MY REVISION NOTES

HODDER EDUCATION

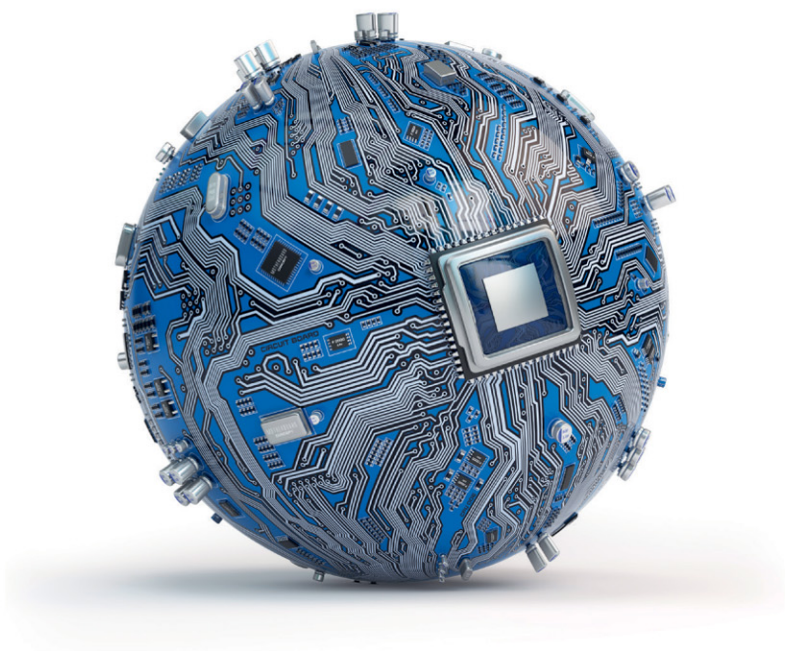
AQA

A-level

COMPUTER SCIENCE

THIRD EDITION

- Plan and organise your revision
- Reinforce skills and understanding
- Practise exam-style questions



Mark Clarkson

MY REVISION NOTES

AQA A-level

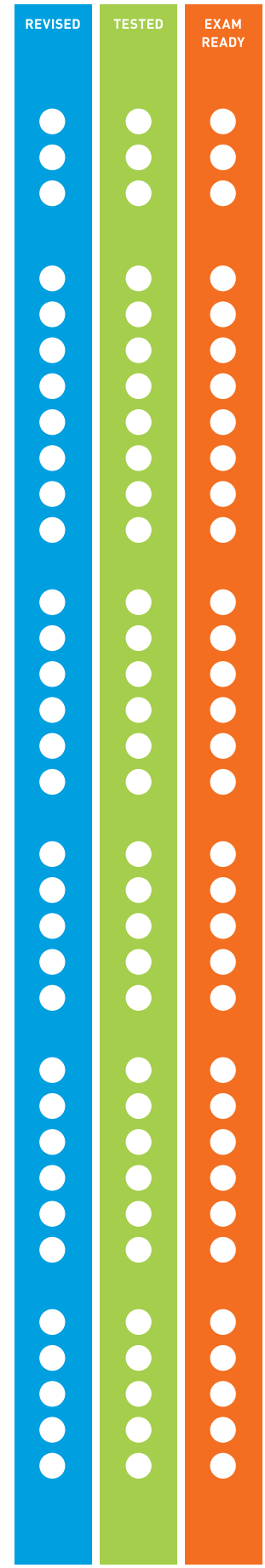
COMPPUTER SCIENCE

Boost

HODDER EDUCATION

LEARN MORE

My Revision Planner



Check your understanding and progress at www.hoddereducation.co.uk/myrevisionnotesdownloads

Introduction

As a student of computer science, it is important that you understand three things in relation to the A-level examinations.

- Assessment objectives
- Command words
- Exams and NEA

Assessment objectives

- AO1 You should be able to demonstrate **knowledge and understanding** of the principles and concepts of computer science, including abstraction, logic, algorithms and data representation.
- AO2 You must be able to **apply** your knowledge and understanding of the principles and concepts of computer science, including to analyse problems in computation terms.
- AO3 You must be able to **design, program and evaluate** computer systems that solve problems, making reasoned judgements about these and presenting conclusions.

Command words

Familiarity with the relevant command words is important. It helps you to avoid wasting time in the exam room (for example, trying to evaluate when there is no requirement for it). The most frequently used command words used for the A-level papers are listed here.

- **Calculate…** requires you to work out the value of something. A correct final answer, to the required degree of accuracy and with the correct units, will score full marks. Working is usually required, and correct working can score marks. (AO2)
- **Compare…** requires you to identify similarities and differences between ideas, technologies, or approaches. (AO1)
- **Create…** requires you to write program code that solves a problem. Even if you struggle to get the syntax correct, marks are awarded for evidence of the approach taken as well as for working code. (AO3)
- **Define…** requires you to specify the meaning of a technical term in order to show that you understand what it means. (AO1)
- **Describe…** requires you to set out the characteristics of a device or a computing concept. These can be very short questions worth 1 mark, or long-answer questions worth up to 12 marks. (AO1)
- **Discuss…** requires you to present the key points. These questions are typically of medium length, worth 4-6 marks. You should aim to present as many points as you can and, where appropriate, provide balance between advantages and disadvantages. (AO1)
- **Draw…** requires you to produce a diagram. These are usually technical diagrams such an E-R diagram for a database, or a logic circuit diagram. (AO2)
- **Explain…** requires you to provide purposes or reasons. These questions are used to assess your knowledge of a topic and can sometimes be used within a specific context. If a question is asked in relation to a particular scenario then you should always make sure you link to the scenario in your answer. (AO1, AO2)
- **Express…** requires you to convert an input into a particular format. Examples might include encrypting a message or calculating a value. (AO2)
- **Modify…** requires you to take an existing section of program code and add to or edit it. This is commonly asked as part of Section D in Component 1, involving changes to the skeleton program. (AO3)
- **State…** requires you to express some knowledge in clear terms. These questions are usually short answer questions and are usually worth 1 mark. (AO1)

- **Suggest…** requires you to present a suitable case or solution. (AO1, AO2)
- **Test…** requires that you run the code that you have written in order to check that it functions as it should. This is usually combined with a write/create/modify question in component 1 and screenshots of the testing process should be included. (AO3)
- **Write…** requires you to create a program or a Boolean expression to solve a specific problem. These questions can be quite lengthy and partially complete answers are usually worth a significant proportion of the marks. (AO2, AO3)

Exams and NEA

The AQA A-level in Computer Science is assessed using three components. There are two exams, both taken at the end of the course, and one piece of non-examination assessment (NEA) which is based on a programming project.

Component 1	Component 2	Component 3
On-screen exam	Written exam	Programming project (NEA)
2.5 hours	2.5 hours	Internally assessed
40% weighting	40% weighting	20% weighting

Component 1

Component 1 is an on-screen exam, largely focused on programming and computation. In advance of the exam, you and your teachers will be provided with a skeleton program and a small pack of background information.

The exam will be taken using a programming language you have studied throughout the course, and will be pre-selected by your teacher. The available languages are:

- C#
- Java
- Pascal/Delphi
- Python
- VB.net

The skeleton program is a working program that is functional but could be improved. Typical examples include text-based role-playing games or simulations.

For A-level students the pre-release information is available from September of that academic year (typically Year 13).

The pre-release information should be given to you by your teacher, but they may choose not to give the information out straight away, depending on how and when they plan to deliver the content.

The A-level exam is split into four sections:

1. **Section A:** Questions about programming and computation; for example, finite state machines, standard algorithms, trace tables, Turing machines and computation logic.
2. **Section B:** A programming problem (unrelated to the skeleton program); for example, writing a program to convert between binary and denary numbers.
3. **Section C:** Questions about the skeleton program; for example, identifying specific variables or programming constructs, hierarchy charts, class diagrams and explaining the purpose of specific elements.
4. **Section D:** Improving the skeleton code; for example, adding an extra menu option, improving exception handling, adding new functionality.

Component 1 tests programming ability as well as theoretical knowledge covered in Chapters 1–4, and the skills covered in Chapter 13.

Programming syntax

Some questions in Component 1 will include algorithms written using AQA pseudo-code. It is important to be able to read and understand this pseudo-code and, in some cases, to write program code in your chosen programming language based on the pseudo-code algorithms provided.

This book is not intended to teach you how to program using one specific language and uses pseudo-code similar to that used by AQA. Any code that is not labelled as being from a particular language is instead written in AQA pseudo-code.

Component 2

Component 2 is a traditional-style written paper, largely focused on the more theoretical components of this course.

The topics covered in this paper test subject content from Chapters 5–12, and typically include:

- data representation
- computer systems
- computer hardware
- consequences of computing
- networking
- databases and Big Data
- functional programming.

As this is a written paper, no practical programming activities are assessed, though some practical elements such as calculations, data conversions and trace tables (especially for assembly language and instruction sets) are likely to appear.

This paper will generally include short answer questions (1–2 marks) that will assess your knowledge, questions that will assess your ability to apply your knowledge (3–4 marks), and a small number of longer-answer questions that require detailed discussion (6–12 marks).

For these longer-answer discussion questions, credit is awarded for identifying, discussing and evaluating potential issues.

- Marks are awarded for identifying relevant knowledge; for example, suggesting appropriate input devices for collecting data or identifying the methods by which wireless data transmissions can be intercepted.
- To reach the top mark bands it is important to follow a line of reasoning, using your knowledge to write in connected sentences in a way that makes sense and relates to the context of the question. Explaining how each point links to the scenario and adding as much technical detail and vocabulary as you can makes it more likely that you will score well on this type of question.
- Always make sure you back up any arguments or suggestions you make with facts, logical arguments and technical details, as unsubstantiated statements don't demonstrate your understanding.

Component 3

Component 3 is a non-examination assessment (NEA) component, based on completing a programming project.

The programming project is extremely open-ended, and it is up to you to identify a real-world problem and then work through each phase of the systems development lifecycle in order to solve it.

It is beyond the scope of this book to go into detail in terms of completing the programming project, but typical examples include online booking and scheduling systems, computer games with a simple AI component, animal population simulations, and so on. There is no definitive list of expected or excluded projects and your teacher will be able to provide with much more specific guidance.

1 Fundamentals of programming

Programming

Data types

REVISED

It is important to declare variables using the correct data type. This will make sure that memory is not wasted, and that the program is able to process the data correctly.

Different data types are processed in different ways; for example, adding two strings produces a different result to adding two integers.

```
"123" + "456" = "123456"

123 + 456 = 579
```

The main data types are:

Data type	Suitable for	Examples
integer	whole numbers	`7, 9, 163`
real/float	fractional numbers	`7.3, 9.6, 0.5`
Boolean	True/False	`True, False`
character	a single letter or other character	`'b', 'W', '@'`
string	text	`"word", "EC4Y 0DZ"`
date/time	a date, time, or combination	`14:18, 17/10/2004`
pointer	a memory address	`#bf4e3a67`
record	a collection of fields	`RECORD username, password`
array	a table of data	`[1,3,5,7,11,13,17]`

Different data types use different amounts of memory. If a numeric variable will always hold a whole number, it will use less memory stored as an integer than it will if stored as a real or float.

Note

All programming languages deal with data types slightly differently.

It is important to note that Python does not support the character data type, using only a string to store text of any length. Python also does not support the array data type. The closest alternative is a list data type.

Most of the data types are quite straightforward.

- Records are more complex as they allow the programmer to group together attributes of different types; for example, a user record may have attributes for `Username`, `Password` and `Last_Login`.
- An array is a table-like structure. Each field in an array must be of the same data type (e.g. an array of integers, or an array of strings).

Making links

Arrays are an important data structure that are used to store sets of data together. The use of arrays can increase the efficiency of programs when processing lots of data. Arrays are covered in detail in the chapter on data structures.

Exam tip

There are many additional data types, some quite specific (for example, for whole numbers Java has a choice of byte, short, int or long). The mark schemes allow for any appropriate answer though only those listed above are strictly required.

Built-in and user-defined data types

Built-in data types are those already defined by a programming language. The exact list varies from language to language, but most of those in the table above are built-in to most programming languages.

It is possible for a user to define their own data types, combining more than one value per variable. The use of user-defined data types means that program code can be simplified, with attributes being grouped together. This can be carried out in various ways in different languages. A good example might be a user data type that is made up of a username and a password.

Now test yourself TESTED

1 What data type is most suitable for storing a whole number?
2 What is the difference between a character and a string?
3 What sort of data would be stored in a pointer?
4 Suggest two fields that could be included in a user-defined record for recording high scores on a game.

Answers available online

Exam tip

User-defined data types are mostly like to appear in the skeleton code. Look for any data types made up of other variables in the skeleton code and make sure that you are comfortable working with them in the syntax used in your chosen language.

Programming concepts

REVISED

For each of the programming concepts you should be familiar with both the **pseudo-code** used by AQA and the **syntax** used in your own programming language.

Declaration

Variable **declaration** is the process of creating a variable.

In languages which are strongly typed (for example, C#, Pascal/Delphi, Java and VB.net) the data type of the variable is stated, followed by the variable's identifier. In these languages it is possible to declare a variable without initially assigning a value to it.

- `int Age;` (C# and Java)
- `var Age: integer;` (Delphi and Pascal)
- `Dim Age As Integer` (VB.net)

In languages which are weakly typed (for example, Python) a value must be assigned to a new variable, and no data type is declared as the data type can be changed during the course of the program.

- `Age = 18` (Python)

Constants are declared in a similar way, but with a key word that indicates the variable cannot be changed. Because the value of a constant cannot be changed later, they must be declared with an initial value:

- `const int MAX = 3;` (C#)
- `Const MAX = 3;` (Pascal/Delphi)
- `final int MIN = 3;` (Java)
- `Const MAX As Integer = 3` (VB.net)

Python does not support the use of constants and therefore it is not possible to prevent the accidental assignment of a new value to a constant.

Making links

It is important to consider where a variable is declared. A variable declared inside an `IF` statement, loop or subroutine can only be used within that section of the code and will be destroyed once that section has ended. This issue is dealt with more in the section on local and global variables.

Pseudo-code A format for program code that is not specific to one programming language. Used extensively in Component 1. Pseudo-code is useful for describing an algorithm that could be coded in one of several different languages.

Syntax The strict rules and structures used within a specific programming language. You will only be assessed in the syntax of one programming language for Component 1.

Declaration The creation of a variable in memory.

Exam tip

In Component 1 you will be presented with pseudo-code algorithms to read, and to turn into program code, but you will not be expected to write answers using pseudo-code. Practical programming questions are intended to be coded using the specific programming language you have been entered for.

Assignment

Changing the value of a variable is called **assignment**.

- Assignment in AQA pseudo-code often uses a left arrow (for example, `Score ← 12`).
- Assignment in most programming languages uses an equals sign (for example, `Score = 12`)

Assignment Changing the value of a variable stored in memory.

Now test yourself TESTED

5 What is meant by the term *declaration*?

6 Describe one difference between the declaration of a variable and the declaration of a constant.

7 What symbol is used to indicate assignment in AQA pseudo-code?

8 What symbol is used to indicate assignment in most programming languages?

Answers available online

Sequencing

Program code is executed in the order in which it appears. For example, given the code:

```
x ← 5
OUTPUT x
x ← x + 1
```

The value 5 will be output before the value of x is increased.

Selection

All but the simplest computer programs need to include decisions. A simple decision of whether to execute a particular block of code is controlled by a **selection** statement, often implemented using an `IF` statement.

```
IF score > 90 THEN
    OUTPUT "Top marks!"
ELSE IF score > 60 THEN
    OUTPUT "Good job."
ELSE
    OUTPUT "Try again."
END IF
```

There are no limits to how many `ELSE IF` statements can be used, but with a large number it is possible to use a **SWITCH** statement instead. A `SWITCH` statement is slightly faster to execute if used with single values (rather than checking for a range)

```
SWITCH (grade)
CASE "A*" THEN
    OUTPUT "Top marks!"
    BREAK
CASE "A" THEN
    OUTPUT "Very good job."
    BREAK
...
END SWITCH
```

Selection A program structure that makes a decision about which block of code to execute, typically implemented using an `IF` statement.

SWITCH An alternative selection structure to an `IF` statement that is slightly quicker to execute if used with exact values rather than a range.

Exam tips

Remember that `IF` and `ELSE IF` commands need a conditional statement with a Boolean (True/False) answer. The `ELSE` command doesn't as it will catch all other possibilities.

You are not required to use a `SWITCH` statement and can score full marks using `IF-ELSE IF-ELSE`. The skeleton code or pseudo-code may include a `SWITCH` statement so you should make sure you are familiar with the concept.

Nested selection

Nested IF statements occur when an IF statement is placed inside an IF statement.

```
IF animal = "dog" THEN
    IF target = "sheep" THEN
        OUTPUT "Dog chases sheep"
    ELSE
        OUTPUT "Dog wags tail"
    END IF
END IF
```

Now test yourself TESTED

9 What programming concept is implemented using an IF statement?

10 Which part of an IF statement does not need a condition?

11 What alternative to an IF statement is sometimes used if there are several possible options?

12 What is meant by a nested IF statement?

Answers available online

Iteration

When a block of code needs to be repeated, this is referred to as **iteration**.

Definite iteration

Definite iteration, or **count-controlled iteration**, refers to the use a FOR loop, where the number of times to repeat is known. Even if the number of times to repeat isn't always the same, if it is known at the start of the loop then a FOR loop should be used.

You should be familiar with the syntax of a FOR loop in your own programming language as well as the pseudo-code you might see in an exam:

```
FOR i ← 1 TO 5
    OUTPUT "This is step " + i
ENDFOR
```

Indefinite iteration

Indefinite iteration or **condition-controlled iteration** refers to a loop where the number of iterations is not known. A typical example might be a validation loop, asking a user to enter a valid input and repeating while their answer is invalid.

Indefinite loops can be further split into those that assess the condition at the start of the loop (a WHILE loop) and those that assess the condition at the end of the loop (a DO-WHILE loop).

In AQA pseudo-code:

A WHILE loop will have the condition at the top of the loop, allowing the program to bypass the code inside completely. For example:

```
value ← 5
WHILE value < 100 DO
    value ← value * 2
ENDWHILE
```

Nested Placing a programming structure inside another programming structure.

Revision activity

- Write a program in your chosen programming language that uses an IF statement to allow the user to choose one of four options.
- Re-write the program to use a SWITCH statement.

Exam tips

Be very careful with the start and end conditions when creating FOR loops. While a pseudo-code algorithm may explicitly start and end at given values, the implementation for a FOR loop may be less clear (for example, in Python `for i in range(1,5)` would stop at `i = 4` and in Java `for (int i = 1; i < 5; i++)` would also stop at `i = 4`).

Although the specification refers to definite and indefinite iteration, question papers typically use the terms count-controlled and condition-controlled iteration.

Iteration The repetition of a process or block of code.

Definite iteration, or count-controlled iteration Iterating a fixed number of times (also known as a count-controlled loop, implemented as a FOR loop).

Indefinite iteration, or condition-controlled iteration Iterating until a condition is met (also known as a condition-controlled loop, implemented in AQA pseudo-code as a WHILE loop or DO-WHILE loop).

A DO-WHILE loop will have the condition at the bottom of the loop, forcing the program to pass through the code at least once. For example:

```
DO
    OUTPUT "Enter shoe size:"
    ShoeSize ← INPUT
WHILE ShoeSize > 12
```

Nested iteration

Nested iteration means having a loop within a loop. This is used in a number of applications, including when working with 2D arrays, and in a number of standard algorithms including the bubble sort.

```
FOR i ← 1 TO 3
    FOR j ← 1 TO 5
        OUTPUT "Outer loop = " + i + " | Inner loop = " + j
    ENDFOR
ENDFOR
```

Now test yourself TESTED

13 What type of loop is also called a count-controlled loop?

14 When should you use a condition-controlled loop?

15 Describe the difference between a WHILE loop and a DO-WHILE loop.

16 Suggest **two** possible uses for nested iteration.

Answers available online

Revision activity

- Write a program that uses iteration to print out the 12 times table.
- Write a program that uses nested iteration to print out all of the times tables from 1 to 12.

Meaningful identifier names

It is important to choose identifiers for variables (and subroutines) that tell other programmers something about the purpose of that variable (or subroutine).

- Var1, Var2 and Var3 are very poor choices as they make it very hard to read the code and to debug if there are any errors.
- UserName, UserAge and DateUserLastLoggedIn are much more meaningful and are key to writing self-documenting code.

It is not usually possible to use spaces in identifier names. Common strategies to aid readability include the use of CamelCase, Kebab-Case or Snake_Case.

Exam tip

In Component 1, Section B and Section D, it is very important to use any identifier names exactly as they are provided in the question. If the identifier name is not given explicitly then remember that making it easier for the examiner to understand the code makes it more likely you will pick up the marks.

Note

The use of meaningful identifier names to produce self-documenting code can have a significant impact on the marks available in the NEA.

Note

Python does not support the use of a DO-WHILE loop, but students are still expected to be familiar with the concept. One solution to this problem is to copy and paste the first iteration of the code before a WHILE loop. Another is to create a REPEAT … UNTIL structure.

Remember

Remember that if you know how many times to loop you should use a FOR loop. If you don't know how many times then use a WHILE or DO-WHILE loop.

Remember that the inner loop is completed multiple times for each step around the outer loop. Selection and iteration statements can be nested inside each other, potentially many layers deep.

Debug The process of identifying and removing errors from program code.

Self-documenting code Program code that uses naming conventions and programming conventions that help other programmers to understand the code without needing to read additional documentation.

Now test yourself

17 Describe **two** advantages of using meaningful identifier names.

18 Explain the term *self-documenting code*.

Answers available online

TESTED

Subroutines

Information on subroutines can be found in the section Purpose of subroutines.

Arithmetic operations

REVISED

There are several key operators you should make sure you are familiar with.

Simple arithmetic operators include those for addition (+), subtraction (-), multiplication (*) and division (/).

It is important to understand the different types of division operation that can be carried out:

- Real or float division will result in a real or float answer; for example, `5 / 2 = 2.5`
- Integer division will strip any fractional part of the answer, effectively rounding down; for example, `5 DIV 2 = 2`
- The modulo operator will find the modulus – the remainder of an integer division, as a whole number; for example, `5 MOD 2 = 1`

DIV and MOD are useful for converting between different number systems, including conversion between units of time, imperial measurements, and between numbers using different bases (for example, denary and hexadecimal).

Exponentiation refers to powers; for example, `2^3 = 2*2*2 = 8`

Rounding can be carried to a given number of decimal places or significant figures, generally using a function; for example, `round(3.14,1) = 3.1`, `round(3.16,1) = 3.2`

Additional functions can be used that always round up or always round down; for example, `roundup(3.142,1) = 3.2`

Rounding down has the same effect as truncation.

Exam tip

Some questions in Component 1 will assess your understanding of programming principles in general and may involve having to read and show understanding of pseudo-code. Other questions will require you to write your own program code in your chosen programming language, so it is important to make sure you are familiar with the specific operators and functions that are used in that programming language.

Exam tip

DIV and MOD are common terms for integer division and modulo. Make sure you are comfortable with how they function.

Operators Symbols used to indicate a function.

Modulus The remainder of the division of one number by another.

Exponentiation The raising of one number to the power of another.

Rounding Reducing the number of digits used to represent a value while maintaining its approximate value.

Truncation Removing any value after a certain number of decimal places.

Now test yourself TESTED

19 What is the difference between float division and integer division?

20 What is the value of 7.86 when rounded to one decimal place?

21 What is the value of 7.86 when truncated to one decimal place?

Answers available online

Revision activity

Write a program that uses DIV and MOD to convert a given number of hours into a number of days.

Relational operations

REVISED

Relational operators are used in comparisons, typically in selection statements and condition-controlled loops. A relational operation, or comparison operation, will always return either True or False.

The relational operators are as follows:

= or ==	Equal to
!= or <> or ≠	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Note

Different programming languages have different syntax rules for the equal to comparison operator.

AQA pseudo-code and VB.net use a single equals (=).

C#, Pascal/Delphi, Java and Python use a double equals (==).

Now test yourself TESTED

State whether each of these is True or False when `x ← 60`.

22 `x != 60`

23 `x < 60`

24 `x >= 60`

25 `x = 60`

Answers available online

Making links

C# and Java are not able to make use of the standard relational operators when comparing strings. String handling operations are discussed on page 16.

Boolean operations

REVISED

Boolean operations can be used to invert the logic of a conditional statement, or to combine two or more conditions together.

NOT	Will invert the logic (that is, a condition that returns True will become False).
AND	Both conditions must be True.
OR	Either one condition must be True, or both.
XOR	One condition must be True and the other False.

Making links

The exam for Component 2 assesses understanding of Boolean logic in much more detail, including questions on Boolean algebra. The same basic principles apply whether combining logical statements in a practical programming setting or solving Boolean equations. For a more in-depth examination of Boolean logic see Chapter 6.

Now test yourself TESTED

Where a = 50 and b = 100, would each overall condition be True or False?

26 `NOT (a = b)`

27 `a < 100 AND b > 100`

28 `a > b OR b = 2*a`

29 `a < b XOR b = 100`

Answers available online

Revision activity

Write a program to calculate the output of simple logic circuits for NOT, AND, OR and XOR.

Variables and constants

REVISED

Concept of a variable

Computer programs are essentially sequences of instructions for how to process data. This data must be stored in **memory** in order to be accessed and the computer program will save and retrieve data as needed.

Variables are made up of four components:

- A memory address – the location of the variable in memory.
- An **identifier** – a name used to identify the variable in the program code.
- A **data type** – a definition of what type of data can be stored in that variable.
- A value – the actual value of the variable at that moment in time.

Memory The location where instructions and data are stored in a computer.

Identifier A technical term for the name of a variable.

Data type A definition of what type of data can be stored in that variable.

Variables are used to store values that are used in a computer program. The value of a variable can be changed while the program is running (such as a score, a running total, a user's name).

A **named constant** is a variable whose value cannot be changed while the program is running. Examples might include constants used in calculations (such as Pi) or user-defined values that should be used throughout the program (such as a maximum number of turns or a maximum size).

It is a common convention to use all caps in the identifier for a constant; for example, `MAX_SIZE` or `MIN_VALUE` and to declare constants at the start of a program.

Named constants are useful because:
- they make it easier to update the program if the value of the constant needs to be changed
- it reduces the need to use absolute values (that is, it is better to use a constant in the condition for a loop than a hard-coded number)
- it stops the value being changed accidentally
- it makes code more readable.

Exam tip

Component 1, Section C will often start with asking you to 'state the identifier for…' some variable or subroutine in the skeleton code. For those questions you must only write down the identifier, and not the whole line of code.

Named constant A variable whose value cannot be changed while the program is running.

Exam tip

Remembering two or three standard advantages for each aspect of programming (such as named constants) will help you answer those questions quickly and accurately in the exam.

Now test yourself TESTED

30 What **four** components make up a variable?

31 What is the meaning of the term *identifier* for a variable?

32 What is the main difference between a variable and named constant?

33 What are the advantages of using a named constant?

Answers available online

String handling operations

REVISED

Working with strings is an important part of programming, and it is very important to be familiar with the specific syntax used in different languages.

Comparing strings is handled using different syntax in different languages.
- In AQA pseudo-code and VB.net, a single equals (=) is used to test for equality
- In Pascal/Delphi and Python, a double equals (==) is used to test for equality.
- In C# and Java a function must be used; for example, `if (StringOne.equals(StringTwo))` (C# and Java).

There are several other string handling operations with which you should be familiar. The exact syntax will vary significantly depending on your chosen programming language and it is important to make sure you are confident in carrying out all of these operations in that language. Some examples are included below, in a variety of different languages, in order to demonstrate each operation.

Concatenate Join two (or more) strings together; for example, ABC + DEF = ABCDEF.

The key operations are as follows.

Key operator	Example
Finding length of a string	`len(string)` or `string.length()`
Finding a character at a given position	`string.charAt(3)` or `string[3]`
Retrieving a substring	`string[0:6]` or `string.Substring(0,6)`
Concatenating two or more strings	`stringOne + stringTwo` or `concat(stringOne,stringTwo)`
Finding the numeric code for a character	`ord('D')` or `(int)'D'`
Finding the character from a given numeric code	`char(68)` or `chr(68)`
Converting strings into different formats, and vice-versa	converting a string into an integer, float or date/time format

Exam tips

Substring methods in different languages use different parameters.

- Python and Java require the start and end positions.
- C#, Delphi and VB.net require the start position and the length.

Make sure you read any pseudo-code questions that involve substrings carefully, and make sure you are comfortable programming with substrings using the language chosen for your exam.

Some languages treat a string as an array of characters and can use an index array to refer to a specific character; for example, `string[3]`. Some languages require the use of functions such as `charAt(int)`.

Make sure you are familiar with all of the operations above in your own language in advance of the Component 1 exam.

Substring A series of characters taken from a string.

Now test yourself TESTED

34 Explain what is meant by *concatenation*.

35 Which string handling operation should be used to extract a person's first name from the string `FullName`?

36 State and describe **two** other string handling operations.

Answers available online

Revision activity

- Write a program that will ask for a string and display each character's numeric code.
- Write a program that will ask for a series of numeric codes and convert them into a single string which is displayed.

Random number generation

REVISED

A large number of programs rely on random number generation.

Most languages have the ability to generate a random float between 0 and 1; for example, in Delphi:

```
float: Random;
```

Generating a random number in a specific range can then be achieved using multiplication; for example, to find a random float between 0 and 3 using VB.net:

```
Rnd() * 3
```

Most languages have a function for finding a random integer; for example, in Python:

```
random.randint(1,3)
```

Exam tip

In most years there is some random number generation included in the skeleton code. If you forget the correct syntax during the exam then make sure you know where to find a working example from the skeleton code to help you.

Revision activity

- Write a program that will ask for a minimum and maximum number and will generate a float between those two values.
- Write a program like that one above that will generate an integer between those two values.

Exception handling

REVISED

Exception handling is used to deal with situations where a program may crash. The program should try to execute a block of code and then catch the error and deal with it appropriately.

Although the exact syntax may vary between languages, the basic construct typically reads:

```
try
{
        strInput = input("Enter a number")
        numInput = int(strInput)
}
catch (Exception)
{
        print("Error, did not recognise a number")
        numInput = -1
}
```

Exam tip

Make sure you can explain the purpose of any exception handling found in the skeleton code as this may be a question asked in Section C.

Making links

Using exception handling in your NEA programming project is a good way to demonstrate good programming techniques.

Revision activity

Write a program that asks users to enter a 2-digit number as a string (such as '12'), convert it to an integer, and then double it. Use exception handling to catch errors such as a non-digit entry (such as 'twelve').

Now test yourself TESTED

37 What does the initialism RNG typically refer to?

38 What runtime error might occur in a program that asks a user to enter numeric values?

39 What programming structure should be used to deal with these types of errors?

Answers available online

Purpose of subroutines

REVISED

A fundamental aspect of improving the efficiency of program code is the use of **subroutines**.

A subroutine allows a programmer to take a section of program code that performs a specific task and move it out of line of the rest of the program. This means that the programmer can **call** the subroutine in order to run that block of code at any point.

Subroutines are called using the subroutine's identifier, followed by any arguments that must be **passed** in parentheses; for example, `DisplayGreeting(name)`.

Subroutines that do not require any parameters must still be called using parentheses; for example, `DisplayDate()`.

The advantages of subroutines are:

- the subroutine can be called multiple times without needing to duplicate the code
- changes to the subroutine only need to be made once
- it is easier to read the code
- it is easier to debug the code if there is a problem
- subroutines can be re-used in other programs
- the job of writing a program can be split, with each programmer tackling their own subroutines.

Exam tip

Make sure you are able to explain the difference between a function and a procedure, and that you can identify which subroutines are which in the skeleton code. This is a common topic in section C.

Subroutine A named block of code designed to carry out one specific task.
Call The process of running a subroutine by stating its identifier, followed by any required arguments in parentheses.
Pass The transfer of a value, or the value of a variable, to a subroutine.

Making links

Using meaningful, self-documenting identifiers for subroutines is important to make your code more readable. It is essential to do this as part of your NEA.

Subroutines are broken down into two types: functions and procedures.

A **function** is a subroutine that returns a value once it has finished executing. A typical use of a function is to carry out a calculation and return the result.

A **procedure** is a subroutine that does not return a value. A typical use of a procedure is to display some data and/or prompts to the user.

Subroutines can be grouped together in a file to form a **module** or **library**. These subroutines can then be re-used in other programs.

It is common to import libraries that have already been written to help solve problems; for example, in Java `import java.util.*` or, in Python, `import random`.

Function A subroutine that returns a value.
Procedure A subroutine that does not return a value.
Module A file that contains one or more subroutines that can be imported and used in another program.
Library A collection of modules that provide related functionality.

Now test yourself TESTED

40 State **three** advantages for using subroutines.
41 Explain why it is important for subroutines to have meaningful identifiers.
42 What is the name for a subroutine that doesn't return a value?
43 What is the name for a subroutine that does return a value?

Answers available online

Parameters

REVISED

The **parameters** of a subroutine are the variables that must be passed to a subroutine when it is called. This is declared when the subroutine is written.

For example, in the code `Procedure DisplayTemperature(int Temp, bool Celsius)`, the subroutine called DisplayTemperature needs to be passed an integer value and a Boolean value.

When a subroutine is called, the values must be passed as **arguments**; for example, `DisplayTemperature(20,True)`.

Parameters The variables that a subroutine needs in order for the subroutine to run.
Arguments The actual values that are passed to the subroutine at runtime.

Returning a value

REVISED

Functions must always return a value at the end of their execution. This is carried out with a **return** statement.

For example:

```
Function Double(int StartVal)
      return 2*StartVal
End function
```

The value that is returned is passed back to the part of the program that called the function.

It is possible for a subroutine to have several different return statements – for example, within a selection structure – but the subroutine will stop once a value has been returned.

Return To pass a value or the contents of a variable back to the place in the program where the function was called.

Now test yourself

44 Describe what is meant by the term *parameter*.
45 Explain the purpose of a return statement.

Answers available online

TESTED

Local variables

REVISED

When a subroutine is called, any variables passed as parameters and any variables declared within that subroutine are referred to as **local variables**. These variables can only be accessed within that subroutine and will be destroyed once the subroutine has finished executing. This is referred to as the **scope** of the variable.

Local variables Variables that are declared within a subroutine and can only be accessed by the subroutine during the execution of that subroutine.
Scope The visibility of variables (either local or global).

The advantages of local variables are:

- uses less memory as local variables are destroyed once the subroutine has finished executing
- less likely to accidentally change the value of a variable somewhere else in the program
- easier to debug
- variable identifiers can be re-used in separate subroutines
- subroutines can be more easily re-used (subroutines are modular).

Modular Independent of other subroutines.

Knowing that local variables are destroyed once that section of code has finished executing, it is important to note that variables declared within a selection statement or iteration structure will also be destroyed at the end of that section of code. It is therefore very important to choose carefully where in the program a variable will be declared.

Global variables

REVISED

Global variables are variables that are declared in the main program and can be read or altered in any subroutine.

Global variables Variables that can be accessed from any subroutine.

Accessing global variables from within subroutines reduces the need for passing parameters and using return statements but should generally be avoided where possible.

Global variables are, however, useful for named constants.

Making links

Try to limit your use of global variables when working on your NEA programming project. Using a modular structure for your subroutines will increase the range of marks you are able to access.

Now test yourself TESTED

46 Explain what is meant by the *scope* of a variable.

47 Explain **three** advantages of using local variables.

48 Explain **three** disadvantages of using global variables.

49 Describe a situation where it would be appropriate to use a global variable

Answers available online

Stack frames

REVISED

When a subroutine is called a stack frame is created. This stack frame contains:

- the return address – where to return to in the program once the subroutine has finished executing
- parameters – the variables to which data was passed when the subroutine was called
- local variables – any variables declared within that subroutine.

Stack frame The collection of data associated with a subroutine call.
Call stack A data structure that stores the stack frames for each active subroutine while the program is running.

Newly called subroutines are added to the top of the call stack and the top stack frame is removed once that subroutine has been completed.

Exam tip

Make sure you can recall the three things stored in a stack frame as this is a common question in Section A.

Making links

Stacks are a complex data structure with a variety of uses in programming. Stacks and other complex data structures are explored further in Chapter 2.

Now test yourself TESTED

50 What **three** things are stored in a stack frame?

51 When is a new stack frame created?

52 Where are stack frames stored?

Answers available online

Note

If a program becomes stuck in a loop that continually calls one or more subroutines then eventually the call stack will become too full, triggering a stack overflow error.

Recursive techniques

REVISED

Some algorithms are best solved by solving smaller and smaller instances of the same problem. To achieve this, a function must call itself repeatedly – this is known as recursion. One example of **recursion** is used in calculating a factorial.

Recursion The process of a function repeatedly calling itself.

A factorial is calculated by multiplying a number by all of the integers less than itself. The '!' is used as the mathematical symbol for 'factorial'. For example:

$5! = 5 \times 4 \times 3 \times 2 \times 1$

This can be simplified as $5 \times 4!$

$4! = 4 \times 3!$

$3! = 3 \times 2!$

$2! = 2 \times 1!$

$1! = 1$

A recursive subroutine will continue to call itself (known as the **general case**) until it reaches a decision that returns a value without calling itself (known as the **base case** or terminating case).

General case A case in which a recursive function is called and must call itself.
Base case The case in which a recursive function terminates and does not call itself.

The pseudo-code algorithm for a factorial calculator might read:

```
Function Factorial (int n)
      IF n <= 1 THEN
            RETURN 1
      ELSE
            RETURN n * Factorial (n-1)
      END IF
End Function
```

If n = 1 or less then the function cannot attempt to break the task down any further, and should stop calling itself recursively, returning the result of 1 (1! = 1). This is the base case.

In all other cases the function will call itself. This is the general case.

It is possible to have more than one general case and more than one base case.

Exam tips

Following a recursive algorithm can be very tricky and it is important to get lots of practice using trace tables to step through recursive algorithms.

Section B questions can require the use of recursion to achieve full marks, though a non-recursive solution will always be possible.

Now test yourself

TESTED

53 What is meant by a recursive function?

54 Explain what is meant by the base case in a recursive algorithm.

55 Explain what is meant by the general case of an algorithm.

56 Using the pseudo-code for a factorial calculator, above, state the maximum number of stack frames on the stack when calculating `Factorial(4)`.

Answers available online

Making links

While Sections B and D may assess your ability to write program code using recursive techniques, Sections A and C may require you to demonstrate your ability to hand trace a recursive algorithm. Trace tables are covered in more detail in Chapter 4.

Revision activity

- Use the pseudo-code provided to program and test a working factorial calculator.
- Use a debugger to step through the factorial calculator and inspect the stack frames.
- Create a program that uses recursion to display the Fibonacci sequence (in which the previous two numbers are added to get the next one; for example, 1,1,2,3,5,8,13…).

Programming paradigms

Programming paradigms

REVISED

A programming **paradigm** is a way to classify a programming style or approach, based on its features and its approach. For Component 1 you are expected to have studied:

- procedural-oriented programming
- object-oriented programming.

Each paradigm is discussed in more detail below, and it is expected that you will have had practice programming using both paradigms throughout your preparation for the exam.

Paradigm A particular style or approach to designing a solution to a problem.
Procedural-oriented programming An approach to solving a problem using subroutines to tackle smaller sub-problems.
Object-oriented programming An approach to solving a problem using a model based on real-world objects that are designed to interact in a realistic way.

Exam tip

Study the skeleton program carefully to help predict what kinds of questions you might be asked. If the skeleton program uses object-oriented techniques then it is more likely that any questions on this topic will appear in Section C and will refer to the skeleton code and pre-release scenario. If the skeleton code doesn't use object-oriented techniques then it is more likely that questions on procedural-oriented programming will appear in Section C, and Section A will include some more generic questions on object-oriented programming.

Now test yourself TESTED

57 What is the meaning of the term *programming paradigm*?
58 What are the two programming paradigms that you should know?
59 Which paradigm uses a model based on real-world objects?

Answers available online

Procedural-oriented programming

REVISED

Procedural-oriented programming is designed to allow programmers to use a structured, top-down approach to solve a given problem.

The program designer uses decomposition to break the problem down into increasingly small sub-problems, each of which can then be solved using a subroutine (either a function or a procedure).

The main program will then be constructed by calling subroutines which will, in turn, call other subroutines in order to solve the original problem.

Data can be passed to subroutines and values can be returned from them, allowing the different parts of the program to interact with each other.

This type of computer program is generally simpler to understand and the subroutines can be re-used at different points in the program without needing to copy it. This type of program is very modular and can easily be updated by changing one subroutine.

Making links

For more on subroutines see the section *Purpose of subroutines* above on page 18.

Top-down approach A method of planning solutions that starts with the big picture and breaks it down into smaller sub-problems.
Decomposition A method of solving a larger problem by breaking it up into smaller and smaller problems until each problem can't be broken down any further.
Hierarchy charts A diagram that shows which subroutines call which other subroutines. More complex versions will also show what data is passed and returned.

Hierarchy charts

Procedural-oriented programs can be represented using hierarchy charts. Simple hierarchy charts show the relationship between subroutines, such as the chart shown in Figure 1.1, with lines used to connect subroutines wherever one subroutine calls another.

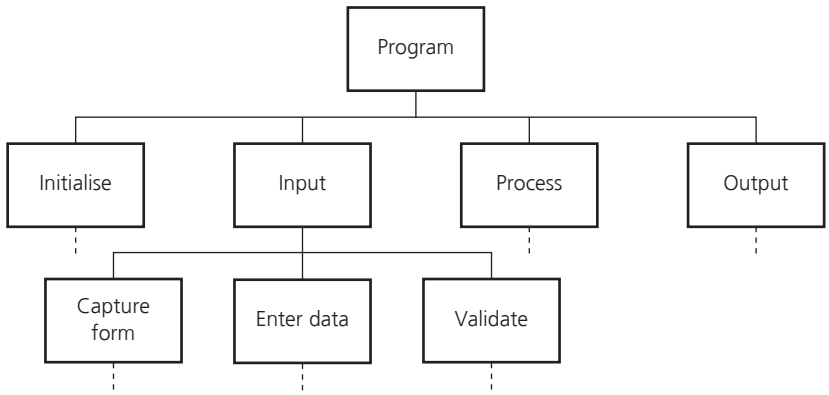


Figure 1.1 A simple hierarchy chart

Exam tip

Questions involving hierarchy charts almost exclusively appear in Section C, referring to the skeleton program, and usually involve a 'fill the gaps' style of question. Make your own hierarchy charts when studying the skeleton code to help you understand how the subroutines fit together and see past papers for example of this style of question.

Now test yourself TESTED

60 How is decomposition used when designing solutions using procedural-oriented programming?

61 In a hierarchy chart, what is the meaning of a line connecting two subroutines?

Answers available online

Revision activity

- Open a complex program you have been working on in your lessons and create a hierarchy chart to show the relationships between each subroutine.
- Consider the steps involved in a two-player game such as Rock-Paper-Scissors or Noughts & Crosses. Create a hierarchy chart to show how the game could be broken down into subroutines and how those subroutines would be related.

Object-oriented programming REVISED

Classes

Object-oriented programming (often referred to as OOP) uses a different approach to programming.

The programmer thinks about real-world objects and creates a **class** to describe the **attributes** and **methods** for that type of object. For instance, in Figure 1.2, the class called Customer has:

- the attributes: Name, Address, and Date of birth
- the methods: Edit customer and Delete customer.

Class The definition of the attributes and methods of a group of similar objects.
Attributes The properties that an object of that type has, implemented using variables.
Methods Processes or actions that an object of that type can do, implemented using subroutines.

Customer		Account
Name Address Date of birth	← Attributes →	Account number Balance
Edit customer Delete customer	← Methods →	Check balance Add interest

Figure 1.2 Classes containing attributes and methods

Attributes are implemented using variables to hold the relevant data.

Methods are implemented using subroutines to carry out each action.

Objects

When the program is run, objects are instantiated (created) based on the class definitions. Once instantiated, the object is stored in a variable.

Object A specific instance of a class.
Instantiation The process of creating an object based on its class definition.
Constructor A special method (with the same name as the class) that is called when the object is instantiated.

Note

The object is stored in memory and the variable stores the memory address of that object so that it can be accessed.

When an object is created, a special method called the constructor is called, and this will initialise the object, typically setting the attributes and sometimes calling other methods.

Each instance of an object will have its own values for each attribute, but all instances of that object will be able to carry out the same methods. For instance, in Figure 1.3:

- the object Account1 is an instance of the class Account, with Account number 12345 and a balance of £7.63
- Account2 is a different instance of the class Account, with a different account number and balance.

All objects of this type have the same attributes, however the value of the attributes can vary from object to object.

Account1	Account2	Account3
Account number: 12345	Account number: 73526	Account number: 92649
Balance +7.63	Balance -3,107.12	Balance +11,836.01
Check balance	Check balance	Check balance
Add interest	Add interest	Add interest

Figure 1.3 Three unique instances of an Account object

Now test yourself TESTED

62 What **two** features are defined in a class?
63 Describe the relationship between an object and its class.
64 What method is called when an object is instantiated?
65 When two objects of the same class are instantiated, describe the features of each object which will definitely be the same and which might be different.

Answers available online

Encapsulation

Deciding how to structure and organise classes can be difficult, but the main rule is to group together objects with common characteristics (attributes) and behaviours (methods). Keeping these features together in one class is called encapsulation.

Encapsulation The concept of grouping similar attributes, data, and methods together in one object.
Information hiding Controlling access to data stored within an object.

Encapsulation is useful because it means that program code is more modular. This means that the code is easier to debug, can be re-used more easily and teams of programmers can work on individual classes without needing to know how other classes are programmed – they only need to know what methods can be accessed.

Encapsulation is also helpful as it allows for information hiding, in which the data stored in the attributes can be kept within that object and access to that data can be controlled.

Access specifiers

Each attribute (variable) and method (subroutine) within a class has an **access specifier** (also commonly known as an **access modifier**) which controls whether other objects are able to directly interact with it.

Access specifier	Symbol	Attributes	Methods
public	+	can be viewed or updated by any object of any class	can be called by any object of any class
private	-	cannot be viewed or updated by any other object, regardless of class	cannot be called by any other object, regardless of class
protected	#	can only be viewed or updated by this object or another object of that class, or a subclass	can only be called by this object or another object of that class, or a subclass

The convention is to declare attributes as private so that other objects cannot directly interact or affect the values that are stored. This make it less likely that a class written by another programmer could adversely affect the overall program.

To allow access to those data, a class should include **getters** and **setters** – public methods that allow other objects to ask an object to return the value of a specific attribute, or that ask an object to update a value.

One example might be a game character that has a score and a number of lives which are both set to private, and uses public methods to allow other objects to interact with those values:

```
Character = Class

    Private:

          Score: Int

          Lives: Int

    Public:

          Function GetScore()

          Function GetLives()

          Procedure AddPoints()

          Procedure LoseALife()

          Procedure GainExtraLife()

End Class
```

Now test yourself TESTED

66 Identify **three** things that are grouped together using encapsulation.

67 Suggest **two** advantages of using encapsulation.

68 State the name, symbol and meaning for the three main access specifiers.

69 Which access specifier should be used, by default, for attributes?

70 Explain how it is possible for private attributes or methods to be accessed by other objects.

Answers available online

Inheritance

Inheritance describes an 'is a' relationship between classes. For example, a horse is a type of animal.

To implement this, **subclasses** (or **child classes**) inherit attributes and methods from a **base class** (or **parent class**).

Access specifier (or **access modifier**) A keyword that sets the accessibility of an attribute or method.
Public An access specifier that allows that attribute or method to be accessed by any other object.
Private An access specifier that protects that attribute or method from being accessed by any other object.
Protected An access specifier that protects that attribute or method from being accessed by other objects unless they are instances of that class or a subclass.
Getter A function used to return the value of an attribute to another object.
Setter A procedure used to allow another object to set or update the value of an attribute.

Revision activity

- Using pseudo-code, describe the class attributes and methods required for a virtual pet.
- Design and build an object-oriented program that uses a `Calculator` class with methods such as `Press0`, `Press1`, `PressPlus` and `PressEquals`.

Inheritance The idea that one class can use the attributes and methods defined in another class.
Subclass (or **child class**) A class that inherits the attributes and methods from another class.
Base class (or **parent class**) A class whose attributes and methods are inherited by another class.

MY **REVISION** NOTES

AQA A-level

COMPUTER SCIENCE

Target exam success with *My Revision Notes*. Our updated approach to revision will help you learn, practise and apply your skills and understanding. Coverage of key content is combined with practical study tips and effective revision strategies to create a guide you can rely on to build both knowledge and confidence.

My Revision Notes: AQA A-level Computer Science will help you:

Develop your subject knowledge by **making links** between topics for better exam answers

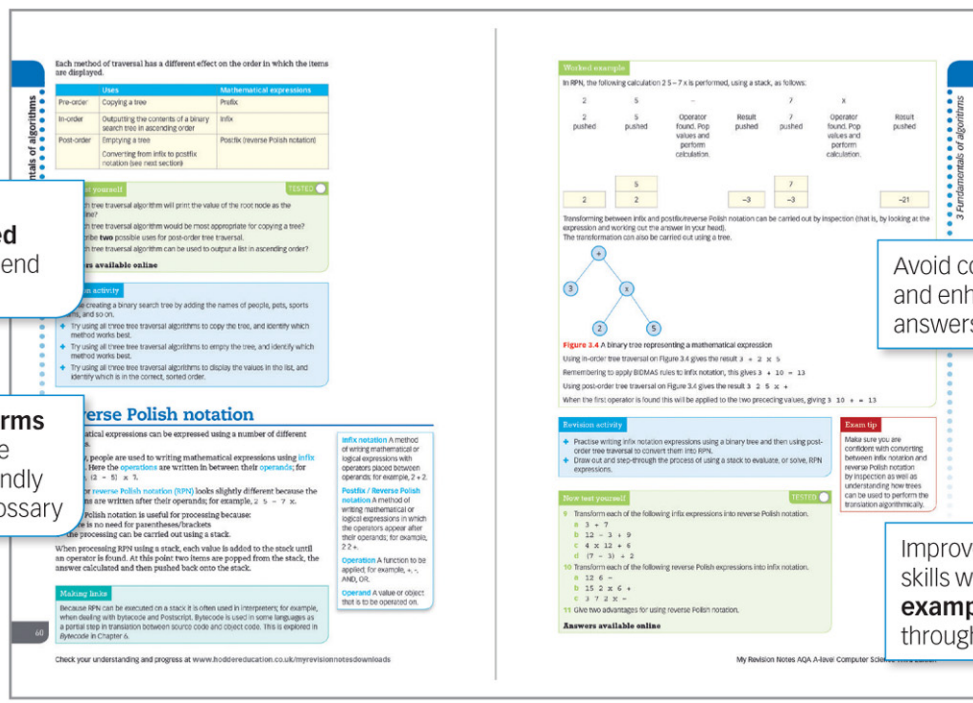
Plan and manage your revision with our **topic-by-topic planner** and exam breakdown introduction

Build quick recall with **bullet-pointed summaries** at the end of each chapter

Avoid common mistakes and enhance your exam answers with **exam tips**

Understand **key terms** you will need for the exam with user-friendly definitions and a glossary

Improve subject-specific skills with '**worked example**' boxes throughout



Practise and apply your skills and knowledge with **exam practice questions** and frequent **now test yourself** questions, with answer guidance online

HODDER EDUCATION

t: 01235 827827

e: education@bookpoint.co.uk

w: hoddereducation.co.uk

ISBN 978-1-3983-2548-7