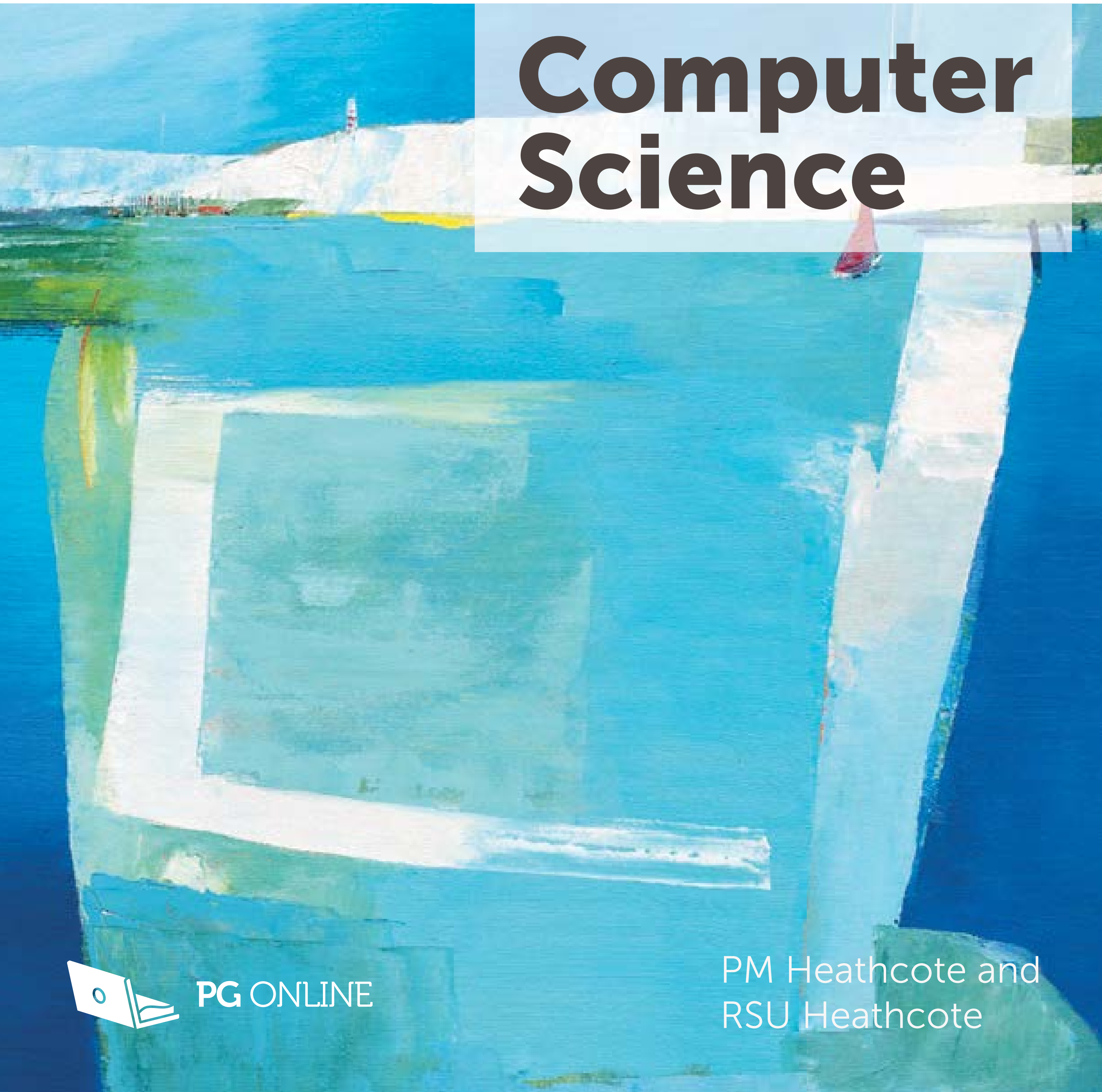
AQA Approved

AQA AS and A Level

Computer Science



PM Heathcote and
RSU Heathcote

PG ONLINE

AQA AS and A Level Computer Science

P.M. Heathcote

R.S.U. Heathcote

Published by
PG Online Limited
The Old Coach House
35 Main Road
Tolpuddle
Dorset
DT2 7EW
United Kingdom
sales@pgonline.co.uk
www.pgonline.co.uk

2016

PG ONLINE

Acknowledgements

We are grateful to the AQA Examination Board for permission to use questions from past papers.

The answers in the Teacher's Supplement are the sole responsibility of the authors and have neither been provided nor approved by the examination board.

We would also like to thank the following for permission to reproduce copyright photographs:

Screenshots of Arriva Bus App © Arriva PLC
Colossus photograph © The National Archives
Google Maps 'StreetView' © Google 2015
Screenshot from Roboform website © Roboform
Alan Turing © By kind permission of the Provost and Fellows, King's College, Cambridge from Archives Centre, King's College, Cambridge. AMT/K/7/12
Trans-continental Internet connections © Telegeography
Internet registries map © Ripe NCC
Other photographic images © Shutterstock

Graphics: Rob Heathcote and Roger Stayte

Cover picture © 'South Coast Sailing' 2014
Oil on canvas 60x60cm
Reproduced with the kind permission of Heather Duncan
www.heatherduncan.com

Design and artwork by OnThree
www.on-three.com

Typeset by Chapter One (London) Ltd, Ian Kingston

First edition 2016, reprinted 2016

A catalogue entry for this book is available from the British Library

ISBN: 978-1-910523-07-0

Printed and bound in Great Britain by Lightning Source Inc., Milton Keynes

Preface

The aim of this textbook is to provide detailed coverage of the topics in the new AQA AS and A Level Computer Science specification.

The book is divided into twelve sections and within each section, each chapter covers material that can comfortably be taught in one or two lessons.

In the first year of this course there will be a strong emphasis on learning to program. You will start by learning the syntax of your chosen programming language – that is, the rules of how to write correct statements that the computer can understand. Then you will code simple programs, building up your skills to the point where you can understand and make additions and amendments to a program consisting of several hundred lines of code.

Sections 1 and 2 of this book can be studied in parallel with your practical programming sessions. It will give you practice in the skills you need to master.

In the second year of this course the focus will turn to algorithms and data structures, covered in Sections 7 and 8. These are followed by sections on regular languages, the Internet and databases.

Object Oriented Programming and functional programming are covered in the final section, which describes basic theoretical concepts in OOP, as well as providing some practical exercises using the functional programming language Haskell. Lists, the fact-based model and 'Big Data' are all described and explained.

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.

The OOP concepts covered may also be helpful in the coursework element of the A Level course.

Each chapter contains exercises and questions, some new and some from past examination papers. Answers to all these are available to teachers only in a Teacher's Supplement which can be ordered from our website **www.pgonline.co.uk**.

Approval message from AQA

This textbook has been approved by AQA for use with our qualification. This means that we have checked that it broadly covers the specification and we are satisfied with the overall quality. Full details of our approval process can be found on our website.

We approve textbooks because we know how important it is for teachers and students to have the right resources to support their teaching and learning. However, the publisher is ultimately responsible for the editorial control and quality of this book.

Please note that when teaching the A Level Computer Science course, you must refer to AQA's specification as your definitive source of information. While this book has been written to match the specification, it cannot provide complete coverage of every aspect of the course.

A wide range of other useful resources can be found on the relevant subject pages of our website: www.aqa.org.uk.

Contents

Section 7

Section 8

Section 9

Section 10

Section 11

Section 12

Appendices and Index

String-handling functions

Programming languages have a number of built-in string-handling methods or functions. Some of the common ones in a typical language are:

`len(string)`	Returns the length of a string
`string.substring(index1,index2)`	Returns a portion of `string` inclusive of the characters at each index position
`string.find(str)`	Determines if `str` occurs in a string. Returns index (the position of the first character in the string) if found, and -1 otherwise. In our pseudocode we will assume that string(1) is the first element of the string, though in Python, for example, the first element is string(0)
`ord("a")`	Returns the integer value of a character (97 in this example)
`chr(97)`	Returns the character represented by an integer ("a" in this example)

Q3: What will be output by the following lines of code?

```
x = "Come into the garden, Maud"
y = len(x)
z = x.find("Maud")
OUTPUT "x= ",x
OUTPUT "y= ",y
OUTPUT "z= ",z
```

1-1

To **concatenate** or join two strings, use the + operator.

e.g. "Johnny" + "Bates" = "JohnnyBates"

String conversion operations

`int("1")`	converts the character "1" to the integer 1
`str(123)`	converts the integer 123 into a string "123"
`float("123.456")`	converts the string "123.456" to the real number 123.456
`str(123.456)`	converts the real number 123.456 to the string "123.456"
`date(year,month,day)`	returns a number that you can calculate with

Converting between strings and dates is usually handled by functions built in to string library modules, e.g. strtodate("01/01/2016").

Example:

```
date1 ← strtodate("18/01/2015")
date2 ← strtodate("30/12/2014")
days ← date1 - date2
OUTPUT date1, date2, days
```

This will output

```
2015-01-18 2014-12-30 19
```

Chapter 12– Finite state machines

Objectives

- Understand what is meant by a finite state machine
- List some of the uses of a finite state machine
- Draw and interpret simple state transition diagrams for finite state machines with no output
- Draw a state transition table for a finite state machine with no output and vice versa

What is a finite state machine?

A finite state machine is a model of computation used to design computer programs and sequential logic circuits. It is not a "machine" in the physical sense of a washing machine, an engine or a power tool, for example, but rather an abstract model of how a machine reacts to an external event. The machine can be in one of a finite number of states and changes from one state to the next state when triggered by some condition or input (say, a signal from a timer).

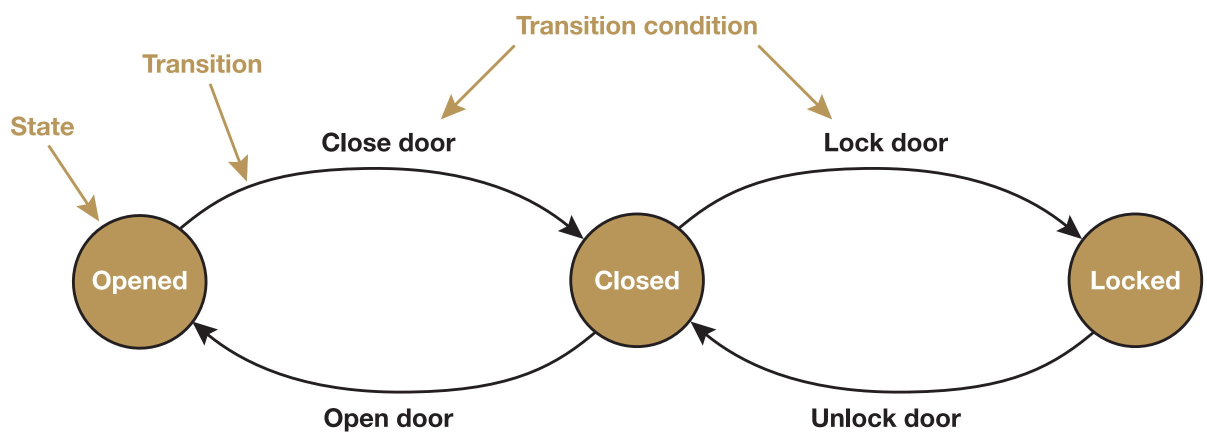
In a finite state machine:

- The machine can only be in one state at a time
- It can change from one state to another in response to an event or condition; this is called a **transition**. Often this is a switch or a binary sensor.
- The Finite State Machine (FSM) is defined by a list of its states and the condition for each transition

There can be outputs linked to the FSM's state, but in this chapter we will be considering only FSMs with no output.

Example 1

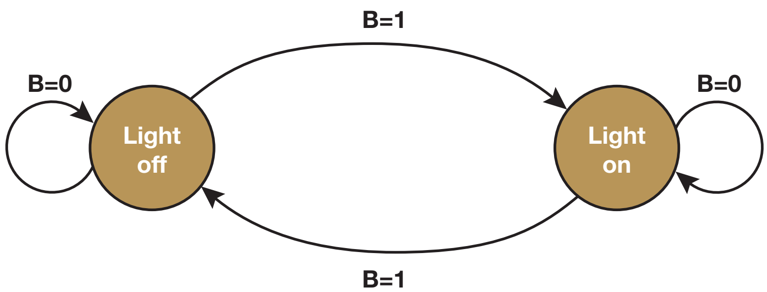
Draw an FSM to model the states and transitions of a door. The door can be open, closed or locked. It can change from the state of being open to closed, from closed to locked, but not, say, from locked to open. (It has to be unlocked first.)



Example 2

Draw an FSM to represent a light switch. When the button is pressed, the light goes on. When the button is pressed again, the light goes off.

There is just one input B to this system: Button pressed (B=1) or Button not pressed (B=0).



Notice that in each state, both the transitions B=0 and B=1 are drawn. If the light is off, the transition B=0 has no effect so the transition results in the same state. Likewise, if the light is on, as long as the button is not pressed, the light will stay on.

Usage of finite state machines

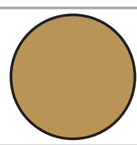
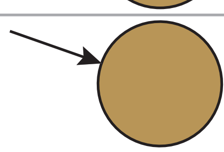
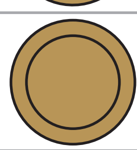
FSMs are widely used in modelling the design of hardware digital systems, compilers and network protocols. They are also used in the definition of languages, and to decide whether a particular word is allowed in the language.

A finite state machine which has no output is also known as a **finite state automaton**. It has a start state and a set of accept states which define whether it accepts or rejects finite strings or symbols. The finite state automaton accepts a string $c_1, c_2 \ldots c_n$ if there is a path for the given input from the start state to an accept state. The language recognised by the finite state automaton consists of all the strings accepted by it.

2-12

If, when you are in a particular state, the next state is uniquely determined by the input, it is a **deterministic final state automaton**. All the examples which follow satisfy this condition.

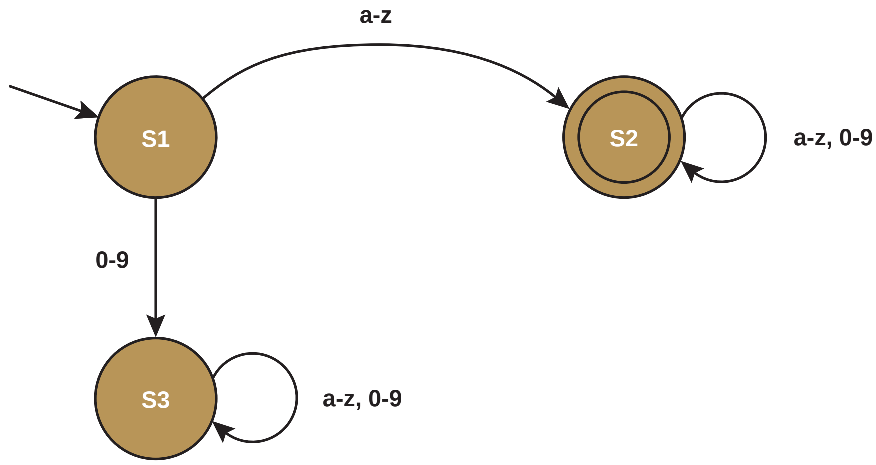
Notation

Symbol	Meaning
	**State**
	**Start state**
	**Accept state**
	**Transition**

Example 3

Use an FSM to represent a valid identifier in a programming language. The rules for a valid identifier for this particular language are:

- The identifier must start with a lowercase letter
- Any combination of letters and lowercase numbers may follow
- There is no limit on the length of the identifier



In this diagram, the **start state** S1 is represented by a circle with an arrow leading into it.

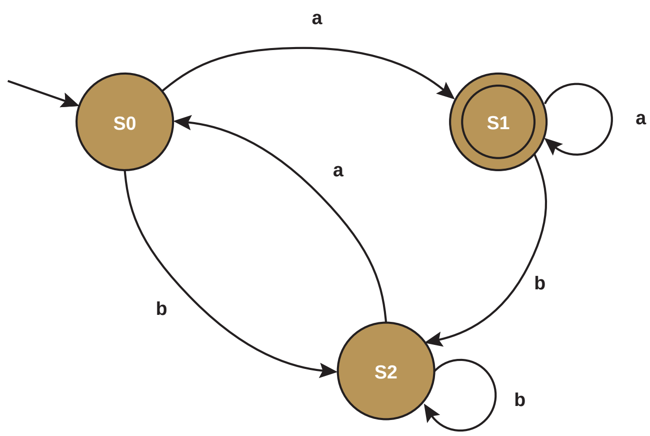
The **accept state** S2 is denoted by a double circle.

S3 is a “dead state” because having arrived here, the string can never reach the accept state.

Each character of the input string is input sequentially to the FSM and if the last character reaches the final state S2 (the **accept** state), the string is valid and is accepted. If it ends up anywhere else the string is invalid.

Note that there can only be one starting state but there may be more than one accept state (or no accept states).

Q1: Which of the following strings is valid and accepted by this finite state machine?
(i) a (ii) bba (iii) abbaa (iv) bbbb



What is encryption?

Encryption is the transformation of data from one form to another to prevent an unauthorised third party from being able to understand it. The original data or message is known as **plaintext**. The encrypted data is known as **ciphertext**. The encryption method or algorithm is known as the **cipher**, and the secret information to lock or unlock the message is known as a **key**.

The Caesar cipher and the Vernam cipher offer polar opposite examples of security. Where the Vernam offers perfect security, the Caesar cipher is very easy to break with little or no computational power. There are many others methods of encryption – some of which may take many computers, many years to break, but these are still breakable and the principles behind them are similar.

The Caesar cipher

Julius Caesar is said to have used this method to keep messages secure. The **Caesar cipher** (also known as a **shift cipher**) is a type of **substitution cipher** and works by shifting the letters of the alphabet along by a given number of characters; this parameter being the key. Below is an example of a shift cipher using a key of 5. (An algorithm for this cipher is given as an example on page 46.)

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E

3-18

Q2: Using the table above, what is the ciphertext for 'JULIUS CAESAR' using a shift of 5?

Q3: What word can be translated from the following ciphertext, which uses a key of `-2:` ZYBECP

You will no doubt be able to see the ease with which you might be able to decrypt a message using this system.

DGYDQFH WR ERUGHU DQG DWWDFN DW GDZQ

Even if you had to attempt a brute force attack on the message above, there are only 25 different possibilities (since a shift of zero means the plaintext and the ciphertext are identical). Otherwise you might begin by guessing the likelihood of certain characters first and go from there. Using cryptanalysis on longer messages, you would quickly find the most common ciphertext letter and could start by assuming this was an E, for example, or perhaps an A. *(Hint.)*

Cryptanalysis and perfect security

Other ciphers that use non-random keys are open to a cryptanalytic attack and can be solved given enough time and resources. Even ciphers that use a computer-generated random key can be broken since mathematically generated random numbers are not actually random; they just appear to be so. A truly random sequence must be collected from a physical and unpredictable phenomenon such as white noise, the timing of a hard disk read/write head or radioactive decay. A truly random key must be used with a Vernam cipher to ensure it is mathematically impossible to break.

The Vernam cipher

The **Vernam cipher**, invented in 1917 by the scientist Gilbert Vernam, is one implementation of a class of ciphers known as **one-time pad ciphers**, all of which offer perfect security if used properly. All others are based on **computational security** and are theoretically discoverable given enough time, ciphertext and computational power. Frequency analysis is a common technique used to break a cipher.

One-time pad

To provide perfect security, the encryption key or **one-time pad** must be equal to or longer in characters than the plaintext, be truly random and be used only once. The sender and recipient must meet in person to securely share the key and destroy it after encryption or decryption. Since the key is random, so will be the distribution of the characters meaning that no amount of cryptanalysis will produce meaningful results.

The bitwise exclusive or XOR

A Boolean XOR operation is carried out between the binary representation of each character of the plaintext and the corresponding character of the one-time pad. The XOR operation is covered in Chapter 23 and you may want to refer to this to verify the output for any combination of 0 and 1. Use the ASCII chart on page 73 for reference.

Plaintext: M	Key: +	XOR: f
1	0	1
0	1	1
0	0	0
1	1	0
1	0	1
0	1	1
1	1	0

Q4: Using the ASCII chart and the XOR operator, what ciphertext character will be produced from the letter E with the key w?

3-18

Using this method, the message "**Meet on the bridge at 0300 hours**" encrypted using a one-time pad of **+tkiGeMxGvnhoQ0xQDIIIVdT4sIJm9qf** will produce the ciphertext:

f◀♫g#X3♂H#Y6!i(=vTg┘Ci"┐L└┘

The encryption process will often produce strange symbols or unprintable ASCII characters as in the above example, but in practice it is not necessary to translate the encrypted code back into character form, as it is transmitted in binary. To decrypt the message, the XOR operation is carried out on the ciphertext using the same one-time pad, which restores it to plaintext.

Exercises

1. Explain the difference between lossy and lossless data compression. [2]

2. Run-length encoding (RLE) is a pattern substitution compression algorithm.
   Data is stored in the format (colour,run) where 0 = White, 1 = Black.

```
(0,1),(1,5),(0,1),
(1,7),
(1,1),(0,2),(1,1),(0,2),(1,1),
(1,7),
(0,1),(1,1),(0,1),(1,1),(0,1),(1,1),(0,1),
(0,1),(1,1),(0,1),(1,1),(0,1),(1,1),(0,1),
(0,1),(1,1),(0,3),(1,1),(0,1)
```

Assembly language instructions

Machine code was the first "language" used to enter programs by early computer programmers. The next advance in programming was to use mnemonics instead of binary codes, and this was called **assembly code** or **assembly language**. Each assembly language instruction translates into one machine code instruction.

Different mnemonic codes are used by different manufacturers, so there are several versions of assembly language.

Typical statements in machine code and assembly language are:

Machine code	Assembly code	Meaning
0100 1100	LDA #12	Load the number 12 into the accumulator
0010 0010	ADD #2	Add the number 2 to the contents of the accumulator
0111 1111	STO 15	Store the result from the accumulator in location 15

The # symbol in this assembly language program signifies that the immediate addressing mode is being used.

5-27

Q5: Write a statement in a high level language which performs an operation equivalent to the three statements in the above machine code program, with the result being stored in a location called TOTAL.

Q6: Write a machine code program, and an equivalent assembly language program, to add the contents of locations 10 and 11 and store the result in location 14.

Exercises

1. A computer with a 16-bit word length uses an instruction set with 6 bits for the opcode, including the addressing mode.

(a) What is an *instruction set*? [1]

(b) How many instructions could be included in the instruction set of this computer? [1]

(c) What is the largest number that can be used as data in the instruction? [1]

(d) What would be the effect of increasing the space allowed for the opcode by 2 bits? [2]

(e) What would be the benefits of increasing the word size of the computer? [2]

2. The high-level language statement

```
X = Y + 6
```

is to be written in assembly language.

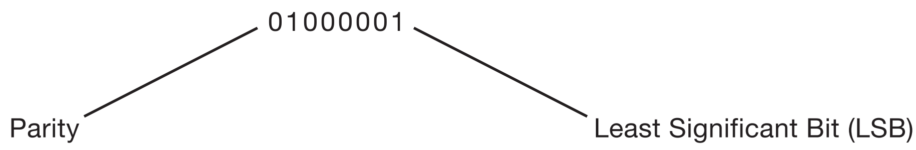
Complete the following assembly language statements, which are to be the equivalent of the above high level language statement. The LOAD and STORE instructions imply the use of the accumulator register.

```
LOAD ...............................
.........................................#6
STORE ............................
```

[3]

Parity

Computers use either even or odd parity. In an even parity machine, the total number of 'on' bits in every byte (including the parity bit) must be an even number. When data is transmitted, the parity bit is set at the transmitting end and parity is checked at the receiving end, and if the wrong number of bits are 'on', an error has occurred. In the diagram below the parity bit is the most significant bit (MSB).



Parity bit in even parity system

Q2: The ASCII codes for P and Q are 1010000 and 1010001 respectively. In an even parity transmission system, what will be the value of the parity bit for the characters P and Q?

Synchronous transmission

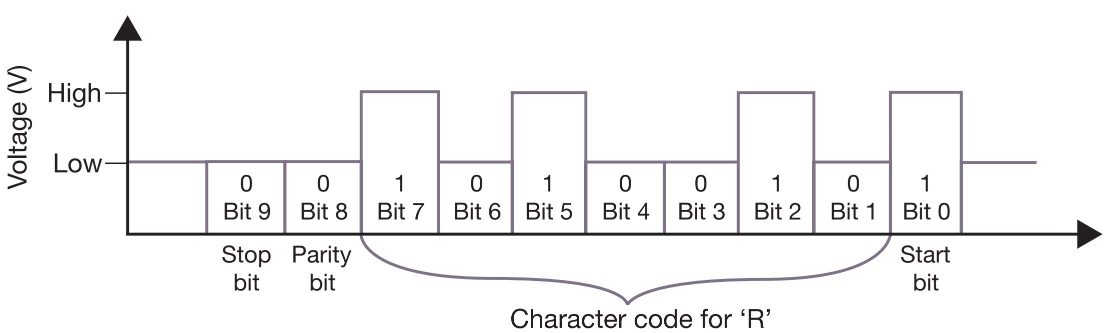
Using **synchronous transmission**, data is transferred at regular intervals that are timed by a clocking signal, allowing for a constant and reliable transmission for time-sensitive data, such as real-time video or voice. Parallel communication typically uses synchronous transmission – for example, in the CPU, the clock emits a signal at regular intervals and transmissions along the address bus, data bus and control bus start on a clock signal, which is shared by both sender and receiver.

Asynchronous transmission

Using **asynchronous transmission**, one byte at a time is sent, with each character being preceded by a start bit and followed by a stop bit.

The start bit alerts the receiving device and synchronises the clock inside the receiver ready to receive the character. The baud rate at the receiving end has to be set up to be the same as the sender's baud rate or the signal will not be received correctly. The stop bit is actually a "stop period", which may be arbitrarily long. This allows the receiver time to identify the next start bit and gives the receiver time to process the data before the next value is transmitted.

A parity bit is also usually included as a check against incorrect transmission. Thus for each character being sent, a total of 10 bits is transmitted, including the parity bit, a start bit and a stop bit. The start bit may be a 0 or a 1, the stop bit is then a 1 or a 0 (always different). A series of electrical pulses is sent down the line as illustrated below:



Asynchronous transmission

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.**

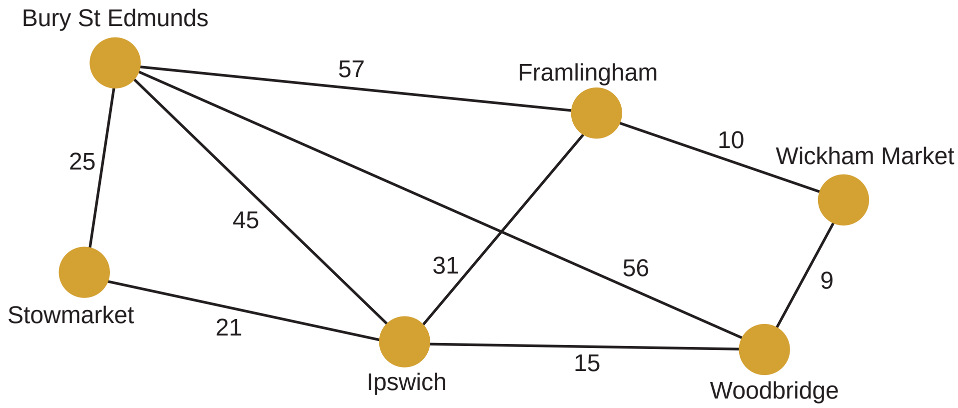


Figure 41.1: An undirected graph with weighted edges

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.

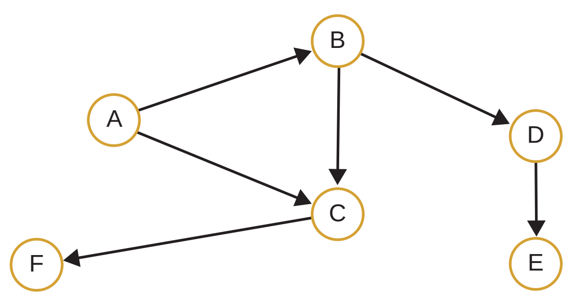


Figure 41.2: A directed, unweighted graph

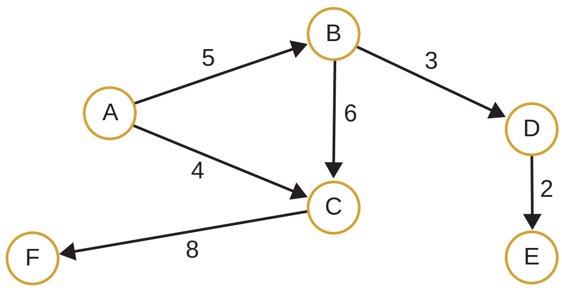
7-41

Implementing a graph

Two possible implementations of a graph are the **adjacency matrix** and the **adjacency list**.

The adjacency matrix

A two-dimensional array can be used to store information about a directed or undirected graph. Each of the rows and columns represents a node, and a value stored in the cell at the intersection of row i, column j indicates that there is an edge connecting node i and node j.



	A	B	C	D	E	F
A		5	4			
B			6	3		
C						8
D					2	
E						
F						

In the case of an **undirected graph**, the adjacency matrix will be symmetric, with the same entry in row 0 column 1 as in row 1 column 0, for example.

An unweighted graph may be represented with 1s instead of weights, in the relevant cells.

7-41

Q1: Draw an adjacency matrix to represent the weighted graph shown in Figure 41.1.

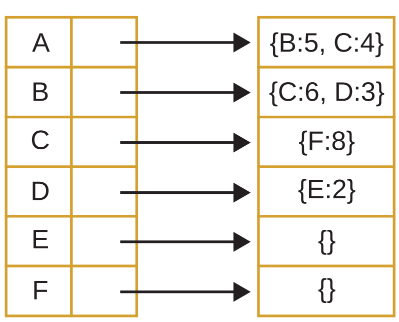
Advantages and disadvantages of the adjacency matrix

An adjacency matrix is very convenient to work with, and adding an edge or testing for the presence of an edge is very simple and quick. However, a sparse graph with many nodes but not many edges will leave most of the cells empty, and the larger the graph, the more memory space will be wasted. Another consideration is that using a static two-dimensional array, it is harder to add or delete nodes.

The adjacency list

An adjacency list is a more space-efficient way to implement a sparsely connected graph. A list of all the nodes is created, and each node points to a list of all the adjacent nodes to which it is directly linked. The adjacency list can be implemented as a list of dictionaries, with the key in each dictionary being the node and the value, the edge weight.

The graph above would be represented as follows:



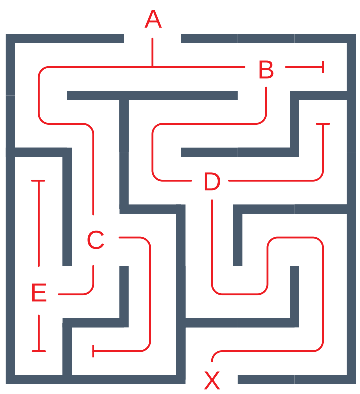
Applications of depth-first search

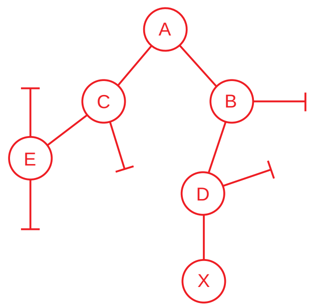
Applications of the depth-first search include the following:

- In scheduling jobs where a series of tasks is to be performed, and certain tasks must be completed before the next one begins.
- In solving problems such as mazes, which can be represented as a graph

Finding a way through a maze

A depth-first search can be used to find a way out of a maze. Junctions where there is a choice of route in the maze are represented as nodes on a graph.





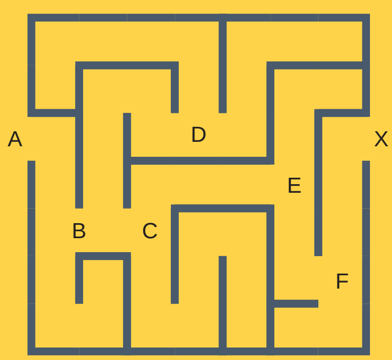
Q1: (a) Redraw the graph without showing the dead ends.

(b) State the properties of this graph that makes it a tree.

(c) Complete the table below to show how the graph would be represented using an adjacency matrix.

	A	B	C	D	E	X
A						
B						
C						
D						
E						
X						

Q2: Draw a graph representing the following maze. Show the dead ends on your graph.



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

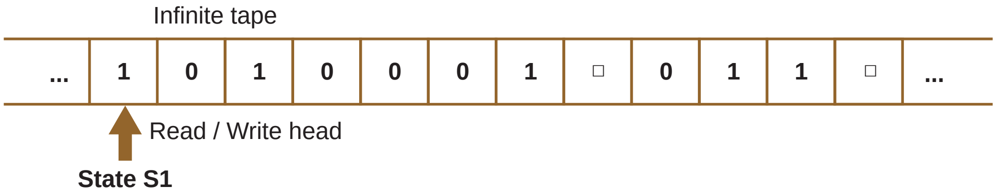
Alan Turing (1912–1954) was a British computer scientist and mathematician, best known for his work at Bletchley Park during the Second World War. While working there, he devised an early computer for breaking German ciphers, work which probably shortened the war by two or more years and saved countless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

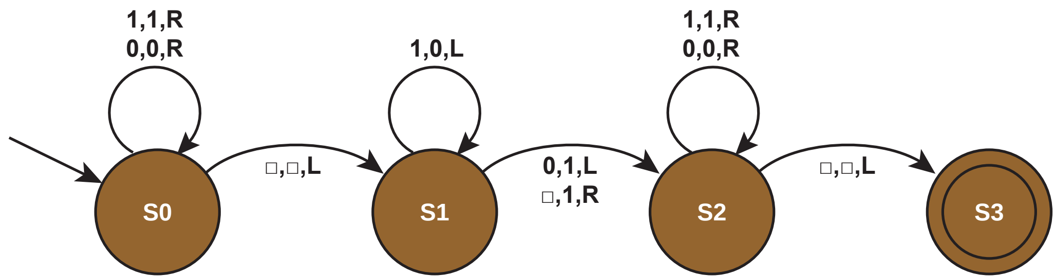
The Turing machine consists of an infinitely long strip of tape divided into squares. It has a read/write head that can read symbols from the tape and make decisions about what to do based on the contents of the cell and its current state.

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.

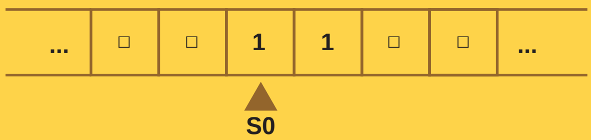


The Turing machine is an early precursor of the modern computer, with input, output and a program which describes its behaviour. Any alphabet may be defined for the Turing machine; for example a binary alphabet of 0, 1 and □ (representing a blank), as shown in the diagram above.

The finite state machine corresponding to the state transition diagram is given below.



Q1: Trace the computation of the Turing machine if the tape starts with the data 11 as shown below.



(You will need to draw ten representations of the tape to complete the computation.)

Transition functions

The transition rules for any Turing machine can be expressed as a **transition function** δ. The rules are written in the form

δ (Current State, Input symbol) = (Next State, Output symbol, Movement).

Thus the rule

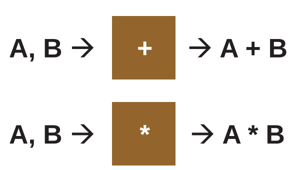
δ (S1, 0) = (S2, 1, L)

means "IF the machine is currently in state S1 and the input symbol read from the tape is 0, THEN write a 1 to the tape, and move left and change state to S2".

Q2: Looking at the state transition diagram above, write the transition rules for inputs of 0, 1 and □ when the machine is in state S0.

The universal Turing machine

A Turing machine can theoretically represent any computation.



Each machine has a different program to compute the desired operation. However, the obvious problem with this is that a different machine has to be created for each operation, which is clearly impractical.

Turing therefore came up with the idea of the **Universal Turing machine**, which could be used to compute any computable sequence. He wrote: "If this machine **U** is supplied with the tape on the beginning of which is written the string of quintuples separated by semicolons of some computing machine **M**, then **U** will compute the same sequence as **M**."

Chapter 68 – Object-oriented design principles

Objectives

- Understand concepts of association, composition and aggregation
- Understand the use of polymorphism and overriding
- Be aware of object-oriented design principles:
    - encapsulate what varies
    - favour composition over inheritance
    - program to interfaces, not implementation
- Be able to draw and interpret class diagrams

Association, aggregation and composition

Recall that inheritance is based on an "is a" relationship between two classes. For example, a cat "is a(n)" animal, a car "is a" vehicle. In a similar fashion, **association** may be loosely described as a "**has a**" relationship between classes. Thus a railway company may be associated with the engines and carriages it owns, or the track that it maintains. A teacher may be associated with a form bi-directionally – a teacher "has a" student, and a student "has a" teacher. However, there is no **ownership** between objects and each has their own lifecycle, and can be created and deleted independently.

Association aggregation, or simply **aggregation**, is a special type of more specific association. It can occur when a class is a collection or container of other classes, but the contained classes do not have a strong lifecycle dependency on the container. For example, a player who is part of a team does not cease to exist if the team is disbanded.

Aggregation may be shown in class diagrams using a hollow diamond shape between the two classes.

Class diagram showing association aggregation

Composition aggregation, or simply **composition**, is a stronger form of aggregation. If the container is destroyed, every instance of the contained class is also destroyed. For example if a hotel is destroyed, every room in the hotel is destroyed.

Composition may be shown in class diagrams using a filled diamond shape. The diamond is at the end of the class that owns the creational responsibility.

Class diagram showing composition aggregation

Q1: Specify whether each of the following describe **association aggregation** or **composition aggregation**.

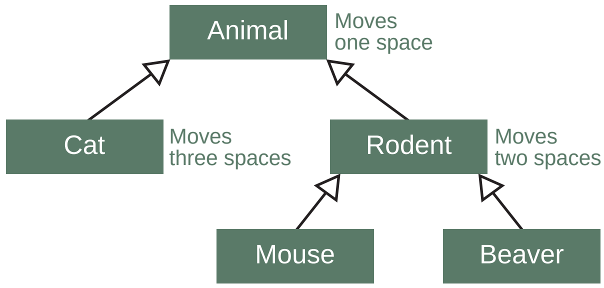
(a) Zoo and ZooAnimal

(b) RaceTrack and TrackSection

(c) Department and Teacher

Polymorphism

Polymorphism refers to a programming language's ability to process objects differently depending on their class. For example, in the last chapter we looked at an application that had a superclass `Animal`, and subclasses `Cat` and `Rodent`. All objects in subclasses of `Animal` can execute the methods `moveLeft`, `moveRight`, which will cause the animal to move one space left or right.



We might decide that a `cat` should move three spaces when a `moveLeft` or `moveRight` message is received, and a `Rodent` should move two spaces. We can define different methods within each of the classes to implement these moves, but keep the same method name for each class.

Defining a method with the same name and formal argument types as a method inherited from a superclass is called **overriding**. In the example above, the `moveLeft` method in each of the `Cat` and `Rodent` classes overrides the method in the superclass `Animal`.

12-68

Q2: Suppose that `tom` is an instance of the `Cat` class, and `jerry` is an instance of the `Mouse` class. What will happen when each of these statements is executed?

tom.moveRight()

jerry.moveRight()

Q3: Looking at the diagram above, what changes do you need to make so that `bertie`, an instance of the `Beaver` class, moves only one space when given a `moveRight()` message?

Class definition including override

Class definitions for the classes `Animal` and `Cat` will be something like this:

```
Animal = Class
        Public
           Procedure moveLeft
           Procedure moveRight
        Protected
           Position: Integer
        End
Cat = Subclass (Animal)
        Public
           Procedure moveLeft (Override)
           Procedure moveRight (Override)
           Procedure pounce
        Private
           Name: String
        End
```

Note: The 'Protected' access modifier is described on page 356.

Index

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**

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

www.heatherduncan.com

This book has been approved by AQA.

PG ONLINE

ISBN 978-1-910523-07-0