

AQA A Level **Year 2**

Computer Science



PG ONLINE

PM Heathcote and
RSU Heathcote

AQA A Level Year 2 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:

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

Cover picture © 'Blue Day'

Acrylic on board

Reproduced with the kind permission of Andrew Bird

www.abirdart.co.uk

Design and artwork by OnThree

www.on-three.com

Typeset by Ian Kingston

First edition 2016, reprinted 2016

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

ISBN: 978-1-910523-03-2

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 A Level Computer Science specification.

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

In the second year of this course there is a strong emphasis on algorithms and data structures, and these are covered in the first two sections of the book. 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

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 |

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**.

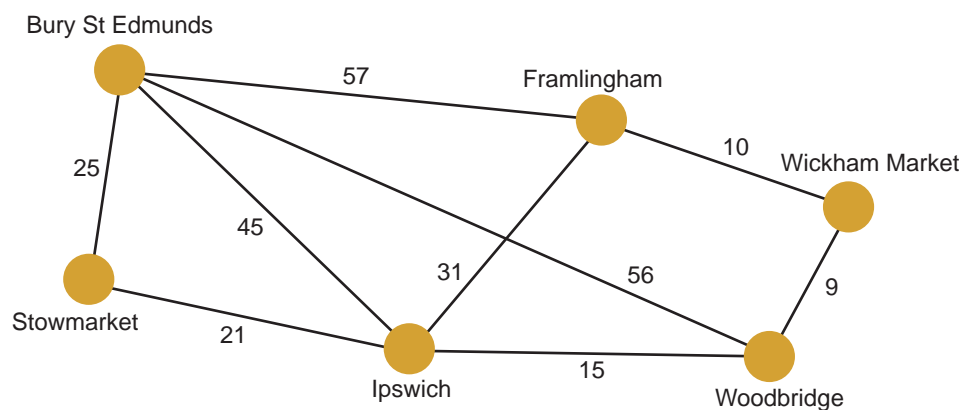


Figure 41.1: An undirected graph with weighted edges

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.

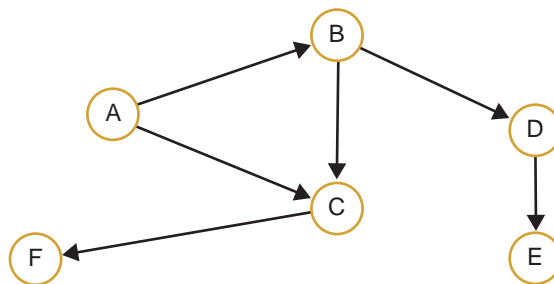


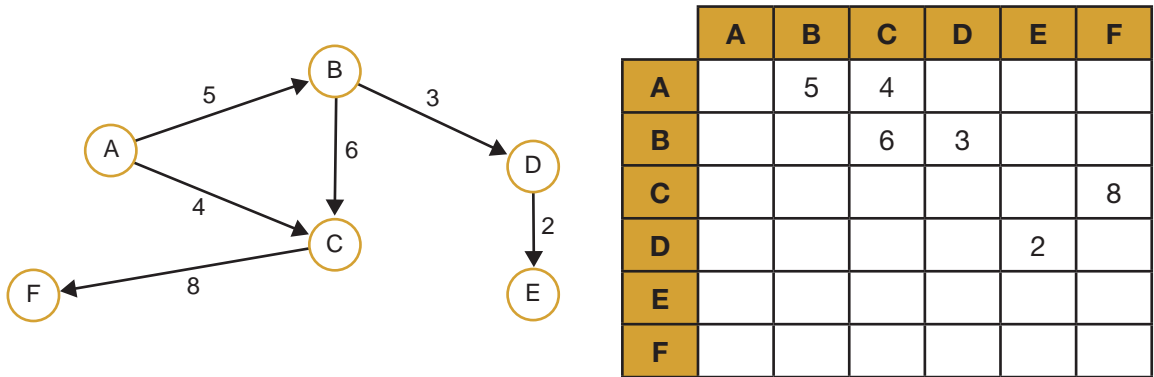
Figure 41.2: A directed, unweighted graph

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.

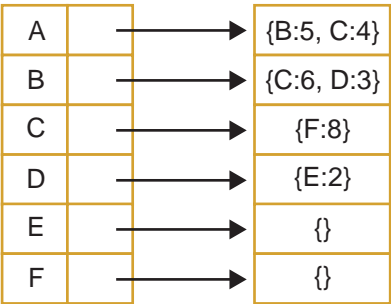
Advantages and disadvantages of the adjacency matrix

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:



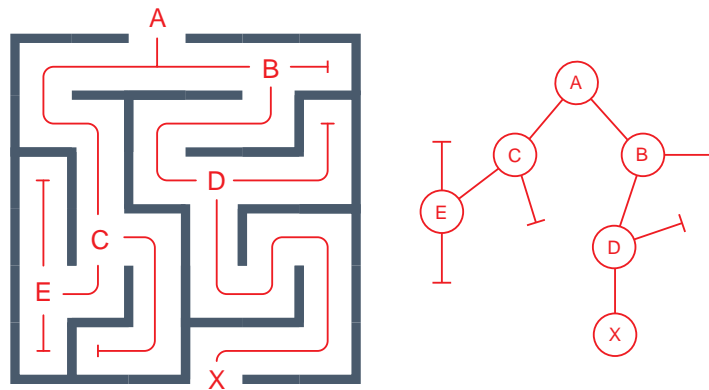
Applications of depth-first search

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

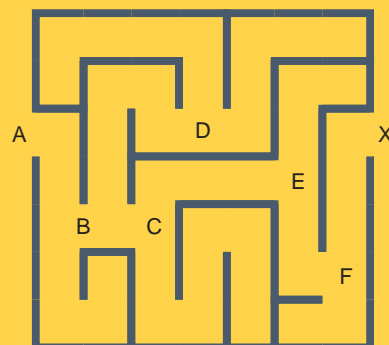
A depth-first search can be used to find a way out of a maze. Junctions where there is a choice of route in the maze are represented as nodes on a graph.



- 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.

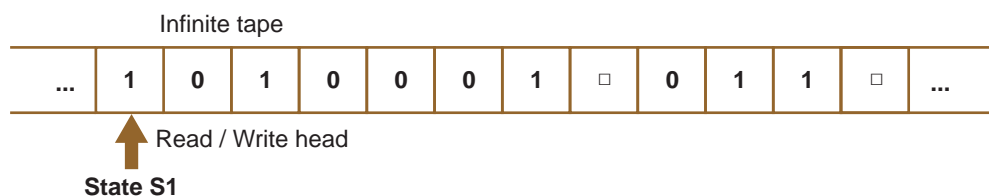


9-53

The Turing machine

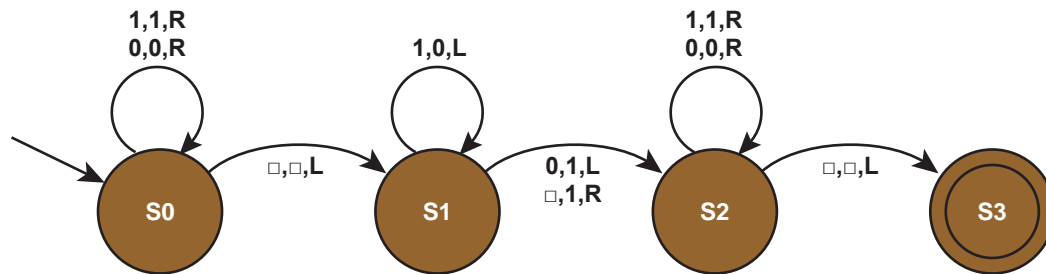
The Turing machine consists of an infinitely long strip of tape divided into squares. It has a read/write head that can read symbols from the tape and make decisions about what to do based on the contents of the cell and its current state.

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

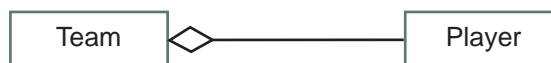
- Understand concepts of association, composition and aggregation
- Understand the use of polymorphism and overriding
- Be aware of object-oriented design principles:
 - encapsulate what varies
 - favour composition over inheritance
 - program to interfaces, not implementation
- Be able to draw and interpret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



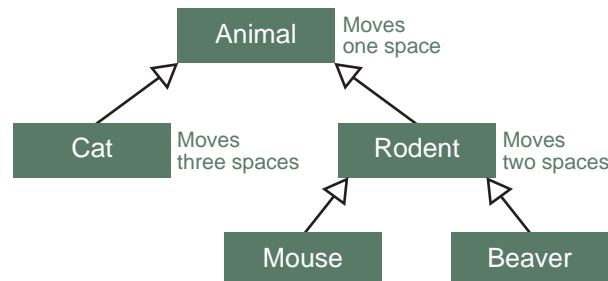
Class diagram showing composition aggregation

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`.

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

A

absolute error, 385
 abstract data types, 188
 adders
 concatenating, 387
 adjacency
 list, 208
 matrix, 208
 ADT, 188
 aggregation, 353
 agile modelling, 342
 Alan Turing, 273
 analysis, 342
 API, 313
 appending, 372
 application layer, 300, 301
 Application Programming
 Interface, 313
 ARPANET, 288
 array, 190
 association, 353
 asymmetric encryption, 296
 attributes, 319, 347

B

Backus-Naur form, 278
 base case, 224
 behaviours, 347
 Big Data, 374
 Big-O notation, 229, 231
 binary expression tree, 286
 binary search, 236
 recursive algorithm, 237
 tree, 212
 binary search tree, 215
 binary tree search, 238
 BNF, 278
 breadth-first
 search, 248
 traversal, 245, 246
 browser, 305
 bubble sort, 238

C

call stack, 200, 225
 cardinality, 265
 Cartesian product, 266
 checksum, 292
 ciphertext, 295

circular queue, 190
 class, 348
 classful addressing, 308
 classless addressing, 308
 client-server
 database, 339
 model, 313
 co-domain, 360
 collision, 202
 resolution, 204
 commitment ordering, 340
 compact representation, 266
 composite data types, 188
 composition, 353
 computability, 273
 computable problems, 256
 constructor, 348
 convex combination, 220
 CRC, 292
 CRUD, 314
 CSS Object Model, 305
 CSSOM, 305
 cyclical redundancy check, 292

D

data abstraction, 188
 data packets, 292
 database
 defining a table, 336
 locking, 340
 normalisation, 324
 relational, 323
 depth-first
 traversal, 243
 design, 343
 dictionary, 205
 digital
 certificate, 297
 signature, 296
 digraph, 207
 Dijkstra's algorithm, 249, 293
 directed graph, 207
 DNS, 290
 Document Object Model, 305
 DOM, 305
 domain, 360
 domain name, 289, 290
 fully qualified, 291
 Domain Name System, 290

dot product, 220
 D-type flip-flop, 388, 389
 dynamic data structure, 190
 dynamic filtering, 295

E

edge, 207
 elementary data types, 188
 encapsulating what varies, 357
 encapsulation, 188, 350
 encryption, 295
 asymmetric, 296
 private key, 296
 public key, 296
 symmetric, 296
 entity, 319
 identifier, 319
 relationship diagram, 320, 321
 evaluation, 344
 exponent, 381
 exponential function, 230

F

fact-based model, 377
 FIFO, 188
 File Transfer Protocol, 303
 filter, 370
 finite set, 265
 finite state
 automaton, 260
 machine, 260
 firewall, 294
 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
 foreign key, 320, 324
 FQDN, 291
 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, 230

G

Galois field, 220

gateway, 293

getter messages, 349

GF(2), 220

graph, 207

- schema, 377
- traversals, 243

H

half-adder, 387

Halting problem, 257

hash table, 202

hashing algorithm, 202

- folding method, 203

Haskell, 360, 361

heuristic methods, 256

higher-order function, 367

HTTP request methods, 314

I

immutable, 363, 372

implementation, 344

infinite set, 266

infix expression, 284

information hiding, 188, 350

inheritance, 351

in-order traversal, 214, 225, 226

instantiation, 348

interface, 357

Internet

- registrars, 289
- registries, 290
- security, 294
- Service Providers, 289

intractable problems, 255

IP address, 291

- private, 309
- public, 309
- structure, 307

ISP, 289

J

JSON, 315, 316

L

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

logarithmic function, 231

M

MAC address, 302

mail server, 304

malicious software, 297

malware, 297

mantissa, 381

many-to-many relationship, 321, 326

map, 369

maze, 247

Mealy machines, 260, 261

Media Access Control, 301

merge sort, 239

- space complexity, 241
- time complexity, 241

meta-languages, 278

modelling data requirements, 343

N

NAT, 310

natural number, 265

network

- interface cards, 294
- layer, 300, 301
- security, 294

Network Address

Translation, 310, 311

NIC, 294

node, 207

non-computable problems, 256

normal form

- first (1NF), 324
- second (2NF), 326
- third (3NF), 326

normalisation, 327

- of databases, 324

- of floating point number, 382

O

object-oriented programming, 347

ORDER BY, 332

oscillator, 388

overflow, 386

override, 354

P

packet filters, 294

packet switching, 292

PageRank algorithm, 209

parity bit checker, 221

partial dependency, 326

partial function application, 368

permutations, 231

phishing, 299

plaintext, 295

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

procedural programming, 347

programming paradigm, 360

proper subset, 266

protected access modifier, 356

prototype, 343

proxy server, 294, 295

public, 348

- modifier, 356

Q

queue, 188

- operations, 189

R

rational number, 265
 real number, 265
 record locking, 340
 recursion, 224
 recursive algorithm, 237
 reference variable, 349
 referential transparency, 363
 regular expressions, 269
 regular language, 270
 rehashing, 204
 relation, 323
 relational database, 320, 323
 relationships, 320
 relative error, 385
 Representational State Transfer, 314
 REST, 314
 Reverse Polish notation, 283
 root node, 211
 rooted tree, 211
 rounding errors, 384
 router, 293

S

scaling vectors, 220
 Second normal form, 326
 Secure Shell, 304
 SELECT .. FROM .. WHERE, 330
 serialisation, 340
 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
 social engineering, 299
 software development, 342
 sorting algorithms, 44, 238
 space complexity, 241
 spam filtering, 299
 specifier

- private, 356
- protected access, 356
- public, 356

SQL, 330, 338
 SSH, 304
 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
 Structured Query Language, 330
 subclass, 351
 subnet mask, 308, 310
 subnetting, 309
 subset, 266
 superclass, 351
 symmetric encryption, 296
 synonym, 202
 syntax diagrams, 280
 system

- vulnerabilities, 298

T

table structure, 336
 TCP/IP protocol stack, 300
 Telnet, 304
 testing, 344
 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
 tractable problems, 255
 transition functions, 276
 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
 TSP, 256
 Turing machine, 273
 typeclasses, 365

U

underflow, 386
 undirected graph, 207
 Uniform Resource Locators, 289
 union, 267
 universal Turing machine, 276
 URLs, 289

V

vector, 217

- adding and subtracting, 218
- convex combination, 220
- dot product, 220
- scaling, 220

 vertex, 207
 viruses, 297

W

web server, 305
 WebSocket protocol, 314
 weighted graph, 207
 World Wide Web, 288
 worms, 297
 WWW, 288

X

XML, 315, 316

AQA A Level **Year 2** **Computer Science**



The aim of this textbook is to provide a detailed understanding of each topic in the second year 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 ability to apply it.

The book is divided into six 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 wellknown and successful author of Computing 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:

'Blue Day'

Acrylic on board 51x61cm

© Andrew Bird

www.abirdart.co.uk

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



PG ONLINE

