

# Computing: A curriculum for schools

---

Computing at School Working Group  
<http://www.computingatschool.org.uk>

October 2011

**(c) Copyright 2011 Computing At School.**

This work is licensed under the Creative Commons Attribution-NonCommercial license; see <http://creativecommons.org/licenses/by-nc/3.0/> for details

## Foreword

---

The Computing at School Working Group<sup>1</sup> believes that Computing is a discipline that, like maths or history, every student should meet at school. The current state of affairs is very different, as we argue in “Computing at school: the state of the nation”<sup>2</sup>.

If Computing should be taught at school we must answer the question “just what is Computing, 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<sup>3</sup>, in the hope that it may thereby have a familiar “shape”:

- **Section 1: Importance.** Why should Computing be learned at school?
- **Section 2: Key Concepts** that arise again and again in Computing.
- **Section 3: Key Processes.** What students should be able to do.
- **Section 4: Range and Content.** What students should know.
- **Section 5: Level descriptions** maps Computing attainment to UK National Curriculum levels.

Because the material is less familiar we have taken space for explanation and examples, so the result is much longer than a typical National Curriculum subject specification.

This document identifies the enduring principles rather than current hot topics, so there is little mention of mobile phones, the cloud, or social networking. These topics are important, and are very likely to play an important role in the effective delivery of Computing lessons, but they will change from year to year and so not the basis for a enduring curriculum.

The purpose of this curriculum is to articulate *what the Computing discipline is*, rather than *how it should be taught*. A companion document, “**Delivering a Computing curriculum**” addresses the latter question.

### Scope

Computing is so important (see Section 1) that:

- Every child at KS2 and KS3 should have the opportunity to learn material that is recognisably “Computing”.
- Every child should have the opportunity to take a GCSE in Computing at KS4.
- 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. algorithm, logic, visualisation, precision, abstraction).

---

<sup>1</sup> <http://www.computingatschool.org.uk/>

<sup>2</sup> [http://www.computingatschool.org.uk/data/uploads/CAS\\_UKCRC\\_report.pdf](http://www.computingatschool.org.uk/data/uploads/CAS_UKCRC_report.pdf)

<sup>3</sup> <http://curriculum.qcda.gov.uk/key-stages-3-and-4/subjects/key-stage-3/>

Most of the document is generic to Computing at school level, from primary (Key Stage 1, 2) to secondary (Key Stage 3, 4), and beyond. However, Section 4 (Range and Content) focuses primarily on Key Stages 3 and 4. *A future revision will extend coverage to include Key Stages 1 and 2.*

## Key stages

Here is a key for those unfamiliar with educational jargon in England:

- **Key Stage 1 (ages 5-8)**
- **Key Stage 2 (ages 8-11)**
- **Key Stage 3 (ages 11-14)**. The transition from primary to secondary school typically takes place at the start of KS3.
- **Key Stage 4 (ages 14-16)** leads up 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.

## The Computing at School Working Group

This document is a working document of the Computing at School Working Group (<http://www.computingatschool.org.uk>).

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)

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: Simon Peyton Jones ([simonpj@microsoft.com](mailto:simonpj@microsoft.com))

## 1. Importance

---

Computing<sup>4</sup> 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.

Computing 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 students 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.

Computing 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 Computing 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.

---

<sup>4</sup> We use the term “Computing” because that is the term used by UK school teachers. A university lecturer would use the term “Computer Science”. If you are more familiar with the latter, simply substitute “Computer Science” every time you read “Computing”.

- **Longevity:** a discipline does not “date” quickly, although the subject advances.
- **Independence from specific technologies,** especially those that have a short shelf-life.

Computing 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 Computing is a STEM discipline

Computing 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, Computing provides students 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 Computing and ICT are complementary, but they are not the same

Computing and ICT are complementary subjects. Computing teaches a student how to be an effective *author* of computational tools (i.e. software), while ICT 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, "Computing is no more about computers than astronomy is about telescopes". More specifically:

- **Computing** is a discipline that seeks to understand and explore the world around us, both natural and artificial, in computational terms. Computing is particularly, but by no means exclusively, concerned with the study, design, and implementation of computer systems, and the principles underlying these designs.
- **ICT** 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.

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 computers.

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 students 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 any 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). A single 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
  - Encryption
- How data is **organised**, for example in data structures; or in databases.

## 2.3 Communication and coordination.

Nowadays computers are often thought of primarily as communication devices: a mobile phone computes in order to communicate. The design and implementation of these communications systems is a recurrent theme in computing:

- Many programs are really **reactive processes**, that perform **actions** in response to **events**. 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.
- Networked computers **communicate and cooperate** using standardised protocols, such as TCP/IP or HTTP.
- 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**.
- The **Internet** is a particular realisation of a network.
- **Distributed algorithms** allow computers to cooperate in the presence of network failures and even malicious agents.

## 2.4 Abstraction and design

Computers are very simple machines. They gain their power through scale: by executing instructions extremely quickly, and by manipulating lots of data. This scale makes it all too easy to construct computer systems that no one can understand. Abstraction is the main mechanism we use to control complexity: hiding a complicated implementation ("how it works") behind a simple interface ("what it does"). Modular design, and powerful abstraction shows up everywhere in computing. For example:

- Computer **hardware** consists of complex boxes interacting through well-defined interfaces (a network cable, a CPU socket, a SATA disk interface).
- All **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.

- **Simulation and modelling.** Scientists, business people, and financial analysts all use computers to 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.

## 2.5 Computers 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.** Computing is about more than computers. Computing opens up philosophical questions such as: can a machine be intelligent? ...be conscious? ...be a person?
- **The natural world.** Computing gives us a way of looking at the living world, ranging from using computers to model the real world (e.g. simulations of animal populations) to thinking of the living world in directly computational terms (e.g. the “program” that is encoded by DNA).
- **Creativity.** Games, online experiences, movies, gallery installations and performing arts are all transformed by computing. Should artistic ways of working be integrated with computational thinking?
- **Privacy.** As our world becomes more interconnected, is there any hope of privacy? Is it even desirable?
- **Intellectual property.** Should software be patentable? What is the role of open source software?

## 3. Key processes: computational thinking

---

A “key process” is something that a student of Computing should be able to *do*; Section 4 deals with what a student should *know*.

In Computing, the key processes can be unified by a single theme: *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 computing 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<sup>5</sup>. 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 problem, 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-*

---

<sup>5</sup> 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).

*known 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 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 a 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 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 (its size and colour, say), 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

Computing is more than programming, but programming is an absolutely central process for Computing. In an educational context, programming encourages creativity, logical thought, and precision, and helps foster the personal, learning and problem-solving 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 student should have repeated opportunities to design, write, run, and debug, an executable program.** What an “executable program” means can range widely, depending on the level of the student 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++, Java, Pascal, PHP, Python, Visual Basic, and so on.
- Scripting languages, such as shell scripts, Flash ActionScript, or JavaScript.
- Spreadsheet formulae

Both interpreted and compiled languages are “executable”.

In every case the underlying concepts are more important than the particular programming language. Moreover, the ability to **understand and explain a program** is much more important than the ability to produce working but incomprehensible code.

Depending on level, students should be able to:

- Design and write programs that include
  - Sequencing: doing one step after another.
  - Choice (if-then-else): doing either one thing or another.
  - Iteration (loops): doing something repeatedly.
  - 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, students 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 computer system goes wrong, how can I fix it? Computers can be so opaque that fault-finding degenerates into a demoralising process of trying randomly generated “solutions” until something works. Programming gives students the opportunity to develop systematic debugging and testing skills, including:

- Reading the manual (of a program library, say)
- Describing to another student why the code should work.
- Manually executing code, perhaps using pencil and paper.
- Isolating or localising faults by adding tracing or profiling code to give more visibility on what is going on
- Adding error checking code to check internal consistency, such as “Here x should be greater than zero”.
- Writing and executing test cases.

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 students who specialise in the subject, but they are hard to motivate for very small scale projects, and they tend to obscure or dominate the other teaching goals.

## 4. Range and content: what a student should know

---

This section says what a student should know by the end of Key Stage 3 (age 14) and Key Stage 4 (age 16). It should not be read as a statement of how the subject should be taught; simply as a summary of what a student 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 student. Almost certainly not all of it will fit, and teachers will need to select material from the range offered here.

At Key Stage 4, much of the content identified for KS3 should be re-visited, but at greater depth.

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

### 4.1 Algorithms

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

KS3

- 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]
- What computers find hard. [in a general sense, rather than formally; things that scale badly; or that we don't know how to program well, like natural language understanding]
- The need for accuracy of both algorithm and data [difficulty of data verification; garbage in, garbage out]

KS4

- The choice of an algorithm may be influenced by the data structure and data values.
- Familiarity with several key algorithms [sorting and searching].
- Different algorithms may have very different performance characteristics. [For example, binary search is much faster than linear search as the size of the list increases. Knowledge of the  $O(n)$  notation is not required.]

### 4.2 Programs

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

KS3

- Sequence
- Iteration

- Choice, including
  - Relational operators
  - Simple use of AND and OR and NOT
  - How relational operators are affected by negation [e.g. NOT (a>b) = a≤b]
- Variables and assignment
- Using a simple linear data structure, such as an array or a list
- Abstraction via functions and procedures (definition and call), including
  - Functions and procedures with parameters
  - Programs with more than one call of a single procedure
- Finding and correcting logical errors

#### KS4

- Manipulation of logical expressions, e.g. truth tables, DeMorgan's rules, and boolean valued variables
- Two-dimensional arrays (and higher).
- Use of nested constructs: a loop body can contain a loop, or a conditional, or a procedure call, etc. Similarly for conditionals
- Procedures that call procedures, to multiple levels. [Building one abstraction on top of another.]
- Programs that read and write persistent data in files.

### 4.3 Data

A student should understand how computers represent data:

#### KS3

- Introduction to binary representation
- 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 [eg 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 see 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].
- Introduction to relational databases; to include the concepts of creating relationships and extrapolation of data with a simple 2 or 3 table relational example. The benefits of relational versus non relational database development

#### KS4

- 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
- An introduction to SQL queries to enable data to be retrieved from databases. The SQL operators could include (but is not limited to) SELECT, arithmetic operators; Boolean operators; special operators such as Like, Between and Null; aggregate Functions; use of Count and Distinct; ordering and grouping; restricting groups using Having; and simple Joins.

## 4.4 Computers

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

#### KS3

- 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 (eg hard disk, main memory), input/output (eg 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

#### KS4

- 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 are running, and provides the filing system) and virtual machines.

## 4.5 Communication and the Internet

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

KS3

- 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]
  - HTML, style sheets
  - 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.

KS4

- 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 students

Computing offers an enormous range of more advanced topics, all of which are accessible to a motivated KS4 student. The list here should not be regarded as exhaustive.

Algorithms

- Modular arithmetic

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

#### Programming

- Recursion
- Pointers and data structures
- Assembler
- Other language types and constructs: object oriented and functional languages

#### 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 computing

---

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 the GCSE in Computing offering a clear benchmark. The range of levels within which the great majority of pupils are expected to work and the expected attainment for the majority of pupils at the end of the key stage 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 orally describe 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 robots.

Pupils recognise patterns 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 refine and improve their instructions.

Pupils present data in a systematic way.

### Level 4

Pupils analyse and represent symbolically a sequence of events, for example, 'make a cup of tea'.

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 iteration.

Pupils can 'think through' an algorithm and produce an output (dry run).

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 their algorithm.

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 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 error-free 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 both 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 appropriate 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 good 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 both problems.

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

Pupils design 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, for example, normalisation.

## **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 which fits most aspects of both problems

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

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 which 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.

## 6. Glossary

---

**Abstraction:** Abstraction is dividing a problem/system/process into sub-problems/systems/processes. In Computing a problem is often broken up into smaller components before the solution is tackled; i.e. *decomposed*. Each sub-part can be represented by using a variety of techniques; such as structure charts.

**Algorithm:** A precise rule (or set of rules) specifying how to solve a problem or carry out a process.

**Assignment:** The process of assigning a value to a variable; understanding the meaning of  $A:=B$ .

**Data Modelling:** The process of identifying entities and the relationships between them using diagrams.

**Debugging:** A systematic approach to finding faults in software.

**Design:** Design is the description interfaces, systems and processes to produce a solution/product for each of the sub-problems/systems/processes.

**Expressions:** A combination of values, variables, operators and functions which are interpreted and then produce another value.

**Implementation:** The internal details of how a particular software or hardware component works. For example, the implementation of a procedure would be the code of the procedure body.

**Interface:** A description of the externally-visible ways in which a user of a software or hardware component can interact with that component. For example, the interface of a procedure would consist of a description of the parameters, their types, and their purpose.

**Iteration:** Programming statements which allow you to repeat a segment of code. For example, loops, such as the DO...WHILE or REPEAT...UNTIL.

**Models:** Computer and paper-based representations of systems which enable exploration and asking 'what if?'

**Pseudo code:** An informal high level description of a computer programming algorithm that is intended for human reading. Usually part of the design process.

**Selection:** Programming statements that enable the program to follow a different path if certain conditions are true. For example, the use of IF statements and CASE...SELECT statements.

**Specification:** A description of what the proposed system/solution should do described in structured English and diagrams; not necessarily how it should be done.