CAMBRIDGE

Brighter Thinking

# COMPUTER SCIENCE

## A/AS Level Component 2
## Teacher's Resource for OCR

Christine Swan and Ilia Avroutine

# How to use Cambridge Elevate resources

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.

If your centre is using the accompanying Cambridge Elevate-enhanced Editions for OCR A/AS Level Computer Science Component 1 and Component 2, you can use the following features to support your teaching:

- You can choose to share annotations, bookmarks, etc., with groups of students. So, for example, if you have different ability levels within your class, you can use groups to share links to more stretching activities with the higher-ability students only.
- You can share links within and between Cambridge Elevate resources with groups of students. For example, if you wanted students to link concepts or learning between Component 1 and Component 2. Please note, however, that the groups need to have access to the relevant titles in order for this to work. For example, you couldn't share a link between Component 1 and Teaching Programming as your students will not have access to Teaching Programming.
- Component 1 and Component 2 include 'Assess to Progress' features, which allow you to set students specific questions from the end-of-chapter tests. Once the student has submitted their work, this feature will allow you to mark their work against the assessment objectives and a set of levelling statements, and support you in monitoring and reporting your students' progress over the duration of the course.

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.

# Chapter 1 Thinking abstractly

## Learning objectives

- To understand the nature of abstraction.
- To understand the need for abstraction.
- To know the differences between an abstraction and reality.
- To devise an abstract model for a variety of situations.

### What your students need to know

- How to do procedural (with parameter passing) and object-oriented programming (with inheritance and polymorphism) in at least one high-level language.
- How to draw class and activity diagrams with superclass and subclass clearly labelled indicating the direction of inheritance and all attributes and methods.
- When to use selection, iteration and abstract data structures such as arrays and lists, especially lists of objects, otherwise known as collections. They should also be aware that multiple dimensions of these structures need to be internalised, and multiple indices need to access individual elements.
- When to use recursion where it results in shorter code, including implementing self-calling and terminating conditions. Each recursive program has an iterative version and students should be able to convert between the two forms of programming.
- How to use file input/output ( I/O) in at least one high-level language to read/write serial and Comma-Separated Values (CSV) files.
- The differences between operating systems including their utility programs, multi-tasking, and CPU slicing.
- The differences between types of computer languages.

### Common misconceptions

Most people are familiar with the word abstract but as it can be used as a noun, verb or adjective . It is important to clarify its meaning in the context of Computer Science where it refers to a technique of breaking down complex problems by identifying common patterns.

There are usually many ways a problem can be split into abstract layers, or parts which might come as a surprise to some students.

## Lesson activity suggestions

| Topic | Lesson suggestions | Follow-up areas/ Homework |
|---|---|---|
| Understand abstraction. | **Starter:**<br>Ask your students to categorise computer games into groups by identifying common features.<br>1. Make a list of computer games that they are aware of (as a class) and try to work out the categories of games. For example, platformer, first-person shooter, role-playing. At least two to three features need to be identified for each game to prove its inclusion into a group (e.g. jumping, obtaining ammunition or improving health, navigating a maze, etc.). Do any games stand on their own?<br>2. Repeat the exercise with musical acts or bands, trying to categorise them into musical styles. Some of the defining features could be the instruments, tempos, political message (if any) etc. Be ready for some hot debate!<br>3. Present two programs side by side that accomplish the same outcome, one making extensive use of abstraction and the other not.<br>**Main activity:**<br>Ask the students to write a program to calculate the minimum number of coins a vending machine should return after an amount is put in the machine to pay for a particular item whose price is known.<br>1. Start the students on this task in groups.<br>2. Watch the progress carefully and note when groups use a very long selection statement (without abstraction), through iteration and substitutions and when other groups employ recursion. Interrupt the activity at this point.<br>3. Explain to the students that recursion usually results in much more concise and compact code. Signpost to your students, how this activity should make the merits of abstraction apparent.<br>**Plenary:**<br>Students discuss any other examples where abstraction would be advantageous. Discuss what 'advantageous' looks like in a big organisation in terms of their hardware costs, CPU time, program completion time (e.g. on production lines). | Talk to your students about device drivers. Give them the name of a device, for example a particular sound card model, and challenge them to find out how many different drivers can be downloaded for this device from the manufacturer's site. When they return with their findings, the discussion should centre on the fact that while every driver is specific to an operating system, they all function in the same way as far as the user is concerned. This insulates the user from the particulars of an operating system via a layer of abstraction. |

| | | |
|---|---|---|
| The need for abstraction. | **Starter:**<br><br>Ask your students: 'What is a force of habit?' and 'How are habits formed?' In answering these questions remind your students that when certain events repeat often enough to be predictable, they can be broken down into standard elements and completed more efficiently.<br><br>When your students are writing computer programs, ask them what the most common lines are that they use.<br><br>Do they wish that they didn't need to type them as often? Discuss with them how they could avoid these steps. As part of your discussion remind the students that most languages allow one program file to use code from another file, known as a library, so most repetitive lines can be packaged into a procedure and imported to all subsequent programs.<br><br>The concept of a push button is a good example of abstraction. What examples of different buttons have students seen in their lives, both in hardware and software? What does the abstract concept of a button represent? Why does it share its name with a very different object used to hold together parts of clothing; surely we don't actually 'push' our clothing buttons?<br><br>**Main activity:**<br><br>Explain that abstraction is about seeing common patterns between situations and problems. Abstraction helps to solve a new problem by applying previous knowledge.<br><br>Abstraction can be split into two main parts: control abstraction and data abstraction. Control abstraction makes high-level languages possible by recognising that selection and iteration are common, and can be implemented in consistent ways between the problems.<br><br>Data abstraction is about data structures; sets of related data stored in a particular way, like arrays and queues.<br><br>Abstraction has many layers, a bit like an onion! Computational thinking is the process of progressively identifying abstract elements in a problem and applying known techniques to deal with them. For a given problem, the first layer of abstraction might be to break it down into subroutines, then for each subroutine loops can be applied, which in turn will iterate through arrays.<br><br>1. On the whiteboard, show your students sections of the code from some operating systems. Show them where tasks have been categorised and broken down into common elements.<br><br>2. Show them the code from sections of a form where buttons have been used.<br><br>3. Show them where elements of software and hardware communicate with each other through a set of standard calls and explain that these are combined into an interface, referred to as an Advanced Programming Interface or API for short.<br><br>4. Make students aware that the core of an Operating System's (OS) functionality is known as the kernel and all parts of a software system need to talk through the kernel which will authorise and handle requests for resources.<br><br>5. Ensure students know that the Java programming language was designed to run on any computer regardless of the OS or CPU, so it is even more abstract than most. An OS-specific Java compiler residing on a computer is responsible for translating generic Java commands into specific hardware calls, such as that for handling 3D graphics, via sets of API calls called software libraries.<br><br>**Plenary:**<br><br>Explain the different levels at which software engineers work. The lowest level of abstraction is at the hardware level. Lowest, as used here, means that very few programs or engineers need to know how hardware works; for example, an engineer doesn't need to know which motor will spin a particular hard disk's platter to reach a particular address and retrieve a file, they just need to know a standard, generic API to find that file. | To illustrate the use of the API, your students could be challenged to implement at least 10 API calls of their choice (this can be done in teams) in their programs. A common API could be to read / write / append a file, open another program and send data to it (e.g. **SendKeys**) or a program to return a list of files to a folder, with their attributes (e.g. size and time of creation). |

| | | |
|---|---|---|
| The differences between abstraction and reality. | **Starter:**<br><br>Have a discussion with your students. Abstraction is not just a computing concept. It is also used in science, culture and art. It is particularly useful for creating approximations or models of real-world situations. The model will be generic but can be applied to a range of situations so that when known data (for given parameters) are applied it is then specific to a real situation. A lot of games use object-oriented programming to group common attributes (object variables) and common methods (object functions and subroutines) to reuse code and speed up development of similar features.<br><br>**Main activity:**<br><br>Ask your students to consider a racing game involving cars. Regardless of the car the player chooses, it will have a steering wheel, an accelerator pedal and gauges. The code required to implement these and other attributes can be reused with small modifications (parameters). Recognising what can be recycled is the process of abstraction.<br><br>1. Arrange for the students to play a simulation of a car racing game, on-screen. Whilst they are playing the game, they should think about the sequence of steps that they are going through as players.<br>2. They need to map out their actions in a flow-chart.<br>3. Students should use the flowcharts to produce a list of the subroutines that they believe will need to be called by the main program.<br>4. As a class, they should review the various lists which they have produced whilst one of the students plays the game. They should try to pause at each stage. The class should check that they have called enough subroutines to run the program and not missed any out.<br><br>**Plenary:**<br><br>Use this opportunity to discuss and reflect with your students that:<br><br>1. They have undertaken a process of stepwise checking of a program.<br>2. They have listed subroutines which are abstract.<br>3. They have seen the program in operation with parameters i.e. the choice of car, the skills of the player, the speed of the player, the skill levels chosen in the game etc. | Students can be asked to design or suggest classes for a rock/paper/scissors game, for example: Game (with attributes of score, time, player ID, etc.), Player (human/computer, one or two, etc.), Sprite (if using graphics). |

## End of chapter answers

1 Abstraction is the process of separating ideas from reality. Only certain details are shown, hiding irrelevant detail.

2 The app could be made more generic and be used for any subject. By considering what the questions in different subjects have in common and removing subject-specific styles, it would be possible to create an app which could be used for any subject.

3 Sarah could refactor her code to accept alpha-numeric and numeric data. By using abstraction she can make her code accept any data type which is comparable, and then use this comparison to perform bubble sort. The type of data has no relevance to sorting.

4 APIs are used in many different scenarios and applications. The designers of the API will have no knowledge as to how their code will be used and they therefore need to make it as abstract as possible. If the code was too specific then this would prevent others from using the API effectively.

# Chapter 2 Thinking ahead

## Learning objectives

- To be able to identify the inputs and outputs for a given situation.
- To be able to determine the preconditions for devising a solution to a problem.
- To understand the nature, benefits and drawbacks of caching.
- To understand the need for reusable program components.

### What your students need to know

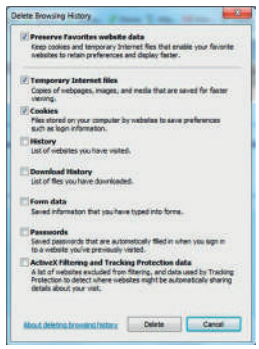
- There are different data types. Students need to be familiar with string, real, integer, Boolean and object variables, as well as abstract data structures like arrays and lists.
- There are different types of memory. Students need to understand how variables are stored in a physical memory and that physical memory is classified as primary, secondary, and cache.

### Common misconceptions

- Many students whilst learning to code do not sufficiently plan their programs at the outset. Students usually dive in and spend time on features that are later found to be unnecessary. Another result of insufficient planning is the need for endless debugging of variables passed between procedures.
- Students should be encouraged to plan their programs on a piece of paper, so they must be shown how it can be done and given examples.
- This chapter ties in well with the Waterfall and System Design part of the syllabus. Discussion of prototyping and agile development will help convince your students of the benefits of planning ahead.
- Some students don't fully understand the Object-oriented Paradigm. This may be due in part to the fact that school exercises are often not complex enough to justify using it. However, it is really important for students to try writing a few programs that use inheritance and polymorphism. Game making, as shown in this chapter, is a good way to introduce these concepts. An even easier approach would be to use classes as records, even without many class methods. A programmer could create a class called employee which has some basic functionality, for example: salary, name, year of joining, method of being hired, fired and promoted, etc. From this employee superclass, subclasses could be created, for example, part-time employee and full-time employee that would inherit from the employee superclass.

## Lesson activity suggestions

| Topic | Lesson suggestions | Follow-up areas/ Homework |
|---|---|---|
| Identify inputs and outputs. | **Starter:**<br><br>Discuss with your students why planning before starting to code is absolutely necessary, especially for complex problems. You may like to explain that professional developers often complete the planning step as a team. A standard planning method is therefore needed to reduce miscommunication within the team and the costly duplication of work.<br><br>**Main activity:**<br><br>Your students should work in teams to create a program to simulate the operations required to run a refreshments stall (as simple or as complex as the students can manage) at a local outdoor event for a day.<br><br>Planning starts with identifying the inputs and outputs of a problem (e.g. supplies, electricity, requests from customers, food and drinks produced, money and change etc.).<br><br>The students should initially consider the overall task, then break it down into smaller tasks and keep repeating that process until the problem can't be broken down any further. For the data inputs and outputs, the students need to look at their types, whether they are a primitive variable, like an integer, or an abstract structure like an array (e.g. to represent elements of the customers' orders).<br><br>They should also consider any preconditions. Which functions, or sub-procedures, must be run before others, (e.g. a login procedure, greet the customer etc., needs to run before most other procedures run). Activity diagrams and other Unified Modelling Language (UML) diagrams are useful in this decision-making process.<br><br>The students should plan who in their group is going to handle which aspect of the problem. Where the problem is a sub-problem they need to plan its interaction with the other parts of the program. They also need to plan the scope of its variables; the way parameters are passed in and out.<br><br>They also need to consider whether data needs to be stored in a database or in a simple log text file. How will the customers' orders be retained until their food order is completed and how will changes to the stock and takings be tracked?<br><br>The data flow needs to be validated. Multiple validations may need to be run and in a particular order. For example, range validation needs to run after the data-type validation. Trying to compare a number, to determine range, to a non-number which could have been erroneously entered will trigger an error. If validation is not complete a programmer needs to supply error-trapping code, so the program displays an error but doesn't crash. The students should consider, for example, what happens if a customer requests an item which is not on the menu board, or they don't provide the right amount of money, or a drink is spilt.<br><br>**Plenary:**<br><br>How do the students feel about their simulation programs? Have a reflective discussion about abstraction. Best practice involves the use of abstraction at the planning stage. In other words, certain requirements and operations are identified as the most important and other details are left until later. This creates layers of abstraction. Did your students use any form of abstraction in their programs? | Wedding ceremonies are often planned well ahead. What is the reason for this? What might happen if a wedding is not planned properly? How is this similar to planning a program? What can go wrong? Individually, the students could show their understanding of the inherent complexity in programming projects by planning a wedding in the same way as they planned for the refreshment stall. |

| | | |
|---|---|---|
| Programming, making use of a cache. | **Starter**:<br><br>Explain to your students that, in addition to the planning required to design a program, planning to optimise running, in terms of both hardware and software, is also advantageous. At the hardware level, when a program runs, certain parts of it are more active and use memory more often. Particular data elements are needed more often, so it makes sense to put them in a special cache memory rather than to have to retrieve these elements from the memory every time. A smaller capacity, faster memory buffer exists at many levels in a computer system; ranging from those built into the CPU or hard disk controller, to those utilised in an internet browser to speed up the loading of frequently revisited pages. This process is also repeated at Internet Service Provider (ISP) level, where the most popular sites are cached on local servers, to save bandwidth between the provider and the internet backbone. For these reasons, computer systems need to be designed so that they can predict what data will be needed; they need to be able to do the equivalent of thinking ahead.<br><br>**Main activity:**<br><br>Discuss with your students why it is easier to get to places in a town that you know compared with one you haven't visited before. Do they agree that perhaps our memory in effect caches details of familiar places which helps us to navigate faster?<br><br>To experience the concept of caching, give the students a map of a big city and challenge them to design a routefinder computer program. The students should be given a number of destinations starting from the railway station (e.g. eight destinations to which their routefinder must give directions). The first player/team, creates a list of directions from the starting to the destination point, which could be written on one piece of paper for each and laid out in front of the classroom. The second team does the same but they need to reuse as many of the steps on the first team's directions as they can. The third team does the same. The advantage of reusing the existing directions – this should save time. While some time is needed to determine which of the existing directions can be used, this is still faster than writing them out on their own piece of card.<br><br>With each subsequent team, the time to lay out the route should be decreasing – illustrating the benefits of caching.<br><br>However, if one route is very different from others where no existing directions can be used, there will be a large time penalty because the player/team has to write all directions themselves, but they will also spend the time looking over the existing directions of the players before them, desperately trying to find something they could use. This is also a reality of caching in the real world.<br><br>The extension could be that the players/teams are given a fixed cache size. For example, they cannot reuse more than five directions from the players/teams before them. This will speed up the process for some of the teams and slow it down for others.<br><br>**Plenary**:<br><br>Discuss with your students the benefits and drawbacks of cache memory and look at why the industry favours it.<br><br>Cache capacity is limited so when a system incorrectly 'guesses' which data will be needed, cache misses occur. The system accesses the cache but doesn't find the needed data so has to go back to the original source and retrieve data from there instead. This is slow. The efficiency of a cache is known as the Hit Rate Ratio; the proportion of successful cache hits to misses. When multiple CPU cores and threads are considered, the Hit Rate Ratio becomes even more important. Splitting tasks between cores needs to take account of how busy they are and whether some are not currently needed so could be switched off to conserve energy.<br><br>Effective cache management requires the use of heuristics which are generic algorithms known to work in similar conditions. Algorithms like the clairvoyant algorithm or Least Recently Used (LRU) allows the cache to keep the bestsellers; the most often used elements which are swapped with the less often used.<br><br>Logistically, the process of filling and refreshing cache from the main memory unit is done through a number of hashing algorithms, such as Direct Mapping, Two- or Four-way Associativity. Different methods have trade-offs between cache size, cache-hit-accuracy and access speeds. Direct Mapped caches will map a single memory location to a single cache location using the formula pc = pm mod n where pc is the location in cache memory, pm is the memory location and n is the size of the cache. In computing, mod (division with remainder) is used to break something into equal-sized groups, often connected with addressing systems, such as the one the operating system uses for random access files. | Students should research and answer the following questions:<br><br>1. Different hardware devices that make up a computer system work at different speeds, for example, while a hard disk works in milliseconds, primary memory works in nanoseconds. How do we smooth out the differences in the speed of devices?<br><br>2. Google search results sometimes provide an option to load a cached version of the page. What is the reason for this feature and how can this be implemented?<br><br>3. Search engine results usually depend on whether a user is logged in to their personal account. How would knowing something about the user affect the search results returned?<br><br>4. Why is it recommended to clear out a web browser's cache memory from time to time?<br><br>  <br><br><br>5. Research Local Area Network (LAN)-based web caching. |

| | | |
|---|---|---|
| Reusable program components. | **Starter:**<br><br>Discuss with your students why programmers don't often write their code from scratch and how it is more time- and cost-efficient to reuse something already written. The current scale of the software industry, tight deadlines and growing demands of product cycles has driven this change. Programmers have access to basic libraries of code in addition to third-party plug-ins which can be open- or closed-source software. Additionally, most programmers accumulate code that they know works, so they reuse parts of their own programs.<br><br>Experience reveals that in many programs there are often similar parts for which the code can be accessed from templates. Templates have the code elements that are commonly used together packaged or encapsulated into objects. This has led to the Object-oriented Paradigm, as it is known, being the most popular way of writing code now supported by all the current programming languages.<br><br>A programmer who reuses their own or another's code should be aware that writing in a more abstract style makes their code more flexible and easier to reuse in multiple programs. For example, rather than hard-coding a certain number, like a VAT rate of 20%, a better program would receive the VAT rate as an entered parameter. When VAT is different, the program can still be used. In another program, the printing of the output of a complex calculation to the screen would, in a better program, instead pass out of the function, so that it can be used in other programs, saved to file or fed into another calculation.<br><br>Inheritance is another very useful feature of the reuse of code. Rather than writing code for an object from scratch, the closest existing object is accessed in order to piggy-back the new code on top of it. The old code is modified by adding different elements, a process known as Polymorphism. The donor or parent class is known as the superclass. The new class which has access to the superclass' features is known as the subclass or child class. Multiple inheritance is also possible, where features are reused from multiple superclasses.<br><br>**Main activity:**<br><br>The students should prepare for a team quiz. You could start by providing a handout containing the following text:<br><br>*The Python programming language places great emphasis on efficient coding, which includes making code reuse as easy as possible. Python packages come as plain text files which can be viewed and imported so a programmer can build on their functionality. A lot of Python features are contributed by third parties, either from open- or closed-source software. Some popular examples are:*<br><br>• *Pygame for game design,*<br>• *Tkinter for Graphical User Interface (GUI) implementation; Python natively doesn't support GUI,*<br>• *Turtle for learning to program,*<br>• *SQLlite for implementing databases.*<br><br>*The file turtle.py is great fun. The file itself inherits features from Tkinter, so Turtle will not work without Tkinter. (Please note that Tkinter doesn't work on mobile platforms, so neither will Turtle). If a user creates and saves their Python program as turtle.py it will disable the original turtle.py as it is assumed that the user is trying to write their own (better) version of Turtle.*<br><br>*Third-party code is distributed in the form of component classes, as they are self-contained and can work in a wide range of applications. Third-party classes are often distributed in collections with clear structure and documentation. These are also known as frameworks. Pygame, Ruby on Rails, QT, and Windows Foundation Classes are examples of frameworks.*<br><br>Then ask your students to spend some time doing some research, for example, reading wikis and online tutorials in order to find out what is available to Python programmers as reusable program components. They should be prepared to answer some questions in a team about the subject.<br><br>**Plenary:**<br><br>Hold a quiz to see who has managed to understand the most information from their research. | Explore a framework with your students called Django. What is its relationship with Python? |

| | | |
|---|---|---|
| Importing a program component. | **Starter:**<br>It will be useful for your students to experience a reusable program component for themselves.<br>**Main activity:**<br>Using Python, students should create a short simple program and then import it into another Python program to explore how the import process works.<br>**Plenary:**<br>Explore this component framework from Pony ORM: What is the purpose of this library and why would somebody use it when there are other libraries available that perform the same tasks? | Explore a framework called SL4A. What is it for? Is it popular? Why or why not? |

## End of chapter answers

1 Development time can be reduced due to using code which has already been tried and tested. As the code has already been thoroughly tested then the final product should be more reliable. Software is modular and can be shared through the use of library software.

2 Hit ratio shows the number of cache hits in comparison to a cache miss. When a cache miss occurs data has to be copied from the main store to the cache which in turn could cause stale data to be ejected. This process is slow and will reduce the overall speed of the system. The more cache hits you get the faster the system will perform therefore a high hit ratio is crucial.

3 If modules destined for reuse were made specific to the implementation they were first used in then it would be harder to use them in a different context. Abstraction is the process of separating the idea from implementation and this is key for reusing code in different contexts.

4 Caching gives faster access to data and can reduce bandwidth and load on resources. However caching increases complexity and can potentially deliver stale data.

# Chapter 3 Thinking procedurally

## Learning objectives

- To be able to identify the components of a problem.
- To be able to identify the components of a solution to a problem.
- To be able to determine the order of the steps needed to solve a problem.
- To be able to identify the sub-procedures necessary to solve a problem.

### What your students need to know

- How to draw up a flowchart including standard shapes for input, output, process, decisions, etc. This includes being able to show nested loops or selection statements which require planning so that all the parallel lines fit on the same sheet of paper.
- How to describe a problem's solution in pseudocode. Standard rules need to be used which include the terms for input, output, assignment, iteration, selection and other common tasks. Pseudocode must not be programming language specific. When reading an algorithm written in pseudocode there should be no indication which language this will be programmed in later. Pseudocode is a planning tool and should not be reverse-engineered from the existing code.
- How to implement parameter passing. This is standard practice in procedural language to indicate how blocks of code communicate with each other. Parameter passing is almost always preferable to using global variables, but it is harder to visualise so less suitable for beginners. More able students should be aware that multiple parameters and lists can be passed, and that optional parameters may or may not be passed.
- That there are variable types, including string (text), numeric (integer, float, real), Boolean (true/false), as well as lists and arrays.
- That practical programming in a high-level programming language supports procedures and modules which are imported as needed.

### Common misconceptions

- Inexperienced programmers often change their code significantly when they can't overcome a particular problem. The reworking often breaks the connection with other procedures which results in unexpected errors.
- Students may not use the correct descriptive names for procedures or follow the correct conventions. For example, in some languages (e.g. Python), procedures must be lower case with underscores, whilst in others, camel case should be used.
- Students need to be careful not to use the same name for a local and a global variable as this results in confusion and random values appearing in calculations. Students should be encouraged to either avoid using global variables or name them appropriately.
- Students can struggle to understand the difference between functions and subroutines. It is advisable to discuss a number of different examples with your students to highlight the differences and to experiment using a function as a parameter to a different function, for example: show_result(compute_result(10)).
- Parameter passing can cause problems for students if they don't plan ahead. The data type returned by one procedure may not be one that another procedure is expecting, for example, if the procedure: get_input() asks the user to enter their age and returns the value as a string to the procedure: compute_result(x) which relies on an integer, the program will crash.

When introducing procedural programming, it is useful to make use of global variables. This way, instead of receiving and passing parameters, the procedures that make up a program just tap into global variables. When relying on global variables, it is possible to avoid parameter passing so less able students may initially prefer to use them. However, all students need to appreciate that global variables are insecure and waste memory space because they remain even when not immediately needed (e.g. when no procedure that uses them is running). This is unlike parameters which are garbage-collected as soon as a procedure stops running.

An important concept related to procedures is unit testing; being able to test a procedure in isolation from the rest of the program. It is very important that students know how to do this and have opportunities to practise it.

Object-oriented programming is more complex than procedural programming. A source of confusion for students when running a method in an object-oriented program is that they are not always aware which object ran its method. For example, when using Python Turtle there can be multiple turtles, each of which can turn left. As each object contains a procedure that will rotate the turtle to the left, students have to make sure they keep track of which turtle has turned left and when.

## Lesson activity suggestions

Procedural Thinking: In every lesson activity, keep reminding your students of these important points:

1. The best programming and testing is done on small, self-contained chunks of code which are easy to write and test and which can be used repeatedly. This is described as a 'divide and conquer' approach or 'bottom-up' design.

2. Procedures, also known as subroutines, or methods in object-oriented programming, are closely related to (and sometimes indistinguishable from) functions. Procedures have: descriptive names, sets of local variables that are specific to the particular procedures and parameters (which are variables that receive data from other parts of a program).

3. Functions in this chapter describe particular types of procedures. Students can distinguish a function by the use of the word return.

   Rather like a Lego block that needs to connect to another, a function returns one or more values (rather like the bumps on the Lego block) that are received by another procedure where they are processed further. Conversely, procedures that are not functions, don't return any values, so they are like Lego blocks with smooth ends, used for roofs or foundations of Lego structures.

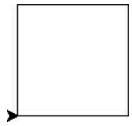
4. Procedures can handle input and output, perform calculations or just call (trigger) other procedures. The Imperative Paradigm, made popular by the Pascal programming language, contains a procedure called **main** that doesn't do any processing itself but serves as a dispatcher to run procedures as needed. Pascal has less commercial use now but does have applications in teaching. If you have not had any prior experience of using Pascal some research of the Imperative Paradigm is recommended.

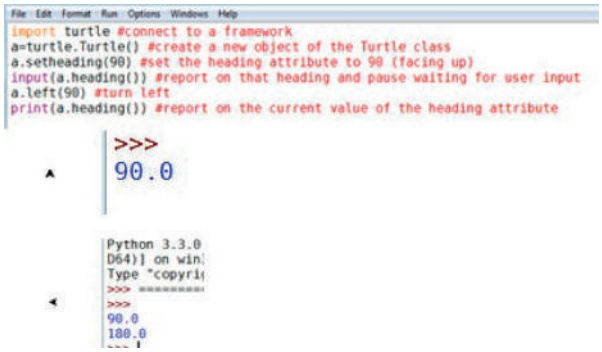
5. Procedures have the additional benefit of improving teamwork as different procedures can be written concurrently by different programmers. With proper planning, each programmer will know how each of the other procedures will interact with the one they are writing. They can then prepare all the piping required to connect their part to the other procedures when they are complete. The set of rules which describe how different procedures interact in a system is known as an **interface**. Students may be familiar with the word interface as used in Human User Interface which describes when humans trigger code elements to run. A procedure interface requires different parts of the program to trigger each other so a program will have its own internal interface. The interface is effectively a contract between different program parts to establish what each part will do for the common goal.

6. Procedures are planned at the development stage through stepwise refinement. This refers to a process of identifying all operations, or steps, in solving a problem and assigning a procedure to each step. Early in the development process, these procedures might be minimally functional. For example, the procedures may have all their validation in place but the code might not be optimised, however it will be sufficiently complete to do a test-run which will assess whether the system is working and the planning was correct.

## Lesson activity suggestions

| Topic | Lesson suggestions | Follow-up areas/ Homework |
|---|---|---|
| Write a procedure. | **Starter:**<br>Introduce programming techniques using a visual tool (e.g. Python Turtle or any of the Logo clones)<br>**Main activity:**<br>Demonstrate the basic commands of the chosen tool and then ask the students to write a program to draw a triangle. They should next write a program to draw a rectangle. The students should then wrap their code for the two separate shapes in individual procedures triggered from the procedure called main. Finally, they should create a procedure that can do any shape given a number of sides (e.g. polygon(5)) which would result in a pentagon.<br>**Plenary:**<br>Discuss with the students the efficiency of having a generic polygon function rather than having to write specific functions for each shape. | |

| | | |
|---|---|---|
| Increase the students' experience of procedures. | **Starter:**<br><br>Most students will have previous experience of procedures. Ask them how they can view one function inside another (n.b. some languages like Python and JavaScript allow this, most others don't).<br><br>**Main activity:**<br><br>Share the following example with your students (it is not good programming practice but food for thought) of a procedure inside another procedure using Python Turtle:<br><br>Code<br>`import turtle #connect to a framework`<br>`a=turtle.Turtle() #create a new object of the Turtle class`<br><br>`def square(): #define the outside procedure`<br>`  def fd_left(): #define the inside procedure`<br>`    a.forward(100)`<br>`    a.left(90)`<br><br>`  for i in range(4): #call the inside procedure 4 times`<br>`    fd_left()`<br><br>`square() #call the outside procedure`<br><br>Which produces this output:  <br><br>If possible, ask the students to set up and run this program.<br><br>**Plenary:**<br><br>Use the following questions to prompt a discussion:<br><br>1. When would one use global variables and when would it be better to use parameter passing?<br>2. What differentiates functions from other procedures?<br>3. Is a function a variable?<br>4. What are we looking for in creating descriptive procedure names?<br>5. Why is it possible to define procedures in a different order from the one they will be called? | |

| | | |
|---|---|---|
| Reflect on the order of steps needed to solve a problem. | **Starter:**<br><br>When planning procedures, we use pseudocode and/or flowcharts. Each procedure will have its own flowchart.<br><br>In the Object-oriented Paradigm, related procedures are grouped together in classes, which act like folders for code allowing programmers to be more organised. Once inside a class, procedures known as **methods** are named using dot notation. For example, Turtle.left(90) describes an object representing a turtle. The notation left describes a method that rotates the turtle to the left, and (90) is a parameter value of 90 degrees which will be added to the current heading, i.e. direction of the turtle. Heading is an attribute of this particular turtle (there can be multiple turtles), and is therefore a variable local to the object. If the turtle in question was facing up (it would have an initial heading of 90 degrees, depending on version and installation settings), after applying the method it will now have a heading of 180 degrees, facing left.<br><br>**Main activity:**<br><br>Ask the students to produce a flowchart to represent the following sequence. You could ask students to run the sequence on a simulation, if the procedures are available to call.<br><br>  <br><br><br>**Plenary:**<br><br>Have a group discussion to decide what has been learnt from this activity. What next steps would the students apply to develop the sequence further or to write alternative sequences? | |

| | | |
|---|---|---|
| Widen the students' experience of procedures and function. | **Starter:**<br>Explain that functions and procedures operate in many different programming languages. Scratch 2.0 uses procedures. Snap supports functions.<br>**Main activity:**<br>Ask the students (in small teams) to complete a task which they have designed using Python Turtle. They should also complete the same task using procedures in Scratch. Then complete the same task in Snap using functions.<br>Students should use a flowchart to consolidate their ideas.<br>**Plenary:**<br>Showcase the students' ideas to the class, using their flowcharts to explain. | More able students could research a computing concept called Closure and another sophisticated concept of first order functions. |
| Understand parameter passing. | **Starter:**<br>Most modern languages support Structured Query Language (SQL) and as database operations are often repetitive, they lend themselves to procedural implementation.<br>**Main activity:**<br>Using SQL to demonstrate parameter passing and the dot notation would be a good idea, as SQL is easy to read and students can follow the logic of query parameters etc.<br>**Plenary:**<br>Ensure that your students understand how important it is to pass parameters between program sections, subroutines, etc. | It may be worthwhile for students to research SQL views which are the most equivalent to procedures in SQL. Information about views can be obtained via this link. |
| Understand database normalisation. | **Starter:**<br>Discuss database normalisation with your students.<br>**Main activity:**<br>Create an activity to explore database normalisation using primary- and foreign-key constraints to illustrate parameter passing.<br>**Plenary:**<br>Discuss what the students have learnt from this activity. | |

## End of chapter answers

1 Procedural thinking describes the process of thinking through a problem and developing a sequence of steps that can be used to reach a solution.

2 A procedural programming language is one that solves problems by executing a sequence of instructions.

3 Selection\branching.

4 Iteration.

5 Sequence.

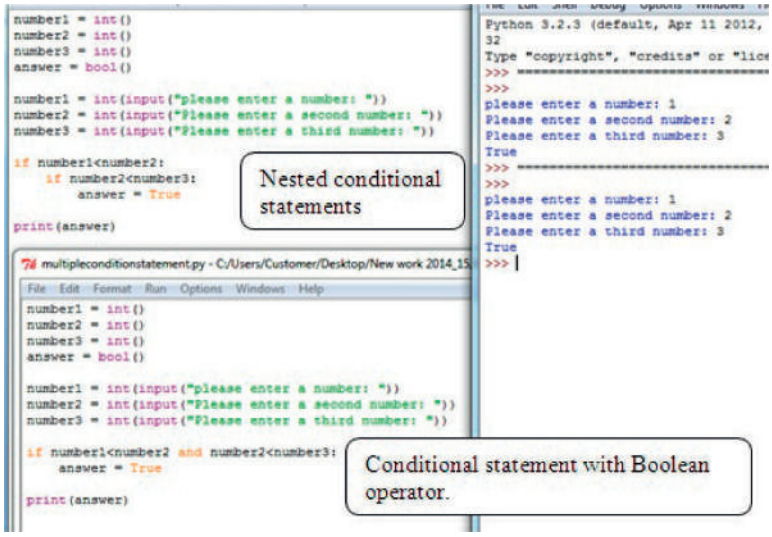
# Chapter 4 Thinking logically

## Learning objectives

- To be able to identify the points in a solution where a decision has to be taken.
- To be able to determine the logical conditions that affect the outcome of a decision.
- To be able to determine how decisions affect flow through a program.

### What your students need to know

- How to be able to identify in an algorithm where decisions need to be implemented.
- Logic is a fundamental component of Computer Science. Students need to be able to break problems down and identify the input and output of an algorithm in order to determine the program flow. Elements of formal logic should be introduced to the students to provide a structured mechanism for determining conditions. Propositional logic evaluates statements to a value of TRUE or FALSE. Compound logic statements use connectives such as AND, OR and NOT. The basic tool of propositional logic is a truth table. These are developed to give the output values from all possible combinations of input.
- Predicate logic uses a different approach. Facts and rules are defined to evaluate a query in order to find a true match.
- How selection can be used in an algorithm to control program flow.
- Students who previously studied GCSE Computer Science will have already created programs that use simple IF statements and a single operator (e.g. to test whether a value is equal to a given number). They should be confident in writing such programs. They should be aware of a wide range of logical operators and be able to use combinations of these to produce the desired conditional statement.
- How multiple conditions can be handled descriptively in pseudocode and practically in a program language.
- Program code can be written using nested statements or by having multiple conditions in a single statement. The following screenshot shows two simple programs that check whether three numbers entered by a user are in ascending order.



- How precedence affects how conditions are executed (e.g. nested conditions and the use of logical operators).
- As illustrated in the program above, the nesting instruction ensures that the outermost conditions are checked first before the inner conditions. If the outer condition is *not* true, the program will not proceed with the inner condition check. It may be that conditions are required to be checked independently of one another regardless of whether a previous statement has been matched. Separate conditions can be constructed as illustrated below.

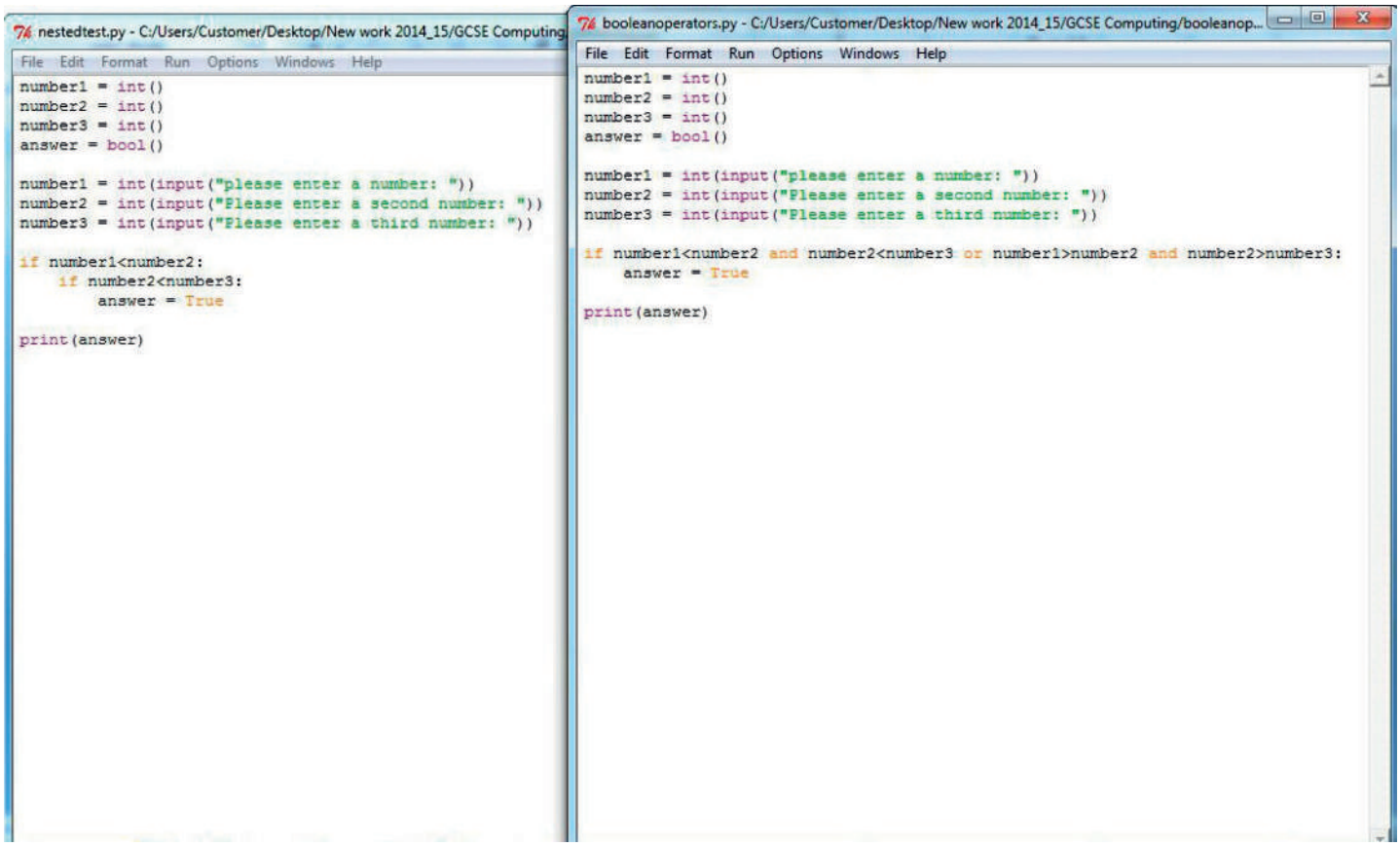
```
number1 = int()
number2 = int()
number3 = int()
answer = bool()

number1 = int(input("please enter a number: "))
number2 = int(input("Please enter a second number: "))
number3 = int(input("Please enter a third number: "))

if number1<number2:
    answer = True
    print(answer)

if number2<number3:
    answer = True
    print(answer)
```

- In the following example, logical operators are used to test whether a user has entered a sequence of three numbers in ascending or descending order.



- About logical operators and their related truth tables.
- Students should know and have used a variety of logical operators:

| Operator | Meaning | Example |
|---|---|---|
| > | Greater than | A>B will return TRUE if the value of A is higher than the value of B otherwise it will return FALSE. |
| < | Less than | A<B will return TRUE if the value of A is less that the value of B otherwise it will return FALSE. |
| <= | Less than or equal to | A<=B will return TRUE if A is the same as or smaller than B otherwise it will return FALSE. |
| >= | Greater than or equal to | A>=B will return TRUE if A is the same as or larger than B otherwise it will return FALSE. |
| != | Not the same as | A!=B will return TRUE if A is not the same as B but FALSE if A is the same as B. |
| NOT | The opposite of | NOT(A) will return TRUE if A is FALSE and FALSE if A is TRUE. |
| EQUALS (usually ==) | The same as | A==B will return TRUE if A is the same as B otherwise it will return FALSE. |
| AND | Both statements must be true for the argument as a whole to be true. | (A == 1) AND (B==4) will return TRUE if A is 1 and Y is 4. It would return FALSE in all other situations. |
| OR | Only one of the statements needs to be true for the argument as a whole to be true. | (A==1)OR(B==4) will return TRUE if A is 1 or B is 4. It would only return FALSE if A wasn't 1 and B wasn't 4. |
| XOR | The argument is false if both statements are true.<br>The argument is false if both statements are false.<br>Otherwise the statement is true. | A XOR B would return true if A and B were different values. |

*Image taken from Student Book*

- They should also be able to understand and be able to produce truth tables for logical operators:

| A | B | OR | AND | XOR | > | < | <= | >= | == | != |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

*Image taken from Student Book*

- How logical operators can be used in combination to achieve the desired output.
- Students need to be able to interpret a problem in terms of input and output data required and how logical operators can achieve this.
- The process of dry running an algorithm to test program logic.
- Students should gain experience and confidence in using trace tables and documenting paths of execution.

## Common misconceptions

- Students should be encouraged to switch their practice of 'trial and error' programming to developing and testing algorithms in pseudocode before starting to program. They are often too focused on a solution and do not take sufficient time to consider the problem holistically. Remind students that developing an algorithm can be completed separately from coding as, at an early stage, the target language may not be known. It is more important to encourage a general problem-solving approach. Students should not become restricted by their capabilities in any one language.

- An understanding of binary logic and truth tables is also important. Students should have encountered these for AND, OR and NOT operators at GCSE level and will have already interpreted combinations of individual logic gates. However, these can be easily confused and the significance of a **binary 1** over a **binary 0** may be overlooked. It is useful to consolidate and expand upon these concepts.
- Inputs and desired outputs need to be interpreted in terms of program logic. Sometimes students find it difficult to get started. It may be helpful to use general logic problems so that students think about problems that are not related to programming. There are numerous examples online and some links are provided at the end of *Chapter 4* in the Student Book.
- Although simple selection constructs will have been encountered at GCSE level, students will not have used nested statements before. A key issue to understanding them is the concept of precedence. The order that statements are executed in is critical in generating the correct output. To avoid errors, students should be encouraged to design their algorithms in a stepwise fashion and consider the output from each line (which becomes the input for the next). It may be helpful to provide examples of problems for which an algorithm can be designed and tested without necessarily having to write a program.

## Lesson activity suggestions

| Topic | Lesson suggestions | Follow-up areas/ Homework |
|---|---|---|
| Identifying decisions. | **Starter:**<br>True or False? Working in small groups, students decide whether an algorithm's output would be true or false depending on the input provided.<br>**Main activity:**<br>Correct Change Algorithm. Students will work through the stages of developing an algorithm for an automatic Point of Sale system. The algorithm should take the cost of an item being purchased and add it to a total cost. The algorithm should also take the amount of money that a shopper has tendered, calculate the change required and show how this should be tendered. The lesson should be a paper activity and will require students to determine where in their programs decisions and logical operations are required.<br>**Plenary:**<br>Discuss the potential drawbacks of writing a program with multiple conditional statements. How does this impact on its performance? What could happen if an algorithm is designed incorrectly? | Design and develop a text-based adventure game. Students will have to think through the logic of a program with an added element of fun. They should consider different themes for their game and have different pathways that lead to different outcomes for a player. |
| Logical conditions and program output. | **Starter:**<br>Spotting Errors.<br>Students work in small groups to identify errors in program code printed onto cards.<br>**Main activity:**<br>Reaction timer.<br>Students write a reaction timer program which will reward the player who presses a key on the keyboard when they believe that a set amount of time has elapsed. This activity will allow them to develop an algorithm that requires the use of logical operators in sequence to meet the requirements of a brief.<br>**Plenary:**<br>Group discussion on the topic of software testing. What techniques are effective in systematically testing program logic? The discussion should include tracing paths of execution. | Students are provided with program code in a language that they are familiar with. The code will contain logic errors. Students should test the program to identify and document the errors. As an additional exercise, they could correct the errors. |

| | | |
|---|---|---|
| Controlling the flow of a program using decisions. | **Starter:**<br>Noughts and Crosses.<br>Students consider a suitable starting move and subsequent moves for a fictional game of Noughts and Crosses. They will need to consider viable options when an opponent's move has been given to them.<br>**Main activity:**<br>Paired programming challenges.<br>Students complete sets of programming challenges that are ranked Green when easy (simple selection), Amber when intermediate (WHILE loops and multiple conditions IF, ELSE IF, ELSE and SELECT CASE) and Red when advanced (multiple logical operators, nested IF statements).<br>**Plenary:**<br>Evaluation of the paired programming activity and completion of an 'exit ticket' covering three questions:<br>1. What skill have you developed in completing this lesson?<br>2. Which programming technique have you found most challenging and why?<br>3. Please write one question that you would like to discuss in the starter of the next lesson. | Research expert systems and their uses. How can these be used to make decisions? |
| Understanding logic in Computer Science. | **Starter:**<br>Discuss what is logic? Students should develop their own definition. What is a logical statement? Can all statements be evaluated as TRUE or FALSE?<br>**Main activity:**<br>An investigation into propositional and predicate logic.<br>A short introduction should be provided that covers these two branches of logic. Students then investigate both and produce a brief summary of each and their relevance/importance in Computer Science.<br>**Plenary:**<br>Group discussion.<br>Is it possible to have a logic value between 1 and 0 (true and false)? What is fuzzy logic and what are its applications? | Students are provided with a series of fun logic-puzzles to think about logic from a non-programming perspective. |

## End of chapter answers

1 FALSE.

2 FALSE.

3 TRUE.

4 FALSE.

# Chapter 5:Thinking concurrently

## Learning objectives

- To be able to determine the parts of a problem that can be tackled at the same time.
- To be able to outline the benefits and trade-offs that might result from concurrent processing in a particular situation.

### What your students need to know

- That *processes* and *threads* are basic execution units. They should know how programs can be written to implement *concurrency* and how this can improve efficiency and performance. They should also understand how modern computer systems facilitate *parallelism* (the ability to run multiple threads or processes simultaneously) in addition to traditional concurrency (the ability to run multiple processes or threads but traditionally using a scheduling algorithm, such as **round-robin**). They should appreciate that the latter is also possible on single-core CPUs. Examples should be provided to illustrate the key principles. Maintaining multiple client connections and programs that require several simultaneous pathways should be discussed. Students should have an opportunity to identify where threads could be programmed in an algorithm.
- The issues related to the implementation of threading (e.g. synchronisation, starvation, deadlock and data races). Students should also appreciate that programs that implement threading are more complex to write. When badly written, programs can cause issues such as deadlock and starvation. Discuss classic algorithms that illustrate these problems. Students should also appreciate the states that threads can exist in: ready, running, waiting and stopped.

### Common misconceptions

- The difference between concurrency and parallelism is a subtle one. Concurrent threads on a single-core CPU are, in reality, being swapped in and out of the processor. Multi-core CPUs permit true parallelism where threads can run simultaneously. Students will appreciate the principles of concurrency more effectively if they can relate them to their understanding of CPU architecture.
- Students should appreciate that implementing concurrency can make programs more complex to write but the gains are always positive. Scheduling algorithms and writing a more complex program relies on more advanced programming skills but the advantages are improved performance during execution. In contrast, poorly written code can cause synchronisation issues which can lead to less efficient programs.

## Lesson activity suggestions

| Topic | Lesson suggestions | Follow-up areas/ Homework |
|---|---|---|
| Identifying where concurrency can be implemented. | **Starter:**<br>Students are asked to make ten paper models as quickly as possible. They can choose how they are going to tackle the challenge. It is likely that they will utilise some form of parallelism. They should then be challenged to think of ways to speed up processing but with a single person (i.e. single-core).<br>**Main activity:**<br>Concurrent programming. Students are challenged to write a simple program that implements threads which start at different intervals during the program. They should draw a simple timeline for their program to show when the threads start and end during the program's lifetime.<br>**Plenary:**<br>Group discussion about the advantages and typical uses of concurrent programming. | Preparing a mind-map is a useful way to appreciate the links between concurrent programming, CPU structure and operating system function which all support multi-threading. |
| Advantages and disadvantages of implementing concurrent programming. | **Starter:**<br>Concurrent programming keyword anagrams. This activity will help to reinforce key terms. As an additional activity, students could write definitions for the words that they have deciphered.<br>**Main activity:**<br>Students research the key advantages and potential disadvantages of concurrent programming. From their findings they will produce a short informative video that would be helpful to A level students.<br>**Plenary:**<br>What next for concurrent programming? Students should discuss those aspects of concurrent programming which are less desirable (e.g. deadlock or starvation, data races, issues due to shared memory space etc.). Challenge your students to consider how these issues could be overcome. | Produce an information leaflet to accompany their video completed in the main activity. |

## End of chapter answers

1 Programs take less time to run.

2 Deadlock; Starvation; Race conditions.

3 They must be completely independent of one another.

4 True.

5 Threading.

# Chapter 6 Programming techniques

## Learning objectives

- To understand the three main programming constructs: sequence, iteration and branching.
- To understand recursion, how it can be used and how it compares to an iterative approach.
- To learn about global and local variables.
- To understand modularity, functions and procedures, and parameter passing by value and by reference.
- To know how to use an IDE to develop/debug a program.

## What your students need to know

- The characteristics of procedural programming.
- Global and local scope of variables.
- The characteristics of object-oriented languages including encapsulation, inheritance and polymorphism.
- The advantages of defining classes and code reuse.
- The differences between classes, objects, attributes and methods.
- The advantages of encapsulation, inheritance and polymorphism.
- How stacks are implemented within programs.
- The purpose of assembly language.
- Direct, immediate, indirect, relative and indexed modes of addressing.
- How programming constructs are implemented in assembly language (e.g. iteration and selection).
- How the Little Man Computer (LMC) simulator can be used to understand assembly language programming.

## Common misconceptions

- Students need to understand the fundamental differences between different programming paradigms. It is also helpful if students learn example languages and understand their typical use. There are a number of technical terms associated with paradigms. A glossary or wiki can be of benefit. Addressing modes can also be the cause of misunderstandings. Clear definitions and examples are helpful but students should be mindful of the similarities of terms (e.g. indexed and indirect).

## Lesson activity suggestions

| Topic | Lesson suggestions | Follow-up areas/ Homework |
|---|---|---|
| Procedural languages. | **Starter:**<br>Provide sequencing instructions to complete a simple task (e.g. drawing a square on a whiteboard with a marker pen or similar activity). Students should then plan the algorithm and, once complete, pass it to another who then 'acts out' their instructions. Explain that procedural languages execute instructions in a particular order and that the program code describes how it will run.<br>**Main activity:**<br>Investigation of a procedural language (e.g. Python). Students will write simple procedural code such as calculating the number of rolls of wallpaper needed to paper a room or the area of a circle.<br>**Plenary:**<br>Produce a procedural programming factsheet. | Simple programming exercises involving mathematical calculations (e.g. averaging numbers, calculating a discount, times tables or finding areas or volumes). |
| The object-oriented Paradigm. | **Starter:**<br>Begin the lesson with a brief introduction to the terms: class, object, attributes and methods.<br>**Main activity:**<br>Define the key characteristics of object-orientated programming. Students can investigate pseudocode or, preferably, an environment such as Eclipse Java.<br>**Plenary:**<br>Compare object-orientated programming to procedural and discuss the strengths, limitations and typical uses of each. | Write a fictional magazine article about the history and uses of Java. |
| Assembly language programming. | **Starter:**<br>Discuss simple devices where computers might be installed (e.g. in a car). What is the computer's purpose in the chosen device?<br>**Main activity:**<br>Create a board game using 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 extended to include storage of instructions, data and memory addresses in registers on the CPU.<br>**Plenary:**<br>Discuss the uses, disadvantages and advantages of assembly language. | A series of challenges that start with simple activities and progress to harder ones. These should be assigned according to students' abilities. |
| Practical experience of assembly language. | **Starter:**<br>Give the students an introduction to the LMC. Show them how to download an emulator.<br>**Main activity:**<br>Give your students a series of challenges using the LMC. Provide a range of increasingly more difficult challenges.<br>**Plenary:**<br>Explain that in the early days of computing, machine code and assembly language were the only ways of input. Explain that computers still rely on these but that we now have interpreters, assemblers and compilers to help. | Students could research and identify the early computer developments at this time in history as this will be useful to them.<br>Additionally they could complete the necessary research to know the difference between interpreters, assemblers and compilers. |

| | | |
|---|---|---|
| Memory addressing modes. | **Starter:**<br>Talk about memory addressing modes and explain how these might have improved the programs created in their last lesson. Provide students with a list of mnemonics. They can fill in the second column with what they believe the instruction to be. Students should also write memory address or no memory address next to each to indicate which need to use a memory address (e.g. HLT (halt) should be no memory address and STA (store) does require one).<br>**Main activity:**<br>Different addressing modes. Students should investigate different addressing modes and produce an interactive poster or presentation to illustrate how they work.<br>**Plenary:**<br>Discuss how different addressing modes are used. Review why different modes are needed.<br>It may be useful to consider an additional lesson that gives the students the opportunity to test out their new knowledge of addressing modes on the LMC emulator. They could be given some tasks that only really work with relative addressing for example. | Web-based quiz on the characteristics of different programming paradigms. |

## End of chapter answers

1 A recursive function is one which repeatedly calls itself until a base case is met.

2 Sequence, selection/branching.

3 A sequence of instructions is repeated a set number of times or until a condition is met.

4 In recursion, a new version of the function is created in memory every time it is called, taking up a lot of memory.

5 A local variable is only available inside a particular subroutine but a global variable can be accessed from anywhere in the code.

# Chapter 7:Computational methods

## Learning objectives

- To understand problem recognition.
- To understand problem decomposition.
- To know when to use divide and conquer.
- How to solve problems using backtracking; data mining; heuristics; performance modelling; pipelining and visualisation.

### What your students need to know

- In order to develop an algorithm a problem must be solvable. Solvable problems are defined using the principles of logic.
- Decidable problems have a true or false answer and will not loop indefinitely. The halting problem is the problem of determining whether a given algorithm and input is decidable (i.e. will finish running) or not within a finite time.
- A Turing machine is a hypothetical problem-solving device.
- Undecidable problems include Entscheidungsproblem, the halting problem and the **Kolmogorov complexity of strings**.
- One algorithm can be used to check another algorithm, for example in the blank tape problem. Reduction is a technique when one algorithm is used to make another solvable.
- Decomposition is the process of breaking a problem down into its constituent parts. Abstraction filters out unnecessary detail in order to focus on the general requirements, including data structures, that are required to solve the problem.
- Divide and conquer describes a process where irrelevant choices for solving a problem are discarded in order to narrow the focus and increase the chance of success with each reduction. Examples include binary search and quick sort. Divide and conquer routines are recursive in nature.
- Parallelism can also employ divide and conquer to work on multiple parts of a problem at the same time (e.g. handling data in arrays). Parallelism is linked to concurrency. Concurrency is the ability to have multiple parts of processes executing at the same time.
- Backtracking is the process of searching for alternative solutions to a problem even when a solution may have already been found. This term is often associated with declarative programming.
- Data mining is the process of analysing large amounts of data to look for patterns or trends. Complex, fast algorithms are required. This technique is widely used by large commercial organisations to refine their services and products.
- Heuristics is an approach to solve problems in the least time. Examples of heuristic algorithms include modern routing protocols, firewalls, Intrusion Protection Systems (IPS) and virus scanners.

- Performance modelling measures system performance and seeks to reduce latency (e.g. queueing). This is usually achieved via simulation and refinement of the current system. Performance modelling is often used in networked systems.
- Pipelining maximises the use of resources by initiating the execution of an instruction before the previous one has completed. Modern processors make extensive use of pipelines and logical execution threads.
- Visualisation is a problem-solving technique often deployed in everyday life to solve commonplace activities. The more abstract the problem becomes, the more difficult it becomes to visualise.

## Common misconceptions

- A number of aspects of this topic can be misunderstood. Students need clear definitions of problem-solving terms and understanding of the distinctions between them (e.g. abstraction and decomposition, concurrency and parallelism).
- Students may not always be able to decide which technique is best to solve a problem. However, it is vital that they know how to recognise whether or not a problem can indeed be solved. Although the theory of solvability involves complex mathematics, *all* students need to know and appreciate that they cannot write an algorithm for an unsolvable problem.
- Students often find the concept of recursion challenging. A useful approach to the subject is to combine programming experimentation with reasoned algorithm development. This way, students can more easily appreciate that there are three possible outcomes. Firstly their algorithm functions correctly because they have included a base case and the correct return value is given. Secondly they have *not* coded a base case (or it is incorrect) resulting in infinite recursions (or until maximum depth is reached). Or thirdly that a step has been incorrectly coded giving the wrong return value to the next call. Experimenting with code and repeatedly asking the question 'why is this right?' can often be a more successful learning method than that of tackling just the algebraic representation. This method can also be particularly useful for students who have less mathematical knowledge or understanding.

## Lesson activity suggestions

| Topic | Lesson suggestions | Follow up ideas/Home-work |
|---|---|---|
| Features that make a problem solvable by computational methods. | **Starter:**<br>Students should discuss some solvable and unsolvable problems in everyday scenarios. For example, the students have purchased several items at a supermarket but when they get to the checkout, they find they don't have enough money to pay for it all; they are £1 short. This is a solvable problem as the students can put items back on the shelf to the value of £1 or more.<br>**Main Activity:**<br>Students will now have a basic understanding of the problem of solvability. They will also be aware that a decision should result in one of two outcomes: true or false.<br>**Plenary:**<br>Discussion of unsolvable algorithms. How can students recognise problems which are unsolvable? | Research the concept and basis of the Turing machine. |
| Problem recognition. | **Starter:**<br>Complete the Harold the Robot activity available from CS unplugged site. (Students are asked to write an algorithm for carrying out an activity such as drawing a square on a whiteboard, painting their nails with nail varnish, placing beads in a jar and screwing on the lid etc.)<br>**Main Activity:**<br>Small groups of students (up to four) should each be given a card with an algorithm name on it (e.g. process scheduling, routing, binary search, quick sort, merge sort etc.). Within their groups students write the algorithm, then use the checklist of requirements for solvable problems against their solution to check for correctness. This should confirm that the algorithms provided are indeed solvable.<br>**Plenary:**<br>Can algorithms 'learn'? If an algorithm does not provide a perfect solution, can it refine itself to provide a closer solution? Group discussion leading into a homework task. | Investigate shortest-path-first algorithms. |
| Problem decomposition. | **Starter:**<br>Ask students to plan a route to a point of interest in your local area. They can travel by any means but need to identify the route which costs the least to complete.<br>**Main Activity:**<br>Working in groups, students should design a new 'casual game'. The 'casual' genre of games consists of puzzles or simple problem-solving challenges with increasing levels of complexity/difficulty. There are many commercially available examples that the students will be familiar with. The students' game needs to include a scoring system and multiple levels. Students must use decomposition to break down the whole problem into constituent parts. Elements of abstraction can also be included. If time is available, students should present their game ideas to the rest of the class.<br>**Plenary:**<br>Can the skills used in decomposing problems help in everyday life? Discuss examples where this might be the case (e.g. cooking a meal). | Produce a poster that presents the algorithms used in game programs. |

| | | |
|---|---|---|
| Use of divide and conquer. | **Starter:**<br>Present students with a practical challenge that will demonstrate the concept of divide and conquer that they should solve in small teams. An example would be to provide a computer with intentional faults (e.g. the monitor is not switched on or a piece of black insulating tape is placed over the LED of an optical mouse). The students should provide a list of typical faults and symptoms. They should complete a table explaining how they eliminated some faults to narrow down possible causes. They can also produce a troubleshooting flowchart (or this could be saved for the plenary).<br>**Main Activity:**<br>Investigate *binary searches* and *quick sort* as examples of algorithms that use divide and conquer. Develop an example of a Twenty Questions game in which questions for which the answer is true or false determine the pathways.<br>Investigate recursive algorithms such as the Fibonacci sequence and the need for a base case.<br>**Plenary:**<br>Provide students with classic recursive images (e.g. patterns drawn by M. C. Escher). Can the students recognise the patterns of recursion? | Students should write a recursive routine to draw a pattern. It will be helpful to provide some background on fractals as a starting point. |
| Abstraction. | **Starter:**<br>Have a discussion of the term 'abstract'.<br>**Main Activity:**<br>Students could complete the 'Knight's Tour' and the 'Tourist Bus' available from CS4FN. These activities will reinforce the principle that by removing details, we begin to see general patterns and trends.<br>**Plenary:**<br>The importance of abstraction and abstract classes in object-oriented programming. | Students could produce a web page with content that acts as a guide for KS3/4 students explaining these key topics in Computer Science: abstraction, decomposition, pattern recognition and algorithm development. |
| Problem-solving techniques. | **Starter:**<br>Students are presented with the classic game known as Fox and Geese. They should investigate the possible outcomes including the quickest way for the fox to win and the quickest way for the geese to win.<br>**Main Activity:**<br>Produce a guide to computational thinking methods as a summary of all that they have learnt.<br>**Plenary:**<br>Discuss which approaches are likely to be successful in solving particular types of problems (e.g. problems involving large amounts of statistical data, a medical diagnosis problem, handling data in arrays etc.). | Produce a report that reviews the use of heuristics in algorithm development (e.g. in satellite navigation systems). |

## End of chapter answers

1 Abstraction could be used to separate key ideas from reality. For example they would want to focus on which stations are connected to each other (and by what line) rather than trivial details such as the size of the station. Symbols and colours can be used to represent reality.

Creating an algorithm which uses route finding would be the next step. Using a route-finding algorithm such as Dijkstra's algorithm would be a starting point. The tube map can be represented as a graph with each station represented as a vertex, each edge as a tube line and the weights as timings between them.

Considering if there are any existing components which could be reused will allow for a more robust solution. For example, using a library for Dijkastra's, or for representing a graph.

2 Divide and conquer is a programming technique which will repeatedly split a problem down into smaller parts until those parts become trivial to solve. In a concurrent environment each processor would be working on a different part of the problem. Therefore if a problem is split up into parts, each part could then be allocated to a different processor. This must be carefully managed and allocated by a single processor. The standard quick sort algorithm would have to be updated.

3 (a) A graph – representing the tube could be reused in other maps or route finding applications.

Algorithms used to perform route finding – For example A* or Dijkstra's.

(b)

Code

```
INPUT  Tube graph as G

INPUT station name as S

FUNCTION findDisabledStation(C, visitedStations)

visitedStations.add (C)

IF all stations visited THEN RETURN FALSE

FOR station IN C.connectedStations

        IF station.disabled == TRUE AND station NOT IN visitedStations  THEN

                RETURN station

        END IF

        findDisabledStation(station, visitedStations)
```

# Chapter 8 Algorithms

## Learning objectives

- To look at how algorithms are designed and analysed for a given situation or problem.
- To appreciate the suitability of algorithms for a given task with regard to execution time and space.
- To understand the complexity of algorithms using the Big O notation and use it to allow comparison between algorithms.
- To understand algorithms that are related to stacks, queues, linked lists and trees.
- To understand and use the following standard algorithms: bubble sort; insertion sort; merge sort; quick sort; Dijkstra's shortest path; A* algorithm; binary search; serial search.

## What your students need to know

- How to draw up a flowchart.
- How to describe a problem's solution in pseudocode.
- The use of **trace** tables and dry runs.
- Understanding of mathematical functions and calculus, including functional domains.

## Common misconceptions

- It should be noted that the Python examples given in this chapter are in Python 2 syntax which is different from Python 3 syntax, so modifications are needed to implement these examples on more modern installations (e.g. the print function now requires brackets, **raw_input** has been replaced with **input**, etc.).

## Lesson activity suggestions

| Topic | Lesson suggestions | Follow-up areas/ Homework |
|---|---|---|
| Develop a model algorithm, using flowcharts. | **Starter:**<br>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.<br>**Main activity:**<br>Create flowcharts to solve the mathematical equations using the following (suggested) steps:<br>1. Assess the problem. Students should be shown graphs of common functions, e.g. $y = x^2$, $y =$ , $y =$ and asked to identify them.<br>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.<br>3. Map out the solution algorithm. Students produce a flowchart of their proposed solution to the problem.<br>4. Assess the algorithm. Students 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.<br>**Plenary:**<br>Discuss how the students' programs ran and what problems they encountered. | |
| Progress from flowchart to pseudocode. | **Starter:**<br>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.<br>**Main activity:**<br>Code the flowcharts from the last activity in pseudocode, using the following (suggested) steps:<br>1. Create the main program which is going to take the inputs and call the various procedures as they are needed.<br>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.<br>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'.<br>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).<br>5. Stop the development process (temporarily) and have a discussion in the plenary with your students.<br>**Plenary:**<br>Explain 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. | |

| | | |
|---|---|---|
| Understanding domains. | **Starter:**<br>Following the starter with the mathematical functions' graphs, students should be asked to talk about the domains of functions (e.g. what might be interesting about the domain of the $y =$ function (zero is excluded), or the square root function ($x$ must be positive).<br>**Main activity:**<br>[Following the examples provided in this section of the student book, students may choose to take one or two algorithms they have done before instead of the example used below, and analyse their domains, ranges and codomains, as well as express the algorithms in pseudocode, paying attention to pseudocode conventions.]<br>Use the algorithm for deciding the winner of a rock/paper/scissors game. In the case of this game the domain is made up of combinations of two inputs which were the choices made by two players (e.g. rock/paper, scissors/rock, etc.), while the range will be: Player 1 wins; Player 2 wins; Draw.<br>Analysis could be followed up by actual implementation which can be timed.<br>**Plenary:**<br>Discussion of problems encountered in the process of performing the tasks. | Undertake the main lesson activity with an algorithm to give the minimum number of coins in change given a purchase price and the money given by a customer. This problem is actually quite similar to the tax calculation used as an example in this section of the student book, so it could be a nice extension activity. |
| Completing the programming project. | **Starter:**<br>The students should now be able to complete their program to solve selected quadratic equations by taking care to define the domains.<br>**Main activity:**<br>Complete the project.<br>**Plenary:**<br>Discuss the process of testing the algorithm on a computer and the steps required to progress from accuracy to performance. Explain how to time the execution of algorithms so that various alternative algorithms can be compared. This is a process known as benchmarking. Most modern programming languages use a built-in timer which enables an appreciation of an algorithm's complexity. | |

| | | |
|---|---|---|
| Complexity and scalability. | **Starter:**<br><br>Scalability is a significant issue when creating a program. It is always important to be able to predict how much an algorithm will slow down in response to a particular increase in data volume. If data volume increases tenfold, will the time taken to process the data also increase tenfold or higher? The complexity of an algorithm affects its performance. Some algorithms are fast with certain types of data and slow with others. Finding the most efficient (best performing) algorithm is therefore a key component of program development.<br><br>Ask the students to find the fastest way to fold a shirt or go around the classroom, touching every chair without stepping on the same place on the floor twice. The physical element of this exercise will ease the transition to this non-trivial section.<br><br>Alternatively, students might be asked to randomly stand and then sort themselves by height. (The last section of this chapter covers sorting algorithms, so it might be a good connection.)<br><br>**Main activity:**<br><br>A better task might be to write an algorithm to find all the English words with at least 2 letter O's in them. Alternatively the students could create a spell-checking program that when given a word, would look at a word list and decide if it was misspelled or not. The students could use a free words text file (used in Linux) located here. There is a 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 get involved in the later discussions about complexity.<br><br>**Plenary:**<br><br>The students should compare the speed of their algorithms with their classmates. | This is a compiled list of the 10 best algorithms. Do you agree with them? What would your top 10 list look like? |

## 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 $n \log n$ 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 $10 \times \log 10 = 33.2$. If bubble sort was used, which has a time complexity of $n^2$, 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. A balanced binary tree will limit the number of levels to $\log n$ meaning that, at worst, we have to compare $\log n$ items.

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/9781107531956
(Cambridge Elevate-enhanced Edition)



First published 2015

*A catalogue record for this publication is available from the British Library*

ISBN 978-1-107-53195-6 Cambridge Elevate-enhanced Edition

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


Acknowledgements
Acknowledgements
Cover provided by Fabian Oefner © 2013 www.fabianoefner.com