Computer Science: A curriculum for schools

Computing at School Working Group

http://www.computingatschool.org.uk

endorsed by BCS, Microsoft, Google and Intellect

March 2012

Foreword

The Computing at School Working Group recognises that Computer Science (CS) and Information Technology (IT) are disciplines within Computing that, like maths or history, every pupil should meet at school. The rationale is argued in "Computing at school: the state of the nation"[1] and reflected in the Royal Society Report, 'Shut down or restart?[2]. The purpose of this document is to describe and explain the content of Computer Science within the school curriculum.

If Computer Science should be taught at school we must answer the question "just what is Computer Science, viewed as a school subject?" Answering that question is the purpose of this document.

Structure and focus

This curriculum is modelled directly on the UK National Curriculum Programmes of Study[3], in the hope that it may thereby have a familiar "shape":

- **Section 1: Importance** of Computer Science at school.
- **Section 2: Key Concepts** that arise repeatedly in Computer Science.
- **Section 3: Key Processes** that pupils should be able to carry out.
- **Section 4: Range and Content** of what pupils should know.
- **Section 5: Level descriptions** of Computer Science attainment.

Because the subject matter might be unfamiliar we have taken space for explanation and examples, so the result is much longer than a typical National Curriculum subject specification.

This document identifies enduring principles rather than current hot topics, so there is little mention of mobile phones, the cloud, or social networking. Topics such as these are important, and are very likely to play a significant role in illustrating the application of underlying principles, aiding the effective delivery of Computer Science lessons, but they will change from year to year and so are not the basis for an enduring curriculum. The purpose of this curriculum is to articulate *what the Computer Science discipline is*, rather than *how it should be taught*.

Scope

Computer Science is so important (see Section 1) that:

- Every pupil at key stage 2 and key stage 3 should have the opportunity to learn material that is recognisably "Computer Science".
- Every pupil should have the opportunity to take a GCSE in Computing or Computer Science at key stage 4.

---

[1] http://www.computingatschool.org.uk/data/uploads/CAS_UKCRC_report.pdf
[2] http://royalsociety.org/education/policy/computing-in-schools/report
[3] http://curriculum.qcda.gov.uk/key-stages-3-and-4/subjects/key-stage-3/

- Every pupil should appreciate that computational ideas inform and illuminate other disciplines, and this should be reflected in the teaching of these disciplines at school. Like numeracy and literacy there is a cognitive strand of computing that offers valuable thinking skills to learners of all ages (e.g. algorithmics, logic, visualisation, precision, abstraction).

Key stages, primary and secondary education

The transition from primary to secondary school typically takes place at the start of key stage 3 and study during key stage 4 leads to the national GCSE examinations, and a variety of vocational equivalents. Post-16 education includes A-level examinations, which are the main university entrance qualification.

- **Key Stage 1 (ages 5-7)**
- **Key Stage 2 (ages 7-11)**
- **Key Stage 3 (ages 11-14)**.
- **Key Stage 4 (ages 14-16)**
- **Post-16 education (ages 16+)**

The Computing at School Working Group (Curriculum)

Miles Berry (University of Roehampton)
Kevin Bond (AQA)
Quintin Cutts (University of Glasgow)
Roger Davies (Queen Elizabeth School, Kirkby Lonsdale)
Mark Dorling (Langley Grammar School and Digital Schoolhouse Project)
Stephen Hunt (University of Hertfordshire)
Jack Lang (University of Cambridge)
Bill Mitchell (British Computer Society)
Adam McNicol (Long Road Sixth Form College, Cambridge)
Simon Peyton Jones (Microsoft Research, Cambridge)
Shahneila Saeed (Graveney School, London)
John Woollard (University of Southampton)
Emma Wright (Harvey Grammar School, Folkestone)

Contact: Curriculum@computingatschool.org.uk

Available: http://www.computingatschool.org.uk

Contents

Endorsements

BCS, The Chartered Institute for IT

BCS is the professional body for IT in the UK. One of its principal objectives, as defined by its Royal Charter, is to support the advancement of computer science education. BCS believe it is essential that schoolchildren from primary school onwards are taught how to create digital technology and software for themselves. In particular this means schoolchildren need to be introduced to the scientific and engineering principles and concepts of Computer Science. Although technology changes ever more rapidly, the principles and concepts that they are built on do not. The Computing At School curriculum lays out these important principles and concepts in an exemplary form that is particularly suitable for secondary schools. We believe this curriculum will be of value to schools for decades to come.

Microsoft

Microsoft has built its business on the knowledge and skills of our talented computer scientists and we therefore fully support the aims of the Computing at School Working Group (CAS). We were a foundation member and we promote their mission through all our networks and blogs. We see the **Curriculum for Computing** developed by CAS as providing a solid foundation for teaching the principles and concepts of computer science in a creative and animated way. This will help deliver the capabilities and experience which Microsoft needs for our workforce and our SME partners. This is particularly important because there is currently a shortage of people with the right Computer Science expertise in the UK.

Computer science is not just good for the economy, it is fun and it gives young people huge opportunities in life. For example, Kodu our visual programming language made specifically for creating games allows pupils to design games in an accessible manner. At university level, Microsoft's Imagine Cup world's premier pupil technology competition gives pupils the opportunity to solve tough problems facing the world today, and maybe even turn their ideas into a business. Furthermore, Microsoft is also supporting Computer Science teachers with resources through the 'Partners in Learning' scheme and our Dreamspark programme provides professional-level developer and design tools to pupils and educators around the world at no charge.

CAS and its partners including Microsoft are confident that they can support the expansion of Computer Science and we would like to see Government encourage schools to adopt it.

Google

As a company and as individual engineers, we at Google feel strongly that it is important that all schoolchildren get both a solid grounding in computational thinking and an opportunity to explore the fundamental skills and concepts that underpin Computing and Software Engineering. In particular, we believe every pupil should have exposure to and experience of the joy of programming, and specifically of engineering new things (out of software, physical materials or a blend of the two) to complement their education in Mathematics, Science, English, Music, Art, Languages and other core subjects that will shape their future career and life choices. There are multiple ways to achieve this, and Google continues to support a range of approaches, through our CS4HS scheme which funds Universities working with schools and through our support for relevant technologies and tools, such as Greenfoot. We directly and indirectly support the work of CAS in these areas, and view this example Curriculum for Computing as a useful and relevant contribution, describing important computing principles and concepts. We hope this will prove to be of widespread value in schools across the UK and beyond. We encourage schools to look at the CAS curriculum and consider it as one possible way to improve their provision in this topic area.

Intellect

Intellect is the trade association for the UK's technology sector, representing over 850 member companies from major multinationals to small technology businesses. Our work includes promoting the evidence of technology's role in driving outcomes in education and championing the value of STEM skills, computer science and digital literacy. The UK technology industry endorses the Computing at School curriculum; while effectively created by teachers for teachers, it establishes an excellent framework for learners to develop computer science principles and disciplines that will benefit them and industry.

1. Importance

Computer Science[4] is the study of principles and practices that underpin an understanding and modelling of computation, and of their application in the development of computer systems. At its heart lies the notion of computational thinking: a mode of thought that goes well beyond software and hardware, and that provides a framework within which to reason about systems and problems. This mode of thinking is supported and complemented by a substantial body of theoretical and practical knowledge, and by a set of powerful techniques for analysing, modelling and solving problems.

Computer Science is deeply concerned with how computers and computer systems work, and how they are designed and programmed. Pupils studying computing gain insight into computational systems of all kinds, whether or not they include computers. Computational thinking influences fields such as biology, chemistry, linguistics, psychology, economics and statistics. It allows us to solve problems, design systems and understand the power and limits of human and machine intelligence. It is a skill that empowers, and that all pupils should be aware of and have some competence in. Furthermore, pupils who can think computationally are better able to conceptualise and understand computer-based technology, and so are better equipped to function in modern society.

Computer Science is a practical subject, where invention and resourcefulness are encouraged. Pupils are expected to apply the academic principles they have learned to the understanding of real-world systems, and to the creation of purposeful artefacts. This combination of principles, practice, and invention makes it an extraordinarily useful and an intensely creative subject, suffused with excitement, both visceral ("it works!") and intellectual ("that is so beautiful").

1.1 Computer Science is a discipline

Education enhances pupils' lives as well as their life skills. It prepares young people for a world that doesn't yet exist, involving technologies that have not yet been invented, and that present technical and ethical challenges of which we are not yet aware.

To do this, education aspires primarily to teach disciplines with long-term value, rather than skills with short-term usefulness, although the latter are certainly useful. A "discipline" is characterised by:

- **A body of knowledge**, including widely-applicable ideas and concepts, and a theoretical framework into which these ideas and concepts fit.
- **A set of techniques and methods** that may be applied in the solution of problems, and in the advancement of knowledge.
- **A way of thinking and working** that provides a perspective on the world that is distinct from other disciplines.
- **Longevity**: a discipline does not "date" quickly, although the subject advances.
- **Independence from specific technologies**, especially those that have a short shelf-life.

[4] The term "Computing" refers to the whole of the curriculum related to the use of computers. The constituent parts are Computer Science (CS), Information Technology (IT), digital literacy (dl) and Technology Enhanced Learning (TEL). This document is concerned with the Computer Science aspect of the Computing curriculum.

Computer Science is a discipline with all of these characteristics. It encompasses foundational principles (such as the theory of computation) and widely applicable ideas and concepts (such as the use of relational models to capture structure in data). It incorporates techniques and methods for solving problems and advancing knowledge (such as abstraction and logical reasoning), and a distinct way of thinking and working that sets it apart from other disciplines (computational thinking). It has longevity (most of the ideas and concepts that were current 20 or more years ago are still applicable today), and every core principle can be taught or illustrated without relying on the use of a specific technology.

1.2 Computer Science is a STEM discipline

Computer Science is a quintessential STEM discipline, sharing attributes with Engineering, Mathematics, Science, and Technology:

- It has its own theoretical foundations and mathematical underpinnings, and involves the application of logic and reasoning.
- It embraces a scientific approach to measurement and experiment.
- It involves the design, construction, and testing of purposeful artefacts.
- It requires understanding, appreciation, and application of a wide range of technologies.

Moreover, Computer Science provides pupils with insights into other STEM disciplines, and with skills and knowledge that can be applied to the solution of problems in those disciplines.

Although they are invisible and intangible, software systems are among the largest and most complex artefacts ever created by human beings. The marriage between software and hardware that is necessary to realize computer-based systems increases the level of complexity, and the complex web of inter-relationships between different systems increases it yet further. Understanding this complexity and bringing it under control is the central challenge of our discipline. In a world where computer-based systems have become all pervasive, those individuals and societies that are best equipped to meet this challenge will have a competitive edge.

The combination of computational thinking, a set of computing principles, and a computational approach to problem solving is uniquely empowering. The ability to bring this combination to bear on practical problems is central to the success of science, engineering, business and commerce in the 21st century.

1.3 Computer Science and Information Technology are complementary, but they are not the same

Computer Science and Information Technology are complementary subjects. Computer Science teaches a pupil how to be an effective *author* of computational tools (i.e. software), while IT teaches how to be a thoughtful *user* of those tools. This neat juxtaposition is only part of the truth, because it focuses too narrowly on computers as a technology, and computing is much broader than that. As Dijkstra famously remarked, “Computer Science is no more about computers than astronomy is about telescopes”. More specifically:

- **Computer Science** is a discipline that seeks to understand and explore the world around us, both natural and artificial, in computational terms. Computer Science is particularly, but by no means exclusively, concerned with the study, design, and implementation of computer systems, and understanding the principles underlying these designs.
- **Information Technology** deals with the purposeful application of computer systems to solve real-world problems, including issues such as the identification of business needs, the specification and installation of hardware and software, and the evaluation of usability. It is the productive, creative and explorative use of technology.

We want our children to understand and play an active role in the digital world that surrounds them, not to be passive consumers of an opaque and mysterious technology. A sound understanding of computing concepts will help them see how to get the best from the systems they use, and how to solve problems when things go wrong. Moreover, citizens able to think in computational terms would be able to understand and rationally argue about issues involving computation, such as software patents, identity theft, genetic engineering, electronic voting systems for elections, and so on. In a world suffused by computation, every school-leaver should have an understanding of computing.

2. Key concepts

A number of key concepts arise repeatedly in computing. They are grouped here under

- Languages, machines, and computation
- Data and representation
- Communication and coordination
- Abstraction and design
- The wider context of computing.

It would not be sensible to teach these concepts as discrete topics in their own right. Rather, they constitute unifying themes that can be used as a way to understand and organise computing knowledge, and are more easily recognised by pupils after they have encountered several concrete examples of the concept in action.

2.1 Languages, machines, and computation

Computers get things done by a "*machine"* executing a "*program"*, written in some *language*.

- **Languages**. There is a huge range of programming languages, ranging from the machine code that the hardware executes directly, to high-level programming languages such as Java or C++. In principle computation can be expressed in any language, but in practice the choice of language is often influenced by the problem to be solved. Indeed, there are many special-purpose (or "domain specific") languages, such as SQL or Excel's formula language, designed for a particular class of applications. Unlike human languages, programming languages are necessarily very precise.
- **Algorithms**. An *algorithm* is a precise method of solving a problem. Algorithms range from the simple (such as instructions for changing a wheel on a car) to the ingenious (such as route-finding), and cover many different application areas (for example, drawing three-dimensional graphics; solving systems of constraints, such as a school timetable; understanding images; numerical simulation, and so on). An algorithm can be expressed as a program in many different programming languages.
- **Machines.** The most obvious "machine" is the hardware CPU, but many software layers implement virtual machines, an engine that to the layer above looks like a device for executing programs. Examples include hypervisors, the Java Virtual Machine, and programming environments such as Scratch.
- **Computational models**. A sequential "program" executes one step at a time, but that is not the only model of computation. Others include parallel computation, and the emergent behaviour of large numbers of simple agents (e.g. the way in which flocks of very simple automata can have unexpected collective behaviour).

2.2 Data and representation.

Much of the power of computers comes from their ability to store and manipulate very large quantities of data. The way in which this data is stored and manipulated can make

enormous differences to the speed, robustness, and security of a computer system. This area of computing includes:

- How data is **represented** using bit patterns: including numbers, text, music, pictures.
- How data is **stored and transmitted**, including:

  redundancy, error checking, error correction;

  data compression and information theory; and

  encryption.
- How data is **organised**, for example, in data structures or in databases.
- How **digital** data is used to represent **analogue** measures, such as temperature, light intensity and sound. How analogue measures are converted to digital values and vice versa and how digital computers may be used to control other devices.

2.3 Communication and coordination.

Computers are communication devices. They enable human-to-human communication by way of machine-to-machine communication: a mobile phone computes in order to help us communicate. The design and implementation of these communications systems is a recurrent theme in computing:

- Many programs are **reactive processes**, that perform **actions** in response to **events**. The input > process > output concept is important. For example, a web server receives a request for a page from the network, and then sends back a response containing the webpage.
- Such processes may run forever, and may (by design) behave differently on different runs.
- Computers **communicate** and **cooperate** through agreed protocols, such as TCP/IP or HTTP standards.
- These protocols may support **packet switching and routing** (to get a message to its destination), **authentication** (proving who you are), **privacy** (keeping a conversation private to the participants), and **anonymity**.
- A **network** is a set of computers connected to share data, facilities or resources; the **Internet** is a particular realisation of a network.

2.4 Abstraction and design

Abstraction is the main mechanism used to deal with complexity and enabling computerisation. Abstraction is both presenting a simplified version through information hiding and making an analysis to identify the essence or essential features.

Aspects of abstraction are:

- Computer **hardware** consists of components (black boxes) interacting through interfaces (a network cable, a CPU socket, a SATA disk interface).
- Computer **software** is built from layers of abstraction. For example, a procedure (or method, or library) implements a specification, but hides its

implementation; an operating system provides facilities to programs but hides the complex implementation; a database implements SQL queries, but hides how the data is stored.

- Scientists, industrialists, engineers, and business people all use computers to **simulate and model** the real world, by abstracting away unnecessary detail and using a computer program to simulate (what they hope is) the essence of the problem.
- Designing and hiding a complicated implementation ("how it works") behind an **interface** ("what it does").
- Representing or **modelling** through visualisations, metaphors or prose the essential features without the confusion of the detail.
- The process of **categorisation** or **classification** that breaks down a complex system into a systematic analysis or representation.

2.5 Computers and computing are part of a wider context

Computer systems have a profound impact on the society we live in, and computational thinking offers a new "lens" through which to look at ourselves and our world. The themes here are very open-ended, taking the form of questions that a thoughtful person might debate, rather than answers that a clever person might know.

- **Intelligence and consciousness.** Computer Science is about more than computers. Computer Science opens up philosophical questions such as: can a machine be intelligent? ...be conscious? ...be a person?
- **The natural world**. Computer Science gives us a way of looking at the natural world, ranging from using computers to model the living world (e.g. simulations of animal populations) to thinking of the natural world in computational terms, for example, the way DNA encodes the sequence of amino acids that make up proteins.
- **Creativity and intellectual property**. Games, music, movies, gallery installations and performing arts are all transformed by computing and online experiences would not be possible without it. Should artistic ways of working be integrated with computational thinking? Should software and other creative products be patentable? What is the role of open source software?
- **Moral and ethical implications** of using computers. For example, as our world becomes more interconnected, we should consider privacy and which information should be private and which open to scrutiny; we should question how the vulnerable or the digitally disenfranchised can be protected.

3. Key processes: computational thinking

A "key process" is something that a pupil of Computer Science should be able to *do*; Section 4 deals with what a pupil should *know*.

In Computer Science, the key processes focus upon *computational thinking*. Computational thinking is the process of *recognising* aspects of computation in the world that surrounds us, and *applying* tools and techniques from computing to understand and reason about both natural and artificial systems and processes.

Computational thinking is something that *people* do (rather than computers), and includes the ability to think logically, algorithmically and (at higher levels) recursively and abstractly. It is, however, a rather broad term. The rest of this section draws out particular aspects of computational thinking that are particularly accessible to, and important for, young people at school.

A well-rounded student of Computer Science will also be proficient in other generic skills and processes, including: thinking critically, reflecting on ones work and that of others, communicating effectively both orally and in writing, being a responsible user of computers, and contributing actively to society.

3.1 Abstraction: modelling, decomposing, and generalising

A key challenge in computational thinking is the scale and complexity of the systems we study or build. The main technique used to manage this complexity is abstraction[5]. The process of abstraction takes many specific forms, such as modelling, decomposing, and generalising. In each case, complexity is dealt with by hiding complicated details behind a simple abstraction, or model, of the situation. For example,

- The London Underground map is a simple model of a complex reality — but it is a model that contains precisely the information necessary to plan a route from one station to another.
- A procedure to compute square roots hides a complicated implementation (iterative approximation to the root, handling special cases) behind a simple interface (give me a number and I will return its square root).

Computational thinking values elegance, simplicity, and modularity over ad-hoc complexity.

Modelling

Modelling is the process of developing a representation of a real world issue, system, or situation, that captures the aspects of the situation that are important for a particular purpose, while omitting everything else. *Examples: London Underground map; storyboards for animations; a web page transition diagram; the position, mass, and velocity of planets orbiting one another.*

Different purposes need different models. *Example: a geographical map of the Underground is more appropriate for computing travel times than the well-known*

[5] Somewhat confusingly, in computing the term "abstraction" is used both as a **verb** (a process, described here), and as a **noun** (a concept, described in Section 3.4).

topological Underground map; a network of nodes and edges can be represented as a picture, or as a table of numbers.

A particular situation may need more than one model. *Example: a web page has a structural model (headings, lists, paragraphs), and a style model (how a heading is displayed, how lists are displayed). A browser combines information from both models as it renders the web page.*

Decomposing

A problem can often be solved by decomposing it into sub-problems, solving them, and composing the solutions together to solve the original problem. For example "Make breakfast" can be broken down into "Make toast; make tea; boil egg". Each of these in turn can be decomposed, so the process is naturally recursive.

The organisation of data can also be decomposed. For example, the data representing the population of a country can be decomposed into entities such as individuals, occupations, places of residence, etc.

Sometimes this top-down approach is the way in which the solution is *developed*; but it can also be a helpful way of *understanding* a solution regardless how it was developed in the first place. For example, an architectural block diagram showing the major components of a computer system (e.g. a client, a server, and a network), and how they communicate with each other, can be a very helpful way of understanding that system.

Generalising and classifying

Complexity is often avoided by generalising specific examples, to make explicit what is shared between the examples and what is different about them. For example, having written a procedure to draw a square of size 3 and another to draw a square of size 5, one might generalise to a procedure to draw a square of any size N, and call that procedure with parameters 3 and 5 respectively. In this way much of the code used in different programs can be written once, debugged once, documented once, and (most important) understood once.

A different example is the classification encouraged by object-oriented languages, whereby a parent class expresses the common features of an object, for example, the size or colour of a shape, while the sub-classes express the distinct features (a square and a triangle, perhaps).

Generalisation is the process of recognising these common patterns, and using them to control complexity by sharing common features.

3.2 Programming

Computer Science is more than programming, but programming is an absolutely central process for Computer Science. In an educational context, programming encourages creativity, logical thought, precision and problem-solving, and helps foster the personal, learning and thinking skills required in the modern school curriculum. Programming gives concrete, tangible form to the idea of "abstraction", and repeatedly shows how useful it is.

Designing and writing programs

Every pupil should have repeated opportunities to design, write, run, and debug, executable programs. What an "executable program" means can vary widely, depending on the level of the pupil and the amount of time available. For example, all of the following are included in "programming":

- Small domain-specific languages, such as instructions to a simple robot, or Logo-style turtle.
- Visual languages such as Scratch BYOB or Kodu.
- Text-based languages, such as C#, C++, Haskell, Java, Pascal, PHP, Python, Visual Basic, and so on.
- Spreadsheet formulae

Both interpreted and compiled languages are "executable". In every case the underlying concepts are more important than the particular programming language or environment.

The ability to **understand and explain a program** is much more important than the ability to produce working but incomprehensible code. Depending on level, pupils should be able to:

- Design and write programs that include
    - Sequencing: doing one step after another.
    - Selection (if-then-else): doing either one thing or another.
    - Repetition (Iterative loops or recursion).
    - Language constructs that support abstraction: wrapping up a computation in a named abstraction, so that it can be re-used. (The most common form of abstraction is the notion of a "procedure" or "function" with parameters.)
    - Some form of interaction with the program's environment, such as input/output, or event-based programming.
- Find and correct errors in their code
- Reflect thoughtfully on their program, including assessing its correctness and fitness for purpose; understanding its efficiency; and describing the system to others.

Abstraction mechanisms

Effective use of the abstraction mechanisms supported by programming languages (functions, procedures, classes, and so on) is central to managing the complexity of large programs. For example, a procedure supports abstraction by hiding the complex details of an *implementation* behind a simple *interface*.

These abstractions may be deeply nested, layer upon layer. *Example: a procedure to draw a square calls a procedure to draw a line; a procedure to draw a line calls a procedure to paint a pixel; the procedure to paint a pixel calls a procedure to calculate a memory address from an (x,y) pixel coordinate.*

As well as using procedures and libraries built by others, pupils should become proficient in creating new abstractions of their own. A typical process is

- Recognise that one is writing more or less the same code repeatedly. *Example: draw a square of size 3; draw a square of size 7.*

- Designing a procedure that generalises these instances. *Example: draw a square of size N.*
- Replace the instances with calls to the procedure. At a higher level, recognising a standard "design pattern", and re-using existing solutions**,** is a key process. For example:
- Simple data structures, such as variables, records, arrays, lists, trees, hash tables.
- Higher level design patterns: divide and conquer, pipelining, caching, sorting, searching, backtracking, recursion, client/server, model/view/controller.

Debugging, testing, and reasoning about programs

When a programmed system goes wrong, how can I fix it? Computers can appear so opaque that fault-finding degenerates into a demoralising process of trying randomly generated "solutions" until something works. Programming gives pupils the opportunity to develop a systematic approach to detecting, diagnosing and correcting faults, and to develop debugging skills, including:

- Reading and understanding documentation.
- Explaining how code works or might not work.
- Manually executing code, perhaps using pencil and paper.
- Isolating or localising faults by adding tracing or profiling.
- Adding comments to make code more human readable.
- Adding error checking code to check internal consistency and logic.
- Finding the code that causes an error and correcting it.
- Choosing test cases and constructing tests.

Note: in Key Stages 2-4 we do not recommend significant attention to software development processes (requirements analysis, specification, documentation, test plans etc.). These are very important topics for pupils who specialise in the subject, but they tend to obscure or dominate the other teaching goals.

4. Range and content: what a pupil should know

This section says what a pupil should know by the end of Key Stage 1, 2, 3 and 4. It should not be read as a statement of how the subject should be taught but simply as a summary of what a pupil should know.

What can actually be taught at, say, key stage 3 depends on how much curriculum time is available, and that will vary from school to school, and with changes in educational policy. Rather than prejudge this issue, this curriculum focuses on age-appropriate material; that is, the Key Stage 3 material should be comprehensible to a Key Stage 3 pupil. Almost certainly not all of it will fit, and teachers will need to select material from the range offered here.

Initially, pupils in key stages 2 to 4 might not have experienced computer science previously and so when planning the curriculum, the range and content of the previous key stage(s) should be considered.

Examples and text in [square brackets] are intended as illustrative, not prescriptive. Material marked (**) is more advanced.

4.1 Algorithms

A pupil should understand what an algorithm is, and what algorithms can be used for.

KEY STAGE 1

- Algorithms are sets of instructions for achieving goals, made up of pre-defined steps [the 'how to' part of a recipe for a cake].
- Algorithms can be represented in simple formats [storyboards and narrative text].
- They can describe everyday activities and can be followed by humans and by computers.
- Computers need more precise instructions than humans do.
- Steps can be repeated and some steps can be made up of smaller steps.

KEY STAGE 2

- Algorithms can be represented symbolically [flowcharts] or using instructions in a clearly defined language [turtle graphics].
- Algorithms can include selection (if) and repetition (loops).
- Algorithms may be decomposed into component parts (procedures), each of which itself contains an algorithm.
- Algorithms should be stated without ambiguity and care and precision are necessary to avoid errors.
- Algorithms are developed according to a plan and then tested. Algorithms are corrected if they fail these tests.
- It can be easier to plan, test and correct parts of an algorithm separately.

KEY STAGE 3

- An algorithm is a sequence of precise steps to solve a given problem.
- A single problem may be solved by several different algorithms.

- The choice of an algorithm to solve a problem is driven by what is required of the solution [such as code complexity, speed, amount of memory used, amount of data, the data source and the outputs required].
- The need for accuracy of both algorithm and data [difficulty of data verification; garbage in, garbage out]

KEY STAGE 4

- The choice of an algorithm should be influenced by the data structure and data values that need to be manipulated.
- Familiarity with several key algorithms [sorting and searching].
- The design of algorithms includes the ability to easily re-author, validate, test and correct the resulting code.
- Different algorithms may have different performance characteristics for the same task.

4.2 Programs

A pupil should know how to write executable programs in at least one language.

KEY STAGE 1

- Computers (understood here to include all devices controlled by a processor, thus including programmable toys, phones, game consoles and PCs) are controlled by sequences of instructions.
- A computer program is like the narrative part of a story, and the computer's job is to do what the narrator says. Computers have no intelligence, and so follow the narrator's instructions blindly.
- Particular tasks can be accomplished by creating a program for a computer. Some computers allow their users to create their own programs.
- Computers typically accept inputs, follow a stored sequence of instructions and produce outputs.
- Programs can include repeated instructions.

KEY STAGE 2

- A computer program is a sequence of instructions written to perform a specified task with a computer.
- The idea of a program as a sequence of *statements* written in a programming language [Scratch]
- One or more mechanisms for *selecting* which statement sequence will be executed, based upon the value of some data item
- One or more mechanisms for *repeating* the execution of a sequence of statements, and using the value of some data item to control the number of times the sequence is repeated
- Programs can model and simulate environments to answer "What if" questions.

- Programs can be created using visual tools. Programs can work with different types of data. They can use a variety of control structures [selections and procedures].
- Programs are unambiguous and that care and precision is necessary to avoid errors.
- Programs are developed according to a plan and then tested. Programs are corrected if they fail these tests.
- The behaviour of a program should be planned.
- A well-written program tells a reader the story of how it works, both in the code and in human-readable comments
- A web page is an HTML script that constructs the visual appearance. It is also the carrier for other code that can be processed by the browser.
- Computers can be programmed so they appear to respond 'intelligently' to certain inputs.

KEY STAGE 3

- Programming is a problem-solving activity, and there are typically many different programs that can solve the same problem.
- Variables and assignment.
- Programs can work with different types of data [integers, characters, strings].
- The use of relational operators and logic to control which program statements are executed, and in what order
    - Simple use of AND, OR and NOT
    - How relational operators are affected by negation [e.g. NOT (a>b) = a≤b].
- Abstraction by using functions and procedures (definition and call), including:
    - Functions and procedures with parameters.
    - Programs with more than one call of a single procedure.
- Documenting programs to explain how they work.
- Understanding the difference between errors in program syntax and errors in meaning. Finding and correcting both kinds of errors.

KEY STAGE 4

- Manipulation of logical expressions, e.g. truth tables and Boolean valued variables.
- Two-dimensional arrays (and higher).
- Programming in a low level language.
- Procedures that call procedures, to multiple levels. [Building one abstraction on top of another.]
- Programs that read and write persistent data in files.
- Programs are developed to meet a specification, and are corrected if they do not meet the specification.
- Documenting programs helps explain how they work.

4.3 Data

A pupil should understand how computers represent data:

KEY STAGE 1

- Information can be stored and communicated in a variety of forms e.g. numbers, text, sound, image, video.
- Computers use binary switches (on/off) to store information.
- Binary (yes/no) answers can directly provide useful information (e.g. present or absent), and be used for decision.

KEY STAGE 2

- Similar information can be represented in multiple.
- Introduction to binary representation [representing names, objects or ideas as sequences of 0s and 1s].
- The difference between constants and variables in programs.
- Difference between data and information.
- Structured data can be stored in tables with rows and columns. Data in tables can be sorted. Tables can be searched to answer questions. Searches can use one or more columns of the table.
- Data may contain errors and that this affects the search results and decisions based on the data. Errors may be reduced using verification and validation.
- Personal information should be accurate, stored securely, used for limited purposes and treated with respect.

KEY STAGE 3

- Introduction to binary manipulation.
- Representations of:
    - Unsigned integers
    - Text. [Key point: each character is represented by a bit pattern. Meaning is by convention only. Examples: Morse code, ASCII.]
    - Sounds [both involving analogue to digital conversion, e.g. WAV, and free of such conversion, e.g. MIDI]
    - Pictures [e.g. bitmap] and video.
- Many different things may share the same representation, or "the meaning of a bit pattern is in the eye of the beholder" [e.g. the same bits could be interpreted as a BMP file or a spreadsheet file; an 8-bit value could be interpreted as a character or as a number].
- The things that we perceive in the human world are not the same as what computers manipulate, and translation in both directions is required [e.g. how sound waves are converted into an MP3 file, and vice versa]
- There are many different ways of representing a single thing in a computer. [For example, a song could be represented as:

    - A scanned image of the musical score, held as pixels
    - A MIDI file of the notes
    - A WAV or MP3 file of a performance]
- Different representations suit different purposes [e.g. searching, editing, size, fidelity].

KEY STAGE 4

- Hexadecimal
- Two's complement signed integers
- String manipulation
- Data compression; lossless and lossy compression algorithms (example JPEG)
- Problems of using discrete binary representations:
    - Quantization: digital representations cannot represent analogue signals with complete accuracy [e.g. a grey-scale picture may have 16, or 256, or more levels of grey, but always a finite number of discrete steps]
    - Sampling frequency: digital representations cannot represent continuous space or time [e.g. a picture is represented using pixels, more or fewer, but never continuous]
    - Representing fractional numbers

4.4 Computers

A pupil should know the main components that make up a computer system, and how they fit together (their architecture).

KEY STAGE 1

- Computers are electronic devices using stored sequences of instructions.
- Computers typically accept input and produce outputs, with examples of each in the context of PCs.
- Many devices now contain computers

KEY STAGE 2

- Computers are devices for executing programs.
- Application software is a computer program designed to perform user tasks.
- The operating system is a software that manages the relationship between the application software and the hardware
- Computers consist of a number of hardware components each with a specific role [e.g. CPU, Memory, Hard disk, mouse, monitor].
- Both the operating system and application software store data (e.g. in memory and a file system)

- The above applies to devices with embedded computers (e.g. digital cameras), handheld technology (e.g. smart phones) and personal computers.
- A variety of operating systems and application software is typically available for the same hardware.
- Users can prevent or fix problems that occur with computers (e.g. connecting hardware, protection against viruses)
- Social and ethical issues raised by the role of computers in our lives.

KEY STAGE 3

- Computers are devices for executing programs
- Computers are general-purpose devices (can be made to do many different things)
- Not every computer is obviously a computer (most electronic devices contain computational devices)
- Basic architecture: CPU, storage (e.g. hard disk, main memory), input/output (e.g. mouse, keyboard)
- Computers are very fast, and getting faster all the time (Moore's law)
- Computers can 'pretend' to do more than one thing at a time, by switching between different things very quickly

KEY STAGE 4

- Logic gates: AND/OR/NOT. Circuits that add. Flip-flops, registers (**).
- Von Neumann architecture: CPU, memory, addressing, the fetch-execute cycle and low-level instruction sets. Assembly code. [LittleMan]
- Compilers and interpreters (what they are; not how to build them).
- Operating systems (control which programs run, and provide the filing system) and virtual machines.

4.5 Communication and the Internet

A pupil should understand the principles underlying how data is transported on the Internet.

KEY STAGE 1

- That the World Wide Web contains a very large amount of information.
- Web browser is a program used to use view pages.
- Each website has a unique name.
- Enter a website address to view a specific website and navigate between pages and sites using the hyperlinks.

KEY STAGE 2

- The Internet is a collection of computers connected together sharing the same way of communicating. The internet is not the web, and the web is not the internet.

- These connections can be made using a range of technologies (e.g. network cables, telephone lines, wifi, mobile signals, carrier pigeons)
- The Internet supports multiple services (e.g. the Web, e-mail, VoIP)
- The relationship between web servers, web browsers, websites and web pages.
- The format of URLs.
- The role of search engines in allowing users to find specific web pages and a basic understanding of how results may be ranked.
- Issues of safety and security from a technical perspective.

KEY STAGE 3

- A network is a collection of computers working together
- An end-to-end understanding of what happens when a user requests a web page in a browser, including:
    - Browser and server exchange messages over the network
    - What is in the messages [http request, and HTML]
    - The structure of a web page - HTML, style sheets, hyperlinking to resources
    - What the server does [fetch the file and send it back]
    - What the browser does [interpret the file, fetch others, and display the lot]
- How data is transported on the Internet
    - Packets and packet switching
    - Simple protocols: an agreed language for two computers to talk to each other. [Radio protocols “over”, “out”; ack/nack; ethernet protocol: first use of shared medium, with backoff.]
- How search engines work and how to search effectively. Advanced search queries with Boolean operators.

KEY STAGE 4

- Client/server model.
- MAC address, IP address, Domain Name service, cookies.
- Some “real” protocol. [Example: use telnet to interact with an HTTP server.]
- Routing
- Deadlock and livelock
- Redundancy and error correction
- Encryption and security

4.6 Optional topics for advanced pupils

Computer Science offers an enormous range of more advanced topics, all of which are accessible to a motivated key stage 4 pupil. The list here should not be regarded as exhaustive.

Algorithms

- Modular arithmetic

- Hashing
- Distributed algorithms
- Optimisation algorithms and heuristics; “good enough” solutions [genetic algorithms, simulated annealing];
- Monte Carlo methods
- Learning systems [matchbox computer]
- Biologically inspired computing; artificial neural networks, Cellular automata, Emergent behaviour
- Graphics [rotating a 3D model]

Programming

- Implementing recursive algorithms
- Programming for the real world
- Robotics
- Other language types and constructs: object oriented and functional languages
- App development
- Developing for different environments
- Programming using SDKs and other hardware
- open source

Data

- List graphs and trees including binary tree traversals
- Pointers and dynamic data structures
- Handling very large data sets
- Handling dynamic data sets (especially internet-based data)
- Floating point representation

Computers

- Interrupts and real-time systems
- Multiprocessor systems
- Memory caches
- Undecidability problems

Communications and Internet

- Asymmetric encryption; key exchange

Human Computer Interaction (HCI)

- Recognition of the importance of the user interface; computers interact with *people*.
- Simple user-interface design guidelines

5. Attainment targets for Computer Science

The level descriptions provide the basis for making judgements about pupils' performance at the end of key stages 1, 2 and 3. At key stage 4, national qualifications are the main means of assessing attainment with GCSEs in Computing and Computer Science offering clear benchmarks.

However, the National Curriculum is under review and there are to be changes in the way Levels are being treated. It is proposed that the level statements should be written as statements summarising the "readiness" of the pupil to reach the next level of development. Hence, it is not "I have done such as such, so I am at Level 4", instead, "I can control one variable, so I am ready to compare this with controlling two variables." The idea is to focus on describing the relevance of what the pupil is learning to the next stage of development - so there is an on-going urge to move forward - not to stop at an artificial point and say "I've done it".

Therefore, in line with the review of the National Curriculum, these Level statements will be revised. This document will continue to evolve to reflect the current progress in the National Curriculum review.

Currently, the range of levels within which the great majority of pupils are expected to work are:

- Key Stage 1 works at levels 1 – 3;
  – at age 7 they are expected to be at Level 2
- Key Stage 2 works at levels 2 – 5;
  – at age 11 they are expected to be at Level 4
- Key Stage 3 works at levels 3 – 7;
  – at age 14 they are expected to be at Level 5

Level 1

Pupils can talk about existing storyboards of everyday activities.

Pupils can order a collection of pictures into the correct sequence.

Pupils recognise that many everyday devices respond to signals and instructions.

Pupils can make programmable toys carry out instructions.

Level 2

Pupils draw their own storyboards of everyday activities.

Pupils plan and give direct commands to make things happen such as playing robots.

Pupils solve simple problems using programmable toys.

Pupils classify items in simple sets of data.

Level 3

Pupils recognise similarities between storyboards of everyday activities.

Pupils plan a linear (non-branching) sequence of instructions.

Pupils give a linear sequence of instructions to make things happen.

Pupils develop and improve their instructions.

Pupils present data in a systematic way.

Level 4

Pupils analyse and represent symbolically a sequence of events.

Pupils recognise different types of data: text; number; instruction.

Pupils understand the need for care and precision of syntax and typography in giving instructions.

Pupils can give instructions involving selection and repetition.

Pupils can 'think through' an algorithm and predict an output.

Pupils can present data in a structured format suitable for processing.

Level 5

Pupils partially decompose a problem into its sub-problems and make use of a notation to represent it.

Pupils analyse and present an algorithm for a given task.

Pupils recognise similarities between simple problems and the commonality in the algorithms used to solve them.

Pupils explore the effects of changing the variables in a model or program.

Pupils develop, try out and refine sequences of instructions, and show efficiency in framing these instructions. They are able to reflect critically on their programs in order to make improvements in subsequent programming exercises.

Pupils are able to make use of procedures without parameters in their programs; Pupils will also be able to manipulate strings and select appropriate data types.

Pupils can design and use simple (1D) data structures.

Level 6

Pupils describe more complex algorithms, for example, sorting or searching algorithms.

Pupils can describe systems and their components using diagrams.

Pupils can fully decompose a problem into its sub-problems and can make use of a notation to represent it.

Pupils can recognise similarities in given simple problems and able to produce a model which fits some aspects of these problems.

Pupils use programming interfaces to make predictions and vary the rules within the *programs*. Pupils assess the validity of their programs by considering or comparing alternative solutions.

Pupils are capable of independently writing or debugging a short program.

Pupils make use of procedures with parameters and functions returning values in their programs and are also able to manipulate 1-dimensional arrays.

Pupils can design and use 2D data structures.

Level 7

Pupils describe key algorithms, for example sorting/searching, parity, and are aware of efficiency.

Pupils can fully decompose a problem into its sub-problems and can make error-free use of an appropriate notation to represent it.

Pupils can recognise similarities in given more complex problems and are able to produce a model which fits some aspects of these problems.

Pupils use pre-constructed modules of code to build a system.

Pupils can design and use complex data structures including relational databases.

Pupils select and use programming tools suited to their work in a variety of contexts, translating specifications expressed in ordinary language into the form required by the system.

Pupils consider the benefits and limitations of programming tools and of the results they produce, and pupils use these results to inform future judgements about the quality of their programming.

Pupils program in a text-based programming language, demonstrating the processes outlined above. Pupils document and demonstrate that their work is maintainable. Pupils can debug statements.

Pupils can analyse complex data structures, use them in programs and simplify them.

Level 8

Pupils independently select appropriate programming constructs for specific tasks, taking into account ease of use and suitability.

Pupils can recognise similarities in more complex problems and are able to produce a model that fits most aspects of these problems

Pupils independently write the program for others to use and apply advanced debugging procedures.

Pupils can analyse, use and simplify complex data structures, for example, normalisation.

Pupils demonstrate an understanding of the relationship between complex real life and the algorithm, logic and visualisations associated with programming.

Exceptional performance

Pupils can recognise similarities between more complex problems, and are able to produce a general model that fits aspects of them all.

Pupils competently and confidently use a general-purpose text-based programming language to produce solutions for problems using code efficiently.