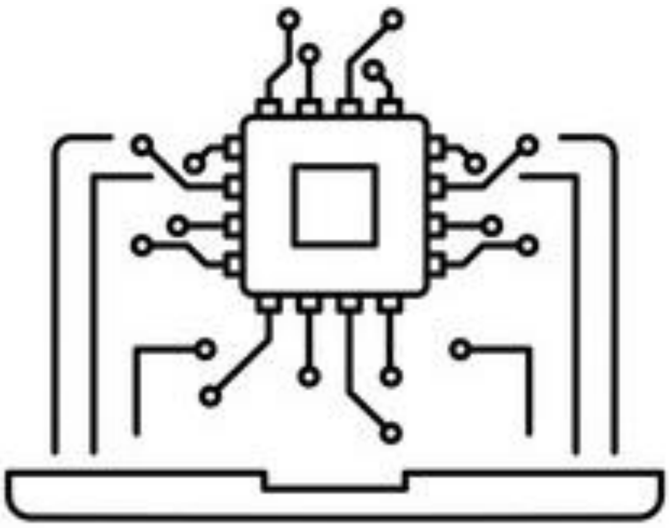
Beckfoot
Sixth Form

A-Level Computer Science Handbook



Contents

Course details

Qualification

AQA A-Level Computer Science

Specification

AQA | AS and A-level | Computer Science | Subject content – A-level

Entry Requirements

- Grade 5 in GCSE English Language or GCSE English Literature
- Grade 6 in GCSE Maths
- If studying Computer Science: Grade 6
- If not studying Computer Science: Grade 7 in Maths

Calendar

Half-term	Year 12	Year 13
J2J 1		Introduction to OOP and NEA introduction. Classes, inheritance, overloading, constructors, overriding.
2	Fundamentals of Programming with VB.Net. Floating point binary. Logic gates & Boolean Algebra.	OOP Programming Techniques using previous skeleton code (Food Magnate or Rabbits & Foxes) and NEA project work. Data structures: Queue, Stack, List.
3	Fundamentals of Algorithms and Networks. Data Rep: Images, sound, compression and encryption.	Data structures: Trees and traversal techniques. Depth first & Breadth first. Paper 1 Pre-Release Material and NEA
4	Theory of Computation and Architecture: CPU F-E cycle. Assembly Language programming. Networking. Finite State and Turing machines.	Paper 1 Pre-release material. Fundamentals of Databases and NEA.
5	Paper 1 Skeleton Code practice using past examination material.	Big Data, Functional Programming including partial function application and composition. Completion of NEA.
6	Problem Solving and Uses of Computing. The internet: TCP/IP protocols. Subnet masking, NAT, DHCP, Port Forwarding.	Revision Exam Prep

Contact details

Head of Computer Science: Miss Sameera Iqbal siqbal01@beckfoot.org

A-Level Computer Science Teacher: Mr Neil Kendall nkendall01@beckfoot.org

Organisation

You are expected to maintain a well-organised folder, which will be checked by a teacher once per half-term. You must use file dividers, and should contain the following sections:

1. Course documents
    a. Specification
    b. Teaching plan
2. Independent learning
    a. 5 hours in... guidance and templates
    b. A-Level Computer Science Revision booklet
3. Topic notes (This may be kept in an exercise book)
    a. Paper 1
    b. Paper 2

Notes from each lesson should have a title and date, and placed into your files so that you have a useful set of notes from which you can revise.

Equipment

You must bring the following to all lessons:

- Black pens
- Green pen
- Mini-whiteboard pen
- Pencil
- Calculator

Assessment

External assessments

A-Level Computer Science is a linear course, and you will sit all external exams at the end of Year 13.

These take the form of two papers and one NEA, as shown below:

Paper 1 (On-screen exam: 2 hours 30 minutes – 40% of A – Level)
What's assessed:
This paper tests a student's ability to program, as well as their theoretical knowledge of Computer Science from subject content 10-13 above and the skills required from section 22 above.
Questions
Students answer a series of short questions and write/adapt/extend programs in an Electronic Answer Document provided by us. We will issue Preliminary Material, a Skeleton Program and, where appropriate, test data, for use in the exam.

Paper 2 (Written exam: 2 hours 30 minutes – 40% of A – Level)
What's assessed:
This paper tests a student's ability to answer questions from subject content 14-21 above.
Questions
Compulsory short-answer and extended-answer questions.

Non Exam Assessment (75 Marks: 20% of A – Level)
What's assessed:
The non-exam assessment assesses student's ability to use the knowledge and skills gained through the course to solve or investigate a practical problem. Students will be expected to follow a systematic approach to problem solving, as shown in section 22 above.

Grade boundaries

Below is an indication of the highest-grade boundaries that have been used in AS and A-Level Computer Science exams, up to 2023. These are indicative only – actual grade boundaries used for in-class assessments may vary.

Grade	AS Computer Science (Year 12)	A-Level Computer Science (Year 13)
A*		85 %
A	74 %	72 %
B	63 %	59 %
C	52 %	47 %
D	41 %	34 %
E	31 %	21 %

Internal assessments

You will have one set of mock exams in Year 12, and two sets of mock exams in Year 13.

These mocks will cover all topics that have been covered up to that point in time.

Is addition to this there will be a formal assessment each half term based on knowledge/topics covered up until that point.

5 hours in... Computer Science

Research shows that the most successful students (i.e. those that make the most progress and get the highest grades) are doing between 20 and 25 hours of independent study per week by the end of Year 13. That may seem a lot, but it's something that you would build up to over the course of your A-Levels. In Year 12, we're talking something more like 15 hours per week. This equates to roughly 5 hours of independent study per A-Level per subject

Remember that your independent study is divided into three types – Consolidation, Reactive and Proactive

Consolidation

The evening following a Computer Science lesson, you should spend 12-15 minutes (24-30 minutes for a double) rereading your notes, writing the summary section at the bottom of your notes and making relevant flashcards e.g. for equations, definitions, facts you need to recall etc

Reactive

This is your 'homework'. If you find this takes more than 1 hour, that's fine, you can take this from the proactive phase (not from the consolidation phase though). Equally, if you find you finish your reactive work quickly, spend more time on your proactive work. Remember you should 'react' to all programming challenges by asking "What can I do to improve the program …"

Proactive

This is the section that will broaden and deepen your overall understanding of the subject you are studying. It will not necessarily involve work that has been set by your teacher, but instead it is about you doing the extra practice questions, reading articles, watching videos, TED talks etc. Remember your Year 13 project (worth 20% of the final grade) is essentially proactive work. In Computer Science, this might contain some of the following:

- Complete a set of practice past paper questions – available on the AQA website (1 hour)
- Use websites to complete and add to class notes (30 minutes)
- Use the specification checklist to evaluate your understanding (10 mins)
- Answer questions in your Workbooks (30 mins)
- Programming, programming & more programming …
- Give yourself a mini (or maxi) programming challenge (15 mins to 15 hours)
- Research possible NEA project ideas
- Practice exam style questions from your Computer Science textbooks (30 mins)
- "Read, Cover, Write and Check" sections of Knowledge organisers (30 mins)
- Complete a section of questions on Isaac Computer Science (30 minutes)
- Watch some videos on Computerphile (30 mins)

Useful links

- AQA Specification https://tinyurl.com/yxxsk7k2
- AQA Past papers and mark schemes https://tinyurl.com/vdua6az
- Computer Science Revision Notes (Physics & Maths Tutor) https://tinyurl.com/qrfc8yl
- Isaac Computer Science https://tinyurl.com/ss34258
- Assembly Language Simulator https://tinyurl.com/wpcaoof
- Unit 1 Skeleton Program Predictions https://tinyurl.com/vrknq96
- Textbook Answers & Resources https://tinyurl.com/u52rqpr
- SQL Tutorial https://www.w3schools.com/sql/
- Computerphile Videos https://tinyurl.com/h3a8slm

Supercurricular - Computer Science

Read

- New Turing Omnibus - https://www.amazon.co.uk/New-Turing-Omnibus-K-Dewdney/dp/0805071660
- The Emperor's New Mind - https://www.amazon.co.uk/Emperors-New-Mind-Concerning-Computers/dp/0192861980
- The Code Book - https://www.amazon.co.uk/Code-Book-Secret-History-Code-breaking/dp/1857028899/

Watch

- The Imitation Game
- The Social Network
- AI Artificial Intelligence
- Ethics in the age of AI https://www3.nhk.or.jp/nhkworld/en/tv/directtalk/20200319/2058617/

Listen

- University of Oxford Computer Science https://podcasts.ox.ac.uk/series/computer-science

Compete

- Bebras https://www.bebras.uk/
- OUCC - Round Two

Online

- Computerphile https://www.youtube.com/user/computerphile/videos
- Project Euler https://projecteuler.net/
- Isaac Computer Science https://isaaccomputerscience.org/
- https://www3.nhk.or.jp/nhkworld/en/tv/directtalk/20200319/2058617/
- https://www.olympiad.org.uk/
- https://www.cipherchallenge.org/
- https://github.com/
- https://microbit.org/
- https://www.chch.ox.ac.uk/admissions/python-challenges-page
- https://www.raspberrypi.org/

Write like a Computer Scientist

It is important that you can explain yourself clearly in your written work. Writing like a Computer Scientist will ensure you are able to get your points across in an accurate and concise manner.

- Follow what the command word tells you.
- Concise and to the point.
- Sequence in a logical order.
- Use specific and key terminology

Command words

Calculate	Work out the value of something.
Compare	Identify similarities and/or differences.
Define	Specify meaning.
Describe	Set out characteristics.
Discuss	Set out characteristics.
Draw	Produce a diagram.
Explain	Set out purposes or reasons.
State	Express in clear terms.
Suggest	Present a possible case/solution.

We will support the following programming languages:

- C#
- Java
- Python*
- VB.Net.

*Python 2 is no longer a supported language.

Schools and colleges will be asked to indicate their programming language preference at the start of the study of the specification.

4.1 Fundamentals of programming

4.1.1 Programming

4.1.1.1 Data types

Content	Additional information
Understand the concept of a data type.	
Understand and use the following appropriately: • integer • real/float • Boolean • character • string • date/time • pointer/reference • records (or equivalent) • arrays (or equivalent).	Variables declared as a pointer or reference data type are used as stores for memory addresses of objects created at runtime, ie dynamically. Not all languages support explicit pointer types, but students should have an opportunity to understand this data type.
Define and use user-defined data types based on language-defined (built-in) data types.	

4.1.1.2 Programming concepts

Content	Additional information
Use, understand and know how the following statement types can be combined in programs: • variable declaration • constant declaration • assignment • iteration • selection • subroutine (procedure/function).	The three combining principles (sequence, iteration/repetition and selection/choice) are basic to all imperative programming languages.
Use definite and indefinite iteration, including indefinite iteration with the condition(s) at the start or the end of the iterative structure. A theoretical understanding of condition(s) at either end of an iterative structure is required, regardless of whether they are supported by the language being used.	
Use nested selection and nested iteration structures.	
Use meaningful identifier names and know why it is important to use them.	

4.1.1.3 Arithmetic operations in a programming language

Content	Additional information
Be familiar with and be able to use: • addition • subtraction • multiplication • real/float division • integer division, including remainders • exponentiation • rounding • truncation.	

4.1.1.4 Relational operations in a programming language

Content	Additional information
Be familiar with and be able to use: • equal to • not equal to • less than • greater than • less than or equal to • greater than or equal to.	

4.1.1.5 Boolean operations in a programming language

Content	Additional information
Be familiar with and be able to use: • NOT • AND • OR • XOR.	

4.1.1.6 Constants and variables in a programming language

Content	Additional information
Be able to explain the differences between a variable and a constant.	
Be able to explain the advantages of using named constants.	

4.1.1.7 String-handling operations in a programming language

Content	Additional information
Be familiar with and be able to use: • length • position • substring • concatenation • character → character code • character code → character • string conversion operations.	Expected string conversion operations: • string to integer • string to float • integer to string • float to string • date/time to string • string to date/time.

4.1.1.8 Random number generation in a programming language

Content	Additional information
Be familiar with, and be able to use, random number generation.	

4.1.1.9 Exception handling

Content	Additional information
Be familiar with the concept of exception handling.	
Know how to use exception handling in a programming language with which students are familiar.	

4.1.1.10 Subroutines (procedures/functions)

Content	Additional information
Be familiar with subroutines and their uses.	
Know that a subroutine is a named 'out of line' block of code that may be executed (called) by simply writing its name in a program statement.	
Be able to explain the advantages of using subroutines in programs.	

4.1.1.11 Parameters of subroutines

Content	Additional information
Be able to describe the use of parameters to pass data within programs.	
Be able to use subroutines with interfaces.	

4.1.1.12 Returning a value/values from a subroutine

Content	Additional information
Be able to use subroutines that return values to the calling routine.	

4.1.1.13 Local variables in subroutines

Content	Additional information
Know that subroutines may declare their own variables, called local variables, and that local variables: • exist only while the subroutine is executing • are accessible only within the subroutine.	
Be able to use local variables and explain why it is good practice to do so.	

4.1.1.14 Global variables in a programming language

Content	Additional information
Be able to contrast local variables with global variables.	

4.1.1.15 Role of stack frames in subroutine calls

Content	Additional information
Be able to explain how a stack frame is used with subroutine calls to store: • return addresses • parameters • local variables.	

4.1.1.16 Recursive techniques

Content	Additional information
Be familiar with the use of recursive techniques in programming languages (general and base cases and the mechanism for implementation).	
Be able to solve simple problems using recursion.	

4.1.2 Programming paradigms

4.1.2.1 Programming paradigms

Content	Additional information
Understand the characteristics of the procedural- and object-oriented programming paradigms, and have experience of programming in each.	

4.1.2.2 Procedural-oriented programming

Content	Additional information
Understand the structured approach to program design and construction.	
Be able to construct and use hierarchy charts when designing programs.	
Be able to explain the advantages of the structured approach.	

4.1.2.3 Object-oriented programming

Content	Additional information
Be familiar with the concepts of: • class • object • instantiation • encapsulation • inheritance • aggregation • composition • polymorphism • overriding.	Students should know that: • a class defines methods and property/ attribute fields that capture the common behaviours and characteristics of objects • objects based on a class are created using a constructor, implicit or explicit, and a reference to the object assigned to a reference variable of the class type • in the Unified Modelling Language (UML) composition is represented by a black diamond line and aggregation by a white diamond line.
Know why the object-oriented paradigm is used.	
Be aware of the following object-oriented design principles: • encapsulate what varies • favour composition over inheritance • program to interfaces, not implementation.	Students would benefit from practical experience of programming to an interface, but will not be explicitly tested on programming to interfaces or be required to program to interfaces in any practical exam.

Content	Additional information
Be able to write object-oriented programs.	Practical experience of coding for user-defined classes involving: • abstract, virtual and static methods • inheritance • aggregation • polymorphism • public, private and protected specifiers.
Be able to draw and interpret class diagrams.	Class diagrams involving single inheritance, composition (black diamond line), aggregation (white diamond line), public (+), private (-) and protected (#) specifiers.

4.2 Fundamentals of data structures

4.2.1 Data structures and abstract data types

4.2.1.1 Data structures

Content	Additional information
Be familiar with the concept of data structures.	It may be helpful to set the concept of a data structure in various contexts that students may already be familiar with. It may also be helpful to suggest/demonstrate how data structures could be used in a practical setting.

4.2.1.2 Single- and multi-dimensional arrays (or equivalent)

Content	Additional information
Use arrays (or equivalent) in the design of solutions to simple problems.	A one-dimensional array is a useful way of representing a vector. A two-dimensional array is a useful way of representing a matrix. More generally, an *n*-dimensional array is a set of elements with the same data type that are indexed by a tuple of *n* integers, where a tuple is an ordered list of elements.

4.2.1.3 Fields, records and files

Content	Additional information
Be able to read/write from/to a text file.	

Content	Additional information
Be able to read/write data from/to a binary (non-text) file.	

4.2.1.4 Abstract data types/data structures

Content	Additional information
Be familiar with the concept and uses of a: • queue • stack • graph • tree • hash table • dictionary • vector.	Be able to use these abstract data types and their equivalent data structures in simple contexts. Students should also be familiar with methods for representing them when a programming language does not support these structures as built-in types.
Be able to distinguish between static and dynamic structures and compare their uses, as well as explaining the advantages and disadvantages of each.	
Describe the creation and maintenance of data within: • queues (linear, circular, priority) • stacks • hash tables.	

4.2.2 Queues

4.2.2.1 Queues

Content	Additional information
Be able to describe and apply the following to linear queues, circular queues and priority queues: • add an item • remove an item • test for an empty queue • test for a full queue.	

4.2.3 Stacks

4.2.3.1 Stacks

Content	Additional information
Be able to describe and apply the following operations: • push • pop • peek or top • test for empty stack • test for stack full.	Peek or top returns the value of the top element without removing it.

4.2.4 Graphs

4.2.4.1 Graphs

Content	Additional information
Be aware of a graph as a data structure used to represent more complex relationships.	
Be familiar with typical uses for graphs.	
Be able to explain the terms: • graph • weighted graph • vertex/node • edge/arc • undirected graph • directed graph.	
Know how an adjacency matrix and an adjacency list may be used to represent a graph.	
Be able to compare the use of adjacency matrices and adjacency lists.	

4.2.5 Trees

4.2.5.1 Trees (including binary trees)

Content	Additional information
Know that a tree is a connected, undirected graph with no cycles.	Note that a tree does not have to have a root.

Content	Additional information
Know that a rooted tree is a tree in which one vertex has been designated as the root. A rooted tree has parent-child relationships between nodes. The root is the only node with no parent and all other nodes are descendants of the root.	
Know that a binary tree is a rooted tree in which each node has at most two children.	A common application of a binary tree is as a binary search tree.
Be familiar with typical uses for rooted trees.	

4.2.6 Hash tables

4.2.6.1 Hash tables

Content	Additional information
Be familiar with the concept of a hash table and its uses.	A hash table is a data structure that creates a mapping between keys and values.
Be able to apply simple hashing algorithms.	
Know what is meant by a collision and how collisions are handled using rehashing.	A collision occurs when two key values compute the same hash.

4.2.7 Dictionaries

4.2.7.1 Dictionaries

Content	Additional information
Be familiar with the concept of a dictionary.	A collection of key-value pairs in which the value is accessed via the associated key.
Be familiar with simple applications of dictionaries, for example information retrieval, and have experience of using a dictionary data structure in a programming language.	Information retrieval: For example, the document 'The green, green grass grows' would be represented by the dictionary: {‘grass’ : 1, ‘green’ : 2, ‘grows’ : 1, ‘the’ : 1} ignoring letter case.

4.2.8 Vectors

4.2.8.1 Vectors

Content	Additional information
Be familiar with the concept of a vector and the following notations for specifying a vector: • [2.0, 3.14159, -1.0, 2.718281828] • 4-vector over ℝ written as $\mathbb{R}^4$ • function interpretation • $0 \mapsto 2.0$ • $1 \mapsto 3.14159$ • $2 \mapsto -1.0$ • $3 \mapsto 2.718281828$ • $\mapsto$ means maps to That all the entries must be drawn from the same field, eg ℝ.	A vector can be represented as a list of numbers, as a function and as a way of representing a geometric point in space. A dictionary is a useful way of representing a vector if a vector is viewed as a function. $f : S \rightarrow \mathbb{R}$ the set S = {0,1,2,3} and the co-domain, ℝ, the set of Reals For example, in Python the 4-vector example could be represented as a dictionary as follows: {0:2.0, 1:3.14159, 2:-1.0, 3:2.718281828}
Dictionary representation of a vector.	See above.
List representation of a vector.	For example, in Python, a 2-vector over ℝ would be written as [2.0,3.0].
1-D array representation of a vector.	For example in VB.Net, a 4-vector over ℝ would be written as *Dim example(3) As Single*.
Visualising a vector as an arrow.	For example a 2-vector [2.0, 3.0] over ℝ can be represented by an arrow with its tail at the origin and its head at (2.0, 3.0).
Vector addition and scalar-vector multiplication.	Know that vector addition achieves translation and scalar-vector multiplication achieves scaling.
Convex combination of two vectors, u and v.	Is an expression of the form $\alpha u + \beta v$ where $\alpha, \beta \geq 0$ and $\alpha + \beta = 1$
Dot or scalar product of two vectors.	The dot product of two vectors, *u* and *v*, $u = [u_1, \ldots, u_n]$ and $v = [v_1, \ldots, v_n]$ is $u \cdot v = u_1v_1 + u_2v_2 + \ldots + u_nv_n$
Applications of dot product.	Finding the angle between two vectors.

4.3 Fundamentals of algorithms

4.3.1 Graph-traversal

4.3.1.1 Simple graph-traversal algorithms

Content	Additional information
Be able to trace breadth-first and depth-first search algorithms and describe typical applications of both.	Breadth-first: shortest path for an unweighted graph. Depth-first: Navigating a maze.

4.3.2 Tree-traversal

4.3.2.1 Simple tree-traversal algorithms

Content	Additional information
Be able to trace the tree-traversal algorithms: • pre-order • post-order • in-order.	
Be able to describe uses of tree-traversal algorithms.	Pre-Order: copying a tree. In-Order: binary search tree, outputting the contents of a binary search tree in ascending order. Post-Order: Infix to RPN (Reverse Polish Notation) conversions, producing a postfix expression from an expression tree, emptying a tree.

4.3.3 Reverse Polish

4.3.3.1 Reverse Polish – infix transformations

Content	Additional information
Be able to convert simple expressions in infix form to Reverse Polish notation (RPN) form and vice versa. Be aware of why and where it is used.	Eliminates need for brackets in sub- expressions. Expressions in a form suitable for evaluation using a stack. Used in interpreters based on a stack for example Postscript and bytecode.

4.3.4 Searching algorithms

4.3.4.1 Linear search

Content	Additional information
Know and be able to trace and analyse the complexity of the linear search algorithm.	Time complexity is O(n).

4.3.4.2 Binary search

Content	Additional information
Know and be able to trace and analyse the time complexity of the binary search algorithm.	Time complexity is O(log n).

4.3.4.3 Binary tree search

Content	Additional information
Be able to trace and analyse the time complexity of the binary tree search algorithm.	Time complexity is O(log n).

4.3.5 Sorting algorithms

4.3.5.1 Bubble sort

Content	Additional information
Know and be able to trace and analyse the time complexity of the bubble sort algorithm.	This is included as an example of a particularly inefficient sorting algorithm, time-wise. Time complexity is O(n^2).

4.3.5.2 Merge sort

Content	Additional information
Be able to trace and analyse the time complexity of the merge sort algorithm.	The 'merge' sort is an example of 'Divide and Conquer' approach to problem solving. Time complexity is O($n$log n).

4.3.6 Optimisation algorithms

4.3.6.1 Dijkstra’s shortest path algorithm

Content	Additional information
Understand and be able to trace Dijkstra’s shortest path algorithm. Be aware of applications of shortest path algorithm.	Students will not be expected to recall the steps in Dijkstra's shortest path algorithm.

4.4 Theory of computation

4.4.1 Abstraction and automation

4.4.1.1 Problem-solving

Content	Additional information
Be able to develop solutions to simple logic problems.	
Be able to check solutions to simple logic problems.	

4.4.1.2 Following and writing algorithms

Content	Additional information
Understand the term algorithm.	A sequence of steps that can be followed to complete a task and that always terminates.
Be able to express the solution to a simple problem as an algorithm using pseudo-code, with the standard constructs: • sequence • assignment • selection • iteration.	
Be able to hand-trace algorithms.	
Be able to convert an algorithm from pseudo- code into high level language program code.	

Content	Additional information
Be able to articulate how a program works, arguing for its correctness and its efficiency using logical reasoning, test data and user feedback.	

4.4.1.3 Abstraction

Content	Additional information
Be familiar with the concept of abstraction as used in computations and know that: • representational abstraction is a representation arrived at by removing unnecessary details • abstraction by generalisation or categorisation is a grouping by common characteristics to arrive at a hierarchical relationship of the 'is a kind of' type.	

4.4.1.4 Information hiding

Content	Additional information
Be familiar with the process of hiding all details of an object that do not contribute to its essential characteristics.	

4.4.1.5 Procedural abstraction

Content	Additional information
Know that procedural abstraction represents a computational method.	The result of abstracting away the actual values used in any particular computation is a computational pattern or computational method - a procedure.

4.4.1.6 Functional abstraction

Content	Additional information
Know that for functional abstraction the particular computation method is hidden.	The result of a procedural abstraction is a procedure, not a function. To get a function requires yet another abstraction, which disregards the particular computation method. This is functional abstraction.

4.4.1.7 Data abstraction

Content	Additional information
Know that details of how data are actually represented are hidden, allowing new kinds of data objects to be constructed from previously defined types of data objects.	Data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed. For example, a stack could be implemented as an array and a pointer for top of stack.

4.4.1.8 Problem abstraction/reduction

Content	Additional information
Know that details are removed until the problem is represented in a way that is possible to solve, because the problem reduces to one that has already been solved.	

4.4.1.9 Decomposition

Content	Additional information
Know that procedural decomposition means breaking a problem into a number of sub-problems, so that each sub-problem accomplishes an identifiable task, which might itself be further subdivided.	

4.4.1.10 Composition

Content	Additional information
Know how to build a composition abstraction by combining procedures to form compound procedures.	
Know how to build data abstractions by combining data objects to form compound data, for example tree data structure.	

4.4.1.11 Automation

Content	Additional information
Understand that automation requires putting models (abstraction of real world objects/ phenomena) into action to solve problems. This is achieved by: • creating algorithms • implementing the algorithms in program code (instructions) • implementing the models in data structures • executing the code.	Computer science is about building clean abstract models (abstractions) of messy, noisy, real world objects or phenomena. Computer scientists have to choose what to include in models and what to discard, to determine the minimum amount of detail necessary to model in order to solve a given problem to the required degree of accuracy. Computer science deals with putting the models into action to solve problems. This involves creating algorithms for performing actions on, and with, the data that has been modelled.

4.4.2 Regular languages

4.4.2.1 Finite state machines (FSMs) with and without output

Content	Additional information
Be able to draw and interpret simple state transition diagrams and state transition tables for FSMs with no output and with output (Mealy machines only).	

4.4.2.2 Maths for regular expressions

Content	Additional information
Be familiar with the concept of a set and the following notations for specifying a set: A = {1, 2, 3, 4, 5 } or set comprehension: $A = \{x \mid x \in \mathbb{N} \land x \geq 1\}$ where A is the set consisting of those objects *x* such that $x \in \mathbb{N}$ and $x \geq 1$ is true. Know that the empty set, {}, is the set with no elements. Know that an alternative symbol for the empty set is Ø.	A set is an unordered collection of values in which each value occurs at most once. Several languages support set construction. In Python, for example, use of curly braces constructs a set: {1, 2, 3 }. | means such that. $x \in \mathbb{N}$ means that x is a member of the set $\mathbb{N}$ consisting of the natural numbers, ie {0, 1, 2, 3, 4, ... }. The symbol $\land$ means AND. The term $\land$ x > = 1 means AND x is greater than or equal to 1. In Python, {2 ∗ x for x in {1, 2, 3 }} constructs {2, 4, 6 }. This is said to be a set comprehension over the set {1, 2, 3 } .
Be familiar with the compact representation of a set, for example, the set $\{0^n1^n \mid n \geq 1\}$. This set contains all strings with an equal number of 0 s and 1s.	For example, $\{0^n1^n \mid n \geq 1\}$ = {01, 0011, 000111, 00001111, ... }
Be familiar with the concept of: • finite sets • infinite sets • countably infinite sets • cardinality of a finite set • Cartesian product of sets.	A finite set is one whose elements can be counted off by natural numbers up to a particular number, for example as: 1st element, 2nd element, ..., 20th (and final) element. The set of natural numbers, $\mathbb{N}$ and the set of real numbers, $\mathbb{R}$ are examples of infinite sets. A countably infinite set is one that can be counted off by the natural numbers. The set of real numbers is not countable. The cardinality of a finite set is the number of elements in a set. Cartesian product of two sets, X and Y, written X x Y and read 'X cross Y', is the set of all ordered pairs (*a, b*) where *a* is a member of A and *b* is a member of B.

Content	Additional information
Be familiar with the meaning of the term: • subset • proper subset • countable set.	{0, 1 , 2 } ⊂ ℕ where ⊂ means proper subset of, that is ℕ contains everything in {0, 1, 2 } but there is at least one element in ℕ that is not in {0, 1, 2 }. {0, 1 , 2 } ⊆ {0, 1, 2, 3 } where ⊆ means subset of. ⊆ includes both ⊂ and =, for example {0, 1, 2, 3 } ⊆ {0, 1, 2, 3 } is also true, because {0, 1, 2, 3 } = {0, 1, 2, 3 }. A countable set is a set with the same cardinality (number of elements) as some subset of natural numbers.
Be familiar with the set operations: • membership • union • intersection • difference.	The set difference A\B (or alternatively A-B) is defined by A\B = {x : x ∈ A and x ∉ B}

4.4.2.3 Regular expressions

Content	Additional information
Know that a regular expression is simply a way of describing a set and that regular expressions allow particular types of languages to be described in a convenient shorthand notation.	For example, the regular expression *a*(*a*|*b*)* generates the set of strings {*a, aa, ab, aaa, aab, aba*, ...}.
Be able to form and use simple regular expressions for string manipulation and matching.	Students should be familiar with the metacharacters: • * (0 or more repetitions) • + (1 or more repetitions) • ? (0 or 1 repetitions, ie optional) • | (alternation, ie or) • ( ) to group regular expressions. Any other metacharacters used in an exam question will be explained as part of the question.
Be able to describe the relationship between regular expressions and FSMs.	Regular expressions and FSMs are equivalent ways of defining a regular language.
Be able to write a regular expression to recognise the same language as a given FSM and vice versa.	A student's ability to write very simple regular expressions and FSMs will be assessed.

4.4.2.4 Regular language

Content	Additional information
Know that a language is called regular if it can be represented by a regular expression.	Also, a regular language is any language that a FSM will accept.

4.4.3 Context-free languages

4.4.3.1 Backus-Naur Form (BNF)/syntax diagrams

Content	Additional information
Be able to check language syntax by referring to BNF or syntax diagrams and formulate simple production rules.	
Be able to explain why BNF can represent some languages that cannot be represented using regular expressions.	

4.4.4 Classification of algorithms

4.4.4.1 Comparing algorithms

Content	Additional information
Understand that algorithms can be compared by expressing their complexity as a function relative to the size of the problem. Understand that the size of the problem is the key issue.	
Understand that some algorithms are more efficient: • time-wise than other algorithms • space-wise than other algorithms.	Efficiently implementing automated abstractions means designing data models and algorithms to run quickly while taking up the minimal amount of resources such as memory.

4.4.4.2 Maths for understanding Big-0 notation

Content	Additional information
Be familiar with the mathematical concept of a function as a mapping from one set of values, the domain, to another set of values, drawn from the co-domain, for example $\mathbb{N} \rightarrow \mathbb{N}$.	

Content	Additional information
Be familiar with the concept of: • a linear function, for example $y = 2x$ • a polynomial function, for example $y = 2x^2$ • an exponential function, for example $y = 2^x$ • a logarithmic function, for example $y = \log_{10} x$.	
Be familiar with the notion of permutation of a set of objects or values, for example, the letters of a word and that the number of permutations of *n* distinct objects is *n* factorial (*n*!).	*n*! is the product of all positive integers less than or equal to *n*.

4.4.4.3 Order of complexity

Content	Additional information
Be familiar with Big-O notation to express time complexity and be able to apply it to cases where the running time requirements of the algorithm grow in: • constant time • logarithmic time • linear time • polynomial time • exponential time.	
Be able to derive the time complexity of an algorithm.	

4.4.4.4 Limits of computation

Content	Additional information
Be aware that algorithmic complexity and hardware impose limits on what can be computed.	

4.4.4.5 Classification of algorithmic problems

Content	Additional information
Know that algorithms may be classified as being either: • tractable - problems that have a polynomial (or less) time solution are called tractable problems. • intractable - problems that have no polynomial (or less) time solution are called intractable problems.	Heuristic methods are often used when tackling intractable problems.

4.4.4.6 Computable and non-computable problems

Content	Additional information
Be aware that some problems cannot be solved algorithmically.	

4.4.4.7 Halting problem

Content	Additional information
Describe the Halting problem (but not prove it), that is the unsolvable problem of determining whether any program will eventually stop if given particular input.	
Understand the significance of the Halting problem for computation.	The Halting problem demonstrates that there are some problems that cannot be solved by a computer.

4.4.5 A model of computation

4.4.5.1 Turing machine

Content	Additional information
Be familiar with the structure and use of Turing machines that perform simple computations.	

Content	Additional information
Know that a Turing machine can be viewed as a computer with a single fixed program, expressed using: • a finite set of states in a state transition diagram • a finite alphabet of symbols • an infinite tape with marked-off squares • a sensing read-write head that can travel along the tape, one square at a time. One of the states is called a start state and states that have no outgoing transitions are called halting states.	Exam questions will only be asked about Turing machines that have one tape that is infinite in one direction.
Understand the equivalence between a transition function and a state transition diagram.	
Be able to: • represent transition rules using a transition function • represent transition rules using a state transition diagram • hand-trace simple Turing machines.	
Be able to explain the importance of Turing machines and the Universal Turing machine to the subject of computation.	Turing machines provide a (general/formal) model of computation and provide a definition of what is computable.

4.5 Fundamentals of data representation

4.5.1 Number systems

4.5.1.1 Natural numbers

Content	Additional information
Be familiar with the concept of a natural number and the set ℕ of natural numbers (including zero).	$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$

4.5.1.2 Integer numbers

Content	Additional information
Be familiar with the concept of an integer and the set ℤ of integers.	ℤ = { ..., -3, -2, -1, 0, 1, 2, 3, ... }

4.5.1.3 Rational numbers

Content	Additional information
Be familiar with the concept of a rational number and the set ℚ of rational numbers, and that this set includes the integers.	ℚ is the set of numbers that can be written as fractions (ratios of integers). Since a number such as 7 can be written as 7/1, all integers are rational numbers.

4.5.1.4 Irrational numbers

Content	Additional information
Be familiar with the concept of an irrational number.	An irrational number is one that cannot be written as a fraction, for example $\sqrt{2}$.

4.5.1.5 Real numbers

Content	Additional information
Be familiar with the concept of a real number and the set ℝ of real numbers, which includes the natural numbers, the rational numbers and the irrational numbers.	ℝ is the set of all 'possible real world quantities'.

4.5.1.6 Ordinal numbers

Content	Additional information
Be familiar with the concept of ordinal numbers and their use to describe the numerical positions of objects.	When objects are placed in order, ordinal numbers are used to tell their position. For example, if we have a well-ordered set S = {'a', 'b', 'c', 'd'}, then 'a' is the 1st object, 'b' the 2nd, and so on.

4.5.1.7 Counting and measurement

Content	Additional information
Be familiar with the use of: • natural numbers for counting • real numbers for measurement.	

4.5.2 Number bases

4.5.2.1 Number base

Content	Additional information
Be familiar with the concept of a number base, in particular: • decimal (base 10) • binary (base 2) • hexadecimal (base 16).	Students should be familiar with expressing a number's base using a subscript as follows: Base 10: Number_{10}, eg 67_{10} Base 2: Number_2, eg 10011011_2 Base 16: Number_{16}, eg AE_{16}
Convert between decimal, binary and hexadecimal number bases.	
Be familiar with, and able to use, hexadecimal as a shorthand for binary and to understand why it is used in this way.	

4.5.3 Units of information

4.5.3.1 Bits and bytes

Content	Additional information
Know that: • the bit is the fundamental unit of information • a byte is a group of 8 bits.	A bit is either 0 or 1.
Know that the 2^n different values can be represented with *n* bits.	For example, 3 bits can be configured in $2^3 = 8$ different ways. 000, 001, 010, 011, 100, 101, 110, 111

4.5.3.2 Units

Content	Additional information
Know that quantities of bytes can be described using binary prefixes representing powers of 2 or using decimal prefixes representing powers of 10, eg one kibibyte is written as 1KiB = 2^{10} B and one kilobyte is written as 1 kB = 10^3 B. Know the names, symbols and corresponding powers of 2 for the binary prefixes: • kibi, Ki - 2^{10} • mebi, Mi - 2^{20} • gibi, Gi - 2^{30} • tebi, Ti - 2^{40} Know the names, symbols and corresponding powers of 10 for the decimal prefixes: • kilo, k - 10^3 • mega, M - 10^6 • giga, G - 10^9 • tera, T - 10^{12}	Historically the terms kilobyte, megabyte, etc have often been used when kibibyte, mebibyte, etc are meant.

4.5.4 Binary number system

4.5.4.1 Unsigned binary

Content	Additional information
Know the difference between unsigned binary and signed binary.	Students are expected to be able to convert between unsigned binary and decimal and vice versa.
Know that in unsigned binary the minimum and maximum values for a given number of bits, *n*, are 0 and 2^n -1 respectively.	

4.5.4.2 Unsigned binary arithmetic

Content	Additional information
Be able to: • add two unsigned binary integers • multiply two unsigned binary integers.	

4.5.4.3 Signed binary using two’s complement

Content	Additional information
Know that signed binary can be used to represent negative integers and that one possible coding scheme is two’s complement.	This is the only representation of negative integers that will be examined. Students are expected to be able to convert between signed binary and decimal and vice versa.
Know how to: • represent negative and positive integers in two’s complement • perform subtraction using two’s complement • calculate the range of a given number of bits, *n*.	

4.5.4.4 Numbers with a fractional part

Content	Additional information
Know how numbers with a fractional part can be represented in: • fixed point form in binary in a given number of bits • floating point form in binary in a given number of bits.	Students are not required to know the Institute of Electrical and Electronic Engineers (IEEE) standard, only to know, understand and be able to use a simplified floating representation consisting of mantissa + exponent.
Be able to convert for each representation from: • decimal to binary of a given number of bits • binary to decimal of a given number of bits.	Exam questions on floating point numbers will use a format in which both the mantissa and exponent are represented using two's complement.

4.5.4.5 Rounding errors

Content	Additional information
Know and be able to explain why both fixed point and floating point representation of decimal numbers may be inaccurate.	Use binary fractions. For a real number to be represented exactly by the binary number system, it must be capable of being represented by a binary fraction in the given number of bits. Some values cannot ever be represented exactly, for example 0.1_{10}.

4.5.4.6 Absolute and relative errors

Content	Additional information
Be able to calculate the absolute error of numerical data stored and processed in computer systems.	
Be able to calculate the relative error of numerical data stored and processed in computer systems.	
Compare absolute and relative errors for large and small magnitude numbers, and numbers close to one.	

4.5.4.7 Range and precision

Content	Additional information
Compare the advantages and disadvantages of fixed point and floating point forms in terms of range, precision and speed of calculation.	

4.5.4.8 Normalisation of floating point form

Content	Additional information
Know why floating point numbers are normalised and be able to normalise un-normalised floating point numbers with positive or negative mantissas.	

4.5.4.9 Underflow and overflow

Content	Additional information
Explain underflow and overflow and describe the circumstances in which they occur.	

4.5.5 Information coding systems

4.5.5.1 Character form of a decimal digit

Content	Additional information
Differentiate between the character code representation of a decimal digit and its pure binary representation.	

4.5.5.2 ASCII and Unicode

Content	Additional information
Describe ASCII and Unicode coding systems for coding character data and explain why Unicode was introduced.	

4.5.5.3 Error checking and correction

Content	Additional information
Describe and explain the use of: • parity bits • majority voting • checksums • check digits.	

4.5.6 Representing images, sound and other data

4.5.6.1 Bit patterns, images, sound and other data

Content	Additional information
Describe how bit patterns may represent other forms of data, including graphics and sound.	

4.5.6.2 Analogue and digital

Content	Additional information
Understand the difference between analogue and digital: • data • signals.	

4.5.6.3 Analogue/digital conversion

Content	Additional information
Describe the principles of operation of: • an analogue to digital converter (ADC) • a digital to analogue converter (DAC).	
Know that ADCs are used with analogue sensors.	

Content	Additional information
Know that the most common use for a DAC is to convert a digital audio signal to an analogue signal.	

4.5.6.4 Bitmapped graphics

Content	Additional information
Explain how bitmaps are represented.	
Explain the following for bitmaps: • resolution • colour depth • size in pixels.	The size of an image is also alternatively sometimes described as the resolution of an image. Size of an image in pixels is width of image in pixels x height of image in pixels. Resolution is expressed as number of dots per inch where a dot is a pixel. Colour depth = number of bits stored for each pixel.
Calculate storage requirements for bitmapped images and be aware that bitmap image files may also contain metadata.	Ignoring metadata, *storage requirements = size in pixels x colour depth* where size in pixels is *width in pixels* x *height in pixels*.
Be familiar with typical metadata.	eg width, height, colour depth.

4.5.6.5 Vector graphics

Content	Additional information
Explain how vector graphics represents images using lists of objects.	The properties of each geometric object/shape in the vector graphic image are stored as a list.
Give examples of typical properties of objects.	
Use vector graphic primitives to create a simple vector graphic.	

4.5.6.6 Vector graphics versus bitmapped graphics

Content	Additional information
Compare the vector graphics approach with the bitmapped graphics approach and understand the advantages and disadvantages of each.	
Be aware of appropriate uses of each approach.	

4.5.6.7 Digital representation of sound

Content	Additional information
Describe the digital representation of sound in terms of: • sample resolution • sampling rate and the Nyquist theorem.	
Calculate sound sample sizes in bytes.	

4.5.6.8 Musical Instrument Digital Interface (MIDI)

Content	Additional information
Describe the purpose of MIDI and the use of event messages in MIDI.	
Describe the advantages of using MIDI files for representing music.	

4.5.6.9 Data compression

Content	Additional information
Know why images and sound files are often compressed and that other files, such as text files, can also be compressed.	
Understand the difference between lossless and lossy compression and explain the advantages and disadvantages of each.	
Explain the principles behind the following techniques for lossless compression: • run length encoding (RLE) • dictionary-based methods.	

4.5.6.10 Encryption

Content	Additional information
Understand what is meant by encryption and be able to define it.	Students should be familiar with the terms cipher, plaintext and ciphertext. Caesar and Vernam ciphers are at opposite extremes. One offers perfect security, the other doesn't. Between these two types are ciphers that are computationally secure – see below. Students will be assessed on the two types. Ciphers other than Caesar may be used to assess students' understanding of the principles involved. These will be explained and be similar in terms of computational complexity.
Be familiar with Caesar cipher and be able to apply it to encrypt a plaintext message and decrypt a ciphertext. Be able to explain why it is easily cracked.	
Be familiar with Vernam cipher or one-time pad and be able to apply it to encrypt a plaintext message and decrypt a ciphertext. Explain why Vernam cipher is considered as a cypher with perfect security.	Since the key *k* is chosen uniformly at random, the ciphertext *c* is also distributed uniformly. The key *k* must be used once only. The key *k* is known as a one-time pad.
Compare Vernam cipher with ciphers that depend on computational security.	Vernam cipher is the only one to have been mathematically proved to be completely secure. The worth of all other ciphers ever devised is based on computational security. In theory, every cryptographic algorithm except for Vernam cipher can be broken, given enough ciphertext and time.

4.6 Fundamentals of computer systems

4.6.1 Hardware and software

4.6.1.1 Relationship between hardware and software

Content	Additional information
Understand the relationship between hardware and software and be able to define the terms: • hardware • software.	

4.6.1.2 Classification of software

Content	Additional information
Explain what is meant by: • system software • application software.	
Understand the need for, and attributes of, different types of software.	

4.6.1.3 System software

Content	Additional information
Understand the need for, and functions of the following system software: • operating systems (OSs) • utility programs • libraries • translators (compiler, assembler, interpreter).	

4.6.1.4 Role of an operating system (OS)

Content	Additional information
Understand that a role of the operating system is to hide the complexities of the hardware.	
Know that the OS handles resource management, managing hardware to allocate processors, memories and I/O devices among competing processes.	

4.6.2 Classification of programming languages

4.6.2.1 Classification of programming languages

Content	Additional information
Show awareness of the development of types of programming languages and their classification into low-and high-level languages.	

Content	Additional information
Know that low-level languages are considered to be: • machine-code • assembly language.	
Know that high-level languages include imperative high-level language.	
Describe machine-code language and assembly language.	
Understand the advantages and disadvantages of machine-code and assembly language programming compared with high-level language programming.	
Explain the term 'imperative high-level language' and its relationship to low-level languages.	

4.6.3 Types of program translator

4.6.3.1 Types of program translator

Content	Additional information
Understand the role of each of the following: • assembler • compiler • interpreter. Explain the differences between compilation and interpretation. Describe situations in which each would be appropriate.	
Explain why an intermediate language such as bytecode is produced as the final output by some compilers and how it is subsequently used.	
Understand the difference between source code and object (executable) code.	

4.6.4 Logic gates

4.6.4.1 Logic gates

Content	Additional information
Construct truth tables for the following logic gates: • NOT • AND • OR • XOR • NAND • NOR.	Students should know and be able to use ANSI/ IEEE standard 91-1984 Distinctive shape logic gate symbols for these logic gates.
Be familiar with drawing and interpreting logic gate circuit diagrams involving one or more of the above gates.	
Complete a truth table for a given logic gate circuit.	
Write a Boolean expression for a given logic gate circuit.	
Draw an equivalent logic gate circuit for a given Boolean expression.	
Recognise and trace the logic of the circuits of a half-adder and a full-adder.	
Construct the circuit for a half-adder.	
Be familiar with the use of the edge-triggered D-type flip-flop as a memory unit.	Knowledge of internal operation of this flip-flop is not required.

4.6.5 Boolean algebra

4.6.5.1 Using Boolean algebra

Content	Additional information
Be familiar with the use of Boolean identities and De Morgan's laws to manipulate and simplify Boolean expressions.	

4.7 Fundamentals of computer organisation and architecture

4.7.1 Internal hardware components of a computer

4.7.1.1 Internal hardware components of a computer

Content	Additional information
Have an understanding and knowledge of the basic internal components of a computer system.	Although exam questions about specific machines will not be asked, it might be useful to base this section on the machines used at the centre.
Understand the role of the following components and how they relate to each other: • processor • main memory • address bus • data bus • control bus • I/O controllers.	
Understand the need for, and means of, communication between components. In particular, understand the concept of a bus and how address, data and control buses are used.	
Be able to explain the difference between von Neumann and Harvard architectures and describe where each is typically used.	Embedded systems such as digital signal processing (DSP) systems use Harvard architecture processors extensively. Von Neumann architecture is used extensively in general purpose computing systems.
Understand the concept of addressable memory.	

4.7.2 The stored program concept

4.7.2.1 The meaning of the stored program concept

Content	Additional information
Be able to describe the stored program concept: machine code instructions stored in main memory are fetched and executed serially by a processor that performs arithmetic and logical operations.	

4.7.3 Structure and role of the processor and its components

4.7.3.1 The processor and its components

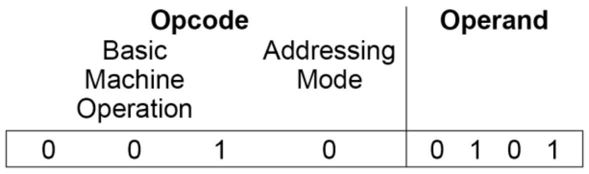
Content	Additional information
Explain the role and operation of a processor and its major components: • arithmetic logic unit • control unit • clock • general-purpose registers • dedicated registers, including: • program counter • current instruction register • memory address register • memory buffer register • status register.	

4.7.3.2 The Fetch-Execute cycle and the role of registers within it

Content	Additional information
Explain how the Fetch-Execute cycle is used to execute machine code programs including the stages in the cycle (fetch, decode, execute) and details of registers used.	

4.7.3.3 The processor instruction set

Content	Additional information
Understand the term 'processor instruction set' and know that an instruction set is processor specific.	

Content	Additional information
Know that instructions consist of an opcode and one or more operands (value, memory address or register).	A simple model will be used in which the addressing mode will be incorporated into the bits allocated to the opcode so the latter defines both the basic machine operation and the addressing mode. Students will not be expected to define opcode, only interpret opcodes in the given context of a question. For example, 4 bits have been allocated to the opcode (3 bits for basic machine operation, eg ADD, and 1 bit for the addressing mode). 4 bits have been allocated to the operand, making the instruction, opcode + operand, 8 bits in length. In this example, 16 different opcodes are possible ($2^4 = 16$).  Opcode Basic Machine Operation Addressing Mode Operand 0 0 1 0 0 1 0 1

4.7.3.4 Addressing modes

Content	Additional information
Understand and apply immediate and direct addressing modes.	Immediate addressing: the operand is the datum. Direct addressing: the operand is the address of the datum. Address to be interpreted as meaning either main memory or register.

4.7.3.5 Machine-code/assembly language operations

Content	Additional information
Understand and apply the basic machine-code operations of: • load • add • subtract • store • branching (conditional and unconditional) • compare • logical bitwise operators (AND, OR, NOT, XOR) • logical • shift right • shift left • halt. Use the basic machine-code operations above when machine-code instructions are expressed in mnemonic form- assembly language, using immediate and direct addressing.	

4.7.3.6 Interrupts

Content	Additional information
Describe the role of interrupts and interrupt service routines (ISRs); their effect on the Fetch-Execute cycle; and the need to save the volatile environment while the interrupt is being serviced.	

4.7.3.7 Factors affecting processor performance

Content	Additional information
Explain the effect on processor performance of: • multiple cores • cache memory • clock speed • word length • address bus width • data bus width.	

4.7.4 External hardware devices

4.7.4.1 Input and output devices

Content	Additional information
Know the main characteristics, purposes and suitability of the devices and understand their principles of operation.	Devices that need to be considered are: • barcode reader • digital camera • laser printer • RFID.

4.7.4.2 Secondary storage devices

Content	Additional information
Explain the need for secondary storage within a computer system.	
Know the main characteristics, purposes, suitability and understand the principles of operation of the following devices: • hard disk • optical disk • solid-state disk (SSD).	SSD = NAND flash memory + a controller that manages pages, and blocks and complexities of writing. Based on floating gate transistors that trap and store charge. A block, made up of many pages, cannot overwrite pages, page has to be erased before it can be written to but technology requires the whole block to be erased. Lower latency and faster transfer speeds than a magnetic disk drive.
Compare the capacity and speed of access of various media and make a judgement about their suitability for different applications.	

4.8 Consequences of uses of computing

4.8.1 Individual (moral), social (ethical), legal and cultural issues and opportunities

Content	Additional information
Show awareness of current individual (moral), social (ethical), legal and cultural opportunities and risks of computing. Understand that: • developments in computer science and the digital technologies have dramatically altered the shape of communications and information flows in societies, enabling massive transformations in the capacity to: ◦ monitor behaviour ◦ amass and analyse personal information ◦ distribute, publish, communicate and disseminate personal information. • computer scientists and software engineers therefore have power, as well as the responsibilities that go with it, in the algorithms that they devise and the code that they deploy. • software and their algorithms embed moral and cultural values. • the issue of scale, for software the whole world over, creates potential for individual computer scientists and software engineers to produce great good, but with it comes the ability to cause great harm. Be able to discuss the challenges facing legislators in the digital age.	Teachers may wish to employ two very powerful techniques, hypotheticals and case studies, to engage students in the issues. Hypotheticals allow students to isolate quickly important ethical principles in an artificially simplified context. For example, a teacher might ask students to explain and defend how, as a Google project manager, they would evaluate a proposal to bring Google's Street View technology to a remote African village. What questions should be asked? Who should be consulted? What benefits, risks and safeguards considered? What are the trade-offs? Case studies allow students to confront the tricky interplay between the sometimes competing ethical values and principles relevant in real world settings. For example, the Google Street View case might be used to tease out the ethical conflicts between individual and cultural expectations, the principle of informed consent, Street View's value as a service, its potential impact on human perceptions and behaviours, and its commercial value to Google and its shareholders. There are many resources available on the Internet to support teaching of this topic.

4.9 Fundamentals of communication and networking

4.9.1 Communication

4.9.1.1 Communication methods

Content	Additional information
Define serial and parallel transmission methods and discuss the advantages of serial over parallel transmission.	
Define and compare synchronous and asynchronous data transmission.	
Describe the purpose of start and stop bits in asynchronous data transmission.	

4.9.1.2 Communication basics

Content	Additional information
Define: • baud rate • bit rate • bandwidth • latency • protocol.	
Differentiate between baud rate and bit rate.	Bit rate can be higher than baud rate if more than one bit is encoded in each signal change.
Understand the relationship between bit rate and bandwidth.	Bit rate is directly proportionate to bandwidth.

4.9.2 Networking

4.9.2.1 Network topology

Content	Additional information
Understand: • physical star topology • logical bus network topology and: • differentiate between them • explain their operation	A network physically wired in star topology can behave logically as a bus network by using a bus protocol and appropriate physical switching.

4.9.2.2 Types of networking between hosts

Content	Additional information
Explain the following and describe situations where they might be used: • peer-to-peer networking • client-server networking.	In a peer-to-peer network, each computer has equal status. In a client-server network, most computers are nominated as clients and one or more as servers. The clients request services from the servers, which provide these services, for example file server, email server.

4.9.2.3 Wireless networking

Content	Additional information
Explain the purpose of WiFi.	A wireless local area network that is based on international standards. Used to enable devices to connect to a network wirelessly.
Be familiar with the components required for wireless networking.	Wireless network adapter. Wireless access point.
Be familiar with how wireless networks are secured.	Strong encryption of transmitted data using WPA (Wifi Protected Access)/WPA2, SSID (Service Set Identifier) broadcast disabled, MAC (Media Access Control) address allow list.
Explain the wireless protocol Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) with and without Request to Send/ Clear to Send (RTS/CTS).	Knowledge of Carrier Sense Multiple Access/ Collection Detection (CSMA/CD) as used in, for example, Ethernet, is not required.
Be familiar with the purpose of Service Set Identifier (SSID).	

4.9.3 The Internet

4.9.3.1 The Internet and how it works

Content	Additional information
Understand the structure of the Internet.	
Understand the role of packet switching and routers.	
Know the main components of a packet.	
Define: • router • gateway. Consider where and why they are used.	
Explain how routing is achieved across the Internet.	
Describe the term 'uniform resource locator' (URL) in the context of internetworking.	
Explain the terms ‘fully qualified domain name’ (FQDN), ‘domain name’ and ‘IP address’.	
Describe how domain names are organised.	
Understand the purpose and function of the domain service and its reliance on the Domain Name Server (DNS) system.	
Explain the service provided by Internet registries and why they are needed.	

4.9.3.2 Internet security

Content	Additional information
Understand how a firewall works (packet filtering, proxy server, stateful inspection).	
Explain symmetric and asymmetric (private/ public key) encryption and key exchange.	
Explain how digital certificates and digital signatures are obtained and used.	

Content	Additional information
Discuss worms, trojans and viruses, and the vulnerabilities that they exploit.	
Discuss how improved code quality, monitoring and protection can be used to address worms, trojans and viruses.	

4.9.4 The Transmission Control Protocol/Internet Protocol (TCP/IP) protocol

4.9.4.1 TCP/IP

Content	Additional information
Describe the role of the four layers of the TCP/IP stack (application, transport, network, link).	
Describe the role of sockets in the TCP/IP stack.	
Be familiar with the role of MAC (Media Access Control) addresses.	
Explain what the well-known ports and client ports are used for and the differences between them.	

4.9.4.2 Standard application layer protocols

Content	Additional information
Be familiar with the following protocols: • FTP (File Transfer Protocol) • HTTP (Hypertext Transfer Protocol) • HTTPS (Hypertext Transfer Protocol Secure) • POP3 (Post Office Protocol (v3)) • SMTP (Simple Mail Transfer Protocol) • SSH (Secure Shell).	
Be familiar with FTP client software and an FTP server, with regard to transferring files using anonymous and non-anonymous access.	
Be familiar with how SSH is used for remote management.	

Content	Additional information
Know how an SSH client is used to make a TCP connection to a remote port for the purpose of sending commands to this port using application level protocols such as GET for HTTP, SMTP commands for sending email and POP3 for retrieving email.	
Be familiar with using SSH to log in securely to a remote computer and execute commands.	
Explain the role of an email server in retrieving and sending email.	
Explain the role of a web server in serving up web pages in text form.	
Understand the role of a web browser in retrieving web pages and web page resources and rendering these accordingly.	

4.9.4.3 IP address structure

Content	Additional information
Know that an IP address is split into a network identifier part and a host identifier part.	

4.9.4.4 Subnet masking

Content	Additional information
Know that networks can be divided into subnets and know how a subnet mask is used to identify the network identifier part of the IP address.	

4.9.4.5 IP standards

Content	Additional information
Know that there are currently two standards of IP address, v4 and v6.	
Know why v6 was introduced.	

4.9.4.6 Public and private IP addresses

Content	Additional information
Distinguish between routable and non-routable IP addresses.	

4.9.4.7 Dynamic Host Configuration Protocol (DHCP)

Content	Additional information
Understand the purpose and function of the DHCP system.	

4.9.4.8 Network Address Translation (NAT)

Content	Additional information
Explain the basic concept of NAT and why it is used.	

4.9.4.9 Port forwarding

Content	Additional information
Explain the basic concept of port forwarding and why it is used.	

4.9.4.10 Client server model

Content	Additional information
Be familiar with the client server model.	Client sends a request message to server, server responds to request by replying with a response message to client.
Be familiar with the Websocket protocol and know why it is used and where it is used.	The Websocket specification defines an API (Application Programming Interface) establishing a full-duplex 'socket' connection between a web browser and a server over TCP. This means that there is a persistent connection between client and server, allowing both parties to send data at any time.

Content	Additional information
Be familiar with the principles of Web CRUD Applications and REST: • CRUD is an acronym for: • C – Create • R – Retrieve • U – Update • D – Delete. • REST enables CRUD to be mapped to database functions (SQL) as follows: • GET → SELECT • POST → INSERT • DELETE → DELETE • PUT → UPDATE.	Students should understand the principles: • database connected to browser using REST – Representational State Transfer - which relies on HTTP request methods • REST allows JavaScript to talk to server through HTTP • REST API (Application Programming Interface) created and run on server, browser Javascript calls API • JSON (JavaScript Object Notation) or XML can be used to transmit data between a server and web application • Javascript referenced by HTML file, eg index.html, is run in browser.
Compare JSON (Java script object notation) with XML.	JSON compared with XML is: • easier for a human to read • more compact • easier to create • easier for computers to parse and therefore quicker to parse.

4.9.4.11 Thin- versus thick-client computing

Content	Additional information
Compare and contrast thin-client computing with thick-client computing.	

4.10 Fundamentals of databases

4.10.1 Conceptual data models and entity relationship modelling

Content	Additional information
Produce a data model from given data requirements for a simple scenario involving multiple entities.	
Produce entity relationship diagrams representing a data model and entity descriptions in the form: Entity1 (Attribute1, Attribute2, ...).	Underlining can be used to identify the attribute(s) which form the entity identifier.

4.10.2 Relational databases

Content	Additional information
Explain the concept of a relational database.	
Be able to define the terms: • attribute • primary key • composite primary key • foreign key.	

4.10.3 Database design and normalisation techniques

Content	Additional information
Normalise relations to third normal form.	Students should know what properties are possessed by a relation in third normal form.
Understand why databases are normalised.	

4.10.4 Structured Query Language (SQL)

Content	Additional information
Be able to use SQL to retrieve, update, insert and delete data from multiple tables of a relational database.	
Be able to use SQL to define a database table.	

4.10.5 Client server databases

Content	Additional information
Know that a client server database system provides simultaneous access to the database for multiple clients. Know how concurrent access can be controlled to preserve the integrity of the database.	Concurrent access can result in the problem of updates being lost if two clients edit a record at the same time. This problem can be managed by the use of record locks, serialisation, timestamp ordering, commitment ordering.

4.11 Big Data

4.11.1 Big Data

Content	Additional information
Know that 'Big Data' is a catch-all term for data that won't fit the usual containers. Big Data can be described in terms of: • volume - too big to fit into a single server • velocity - streaming data, milliseconds to seconds to respond • variety - data in many forms such as structured, unstructured, text, multimedia.	Whilst its size receives all the attention, the most difficult aspect of Big Data really involves its lack of structure. This lack of structure poses challenges because: • analysing the data is made significantly more difficult • relational databases are not appropriate because they require the data to fit into a row-and-column format. Machine learning techniques are needed to discern patterns in the data and to extract useful information. 'Big' is a relative term, but size impacts when the data doesn't fit onto a single server because relational databases don't scale well across multiple machines. Data from networked sensors, smartphones, video surveillance, mouse clicks etc are continuously streamed.
Know that when data sizes are so big as not to fit on to a single server: • the processing must be distributed across more than one machine • functional programming is a solution, because it makes it easier to write correct and efficient distributed code. Know what features of functional programming make it easier to write: • correct code • code that can be distributed to run across more than one server.	Functional programming languages support: • immutable data structures • statelessness • higher-order functions.
Be familiar with the: • fact-based model for representing data • graph schema for capturing the structure of the dataset • nodes, edges and properties in graph schema.	Each fact within a fact-based model captures a single piece of information.

4.12 Fundamentals of functional programming

4.12.1 Functional programming paradigm

4.12.1.1 Function type

Content	Additional information
Know that a function, f, has a function type f: A → B (where the type is A → B, A is the argument type, and B is the result type). Know that A is called the domain and B is called the co-domain. Know that the domain and co-domain are always subsets of objects in some data type.	Loosely speaking, a *function* is a rule that, for each element in some set A of inputs, assigns an output chosen from set B, but without necessarily using every member of B. For example, f: {a,b,c,...z} → {0,1,2,...,25} could use the rule that maps *a* to 0, *b* to 1, and so on, using all values which are members of set B. The *domain* is a set from which the function's input values are chosen. The *co-domain* is a set from which the function's output values are chosen. Not all of the co-domain's members need to be outputs.

4.12.1.2 First-class object

Content	Additional information
Know that a function is a first-class object in functional programming languages and in imperative programming languages that support such objects. This means that it can be an argument to another function as well as the result of a function call.	First-class objects (or values) are objects which may: • appear in expressions • be assigned to a variable • be assigned as arguments • be returned in function calls. For example, integers, floating-point values, characters and strings are first class objects in many programming languages.

4.12.1.3 Function application

Content	Additional information
Know that function application means a function applied to its arguments.	The process of giving particular inputs to a function is called *function application*, for example add(3,4) represents the application of the function *add* to integer arguments 3 and 4. The type of the function is *f: integer* x *integer* → *integer* where *integer* x *integer* is the Cartesian product of the set *integer* with itself. Although we would say that function *f* takes two arguments, in fact it takes only one argument, which is a pair, for example (3,4).

4.12.1.4 Partial function application

Content	Additional information
Know what is meant by partial function application for one, two and three argument functions and be able to use the notations shown opposite.	The function *add* takes two *integers* as arguments and gives an *integer* as a result. Viewed as follows in the partial function application scheme: *add: integer → (integer → integer)* add 4 returns a function which when applied to another integer adds 4 to that integer. The brackets may be dropped so function *add* becomes add: *integer → integer → integer* The function add is now viewed as taking one argument after another and returning a result of data type *integer*.

4.12.1.5 Composition of functions

Content	Additional information
Know what is meant by composition of functions.	The operation *functional composition* combines two functions to get a new function. Given two functions $f: A \rightarrow B$ $g: B \rightarrow C$ function $g \bigcirc f$, called the *composition* of g and f, is a function whose domain is A and co-domain is C. If the domain and co-domains of f and g are $\mathbb{R}$, and $f(x) = (x + 2)$ and $g(y) = y^3$. Then $g \bigcirc f = (x + 2)^3$ f is applied first and then g is applied to the result returned by f.

4.12.2 Writing functional programs

4.12.2.1 Functional language programs

Content	Additional information
Show experience of constructing simple programs in a functional programming language.	The following is a list of functional programming languages that could be used: • Haskell • Standard ML • Scheme • Lisp. Other languages with built-in support for programming in a functional paradigm as well as other paradigms are: • Python • F# • C# • Scala • Java 8 • Delphi XE versions onwards • VB.NET 2008 onwards.
Higher-order functions.	A function is higher-order if it takes a function as an argument or returns a function as a result, or does both.

Content	Additional information
Have experience of using the following in a functional programming language: • map • filter • reduce or fold.	*map* is the name of a higher-order function that applies a given function to each element of a list, returning a list of results. *filter* is the name of a higher-order function that processes a data structure, typically a list, in some order to produce a new data structure containing exactly those elements of the original data structure that match a given condition. *reduce* or *fold* is the name of a higher-order function which reduces a list of values to a single value by repeatedly applying a combining function to the list values.

4.12.3 Lists in functional programming

4.12.3.1 List processing

Content	Additional information
Be familiar with representing a list as a concatenation of a head and a tail. Know that the head is an element of a list and the tail is a list. Know that a list can be empty. Describe and apply the following operations: • return head of list • return tail of list • test for empty list • return length of list • construct an empty list • prepend an item to a list • append an item to a list. Have experience writing programs for the list operations mentioned above in a functional programming language or in a language with support for the functional paradigm.	For example, in Haskell the list [4, 3, 5] can be written in the form *head:tail* where *head* is the first item in the list and *tail* is the remainder of the list. In the example, we have 4:[3, 5]. We call 4 the *head* of the list and [3, 5] the *tail*. [] is the *empty list*.

4.13 Systematic approach to problem solving

4.13.1 Aspects of software development

4.13.1.1 Analysis

Content	Additional information
Be aware that before a problem can be solved, it must be defined, the requirements of the system that solves the problem must be established and a data model created. Requirements of system must be established by interaction with the intended users of the system. The process of clarifying requirements may involve prototyping/agile approach.	Students should have experience of using abstraction to model aspects of the external world in a program.

4.13.1.2 Design

Content	Additional information
Be aware that before constructing a solution, the solution should be designed and specified, for example planning data structures for the data model, designing algorithms, designing an appropriate modular structure for the solution and designing the human user interface.	Students should have sufficient experience of successfully structuring programs into modular parts with clear documented interfaces to enable them to design appropriate modular structures for solutions.
Be aware that design can be an iterative process involving a prototyping/agile approach.	

4.13.1.3 Implementation

Content	Additional information
Be aware that the models and algorithms need to be implemented in the form of data structures and code (instructions) that a computer can understand.	Students should have sufficient practice of writing, debugging and testing programs to enable them to develop the skills to articulate how programs work arguing for their correctness and efficiency using logical reasoning, test data and user feedback.
Be aware that the final solution may be arrived at using an iterative process employing prototyping/an agile approach with a focus on solving the critical path first.	

4.13.1.4 Testing

Content	Additional information
Be aware that the implementation must be tested for the presence of errors, using selected test data covering normal (typical), boundary and erroneous data.	Students should have practical experience of designing and applying test data, normal, boundary and erroneous to the testing of programs so that they are familiar with these test data types and the purpose of testing.
It should also undergo acceptance testing with the intended user(s) of the system to ensure that the intended solution meets its specification.	Students will only need to provide evidence of user feedback not details of the tests carried out by the end user.

4.13.1.5 Evaluation

Content	Additional information
Know the criteria for evaluating a computer system.	

4.14 Non-exam assessment - the computing practical project

4.14.1 Overview

4.14.1.1 Purpose of the project

The project allows students to develop their practical skills in the context of solving a realistic problem or carrying out an investigation. The project is intended to be as much a learning experience as a method of assessment; students have the opportunity to work independently on a problem of interest over an extended period, during which they can extend their programming skills and deepen their understanding of computer science.

The most important skill that should be assessed through the project is a student's ability to create a programmed solution to a problem or investigation. This is recognised by allocating 42 of the 75 available marks to the technical solution and a lower proportion of marks for supporting documentation to reflect the expectation that reporting of the problem, its analysis, the design of a solution or plan of an investigation and testing and evaluation will be concise.

4.14.1.2 Types of problem/investigation

Students are encouraged to choose a problem to solve or investigate that will interest them and that relates to a field that they have some knowledge of. There are no restrictions on the types of problem/investigation that can be submitted or the development tools (for example programming language) that can be used. The two key questions to ask when selecting a problem/investigation are:

- Does the student have existing knowledge of the field, or are they in a position to find out about it?
- Is a solution to the problem/investigation likely to give the student the opportunity to demonstrate the necessary degree of technical skill to achieve a mark that reflects their potential?

Some examples of the types of problem to solve or investigate are:

- a simulation for example, of a business or scientific nature, or an investigation of a well-known problem such as the game of life
- a solution to a data processing problem for an organisation, such as membership systems
- the solution of an optimisation problem, such as production of a rota, shortest-path problems or route finding
- a computer game
- an application of artificial intelligence
- a control system, operated using a device such as an Arduino board
- a website with dynamic content, driven by a database back-end
- an app for a mobile phone or tablet
- an investigation into an area of computing, such as rendering a three-dimensional world on screen
- investigating an area of data science using, for example, Twitter feed data or online public data sets
- investigating machine learning algorithms.

There is an expectation that within a centre, the problems chosen by students to solve or investigate will be sufficiently different to avoid the work of one student informing the work of another because they are working on the same problem or investigation. Teachers will be required to record on the Candidate Record Form for each student that they have followed this guideline. If in any doubt on whether problems chosen by students have the potential to raise this issue, please contact your AQA adviser.

Table 1 and Table 2 show the technical skills and coding styles required for an A-level standard project. If a problem/investigation is selected that is not of A-level standard then the marks available in each section will be restricted.

4.14.1.3 Project documentation structure

The project is assessed in five sections. The table below lists the maximum available mark for each section of the project:

Section		Max mark
1	Analysis	9
2	Documented design	12
3	Technical solution	42
4	Testing	8
5	Evaluation	4
Total		**75**

For marking purposes, the project documentation should be presented in the order indicated in the table above. The table does not imply that students are expected to follow a traditional systems life cycle approach when working on their projects, whereby a preceding stage must be completed before the next can be tackled. It is recognised that this approach is unsuited to the vast majority of project work, and that project development is likely to be an iterative process, with earlier parts of the project being revisited as a result of discoveries made in later parts. Students should be encouraged to start prototyping and writing code early on in the project process. A recommended strategy is to tackle the critical path early in the project development process. The critical path is the part of the project that everything else depends on for a working system or a complete investigation result to be achieved.

4.14.2 Using a level of response mark scheme

Level of response mark schemes are broken down into a number of levels, each of which has a descriptor. The descriptor for the level shows the average performance for the level. There are a range of marks in each level. The descriptor for the level represents a typical mid-mark performance in that level.

Before applying the mark scheme to a student’s project, read it through and annotate it to show the qualities that are being looked for. You can then apply the mark scheme.

4.14.2.1 Step 1 Determine a level

Start at the lowest level of the mark scheme and use it as a ladder to see whether the performance in that section of the project meets the descriptor for that level. The descriptor for the level indicates the different qualities that might be seen in the student’s work for that level. If it meets the lowest level then go to the next one and decide if it meets this level, and so on, until you have a match between the level descriptor and the work. With practice and familiarity you will find you will be able to quickly skip through the lower levels of the mark scheme.

When assigning a level you should look at the overall quality of the work rather than any small or specific parts where the student has not performed quite as the level descriptor. If the work covers different aspects of different levels of the mark scheme you should use a best fit approach for defining the level and then use the variability of the response to help decide the mark within the level. ie if the response is predominantly level 3 with a small amount of level 4 material it would be placed in level 3 but be awarded a mark near the top of the level because of the level 4 content.

4.14.2.2 Step 2 Determine a mark

Once you have assigned a level you need to decide on the mark. The exemplar materials used for standardisation will help. This work will have been awarded a mark by AQA. You can compare your student’s work with the exemplar to determine if it is the same standard, better or worse. You can then use this to allocate a mark for the work based on AQA's mark on the exemplar.

You may well need to read back through the work as you apply the mark scheme to clarify points and assure yourself that the level and the mark are appropriate.

Work which contains nothing of relevance to the project area being assessed must be awarded no marks for that area.

4.14.3 Marking criteria

4.14.3.1 Analysis (9 marks)

Level	Mark range	Description
3	7-9	Fully or nearly fully scoped analysis of a real problem, presented in a way that a third party can understand. Requirements fully documented in a set of measurable and appropriate specific objectives, covering all required functionality of the solution or areas of investigation. Requirements arrived at by considering, through dialogue, the needs of the intended users of the system, or recipients of the outcomes for investigative projects. Problem sufficiently well modelled to be of use in subsequent stages.
2	4-6	Well scoped analysis (but with some omissions that are not serious enough to undermine later design) of a real problem. Most, but not all, requirements documented in a set of, in the main, measurable and appropriate specific objectives that cover most of the required functionality of a solution or areas of investigation. Requirements arrived at, in the main, by considering, through dialogue, the needs of the intended users of the system, or recipients of the outcomes for investigative projects. Problem sufficiently well modelled to be of use in subsequent stages.
1	1-3	Partly scoped analysis of a problem. Requirements partly documented in a set of specific objectives, not all of which are measurable or appropriate for developing a solution. The required functionality or areas of investigation are only partly addressed. Some attempt to consider, through dialogue, the needs of the intended users of the system, or recipients of the outcomes for investigative projects. Problem partly modelled and of some use in subsequent stages.

4.14.3.2 Documented design (12 marks)

Level	Mark range	Description
4	10-12	Fully or nearly fully articulated design for a real problem, that describes how all or almost all of the key aspects of the solution/investigation are to be structured/are structured.
3	7-9	Adequately articulated design for a real problem that describes how most of the key aspects of the solution/ investigation are to be structured/are structured.
2	4-6	Partially articulated design for a real problem that describes how some aspects of the solution/investigation are to be structured/are structured.
1	1-3	Inadequate articulation of the design of the solution so that it is difficult to obtain a picture of how the solution/ investigation is to be structured/is structured without resorting to looking directly at the programmed solution.

4.14.3.3 Technical solution (42 marks)

4.14.3.3.1 *Completeness of solution (15 marks)*

Level	Mark range	Description
3	11-15	A system that meets almost all of the requirements of a solution/an investigation (ignoring any requirements that go beyond the demands of A-level).
2	6-10	A system that achieves many of the requirements but not all. The marks at the top end of the band are for systems that include some of the most important requirements.
1	1-5	A system that tackles some aspects of the problem or investigation.

4.14.3.3.2 Techniques used (27 marks)

Level	Mark Range	Description	Additional information
3	19-27	The techniques used are appropriate and demonstrate a level of technical skill equivalent to those listed in Group A in Table 1. Program(s) demonstrate(s) that the skill required for this level has been applied sufficiently to demonstrate proficiency.	Above average performance: Group A equivalent algorithms and model programmed more than well to excellent; all or almost all excellent coding style characteristics; more than to highly effective solution. Average performance: Group A equivalent algorithms and/or model programmed well; majority of excellent coding style characteristics; an effective solution. Below average performance: Group A equivalent algorithms and/or model programmed just adequately to fully adequate; some excellent coding style characteristics; less than effective to fairly effective solution.
2	10-18	The techniques used are appropriate and demonstrate a level of technical skill equivalent to those listed in Group B in Table 1. Program(s) demonstrate(s) that the skill required for this level has been applied sufficiently to demonstrate proficiency.	Above average performance: Group B equivalent algorithms and model programmed more than well to excellent; majority of excellent coding style characteristics; more than to highly effective solution. Average performance: Group B equivalent algorithms and/or model programmed well; some excellent coding style characteristics; effective solution. Below average performance: Group B equivalent algorithms and/or model programmed just adequately to fully adequate; all or almost all relevant good coding style characteristics but possibly one example at most of excellent characteristics; less than effective to fairly effective solution.

Level	Mark Range	Description	Additional information
1	1-9	The techniques used demonstrate a level of technical skill equivalent to those listed in Group C in Table 1. Program(s) demonstrate(s) that the skill required for this level has been applied sufficiently to demonstrate proficiency.	Above average performance: Group C equivalent model and algorithms programmed more than well to excellent; almost all relevant good coding style characteristics; more than to highly effective simple solution. Average performance: Group C equivalent model and algorithms programmed well; some relevant good coding style characteristics; effective simple solution. Below average performance: Group C equivalent algorithms and/or model programmed in a severely limited to limited way; basic coding style characteristics; trivial to lacking in effectiveness simple solution.

Select the band, 1, 2 or 3 with level of demand description that best matches the techniques and skill that the student's program attempts to cover. The emphasis is on what the student has actually achieved that demonstrates proficiency at this level rather than what the student has set out to use and do but failed to demonstrate, eg because of poor execution. Check the proficiency demonstrated in the program. If the student fails to demonstrate proficiency at the initial level of choice, drop down a level to see if what the student has done demonstrates proficiency at this level for the lower demand until a match is obtained. Table 1 is indicative of the standard required and is not to be treated as just a list of things for students to select from and to be automatically credited for including in their work.

As indicated above, having selected the appropriate level for techniques used and proficiency in their use, the exact mark to award should be determined based upon:

- the extent to which the criteria for the mark band have been achieved
- the quality of the coding style that the student has demonstrated (see Table 2 for exemplification of what is expected)
- the effectiveness of the solution.

4.14.3.4 Example technical skills

4.14.3.4.1 *Table 1: Example technical skills*

Group	Model (including data model/structure)	Algorithms
A	Complex data model in database (eg several interlinked tables)	Cross-table parameterised SQL Aggregate SQL functions User/CASE-generated DDL script
	Hash tables, lists, stacks, queues, graphs, trees or structures of equivalent standard Files(s) organised for direct access	Graph/Tree Traversal List operations Linked list maintenance Stack/Queue Operations Hashing
	Complex scientific/mathematical/robotics/ control/business model	Advanced matrix operations Recursive algorithms Complex user-defined algorithms (eg optimisation, minimisation, scheduling, pattern matching) or equivalent difficulty Mergesort or similarly efficient sort
	Complex user-defined use of object-orientated programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces Complex client-server model	Dynamic generation of objects based on complex user-defined use of OOP model Server-side scripting using request and response objects and server-side extensions for a complex client-server model Calling parameterised Web service APIs and parsing JSON/XML to service a complex client-server model

Group	Model (including data model/structure)	Algorithms
B	Simple data model in database (eg two or three interlinked tables) Multi-dimensional arrays Dictionaries Records Text files File(s) organised for sequential access Simple scientific/mathematical /robotics/ control/business model Simple OOP model Simple client-server model	Single table or non-parameterised SQL Bubble sort Binary search Writing and reading from files Simple user defined algorithms (eg a range of mathematical/statistical calculations) Generation of objects based on simple OOP model Server-side scripting using request and response objects and server-side extensions for a simple client-server model Calling Web service APIs and parsing JSON/XML to service a simple client-server model
C	Single-dimensional arrays Appropriate choice of simple data types Single table database	Linear search Simple mathematical calculations (eg average) Non-SQL table access

Note that the contents of Table 1 are examples, selected to illustrate the level of demand of the technical skills that would be expected to be demonstrated in each group. The use of alternative algorithms and data models is encouraged. If a project cannot easily be marked against Table 1 (for example, a project with a considerable hardware component) then please consult your AQA non-exam assessment Adviser or provide a full explanation of how you have arrived at the mark for this section when submitting work for moderation.

4.14.3.4.2 *Table 2: Coding styles*

Style	Characteristic
Excellent	Modules (subroutines) with appropriate interfaces. Loosely coupled modules (subroutines) – module code interacts with other parts of the program through its interface only. Cohesive modules (subroutines) – module code does just one thing. Modules(collections of subroutines) – subroutines with common purpose grouped. Defensive programming. Good exception handling.
Good	Well-designed user interface Modularisation of code Good use of local variables Minimal use of global variables Managed casting of types Use of constants Appropriate indentation Self-documenting code Consistent style throughout File paths parameterised
Basic	Meaningful identifier names Annotation used effectively where required

The descriptions in Table 2 are cumulative, ie for a program to be classified as excellent it would be expected to exhibit the characteristics listed as excellent, good and basic not just those listed as excellent.

4.14.3.5 Testing (8 marks)

Level	Mark range	Description
4	7-8	Clear evidence, in the form of carefully selected representative samples, that thorough testing has been carried out. This demonstrates the robustness of the complete or nearly complete solution/thoroughness of investigation and that the requirements of the solution/ investigation have been achieved.

Level	Mark range	Description
3	5-6	Extensive testing has been carried out, but the evidence presented in the form of representative samples does not make clear that all of the core requirements of the solution/ investigation have been achieved. This may be due to some key aspects not being tested or because the evidence is not always presented clearly.
2	3-4	Evidence in the form of representative samples of moderately extensive testing, but falling short of demonstrating that the requirements of the solution/ investigation have been achieved and the solution is robust/ investigation thorough. The evidence presented is explained.
1	1-2	A small number of tests have been carried out, which demonstrate that some parts of the solution work/some outcomes of the investigation are achieved. The evidence presented may not be entirely clear.

Evidence for the testing section may be produced after the system has been fully coded or during the coding process. It is expected that tests will either be planned in a test plan or that the tests will be fully explained alongside the evidence for them. Only carefully selected representative samples are required.

4.14.3.6 Evaluation (4 marks)

Level	Mark	Description
4	4	Full consideration given to how well the outcome meets all of its requirements. How the outcome could be improved if the problem was revisited is discussed and given detailed consideration. Independent feedback obtained of a useful and realistic nature, evaluated and discussed in a meaningful way.
3	3	Full or nearly full consideration given to how well the outcome meets all of its requirements. How the outcome could be improved if the problem was revisited is discussed but consideration given is limited. Independent feedback obtained of a useful and realistic nature but is not evaluated and discussed in a meaningful way, if at all.

Level	Mark	Description
2	2	The outcome is discussed but not all aspects are fully addressed either by omission or because some of the requirements have not been met and those requirements not met have been ignored in the evaluation. No independent feedback obtained or if obtained is not sufficiently useful or realistic to be evaluated in a meaningfully way even if attempted.
1	1	Some of the outcomes are assessed but only in a superficial way. No independent feedback obtained or if obtained is so basic as to be not worthy of evaluation.

4.14.4 Project tasks that are not of A-level standard

If the task (problem or investigation) selected for a project is not of A-level standard, mark the project against the criteria given, but adjust, the mark awarded downwards by two marking levels (two marks in the case of evaluation) in each section for all but the technical solution. You should have already taken the standard into account for this, by directly applying the criteria. For example, if a student had produced a 'fully or nearly fully articulated design of a real problem describing how solution is to be structured/is structured'. This would, for an A-level standard project, achieve a mark in Level Four for Documented Design (10-12 marks). If the problem selected was too simple to be of A-level standard but the same criteria had been fulfilled, shift the mark awarded down by two levels, into Level Two, an award of 4-6 marks. If a downward shift by two levels is not possible, then a mark in the lowest level should be awarded.

4.14.5 Guide to non-exam assessment documentation

4.14.5.1 Analysis

Students are expected to:

- produce a clear statement that describes the problem area and specific problem that is being solved/investigated
- outline how they researched the problem
- state for whom the problem is being solved/investigated
- provide background in sufficient detail for a third party to understand the problem being solved/investigated
- produce a numbered list of measurable, "appropriate" specific objectives, covering all required functionality of the solution or areas of investigation (Appropriate means that the specific objectives are single purpose and at a level of detail that is without ambiguity.)
- report any modelling of the problem that will inform the Design stage, for example a graph/ network model of Facebook connections or an E-R model.

A fully scoped analysis is one that has:

- researched the problem thoroughly
- has clearly defined the problem being solved/investigated
- omitted nothing that is relevant to subsequent stages

- statements of objectives which clearly and unambiguously identify the scope of the project
- modelled the problem for the Design stage where this is possible and necessary.

4.14.5.2 Design

Students are expected to articulate their design in a manner appropriate to the task and with sufficient clarity for a third party to understand how the key aspects of the solution/investigation are structured and on what the design will rely, eg use of numerical and scientific package libraries, data visualisation package library, particular relational database and/or web design framework. The emphasis is on communicating the design; therefore it is acceptable to provide a description of the design in a combination of diagrams and prose as appropriate, as well as a description of algorithms, SQL, data structures, database relations as appropriate, and using relevant technical description languages, such as pseudo-code. Where design of a user interface is relevant, screen shots of actual screens are acceptable.

4.14.5.3 Technical solution

Students should provide program listing(s) that demostrate their technical skill. The program listing(s) should be appropriately annotated and self-documenting (an approach that uses meaningful identifiers, with well structured code that minimises instances where program comments are necessary).

Students should present their work in a way that will enable a third party to discern the quality and purpose of the coding. This could take the form of:

- an overview guide which amongst other things includes the names of entities such as executables, data filenames/urls, database names, pathnames so that a third party can, if they so desire, run the solution/investigation
- explanations of particularly difficult-to-understand code sections; a careful division of the presentation of the code listing into appropriately labelled sections to make navigation as easy as possible for a third party reading the code listing.

Achievement of the latter, to an extent, is linked to the skill in applying a structured approach during the course of developing the solution or carrying out the investigation.

4.14.5.4 Testing

Students must provide and present in a structured way for example in tabular form, clear evidence of testing. This should take the form of carefully selected and representative samples, which demonstrate the robustness of the complete, or nearly complete, solution/thoroughness of investigation and which demonstrate that the requirements of the solution/investigation have been achieved. The emphasis should be on producing a representative sample in a balanced way and not on recording every possible test and test outcome. Students should explain the tests carried out alongside the evidence for them. This could take the form of:

- an introduction and overview
- the test performed
- its purpose if not self-evident
- the test data
- the expected test outcome
- the actual outcome with a sample of the evidence, for example screen shots of before and after the test, etc, sampled in order to limit volume.

4.14.5.5 Evaluation

Students should consider and assess how well the outcome meets its requirements. Students should obtain independent feedback on how well the outcome meets its requirements and discuss this feedback. Some of this feedback could be generated during prototyping. If so, this feedback, and how/why it was taken account must be presented and referenced so it can be found easily.

Students should also consider and discuss how the outcome could be improved more realistically if the problem/investigation were to be revisited.

4.14.6 Assessment objective breakdown for non-exam assessment

Section	Total	AO2	AO3	Elements
Analysis	9	9		AO2b
Design	12		12	AO3a
Technical Solution	42		42	AO3b
Testing	8		8	AO3c
Evaluation	4		4	AO3c
Totals	75	9	66	