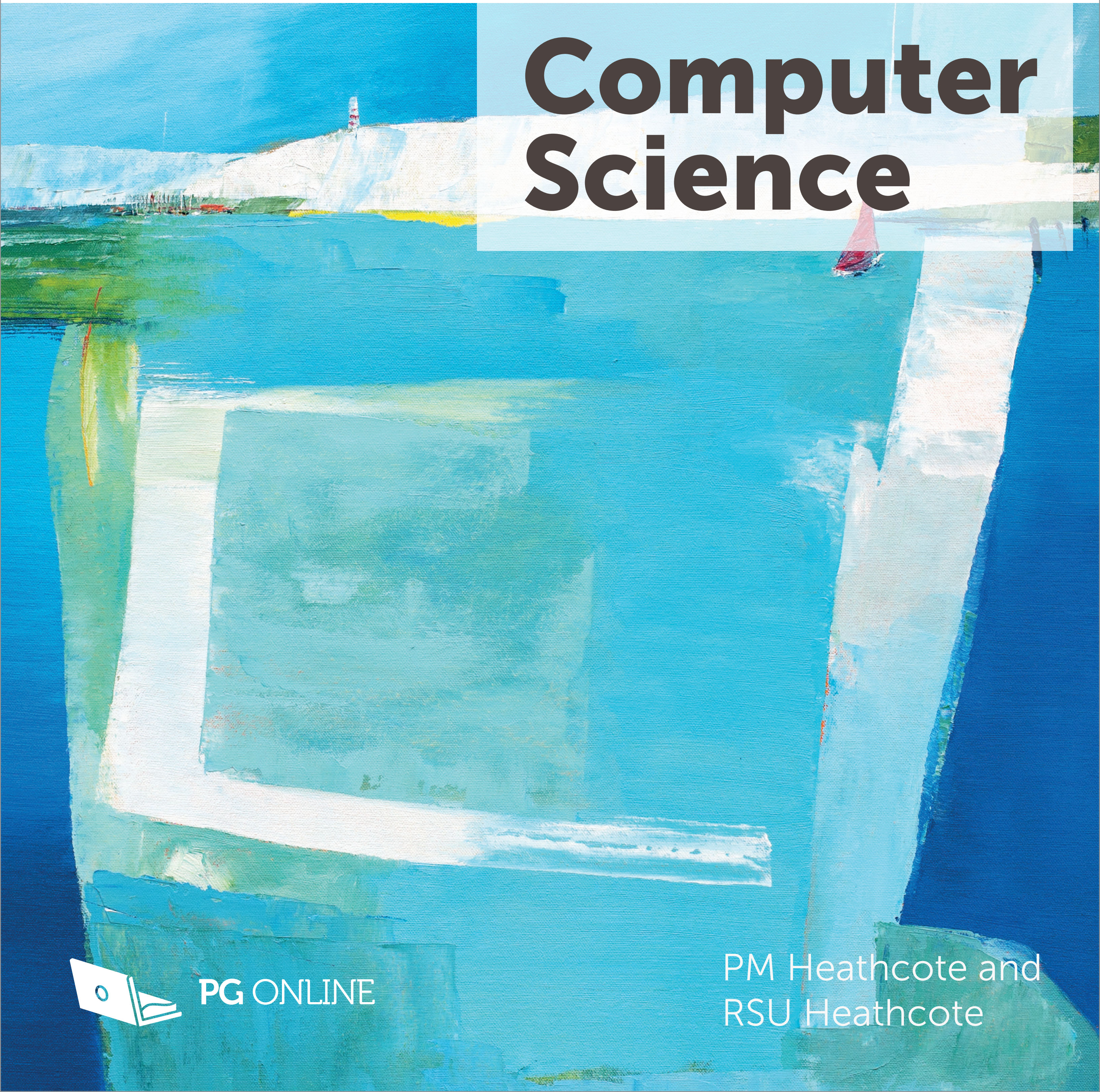
AQA Approved

AQA AS and A Level
Computer Science



PM Heathcote and
RSU Heathcote

PG ONLINE

Contents

Section 10

Section 11

Section 12

Appendices and Index

Chapter 1 – Programming basics

Objectives

- Define what is meant by an algorithm and pseudocode
- Learn how and when different data types are used
- Learn the basic arithmetic operations available in a typical programming language
- Become familiar with basic string handling operations
- Distinguish between variables and constants

What is an algorithm?

An algorithm is a set of rules or a sequence of steps specifying how to solve a problem. A recipe for chocolate cake, a knitting pattern for a sweater or a set of directions to get from A to B, are all algorithms of a kind. Each of them has **input**, **processing** and **output**. We will be looking in more detail at properties of algorithms in Section 2 of this book.

Q1: What are the inputs and outputs in a recipe, a knitting pattern and a set of directions?

Ingredients	Method
100g plain flour	Put flour and salt into a large mixing bowl and make a well in the centre.
2 eggs	Crack the eggs into the middle.
300ml milk	Pour in about 50ml milk and the oil.
1tbsp oil	Start whisking from the centre, gradually drawing the flour into the eggs, milk and oil, etc.
Pinch salt	

In the context of programming, the series of steps has to be written in such a way that it can be translated into program code which is then translated into machine code and executed by the computer.

Using pseudocode

Whatever programming language you are using in your practical work, as your programs get more complicated you will need some way of working out what the steps are before you sit down at the computer to type in the program code. A useful tool for developing algorithms is **pseudocode**, which is a sort of halfway house between English and program statements. There are no concrete rules or syntax for how pseudocode has to be written, and there are different ways of writing most statements. We will use a standard way of writing pseudocode that translates easily into a programming language such as Python, Pascal or whatever procedural language you are learning.

This book does not teach you how to program in any particular programming language – you will learn how to write programs in your practical sessions – but it will help you to understand and develop your own algorithms to solve problems.

Example 3 is a classic logic problem, which has many different variations on the same theme.

Q3: A man has to get a fox, a chicken, and a sack of corn across a river.

He has a rowing boat, which can carry only him and one other thing.

If the fox and the chicken are left together, the fox will eat the chicken.

If the chicken and the corn are left together, the chicken will eat the corn.

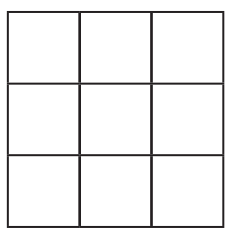
How does the man do it?

Strategies for problem solving

There are some general strategies for designing algorithms which are useful for solving many problems in computer science. First of all it is useful to note that there are two types of algorithmic puzzle. Every puzzle has an **input**, which defines an **instance** of the puzzle. The instance can be either **specific** (e.g. fill a magic square with 3 rows and 3 columns), or **general** (n rows and n columns). Even when given a general instance of a problem, it is often helpful to solve a specific instance of it, which may give an insight into solving a more general case.

Exhaustive search

For example, suppose you are asked to fill a 'magic square' with 3 rows and 3 columns with distinct integers 1-9 so that the sum of the numbers in each row, column and corner-to-corner diagonal is the same.



This is a **specific** instance of a more **general** problem in which there are n rows and n columns. Some problems can be solved by **exhaustive search** – in this example, by trying every possible combination of numbers. We can put any one of 9 integers in the first square, and any of the remaining 8 in the second square, giving 9x8 = 72 possibilities for just the first two squares. There are 9x8x7x6x5x4x3x2 = 362,880 ways of filling the square. If you are a mathematician you will know that this is denoted by 9!, spoken as "nine factorial".

You might think a computer could do this in a fraction of a second. However, looking at the more general problem, where you have n x n squares, you will find that even for a 5 x 5 square, there are so many different combinations (25! or 25 factorial) that it would take a computer performing 10 trillion operations a second, about 49,000 years to find the answer!

So, to solve this problem we need to come up with a better algorithm. It turns out to be not very difficult to work out that for a 3 x 3 square, each row, column and diagonal must add up to 15 and the middle number must be 5, which considerably reduces the size of the problem. (The details of the algorithm are not discussed here.)

Q4: Fill the magic square to solve the problem.

Exercises

1. Figure 2 shows the state transition diagram of a finite state machine (FSM) used to control a vending machine.

The vending machine dispenses a drink when a customer has inserted exactly 50 pence.

A transaction is cancelled and coins returned to the customer if more than 50 pence is inserted or the reject button (R) is pressed. The vending machine accepts 10, 20 and 50 pence coins. Only one type of drink is available.

The only acceptable inputs for the FSM are 10, 20, 50 and R.

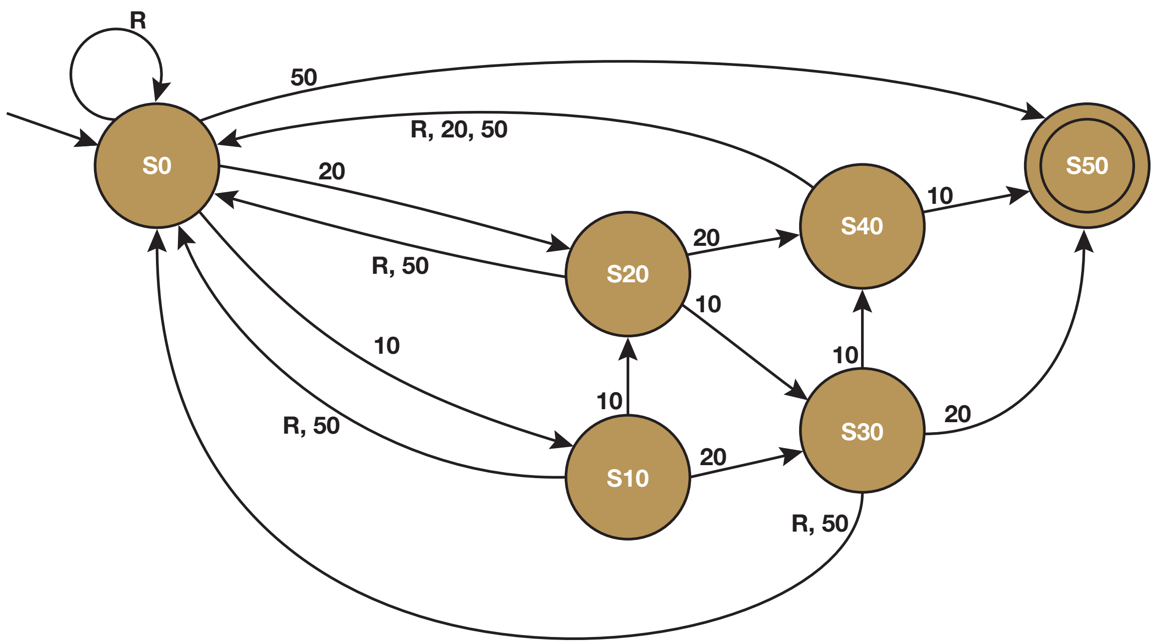


Figure 2

An FSM can be represented as a state transition diagram or as a state transition table. Table 2 is an incomplete state transition table for part of Figure 2.

(a) Complete the missing sections of the four rows of Table 2.

Original state	Input	New state
S0	10	S10
S0		
S0		
S0		

Table 2 [3]

There are different ways that a customer can provide **exactly three** inputs that will result in the vending machine dispensing a drink. Three possible permutations are "20, 10, 20", "10, R, 50" and "10, 50, 50".

(b) List **four** other possible permutations of **exactly three** inputs that will be accepted by the FSM shown in Figure 2. [4]

AQA Comp1 Qu 4 June 2012

2-12

Chapter 17 – Digital representation of sound

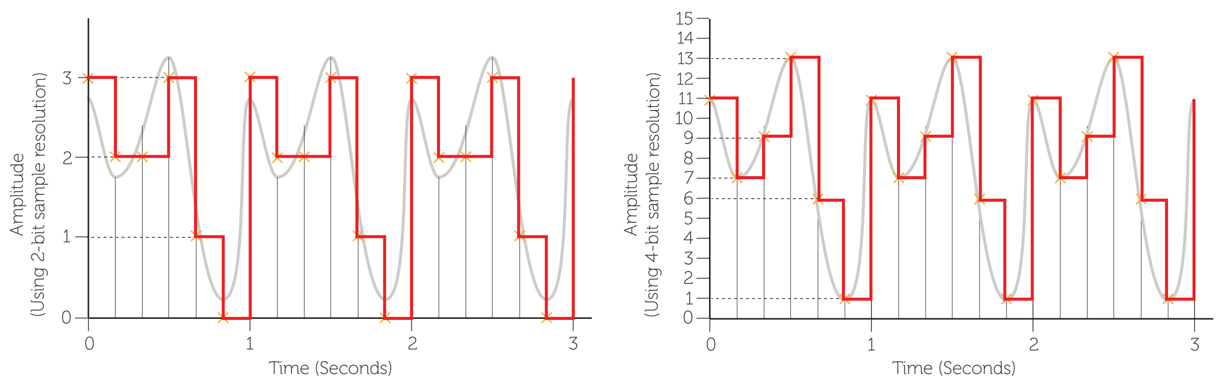
Objectives

- Describe the digital representation of sound in terms of sampling rate and resolution
- Describe the principles of operation of an analogue to digital converter and a digital to analogue converter
- Understand and apply the Nyquist theorem
- Calculate sound sample sizes in bytes
- Describe the purpose of MIDI and the use of event messages
- Describe the advantages of using MIDI files for representing music

Sound sampling and resolution

Sound waves are naturally in a continuous, analogue form. To represent sound in a computer, the (**continuous**) analogue sound waves have to be converted to a (**discrete**) digital format. This can be done by measuring and recording the amplitude of the sound wave at given time intervals (several thousand times per second). The more frequently the samples are taken, the more accurately the sound will be represented. The frequency at which samples are taken is measured in **hertz (Hz)**, a unit of frequency equal to one cycle per second.

In addition, in the same way that an image's quality is improved with a more precise representation of colour enabled by a greater colour depth, the accuracy of a sound recording increases with a greater audio bit depth. Increasing the number of points of amplitude (represented on the y axis below) increases the accuracy at which you can record a sound's amplitude (or wave height) at a given point in time.



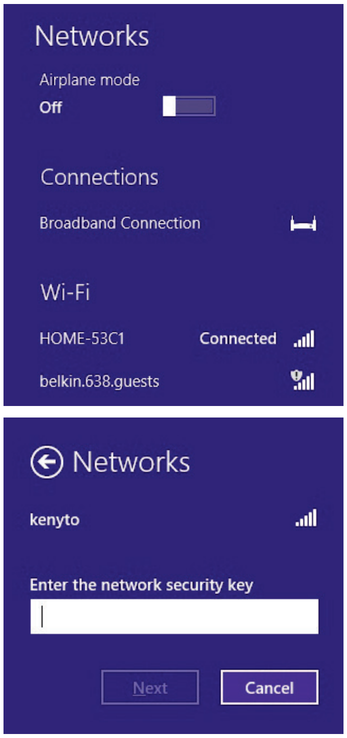
Q1: Which of the graphs above represents a more accurate recording? Why?

Sample rate

The **sampling rate** is the frequency with which you record the amplitude of the sound. The more often you take a sample, the smoother the playback will sound. The disadvantage of this, is that every time you take a sample, at a resolution of say 16 bits, you need to store another 2 bytes of data. A typical CD recording is made at 44,100Hz, or 44,100 times per second. This means that for every second of sound, 2 bytes x 44,100 = 88,200 / 1000 = 88.2KB is required and for every minute, approximately 5.3MB is required. For stereo sound, this is doubled to provide samples for left and right channels.

Securing a wireless network

Wi-Fi Protected Access (WPA) and **Wi-Fi Protected Access II (WPA2)**, which has replaced it, are two security protocols and security certification programs used to secure wireless networks. WPA2 is built into wireless network interface cards, and provides strong encryption of data transmissions, with a new 128-bit key being generated for each packet sent.



Each wireless network has a Service Set Identification (SSID) which is the informal name of the local network – for example, HOME-53C1. The purpose of the SSID is to identify the network, and if, for example, you visit someone else's house with a laptop and wish to connect to their Wi-Fi network in order to use the Internet, when you try to log on to the Internet the computer will ask you to enter the name of the network.

Your computer may be within the range of several networks, so having chosen the correct SSID you will then be asked for the password or security key - an identifier of up to 32 bytes, usually a human-readable string. SSIDs must be locally unique.

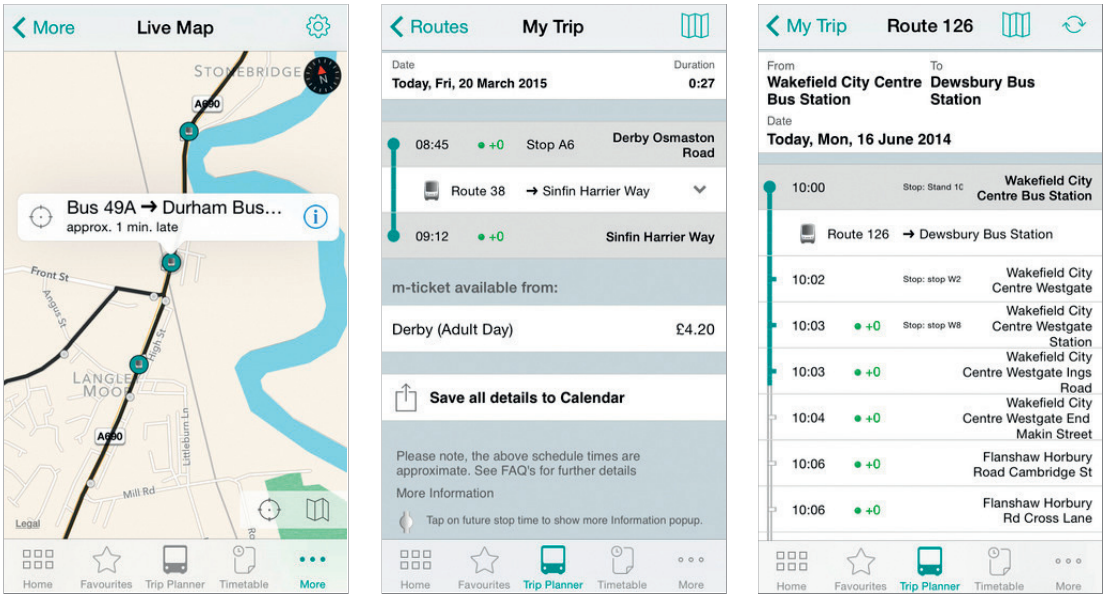
It is possible to disable the broadcast of your SSID to hide your network from others looking to connect to a named local network. However, this will not hide your network completely.

6-34

Whitelists

Some network administrators set up **MAC address whitelists** (the opposite of blacklists) to control who is allowed on their networks. (The MAC address is a unique identifier assigned to a network interface card by the manufacturer: see page 167.)

Q1: Research some of the applications of "location-based services" such as *Presence Orb*. What are some of the benefits and some of the drawbacks to individuals of tracking software?



Arriva's Bus App

Chapter 41 – Graphs

Objectives

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

Definition of a graph

A graph is a set of **vertices** or **nodes** connected by **edges** or **arcs**. The edges may be one-way or two way. In an **undirected graph**, all edges are bidirectional. If the edges in a graph are all one-way, the graph is said to be a **directed graph** or **digraph.**

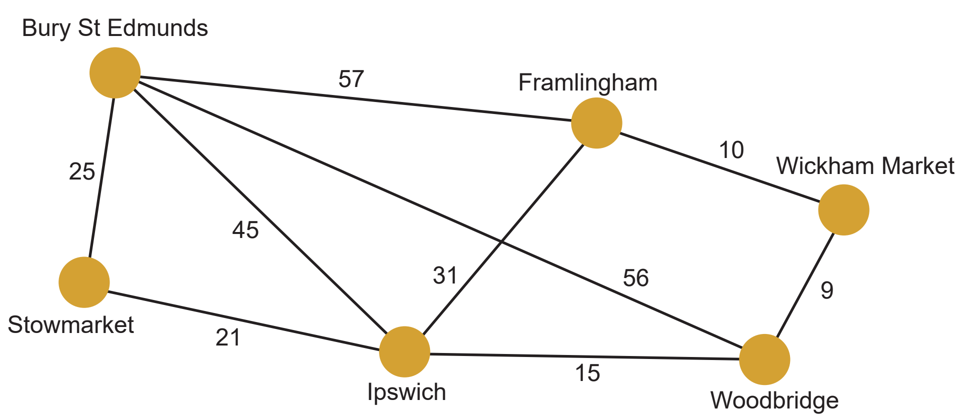


Figure 41.1: An undirected graph with weighted edges

The edges may be **weighted** to show there is a cost to go from one vertex to another as in Figure 41.1. The weights in this example represent distances between towns. A human driver can find their way from one town to another by following a map, but a computer needs to represent the information about distances and connections in a structured, numerical representation.

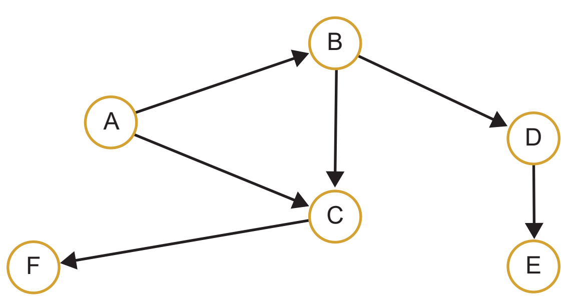
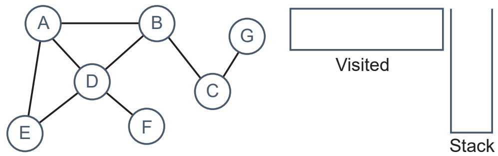
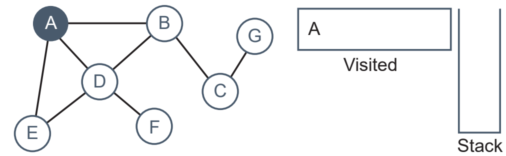


Figure 41.2: A directed, unweighted graph

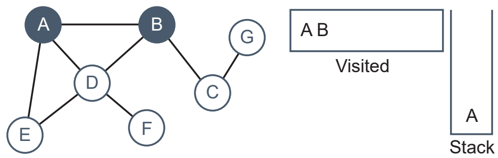
It is easiest to understand how this works by looking at the graphs below. This shows the state of the **stack** (here it just shows the current node when a recursive call is made), and the contents of the **visited** list. Each visited node is coloured dark blue.



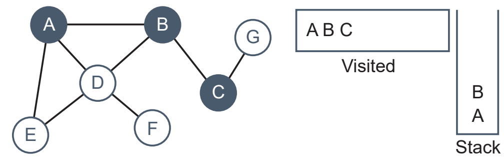
1. Start the routine with an empty stack and an empty list of visited nodes.



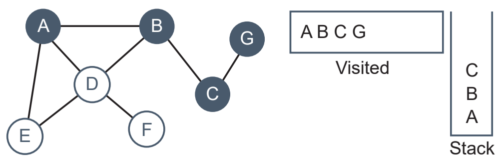
2. Visit A, add it to the visited list. Colour it to show it has been visited.



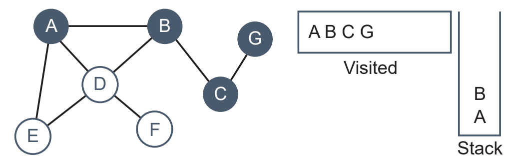
3. Push A onto the stack to keep track of where we have come from and visit A's first neighbour, B. Add it to the visited list. Colour it to show it has been visited.



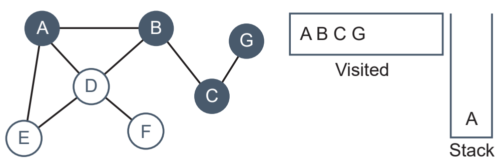
4. Push B onto the stack and from B, visit the next unvisited node, C. Add it to the visited list. Colour it to show it has been visited.



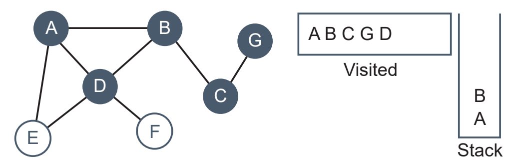
5. Push C onto the stack and from C, visit the next unvisited node, G. Add it to the visited list. Colour it to show it has been visited.



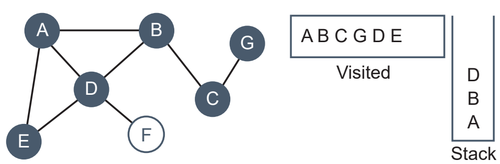
6. At G, there are no unvisited nodes so we backtrack. Pop the previous node C off the stack and return to C



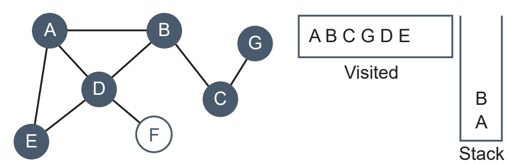
7. At C, all adjacent nodes have been visited, so backtrack again. Pop B off the stack and return to B.



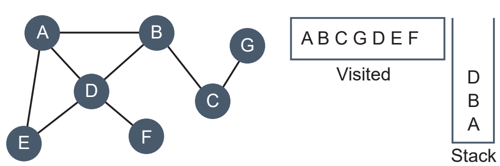
8. Push B back onto the stack to keep track of where we have come from and visit D. Add it to the visited list. Colour it to show it has been visited.



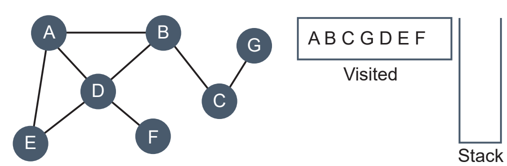
9. Push D onto the stack and visit E. Add it to the visited list. Colour it to show it has been visited.



10. From E, A and D have already been visited so pop D off the stack and return to D.



11. Push D back onto the stack and visit F. Add it to the visited list. Colour it to show it has been visited.



12. At F, there are no unvisited nodes so we pop D, then B, then A, whose neighbours have all been visited. The stack is now empty which means every node has been visited and the algorithm has completed.

Regular language

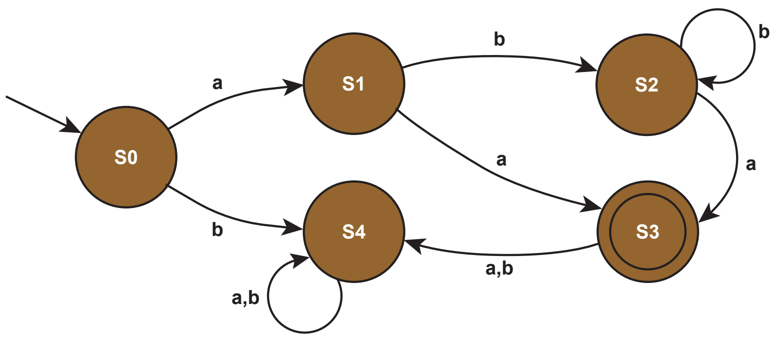
A language is called **regular** if it can be represented by a regular expression. A regular language can also be defined as any language that a **finite state machine** will accept. Any finite language (one containing only a finite number of words) is a regular language, since a regular expression can be created that is the union of every word in the language.

Example 1

A regular language consists of all words beginning and ending in *a*, with zero or more instances of *b* in between, e.g. aa, aba, abba, abbba.

Write a regular expression that describes this language, and draw the corresponding finite state machine (FSM).

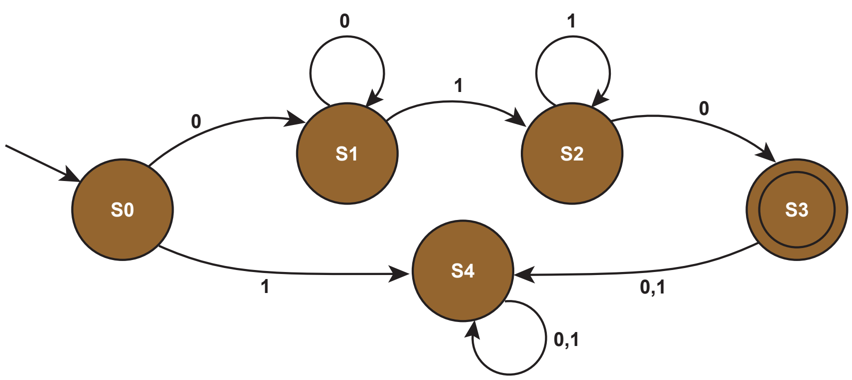
Answer: R = ab*a. Note that the FSM is drawn with an outgoing transition from every state for every possible input symbol.



Example 2

Describe the set of strings found by 0^+1^+0 and draw the FSM.

Answer: It would find all strings with one or more zeros followed by one or more ones followed by one zero. e.g. 010, 0010, 00010, 0010, 00110



Q1: Write a regular expression to find all the occurrences of "color" or "colour" in a document.

Q2: Write a regular expression that matches any non-empty string that starts with zero or more "a"s, followed by one or more "b"s.

Q3: Which of the following strings is matched by the regular expression Sc(o⁺)(b|d)*y?

Scooby Scoby Scddy Scobby Scoobdbdbdy

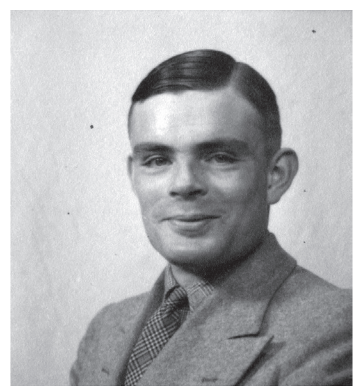
Draw an FSM that recognises the same language.

Chapter 53 – The Turing machine

Objectives

- Know that a Turing machine can be viewed as a computer with a single fixed program, expressed using
    - a finite set of states in a state transition diagram
    - a finite alphabet of symbols
    - an infinite tape with marked off squares
    - a sensing read-write head that can travel along the tape, one square at a time
- Understand the equivalence between a transition function and a state transition diagram
- Be able to:
    - represent transition rules using a transition function
    - represent transition rules using a state transition diagram
    - hand-trace simple Turing machines
- Explain the importance of Turing machines and the Universal Turing machine to the subject of computation

Alan Turing



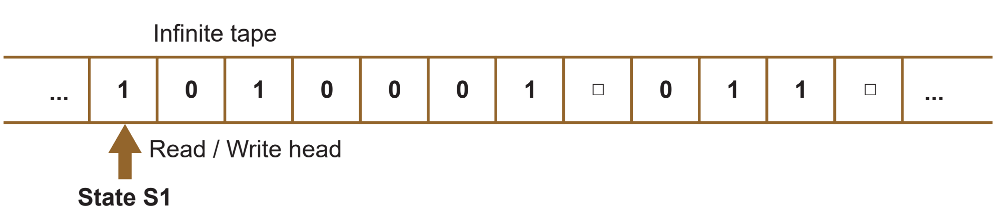
Alan Turing (1912–1954) was a British computer scientist and mathematician, best known for his work at Bletchley Park during the Second World War. While working there, he devised an early computer for breaking German ciphers, work which probably shortened the war by two or more years and saved countless lives.

Turing was interested in the question of **computability**, and the answer to the question "Is every mathematical task computable?" In 1936 he invented a theoretical machine, which became known as the **Turing machine**, to answer this question.

The Turing machine

The Turing machine consists of an infinitely long strip of tape divided into squares. It has a read/write head that can read symbols from the tape and make decisions about what to do based on the contents of the cell and its current state.

Essentially, this is a finite state machine with the addition of an infinite memory on tape. The FSM specifies the task to be performed; it can erase or write a different symbol in the current cell, and it can move the read/write head either left or right.



The Turing machine is an early precursor of the modern computer, with input, output and a program which describes its behaviour. Any alphabet may be defined for the Turing machine; for example a binary alphabet of 0, 1 and □ (representing a blank), as shown in the diagram above.

A computer sending data across a network will use a **subnet mask** and the destination IP address to determine from the network ID whether or not the destination computer is on the same subnetwork. This is done by performing the same AND operation between the computer's own IP address and the subnet mask; if the two network IDs produced are the same then the computers are on the same network so data can be sent directly between them. Otherwise the sending computer must send the data to a router for forwarding to the network that the destination computer is on.

	128	64	32	16	8	4	2	1		128	64	32	16	8	4	2	1		128	64	32	16	8	4	2	1		128	64	32	16	8	4	2	1
	140								.	**24**								.	**112**								.	**57**							
IP Address:	1	0	0	0	1	1	0	0	.	0	0	0	1	1	0	0	0	.	0	1	1	1	0	0	0	0	.	0	0	1	1	1	0	0	1
Subnet mask:	1	1	1	1	1	1	1	1	.	1	1	1	1	1	1	1	1	.	1	1	1	1	1	1	1	1	.	0	0	0	0	0	0	0	0
Network ID:	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**.**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**.**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	.	0	0	0	0	0	0	0	0

Subnetting

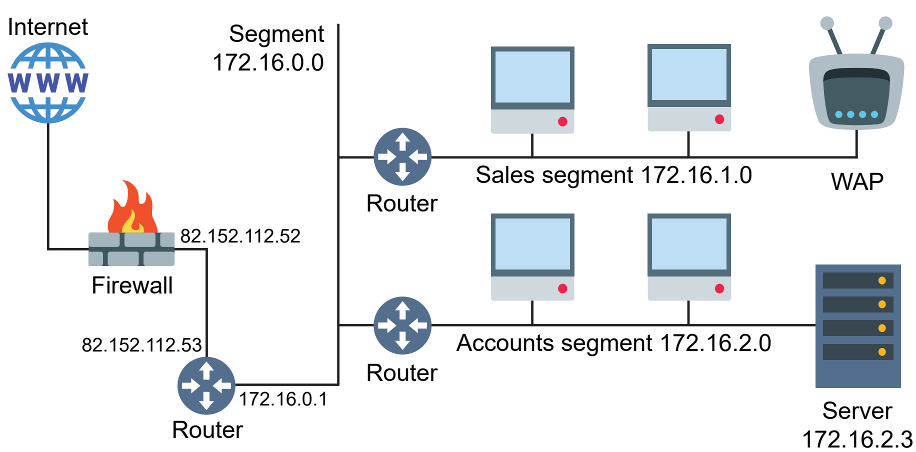
A network administrator of a large organisation using an **IP address** with a 16-bit network ID may wish to create **subnetwork** segments within their own larger IP network in order to ease management and improve efficiency by routing data through one segment only. Using a bus network, this would allow two computers in subnetwork A to communicate at the same time as two computers in subnetwork B avoiding any collisions. **Subnetting** reduces the size of the broadcast domain which can improve security, speed and reliability.

A **subnet ID** is created by using the most significant bits from the host ID section of the IP addresses. In the example below, the eight most significant bits of the 16-bit host ID have been used as a subnet ID leaving 8 bits or 254 (28 = 254-2 to exclude all-zero and all-one) unique host addresses in each of 256 (28) new subnetworks. The term Subnet ID is often used to cover the Network ID and Subnet ID together. For example, if you configure a computer or home router no distinction is made between the two.

10-60

	128	64	32	16	8	4	2	1		128	64	32	16	8	4	2	1		128	64	32	16	8	4	2	1		128	64	32	16	8	4	2	1
	172								.	**16**								.	**1**								.	**5**							
IP Address:	1	0	1	0	1	1	0	0	.	0	0	0	1	0	0	0	0	.	0	0	0	0	0	0	0	1	.	0	0	0	0	0	1	0	1
Subnet mask:	1	1	1	1	1	1	1	1	.	1	1	1	1	1	1	1	1	.	1	1	1	1	1	1	1	1	.	0	0	0	0	0	0	0	0
	1	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**.**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	.	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	.	0	0	0	0	0	0	0	0
	Network ID																		**Subnet ID**									**Host ID**							

A network diagram showing subnetwork segments might look like this:



Q2: Suggest a suitable IP address for the Wireless Access Point in the diagram above.

Chapter 70 – Function application

Objectives

- Understand what is meant by partial function application
- Know that a function takes only one argument which may itself be a function
- Define and use higher-order functions, including map, filter and fold

Higher-order functions

A **higher-order function** is one which either takes a function as an argument or returns a function as a result, or both. Later in this chapter we will be looking at the higher-order functions **map**, **filter** and **fold**, in which the first argument is a function and the second argument is a list on which the function operates, returning a list as a result.

Every function in Haskell takes only one argument. This may seem like a contradiction because we have seen many functions, such as the one below which adds three integers,

```
add3Integers x y z = x + y + z
```

which appear to take several arguments. So how can this be true?

Any function takes only one parameter at a time

Taken at face value and assuming the function takes three integer parameters and returns an integer result, the type declaration for this function would normally be written

```
add3Integers :: integer -> integer -> integer -> integer
```

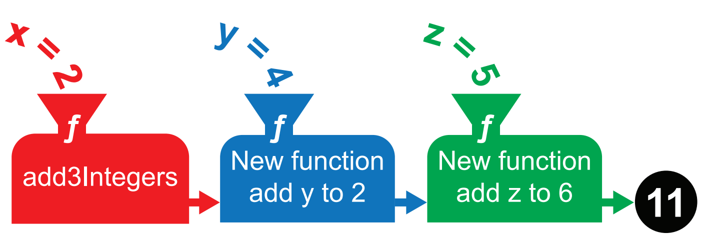
It could also be written

```
add3Integers :: integer -> (integer -> (integer -> integer))
```


How the function is evaluated

What happens when you write `add3Integers 2 4 5` ?

The function `add3Integers` is applied to the arguments. It takes the first argument 2 and produces a new function (shown in blue above) which will add 2 to its arguments, 4 and 5.



This function (shown in blue) produces a new function (shown in green) that takes the argument 5 and adds it to 6, returning the result, 11.

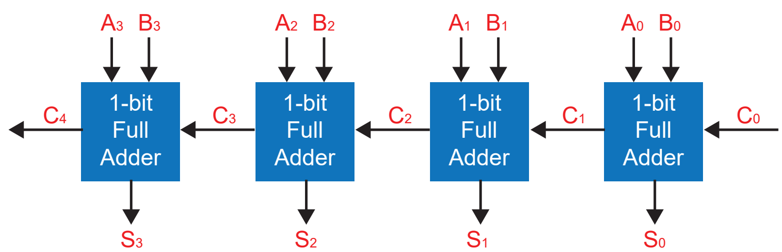
Our function `add3Integers` takes an integer argument (2) and returns a function of type

```
(integer -> (integer -> integer))
```

Concatenating full adders

Multiple full adders can be connected together. Using this construct, n full adders can be connected together in order to input the carry bit into a subsequent adder along with two new inputs to create a concatenated adder capable of adding a binary number of n bits.

	0^{C_3}	1^{C_2}	1^{C_1}		
	0^{A_3}	0^{A_2}	1^{A_1}	1^{A_0}	3
+	1^{B_3}	0^{B_2}	1^{B_1}	1^{B_0}	11
=	1^{S_3}	1^{S_2}	1^{S_1}	0^{S_0}	**14**



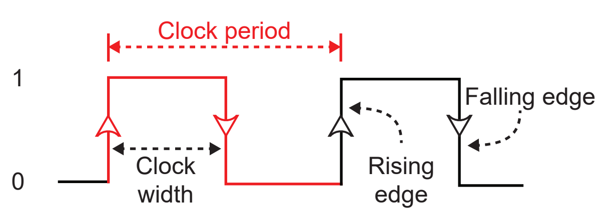
Q1: What would be the output S_4 from a fifth adder connected to the diagram above if the inputs for A_4 and B_4 were 0 and 1? What would be the output C_5?

B

D-type flip-flops

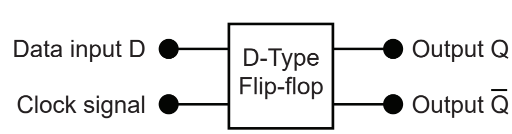
A flip-flop is an elemental **sequential logic circuit** that can store one bit and flip between two states, 0 and 1. It has two inputs, a control input labelled D and a clock signal.

The **clock** or **oscillator** is another type of sequential circuit that changes state at regular time intervals. Clocks are needed to synchronise the change of state of flip-flop circuits.



The **D-type flip-flop** (D stands for Data or Delay) is a positive **edge-triggered flip-flop**, meaning that it can only change the output value from 1 to 0 or vice versa when the clock is at a rising or positive edge, i.e. at the beginning of a clock period.

When the clock is not at a positive edge, the input value is held and does not change. **The flip-flop circuit is important because it can be used as a memory cell to store the state of a bit.**

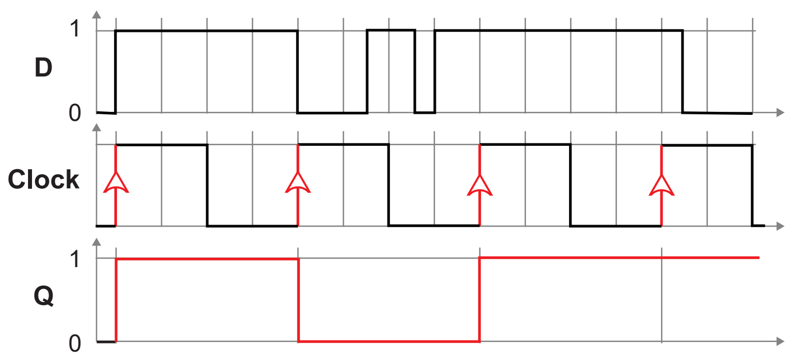


Output Q only takes on a new value if the value at D has changed at the point of a clock pulse. This means that the clock pulse will freeze or 'store' the input value at D until the next clock pulse. If D remains the same on the next clock pulse, the flip-flop will hold the same value.

The use of a D-type flip-flop as a memory unit

A flip-flop comprises several NAND (or AND and OR) gates and is effectively 1-bit memory. To store eight bits, eight flip-flops are required. **Register memories** are constructed by connecting a series of flip-flops in a row and are typically used for the intermediate storage needed during arithmetic operations. Static RAM is also created using D-type flip-flops. Imagine trying to assemble 16GB of memory in this way!

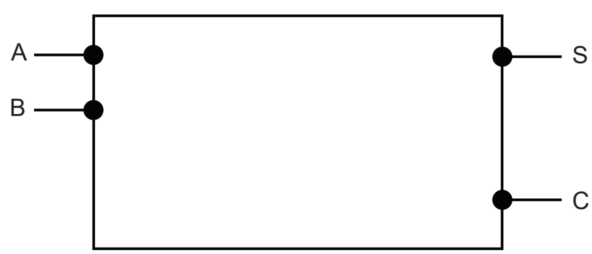
The graph below illustrates how the output Q only changes to match the input D in response to the rising edge on the clock signal. Q therefore delays, or 'stores' the value of D by up to one clock cycle.



Exercises

1. A half-adder is used to find the sum of the addition of two binary digits.

(a) Complete the diagram below to construct a half adder circuit. [1]



(b) Complete the following truth table for a half adder's outputs S and C.

A	B	S	C

[2]

(c) How does a full adder differ from a half adder in terms of its inputs? [2]

Index

Index

Index

Index

AQA AS and A Level
Computer Science

The aim of this textbook is to provide a detailed understanding of each topic of the new AQA A Level Computer Science specification. It is presented in an accessible and interesting way, with many in-text questions to test students' understanding of the material and their ability to apply it.

The book is divided into 12 sections, each containing roughly six chapters. Each chapter covers material that can comfortably be taught in one or two lessons. It will also be a useful reference and revision guide for students throughout the A Level course. Two short appendices contain A Level content that could be taught in the first year of the course as an extension to related AS topics.

Each chapter contains exercises, some new and some from past examination papers, which can be set as homework. Answers to all these are available to teachers only, in a Teachers Supplement which can be ordered from our website
www.pgonline.co.uk

PG ONLINE

About the authors
Pat Heathcote is a well-known and successful author of Computer Science textbooks. She has spent many years as a teacher of A Level Computing courses with significant examining experience. She has also worked as a programmer and systems analyst, and was Managing Director of Payne-Gallway Publishers until 2005.

Rob Heathcote has many years of experience teaching Computer Science and is the author of several popular textbooks on Computing. He is now Managing Director of PG Online, and writes and edits a substantial number of the online teaching materials published by the company.

Cover picture:

'South Coast Sailing'
Oil on canvas, 60x60cm
© Heather Duncan
www.heatherduncan.com

This book has been approved by AQA.

ISBN: 978-1-910523-07-0
9 781910 523070