A Level Computing

Course Content Checklist

Name:	
Tutor Group:	
Teaching Group:	
Target Grade:	

A Level Computing

Paper 1

Content
- Fundamentals of programming
- Fundamentals of problem solving
- Fundamentals of algorithms
- Theory of computation
- Available in a wide range of programming languages (C#, Java, Pascal/Delphi, Python (2,3), VB.Net)

Assessed
- 2 hour 30 minutes on-screen exam
- 100 marks in total
- Weighting 40% of A-level

Students are required to
- **Theory**: Answer a range of short and extended answer questions
- **Practical**: Complete a range of programming tasks based on seen and unseen material

+

Paper 2

Content
- Fundamentals of data representation
- Fundamentals of computer systems
- Fundamentals of computer organisation and architecture
- Consequences of uses of computing
- Fundamentals of communication and networking
- Fundamentals of data bases
- Big Data
- Fundamentals of functional programming

Assessed
- 2 hour 30 minutes written exam
- 100 marks in total
- Weighting 40% of A-level

Students are required to
- Answer a range of compulsory short and extended answer questions

+

Assessment 3

Non-exam assessment

Provisional range of content
- System development/investigative project
- Synoptic requirements are covered

Assessed
- Written report
- 75 marks in total
- Weighting 20% of A-level

Students are required to
- Produce a project that either meets the needs of a particular end user or investigates a particular aspect of how computers can be used to explore solutions to problems

Paper 1

This paper tests a student's ability to program, as well as their theoretical knowledge of computer science from subject content 10 – 13 and 22.

- On Screen Exam, 2 hours 30 minutes
- 40% of A-Level

Paper 2

This paper tests a student's ability to program, as well as their theoretical knowledge of computer science from subject content 14 – 21.

- On Screen Exam, 2 hours 30 minutes
- 40% of A-Level

Non Exam 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.

- 75 Marks
- 20% of A-Level

Course Content Checklist

4.1 Fundamentals of Programming

4.1.1 Programming

4.1.1.1 Data types	Covered	Revised
Understand the concept of a data type.		
Understand and use the following appropriately: • integer • real/float • Boolean • character • string • date/time • 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		
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		
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		
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		
Be familiar with and be able to use: • NOT • AND • OR • XOR		
4.1.1.6 Constants and variables in a programming language		
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		
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		
Be familiar with, and be able to use, random number generation.		

4.1.1.9 Exception handling		
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)		
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		
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		
Be able to use subroutines that return values to the calling routine.		
4.1.1.13 Local variables in subroutines		
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		
Be able to contrast local variables with global variables.		
4.1.1.15 Role of stack frames in subroutine calls		
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		
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			
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			
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			
Be familiar with the concepts of: • class • object • instantiation • encapsulation • inheritance • aggregation • association aggregation • composition aggregation • 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.		
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			
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			

4.2 Fundamentals of Data Structures

4.2.1 Data structures and abstract data types

4.2.1.1 Data structures		
Be familiar with the concept of data structures.		
4.2.1.2 Single- and multi-dimensional arrays (or equivalent)		
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		
Be able to read/write from/to a text file.		
Be able to read/write data from/to a binary (non- text) file.		
4.2.1.4 Abstract data types/data structures		
Be familiar with the concept and uses of a: • queue • stack • list • graph • tree • hash table • dictionary • vector		
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		
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		
Be able to describe and apply the following operations: • push • pop • peek or top • test for empty stack • test for stack full.		

4.2.4 Graphs

4.2.4.1 Graphs		
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		
Know that a tree is a connected, undirected graph with no cycles. Note that a tree does not have to have a root.		

Know that a rooted tree is a tree in which one vertex has been designed as the root and every edge is directed away from 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		
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		
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		
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 ℝ4 • function interpretation • 0 ↦ 2.0 • 1 ↦ 3.14159 • 2 ↦ -1.0 • 3 ↦ 2.718281828 • ↦ 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 → 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.		
List representation of a vector.		
1-D array representation of a vector.		
Visualising a vector as an arrow.		
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 αu + βv where α, β ≥ 0 and α + β = 1		
Dot or scalar product of two vectors. The dot product of two vectors, u and v, u = [u1, …., un] and v = [v1, ….., vn] is u · v = u1v1 + u2v2 + …… + unvn		

Applications of dot product. Generating parity given two vectors u and v over GF(2) : u = [1, 1, 1, 1] and v = [1, 0, 1, 1] u · v = 1 where GF(2) has two elements, 0 and 1. Arithmetic over GF(2) can be summarised in two small tables:		

*	0	1
0	0	0
1	0	1

This can be achieved by bitwise AND operation.

+	0	1
0	0	1
1	1	0

This can be achieved by bitwise XOR operation. Subtraction is identical to addition, -1 = 1 and -0 = 0.

4.3 Fundamentals of Algorithms

4.3.1 Graph Traversal

4.3.1.1 Simple graph-traversal algorithms		
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		
Be able to trace the tree-traversal algorithms: • pre-order • post-order • in-order ✓ Pre-order is a depth-first traversal. ✓ Post-order is a breadth-first traversal.		
Be able to describe uses of tree-traversal algorithms (Pre-Order, In-Order, Post-Order) • Pre-Order: copying a tree, producing prefix expression from an expression tree. • In-Order: binary search tree. • 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		
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		
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		
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		
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		
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(n2).		
4.3.5.2 Merge Sort		
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. It’s time complexity is O(nlog n).		

4.3.6 Optimisation algorithms

4.3.6.1 Dijkstra’s shortest path algorithm		
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		
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		
Understand the term algorithm.		
Be able to express the solution to a simple problem as an algorithm using pseudo-code, with the standard constructs: • sequence • assignment (x=1) • selection (IF ….. Then) • iteration (loops)		
Be able to hand-trace algorithms.		
Be able to convert an algorithm from pseudo-code into high level language program code.		
Be able to articulate how a program works, arguing for its correctness and its efficiency using logical reasoning, test data and user feedback.		
3.4.1.3 Abstraction		
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		
Be familiar with the process of hiding all details of an object that do not on tribute to its essential characteristics.		
4.4.1.5 Procedural abstraction		
Know that procedural abstraction represents a computational method.		
4.4.1.6 Functional abstraction		
Know that for functional abstraction the particular computation method is hidden.		
4.4.1.7 Data abstraction		
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.		
4.4.1.8 Problem abstraction/reduction		
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		
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		
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		
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.		

4.4.2 Regular Languages

4.4.2.1 Finite state machines (FSMs) without output		
Be able to draw and interpret simple state transition diagrams and state transition tables for FSMs with no output.		
4.4.2.2 Maths for regular expressions		
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.		
Be familiar with the compact representation of a set, for example, the set {0n1n | n ≥ 1}. This set contains all strings with an equal number of 0 s and 1s.		
Be familiar with the concept of: • finite sets • infinite sets • countably infinite sets • cardinality of a finite set • Cartesian product of sets.		

Be familiar with the meaning of the term: • subset • proper subset • countable set.		
Be familiar with the set operations: • membership • union • intersection • difference.		
4.4.2.3 Regular Expression		
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.		
Be able to form and use simple regular expressions for string manipulation and matching.		
Be able to describe the relationship between regular expressions and FSMs.		
Be able to write a regular expression to recognise the same language as a given FSM and vice versa.		
4.4.2.4 Regular Language		
Know that a language is called regular if it can be represented by a regular expression.		

4.4.3 Context Free Languages

4.4.3.1 Backus-Naur Form (BNF)/syntax diagrams		
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		
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.		
4.4.4.2 Maths for understanding Big-0 notation		

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 ℕ → ℕ.		
Be familiar with the concept of: • a linear function, for example y = 2x • a polynomial function, for example y = 2x2 • an exponential function, for example y = 2x • a logarithmic function, for example y = log10 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 permutation of n distinct objects is n factorial (n!).		
4.4.4.3 Order of complexity		
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		
Be aware that algorithmic complexity and hardware impose limits on what can be computed.		
4.4.4.5 Classification of algorithmic problems		
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.		
4.4.4.6 Computable and non-computable problems		
Be aware that some problems cannot be solved algorithmically.		
4.4.4.7 Halting problem		
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.		

4.4.5 A model of Computation

4.4.5.1 Turing machine		
Be familiar with the structure and use of Turing machines that perform simple computations.		

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.		
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.		

4.5 Fundamentals of Data Representation

4.5.1 Number systems

4.5.1.1 Natural numbers		
Be familiar with the concept of a natural number and the set ℕ of natural numbers (including zero).		
4.5.1.2 Integer numbers		
Be familiar with the concept of an integer and the set ℤ of integers.		
4.5.1.3 Rational numbers		
Be familiar with the concept of a rational number and the set ℚ of rational numbers, and that this set includes the integers.		
4.5.1.4 Irrational numbers		
Be familiar with the concept of an irrational number.		
4.5.1.5 Real numbers		
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.		
4.5.1.6 Ordinal numbers		
Be familiar with the concept of ordinal numbers and their use to describe the numerical positions of objects.		
4.5.1.7 Counting and measurement		
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		
Be familiar with the concept of a number base, in particular: • decimal (base 10) • binary (base 2) • hexadecimal (base 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		
Know that: • the bit is the fundamental unit of information • a byte is a group of 8 bits.		
Know that the 2n different values can be represented with n bits.		
4.5.3.2 Units		
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 = 210 B and one kilobyte is written as 1 kB = 103 B. Know the names, symbols and corresponding powers of 2 for the binary prefixes: • kibi, Ki - 210 • mebi, Mi - 220 • gibi, Gi - 230 • tebi, Ti - 240 Know the names, symbols and corresponding powers of 10 for the decimal prefixes: • kilo, k - 103 • mega, M - 106 • giga, G - 109 • tera, T - 1012		

4.5.4 Binary number system

4.5.4.1 Unsigned binary		
Know the difference between unsigned binary and signed binary.		
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		
Be able to: • add two unsigned binary integers • multiply two unsigned binary integers.		
4.5.4.3 Signed binary using two's complement		
Know that signed binary can be used to represent negative integers and that one possible coding scheme is two's complement.		
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		
Know how numbers with a fractional part can be represented in: • fixed point form in binary in a given number of bits.		
Be able to convert from: • decimal to binary of a given number of bits • binary to decimal of a given number of bits.		
4.5.4.5 Rounding errors		
Know and be able to explain why both fixed point and floating point representation of decimal numbers may be inaccurate. Some values cannot ever be represented exactly, for example 0.1_{10}.		
4.5.4.6 Absolute and relative errors		
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		
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		
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		
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		
Differentiate between the character code representation of a decimal digit and its pure binary representation.		
4.5.5.2 ASCII and Unicode		
Describe ASCII and Unicode coding systems for coding character data and explain why Unicode was introduced.		
4.5.5.3 Error checking and correction		

Describe and explain the use of: • parity bits • majority voting • check digits.		

3.5.6 Representing images, sound and other data

4.5.6.1 Bit patterns, images, sound and other data		
Describe how bit patterns may represent other forms of data, including graphics and sound.		
4.5.6.2 Analogue and digital		
Understand the difference between analogue and digital: • data • signals.		
4.5.6.3 Analogue/digital conversion		
Describe the principles of operation of: • an analogue to digital converter (ADC) • a digital to analogue converter (DAC).		
4.5.6.4 Bitmapped graphics		
Explain how bitmaps are represented.		
Explain the following for bitmaps: • resolution • colour depth • size in pixels.		
Calculate storage requirements for bitmapped images and be aware that bitmap image files may also contain metadata.		
Be familiar with typical metadata.		
4.5.6.5 Vector graphics		
Explain how vector graphics represents images using lists of objects.		
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		
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		
Describe the digital representation of sound in terms of: • sample resolution • sampling rate and the Nyquist theorem.		

Calculate sound sample sizes in bytes.		
4.5.6.6 Musical Instrument Digital Interface (MIDI)		
Describe the purpose of MIDI and the use of event messages in MIDI.		
Describe the advantages of using MIDI files for representing music.		
4.5.6.7 Data compression		
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.		
4.5.6.8 Encryption		
Understand what is meant by encryption and be able to define it.		
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.		
Compare Vernam cipher with ciphers that depend on computational security.		

4.6 Fundamentals of Computer Systems

4.6.1 Hardware and software

4.6.1.1 Relationship between hardware and software		
Understand the relationship between hardware and software and be able to define the terms: • hardware • software.		
4.6.1.2 Classification of software		
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		
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)		
Understand that the role of the operating system is to create a virtual machine. This means that the complexities of the hardware are hidden from the user.		
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		
Show awareness of the development of types of programming languages and their classification into low-and high-level languages.		
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		
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 and object (executable) code.		

4.6.4 Logic gates

4.6.4.1 Logic gates		
Construct truth tables for the following logic gates: • NOT • AND • OR • XOR • NAND • NOR.		
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.		

4.6.5 Boolean algebra

4.6.5.1 Using Boolean algebra		
Be familiar with the use of Boolean identities and De Morgan's laws to manipulate and simplify Boolean expressions.		

4.7 Computer Organization and Architecture

4.7.1 Internal hardware components of a computer

4.7.1.1 Internal hardware components of a computer		
Have an understanding and knowledge of the basic internal components of a computer system.		
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.		
Understand the concept of addressable memory.		

4.7.2 The stored program concept

4.7.2.1 The meaning of the stored program concept		
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		
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.		

3.7.3.2 The Fetch-Execute cycle and the role of registers within it		
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.		
3.7.3.3 The processor instruction set		
Understand the term 'processor instruction set' and know that an instruction set is processor specific.		
Know that instructions consist of an opcode and one or more operands (value, memory address or register).		
3.7.3.4 Addressing modes		
Understand and apply immediate and direct addressing modes.		
3.7.3.5 Machine-code/assembly language operations		
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		
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		
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		
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		
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).		
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

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

4.9 Communication and Networking

4.9.1 Communication

4.9.1.1 Communication methods		
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		
Define: • baud rate • bit rate • bandwidth • latency • protocol		
Differentiate between baud rate and bit rate.		
Understand the relationship between bit rate and bandwidth.		

4.9.2 Networking

4.9.2.1 Network topology		
Understand: • physical star topology • logical bus network topology and: • differentiate between them • explain their operation compare each (advantages and disadvantages).		
4.9.2.2 Types of networking between hosts		
Explain the following and describe situations where they might be used: • peer-to-peer networking • client-server networking.		
4.9.2.3 Wireless networking		
Explain the purpose of WiFi.		
Be familiar with the components required for wireless networking.		
Be familiar with how wireless networks are secured.		
Explain the wireless protocol Carrier Sense Multiple Access with Collision Avoidance (CSMA/ CA) with and without Request to Send/Clear to Send (RTS/CTS).		
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		
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 '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		
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.		
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		
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		

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.		
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		
Know that an IP address is split into a network identifier part and a host identifier part.		
4.9.4.4 Subnet masking		
Know how a subnet mask is used to identify the network identifier part of the IP address.		
4.9.4.5 IP standards		
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		
Distinguish between routable and non-routable IP addresses.		
4.9.4.7 Dynamic Host Configuration Protocol (DHCP)		
Understand the purpose and function of the DHCP system.		
4.9.4.8 Network Address Translation (NAT)		
Explain the basic concept of NAT and why it is used.		
4.9.4.9 Network Address Translation (NAT)		
Explain the basic concept of port forwarding and why it is used.		
4.9.4.10 Client server model		
Be familiar with the client server model.		

Be familiar with the Websocket protocol and know why it is used and where it is used.		
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.		
Compare JSON (Java script object notation) with XML.		
4.9.4.11 Thin- versus thick-client computing		
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		
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, ...).		
4.10.2 Relational databases		
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		
Normalise relations to third normal form.		
Understand why databases are normalised.		
4.10.4 Structured Query Language (SQL)		
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		
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		
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.		
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		
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		
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		
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		
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)* *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		
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 → B* *g: B → C* *function g ○ 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 ℝ, and f(x) = (x + 2) and g(y) = y^3. Then* *g ○ 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		
Show experience of constructing simple programs in a functional programming language.		
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.*		
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		
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 Software Development (Project)

4.13.1 Aspects of software development

4.13.1.1 Analysis		
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.		
4.13.1.2 Design		
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.		
Be aware that design can be an iterative process involving a prototyping/agile approach.		
3.3.1.3 Implementation		
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.		
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.		
3.3.1.4 Testing		
Be aware that the implementation must be tested for the presence of errors, using selected test data covering normal (typical), boundary and erroneous data.		
It should also undergo acceptance testing with the intended user(s) of the system to ensure that the intended solution meets its specification.		
3.3.1.5 Evaluation		
Know the criteria for evaluating a computer system.		