹ Statter, D., & Armoni, M. Teaching abstraction in computer science to 7th grade students. *ACM Transactions on Computing Education*, 2020, 20(1), 8. <https://doi.org/10.1145/3372143>

Run the project

At the lowest level, the programmer is concerned with how the program behaves when run. Does it run? Are there errors? Does the program behave as expected? These are all important questions at this level. Here, learners need to know how to test their programs, find and correct errors, as well as trace the execution of their code to ensure they understand its behaviour

These levels of abstraction do not represent a linear pathway to developing a project. Whilst learners will often begin at the task level and generally progress towards running the code, they will frequently need to switch back and forth between levels.

3.4. Progression in programming and algorithms

To explore progression within these crucial strands, we consulted the objectives from the Teach Computing Curriculum for key stages 1 to 4 (ages 5 to 16), including the GCSE content. To examine progression further, we considered the curriculum objectives covered by the Isaac Computer Science website, which represents all A level exam boards in England. Each objective is already categorised to the NCCE taxonomy, which makes it easier to show those objectives that sit within the programming and algorithms strands.

Each of these objectives were then collaboratively categorised by the levels of abstraction that best described them. This allows us to see a broad representation of the progression from each key stage to the next; in particular, we see how initially the focus is on developing design and coding skills, but how running the code as part of the testing and debugging process becomes much more prominent in later year groups. The following sections present a breakdown of this progression by key stage, along with the concepts explored by learners at each point in their journey.

Whilst each unit in the Teach Computing Curriculum will see learners move between all four levels of abstraction, different units may focus on some levels more than others. For example, most primary units have students working towards a common (and tightly defined) task, which the learners build a design around. In later years, learners have more scope to direct, define, and research their own task. Similarly, whilst almost all units involve pupils running and testing their code, it may not be the particular focus of that unit. The tables included below for each unit show the proportion of objects within that unit that explicitly relate to each level.

Each stage of the curriculum is different; some are longer stages than others, some are statutory while others are elective, and those resulting in a qualification generally involve many more teaching hours. As might be expected, each key stage includes a recap of concepts that have been encountered at an earlier stage. This is particularly noticeable at points of transition, such as when learners move from primary to secondary education or when learners choose to study a GCSE or A level qualification in computing. Due to varying provision in different schools, prior knowledge cannot be assumed. This creates a degree of overlap between key stages. These differences and intersections make direct comparisons between key stages challenging. Instead, this report attempts to describe the focus and progression within each educational stage.

Key stage 1

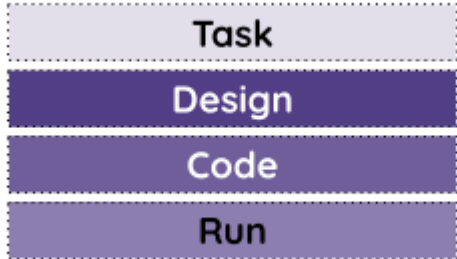
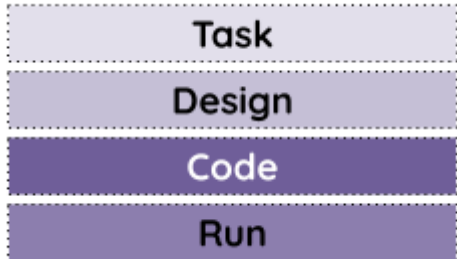
At this key stage, pupils first explore single commands, learning that each produces a fixed repeatable outcome. Once this becomes concrete for learners, they begin to create simple sequences of commands, creating programs that meet given tasks. To do this, pupils use educational floor robots and ScratchJr. These programming environments both use symbols to represent commands, which allows pupils with lower literacy levels to engage in learning to program.

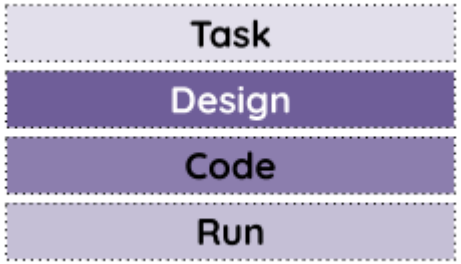
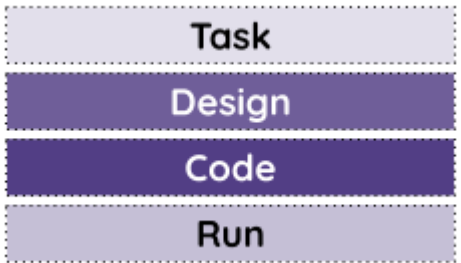
Throughout the programming units, pupils are introduced to program design and create simple algorithms, which they represent by using symbol cards and sketches. They use design templates and create artwork for use with their programs. This design work, and the algorithms they create as part of this, then guide the pupils' program creation.

As programs are created, learners run and debug them. Learners use logical reasoning and the technique of stepping through their programs to identify where errors occur. Once bugs are identified, pupils attempt to address them and re-test their code.

Towards the end of this key stage, pupils start to use events in ScratchJr to trigger program actions and assign values to commands, reducing the number of commands needed in their projects.

Table 3: Key skills and concepts at key stage 1.

Curriculum units	Proportion of content at each level	Key concepts and skills
<p>Year 1: Programming A – Moving a robot</p> <p><i>Writing short algorithms and programs for floor robots, and predicting program outcomes.</i></p>		<ul style="list-style-type: none"> • Sequence – move from single commands to short sequences • Design – create short algorithms to complete a task • Representation – use symbol cards to create algorithms • Testing – match algorithm to program and expected outcomes to identify and rectify issues
<p>Year 1: Programming B – Introduction to animation</p> <p><i>Designing and programming the movement of a character on screen to tell stories.</i></p>		<ul style="list-style-type: none"> • Sequence – identify start and end; create multiple program sequences • Design – choose artwork and create short algorithms • Representation – use a design template to sketch their algorithm • Testing – test output against expectations

<p>Year 2: Programming A – Robot algorithms</p> <p><i>Creating and debugging programs, and using logical reasoning to make predictions.</i></p>		<ul style="list-style-type: none"> • Sequence – see different sequences of the same commands may create different outputs; predict the outcome of sequences • Design – choose and create artwork • Representation – create algorithms using symbol cards and symbol drawings • Decomposition – break a task down into two smaller tasks • Testing – match output to algorithm to identify issues
<p>Year 2: Programming B – An introduction to quizzes</p> <p><i>Designing algorithms and programs that use events to trigger sequences of code to make an interactive quiz.</i></p>		<ul style="list-style-type: none"> • Sequence – predict the outcome of sequences of commands; use user input to start sequences • Design – change a design; use a design template to plan a solution; choose artwork • Representation – create written algorithms • Testing – match code to algorithm to identify issues

Key stage 2

At this key stage, pupils are first introduced to the core constructs of programming: sequence, repetition, selection, and variables. They continue to develop their understanding of design and algorithms by producing algorithms in different formats, including annotated sketches, flowcharts, and ordered lists. Learners also develop their program design skills by choosing and creating artwork, planning and building models, and using a variety of design templates.

These core constructs and design skills are then used in a range of environments such as Scratch, Logo, Crumble, and micro:bit makecode. This allows pupils to apply the learnt concepts in a variety of contexts. The use of Crumble and micro:bit allows students to engage with and develop their understanding of physical computing.

In year 3, pupils create sequences in Scratch to make music. They then move on creating a simple maze game, which uses events to trigger sequences of actions. In year 4, pupils are introduced to iteration in the form of infinite, count-controlled, and condition-controlled loops. Students use count-controlled loops to draw shapes and patterns in Logo and produce a game in Scratch.

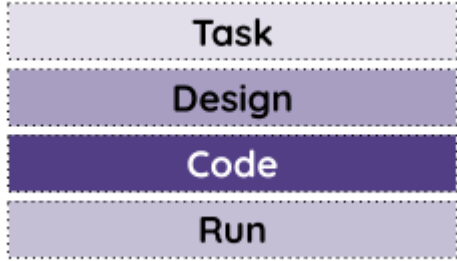


As each year group progresses, a new construct is added whilst consolidating the previous concepts. In year 5, pupils learn about conditional statements, which they use with selection in their programs. After this, they create models and interactive quizzes, which they control by using repetition and selection in their programs. Finally, in year 6, pupils are taught about variables and use them in their programs. They create games in Scratch that use variables for different purposes and they use makecode to read and respond to variable data from micro:bit sensors.

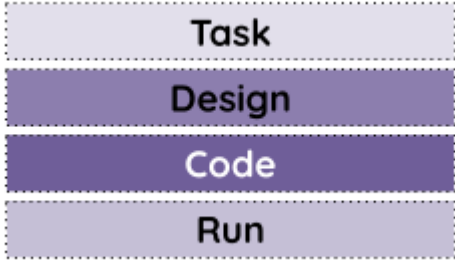
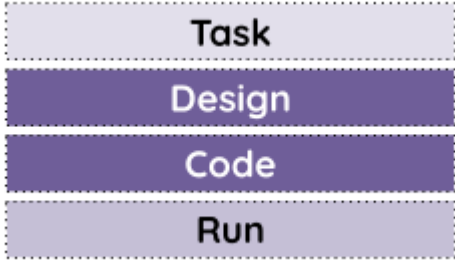
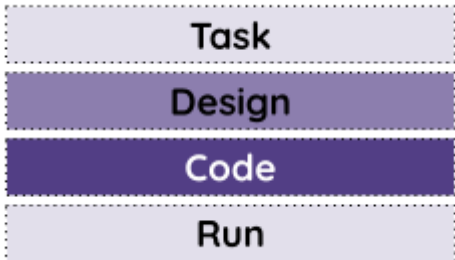
Throughout the key stage, learners develop their debugging skills. Pupils test their code to see if the output meets their expectations. They check their project's outcome against their design, including physical models they have built. Learners also use code tracing, reading code, and pattern matching to identify issues and fix them.

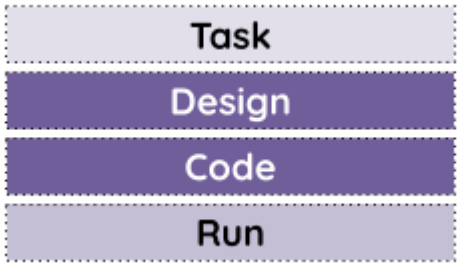
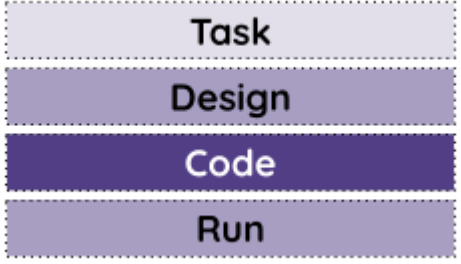
- Checking outcome against **expectations**
- **Testing** against **design** through code **tracing, reading code, and pattern matching**
- Identifying issues where output does meet expectations
- Identifying issues with a **constructed model and the code that controls it**
- Testing and fixing issues
- Testing code against **given criteria and fixing issues**

Towards the end of this key stage, pupils are able to recognise when each of the constructs is needed for their task and how to implement it in their programs. They can build programs that need different algorithms for different parts, and are aware of the need to test and debug regularly throughout the process.

Table 4: Key skills and concepts at key stage 2.

Curriculum units	Proportion of content at each level	Key concepts and skills
Year 3: Programming A – Sequence in music <i>Creating sequences in a block-based programming language to make music.</i>		<ul style="list-style-type: none"> Sequence – create simple sequences that start by different methods; recognise the importance of command order Design – create a design using a given template Representation – produce a written algorithm Testing – check outcome against expectations
Year 3: Programming B – Events and actions <i>Writing algorithms and programs that use a range of events to trigger sequences of actions.</i>		<ul style="list-style-type: none"> Sequence – create sequences that execute after the user's key press; use commands to set up a project so it always starts the same way Design – create a project design using a template Representation – produce written algorithms Testing – test against design
Year 4: Programming A – Repetition in shapes <i>Using a text-based programming language to explore count-controlled loops when drawing shapes.</i>		<ul style="list-style-type: none"> Sequence – understand that longer sequences of repeated instructions can be replaced with a loop, e.g. to draw a square Iteration – use count-controlled loops to draw shapes Subroutines – create subroutines within loops to draw patterns Design – plan algorithms to draw letters and shapes Representation – create algorithms as annotated sketches Decomposition – cut task into smaller challenges; create subroutines Testing – code tracing; reading code; pattern matching

<p>Year 4: Programming B – Repetition in games</p> <p><i>Using a block-based programming language to explore count-controlled and infinite loops.</i></p>		<ul style="list-style-type: none"> • Iteration – use infinite and count-controlled loops • Design – use a design template; select artwork; plan actions • Testing – identify issues where output does meet expectations
<p>Year 5: Programming A – Selection in physical computing</p> <p><i>Exploring conditions and selection using a programmable microcontroller.</i></p>		<ul style="list-style-type: none"> • Iteration – use count- and condition-controlled loops • Conditions – use input from a button as a condition • Selection – use 'if...then...' to control a physical model • I/O – work with components connected to a microcontroller as inputs and outputs (I/O) • Design – sketch a model's construction and wiring; create written algorithms • Testing – identify issues with a constructed model and the code that controls it
<p>Year 5: Programming B – Selection in quizzes</p> <p><i>Exploring selection in programming to design and code an interactive quiz.</i></p>		<ul style="list-style-type: none"> • Iteration – use infinite loops to repeatedly check conditions • Conditions – create conditions, including system variables • Selection – use 'if...then...else...' • Design – choose conditions and outputs for selection • Representation – create algorithms as flowcharts • Testing – test and fix issues

<p>Year 6: Programming A - Variables in games</p> <p><i>Exploring variables when designing and coding a game.</i></p>		<ul style="list-style-type: none"> • Sequence – see the effect of changing variables at different places in programs • Variables – change variables in different ways and by different amounts • Operators – use arithmetic operators to change variables by different amounts • Design – choose artwork; create a design sketch • Representation – make annotated sketches and written algorithms • Testing – test code against given criteria and fix issues
<p>Year 6: Programming B - Sensing</p> <p><i>Designing and coding a project that captures inputs from a physical device.</i></p>		<ul style="list-style-type: none"> • Conditions – use conditions to change variables • Selection – use 'if...else...', 'if...then...' • Variables – set variables with sensor values • I/O – use sensors and buttons as inputs; create different outputs • Design – create a written design and program flowchart • Testing – use a range of approaches to find and fix bugs

Key stage 3

For most pupils, moving into this key stage represents a moment of transition as they move to secondary school. To accommodate the wide variety of programming experiences that pupils from different schools bring, an initial focus of key stage 3 is to consolidate and reinforce the understanding of core constructs of programming. These core components – sequence, repetition, selection, and variables – are explored through an extended project in the familiar block-based Scratch environment.

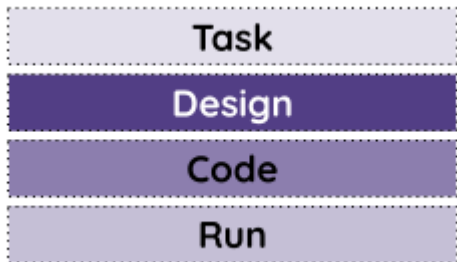
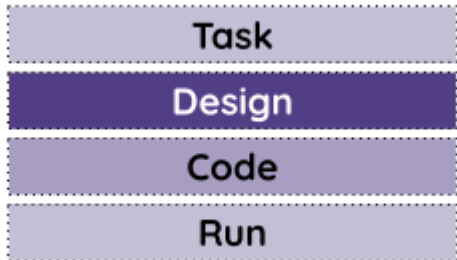
Whilst pupils are likely to have had experience of subroutines at earlier stages, at key stage 3 pupils will develop their use of subroutines as well as extend their understanding of simple variables to incorporate the list data structure.



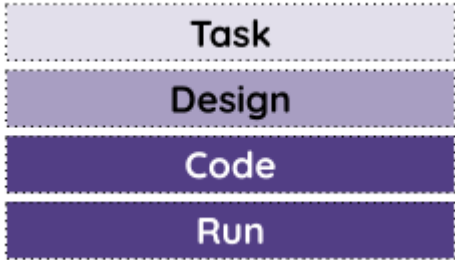
As pupils progress through the key stage, they will apply and combine these core constructs and develop new programming patterns. These patterns can then also be implemented in other contexts and programming languages. In particular, pupils in year 8 will apply their programming knowledge to the development of a mobile app using an event-based programming language.

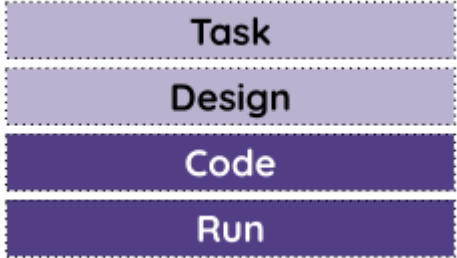
Another significant step for pupils is the shift in year 8 from familiar block-based languages to writing programs using text. Whilst the fundamental constructs they will use remain the same, a text-based language introduces additional opportunities as well as challenges. Whilst languages like Python provide increased control and flexibility, they also introduce the possibility of syntax errors and depend on pupils' language and typing skills. Additionally, features that they may have taken for granted in block-based environments such as Scratch (e.g. graphics, concurrency) are neither available by default nor easily achievable.

Pupils continue to apply their algorithm design and representation skills but with increased independence and agency. Projects undertaken throughout this key stage also lead them to make more independent choices about when and how to decompose a problem as well as taking more responsibility for testing and evaluating their programs.

Table 5: Key skills and concepts at key stage 3.

Curriculum units	Proportion of content at each level	Key concepts and skills
<p>Year 7: Programming essentials in Scratch – part I</p> <p><i>Applying the programming constructs of sequence, selection, and iteration in Scratch.</i></p>		<ul style="list-style-type: none"> • Sequence – follow, modify, and create sequences of instructions • Variables – manipulate variables and trace their value • Conditions – use conditions using logic and comparison operators • Selection – use ‘if...then...else...’ to control program flow • Iteration – identify where to use count-controlled loops • I/O – respond to user input and provide meaningful output • Testing – find and fix errors where they occur
<p>Year 7: Programming essentials in Scratch – part II</p> <p><i>Using subroutines to decompose a problem that incorporates lists in Scratch.</i></p>		<ul style="list-style-type: none"> • Decomposition – identify smaller sub-tasks of a larger task • Subroutines – create sequences of instructions for each sub-task • Iteration – identify where to use condition-controlled loops • Testing – find and fix errors where they occur • Data structures – add, remove, and retrieve items from a list • Design – combine programming techniques to solve a problem

<p>Year 8: Introduction to Python programming</p> <p><i>Applying the programming constructs of sequence, selection, and iteration in Python.</i></p>		<ul style="list-style-type: none"> • Sequence – follow, modify, and create text-based sequences • Variables – assign and manipulate variables using text • Selection – structure ‘if...then...else...’ blocks with syntax and/or indentation • Iteration – create for and while loops using text • I/O – respond to user input and provide meaningful output using text • Testing – define, identify, and fix syntax errors in programs
<p>Year 8: Mobile app development</p> <p><i>Using event-driven programming to create an online gaming app.</i></p>		<ul style="list-style-type: none"> • Decomposition – identify when a task needs to be broken down • Conditions – use conditions as a way to trigger events • Selection – interpret events as a form of selection (abstraction) • Iteration – summarise the role of the event loop • Variables – pass values of variables between parts of their app • Decomposition – break a task down into events and related actions • Testing – debugging within an event-based environment
<p>Year 9: Python programming with sequences of data</p> <p><i>Manipulating strings and lists. Creating a programming project.</i></p>		<ul style="list-style-type: none"> • Data structures – add, remove, and retrieve items from lists • Variables – use alongside lists to store counts, sums, averages, etc. • Iteration – use count- and condition-controlled loops to iterate over a list • Design – apply lists and related techniques to solve a problem • Testing – identify and fix errors related to indexing lists

<p>Year 9: Physical computing</p> <p><i>Sensing and controlling with the micro:bit.</i></p>		<ul style="list-style-type: none"> • Conditions – use the state of input devices within a condition • I/O – use physical inputs/outputs to sense and interact with the physical world • Selection – use physical inputs to control program flow • Design – combine text-based programming and physical devices to solve a problem
---	---	---

GCSE computer science and key stage 4

Whilst all students at key stage 4 should have the opportunity to continue to develop their programming and algorithm skills, many will choose to delve deeper by studying GCSE computer science. The Teach Computing Curriculum provides for both groups of students by including a comprehensive scheme suitable for all exam boards and two additional units that provide learning outside the GCSE specifications. Each of these units can be used with students not working towards a GCSE qualification.

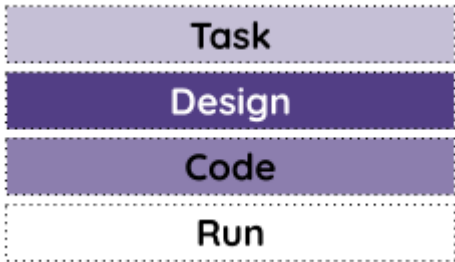
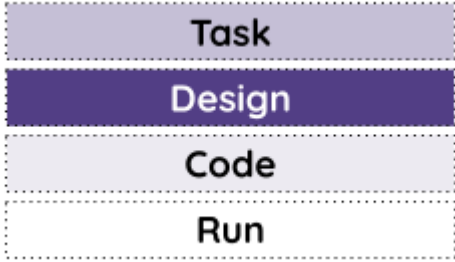
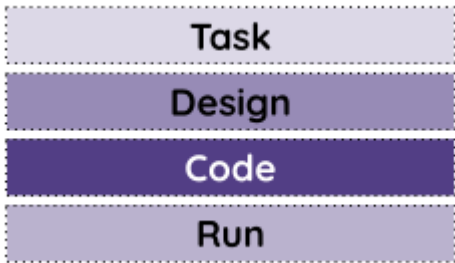
The fundamental concepts that students explore during the programming aspects of their GCSE do not change. However, students explore these concepts in greater depth and combine them in more structured and elaborate ways. A significant focus at this stage is on students decomposing larger problems into ever smaller elements, building subroutines, and passing data between them using arguments and return values.

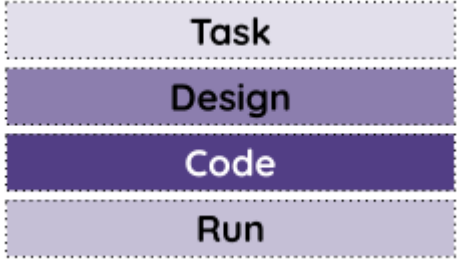
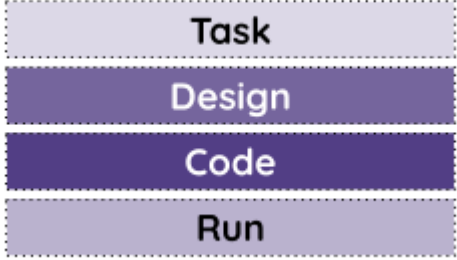
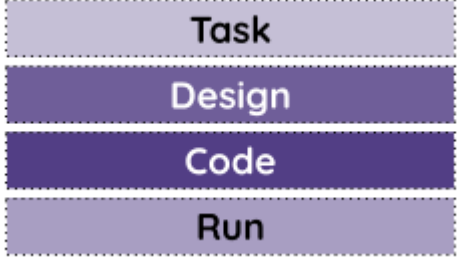
They begin to explore more complex combinations of standard control structures, particularly selection and iteration, nesting them to elicit more elaborate behaviours. Students will also apply control structures to enhance the robustness of their programs through techniques such as validation.

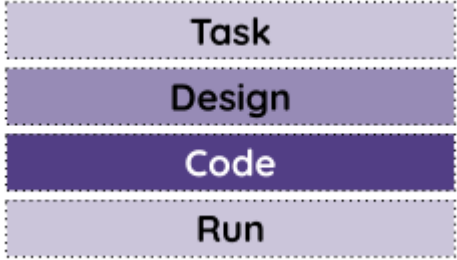
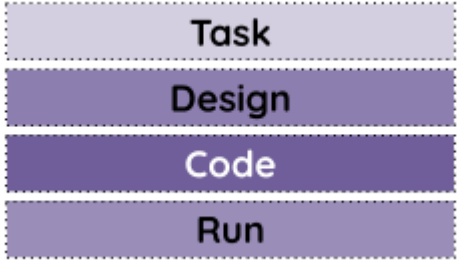
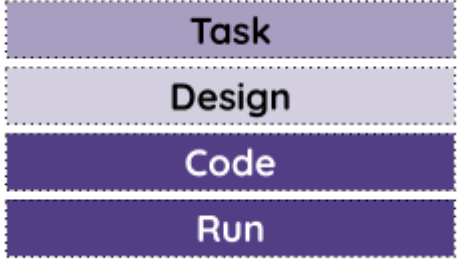
Their understanding of variables and data structures will expand to consider the scope of variables within their programs and the benefits of local and global variables. Building on their learning in the previous key stage, they will explore multi-dimensional lists as well as records and dictionaries in order to store and process more complex data.

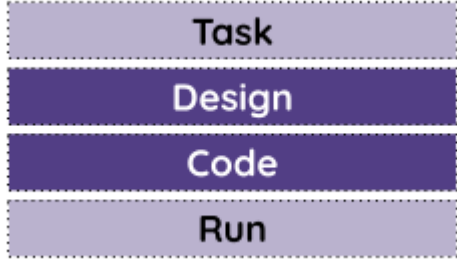
Students' wider development skills will also expand and become more formalised, from reading and tracing algorithms expressed in pseudocode to applying specific testing strategies. They will also study some common algorithms (searching and sorting) in order to reflect on the affordances and relative efficiency of different solutions.

Table 6: Key skills and concepts at key stage 4.

Curriculum units	Proportion of content at each level	Key concepts and skills
<p>GCSE: Algorithms part 1</p> <p><i>Define the terms 'decomposition', 'abstraction', and 'algorithmic thinking'. Use trace tables.</i></p>		<ul style="list-style-type: none"> • Representations – use flowcharts to represent algorithms • Testing – use trace tables to walk-through an algorithm and locate errors
<p>GCSE: Algorithms part 2</p> <p><i>Describe a linear and binary search. Explain the key algorithms for a bubble, merge, and insertion sort.</i></p>		<ul style="list-style-type: none"> • Sequence – describe the steps involved in common searching and sorting algorithms • Data structures – traverse and manipulate a list, inserting and removing elements • Design – implement common searching and sorting algorithms in code • Evaluation – factors that affect the efficiency of searching and sorting algorithms
<p>GCSE: Programming part 1 – Sequence</p> <p><i>Determine the need for translators. Use sequences, variables, and inputs. Design programs with flowcharts.</i></p>		<ul style="list-style-type: none"> • Sequence – describe how sequences of instructions are translated from high-level to low-level languages • Variables – demonstrate how to declare, initialise, assign, and cast variables of different data types • Testing – characterise different types of errors and strategies to avoid them • Design – use a flowchart to design a program before implementing it

<p>GCSE: Programming part 2 – Selection</p> <p><i>Use randomisation in programs. Work with arithmetic and logical expressions. Use selection and nested selection in Python.</i></p>		<ul style="list-style-type: none"> • Decomposition/sequence – integrate external modules or third party code • Operators – incorporate additional arithmetic operators (floor division, modulo, powers) into expressions • Selection – interpreting code that contains nested selection • Design – represent selection using standard flowchart symbols
<p>GCSE: Programming part 3 – Iteration</p> <p><i>Use a while loop and a for loop in Python. Perform validation checks on data entry. Design programs using pseudocode.</i></p>		<ul style="list-style-type: none"> • Iteration – application of condition-controlled loops in validation routines • Design – use pseudocode to design a program before implementing it
<p>GCSE: Programming part 4 – Subroutines</p> <p><i>Use functions and procedures as part of the structured approach to programming. Describe scope of variables. Test a program for robustness.</i></p>		<ul style="list-style-type: none"> • Subroutine – use parameters and return within subroutines • Variables – compare the use and scope of local and global variables • Operators – describe the function and logic of the XOR operator before implementing in code • Design – summarise and apply the structured approach to programming, breaking a large problem in many smaller subroutines • Testing – describe the role of iterative testing and types of testing involved (erroneous, boundary, normal)

<p>GCSE: Programming part 5 – Strings and lists</p> <p><i>Perform string handling operations. Describe the differences between a list and an array. Manipulate a list. Work with 2D lists.</i></p>		<ul style="list-style-type: none"> • Data structures – describe and make use of lists (including 2D lists) and their associated methods within programs • Operators – apply common string operations within code, including concatenation, substringing, and slicing • Subroutine – use a data structure such as a list as a return value from a function
<p>GCSE: Programming part 6 – Dictionaries and datafiles</p> <p><i>Use a record and a dictionary data structure. Access and modify external data files. Complete a complex programming project.</i></p>		<ul style="list-style-type: none"> • Data structures – describe a record data structure and implement it using dictionaries and lists • I/O – build code that reads, writes, and appends structured data to and from external files • Design – analyse a task and design a solution using flowcharts and/or pseudocode • Testing – carry out testing of a program built for a specific task • Evaluation – judge the success of a program against a specification and user needs
<p>Optional: Physical computing project</p>		<ul style="list-style-type: none"> • I/O – build programs that use a text-based language to sense the environment and move something in the physical world • Design – apply physical computing principles to solve a problem • Variables – process and use data provided by a physical sensor to make decisions

<p>Optional: Object-oriented programming</p> <p><i>Apply the principles of object-oriented programming. Create a class and use its attributes and methods.</i></p>	 <p>The diagram shows a vertical sequence of four rectangular boxes. The top and bottom boxes are light purple and labeled 'Task' and 'Run' respectively. The middle two boxes are dark purple and labeled 'Design' and 'Code' respectively. Each box has a dashed border.</p>	<ul style="list-style-type: none"> • Data structures – explain the relationship between objects and how they can be seen as custom data structures • Decomposition – compare subroutines and variables with methods and parameters, respectively • Variables – relate variables and their scope to object parameters
--	---	---

A level computer science

The theory and practice of programming are a major part of A level qualifications in computer science. Students will both expand and deepen their knowledge of programming, algorithms, and software development practices.

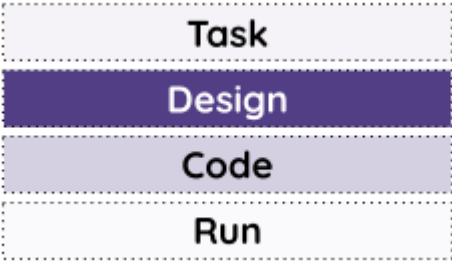
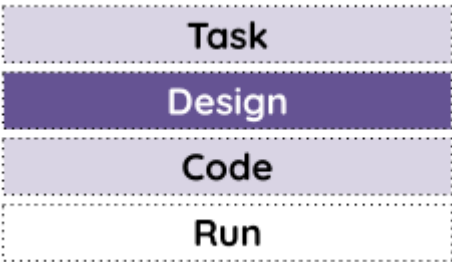
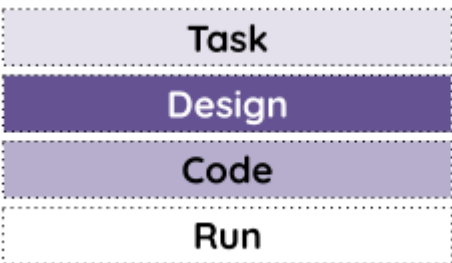
At this stage, they will learn to represent algorithms, programs, machine states, and logic in new ways, including finite state machines, regular expressions, pseudocode, and structured English. They will also look beyond common searching and sorting algorithms to explore approaches to path finding and graph traversal. Alongside their understanding of how different algorithms work, students will also explore complexity within algorithms and how we measure and compare it.

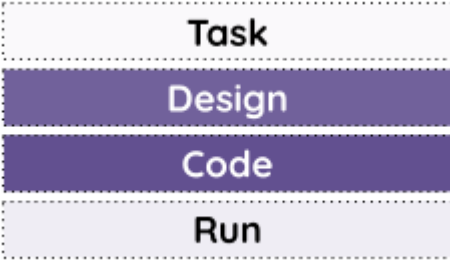
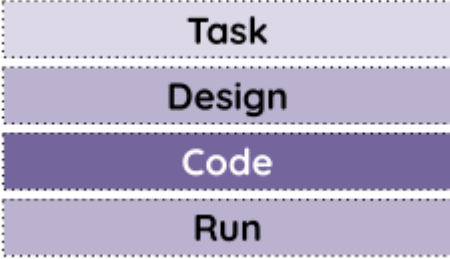
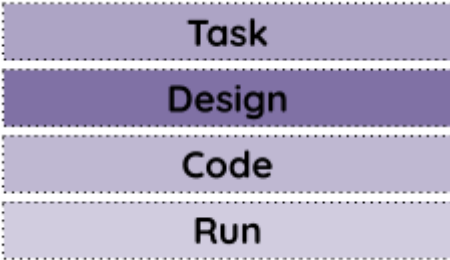
For most students, this stage will be the first time they encounter the concept and practical application of recursion within programs. A level students will also explore additional programming paradigms such as functional programming and take an in-depth look at object-oriented programming. Alongside these high-level programming experiences, students will also spend more time tracing and writing simple programs in assembly language.

Students will expand their understanding of data structures, going beyond variables, lists, and dictionaries to explore more abstract structures and how they can be implemented in code. Graphs, hash tables, linked lists, and trees are just some of the structures that students will study in terms of their structure, use, and application.

The final significant area is that of software development, in which students explore all aspects of a software development lifecycle. They will construct detailed plans and success criteria before developing software, which is thoroughly tested. Students will adopt a holistic approach to evaluate a solution, considering its functionality, usability, robustness, and more.

Table 7: Key skills and concepts within A level computer science.

Isaac Computer Science strand	Proportion of content at each level	Key concepts and skills
Data structures and algorithms		<ul style="list-style-type: none"> Design – explain the importance of considering and measuring complexity of algorithms and the application of heuristics Data structures – apply a range of data structures, including graphs, hash tables, linked lists, queues, stacks, trees, and vectors to solve common problems including search, sorting, and pathfinding algorithms
Theory of computation		<ul style="list-style-type: none"> Decomposition – apply decomposition techniques, including problem reduction, and divide and conquer methods; describe a range of abstraction techniques that can be applied to a problem or scenario Representation – make use of finite state machines, state transition diagrams, regular expressions, and Turing machines to represent aspects of computation
Programming paradigms		<ul style="list-style-type: none"> Sequence – describe and compare sequential and non-sequential programming approaches, including procedural, event-driven, functional, and object-oriented programming Decomposition – apply each programming paradigm to scenarios, breaking down the task using events, subroutines, functions, or objects

Programming fundamentals		<ul style="list-style-type: none"> • Iteration – apply nested iteration to solve problems • Subroutines – apply recursive techniques and describe how stack frames are created during subroutine execution • Testing – explore a wide range of testing techniques and understand common errors introduced by new paradigms and techniques • I/O – make use of binary files within programs
Computer systems		<ul style="list-style-type: none"> • Sequence – distinguish between high- and low-level languages and understand how programs are translated between them • I/O – compare machine code, assembly language, and different instruction sets as well as build programs in low-level languages
Software engineering		<ul style="list-style-type: none"> • Design – compare different approaches to software development including the stages involved, methodologies, and tools; research a scenario to capture all of its requirements • Decomposition – identify the reusable program components during design • Representation – use a combination of structure charts, class diagrams, flowcharts, and pseudocode to represent aspects of the designed solution • Testing – generate test plans and apply debugging to build a robust solution

3.5. Progression across key stages

After exploring the focus and progression within each key stage, it becomes possible to step back further and examine progression between stages. To capture this broad progression across the domain of programming and algorithms, it has been divided into two areas.

Figure 2 provides a summary of the programming constructs and patterns that learners encounter as they progress through the Teach Computing Curriculum. Each box represents a progressive step relating to one or more programming constructs (sequencing, selection, iteration, variables, and data structures) and is positioned according to where that step is explicitly taught. For example, whilst learners will likely encounter nested selection statements during key stage 2, they aren't formally taught how to apply them effectively until key stage 4.

The wider skills that learners need and develop whilst learning to program are captured in Figure 3 and divided into five broad categories. Whilst there are clearly some interconnections between the categories and skills, for simplicity and clarity these aren't shown. Many of these skills are developed over a period of time across multiple contexts, projects, and programming languages and hence span multiple key stages.

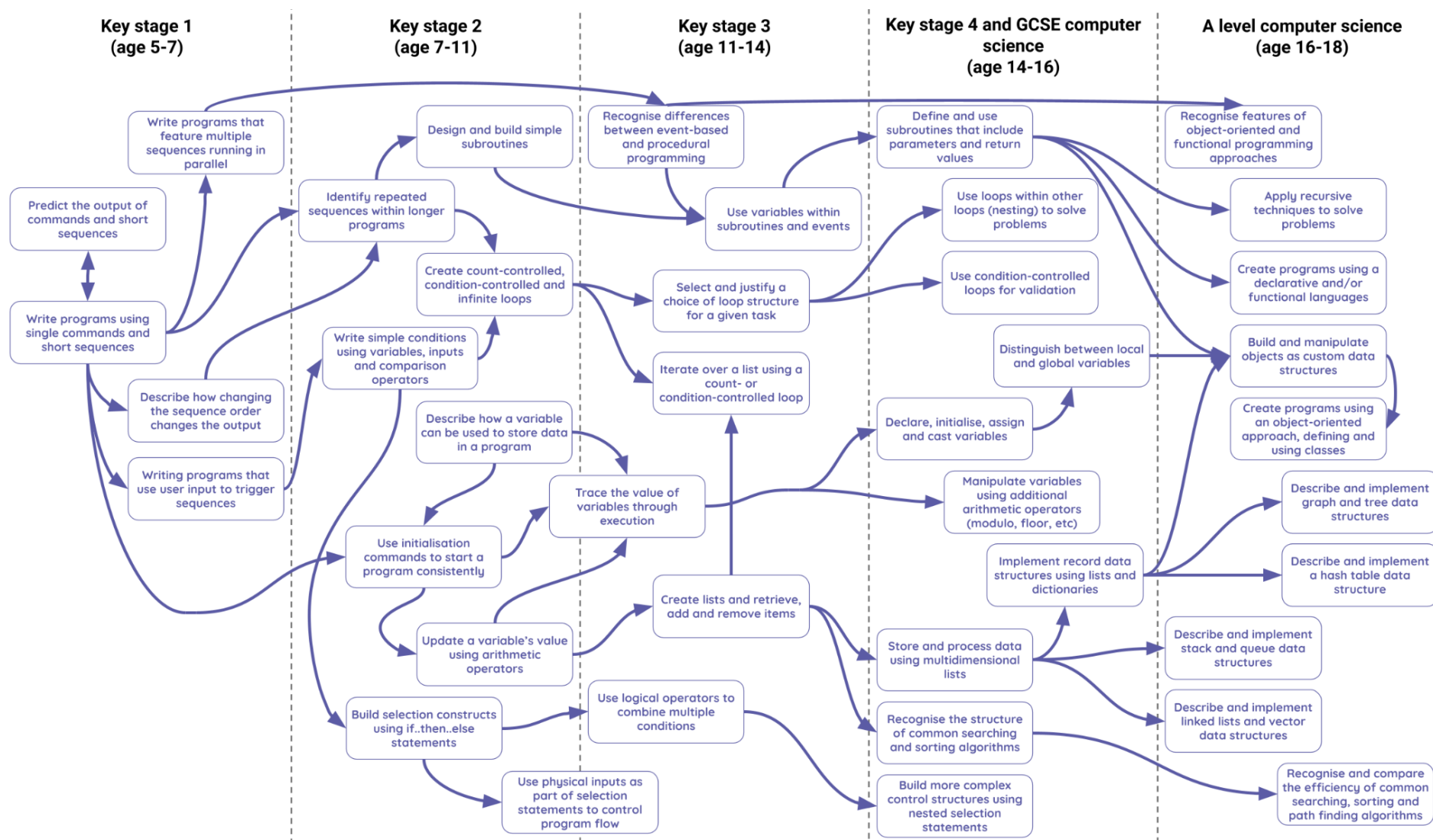


Figure 2: Programming constructs and patterns encountered throughout the Teach Computing Curriculum.

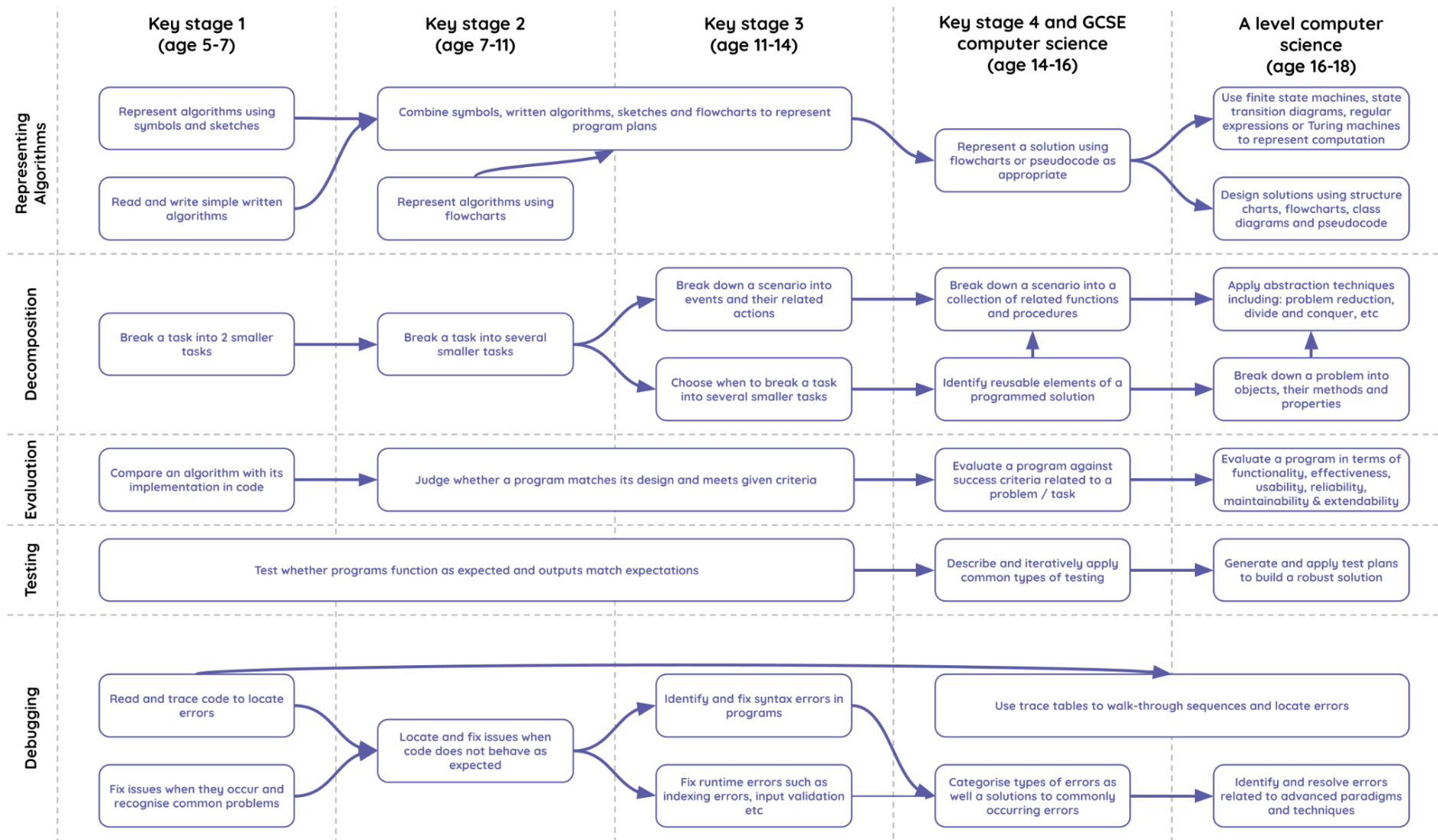


Figure 3: Wider skills required by students of programming.

4. Pedagogical strategies for programming and algorithms

4.1. Pedagogy principles

The work of the National Centre for Computing Education is underpinned by 12 pedagogical principles that can be exemplified by a range of evidence, informed practices, and strategies. These principles apply across the teaching of computing; however, some are more applicable than others in each strand of the curriculum.



Figure 4: NCCE's 12 pedagogy principles.



Programming and algorithms and how to teach these concepts have been an area of focus within the body of computing education research for some time. Whilst historically much of this research is based on studies within higher education settings, there are many approaches and practices that are thought to be effective based on the available evidence. Many of these individual approaches, ideas, and pedagogies can be explored in detail from a number of sources:

- The NCCE's collection of Pedagogy Quick Reads provides short digestible summaries on a range of topics¹⁰
- Hello World's *Big Book of Computing Pedagogy* provides similar summaries along with teacher stories and wider research insight¹¹
- A comprehensive review of programming pedagogies can be found in a recent report from the Raspberry Pi Foundation¹²

Here, we present the most relevant of our 12 pedagogical principles for programming and algorithms as well as some illustration of what they look like in practice.

Lead with concepts

Programming and algorithms is an area of computing where it is crucial to distinguish between the concepts being studied and the context, programming language, and tools being used. Throughout their journey with the Teach Computing Curriculum, pupils will encounter a range of programming languages and tools. By building their confidence and understanding of the fundamental programming constructs, students will likely be more readily able to transfer this knowledge to new languages.

- Language and vocabulary (and its underlying meaning) are really important, from how correctly and consistently we use key terms to how we describe the focus of the learning.
- Using tools such as displays, concept maps¹³, wikis, posters, etc. helps to develop a shared understanding amongst pupils of key concepts and consistent terms to label them.
- We should avoid conflating the learning of specific programming concepts and constructs with the specific languages or tools being used. Pupils aren't learning Scratch or Python, but instead learning to apply their programming skills to those languages.
- To support pupils in developing and transferring their conceptual understanding between languages, we should avoid (at least at first) some of the many short cuts or

¹⁰ National Centre for Computing Education. *Quick Reads*. <https://blog.teachcomputing.org/tag/quickread/> [Accessed 27 January 2022]

¹¹ Hello World. *Big Book of Computing Pedagogy*. Cambridge: Raspberry Pi Foundation. 2021.

¹² Waite, J, & Sentance, S. *Teaching programming in school: A review of approaches and strategies*. Raspberry Pi Foundation. 2021.
<https://www.raspberrypi.org/app/uploads/2021/11/Teaching-programming-in-schools-pedagogy-review-Raspberry-Pi-Foundation.pdf>

¹³ Raspberry Pi Foundation. *Quick Read: Using concept maps to capture, communicate, construct, and assess knowledge*. National Centre for Computing Education. 2020.
<https://blog.teachcomputing.org/using-concept-maps-to-capture-communicate-construct-and-assess-knowledge/>



idiosyncrasies provided by different languages. A good example of this is the standard `print` function in Python, which allows programmers to concatenate multiple values by supplying each as a parameter, or the `forever if` block, which exists in several block-based languages. Both examples hide some underlying complexities that aren't found in other languages.

- In a similar manner, we risk overwhelming students when we as experts create complex and compound lines of code that contain many concepts. Those same concepts spread over several lines of code are easier to read, decode, and trace and students can condense later (if they want) when their understanding is secure.

Model everything

As well as being a concept-rich part of the computing curriculum, programming and developing algorithms also involves many new skills that educators need to model.

- Use worked examples¹⁴ to model chunks or programming patterns that learners can read, trace, adapt, or complete, gradually removing this scaffold over time.
- Model the process of programming through live coding¹⁵, create programs live with and for your learners. Talk through your thoughts, the steps you are taking, and mistakes you are making.
- Modelling good programming practices is essential to help learners avoid forming bad habits, which may lead to misconceptions and ultimately hinder future learning. Some particular practices to be aware of are:
 - Always model the use of appropriate names for variables, functions, objects, etc.
 - Avoid having too many concepts or steps embedded within a single line of code; this makes them less readable and harder to debug.
 - Infinite loops are best avoided where possible as they provide no condition under which they will terminate short of terminating the whole program. Using a conditional loop ensures the loop can always gracefully exit and also provides clarity to anyone reading the code.
- Model to learners the choice of language, tool, and programming paradigm, and how to select the most appropriate of each. As learners progress, they experience new (and potentially more complex) tools and languages. However, each new tool or language isn't inherently better but instead compliments their existing tools for solving programming problems.

¹⁴ Raspberry Pi Foundation. *Quick Read: Using worked examples to support novice learners*. National Centre for Computing Education. 2019.

<https://blog.teachcomputing.org/using-worked-examples-to-support-novice-learners/>

¹⁵ Raspberry Pi Foundation. *Quick Read: Using live coding to bring coding to life*. National Centre for Computing Education. 2020. <https://blog.teachcomputing.org/quick-read-5-live-coding/>

Make concrete

Bring abstract concepts to life with real-world, contextual examples and a focus on interdependencies with other curriculum subjects.

- Talk about the programs and algorithms that surround your learners, particularly at home and school, but also present in their everyday lives.
- Where possible, adapt activities provided in the Teach Computing Curriculum to your learners' local community, experiences, cultural background¹⁶. Use programs to solve problems that matter to them.

Unplug, unpack, repack

To teach new concepts, first unpack complex terms and ideas, then explore these ideas in unplugged and familiar contexts, before you repack this new understanding into the original concept.

To help, teachers can apply a semantic wave¹⁷ approach. In simple terms, this encourages educators to:

- Present learners with an abstract concept: "A variable is a named reference to a space in memory, which a program can use to store, retrieve, and update data".
- Unpack the meanings within the concept and relate it to a familiar concept: "A variable is a little like a physical box with a name written on the side. It can store one thing at a time and to use it we can either replace its contents with something new (assignment) or examine, but not replace, the value it currently holds."
- Explore the concept within this familiar analogous context, perhaps using physical props to demonstrate how data is stored, retrieved, and updated.
- Repack the meanings of the original concept and draw similarities and differences between the analogy and the original computing context: "Unlike physical boxes, our variables can only store one item at a time. We use different types of variables to store different types of data including numbers, text, or Boolean data."
- Finally, return the original concept in its own context: "Variables are named references to parts of a computer's memory that a program can name and use to store, retrieve, and update different types of data."

¹⁶ Raspberry Pi Foundation. *Quick Read: Culturally relevant pedagogy*. National Centre for Computing Education. 2021. <https://blog.teachcomputing.org/quick-read-culturally-relevant-pedagogy/>

¹⁷ Raspberry Pi Foundation. *Quick Read: Using semantic waves to improve explanations and learning activities in computing*. National Centre for Computing Education. 2020. <https://blog.teachcomputing.org/quick-read-6-semantic-waves/>

Challenge misconceptions

Regardless of how well a concept is taught, there is always space for alternate conceptions¹⁸ (commonly known as misconceptions) to develop. In fact, sometimes we may knowingly introduce a misconception in order to simplify a concept or make it accessible. Recognising those misconceptions and knowing how to mitigate them is important, especially in an area of the curriculum that focuses on concepts.

- Teachers should make a conscious effort to seek out misconceptions and challenge them. Using regular formative assessment can help uncover misconceptions.
- Carefully written multiple choice questions can be used diagnostically¹⁹ with distractors (wrong answers) that each result from a specific misconception.
- Concept mapping is another useful tool. If learners create their own maps, these should be a reflection of their internal understanding and can help identify the root of a misconception.
- Peer instruction²⁰ is a particularly effective technique based on a flipped learning approach. Learners complete a task before the lesson, in which they 'learn' new concepts. The lesson time is then used to answer diagnostic questions collaboratively and relies on peer discussion to build consensus around a concept. It not only helps identify misconceptions, but also helps address and correct them.

Create projects

Pupils need opportunities to apply the skills, knowledge, and understanding that they have developed, and project-based²¹ activities can be a great way to facilitate this.

- 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 that they will use.
- Projects are a valuable context in which pupils can develop their design, analysis, and evaluation skills, as well as providing opportunities for collaboration.
- Projects rooted in the learner's experience and environment allow learners to solve problems that matter to them, increasing their intrinsic motivation to learn.
- Projects help learners develop their skills and understanding beyond computing as they involve them imagining, making, and sharing their ideas. Over the course of a project, learners will have to practice planning, organise their tasks and time available, and communicate their ideas and progress with stakeholders.

¹⁸ Raspberry Pi Foundation. *Quick Read: Addressing learners' alternate conceptions in computing*. National Centre for Computing Education. 2022.

¹⁹ Eedi. *Teach Computing NCCE*. <https://eedi.com/projects/teach-computing> [Accessed 21 June 2021]

²⁰ Raspberry Pi Foundation. *Quick Read: Using peer instruction to discuss computing concepts*. National Centre for Computing Education. 2019. <https://blog.teachcomputing.org/quick-read-4-peer-instruction/>

²¹ Raspberry Pi Foundation. *Quick Read: Using project-based learning to apply programming knowledge to real-world scenarios*. National Centre for Computing Education. 2021. <https://blog.teachcomputing.org/project-based-learning/>

Structure lessons

Like in any other subject, computing lessons benefit from structure, planning, and a well-thought-out learning journey. There are several frameworks that educators can use to help them structure their programming lessons.

- For example, before asking students to create something new, you might ask them first to use an existing example and modify that example before they create their own. This is the Use–Modify–Create framework and it can be really helpful in supporting learners to move from the examples of experts to building something that they understand and own themselves.
- A well-evidenced and popular framework that has been developed over the last few years is PRIMM²². PRIMM stands for Predict, Run, Investigate, Modify, and Make, representing the different stages of a lesson or series of lessons.
- Structuring lessons using such frameworks ensures that differentiation can be built in at various stages of the lesson. It ensures that cognitive load is managed and that students are supported, engaged, and challenged at the right moments.

Work together

Collaboration is crucial: not only is it highly prevalent in modern computing professions, but it is also a valuable way for individual pupils to learn from their peers.

- Working together stimulates classroom dialogue, articulation of concepts, and the development of shared understanding.
- Pair programming²³ is a really effective approach to programming, where pupils share the cognitive load placed upon them. Research demonstrates that this approach will support learners in developing their programming confidence and attainment.

Read and explore code

There is no doubt that programming can be a highly rewarding, highly satisfying experience for all. However, you should be in no rush to have your students write their first independent program; in doing so, you may miss out some important steps.

We typically wouldn't ask pupils to write before they had learnt the basics of reading, or encourage them to write number sentences before they could count. Likewise, there is a body of evidence that suggests that pupils who engage in reading code before they write code can enhance and improve their ability to write better code later on.

Research suggests that educators should be encouraging their pupils to engage with other people's code prior to writing their own. They should review it, interpret it, understand it, and manipulate it. This approach applies to all sorts of programming experiences, whether they are text-based (for example, Python) or block-based (for example, Scratch).

²² Raspberry Pi Foundation. *Quick Read: Using PRIMM to structure programming lessons*. National Centre for Computing Education. 2020. <https://blog.teachcomputing.org/using-primm-to-structure-programming-lessons/>

²³ Raspberry Pi Foundation. *Quick Read: Using pair programming to support learners*. National Centre for Computing Education. 2019. <https://blog.teachcomputing.org/quick-read-pair-programming-supports-learners/>

Get hands on

Physical computing²⁴ and making activities are shown to be highly engaging approaches for learners, giving them a sensory, tactile, and creative experience in which they can combine computing with art, craft, and design. Physical computing is both a tool to engage learners and a strategy to help them develop their understanding in more creative ways.

Through physical computing, learners can encounter, develop, and practise the whole range of programming skills and concepts, including sequences, loops, conditions, functions, and data structures. Alongside applying these concepts, learners will also encounter other languages, models of programming, and novel computer systems.

Foster program comprehension

When teaching programming, it is important for learners to understand a program from multiple perspectives, including how it is written (syntax and symbols) and how it executes, as well as its function or purpose.

There are many ways in which you might support program comprehension²⁵, but particular examples include activities that promote debugging, tracing, and the use of Parson's Problems. Additionally, tasks that ask pupils to consider the purpose of a program (or a small part of a program) are very helpful. Other beneficial tasks include selecting appropriate names for variables, functions, or entire programs; predicting the output of a program; or matching programs to their purpose.

²⁴ Raspberry Pi Foundation. *Quick Read: Physical computing*. National Centre for Computing Education. 2021. <https://blog.teachcomputing.org/quick-read-physical-computing/>

²⁵ Raspberry Pi Foundation. *Quick Read: Improving program comprehension through Parson's Problems*. National Centre for Computing Education. 2021. <https://blog.teachcomputing.org/quick-read-improving-program-comprehension-throughparsons-problems/>

5. Professional development for computing teachers

A core part of the NCCE's role is to help teachers develop their subject knowledge and pedagogy through continued professional development (CPD). There are a number of routes for teachers to participate in CPD to support their understanding of programming and algorithms.

Table 8 provides a sample of some of the courses that are available as part of the NCCE; these are designed to support teachers' development of their programming and algorithms subject knowledge. Many more courses, both online and face-to-face can be found on the [Teach Computing website](https://teachcomputing.org/courses)²⁶.

Teachers of A level computer science can find an additional range of [bespoke courses](https://isaaccomputerscience.org/events)²⁷ organised by Isaac Computer Science on a range of relevant topics.

Table 8: Courses to support teachers' development of programming and algorithms knowledge.

Key stage 1	Key stage 2	Key stage 3	Key stage 4	Key stage 5
Teaching key stage 1 computing	Teaching key stage 2 computing	Programming 101: An introduction to Python for educators		Object-oriented programming
Primary programming and algorithms		Programming pedagogy in secondary schools: Inspiring computing teaching		Programming a relational database using SQL
Programming pedagogy in primary schools: Developing computing teaching		Scratch to Python: Moving from block-to text-based programming	An introduction to algorithms, programming, and data in GCSE computer science	Dijkstra's algorithm and A*
	Physical computing – KS2 (Crumble)	Physical computing – KS3 (micro:bit)	Physical computing – KS4 Raspberry Pi Pico	Assembly language

Beyond accessing formal courses, there are many opportunities for computing teachers to learn through networks, such as Computing at Schools (CAS). These local communities continue to meet regularly and share best practices and skills and are therefore a great source of inspiration and development for teachers. As well as local support and meetups, teachers can find many [CPD focused events](https://community.computingatschool.org.uk/events)²⁸.

²⁶ Teach Computing. *Computing courses for teachers*. <https://teachcomputing.org/courses> [Accessed 21 June 2021]

²⁷ Isaac Computer Science. *Events*. <https://isaaccomputerscience.org/events> [Accessed 21 June 2021]

²⁸ Computing at School. *Upcoming events*. <https://community.computingatschool.org.uk/events> [Accessed 21 June 2021]

6. Conclusion

In this report, we explored programming and algorithms as a fundamental component of computing education. Developing knowledge and skills in this area enables learners to create, solve problems, and express ideas using computing devices. Regardless of their future aspirations, study, and careers, a firm understanding of the essentials of programming and algorithms equips students with a versatile skill for the future.

In reflecting on this area of the computing curriculum, we have used the Levels of Abstraction (LOA) framework to describe four key themes within programming and argued that an understanding of each level and the ability to move between them is a key factor in learners developing as programmers; these themes are a key part of the design principle behind the Teach Computing Curriculum (TCC). Alongside promoting understanding and movement between these levels, the Teach Computing Curriculum explores programming through a range of contexts, supporting learners in collecting and recalling common programming patterns, and gradually increasing their ownership over programming tasks and projects.

We have reviewed all 34 Teach Computing units and Isaac Computer Science topics, to provide an overarching view that highlights their position and focus within the levels of abstraction framework along with the key concepts and skills studied. From this, we can demonstrate varied coverage of the four levels of abstraction, with most units having a component of each of the four levels, whilst others (particularly at higher key stages) have a deeper focus on one or more particular levels compared with the others.

This same review has allowed us to uncover the high-level steps in progression present in the Teach Computing Curriculum and therefore the national curriculum it covers. This progression has been captured as two learning graph style diagrams showing the progression of programming constructs and the skills that sit alongside.

We hope you find this report useful and welcome feedback on it, via research@teachcomputing.org. In addition to this *Programming and Algorithms* report and our previous reports on [Computer Systems and Networking](#) and [Digital Literacy within the Computing Curriculum](#), we plan to publish similar reports on other topic areas within the computing curriculum in due course.