CAMBRIDGE

Brighter Thinking

COMPUTER SCIENCE

A/AS Level for WJEC/Eduqas

Teacher's Resource Component 1

Christine Swan and Ilia Avroutine

Cambridge Elevate includes a number of features that enables you to customise this resource:

- You can re-order the contents of this resource through Cambridge Elevate, to suit your teaching sequence.
- You can create your own notes (highlights, annotations, voice notes) to help build information within the resource.
- You can also add hyperlinks between sections within the resource, to sections within other resources within Cambridge Elevate, or to web pages.

You can find more information about the functionality available to you within Cambridge Elevate and how you can use it in your teaching within the Cambridge Elevate Student and Teacher User Guide, which is available from your Home screen.

The Cambridge A/AS Level Computing for WJEC/Eduqas suite of resources covers the full AS and A Level specifications for the **Eduqas (England)** and **WJEC (Wales)** qualifications. This outline explains how to use these resources to teach the different pathways.

A/AS Level Computer Science for WJEC/Eduqas Component 1

This product is designed to prepare students for written examinations. It covers **Component 1 of the Eduqas A Level specification (England)** and **Component 3 of the WJEC A Level specification (Wales)**. It also covers Components 1 and 2 of the **Eduqas AS Level** and **WJEC AS/A Level** specifications.

A/AS Level Computer Science for WJEC/Eduqas Component 2

This product is designed to prepare students for written examinations. It covers **Component 2 of the Eduqas A Level specification (England)** and **Component 4 of the WJEC A Level specification (Wales)**. It also covers Components 1 and 2 of the **Eduqas AS Level** and **WJEC AS/A Level** specifications.

A/AS Level Computer Science for WJEC/Eduqas Teaching Programming

This product is designed to help teachers prepare students for on-screen and practical examinations/coursework. It can be used for **Component 3 of the Eduqas A Level specification (England)** and **Component 5 of the WJEC A Level specification (Wales)**. It can also be used for Component 2 of the **Eduqas AS Level** and **WJEC AS/A Level** specifications.

In order to teach A Level in England or Wales, you will need all three of these products.

AS Level

The AS Level specifications for England and Wales are identical. In order to teach AS Level in England or Wales, you will need all three of these products. However, the content in these products is presented in the order in which it is studied at A Level in England and A2 in Wales, so you will need to approach it in a slightly different order to teach AS Level.

Here's how we suggest you approach the AS Level content:

1. **Hardware and communication**: Component 2 Chapter 1
2. **Logical operations**: Component 1 Chapter 2
3. **Data transmission**: Component 2 Chapter 2
4. **Data representation and data types**: Component 2 Chapter 3
5. **Data structures**: Component 1 Chapter 1
6. **Organisation and structure of data**: Component 2 Chapter 4
7. **Databases and distributed systems**: Component 2 Chapter 5
8. **The operating system**: Component 2 Chapter 6
9. **Algorithms and programs**: Component 1 Chapter 3
10. **Principles of programming**: Component 1 Chapter 4
11. **Systems analysis**: Component 1 Chapter 5
12. **Software engineering**: Component 1 Chapter 7
13. **Program construction**: Component 2 Chapter 8
14. **The need for different types of software system and their attributes**: Component 2 Chapter 7
15. **Practical programming**: Component 3
16. **Data security and integrity processes**: Component 2 Chapter 8
17. **Economic, moral, legal, ethical and cultural issues relating to computer science**: Component 1 Chapter 9

If you're co-teaching AS and A Level in England, we'd suggest that you approach the content in this way for the first year of teaching, and then review it when you teach the added A Level content in the second year.

Chapter 1: Data structures

LEARNING OBJECTIVES

- Describe, interpret and manipulate data structures including arrays (up to three dimensions), records, stacks, queues, trees, linked lists and hash tables.
- Describe the manipulation of records and arrays.
- Represent the operation of stacks and queues using pointers and arrays.
- Represent the operation of linked lists and trees using pointers and arrays.
- Select, identify and justify appropriate data structures for given situations.

What your students need to know

- Data structures enable related data to be efficiently organised in a variety of formats depending on the application and memory requirements of a particular task. Structures can either be static (the size of the structure doesn't change during the program run) or dynamic (the size might change during a run). Although more efficient with regard to space requirements, dynamic structures are harder to program and are less predictable. They present a greater challenge therefore for the trainee programmer.
- The most common data structures are: one- or two-dimensional (2D) arrays (three-dimensional (3D) arrays are also used); linked lists; stacks; queues; binary trees and hash tables.
- Complex data structures are often simple data structures arranged one inside the other (nested). Binary trees are implemented with linked lists and pointers, while stacks can be implemented in an array (or natively supported by a central processing unit on a hardware level).
- Most data structures share the same operations for the elements inside: inserting; deleting; sorting and searching. The efficiency of these operators depends upon a good match between the structure and the task.
- One-dimensional arrays (called 'lists' in Python) were covered at GCSE. Each variable inside an array is called an element and each element's position in the array is known as its index. Arrays make it easy to change or assess the data inside them. Each element can be directly referred to by its index (e.g. **Names[6]**). This syllabus uses indexes starting from one but many programming languages (and algorithms) use indexes starting from zero.
- Two-dimensional arrays are like tables, with rows and columns. They have a double index, one for row, the other for column (e.g. **Names[6][3]**). Three-dimensional arrays just carry on with this pattern by requiring three indices (e.g. **Names[6][3][9]**).
- Most spreadsheets can be described as three-dimensional. A location of a cell is given by its sheet number, column, row (e.g. **'=Sheet2!C10'**). An array of x dimensions can be thought of as an array where each element is an array of x–1 dimensions.
- In most programming languages arrays are static; their size is predefined at the beginning of the program. Adding new data is accomplished by finding an empty index and putting data in it. That involves searching. Deletion is likewise accomplished by searching for the required element and then setting its value to null. In Python, lists / arrays are collections of objects and are dynamic. Data can be either inserted at any index, or appended at the end of the list, but that is unusual.
- A typical algorithm uses an array of a fixed size. When that array gets full, a new array is created that is double the size of the original and the contents are copied into it. Then the initial array is disposed of.
- Students need to know either pseudocode or actual code snippets to enable them to insert new data into an array of one, two or three dimensions.
- Records are similar to arrays in that they hold related data items. Unlike arrays, records can contain data items ('fields') of more than one data type and, in most languages you can retrieve fields by a field name rather than by index, e.g. room.price.
- Big O notation describes the changes in the performance (speed) of an algorithm (e.g. sorting or searching) as the volume of data to process increases.
- Big O notation assumes the worst case scenario (e.g. the search item is located at the last list element

or that elements are not pre-sorted). Searching can't use any of the shortcuts that are available when material is only somewhat sorted.

- A process that takes the same time to complete for any data size is said to be O(1), (e.g. reading the first element of an array). If the length of time to complete this process / algorithm is directly proportional to the data size, it is said to be in O(***n***) (e.g. a simple serial search that looks at every element). If the length of time the process takes is proportional to the square of the amount of data it is said to be O(n^2) and this is common for nested loops.
- All algorithms and data structures have a known Big O efficiency rating. Similar to the fact that all housing in the UK is now rated for electrical efficiency. The Big O rating is needed so that there is the best match between a data structure and algorithm for a task. However, there is often a trade-off required. The most efficient algorithms can be the most complex and most difficult to implement, so when you know that ***n*** is going to be small, choosing a simpler, less efficient algorithm might, in fact, be the right thing to do.
- Linked lists allow for efficient sorting without moving the data itself as just the pointers are rearranged. The performance of adding / removing nodes is more significant. Sorting a linked list won't be significantly faster than sorting an array, because swapping elements with a temporary variable is quick to do.
- Linked lists are very common and very dynamic. They acquire new 'nodes', referred to as elements. Each node contains data and is linked to both the previously added node and the node added after via pointers. Like clues for a treasure hunt, pointers indicate which node is next and which was previous. Students need to know the algorithm for adding and removing data to/from a linked list. Simpler linked lists, with just one pointer to the next node, are also common.
- Stacks are Last-In, First-Out (LIFO) structures. The last added (pushed) element is the first to be taken off (popped). They can be implemented either with arrays or with linked lists.
- Queues are First-In, First-Out (FIFO) structures. Elements can only be processed in the order that they were added to the queue. They are easily implemented using an array and two variables for the front and back of the queue (which happens to be the next free space).
- Binary trees are composed of nodes (called leaves) and links between them (called branches). In a binary tree, each node has two child nodes. Binary search trees are a special case of binary trees where the data on the left branch of a node must be less than the data in the node, while the data on the right must be greater in order that they are sorted. Binary trees are multidimensional, so there are more ways to iterate or traverse the tree. The type of traversal (running through a tree) will determine the order of the results.
- A subtree is a tree that is actually part of a larger tree.
- There are three types of traversal.
  - Pre-order traversal (depth-first search) – start at the root node, traverse the left subtree, and then traverse the right subtree.
  - In-order traversal– traverse the left subtree, then visit the root node, and then traverse the right subtree.
  - Post-order traversal – traverse the left subtree, then traverse the right subtree, and then visit the root node.

Traversals are implemented using an array or a linked list (this is better as it is dynamic and node-based).

- In addition to traversals, students need to know how to implement a tree with a 2D array, as well as the code for adding and removing nodes to and from a binary tree. Special cases with a single branch and single leaf should also be considered.
- Hash tables contain data and mapping functions to determine which data is moved into which 'cell'. The hashing function takes an attribute of a data element (e.g. the length of a string entered) then performs a modular division on it by the size of the table. The remainder generate the string's index number. Although this process enables faster placement / storage of elements, the function is not very aware of the nature of the data it stores. It is possible therefore for multiple elements to be assigned to the same address / place which is then called 'data collision'. To overcome this issue, linked lists are implemented inside a hash table to add nodes where two elements are 'double-booked'. This process, called separate chaining, avoids data collisions but it does lead to slower operation speeds.

Common misconceptions

- Students often confuse stacks with queues and expect both to behave like arrays, which they can't.
- For queues and stacks, reading data is also deleting it which is not an obvious concept for students to grasp.
- Students often mix up the tree traversal algorithms; they forget how binary search trees are sorted.
- Hash tables do not have indices and students need to be reminded of this fact.
- There is not just one, but multiple appropriate data structures that can solve a problem.

Lesson activity suggestions

- This is a very hands-on topic, so practical demonstrations may prove a useful way to support learning. For example, students arranged in a line could literally 'point' at each other in order to represent the steps needed to add and remove nodes from a linked list. As pointers are a tricky concept to understand, such an activity may help your students to visualise what's going on. Lessons should also include a mixture of addressing past exam questions with actual programming data structures on a computer.
- Programs can be more easily understood by students when the data involved are relevant to everyday life. For example, a program designed to implement a queue could be designed to handle customers at a furniture superstore waiting to return products. Students will more readily relate to this experience than if abstract data sources are used. Such a program involves entering a new customer into a queue and then removing them when they are served. Alternatives might include foreign passport appointments or landing patterns for aeroplanes.
- Stacks can be made more relevant by storing a list of actions that are undone. Repairing a laptop, for example, requires a lot of disassembly and it is hard to remember the sequence of actions. A program could be made that keeps track of all the actions (e.g. remove power supply, undo the screws holding the keyboard ribbon cable, etc.). Then, when the laptop is being put back together, the program should return all actions but in the opposite order.
- A linked list program could perhaps revolve around a secret society in which members start to disappear and for which links need to be restored (e.g. the link that pointed previously at the node 'treasurer' will have to point at somebody else now).
- In order to explain binary trees, you could use data from a role-playing game (e.g. Dungeons and Dragons) where, depending on your previous decision (e.g. to enter a room or to challenge a dragon), you get two additional choices.
- Hash tables may be more easily understood if the students get to implement more than one hashing algorithm and then measure the number of data collisions to help them choose the 'best' algorithm for the job.
- For arrays it may be useful to view these as if they were a shopping list (e.g. items are added and removed in no particular order as a person moves around the store and runs into the items on the list).

Topic: All topics

Look at some past paper questions.

Implement data structures in actual programs using the student's choice of programming language.

Follow-up ideas/Homework

Practise more past paper questions.

Extensions of programs could be assigned as homework (e.g. better interface, storage to CSV files).

Topic: Big O notation

Starter:
Show students this **animation** which illustrates multiple sorting algorithms. Ask your students why a slower algorithm might be used.

Main activity:
Implement a bubble sort of students by height. Initially use only half of the class and time the sorting. Then use the whole class and time the sorting. If the time to sort between the two attempts increases

proportionally by more than one to one, we can calculate the Big O value, which would be expected to be $O(n^2)$ for a bubble sort.

Plenary:
Discuss with the class whether, or not, Moore's law is making Big O ranking of algorithms less relevant.

Follow-up ideas/Homework

Ask your students to use a timestamp feature in a programming language (e.g. time.time() in Python) to implement a sorting algorithm, saving timestamps to beginning and end variables, respectively. This should encourage a discussion of how long a procedure takes with different sets of data and enable Big O ratings to be calculated.

Topic: Stacks and queues

Starter:
Present students with the following paired scenarios:

(1) A waiting list versus a queue at the canteen.

(2) A stack of unwashed dishes versus re-parking three cars on a narrow driveway that is only one car wide.

Ask students why they think these scenarios were paired and what do they have in common?

Main activity:
To continue work looking at the definition of stacks and queues, ask students to work out how they might act out the algorithms for adding a new item to a stack or queue using themselves as data elements (a queue is obviously the easiest and the most common sense to implement).

Plenary:
Discuss which of stacks and queues requires less processing power and why.

Follow-up ideas/Homework

Queues – ask students to write a program that could run in a self-help kiosk at a custom service centre. The program should keep track of people in a queue by allowing users to receive waiting list numbers; the program should then process these numbers at random time intervals and then retire the numbers that have been 'served'.

Stacks – ask students to write a program that will accept a user's input of a formula involving brackets and parse this input to execute the formula (e.g. $(4 + 6) \times 3$).

Topic: Hash tables

Starter:
Illustrate the use of dictionaries in either Python or JavaScript, associative arrays in the Visual Basic family of languages, and maps in C++ or C#'s HashSet.

Make clear to your students that dictionaries can't be sorted and discuss why this is the case. [*Answer*: dictionaries internally perform calculations on the keys to come up with addresses for values. This is unlike lists or arrays where items are stored by sequential and unique indices which can be sorted]

Main activity:
Work out the examples of hash tables given in a book. Students should be encouraged to try different formulas, more complex than the ones given (e.g. for every name in a list to be placed into a hash table, find the ASCII value for every letter and use the sum of ASCII values of all letters in a name MOD the size of the table).

Students should be challenged to come up with the best algorithm which places a list of words into slots in an array. The 'best' solution will involve fewer collisions and the most uniform distribution of names across the table.

Plenary:
Instead of using pen and paper, try a more practical demonstration of hashing using the student seating plan in the classroom

Complete these steps:

(1) Assign each seat in the classroom an index.

(2) Ask a student or group of students to write an algorithm for where each student should sit.

(3) All students then stand at the front of the class.

(4) The student(s) who wrote the algorithm then instruct the students where to sit using some information about them (e.g., their surname) as the input of a hash algorithm.

(5) If there are no collisions, the number of seats available should be reduced, until the first collision happens.

(6) Ask another student or group of students to write a new algorithm and repeat the steps.

Follow-up ideas/Homework

Students should be asked to try to implement the exercises in the following **book** by Zed A. Shaw.

There are a number of available websites that claim to shorten long urls (e.g. **tinyurl** and **goo**). They often feature on Twitter where word limit is a consideration. Students should investigate this technology.

Topic: Binary trees and linked lists

Starter:
Introduce the concept of binary trees by looking at a role-playing game (e.g. Dungeons and Dragons). During such games, depending on a player's previous decision (e.g. to enter a room or to challenge a dragon), they then get two additional choices.

Main activity:
Students should write down the menus in an ATM as a binary tree of choices made at a previous screen. Discuss as a class the type of traversal which is typical for using ATM menus.

Plenary:
Split the students into two groups. The first group should come up with a binary, while the other group prepares a linked list. The two groups should then swap their outputs, convert between the two forms and see how they are inter-related.

Follow-up ideas/Homework

Provide students with a genealogical tree of the UK Royal Family. This is not a binary tree as it has more than two branches at most nodes. The students should therefore create a linked list to show only the monarchs that have ruled.

End-of-chapter answers

1

Big O notation.

2

A collection of variables, all of the same type stored under a single identifier.

3

Data and a pointer to the next element in the list.

4

Queue.

Chapter 2: Logical operations

LEARNING OBJECTIVES

- Draw truth tables for Boolean expressions consisting of AND, OR, NOT, XOR, NAND and NOR logical operations.
- Apply logical operations to combinations of conditions in programming, including clearing registers, masking, and encryption.
- Simplify Boolean expressions using Boolean identities, rules and de Morgan's laws.

What your students need to know

- A program involves a series of decisions which are dependent on the data provided by users in order to decide which lines of code run and when, in order, to perform operations.
- Most of these decisions can be neatly summarised in tables representing different operators of Boolean algebra: AND, OR, NOT, XOR, NAND and NOR logical operations.
- The operation AND describes serial circuits where both conditions have to be TRUE or ON for the electric signal to complete a circuit.
- The operation OR is a parallel circuit, so that either input has to be ON, for the output signal to be ON.
- The operation NOT reverses the signal between ON and OFF.
- The operation XOR is looking at two inputs and checks if they are different. If the inputs are different then the output signal will be ON.
- The operation NAND is the opposite of AND, blocking the signal if both inputs are ON and setting the output signal to OFF, otherwise the output signal is ON.
- The operation NOR is the opposite of OR, if at least one of the input is ON, the output is OFF.
- Masking is a technique that enables you to look at the value of a selection of data from within a larger data unit. Masking of a byte is analogous to painters using masking tape to create a boundary to an area when painting. The paint cannot go under the tape, which leaves a clean and straight edge. Similarly, by applying an AND operation to all bits in a byte with a binary number where all bits are zero except the one opposite the bit we are interested in – we can isolate it as all the other bits 'AND' to zero.
- Masking is often used to clear registers. This means resetting all bits to OFF (which is zero). Using either AND with the original data with an all-zero byte, or XOR for the original data with itself will achieve the same result. The advantage of using XOR over AND for this purpose is you don't have to store an all-zero byte, you can just reuse the original data.
- XOR is also used in encryption. By XORing original data with another binary number, called a 'key' as in 'encryption key', the output is a seemingly random unrelated ciphered data. However, if you XOR this new data with the key, you will get your original data back which is how it can be used in decryption.
- Propositional logic is defined in terms of True and False and follows mathematical rules. Most of the time a proposition could state that A > B (which can be either True or False), while another could state B > C (again, can be either True of False). If both of them are true, we can say that A > C must be True.
- Propositions are labelled with capital letters, like Q or P, and use the connective symbols for AND (conjugation $\wedge$), OR (disjunction V), NOT (negation ¬), IF (implication →) and equality (biconditional equivalence ↔). Using this, we evaluate two input propositions (one for NOT) and can generate an output proposition. Each of these comes with a table of possibilities that will need to be understood and recalled.
- Conjugation (AND) requires that both input propositions are True. If either one is False, the output is False. It is equivalent to a simple multiplication; for example, True AND False = False is the same as $1 \times 0 = 0$, if we hold True to be 1 and False to be 0.
- Disjunction (OR) requires that at least one of the input propositions is True and is equivalent to a simple addition; for example, True OR False = True is the same as $1 + 0 = 1$, if we hold True to be 1 and False to be 0.
- Negation is a unary operator that works only on one proposition.

- A tautological proposition is the one that outputs True regardless of what truth values of input propositions are, for example X AND NOT(X) (xV¬x).
- Biconditional equivalence means 'if and only if' both sides (input propositions) are True, the output is True.
- Using biconditional equivalence we can prove that some propositions are tautological and can be omitted. The logical chains can be shortened as a result.
- Logical laws of deduction include: commutation – the order of propositions for both conjunction and disjunction do not matter and they can be interchanged and are biconditional; association – the statements in brackets are evaluated first; distribution – we can move letters inside the brackets and outside the brackets, provided we do it the right way (the order of brackets matters), (e.g. T AND (P OR Q) = (T AND P) OR (T AND Q)).
- Double negation is the same (tautological) as no negation. With NOT, just as −(−2) is 2, NOT(NOT P)↔P.
- de Morgan's law states that the negation of disjunctions is the conjunction of the negations (NOT(P OR Q) = NOT(P) AND NOT(Q)) and that the negation of conjunction is the disjunction of the negations (NOT(P AND Q) = NOT(P) OR NOT(Q)). This is significant because it allows us to transform AND into OR and vice versa for the purpose of simplifying long logical chains.
- Plain English propositions can be converted into the language of Boolean algebra by paying attention to 'command words' (AND, OR, must, could, all, some, etc.) inside these propositions.
- A bi-directional condition occurs when you can reverse the implication and still have a True statement.
- You can use Boolean algebra to simplify complex conditions for control statements such as WHILE loops or IF statements.

Common misconceptions

- Students can get confused by abbreviations and as a result often feel they need to memorise rule tables. However, knowing the first principles of every rule should enable them to construct a table easily for themselves, especially as they gain confidence.
- Some of the terminology in this topic is quite complex and it can be easy to confuse the terminology from Boolean algebra in particular. Students not exposed to Decision mathematics, may be rather unfamiliar with the terms initially.
- Students often think that de Morgan's laws are difficult. However, they are similar to the basic mathematics distribution principle.
- Students may believe that logic is not for everybody and requires a certain mindset. However, any one can complete a logic chain applying the rules of Boolean operations explained in this Chapter.

Lesson activity suggestions

Topic: How to define problems using Boolean algebra

Starter:

Review the book *The Lady, or the Tiger?* by Frank R. Stockton.

Main activity:

In a Court of Law, it is essential to establish guilt by finding evidence of motive and means to commit a crime. Ask your students to come up with a list of cases where neither, both or either of these conditions is met to illustrate AND and NOT. Ask students to follow a similar line of reasoning to illustrate OR. What life situation would help to describe OR?

Plenary:

Discuss as a class what is common between: Boolean OR and arithmetic addition; Boolean AND and arithmetic multiplication?

Follow-up ideas/Homework

Raymond Merrill Smullyan was a very important person in the development of popular logic thinking. Ask your students to complete their own research into his career, his connection to Alan Turing and look at his logical puzzle collections.

Topic: To derive or simplify Boolean algebra

Starter:

Ask students to come up with the most difficult Boolean expression under 20 elements. Then the class can identify the three most difficult of those submitted.

Main activity:
Ask students to do exercises that illustrate de Morgan's laws.

This could be done in a practical way where one student uses other students to act out a logic gate. Students holding hands represent an 'ON' circuit and when hands are not held, the circuit is 'OFF'. Please note that if any students aren't comfortable holding hands, they could hold the ends of pencils instead.

Plenary:
Discuss the practical applications of de Morgan's laws.

Follow-up ideas/Homework

Students should work as 'creator–editor' pairs to create problem sets that cover each of the seven Boolean operators and optionally, some of the simplification rules.

Topic: Stretch lesson on Logical laws of deduction

Starter:
Demonstrate to the class a feature of Python iterables, that of **sets**.

Sets are not quite lists, they must contain unique elements and are perfect for running through logical operators (e.g. the **isdisjoint()** operator).

Main:
Students should then construct SQL queries that use multiple criteria and nested SQL statements (e.g.

```
'SELECT Name,(score1+score2) FROM Pupils WHERE (score1+score2)=(SELECT
MAX(score1+score2) FROM Pupils) AND (score1<80 OR score2<40)).
```

The student that uses the biggest number of Boolean operators (and produces a meaningful result) is the winner.

Consult the following link **here** or **here** for further information / ideas.

Plenary:
Discuss as a class how you could explain to a much younger student the concept of disjunction.

Follow-up ideas/Homework

Students should write a program that illustrates *all* of the logical operations in this chapter, performing them on individual bits of a byte. The program should have a menu giving a user a chance to enter one or more binary 8-bit numbers and then to see the results of applying the following operations to these two bytes:

AND, OR, XOR, NOR, NAND, NOT (just one number needed for that).

End-of-chapter answers

1

de Morgan's law states that the negation of a conjunction is the disjunction of the negated items. $\neg(P \wedge Q)$ is the negation of a conjunction. So if P and Q were both true, then the final result would be false. This is equivalent to $\neg P \vee \neg Q$ as $\neg P$ is false and $\neg Q$ is also false. The disjunction of two false values is also false. The truth table below shows their equivalence.

P	Q	¬(P ∧ Q)	¬P ∨ ¬Q
0	0	1	1
1	0	1	1
0	1	1	1
1	1	0	0

2

(a) Conjunction uses the ∧ symbol and both elements must be true in order for the final value to be true.

(b) Disjunction uses the ∨ symbol and either elements can be true for the whole statement to be true.

(c) Implication uses the → symbol and means that if the equation on the left is true then the equation on the right must also be true.

(d) Negation uses the ¬ symbol and will swap a truth value around.

3

$\neg(A \vee \neg B) \equiv \neg A \wedge \neg\neg B$ de Morgan's law

$\neg A \wedge \neg\neg B \equiv \neg A \wedge B$ Double negation

$\neg A \wedge B \equiv B \wedge \neg A$ Law of commutation

4

(a) $\neg P \rightarrow \neg T$

(b) $P \wedge Q \rightarrow T$

(c) $Q \wedge P \wedge R \rightarrow \neg T$

5

P	Q	P → Q	P ↔ Q
0	0	1	1
1	0	0	0
0	1	1	0
1	1	1	1

6

In packet switching, the packets will arrive in a different order while circuit switched networks packets will always arrive in order.

In a circuit switched network, if part of the line breaks then communication will fail while packet-switched networks will find alternative routes.

7

3 / 8 = 0.375 Mb/s

5 / 0.375 = 13.3 seconds

Chapter 3: Algorithms and programs

LEARNING OBJECTIVES

- Explain the term algorithm and describe common methods of defining algorithms, including pseudocode and flowcharts.
- Identify and explain the use of constants and variables in algorithms and programs.
- Describe why the use of self-documenting identifiers, annotation and program layout are important in programs.
- Give examples of self-documenting identifiers, annotation and appropriate program layout.
- Describe the scope and lifetime of variables in algorithms and programs.
- Explain the purpose and effect of procedure calling, parameter passing and return, call by reference and call by value.
- Explain the use of recursion in algorithms and programs and consider the potential elegance of this approach.
- Identify, explain and use mathematical operations in algorithms, including DIV and MOD.
- Identify, explain and apply appropriate techniques of validation and verification in algorithms and programs.
- Explain the need for a variety of sorting algorithms both recursive and non-recursive.
- Describe the characteristics of sorting algorithms: bubble sort, insertion sort, quicksort.
- Explain the effect of storage space required, number of comparisons of data items, number of exchanges needed and number of passes through the data on the efficiency of a sorting algorithm.
- Use Big O notation to determine the efficiency of different sorting algorithms in terms of their time and space requirements and to compare the efficiency of different sorting algorithms.
- Explain and apply a linear search algorithm.
- Explain and apply the binary search algorithm.
- Explain and apply a shortest-path algorithm.
- Describe appropriate circumstances for the use of each searching technique.
- Use Big O notation to determine the efficiency of linear and binary searches in terms of execution time and space requirements and to compare the efficiency of different searching algorithms.
- Follow search and sort algorithms and programs and make alterations to such algorithms.
- Write search and sort algorithms and programs.
- Identify, explain and use sequence, selection and repetition in algorithms and programs.
- Identify, explain and use counts and rogue values in algorithms and programs.
- Follow and make alterations to algorithms and programs involving sequence, selection, and repetition.

Learning objectives (continued)

- Write algorithms and programs involving sequence, selection, and repetition to solve non-standard problems.
- Identify and explain the nature, use and possible benefits of standard functions, standard modules and user defined subprograms.
- Identify, use and explain the logical operators AND, OR, NOT, XOR, NAND and NOR in algorithms and programs.
- Write and interpret algorithms used in the traversal of data structures.
- Explain data compression and how data compression algorithms are used.
- Compare and explain the efficiency of data compression algorithms in terms of compression ratio,

compression time, decompression time and saving percentage.

- Select appropriate test data.
- Dry-run a program or algorithm in order to identify possible errors, including logical errors.
- Explain the purpose of a given algorithm by showing the effects of test data.
- Use Big O notation to determine the complexity and efficiency of given algorithms in terms of their execution time, their memory requirements and between algorithms that perform the same task.

What your students need to know

- All of the standard flowchart shapes (e.g. input; output; processing; decisions, etc.).
- That pseudocode is language-agnostic. It should not contain any specific features of programming languages (e.g. no curly brackets or 'elif' instead of ELSE IF). Students should also be made aware of the various 'dialects' of pseudocode. These are fine to use, as long as a student is consistent (e.g. if they decide to use the 'SET *x* TO 5' format they should not combine it with another format such as '*x* < –5').
- The programming languages Pascal and Scratch use code that is very similar to standard pseudocode so could be used to scaffold pseudocode creation if a student is struggling.
- Pseudocode needs to be written ***before*** actual code. At the pseudocode stage, the programming language hasn't been chosen (students should remember that a professional would be expected to select the most appropriate programming language for a job).
- Every program needs to consider the following: computer memory; storage of values; overwriting; use of labels to interact between different subprograms.
- References to memory locations are known as identifiers. They include ***variables*** (values that change as a program runs), ***constants*** (values that don't change, such as the SPEED_OF_LIGHT – traditionally capitalised), and ***procedure / function*** names.
- A complex program may have many identifiers so it is considered good practice to name them in descriptive ways (e.g. '**row_counter**' instead of 'j', '**multiplier**' instead of 'num2', etc.).
- The value of a variable is code for something. For example, –999 is often used as a 'rogue value' to indicate the end of data. In the list: '34, 67, 2, 78, –999', it is clear, that –999 doesn't belong in that list and doesn't represent what the other numbers are standing for, in this case the number of cars passing a junction per minute.
- As soon as a value is no longer needed its memory location is released (also known as 'garbage collected'). Subprograms, functions and procedures all finish before the main program, so their variables are released early. They are called ***local variables*** as the rest of the program doesn't know about them. ***Global variables*** are variables or constants that persist for the duration of a program and are therefore visible to all parts of a program.
- The scope of a variable requires an understanding of when that variable is no longer required and becomes unavailable to the rest of a program.
- Procedures are chunks of reusable code. They increase the efficiency of a program as they allow more tasks to be completed but require fewer lines of code. When a particular operation needs to happen more than once within a program, it is possible to first define it and then create a link to it. This is similar to the way in which people share information available on the internet; information from a site isn't copy and pasted but, a link to the original site is provided instead.
- Parameter passing enables procedures and functions to be made more efficient. They can run differently within the same program using special local variables configured to receive data from outside of the procedure or function. Parameters behave just like other local variables, except they are not declared on a separate line but rather in the function definition line. Functions, unlike procedures, can also return values which might flow into the parameters of another function or procedure.
- Recursion takes the use of functions to another level and allows us to replace iteration with selection. Instead of running a loop inside a procedure (an 'iterative' solution), we get the procedure to call itself until the terminating condition is True. Recursion, while harder to write than iteration, is great for repetitive code such as drawing fractals or traversing binary trees, replacing nested loops, as it results in fewer lines of code.
- Sorting and searching algorithms are often repetitive, so the faster ones have to utilise recursion. Recursive solutions have an equivalent iterative solution but they are not as elegant or natural than

when in the recursive form.

- Operators such as MOD allow us to test values for certain properties (e.g. whether a value is odd or even, or if it is a prime number).
- Validation and verification are techniques to make programs 'fool proof' and avoid errors of all types. Validation is done through algorithms, while verification usually involves manual checking or the use of an external reference resource (e.g. a postcode database).
- A range of sorting and searching algorithms are needed because some work better on small data sets, whilst others are better for larger sets. Additionally some algorithms work better with sorted data (binary search) whilst others work best with unsorted data.
- Compression algorithms work differently for different types of data. There are also differences between lossy and lossless types of data. Computer storage space and network bandwidth are both limited. If the amount of file bytes can be reduced by removing unnecessary bits of data, more storage space is available and faster transmission speeds are achieved.
- The level of compression achieved is expressed as a compression ratio (size of compressed file:size of the original). The smaller the ratio, the better.
- A number of compression formats exist (e.g. zip, rar, tgz and 7-Zip). The compression performance of these formats is slightly different. For example, 7-Zip is open source and uses multiple algorithms to achieve very high compression ratios, as well as 256-bit encryption.
- Big O is a rating system that determines how fast an algorithm is and to assess the impact on algorithm speed of any change to the work required of it (e.g. if the work of an algorithm doubles will the time taken also double or will it quadruple?).

Common misconceptions

- Students often write programs, make sure they work and only then retrospectively write the pseudocode or create a flowchart. This is not good practice as it doesn't develop their planning skills.
- Students need to appreciate that ***rogue*** values are not meaningful values that have to be calculated.
- Students confuse validation and verification as they start and end with similar letters. They need to be reminded that validation is done entirely within an algorithm (e.g. checking if a value is in a certain range). Verification involves an external value (e.g. asking a password to be entered twice).
- Recursion runs code repeatedly but there is no loop, just selection. Students often forget whilst creating a program to put the terminating condition into their recursive functions. It is also important not to lose track of the states of variables passed recursive at each stage of recursion.
- It is common for students to be rather confused by the different sorting and searching algorithms that are available. It may therefore be sensible to introduce them separately at different times so that students don't attempt to learn them all in one go.
- Big O can be perceived as confusing but it is just a rating system. Remind students that this is similar to how all electrical appliances must now be rated in terms of their consumption of electricity.

Lesson activity suggestions

- Students will benefit from lots of practical experience creating pseudocode of algorithms. They should then be familiar with writing and running programs to implement algorithms in their preferred (or instructed) choice of programming language. They also need to be able to both identify errors and omissions in pseudocode or program code during an exam and be able to dry-run their programs using pen and paper before correcting and completing them.
- Advanced students will benefit from researching 'closures' and exploring their use in languages like Python and JavaScript.
- To help students understand the nature, use and possible benefits of standard functions, standard modules and user defined subprograms they should explore the Python installation folder. As Python code is completely open source and every part of Python is viewable in the plain text view, students can take a few files, (e.g. **random.py** and see how many functions it contains). Encourage them to copy the contents of **random.py** into another file with a different name and try to add their own functions to it. Then they can use the 'import' facility to use this code in their programs – this will help to illustrate the concept of standard modules and libraries.
- To help students become familiar with the logical operators: AND, OR, NOT, XOR, NAND and NOR,

students could be asked to write a login program requiring a username and pin code. The program should be able to validate both the username and pin for presence, data type, range, etc. The code should use as many of the logical operators as possible. An alternative exercise would be to ask the students to write a game of Pong using a suitable language with Canvas properties (e.g. JavaScript or Python Canvas/Turtle). Use of Python's Turtle module with its quadrant structure would provide an easy introduction to the use of NOT AND (NAND).

Topic: Develop a model algorithm, using flowcharts

Starter:
Explain to the students that an algorithm can be designed to be repeatable and predictable. The students should assess the inputs and outputs to an algorithm that they are going to design.

Main activity:
Create flowcharts to solve the mathematical equations using the following (suggested) steps:

(1) Assess the problem.

Students should be shown graphs of common functions (e.g. $y = x^2$, $y = \frac{1}{x}$, $y = \sqrt{x}$) and asked to identify them.

(2) Establish the stages of the solution
Students should establish which formulae are to be used to solve the problem and / or whether iterations (repeated application) of various processes are needed.

(3) Map out the solution algorithm
Students produce a flowchart of their proposed solution to the problem.

(4) Assess the algorithm
Student test their algorithms using dry runs, i.e. pretending to be a computer and finding values of variables at every step / iteration of the program.

Plenary:
Discuss how the students' programs ran and what problems they encountered.

Follow-up areas/Homework

Students create a flowchart for a game of Monopoly. As it is a complex game, they might want to take upon different respective features. This activity will particularly dwell on the concept of iteration which is sometimes tricky to show well in flowcharts.

Students need to create a flowchart-to-code cue cards. E.g. if the class are learning Python, all the Python commands learned so far (stretch: research those that haven't been learned, yet) in context on one side of the cue card and the flowchart symbol on the other. E.g. Python keyword try/except/else lends itself well to a combination of diamond flowchart shapes.

Create a flowchart for debugging a program. The most common mistakes should appear quite early in the flowchart.

Topic: Progress from flowchart to pseudocode

Starter:
Explain to the students that using pseudocode is the usual way that programmers progress a program from the flowchart stage. At this stage, it may not be known exactly which programming language is to be used.

Main activity:
Code the flowcharts from the last activity in pseudocode, using the following (suggested) steps:

(1) Create the main program which is going to take the inputs and call the various procedures as they are needed.

(2) Write each procedure as a separate entity for each task. This would work well as a team building exercise as the students would experience an aspect of a programmer's professional life where individual team members write different elements of a program.

(3) Assess the main program and the procedures. The students could take their other team member's work and try it out by acting as computers and 'running the code'.

(4) By Stage 3, the students (ideally) will run into problems because the domains have not been defined. If they don't detect the problems, they need to use numbers to test their program which will cause problems (e.g. require the square roots of negative numbers).

(5) Stop the development process (temporarily) and have a discussion in the plenary with your students.

Plenary:
Explain to the students that they will be continuing with the programming task but that another area of their understanding needs to be discussed prior to continuing. The next lesson will address the problem of domains.

Follow-up areas/Homework

Students create pseudocode-to-flowchart cue cards and flowchart-to-pseudocode ones, as well.

Create an imaginary programming language, based on the one the students know, but replace all the keywords with either (a) backward versions, e.g. tnirp instead of print in Python; (b) names of cartoon/Pokemon characters; (c) baby words/"LOLCat speak". Given either flowchart or pseudocode, students are to write a simple program in this language.

Given a program, learners create a flowchart/pseudocode. Good idea to comment when this might be used (Answer: when a software company takes over an old project for which the documentation has been lost, while the company needs to know how this will integrate with their own software and its logic.)

Topic: Identifiers: naming and scope, constants, and rogue values

Starter:
Suggest that you look around the classroom then point to objects and ask the students: 'If this object was part of a computer program, how would you describe it?'

Alternatively you might ask your students to discuss why we need passports to travel abroad and not for internal travel? What is it about the scope of the passport that makes it necessary for travel?

Main activity:
Provide a program which has non-descriptive identifiers, or annotations and then ask your students to fix the identifiers by naming them with descriptive names. Annotations should be supplied. A variation on this activity would be to supply the annotations but the lines of code are obscured or masked so that the students need to reconstruct them from the annotations provided.

Plenary:
Encourage a discussion of the problems encountered in the process of performing the tasks in the main activity.

Follow-up areas/Homework

Students should complete the necessary research to explain why the word 'over' is used in radio communications (e.g. when using walkie-talkies). They should then think about which aspect from this lesson is the computer version of saying 'over'. [*Answer*: it is rogue values.]

Topic: Parameter passing and recursion

Starter:
Begin by looking at the dictionary definition of the word 'parameter' in everyday life.

Then review some formulas and mathematical functions (e.g. $y(x) = x \times 2 - 5$). Discuss the purpose of the parameters that are within brackets. Review how the output of a function changes depending on which aspects are within or outside brackets.

Main activity:
Students should be provided with a program that has no global variables, just parameter passing. They should then modify the program to only use global variables.

Extensions to this activity include:

(1) To complete the opposite process, which is harder. Provide a program with global variables, and ask the students to remove these and replace them with parameter passing.

(2) Provide a recursive solution and ask the students to convert it to an iterative one. Then ask the students to complete the opposite process, i.e. given an iterative solution, convert it to a recursive one.

Plenary:
Compare the number of lines of code in a recursive versus an iterative solution and comment on the meaning of the word 'elegance' when applied to computer coding.

Follow-up areas/Homework

Students should examine their preferred programming language and look at how functions or procedures are created using arbitrary numbers for parameters. Students should observe, for example that a function that calculates the geometric mean of a series of numbers needs to be flexible enough to accept any number of inputs.

Please note:

(1) Students should not allow solutions which involve arrays / lists to be passed into a function. Although these are perfectly legitimate, they are not helpful in this particular exercise.

(2) Some programming languages, like C are not suitable for this exercise.

Topic: Mathematical and logical operations

Starter:
Present the rule for determining leap years which contains a nested selection and the use of the remainder division.

Students should convert the structured rule into pseudocode or a flowchart.

Main activity:
Students should then implement the program using their preferred choice of programming language. They should also review whether a recursive solution could be created.

Plenary:
Encourage a discussion of the problems encountered in the process of performing the tasks.

Follow-up areas/Homework

Ask students to review the computer game *Space Invaders*. In the game, aliens slide to the left a number of steps, then move down a row, then slide to the right a number of steps then move down a row. Can this solution be simplified using a remainder division?

Alternatively they should explore *Turtle*. In this program it is possible for the turtle character to turn left or right. If a turtle turned 1400 degrees to the left, how would it be possible using mathematical operators to decide which of the four quadrants it would face?

Topic: Validation and verification

Starter:
Discuss as a class what the students understand by the term 'fool proof'. What are 'typical errors' which users make? How do the students think errors can be prevented?

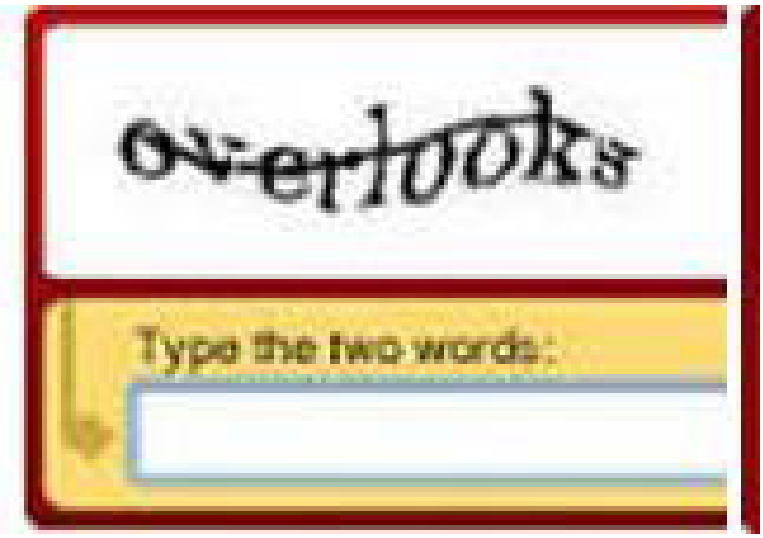
Main activity:
In pairs, one student should try and create a 'fool proof' program whilst the other tries to break it by playing the role of a user.

Plenary:
When a customer returns an item they have bought and requests a refund, the cashier often has to call a manager to authorise the transaction. Is such a situation an example of validation or verification? Encourage a discussion of other examples of verification in everyday life rather than computer programming.

Follow-up areas/Homework

Ask students to complete some research into *regular expressions*. The students should be able to demonstrate their use in conditional formatting (e.g. within Microsoft Excel) and / or in a program of their own creation. Which mistakes cannot be prevented by regular expression validation?

Alternatively, or additionally, students should research how **CAPTCHA** scripts work and the reasons for having them.



Topic: Sorting

Starter:
Explain the bubble sort algorithm. Students should then form a queue alphabetically by name. Following that they should then sort themselves by height using a pairwise comparison and swapping. If a student will have to move almost the whole length of the queue, (e.g. they are short in height but come last alphabetically by name), give them a card that says '*I'm a bubble*'. Once the sorting process is complete ask the students to review why the sort is called a bubble sort?

[*Answer*: upon observation, it appears that out-of-order elements are iteratively moving to their ultimate destination, like bubbles rising to the surface in liquid.]

Main activity:
Review the advantages and disadvantages of different sorting algorithms available. These **animations** could be a useful way to do this.

Plenary:
Hold a Q&A session with the class. The questions should describe a data set to be sorted. The students should then indicate which sorting algorithm should be used. Students should provide a justification for their selection.

Follow-up areas/Homework

Research the algorithm that their favourite language uses for sorting, this information is usually widely available.

Pupils can prepare a report about more than one language's sorting algorithm and comment how this affects the choice of a language for a particular language, noting the speed and memory requirements. E.g. SQL uses quick sort and merge sort.

Topic: Searching

Starter:
Encourage a discussion of how we search for items in everyday life (e.g. finding keys or looking for a pair of matching socks). Discuss the 'tricks' that they use to reduce their searching time (e.g. colour coding of socks).

Main activity:
Provide examples of searching algorithms but with some parts of the code missing. The students should be challenged to complete the parts which are missing. If possible, the students should work without recourse to external sources but if that isn't possible they can be allowed to seek help.

Once the algorithms are complete, the students should implement them in their preferred choice of programming language.

Plenary:
Encourage a discussion of the problems encountered in the process of implementing the algorithm.

Follow-up areas/Homework

Students should be encouraged to develop their algorithm further by adding a timestamp to the beginning and end of the search. They should then compare, within the class, how much time each of their algorithms takes to run.

Topic: Testing and dry-runs

Starter:
Demonstrate to the class how to complete a dry-run of an algorithm.

Main activity:
Students should complete their own dry-runs of algorithms they have written.

It may be helpful to view this **Python website** where variables are visualised for either user provided or built-in algorithms.

Students who choose to write using Python code, should be encouraged to complete a dry-run of their program using this website.

Plenary:
Explore trace tables with the class and review how it is possible to tell which program generated a table when you don't know what was in the original program.

Follow-up areas/Homework

Students should explore the debugging features of their Integrated Development Environment (IDE) of choice (e.g. variable watch) and understand how, and when, the variables change as their program runs.

Topic: Identifying patterns and compressing them

Starter:
Select a school poster or rules banner from the classroom and ask your students to identify if there are any repeating patterns within it.

Main activity:
Working in teams, the students should then try to 'compress' the text of the selected poster and calculate the compression ratio.

Plenary:
Encourage a discussion to review the types of data which yield the highest compression ratios.

Follow-up areas/Homework

Students should research alternative methods of lossless compression (e.g. a dictionary approach). They can compare the results of these methods with that used in the main activity.

To illustrate the fact that longer files yield better compression ratios, students should first try compressing separate texts, then combine them into one text and see if the compression ratio goes down.

Topic: Complexity and scalability

Starter:
As it is essential to be able to predict how much an algorithm will slow down in response to a particular increase in data volume, scalability is important. If data volume increases tenfold, will the time taken to process the data also increase tenfold or higher? The complexity of an algorithm also affects its performance. Some algorithms are fast with certain types of data but slow with others. Finding the most efficient (best performing) algorithm is therefore a key component of program development.

To introduce these concepts, challenge your students to find the fastest way for them to go around the classroom and touch every chair but without stepping on the same place on the floor twice.

Alternatively all the students could be asked to stand up so that the class can then find the fastest method to sort themselves by height.

Main activity:
Ask your students to write an algorithm that finds all words that contain at least two letter Os'. Alternatively, students could create a spell-checking program. When given a word, their programs should be able to compare it with a list in order to decide if it is misspelled or not. The students could use a free words text file (used in Linux) located **here**. There is also this more sophisticated **site** which offers multiple word lists by English dialect, so an algorithm to read the words and find the ones that match could be created. Most

students with programming experience, should be able to create at least two versions of their algorithm. They can then benchmark their algorithms and discuss their complexity particularly in terms of the Big O notation. Students should also demonstrate that they understand how their algorithms compare to the complexity of classic algorithms (e.g. those used for sorting data and searching for data in arrays).

Plenary:
The students should compare the speed of their algorithms with those created by their classmates.

Follow-up areas/Homework

Students should review this **list** of the apparent 10 best algorithms. Do they agree with this selection? What would their top 10 list look like?

A good **summary** of algorithm complexities has been created by **Eric Rowell**, a writer and Lead Data Visualization Engineer.

Encourage students to read the section on array sorting algorithms. They should pick two algorithms and implement them in their preferred choice of programming language. Then after generating an array of random numbers, they should apply a timestamp feature and calculate the time it takes to sort the array (e.g. take a time stamp when the sorting begins and another timestamp when the sorting is over). Does their answer verify the information presented on the site?

Most programming languages provide a way to calculate how much memory a process occupies. In Python it is possible to do this using the **sys.getsizeof() method** as shown below:

Code

```
import sys
import random
import string


a = [] #this list will hold the same string 100 times
b = [] #this will hold 100 random uppercase characters gen'd using
Python's #string module



for i in range (0,100):
      a.append ('Hey ')


print (a,'\ntakes up', sys.getsizeof(a) "bytes")


for i in range(0,100):
      b.append (random.choice (string.ascii_uppercase))


print (b,'\ntakes up', sys.getsizeof(a) "bytes")
```

With the resulting printout:

```
>>> ================================ RESTART ================================
>>>
['Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ',
 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ', 'Hey ']

Takes up 460 bytes.
['Q', 'B', 'Z', 'A', 'S', 'E', 'B', 'R', 'W', 'G', 'Q', 'Q', 'A', 'C', 'M', 'U',
 'R', 'J', 'U', 'M', 'I', 'A', 'C', 'P', 'I', 'M', 'W', 'C', 'K', 'Q', 'O', 'X',
 'Z', 'E', 'I', 'J', 'W', 'E', 'B', 'P', 'A', 'P', 'D', 'J', 'J', 'N', 'P', 'C',
 'K', 'T', 'W', 'Z', 'L', 'P', 'V', 'U', 'R', 'H', 'P', 'D', 'W', 'F', 'C', 'G',
 'S', 'S', 'E', 'J', 'M', 'C', 'T', 'X', 'T', 'W', 'J', 'R', 'X', 'V', 'L', 'B',
 'F', 'K', 'K', 'Y', 'M', 'W', 'V', 'U', 'A', 'P', 'V', 'A', 'W', 'T', 'F', 'K',
 'H', 'P', 'X', 'G']
Takes up 460 bytes.
>>>
```

Ask students to consider why the lists take up the same amount of memory. Additionally they should think whether this will complicate looking for memory consumption using this module or if it can be circumvented?

Encourage students to investigate how they can test space complexity of an algorithm. They should start by looking at how this is achieved in **Python**

Memory requirements can also be tested through the serialising of a data structure. For example, a list would be converted into a string. Python uses the 'pickle' module for this. Students should repeat the space complexity test above but this time using the pickle module. Do they get the same results? Why does space complexity matter?

End-of-chapter answers

1

For logarithmic growth as *x* increases *y* increases more slowly than for exponential growth. Logarithmic growth is the inverse of exponential growth.

2

Heuristic methods take a best-guess approach when it is not feasible to account for all possible scenarios. A heuristic algorithm can be used to detect patterns in network traffic. If the pattern matches a known hacking style then that IP address could be blocked. The heuristic will take a best-guess approach and may sometimes miss new hacking methods or block legitimate traffic.

3

Quick sort is faster than other types of sorting methods, but it is also more complicated to implement. It has a time complexity of n logn which means that, because the number of bands will be a small number, we do not get a huge benefit. For example, if the number of bands was 10, then 10 x log10 = 33.2. If bubble sort was used, which has a time complexity of n2, then it would be 100. In processing terms, this difference is negligible which means that bubble sort would be more effective due to its simpler implementation.

4

A balanced binary tree will only have enough levels to represent the data. For the worst-case scenario when searching a binary tree, you have to look at each level exactly once. This means that the efficiency of the tree is directly proportional to the number of levels it contains. A binary tree could have one item per level, if it was poorly constructed, which means that it would have to look at n number of levels to log n meaning that, at worst, we have to compare log n items.

5

Global X
Local Y or Z.

Chapter 4: Principles of programming

LEARNING OBJECTIVES

- Explain the nature and relative advantages of different programming paradigms, and identify possible situations where they may be used.
- Describe the distinguishing features of different types of programming paradigms, including procedural, event-driven, visual and mark-up languages.
- Describe the role of an object-oriented approach to programming and the relationship between object, class and method.
- Describe the need for the standardisation of computer languages, and the potential difficulties involved in agreeing and implementing standards.
- Identify ambiguities in natural (human) language and explain the need for computer languages to have an unambiguous syntax.
- Interpret and use formal methods of expressing language syntax: syntax diagrams and Backus-Naur form (extended Backus-Naur form is not to be used).
- Describe the differences between high-level and low-level languages.
- Identify and describe situations that require the use of a high-level or a low-level language.
- Identify and justify which type of language would be best suited to develop a solution to a given problem.

What your students need to know

- The characteristics of object-oriented (OO) languages, including encapsulation, inheritance and polymorphism. Encapsulation is the principle of keeping data private within a class and only allowing it to be accessed by public methods. Inheritance is another core principle of OO programming which allows a subclass, or derived class, to inherit all of the attributes (properties) and operations (methods) of its parent or superclass. Polymorphism allows methods to be adapted for objects from derived classes but methods from the superclass will always give the same results. For example, using a slogan 'same name different action', we can modify a class' method without rewriting the rest of the program which requires the method by this name (e.g. '**cheeta.run()**' is not the same process as '**human.run()**', yet, they are both 'run' methods for their respective classes).
- The advantages of defining classes and code reuse. Classes define all of the attributes and operations that objects of that class will support. Encapsulation allows classes to be portable between programs. Once a class has been imported, it means that all of its public methods are available for use in the program. A key advantage of OO programming is that classes can be reused in many programs. It also means you can create a model in code that corresponds more closely with the problem being solved. For example, in a program modelling a library you might have classes for book, shelf and borrower.
- How to justify their choice of a programming paradigm in a particular situation (e.g. object-oriented for a game and procedural for a short calculation) and name the most common applications of these paradigms. Similarly, they should be able to suggest a particular language for a particular application based on the language's advantages and disadvantages (e.g. Java is an industry standard, C is fast and can run on lower specification systems, etc.).
- Different versions of standards exist and a program written in standard (or floating point data stored in one standard format) will not work in other standards (e.g. Visual Basic 6 and Visual Basic.NET, Python 2.xx and Python 3.xx).
- Lots of standards currently exist, often covering more than one language. Looking at the ECMA-262 standard could be a good way of exploring this point with your students.
- Agreement within the community of new standards might improve this situation but is likely to alienate some programmers.
- Unlike in human languages, every statement in a computer language must only be interpretable in one way, i.e. without ambiguity. Legal language is similar in that whilst difficult to read for the lay person its aim is to avoid any misunderstandings when dealing with laws and contracts. Syntax diagrams and

BNF notation are used in problem solving and designing languages themselves.

- Assembly language is a low-level programming language that is CPU specific. Programs compile and run very quickly. However, even simple programs require many lines of code as each instruction only performs a simple operation.
- How programming constructs are implemented in Assembly language (e.g. iteration and selection). Branch instructions and labels allow complex Assembly language programs to be written that manage the flow of control.
- How the Little Man Computer (LMC) simulator can be used to understand Assembly language programming. This is a useful activity to allow students to visualise the execution of low-level programs in terms of CPU components.

Common misconceptions

- Students don't always initially understand the benefit of object-oriented programming as it appears that converting a short program from procedural to object-oriented increases the amount of code and makes it harder to follow.
- When working with JavaScript, it can often be challenging for students to separate the functions from the HTML mark-up.
- Encapsulation can be a hard concept for students to implement practically. The emphasis initially should be on 'hiding' an object's attributes from other objects.
- The recursive nature of Backus-Naur form (BNF) can confuse some students.
- Programming similar solutions in more than one language can cause confusion for some students but it is necessary for students to both learn and practise using different languages in order to be able to justify a language for a particular situation.

Lesson activity suggestions

Topic: Procedural languages

Starter:
Demonstrate for the class how to produce sequencing instructions to complete a simple task (e.g. drawing a square on a whiteboard with a marker pen or similar activity). Students should plan the algorithm and then pass it to one another to 'act out' their instructions. Explain that procedural languages execute instructions in order but that the program code describes how they will run.

Main activity:
Investigate a procedural language (e.g. Python). Students should write simple procedural code such as that required to calculate the number of rolls of wallpaper needed to paper a room or to calculate the area of a circle.

Plenary:
Produce a procedural programming factsheet.

Follow-up ideas/Homework

Set simple programming exercises involving mathematical calculations (e.g. averaging numbers, calculating a discount, multiplication tables or finding areas or volumes) that the students should tackle on their own.

Topic: The object-oriented paradigm

Starter:
Begin the lesson with a brief introduction to the terms 'class', 'object', 'attributes' and 'methods'.

Illustrate these points with **Conway's Game of Life**. Ask students which methods and attributes need to be implemented to make this game possible.

Main activity:
Define the key characteristics of OO programming. Students can investigate pseudocode or, preferably, an environment such as Eclipse Java.

Plenary:
Compare OO programming to procedural. Discuss the strengths, limitations and typical uses of each.

Follow-up ideas/Homework

Students should research and write a magazine article about the history and uses of Java.

Students should choose a problem (e.g. finding the sum of two integers or comparing three integers to display the largest) and then implement it using one of the following: low-level, procedural, event-driven, OOP, or visual languages.

Students should attempt the following problem (actually coding a solution is optional here, although very much recommended).

A company has a stock control database with the following tables:

Suppliers (SupplierID, CompanyName, Address)

Items (ItemID, Description, Cost)

Orders (OrderID, ItemID, SupplierID)

Stock (StockID, ItemID, Quantity)

Sales (SalesID, StockID, Price)

This database needs to be implemented using classes where data is stored either in CSV files (one file for every table) or in an SQL database.

(1) Identify the classes needed to program this database in an OOP paradigm, including their class names, attributes and methods.

(2) Identify attributes and methods common between these classes. It would become apparent that all objects will have some of the same methods: load from file, write to file, update, delete, display to screen, and instantiate.

(3) Based on these common features, create a superclass that will have all the features common to the classes – this superclass will include loading, saving, updating, displaying to screen, deleting, instantiating. It might also contain an array of fields, so that file operation methods do not need to know the name of the field specifically.

(4) Implement the tables as subclasses inheriting the attributes and methods of the superclass and create a class diagram.

(5) Use any other UML charts that will help with the design of this solution.

Topic: Assembly language programming

Starter:

Discuss simple devices where computers might be installed (e.g. in a car). In each case, what is the computer's purpose?

Main activity:

Use a board with an 8 × 8 grid to represent RAM and individual cards containing opcodes, memory addresses and data. Put together a simple program that adds two numbers. Allocate memory to instructions by placing the card in a grid space. Take cards out in sequence to illustrate the instruction being executed. This 'board game' could be played in the class and could be extended to include storage of instructions, data and memory addresses in registers on the central processing unit.

Plenary:

Discuss the uses, advantages and disadvantages of Assembly language.

Follow-up ideas/Homework

Encourage your students to attempt some LMC challenges (such as those listed **here**: beginning with simple activities and progressing to harder ones.

Ask the students to research the 'C' language, then organise a debate within the class to review whether it is a high- or low-level language. The students should provide code snippets (not necessarily written by themselves) to support each point of view.

Topic: Standardisation of computer languages

Starter:

Ask your students to discuss if using the steering wheel is really the best way to control a car. A short brainstorming session would be a good start looking at possible improvements (n.b. a joystick is not suitable as drivers leaning into the turn could lean on it and push it too far which could flip the car). Then follow that with a short discussion reviewing how similar all cars' controls are and why that might be.

Main activity:
Review together the group's favourite programming language. Discuss all the aspects of the language that the students don't like about it (e.g. the keywords are too long – require too much typing). Investigate ways in which this language could be improved. Discuss the problems that could result.

Plenary:
Discuss if modern computers should still be able to run older programs, from 10, 20, or even 30 years ago. What are the pros and cons of this approach? The computing companies Microsoft and Apple have different positions on this issue. Discuss these different positions in reference to PowerPC versus Intel-based Macs and / or Microsoft's switch to the Metro interface (the tile-like interface first found in Windows 8)

Follow-up ideas/Homework

Students should complete some self-driven research into the International Organization for Standardization (**ISO**).

Students should be encouraged to look at the **GitHub** website and investigate the concept of 'forking' a programming language or any of its modules. A good example would be to look at SL4A (Python and other languages that run on the Android platform).

Students could also investigate and research a project that has been made non-standard by its developers (e.g. PyQT versus PySide).

Students would also benefit from completing some research into a language called **Algol 68**. What happened to it and why? What is the connection between this language and Backaus-Naur form?

End-of-chapter answers

1

(a) A set of related items in programming which share common attributes and methods.

(b) A specific instance of a class representing a single element which has its attributes populated.

(c) When a class inherits the actions and attributes of another class. This class can then specialise further by adding more methods or overwriting existing ones.

(d) Where the details of how a class works are hidden from the developer and only the public attributes and methods are exposed.

(e) When two classes which share common hierarchies from inheritance can be referred to as their superclass. A subclass can be referred to by its superclass, but not vice versa.

2

(a) Stack myStack = new Stack()

(b) myStack.push("football")

myStack.push("Drawing")

(c) print myStack.pop() will output "Drawing"

3

(a) Sprites in a platform game do not just move up and down but will jump instead. This requires use to overwrite the functionality of the sprite class, but still retain the rest of the functionality.

(b) If every sprite had a drawing method, then this can be involved on any class inheriting from Sprite. If you had subclasses of sprites, for example, a jumping sprite, then you could have an array of all sprites which you could iterate over and invoke draw() on all of them. How they draw will be encapsulated by the class itself.

4

<PRODUCT> = <LETTER><LETTER><DIGIT><DIGIT><DIGIT>

Chapter 5: Systems analysis

LEARNING OBJECTIVES

- Describe different appropriate approaches to analysis and design, including Waterfall and Agile
- Describe the purpose of a feasibility study and describe the processes that an analyst would carry out during a feasibility study
- Explain that proposed solutions must be cost effective, developed to an agreed time scale and within an agreed budget
- Describe the different methods of investigation
- Analyse a problem using appropriate techniques of abstraction and decomposition
- Represent and interpret systems in an appropriate diagrammatic form showing the flow of data and the information processing requirements
- Describe the selection of suitable software and hardware to address the requirements of a problem
- Describe the various methods of changeover: direct, pilot, phased and parallel, identify the most suitable method for a given situation and its relative merits
- Describe the use of alpha, beta and acceptance testing
- Describe the nature and use made of perfective, adaptive and corrective maintenance
- Describe different procedures for backing up data
- Explain how data might be recovered if lost
- Explain at which stage of the development each piece of documentation would be produced
- Describe the contents and use made of user documentation and maintenance documentation
- Describe the components of maintenance documentation, including annotated listings, variable lists, algorithms and data dictionaries

What your students need to know

- The waterfall model of software development consists of the traditional phases: problem definition; system investigation; analysis; design; code; test; evaluation and maintenance. Once a phase has been completed, the next phase begins. The process proceeds in a stepwise fashion until the project is finished. At each stage, a document is produced. Agile Methodology aims to complete a project swiftly using daily meetings to overcome problems revealed by testing. Teams using Agile Methodology constantly tweak their approach as they go along (i.e. as they understand more about the problem they are trying to solve) instead of trying to plan everything in detail up front. A key feature of Agile Methodology is collaboration.
- Extreme Programming is an implementation system that encourages paired programming and frequent interaction with the client.
- Scrum is another methodology which breaks down phases into short timescale iterations called sprints. Daily meetings are held (called scrum meetings) with an emphasis on the development and testing of prototypes.
- A feasibility study is an important document produced prior to the design of a system. Its purpose is to analyse whether or not it is possible to deliver the system required or desired. The considerations of all feasibility studies include the following.
  - Technical feasibility – the solution must be technically possible.
  - Time feasibility – it must be deliverable within the given timescale.
  - Financial feasibility – it must be possible to complete within the given budget.
  - Legal feasibility – the proposed system must not breach any laws.
  - Political feasibility – the system must not generate negative views and must be attuned to current political landscape.
- All systems can be investigated by direct observation, interviews, questionnaires and examination of documents.

- Decomposition of a problem is the process of breaking it down into its constituent parts. For example, a complex program will need to be broken into modules or, in the case of an object oriented program, super classes and derived classes. Abstraction filters out unnecessary detail and focuses on the general requirements including data structures and control of program flow required to solve the problem.
- Diagrams provide a useful visual representation of a proposed system. Each type illustrates a different aspect of a solution but they are language and platform independent (see summary table below). Agile Methodology often uses the Unified Modelling Language (specifically, UML 2), to represent system designs. The waterfall model is a more mature one and students should be aware of diagrams used by non-object oriented systems (e.g. Data Flow Diagrams (DFD) and Structure Diagrams (SD).

Diagram type	Purpose
Data Flow Diagram (DFD)	Shows the flow of data between entities, how the data is processed and how it is stored by the system.
Entity Relationship Diagram (ERD)	Shows the relationship between entities which will be either one to many, one to one or many to many.
Structure Diagram (SD)	These are top down design tools. They show the hierarchical decomposition of a problem into constituent modules.
UML Object Diagram	A diagram to show instances of a class which may be named or anonymous.
UML Class Diagram	Shows the name, attributes and methods of a class. Class diagrams are often drawn as a hierarchy showing super class and derived classes.
UML Communication Diagram	These show the interaction between objects in a system.
UML Use Case Diagram	A use case is a service that the system can provide. The actor is the initiator of the service (drawn as a 'stick man').
UML Sequence Diagram	These diagrams show the flow of program logic and the order in which messages are passed between objects.
UML Activity Diagram	These can appear quite similar to flow charts. In reality, they serve the purpose of both a flow chart and a Data Flow Diagram in Agile Methodology. They show processes involved in a single use case from beginning to end.
UML State Machine Diagram	Shows the behaviour of an object during its lifetime.

- Diagrams are used during the analysis and design phases of system development. During analysis, they are used to document the current system and gain a better understanding of how it functions. During the design phase, the proposed system is represented diagrammatically.
- An important component of a design specification is the definition of the target platform for the new system. This includes all of the hardware and software required and must include precise versions of software including system software.
- Before the system is ready to be implemented, the method of installation needs to be planned. The four main options are compared in the table below.

Method of installation and typical use	Advantages	Disadvantages
Direct – for entirely new systems or where the old system was faulty.	Rapid changeover from old to new system.	If unforeseen problems with the new system require it to be rolled back, this can be very inconvenient. If the system fails, this could result in loss of business. Unless staff have been well-trained in using the new system, they may work more slowly and be more prone to errors.
Pilot – for organisations with many similar branches.	The system can be tested in the intended environment in one branch or area. If it is successful, it can be rolled out more widely. This allows for unexpected problems to be dealt with prior to the main rollout. Staff involved in the pilot can train other members of staff.	Running a pilot installation will prolong the overall installation process. The pilot installation may not detect all potential issues with the new system. The old system will still need to be running throughout the pilot.
Phased – for large projects with many modules.	As above, this allows for potential issues to be resolved in a manageable way without committing to the new system entirely.	In a phased installation, parts of the system will be new and some parts old. The parts must be compatible otherwise the system could fail.
Parallel – for critical systems where safety is an issue or financial transactions are being carried out.	As the two systems are running during the changeover period, there is minimal risk of system downtime and associated loss of business.	Data will need to be entered into both the old and new systems resulting in additional workload for staff and data storage requirements.

- Testing is carried out during the development of a new system. Students need to know the activities carried out during each phase of testing. These are summarised in the table below.

Type of testing	Activities
Alpha	Carried out by development team. Tests correct functionality without bugs.
Beta	Carried out by customer / intended end users. Tests usability and unexpected bugs.
Acceptance	Carried out by demonstrating product to customer. Tests that software meets the user requirements.

- Maintenance is carried out during the lifetime of a system. Students need to be aware of the three main types of maintenance (see table below).

Type of maintenance	Activities
Adaptive	Ensuring that the system is kept up to date with new, relevant technology (e.g. for inputting data). If a faster, more secure way of doing this is developed, the system is even better than when it was released.
Perfective	Removing as many bugs as possible, no matter how minor, to create a near perfect bug-free system.
Corrective	Problems are fixed as they arise to ensure that the system meets the original specification exactly.

- Data from the system should be regularly copied and backed up onto storage designated for the purpose. A third party may have responsibility for this or backups may be carried out in-house. Either way, a full copy of all data is usually taken every week. In the event of a disc failure, the backup can be restored onto the live system. Backups usually run overnight as the reduction in network throughput

while it is running would disrupt normal business activity. During the week, smaller backups are taken of files which have been modified since the last full backup. Differential backups will copy any file that has been modified since the last full backup. However, if differential backups are run again during the week, they will be backed up again until the next full backup runs. Incremental backups only copy files that have changed since the last full backup but they reset the archive bit every time. This means that a file modified on Monday but not modified again during the week, will only get backed up on Monday night and not again until the full backup.

- Old data is usually archived off-site so that it is available for referencing if needed but does not require frequent access.
- Writing documentation is another important task when developing software. User documentation will include an installation guide, tutorial showing normal use and advanced features, online help, FAQ, and support. With the advent of improved offline and online support, the need for paper-based documentation has reduced. Technical or maintenance documentation contains the following key information: the hardware and software requirements; algorithms; data dictionary; data structures; variable listing; code listing; design documentation; configuration and maintenance procedures.

Common misconceptions

- Students need to be clear on the fundamental differences between available software development methodologies as these can be easily confused. Specifically, students need to be able to distinguish between the traditional waterfall model and Agile Methodology. They also need to be aware of the advantages of a collaborative, flexible development approach or a traditional, more rigid approach. Completing compare and contrast activities will be useful to cement understanding and assist with revision.
- It can be challenging to remember the correct sequence of the development phases and the documentation required at each stage, including diagram types. Activities to reinforce the terminology, sequence of stages, names and purpose of diagrams etc. can be very helpful (see activities below for ideas).

Lesson activity suggestions

Topic: Software development methodologies

Starter:
Students should be asked to produce a sequence of activities necessary to cook a three course meal. Students need to consider the order of activities and think about which ones can be carried out in parallel. Additional challenges could be added such as catering for an additional vegetarian guest.

Main Activity:
Divide the class into groups and host a debate for and against Agile Methodology.

Plenary:
Students should produce a summary of available software development methodologies.

Follow-up ideas/Homework

Challenge students to produce a reference guide to UML.

Topic: Analysing a problem

Starter:
Provide students with a classic logic puzzle. Lots of suitable examples are available from the Maths is Fun **website**

Main activity:
Devise a project which requires students to investigate the required improvements for a school or college Management Information System (MIS). Students should devise suitable information gathering strategies (e.g. interviews, questionnaires, examining existing blank documents). This project can really bring the topic of systems analysis alive if the staff and school are willing to cooperate. Note: there needs to be a mutual understanding that data held on an MIS is sensitive and confidential. It must be made clear to students and staff that at no point do they need access to data – just an overview of the system itself and categories of data being held (e.g. reports that can be generated, usability, access levels etc.).

An efficient way of managing such an activity for a larger group is to allocate smaller groups to specific categories of staff (e.g. business support, SLT, teaching staff and learning-support staff). Each group will need to have prepared their interview questions or a questionnaire in advance and be aware of their role in gathering relevant information. Students will then report back to the group in a 'meeting' and present their findings. From this the requirements specification can be developed.

Plenary:
Within their groups, the students should produce a list of user requirements in the form of a collaborative document.

Follow-up ideas/Homework

Having completed the main activity students could then document the existing system as a DFD.

Topic: Designing a solution

Starter:
Start this lesson with a summary of the key success criteria for the design of the improved school or college MIS.

Main activity:
Students should then design the new, improved system based on the user requirements gleaned from the previous session. Working in small teams, each team should aim to produce the interface designs for data input and outputs (reports) *and* diagrams showing system design (UML class, communication, sequence and activity diagrams or DFD and structure chart).

Plenary:
Students could collaboratively produce a design specification.

Follow-up ideas/Homework

To help reinforce the earlier lesson, students may benefit from producing a poster which describes the five golden rules of system design.

Topic: Installing a system

Starter:
Discuss with the students when the best time is to install a new software system. Revisit the sequencing starter from the first lesson in the series.

Main activity:
Students should then work in pairs or small groups to produce training materials either for their new system design or alternatively, for an existing piece of software.

Plenary:
Students should present an overview demonstration of their training materials.

Follow-up ideas/Homework

Encourage students to look at and evaluate a range of software training manuals – both online and offline. They should review and consider how helpful the materials will be to a novice user, an experienced user, and a member of technical support staff.

Topic: Testing and maintaining a system

Starter:
A matching activity looking at types of testing (e.g. alpha, beta, acceptance, white box and black box) with their definitions could be a good opener for this topic.

Main activity:
Students, in groups, should create a maintenance schedule for the school or college MIS that they redesigned in an earlier session. Their maintenance plan should include backups and disaster recovery.

Additionally you should provide students with a 'broken' program to test. They should initially only use the console or Graphical User Interface (GUI) and not have access to the code (e.g. black box testing). Students should subsequently be provided with the code to permit white box testing. It is suggested that only minimal information be provided about the purpose of the program in order that the students use their deductive skills to investigate it fully.

Plenary:
Each group presents a test report and summary maintenance plan for the proposed MIS to the rest of the group.

Follow-up ideas/Homework

Why does system testing matter? Students should be encouraged to investigate systems that have failed and look at how testing can improve reliability. A good example is **NASA's IV&V service**.

End-of-chapter answers

1

During analysis information about the current system and what problems are meant to be solved must be gathered. Feasibility must also be done to ensure that the project is worthwhile. This will include checking to see if the project can be done on time and on budget. Also, is the project technically feasible and will it conform to current law? If the project is not feasible, then it will end here. Fact finding is done through a combination of methods including interview, document collection, observation and questionnaires. Once all of the information is gathered it is then analysed by the analyst and a requirements specification produced. The requirements specification will include requirements related to the input and output, what processing is required and what hardware will be needed.

2

(a) A prototype is a version of the full product but has a limited or restricted amount of features.

(b) In a single iteration of the spiral model, a prototype will be produced.

(c) This is then evaluated and tested by the customer and the feedback used to inform the next iteration of the prototype.

3

In the agile methodology there is close contact with the client meaning that communication is much faster and valued more highly. This allows the developer to respond much faster to changing requirements as the requests would be picked up faster. There is regular feedback included in the methodology so if the client is not happy or notices any issues, agile programmers can respond more quickly. Prototypes are produced, which have limited functionality, in small set time frames. As these prototypes are small they can be shown to the client on a regular basis, again ensuring more opportunity for feedback and the ability to respond to change more quickly.

4

The Waterfall approach is not suitable for projects where the requirements may not be fully understood at the start.

5

An unfinished version of the product that can be refined or built on to create the final solution.

6

Collaboration.

7

Methodologies.

Chapter 6: System design

LEARNING OBJECTIVES

- Discuss contemporary approaches to the problem of communication with computers.
- Describe the potential for a natural language interface.
- Describe the problems of ambiguity that can be associated with input that is spoken.
- Explain the need for a design review to: check the correspondence between a design and its specification; confirm that the most appropriate techniques have been used; confirm that the user interface is appropriate.
- Describe criteria for the evaluation of computer based solutions.

What your students need to know

- Interaction with early computers required specialised knowledge as it involved setting circuits using switches and relays and an ability to program in machine code. Interaction via a Command Line Interface (CLI) allowed the input of shell commands via a keyboard and these could be viewed on a monitor. The arrival of Graphical User Interfaces (GUI) saw the advent of 'point and click' control using a mouse. Contemporary user interfaces may still utilise a GUI but it is more likely now to be manipulated using touch screen gestures. The development of capacitive touch screen technology allows for user interaction via swiping, pinching and stretching. Virtual reality is enjoying a resurgence in interest following the development of products such as **Oculus Rift**.
- A natural (human) language interface permits a user to interact with a computer using spoken or written words. It aims to enable the computer to communicate using familiar terms and phrases based on the user's language so that the dialogue appears natural and intuitive. Current proprietary examples of natural language interfaces include *Siri* (Apple), *Cortana* (Microsoft) and *Now* (Google). All three contain voice recognition software. However, strong accents, dialects, background noise and use of informal language can cause problems for the current software. Natural language was initially developed for mobile devices but Microsoft have integrated voice input into their latest desktop operating system, Windows 10. With some limitations, systems can 'learn' the meaning of certain phrases. However, such systems are unable to interpret intonation in the speaker's voice which can alter the meaning of the instruction. An example of this is when you say 'It is Wednesday today' which is intended as a statement but by changing intonation, the same phrase can also be a question 'It is Wednesday today?' Nevertheless, current systems may be able to determine the context of a question or search by using any additional information available to them (e.g. the last person called using the same device or the current location of the user, both can aid interpretation of the spoken words used).
- Design reviews ensure that the software being developed meets user requirements. The client will be required to test the interface to ensure that this is suitable. Early versions are called low-fidelity prototypes and although they often allow user interaction, the GUI may lack the intended level of finish. In addition to holding meetings, it is also helpful for developers to observe their software being used by users. This reveals errors made and lets the developers know if the interface is slow to learn or confusing. The design can then be adapted as a result of the review. The improved prototype design once available will then be reviewed again until the team and client feel that it is stable, usable and fully functional.
- All designs are evaluated against the following criteria: requirements; performance; usability; robustness; cost.

Common misconceptions

- Terminology associated with modern technology can confuse students. Although they may use acronyms to refer to interface types they also need to know the full term and associate that with a particular technology (e.g. CLI (Command Line Interface), GUI (Graphic User Interface), NLUI (Natural Language User Interface), VR (Virtual Reality) and AR (Augmented Reality)).
- Students need to refer to the stages of the system development lifecycle to understand the timing of the design validation stage and its importance in checking that the software being built meets user

requirements.

Lesson activity suggestions

Topic: Understand contemporary computer interfaces

Starter:
Show students an image of the user interface for **Bob** from Microsoft (there are alternative image sources and YouTube videos). Ask students to identify each of the metaphors and to appraise the usability of the interface.

Main Activity:
Students should investigate a type of computer interface assigned to them. One of the following: Command Line Interface; Graphic User Interface; Virtual Reality; Augmented Reality; touch screens; haptic devices; wearable technology; voice input. Each student should then contribute to a collaborative document or presentation for the benefit of the whole class.

Plenary:
Students produce a summary document outlining the uses of different types of contemporary interface.

Follow-up ideas/Homework

Students could produce a newspaper article about ground-breaking computer interfaces.

Topic: Natural Language User Interfaces (NLUI)

Starter:
Divide students into small teams then challenge each team to determine the voice instructions needed to carry out some simple instructions (e.g. drawing a square on a piece of paper or searching for a key word using the internet). How many instructions must the computer know? What might affect the computer's ability to interpret the commands?

Main activity:
Students should first investigate existing NLUI products available. In groups, they should then be able to produce a list of keywords and phrases that would be useful, for example, for a smart watch.

Students should then investigate the uses of chatbots as an example of a natural language interface. The **Pandorabots website** allows students to create a chatbot after setting up a free account. Alternatively, chatbot programs can be created using a programming language to generate the appropriate output.

Students should be able to assess how accurate NLUI are. They could do this by entering phrases into examples of different products and then testing accuracy of output (e.g. students could dictate a poem and then count the number of errors in the generated text).

Plenary:
Encourage a discussion to address how the development of NLUI could help different groups of people (e.g. pre-school children, the elderly and disabled people) to more successfully interact with computer systems.

Follow-up ideas/Homework

Students should develop an address book application (app) which includes all available input commands and their appropriate actions. Students could also design a GUI for the same app and compare the effectiveness and use of the two interfaces.

Topic: Validating and evaluating a system design

Starter:
Students create a 'Hall of Fame' and 'Hall of Shame' of Graphic User Interfaces. Provide students with images of interfaces which they need to categorise into 'effective' or 'ineffective' based on their design features.

Main activity:
Provide students with a fictional brief for a design project (e.g. to design the interface for a media player, an address book or a calendar). They can complete the task using a drawing package or a 'drag and drop' GUI development environment. The interface does not need to function but its purpose must be clear. Ask the students to validate the features of their classmate's interfaces against the original requirements.

Plenary:
Students should produce a summary document to outline the consequences of an interface that is ***not:*** robust; cost effective; easy to use; does not meet requirements; or perform well.

Follow-up ideas/Homework

Students could develop a checklist for validating interfaces in the form of a poster.

End-of-chapter answers

1

Background noise may interfere making it hard to distinguish the voice from the background.

2

Force feedback – Vibrations in the game controller will inform the user that something has happened on screen, for example, an explosion.

3

A tourist can point their phone at a landmark and find out useful information about it. They may also be able to use the augmented reality to navigate their way around.

4

Performance, robustness, cost, usability and if it meets the requirements.

Chapter 7: Software Engineering

LEARNING OBJECTIVES

- Describe the software tools that have been designed to assist the software development process.
- Explain the role of appropriate software packages in systems analysis, systems specification, systems design and testing.
- Explain the role of the Integrated Development Environment (IDE) tools in developing and debugging programs.
- Explain program version management.

What your students need to know

- Computer Aided Software Engineering (CASE) tools automate some aspects of system design, development and testing. Automated functions include: code generation; diagram production; project management tools and version control. Integrated CASE tools are used throughout the systems lifecycle. Upper CASE tools are used during analysis and lower CASE tools during development.
- The Integrated Development Environment (IDE) is software that developers use to write code. It contains a code editor which can highlight keywords in colour, perform syntax checking and provide predictive suggestions. It will also provide debugging tools including the ability to watch variable values, set breakpoints and step through and over lines of code. A compiler in an IDE will generate an error report including the line numbers where errors have occurred.
- Version management control allows software to evolve as it is developed whilst retaining the ability to 'roll back' or reverse any changes that result in bugs in the most recent version. Version management also permits teams to collaborate on large projects more effectively. Project code will be stored in a repository. Individual modules or files will be stored in branches of the repository. As each file is modified, its revision number is incremented. With versioning software such as Git, a hierarchical structure is produced. Each node of a branch can be identified by the file and the number of commits and revisions. Team members can work on the same file at the same time and their changes merged with other versions. Version numbers are usually organised to give a major, minor and patch revision number. Alpha revisions are those still in development and not yet released to clients. Beta versions may be released to a limited audience for test purposes. Software in development is usually given a 0 value for the major version. The first release is given a major version of 1. For example, software in development may have a version of 0.1.2 which means: first development version; first revision; second patch. A piece of software that has been released to clients may have a version of 1.2.3 (first release version; second minor revision; third patch). Major versions are not usually compatible with previous versions and are usually significantly different to the most recent major release.

Common misconceptions

- It will be helpful for students to produce their own glossary of key terms associated with this topic. There are a number of important keywords that they need to recognise and to be able to define. Examples include IDE, CASE, debug, breakpoint, watch, step through, step over, version control, version management software, repository, branch, version number. As confusion can arise between key terms, clarity is always beneficial.

Lesson activity suggestions

Topic: Software tools and IDEs

Starter:

A good starting point might be to play a round of programming language 'relay'. Each student must name a different programming language in turn without anyone repeating a name twice.

Main Activity:

Students should complete the necessary preparation to provide a tutorial for using an IDE of their choice. The student should be familiar with the selected system and their tutorial should cover as many features and tools as possible.

Plenary:
Encourage a discussion of the advantages and disadvantages of using CASE tools.

Follow-up ideas/Homework

Encourage students to complete their own project to investigate the use of CASE tools. They may find **this link** and the following **link** useful starting points.

Topic: Version management in software development

Starter:
Hold a brainstorm event to establish with the students what they think the most important skills for successful teamwork are.

Main activity:
After demonstrating versioning in software development, students should investigate available version management packages for software development. This **Git** book might be useful. The students should then produce a collaborative report in a format (e.g. **here**) that allows version control. This will allow the students to track the revisions to the document during its creation.

Plenary:
Students could work as a group to create a set of 'best practice' rules or guidelines for teams working on a collaborative project.

Follow-up ideas/Homework

Students should investigate and implement a version control system for a programming project they are involved in using a suitable tool.

End-of-chapter answers

1

Computer aided software engineering.

2

Gliffy is a CASE tool which can be used to create flowcharts, UML and other key diagrams used during the systems lifecycle.

3

Pretty print and debugging.

4

Multiple developers will be working on the same set of files. Version management allows developers to check out code, edit it, and then return it to the repository. When code is checked out, other developers will be made aware so they do not make changes to the same code. Once checked back in, all developers will be able to get a copy of the new code automatically. Version control also allows branches to be created so prototypes can be built and tested without impacting the main code base.

Chapter 8: Program construction

LEARNING OBJECTIVES

- Describe the function of translation programs in making source programs executable by the computer.
- Describe the purpose and give examples of the use of compilers, interpreters and assemblers, and distinguish between them.
- Describe the principal stages involved in the compilation process: lexical analysis, symbol table construction, syntax analysis, semantic analysis, code generation and optimisation.
- Distinguish between and give examples of translation and execution errors.

What your students need to know

- High-level code needs to be translated into machine code before it can be executed by the Graphics Processing Unit (GPU). The program that performs this task is called a translator.
- Compilers check entire code syntax and produce an executable from the source code. Once compiled, the source code is no longer visible. Libraries can be packaged with the executable making compiled code more straightforward to distribute.
- Interpreters check code line by line and do not produce an executable file. Code needs to be interpreted every time the program runs. The interpreter and associated library files must be present for the program to run. This could present a potential issue for different platforms as they will have different instruction sets and require different interpreters. One solution is the use of a Virtual Machine. Some programming languages are both compiled and interpreted. The Java programming language is compiled to intermediate byte code and then interpreted.
- Assemblers translate assembly language code into machine code. Each CPU supports its own instruction set and so will have its own assembly language. Keywords are identified by simple mnemonics.
- Compilation takes place in distinct stages, summarised below.
  - The first stage of compilation is called Lexical Analysis. In this stage keywords, variable identifiers (labels) etc. are stored in a symbol table. The stream of tokens produced moves onto the next stage.
  - Next, the grammar of the code must be checked to ensure that it conforms. Any errors will be reported by the compiler. This stage is called syntax analysis or parsing. Semantic analysis checks that the construction of expressions is complete and that the meaning of each line is correct.
  - After syntax analysis, machine code is produced in the Code Generation phase. Memory is allocated for each of the tokens, arranged in the correct order of operation.
  - The code is then optimised to ensure that it will run as efficiently as possible. Some compilers optimise for speed of execution and others for memory usage. Developers can also pass parameters to the compiler to optimise efficiency for their program.
- Translation errors are detected during compilation. These are also called syntax errors. Execution errors occur when the program is running. They are also called runtime errors.

Common misconceptions

- Students can become confused with the terminology associated with translation. Encouraging them to create a glossary may be helpful as well as an annotated flowchart of the compilation process.
- Students need to understand that syntax of a programming language is defined using notation such as Backus Naur form (BNF); if a symbol cannot be recognised, it is because it has not been defined.
- Students should appreciate the link between the order of tokens in a stream and the outcome of execution; this can be represented using a tree structure. Reverse Polish Notation (RPN) allows conversion of expressions between infix notation (using brackets as would be seen in high-level code) and reverse polish notation which orders the tokens in the correct sequence without using brackets.

Lesson activity suggestions

Topic: Describe translators

Starter:
Provide students with phrases in foreign languages that they must guess the meaning of. Choose common phrases but languages not likely to be known by students. Ask students to think about what clues there were in the phrases as to their meaning. How could they translate a whole passage of text?

Main Activity:
Students, in groups, should produce a table comparing the features of interpreters, (including the use of Virtual Machines), compilers and assemblers with example languages.

Plenary:
Discuss as a class whether the process of translation can be avoided, or not. What does programming directly in machine code look like? What are the requirements for programming directly in machine code?

Follow-up ideas/Homework

Encourage students to research the history of programming languages and translators and prepare a short project / report on this subject.

Topic: Describe the process of compilation

Starter:
Provide students with random words on small cards and then challenge them to make a complete sentence that can be read. Review the process that was required to do this.

Main activity:
Students should produce a 'stickman storyboard' that illustrates the entire compilation process.

Plenary:
Hold a quick quiz on key terms associated with compilation.

Follow-up ideas/Homework

Students could research the development of Backus Naur form (BNF) and its role in defining program syntax.

Topic: Distinguish between execution errors and syntax errors

Starter:
Demonstrate and discuss the types of errors encountered in programs and their causes.

Main activity:
Provide students with a number of programs containing errors. Some should be syntax errors (e.g. brackets not closed, missing semicolons, misspelled keywords etc.) and some runtime errors (e.g. divide by zero, maximum recursion depth exceeded, attempting to access an array element out of range etc.) The students should not only identify the types of error but, if time permits, attempt to correct them.

Plenary:
Discuss in class how logic errors can be prevented. How do these differ from syntax and runtime errors?

Follow-up ideas/Homework

Students should be encouraged to produce a leaflet for novice programmers entitled 'Error Free Programming' with the aim of helping them to avoid introducing errors in their code.

End-of-chapter answers

1

Languages which are compiled and interpreted will make use of intermediate code. The source code will be compiled into intermediate code, which uses a generic instruction set which no real computer can run. In order to run the program a virtual machine (VM) must be used. The VM will understand the generic instruction set and will provide a sandboxed environment for the program to run. The VM will interpret and translate the generic instructions into real machine code. This has the advantage that programs can be compiled into intermediate code and then run on any architecture which has a VM.

2

(a) Lexical analysis will tokenise the source code, removing white space and grouping characters into more meaningful tokens. For example, “myVariable” would be turned into a single variable token. This makes syntax analysis simpler. It will produce a token stream and a symbol table.

(b) Syntax analysis will check the order of tokens against the rules of the language’s grammar. If a series of tokens fail to match a rule then a syntax error is produced. An abstract syntax tree is produced at the end or translator diagnostics are reported back.

(c) Common patterns of tokens are turned into sections of machine code. Memory addresses are allocated as well as registers. Optimisation will also occur and will either be optimising for running speed or final size. This will produce object code.

Chapter 9: Economic, moral, legal, ethical and cultural issues relating to computer science

LEARNING OBJECTIVES

- Describe social and economic changes occurring as a result of developments in computing and computer use, and their moral, ethical, legal, cultural and other consequences.
- Describe the role of codes of conduct in promoting professional behaviour.
- Effect on employment – Describe the possible effects of computers on the nature of employment in the computing industry and wider society.
- Legislation – Explain how current legislation impacts on security, privacy, data protection and freedom of information.

What your students need to know

- The use of computers raises a number of ethical issues, including their use in the workplace, artificial intelligence and automated decision making, environmental effects and censorship.
- A lot of modern crime, such as **software piracy** or **phishing** attacks, didn't exist even decades ago.
- Combating **cybercrime** requires a lot of very specific knowledge, made difficult by development of computers and the communities that use them. Computers are changing faster than laws can be drafted.
- Modern technology allows enormous amounts of data on third parties to be stored and transferred around the world, waiting to be stolen or used in a way that erodes other parties' privacy. This is especially true with cloud computing, where more data is stored online, rather than on disks locally. The Data Protection Act (1998) gives individuals the right to know what data is stored about them and who is storing it, and the right to see any data a company is storing about them and have it amended if necessary. This Act also defines two information categories: personal data (widely available information on a person, such as their address) and sensitive data that by default you wouldn't want to be available (e.g. medical history, ethnic origin).
- The Act also sets forward eight principles of data protection that apply to any data stored. These include legitimate grounds for collective information (so, 'for blackmail' doesn't count), and it can't be used to hurt you. Whoever collects the data must explain in advance how they will use the data and give individuals notice that their data is being collected. There is also a responsibility to safeguard data, so it can't be used by somebody else (e.g. a hacker or a hostile government) to do anything unlawful.
- Collecting more data than necessary and claimed in advance is not lawful under the Act, it needs to be accurate, kept up to date, safe from early deletion, and discarded securely when it is no longer needed for the purpose for which it was originally collected.
- The person has a right to access a copy of information stored on them, and to object if it will cause them damage or distress, or if it's going to be used for direct marketing. Otherwise, if this right is denied, the collector of data will have to pay compensation set by the Act.
- Last but not least, personal data can't be sent to a country outside the European Economic Area unless that country or territory ensures an adequate level of protection for the rights and freedoms of data subjects in relation to the processing of personal data. As recent cases in the United States demonstrate, this is not always easy to enforce, so by default, companies should avoid sending data to countries not in the **European Economic Area**. Foreign governments can enact laws that override the Data Protection Act commitments, and this is a problem, considering how much of the world's data is stored on US servers, in particular.
- The Information Commissioner is the person whom parliament has empowered to enforce the Act. The Act establishes the following terms: the 'data controller' – a person or company that collects and keeps data about data subjects (people); and the 'data subject' – someone who has data about them stored by a third party.
- The Computer Misuse Act (1990) prohibits using a computer to secure access to any program or data

held electronically if the access is unauthorised and this is known by the perpetuator at the time, especially if this is with intent to commit or facilitate commission of further offences.

- It is also illegal to create, adapt, supply or offer to supply any materials with the intention that they be used to commit a crime, including doing that with a view to passing them on for somebody else to commit the crime.
- The Copyright, Designs and Patents Act (1988) states that creators of artistic and creative works have the right to choose how their material is used. This includes use in application software, games, films, music, dramatic productions, sculptures and books.
- **Copyright** is an automatic right from the moment of creation of a work. This means that there is no need to put a copyright symbol on your work and it is valid for 70 years after the creator dies for literary, dramatic, musical and artistic works and films and 50 years from when the work was first released for sound recordings.
- There are exceptions to the Copyright, Designs and Patents Act (1988) that allow using copyrighted material without permission. This is for educational purposes where no profit is being made, to criticise or report on them, so that libraries can copy and lend with a special licence, for recording a broadcast for viewing/listening to later, for creating backups for personal use, to play recordings in a club or society and to copy a program to study and test to learn how it works.
- Breaches of this Act are punishable by up to two years in prison and a fine, including any damages caused.
- The Regulation of Investigatory Powers Act (2000) is a controversial 'Big Brother' Act that particularly focuses on electronic surveillance, the legal decryption of encrypted data and the interception of electronic communications by the security services, the Secret Intelligence Service and the Government Communications Headquarters (GCHQ). Under the Act, they can intercept e-mails, access private communications and plant surveillance devices. There are reasons to believe that the Act is being used by the government agencies more often than originally intended (with a very low level of convictions made based on the information gathered).
- Students need to know specific examples for every principle of every Act mentioned in this chapter.
- Computer technology raises ethical issues. While they helped to cut costs, the introduction of computers resulted in higher unemployment and the movement of jobs overseas. However, many dangerous and repetitive jobs have been taken over by computers. Widespread production of computing technology and its fast obsolescence and replacement result in a very pollution-creating industry and one which uses up limited resources. It gives rise to exploitation of developing countries and even finances civil wars. Disposal of old computer equipment is also a big environmental issue that seems to hurt poor countries with inadequate environmental protection.
- **Artificial intelligence (AI)** aims to replace human decision making, such as with Google's driverless cars, but at the current state of technology it can cause problems, such as herd behaviour by stock market computers all following an instruction to sell a stock that hits a certain price – causing the whole stock market to crash as the volume of trade increases exponentially. It can also cause overreliance, such as cases of cars falling off cliffs or driving the wrong way on motorways following wrong or misinterpreted satellite navigation directions. The free and wide nature of the internet brings up issues of **censorship**. Schools censor the internet to avoid distractions to students. China censors the internet to restrict criticism of the government. Many countries restrict access to sites promoting software piracy and pornography. The optimal amount of censorship and its place in a society, as well as the ethics of enforcing it, are widely discussed issues and it is likely our opinions on what is or is not acceptable on the internet will change in the future.

Common misconceptions

- Students may think that cybercrime is punished lightly being non-violent crime.
- To avoid being cyberbullied, students shouldn't give out any information – but giving information out is not an agreement to having it used unlawfully.
- Students can stop third parties collecting information about them and they don't need to communicate the reasons why they don't want that data collected – in reality, they have to prove this will cause them objective distress or damage.
- Some may think that helping others to commit cybercrime will not make them liable, as long as it is on

someone else's computer.

- Students may think that copyright needs to be registered – this is not true, it is automatic. They may also think that copyright is 25 years, like patents – again this is not true, it is much longer.
- Students may also have concerns over artificial intelligence from its mixed portrayal in popular culture.

Lesson activity suggestions

Lessons on the Acts could revolve around discussions of legal precedents and how much the students themselves are aware of the scope of some of their everyday activities. Students can be asked if it is acceptable for schools to photocopy textbook pages, or if taking notes from a textbook is breaching the author's copyright. Is it different for electronic textbooks?

Students can be asked to look up the difference between these two copyrights in music: 'c in the circle' versus 'p in the circle' (author's versus performer's copyright, respectively). How much is a school paying a copyright holder company to stage a school play like *Guys and Dolls*?

Talking of the jobs that have changed or disappeared/appeared because of the computing revolution is a good way to get started on the *Computers and the workplace* topic.

Topic: The Data Protection Act (1998)

Starter:
Look at the wording of the Patriot Act (2001) and then look at the case of WikiLeaks.

Main activity:
Present the Data Protection Act's main articles backed up by case studies.

Plenary:
Ask students which of these Acts are easier to convict somebody on.

Follow-up ideas/Homework

Research Snowden's revelations and write up a cause for suspending information sharing between the UK and US unless specifically authorised by the government.

Alternatively, identify three other countries you would not want to send information to.

Topic: The Computer Misuse Act (1990)

Starter:
Brainstorm a list of activities/scenarios that would breach this Act.

Main activity:
Present the Computer Misuse Act's main articles backed up by case studies.

Plenary:
Ask students which of these articles are easier to convict somebody on.

Follow-up ideas/Homework

Research three cases of people prosecuted under this Act.

Topic: Computers and the workplace and Copyright Law

Starter:
Discuss with the class how computers have changed the job of a teacher over the past decade? Has anything changed since students have been in the school?

Main activity:
Conduct a class discussion resulting in the compiling of a list of ten jobs that have been replaced by computers and ten new jobs that have been created over the last 20 years as a result of the introduction of computers.

Plenary:
Discuss the rationale behind 'Project Gutenberg'.

Follow-up ideas/Homework

Research a list of books that have recently gone out of copyright.

Topic: Artificial intelligence

Starter:
Discuss Isaac Asimov's laws for AI – can they prevent computers harming people?

Main activity:
As a class, brainstorm modifications needed for our roads before we can have driverless cars.

Plenary:
Discuss if there will be things that will never be replaced by artificial intelligence.

Follow-up ideas/Homework

Look at excerpts of Asimov's *I, Robot*.

Research modifications needed for our roads before we can have driverless cars.

End-of-chapter answers

1

- Personal data shall be processed fairly and lawfully.
- Personal data shall be obtained only for lawful purposes.
- Personal data shall be adequate, relevant and not excessive.
- Personal data shall be accurate and, where necessary, kept up-to-date.
- Personal data shall not be kept for longer than is necessary.
- Personal data shall be processed in accordance with the rights of data subjects.
- Data must be kept secure.
- Personal data shall not be transferred outside the EEA.

2

To unlawfully access computer systems or modify computer material.

3

Regulation of Investigatory Powers Act 2000.

4

The Copyright, Designs and Patents Act 1988.

5

The Information Commissioner.

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom
Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: **www.cambridge.org/ukschools/9781107549647** (Cambridge Elevate-enhanced Edition)

First published 2015

A catalogue record for this publication is available from the British Library
ISBN 978-1-107-54964-7 Cambridge Elevate-enhanced Edition

Additional resources for this publication at **www.cambridge.org/ukschools**

Acknowledgements

Cover © 2013 Fabian Oefner **www.FabianOefner.com**