



AQA AS and A Level

Computer Science



PG ONLINE

PM Heathcote and
RSU Heathcote

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

Copyright © P.M.Heathcote and R.S.U.Heathcote 2016

All rights reserved

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the prior written permission of the copyright owner.

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 1

Fundamentals of programming		1
Chapter 1	Programming basics	2
Chapter 2	Selection	8
Chapter 3	Iteration	13
Chapter 4	Arrays	17
Chapter 5	Subroutines	21
Chapter 6	Files and exception handling	29

Section 2

Problem solving and theory of computation		33
Chapter 7	Solving logic problems	34
Chapter 8	Structured programming	39
Chapter 9	Writing and interpreting algorithms	42
Chapter 10	Testing and evaluation	48
Chapter 11	Abstraction and automation	52
Chapter 12	Finite state machines	60

Section 3

Data representation		67
Chapter 13	Number systems	68
Chapter 14	Bits, bytes and binary	72
Chapter 15	Binary arithmetic and the representation of fractions	77
Chapter 16	Bitmapped graphics	83
Chapter 17	Digital representation of sound	88
Chapter 18	Data compression and encryption algorithms	93

Section 4

Hardware and software	99
Chapter 19 Hardware and software	100
Chapter 20 Role of an operating system	103
Chapter 21 Programming language classification	106
Chapter 22 Programming language translators	110
Chapter 23 Logic gates	114
Chapter 24 Boolean algebra	118

Section 5

Computer organisation and architecture	125
Chapter 25 Internal computer hardware	126
Chapter 26 The processor	132
Chapter 27 The processor instruction set	138
Chapter 28 Assembly language	142
Chapter 29 Input-output devices	148
Chapter 30 Secondary storage devices	154

Section 6

Communication: technology and consequences	158
Chapter 31 Communication methods	159
Chapter 32 Network topology	164
Chapter 33 Client-server and peer-to-peer	168
Chapter 34 Wireless networking, CSMA and SSID	171
Chapter 35 Communication and privacy	176
Chapter 36 The challenges of the digital age	179

Section 7

Data structures

187

Chapter 37	Queues	188
Chapter 38	Lists	194
Chapter 39	Stacks	198
Chapter 40	Hash tables and dictionaries	202
Chapter 41	Graphs	207
Chapter 42	Trees	211
Chapter 43	Vectors	217

Section 8

Algorithms

223

Chapter 44	Recursive algorithms	224
Chapter 45	Big-O notation	229
Chapter 46	Searching and sorting	235
Chapter 47	Graph-traversal algorithms	243
Chapter 48	Optimisation algorithms	249
Chapter 49	Limits of computation	254

Section 9

Regular languages

259

Chapter 50	Mealy machines	260
Chapter 51	Sets	265
Chapter 52	Regular expressions	269
Chapter 53	The Turing machine	273
Chapter 54	Backus-Naur Form	278
Chapter 55	Reverse Polish notation	283

Section 10

The Internet	287
Chapter 56 Structure of the Internet	288
Chapter 57 Packet switching and routers	292
Chapter 58 Internet security	294
Chapter 59 TCP/IP, standard application layer protocols	300
Chapter 60 IP addresses	307
Chapter 61 Client server model	313

Section 11

Databases and software development	318
Chapter 62 Entity relationship modelling	319
Chapter 63 Relational databases and normalisation	323
Chapter 64 Introduction to SQL	330
Chapter 65 Defining and updating tables using SQL	336
Chapter 66 Systematic approach to problem solving	342

Section 12

OOP and functional programming	346
Chapter 67 Basic concepts of object-oriented programming	347
Chapter 68 Object-oriented design principles	353
Chapter 69 Functional programming	360
Chapter 70 Function application	367
Chapter 71 Lists in functional programming	371
Chapter 72 Big Data	374
References	379

Appendices and Index

Appendix A Floating point form	380
Appendix B Adders and D-type flip-flops	387
Index	391

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:

<code>len(string)</code>	Returns the length of a string
<code>string.substring(index1, index2)</code>	Returns a portion of <code>string</code> inclusive of the characters at each index position
<code>string.find(str)</code>	Determines if <code>str</code> 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 <code>string(1)</code> is the first element of the string, though in Python, for example, the first element is <code>string(0)</code>
<code>ord("a")</code>	Returns the integer value of a character (97 in this example)
<code>chr(97)</code>	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
```

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

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

String conversion operations

<code>int("1")</code>	converts the character "1" to the integer 1
<code>str(123)</code>	converts the integer 123 into a string "123"
<code>float("123.456")</code>	converts the string "123.456" to the real number 123.456
<code>str(123.456)</code>	converts the real number 123.456 to the string "123.456"
<code>date(year, month, day)</code>	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).

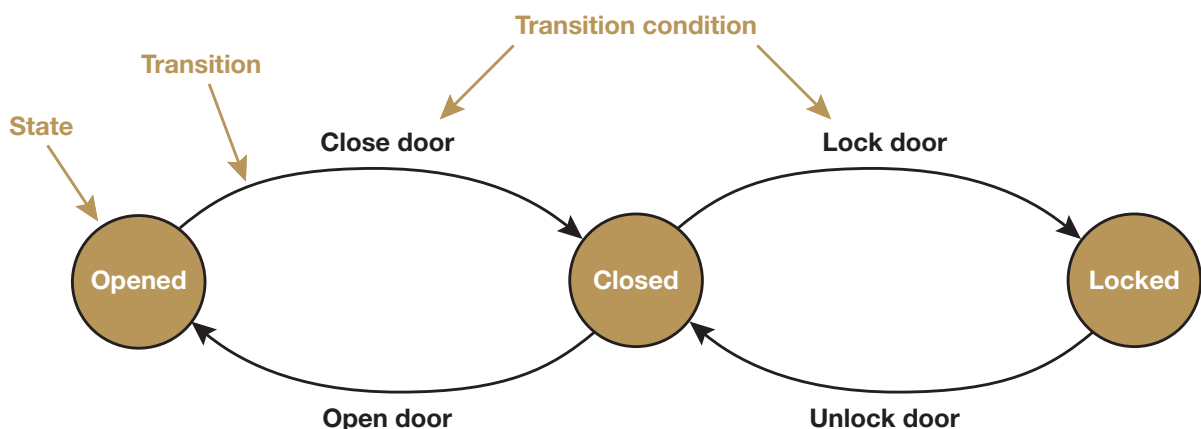
In a finite state machine:

- 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

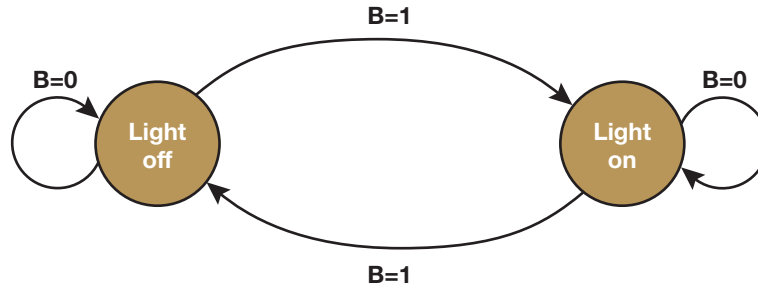
Draw an FSM to model the states and transitions of a door. The door can be open, closed or locked. It can change from the state of being open to closed, from closed to locked, but not, say, from locked to open. (It has to be unlocked first.)



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.

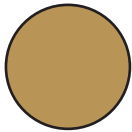
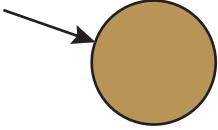


Usage of finite state machines

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, \dots, 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.

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.

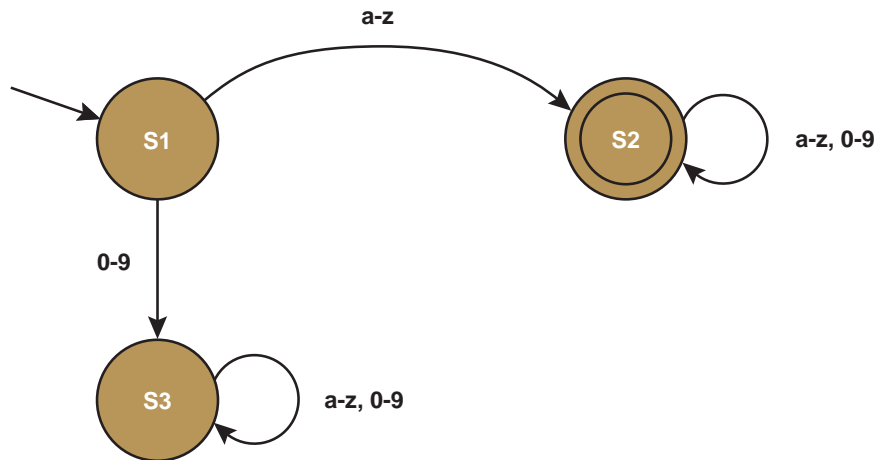
Notation

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.

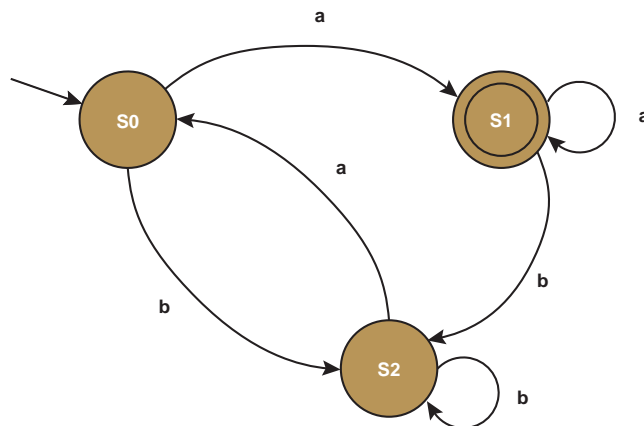
Each character of the input string is input sequentially to the FSM and if the last character reaches the final state S2 (the **accept** state), the string is valid and is accepted. If it ends up anywhere else the string is invalid.

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

2-12

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

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.

DG YD QFH WR ER UGHU 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?

Using this method, the message **“Meet on the bridge at 0300 hours”** encrypted using a one-time pad of **+tkiGeMxGvnhoQ0xQDIIVdT4slJm9qf** will produce the ciphertext:

f#X3H#Y6!i(=vTgCi"γL

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.

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.

5-27

Exercises

- A computer with a 16-bit word length uses an instruction set with 6 bits for the opcode, including the addressing mode.
 - What is an *instruction set*? [1]
 - How many instructions could be included in the instruction set of this computer? [1]
 - What is the largest number that can be used as data in the instruction? [1]
 - What would be the effect of increasing the space allowed for the opcode by 2 bits? [2]
 - What would be the benefits of increasing the word size of the computer? [2]

- The high-level language statement

$$X = Y + 6$$

is to be written in assembly language.

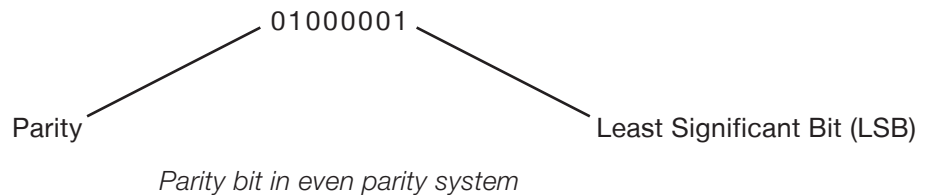
Complete the following assembly language statements, which are to be the equivalent of the above high level language statement. The LOAD and STORE instructions imply the use of the accumulator register.

```
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).



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.

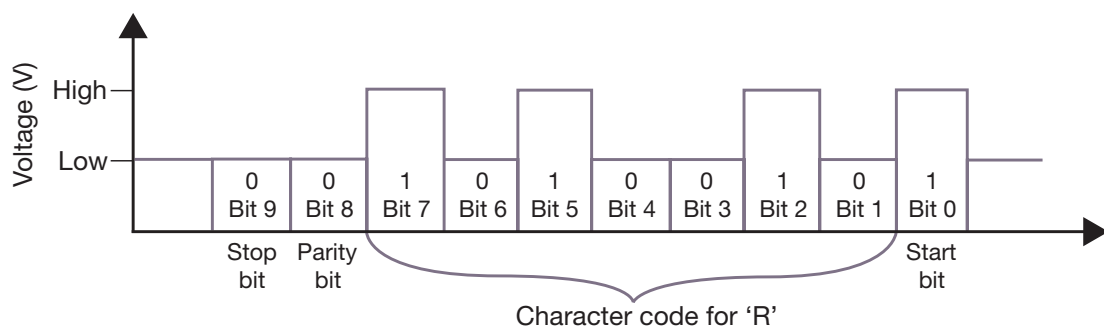
6-31

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**.

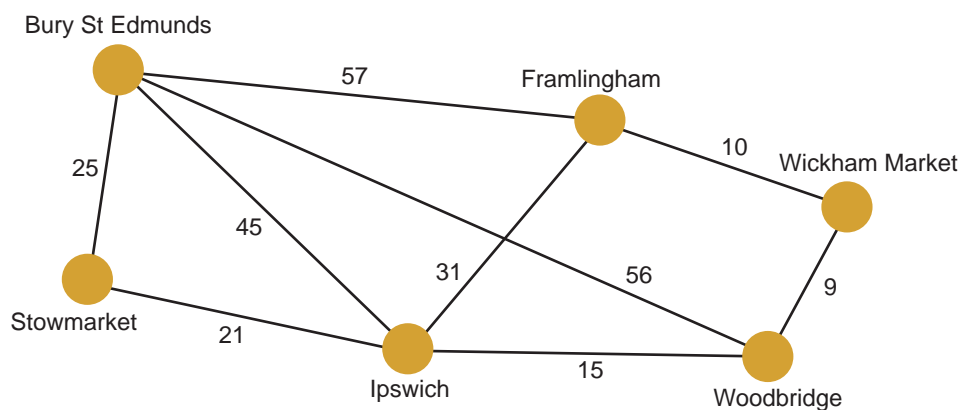


Figure 41.1: An undirected graph with weighted edges

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.

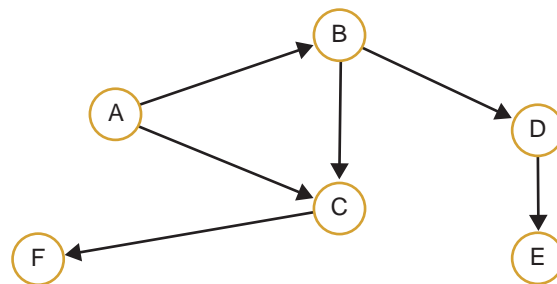


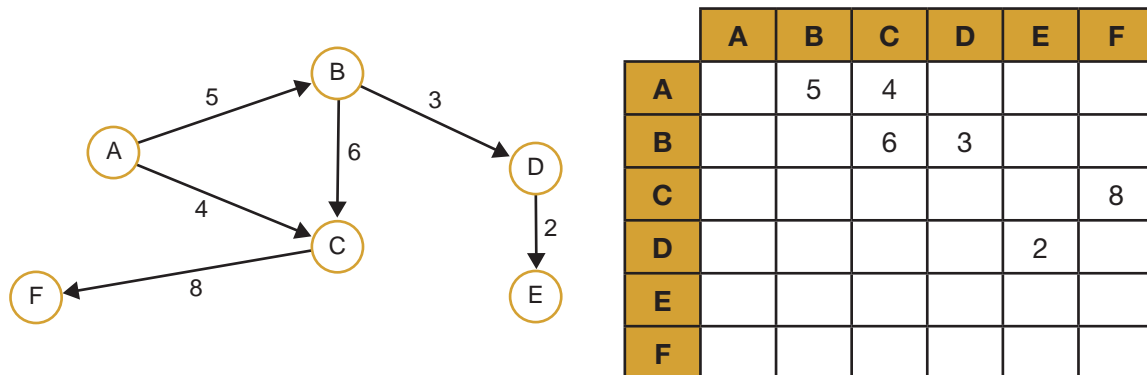
Figure 41.2: A directed, unweighted graph

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 .



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.

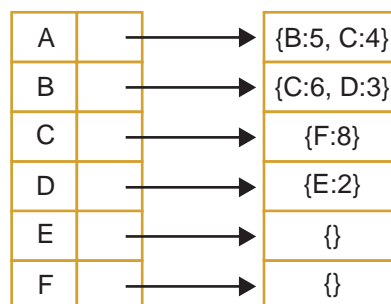
Advantages and disadvantages of the adjacency matrix

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:



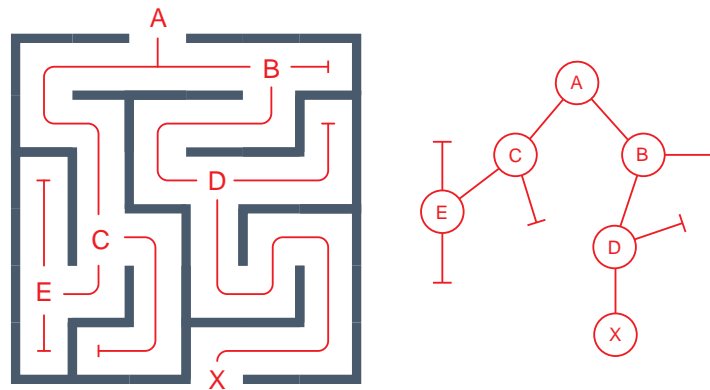
Applications of depth-first search

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

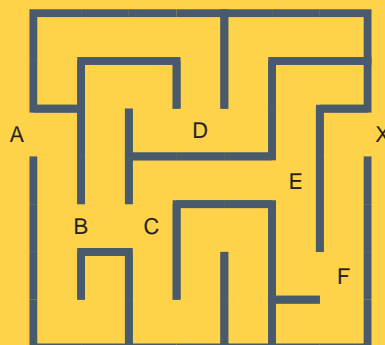
A depth-first search can be used to find a way out of a maze. Junctions where there is a choice of route in the maze are represented as nodes on a graph.



- 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.

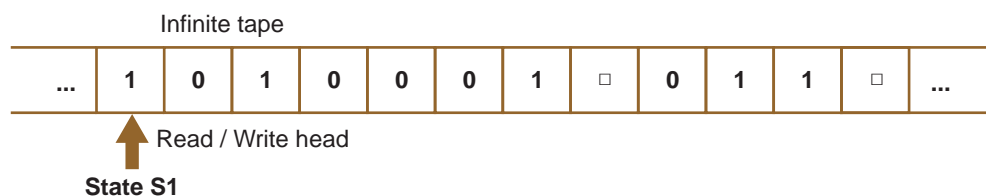


9-53

The Turing machine

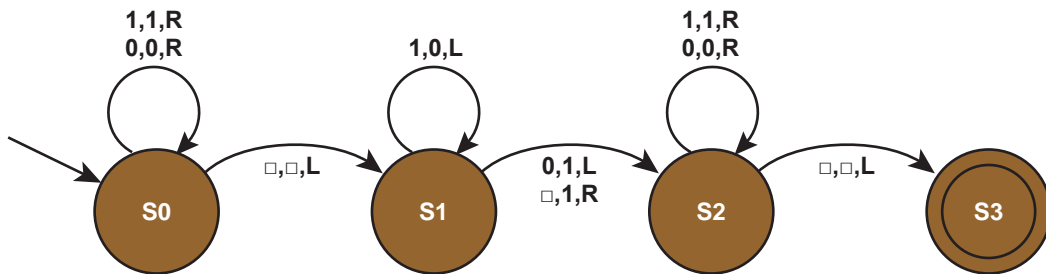
The Turing machine consists of an infinitely long strip of tape divided into squares. It has a read/write head that can read symbols from the tape and make decisions about what to do based on the contents of the cell and its current state.

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

$$\delta (\text{Current State, Input symbol}) = (\text{Next State, Output symbol, Movement}).$$

Thus the rule

$$\delta (S1, 0) = (S2, 1, L)$$

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 \square when the machine is in state S0.

The universal Turing machine

A Turing machine can theoretically represent any computation.

$$A, B \rightarrow \boxed{+} \rightarrow A + B$$

$$A, B \rightarrow \boxed{*} \rightarrow A * B$$

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

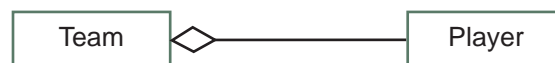
- Understand concepts of association, composition and aggregation
- Understand the use of polymorphism and overriding
- Be aware of object-oriented design principles:
 - encapsulate what varies
 - favour composition over inheritance
 - program to interfaces, not implementation
- Be able to draw and interpret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



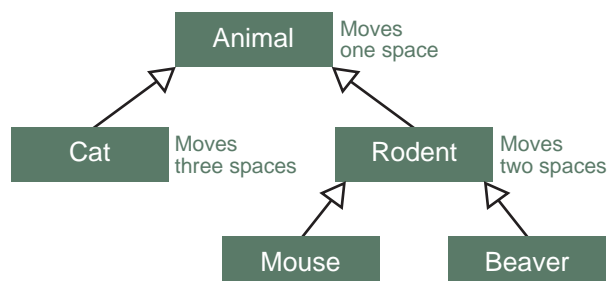
Class diagram showing composition aggregation

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

- Zoo and ZooAnimal
- RaceTrack and TrackSection
- 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`.

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

A

absolute error, 385
 abstract data types, 188
 abstraction, 52, 108
 data, 57
 functional, 56
 problem, 57
 procedural, 55
 accumulator, 132, 138
 active tags, 152
 ADC, 90
 adders
 concatenating, 387
 address bus, 127, 128, 135
 addressing mode
 direct, 139
 immediate, 139
 adjacency
 list, 208
 matrix, 208
 ADT, 188
 aggregation, 353
 agile modelling, 342
 Alan Turing, 273
 algorithm, 2
 ALU, 132
 Amazon, 179
 analogue
 data, 89
 to digital conversion, 90
 analysis, 34, 342
 AND, 10, 144
 AND gate, 115
 API, 313
 appending, 372
 application layer, 300, 301
 Application Programming
 Interface, 103, 313
 application software, 102
 arithmetic logic unit, 127, 132
 arithmetic operations, 3, 127, 143
 ARPANET, 288
 array, 17, 19, 190
 ASCII, 73
 assembler, 110
 assembly language, 108,
 109, 140, 142
 association, 353
 asymmetric encryption, 296

asynchronous transmission, 162
 attributes, 319, 347
 audio bit depth, 88
 automation, 58
 automaton, 61

B

backing store management, 104
 Backus-Naur form, 278
 bandwidth, 161
 barcode reader, 149
 barcodes
 2-D, 148
 linear, 148
 base case, 224
 baud rate, 161
 behaviours, 347
 Big Data, 374
 Big-O notation, 229, 231
 binary
 addition, 77
 converting to and from decimal, 69
 file, 31
 fixed point, 80
 floating point, 81
 multiplication, 78
 negative numbers, 79
 number system, 69
 subtraction, 80
 binary expression tree, 286
 binary search, 236
 recursive algorithm, 237
 tree, 212
 binary search tree, 215
 binary tree search, 238
 bit, 72
 depth, 88
 rate, 161
 bitmap image, 83
 block-structured languages, 39
 Blu-Ray, 155
 BNF, 278
 Boolean algebra, 120
 Absorption rules, 120
 Associative rules, 120
 Commutative rules, 120
 Distributive rules, 120
 Boolean operators, 10

breadth-first
 search, 248
 traversal, 245, 246
 bridges of Königsberg, 54
 browser, 305
 bubble sort, 44, 238
 bus, 127
 address, 128
 control, 128
 data, 128
 byte, 72
 bytecode, 112

C

cache memory, 135
 Caesar cipher, 96
 call stack, 200, 225
 camera-based readers, 150
 cardinality, 265
 carry, 78
 Cartesian product, 266
 CASE, 10
 CCD reader, 150
 CD-ROM, 155
 Central Processing Unit, 126
 check digit, 75
 checksum, 75, 292
 ciphertext, 96, 295
 CIR, 133
 circular queue, 190
 class, 348
 classful addressing, 308
 classless addressing, 308
 client-server
 database, 339
 model, 313
 network, 168
 clock speed, 135
 CMOS, 151
 co-domain, 360
 collision, 202
 resolution, 204
 Colossus computer, 106
 colour depth, 83
 comments, 3
 commitment ordering, 340
 compact representation, 266
 compare and branch
 instructions, 143

compiler, 110, 112
 composite data types, 188
 composition, 57, 353
 compression
 dictionary-based, 95
 lossless, 93
 lossy, 93
 computability, 273
 computable problems, 256
 computational thinking, 35, 52
 Computer Misuse Act, 183
 constant, 6
 constructor, 348
 control bus, 127, 128
 control unit, 127, 132
 convex combination, 220
 Copyright, Designs and
 Patents Act (1988), 183
 CPU, 126
 CRC, 292
 CRUD, 314
 cryptanalysis, 96, 97
 CSMA/CA, 173
 CSMA/CD, 166
 CSS Object Model, 305
 CSSOM, 305
 current instruction register, 133
 cyber-attack, 177
 cyber-bullying, 181
 cyclical redundancy check, 292

D

DAC, 90
 data
 analogue, 89
 boundary, 48
 bus, 127, 128, 135
 communication, 159
 digital, 89
 erroneous, 48
 normal, 48
 structures, 17
 transfer operations, 143
 types, 3
 user-defined type, 29
 data abstraction, 188
 data packets, 292
 Data Protection Act (1998), 183
 database

 defining a table, 336
 locking, 340
 normalisation, 324
 relational, 323
 De Morgan's laws, 118
 decomposition, 57
 denary, 80
 depth-first
 traversal, 243
 design, 34, 343
 destruction of jobs, 180
 dictionary, 205
 dictionary based compression, 95
 digital
 camera, 151
 certificate, 297
 data, 89
 signature, 296
 to analogue conversion, 90
 digraph, 207
 Dijkstra's algorithm, 249, 293
 directed graph, 207
 disk defragmenter, 101
 divide and conquer, 43
 DNS, 290
 Document Object Model, 305
 DOM, 305
 domain, 360
 domain name, 289, 290
 fully qualified, 291
 Domain Name System, 290
 dot product, 220
 DPI, 83
 driverless cars, 182
 dry run, 49
 D-type flip-flop, 388, 389
 dual-core processor, 134
 dynamic data structure, 190
 dynamic filtering, 295

E

EAN, 76
 early computers, 106
 eBay, 179
 edge, 207
 elementary data types, 17, 188
 embedded systems, 130
 encapsulating what varies, 357
 encapsulation, 188, 350

encryption, 96, 295
 asymmetric, 296
 private key, 296
 public key, 296
 symmetric, 296
 Enigma code, 106
 entity, 319
 identifier, 319
 relationship diagram, 320, 321
 error checking, 74
 ethics, 182
 evaluating a program, 46
 evaluation, 50, 344
 event messages, 91
 exbi, 72
 exponent, 381
 exponential function, 230

F

fact-based model, 377
 fetch-execute cycle, 134
 field, 29
 FIFO, 188
 file, 29
 binary, 31
 server, 168
 text, 29
 File Transfer Protocol, 303
 filter, 370
 finite set, 265
 finite state
 automaton, 61, 260
 machine, 60, 260
 firewall, 294
 first generation language, 53
 First In First Out, 188
 First normal form, 324
 first-class object, 362
 fixed point, 385
 floating point, 385
 binary numbers, 381
 fold (reduce), 370
 folding method, 203
 FOR ... ENDFOR, 15
 foreign key, 320, 324
 FQDN, 291
 frequency of a sound, 90
 FSM, 260
 FTP, 303

full adder, 387
Fully Qualified Domain Names, 291
function, 360
 application, 362
 higher-order, 367
functional
 composition, 364
 programming, 360
functions, 5, 21, 230
 string-handling, 5

G

Galois field, 220
gate
 NOT, AND, OR, 114
 XOR, NAND, NOR, 116
gateway, 293
general purpose registers, 132
getter messages, 349
GF(2), 220
gibi, 72
Google, 179
 Street View, 178
graph, 207
 schema, 377
 theory, 55
 traversals, 243
half-adder, 387
Halting problem, 257
hard disk, 154
hardware, 100
Harvard architecture, 130
hash table, 202
hashing algorithm, 202
folding method, 203

H

Haskell, 360, 361
heuristic methods, 256
hexadecimal, 70
hierarchy chart, 40
higher-order function, 367
high-level languages, 109
HTTP request methods, 314

I

I/O controller, 127, 129
IF ... THEN, 8
image resolution, 83

immutable, 363, 372
imperative language, 109
implementation, 344
infinite set, 266
infix expression, 284
information hiding, 54, 57, 188, 350
inheritance, 351
in-order traversal, 214, 225, 226
Instagram, 181
instantiation, 348
instruction set, 107, 110
interface, 23, 129, 357
Internet
 registrars, 289
 registries, 290
 security, 172, 294
 Service Providers, 289
Internet of things, 182
interpreter, 111, 112
interrupt, 136
 handling, 105
Interrupt Service Routine, 136
intractable problems, 255
IP address, 291
 private, 309
 public, 309
 structure, 307
irrational number, 68
ISBN, 76
ISP, 289
Iteration, 13

J

Java Virtual Machine, 112
JSON, 315, 316

K

kibi, 72
kilobyte, 72

L

LAN, 164
laser
 printer, 152
 scanner, 150
latency, 161
legislation, 183
library programs, 101
limits of computation, 254

linear function, 230
linear search, 235
link layer, 300, 301
linking database tables, 324
list, 194, 371
 appending to, 372
 prepending to, 372
loader, 103
local area network, 164
logarithmic function, 231
logic gates, 114
logical bitwise operators, 144
logical operations, 127
low-level language, 108

M

MAC address, 167, 302
machine code, 106
 instruction format, 138
mail server, 304
majority voting, 75
malicious software, 297
malware, 297
mantissa, 381
many-to-many relationship, 321, 326
map, 369
MAR, 133
maze, 247
MBR, 133
Mealy machines, 260, 261
mebi, 72
Media Access Control, 301
memory
 address register, 133
 buffer register, 133
 data register, 133
 management, 104
merge sort, 239
 space complexity, 241
 time complexity, 241
metadata, 84
meta-languages, 278
MIDI, 91
metadata, 91
mnemonics, 142
modelling data requirements, 343
modular programming, 25
module, 39
modulo 10 system, 76

N

NAND gate, 116
 NAT, 310
 natural number, 68, 265
 nested loops, 15
 network
 client-server, 168
 interface cards, 294
 layer, 300, 301
 peer-to-peer, 169
 security, 172, 294
 station, 171
 Network Address
 Translation, 310, 311
 nibble, 72
 NIC, 294
 node, 207
 non-computable problems, 256
 NOR gate, 116
 normal form
 first 1NF, 324
 second 2NF, 326
 third 3NF, 326
 normalisation, 327
 of databases, 324
 of floating point number, 382
 NOT, 10, 11, 144
 gate, 114
 number
 irrational, 68
 natural, 68
 ordinal, 68
 rational, 68
 real, 68
 Nyquist's theorem, 90

O

object code, 110
 object-oriented programming, 347
 one-time pad, 97
 opcode, 106, 138
 operand, 106, 138
 operating system, 100, 103
 operation code, 106, 138
 optical disk, 155
 OR, 10, 144
 gate, 115
 ORDER BY, 332
 ordinal number, 68

oscillator, 388
 overflow, 78, 386
 override, 354
 Oyster card, 152

P

packet filters, 294
 packet switching, 292
 PageRank algorithm, 209
 parallel data transmission, 160
 parity, 162
 bit, 74
 parity bit checker, 221
 partial dependency, 326
 partial function application, 368
 passive tags, 152
 PC, 133
 pebi, 72
 peer-to-peer network, 169
 pen-type reader, 149
 peripheral management, 105
 permutations, 231
 phishing, 299
 piracy, 170
 pixel, 83
 plaintext, 96, 295
 platform independence, 112
 polymorphism, 354
 polynomial function, 230
 polynomial-time solution, 255
 POP3, 304
 port forwarding, 311
 Post Office Protocol (v3), 304
 postfix
 expression, 284
 notation, 283
 post-order traversal, 214, 227
 precedence rules, 283
 pre-order traversal, 213, 227
 prepending, 372
 primary key, 319
 priority queue, 192
 private, 348
 key encryption, 296
 modifier, 356
 problem solving strategies, 36
 procedural programming, 347
 procedure, 21
 procedure interface, 56

processor, 127
 instruction set, 138
 performance, 134
 scheduling, 104
 program
 constructs, 8
 counter, 133
 programming paradigm, 360
 proper subset, 266
 protected access modifier, 356
 protocol, 163
 prototype, 343
 proxy server, 294, 295
 pseudocode, 2
 public, 348
 modifier, 356

Q

quad-core processor, 134
 queue, 188
 operations, 189
 Quick Response (QR) code, 148

R

Radio Frequency Identification, 151
 range, 79
 raster, 83
 rational number, 68, 265
 real number, 265
 record, 29
 record locking, 340
 recursion, 224
 recursive algorithm, 237
 reference variable, 349
 referential transparency, 363
 register, 127
 regular expressions, 269
 regular language, 270
 rehashing, 204
 relation, 323
 relational database, 320, 323
 relational operators, 8
 relationships, 320
 relative error, 385
 REPEAT ... UNTIL, 14
 Representational State Transfer, 314
 resolution, 83
 resource management, 100
 REST, 314

- Reverse Polish notation, 283
 - RFID, 151
 - RLE, 94
 - root node, 211
 - rooted tree, 211
 - rounding errors, 384
 - router, 171, 293
 - RTS/CTS, 173
 - Run Length Encoding, 94
- S**
- sample resolution, 88
 - scaling vectors, 220
 - Second normal form, 326
 - secondary storage, 154
 - Secure Shell, 304
 - SELECT .. FROM .. WHERE, 330
 - selection statement, 8
 - serial data transmission, 159
 - serialisation, 340
 - server
 - database, 168
 - file, 168
 - mail, 168
 - print, 168
 - web, 168
 - Service Set Identification, 172
 - set, 265
 - compact representation, 266
 - comprehension, 266
 - countable, 266
 - countably infinite, 266
 - difference, 267
 - intersection, 267
 - union, 267
 - setter messages, 349
 - side effects, 363
 - simulation, 188
 - Snowden, Edward, 176
 - social engineering, 299
 - software, 34, 100, 102
 - application, 102
 - bespoke, 102
 - development, 342
 - off-the-shelf, 102
 - system, 100
 - utility, 101
 - solid-state disk, 156
 - sorting algorithms, 44, 238
 - sound sample size, 89
 - source code, 110
 - space complexity, 241
 - spam filtering, 299
 - specifier
 - private, 356
 - protected access, 356
 - public, 356
 - SQL, 330, 338
 - SSD, 156
 - SSH, 304
 - SSID, 172
 - stack, 198
 - call, 200
 - frame, 201
 - overflow, 200
 - underflow, 200
 - state, 347
 - transition diagrams, 260
 - transition table, 261
 - stateful inspection, 295
 - stateless, 363
 - static data structure, 190
 - static filtering, 294
 - Static IP addressing, 310
 - stored program concept, 129
 - string conversion, 5
 - structured programming, 39
 - Structured Query Language, 330
 - subclass, 351
 - subnet mask, 308, 310
 - subnetting, 309
 - subroutines, 21
 - advantages of using, 25
 - user-written, 22
 - with interfaces, 23
 - subset, 266
 - substitution cipher, 96
 - superclass, 351
 - symmetric encryption, 296
 - synchronous transmission, 162
 - synonym, 202
 - syntax diagrams, 280
 - syntax error, 111
 - system
 - bus, 127
 - clock, 132
 - vulnerabilities, 298
- T**
- table structure, 336
 - TCP/IP protocol stack, 300
 - tebi, 72
 - Telnet, 304
 - test plan, 48
 - testing, 48, 344
 - text file, 29
 - thick-client computing, 316
 - thin-client computing, 316
 - Third normal form, 326
 - Tim Berners-Lee, 288
 - time complexity, 229, 233, 235, 236
 - of merge sort, 241
 - timestamp ordering, 340
 - topology
 - logical, 166
 - physical, 166
 - physical bus, 164
 - physical star, 165
 - trace table, 14, 49, 107
 - tractable problems, 255
 - transition functions, 276
 - translators, 101
 - transmission rate, 161
 - transport layer, 300, 301
 - travelling salesman problem, 254, 256
 - traversing a binary tree, 213
 - tree, 211
 - child, 211
 - edge, 211
 - leaf node, 211
 - node, 211
 - parent, 211
 - root, 211
 - subtree, 211
 - traversal algorithms, 225
 - trojans, 298
 - trolls, 181
 - truth tables, 114
 - TSP, 256
 - Turing machine, 273
 - two's complement, 80
 - typeclasses, 365
- U**
- underflow, 386
 - undirected graph, 207
 - Unicode, 74

Uniform Resource Locators, 289
union, 267
universal Turing machine, 276
URLs, 289
user generated content, 181
user interface, 100
user-defined data type, 29
utility software, 101

V

variables, 6
 global, 24
 local, 24
vector, 217
 adding and
 subtracting, 218
 convex
 combination, 220
 dot product, 220
 scaling, 220
vector graphics, 85
Vernam cipher, 96
vertex, 207
virtual memory, 104
virus checker, 101
viruses, 297
von Neumann, 100
 machine, 129

W

WAP, 171
web server, 305
WebSocket protocol, 314
weighted graph, 207
WHILE ... ENDWHILE, 13
whitelist, 172
Wi-Fi, 171
 Protected Access, 172
Wilkes, Maurice, 100
WinZip, 101
wireless network
 access point, 171
 interface controller, 171
word, 128
word length, 135
World Wide Web, 288
worms, 297
WPA, 172
WWW, 288

X

XML, 315, 316
XOR, 11, 144
 gate, 116

Y

yobi, 72

Z

zebi, 72

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

© Heather Duncan

www.heatherduncan.com

**This book has been
approved by AQA.**

