AQA Approved

A Level Computer Science for AQA Unit 1

Kevin R Bond

Educational Computing Services

Single licence - Abingdon School

A Level Computer Science for AQA Unit 1

Kevin R Bond

Educational Computing Services Ltd

Single licence - Abingdon School

Structure of the book

The structure of this book follows closely the structure of AQA's A Level Computer Science specification for first teaching from September 2015. The content of the book has been constructed with the aim of promoting good teaching and learning, so where relevant practical activities have been suggested and questions posed for the student to answer. The book includes stimulus material to promote discussion and deeper thinking about the subject. Additional material to support teaching and learning will be available from the publisher's website. Please note that this additional material has not been entered in an AQA approval process.

About the author

Dr Kevin R Bond is an experienced author with a proven track record of successful writing for AQA's A Level Computing specifications. Kevin has 24 years of examining experience. He also has many more years of experience teaching AS and A Level Computing and Computer Science. Before becoming a computer science teacher, he worked in industry as a senior development engineer and systems analyst designing both hardware and software systems.

Single licence - Abingdon School

Published in 2017 by
Educational Computing Services Ltd
42 Mellstock Road
Aylesbury
Bucks
HP21 7NU
United Kingdom
Tel: 01296 433004
e-mail: mail@educational-computing.co.uk

Every effort has been made to trace copyright holders and to obtain their permission for the use of copyrighted material. We apologise if any have been overlooked. The authors and publishers will gladly receive information enabling them to rectify any reference or credit in future editions.

First published in 2017

978-0-9927536-5-8

Text © Kevin R Bond 2017
Original illustrations © Kevin R Bond 2017
Cover photograph © Kevin R Bond 2017

The right of Kevin R Bond to be identified as author of this work has been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording or any information storage retrieval system, without permission in writing from the publisher or under licence from the Copyright Licensing Agency Limited, of Saffron House, 6 -10 Kirby Street, London, EC1N 8TS.

Approval message from AQA

The core content of this digital textbook has been approved by AQA for use with our qualification. This means that we have checked that it broadly covers the specification and that we are satisfied with the overall quality. We have also approved the printed version of this book. We do not however check or approve any links or any functionality. Full details of our approval process can be found on our website.

We approve print and digital 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 digital book.

Please note that when teaching the A-level (7517) 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: aqa.org.uk.

Single licence - Abingdon School

Acknowledgements

The author and publisher are grateful to the following for permission to reproduce images, clipart and other copyright material in this book under licence or otherwise:

Chapter 1.1.3

Figure 1.1.3.2 Capstan Shutterstock / 381034510.

Chapter 1.1.8

Figure 1.1.8.1 Rolling dice Shutterstock / 329817896.

Chapter 1.1.16

Figure 1.1.16.4 Billiard balls in frame Shutterstock / 263075000.

Figure 1.1.16.9 Pyramid of balls Shutterstock / 85235152.

Chapter 1.2.2

Figure 1.2.2.2 Head silhouette Shutterstock / 152509136.

Chapter 1.2.3(1)

Figure 1.2.3.1 Stack of books Shutterstock / 976714136.

Chapter 1.2.3(3)

Microsoft® is a registered trademark of Microsoft Corporation.

Chapter 1.2.3(5)

Figure 1.2.3.9 Rolling dice Shutterstock / 329817896.

Chapter 2.1.1

Figure 2.1.1.1 Queue of people in silhouette Shutterstock / 253319245.

Chapter 2.1.2

Thermometer Fotolia 113686553_V.

Chapter 2.1.4

™ Registered trademark of TomTom International.

Figure 2.1.4.1 Queue of people in silhouette Shutterstock / 253319245.

Figure 2.1.4.8 Stack of books Shutterstock / 976714136.

Figure 2.1.4.13 Stack of books Shutterstock / 976714136.

Figure 2.1.4.22 Garden maze iStockphoto iStock_000003304776.

Figure 2.1.4.26 Complex maze iStockphoto iStock_12801988.

Figure 2.1.4.29 Road map showing some Welsh villages/towns (Anquet Technology Ltd © Crown copyright 2016 OS GB map).

Chapter 3.1.1

Figure 3.1.1.7 Garden maze iStockphoto iStock_0000033047765.

Figure 3.1.1.15 Complex maze iStockphoto iStock_12801988.

Chapter 3.5.1

The Art of Computer Programming, Volume 3, Sorting and Searching, © Addison-Wesley 1998.

Chapter 3.6.1

Figure 3.6.1.3 Map of Australia Shutterstock / 33329851.

Single licence - Abingdon School

Chapter 4.1.2

Figure 4.1.2.1 Ball of wool Shutterstock / 59259781.

Figure 4.1.2.1 Woollen pullover Shutterstock / 85713035.

Table 4.1.2.1 Music stave Shutterstock / 85713035.

Figure 4.1.2.2 Capstan Sutterstock / 381034510.

Task 1 - based on an exercise from CSInside Algorithm Development, CS department, Glasgow university.

Chapter 4.1.3

Figure 4.1.3.2 London underground map Reg. User No 16/E/3021/P Pulse Creative Ltd.

Figure 4.1.3.5 Shutterstock / 293938154.

Figure 4.1.3.8 Shutterstock / 236943415 / 269590388 / 403734298 / 49052899.

Chapter 4.1.7

Figure 4.1.7.1 ATM Queue Shutterstock / 276171293.

Figure 4.1.7.2 Queue of people in silhouette Shutterstock / 253319245.

Figure 4.1.7.8 Stack of books Shutterstock / 976714136.

Chapter 4.1.8

Figure 4.1.8.3 Anquet Technology Ltd © Crown copyright 2016 OS 100057707.

Chapter 4.1.10

Figure 4.1.10.1 Lego bricks Shutterstock / 197086964.

Chapter 4.2.3

Figure 4.2.3.1 Pencil with lead hardness HB Shutterstock / 131161022.

Chapter 4.4.3

Figure 4.4.3.4 Meat grinder Shutterstock / 195885296.

Chapter 4.4.5

Kennedy, R. C. (2008) Fat, Fatter, Fattest: Microsoft's Kings of Bloat. InfoWorld Applications, 14 April.

(Used with permission of InfoWorld Copyright 2017. All rights reserved).

Chapter 4.5.1

Figure 4.5.1.1 Turing machine: © GabrielF (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons; https://commons.wikimedia.org/wiki/File%3AModel_of_a_Turing_machine.jpg

Figure 4.5.1.10 Universal machine: © Cbuckley (Own work) [GFDL (http://www.gnu.org/copyleft/fdl.html); https://commons.wikimedia.org/wiki/File%3AUniversal_Turing_machine.svg

The author would like to thank

- Marjory Joan and Edwin Sidney Bond for their unstinting sand devoted support over the years and for laying the foundations that made this book possible
- Sue Poh-Cheng Bond for her patience and constant support during the making of this book.

Single licence - Abingdon School

Single licence - Abingdon School

Contents

Contents

How to use this book xi

Introduction xii

1.1 Programming 1

1.1.1 Data types 1
1.1.2 Programming concepts 16
1.1.3 Arithmetic operations in a programming language 44
1.1.4 Relational operators in a programming language 52
1.1.5 Boolean operations in a programming language 54
1.1.6 Constants and variables in a programming language 57
1.1.7 String-handling operations 59
1.1.8 Random number generation in a programming language 88
1.1.9 Exception handling 93
1.1.10 Subroutines (procedures/functions) 98
1.1.11 Parameters of subroutines 101
1.1.12 Returning a value(s) from a subroutine 105
1.1.13 Local variables in subroutines 109
1.1.14 Global variables in a programming language 112
1.1.15 Role of stack frames in subroutine calls (see Chapter 2.1.4a) 233
1.1.16 Recursive techniques 116

1.2 Programming paradigms 124

1.2.1 Programming paradigms (see Chapters1.2.2 and 1.2.3) 124
1.2.2 Procedural-oriented programming 124
1.2.3(1) Object-oriented programming - introduction 139
1.2.3(2) Object-oriented programming - aggregation and composition 155
1.2.3(3) Object-oriented programming - why object-oriented programming is used 160
1.2.3(4) Object-oriented programming - object-oriented design principles 162
1.2.3(5) Object-oriented programming - writing object-oriented programs 168
1.2.3(6) Object-oriented programming - class diagrams 185

2.1 Data structures and abstract data types 191

2.1.1 Data structures 191
2.1.2 Single- and multi-dimensional arrays (or equivalent) 193
2.1.3 Fields, records and files 216
2.1.4a Abstract data types/data structures 227
2.1.4b Static and dynamic structures 250

2.2 Queues 253

2.2.1 Queues 253

Single licence - Abingdon School

2.3 Stacks 259
2.3.1 Stacks 259
2.4 Graphs 262
2.4.1 Graphs 262
2.5 Trees 271
2.5.1 Trees (including binary trees) 271
2.6 Hash tables 285
2.6.1 Hash tables 285
2.7 Dictionaries 299
2.7.1 Dictionaries 299
2.8 Vectors 300
2.8.1 Vectors 300
3.1 Graph-traversal 304
3.1.1 Simple graph-traversal algorithms 304
3.2 Tree-traversal 312
3.2.1 Simple tree-traversal algorithms 312
3.3 Reverse Polish 318
3.3.1 Reverse Polish - infix transformations 318
3.4 Searching algorithms 322
3.4.1 Linear search 322
3.4.2 Binary search 324
3.4.3 Binary search tree 328
3.5 Sorting algorithms 336
3.5.1 Bubble sort 336
3.5.2 Merge sort 341
3.6 Optimisation algorithms 348
3.6.1 Dijkstra's shortest path algorithm 348
4.1 Abstraction and automation 357
4.1.1 Problem solving 357
4.1.2 Following and writing algorithms 364
4.1.3 Abstraction 373
4.1.4 Information hiding 377
4.1.5 Procedural abstraction 382

Single licence - Abingdon School

4.1.6 Functional abstraction 384
4.1.7 Data abstraction 386
4.1.8 Problem abstraction/reduction 390
4.1.9 Decomposition 395
4.1.10 Composition 396
4.1.11 Automation 400
4.2 Regular languages 401
4.2.1 Finite state machines (FSM) 401
4.2.2 Maths for regular expressions 408
4.2.3 Regular expressions 416
4.2.4 Regular language (see Chapter 4.2.3) 423
4.3 Context-free languages 427
4.3.1 Backus-Naur Form (BNF)/syntax diagrams 427
4.4 Classification of algorithms 435
4.4.1 Comparing algorithms 435
4.4.2 Maths for understanding Big-O notation 445
4.4.3 Order of complexity 453
4.4.4 Limits of computation 460
4.4.5 Classification of algorithmic problems 462
4.4.6 Computable and non-computable problems 465
4.4.7 Halting problem 469
4.5 A model of computation 473
4.5.1 Turing machine 473
Index 484
Glossary - www.educational-computing.co.uk/CS/Unit1/Glossary.pdf
Exam practice questions -
www.educational-computing.co.uk/CS/Unit1/ExamPracticeQuestions.pdf
Exam practice solutions -
www.educational-computing.co.uk/CS/Unit1/ExamPracticeSolutions.pdf

Single licence - Abingdon School

Single licence - Abingdon School

How to use this book

The structure and content of this textbook maps to sections 4.1 to 4.4 of AQA's A Level Computer Science specification (7517). For example, the chapter number 1.1.1 corresponds to specification section 4.1.1.1. The chapter title is Programming: Data types. The chapters in the book do not use the leading 4 as this designates Subject content – A-level in the specification.

Flipped classroom

This textbook has been written with the flipped classroom approach very much in mind. This approach reverses the conventional classroom lesson and homework model of teaching. Instead, chapters in this textbook should be used to prepare for a lesson so that classroom-time can be devoted to exercises, projects, and discussions.

The features in this book include:

Learning objectives

Learning objectives linked to the requirements of the specification are specified at the beginning of each chapter.

Key concept

Concepts that you will need to understand and to be able to define or explain are highlighted in blue and emboldened, e.g. **Integers**. The same concepts appear in the glossary for ease of reference.

Key principle

Principles that you will need to understand and to be able to define or explain are highlighted in blue and emboldened, e.g. **Abstraction**. The same principles appear in the glossary for ease of reference.

Key fact **Key point**

Facts and points that you are useful to know because they aid in understanding concepts and principles are highlighted in blue and emboldened, e.g. **Whole number:** Whole number is another name for an integer number.

Information **Background**

References information that has the potential to assist and contribute to a student's learning, e.g. Read Unit 1 section 4.2.2 for more background on sets and set comprehension. Background knowledge that could also contribute to a student's learning.

Did you know? **Extension Material**

"Did you know?" - interesting facts to enliven learning. "Extension Material" - content that lies beyond the specification.

Task

Activity to deepen understanding and reinforce learning.

Programming tasks

Practical activity involving the use of a programming language to deepen understanding and reinforce learning of concepts and principles.

Questions

Short questions that probe and develop your understanding of concepts and principles as well as creating opportunities to apply and reinforce your knowledge and skills.

Web links for this book

Resources, solutions to questions, errata and the URLs of all websites referenced in this book are recorded at **www.educational-computing.co.uk/aqacs/alevelcs.html**

Educational Computing Services are not responsible for third party content online, there may be some changes to this content that are outside our control. If you find that a Web link doesn't work please email **webadmin@educational-computing.co.uk** with the details and we will endeavour to fix the problem or to provide an alternative. Please also note that these links are not AQA approved.

Single licence - Abingdon School

Introduction

If you are reading this book then you will already have chosen to be a part of an exciting future, for Computer Science is at the heart of an information processing revolution. This revolution applies not just to seeking patterns of meaning in data accumulated on an unprecedented scale by the huge growth in connected computing devices but also the realisation that all forms of life are controlled by genetic codes. Genetic codes are instructions in a procedural information sense which together with the environment that they inhabit control and guide the development of organisms.

Computer scientists concern themselves with

- representations of information in patterns of symbols, known as data or data representations,
- the most appropriate representation for this data
- the procedures in the form of instructions that can transform this data into new forms of information.

The procedures themselves are also a form of information of an instructional kind.

The key process in Computer Science is **abstraction** which means building models which represent aspects of behaviour in the real-world which are of interest. For example, if we wanted to build an automated recommendation system for an online book store, we might choose to record the types of book and number of each type purchased as well as details that identify the respective customer.

Computer Science is not alone in building abstractions, mathematics and the natural sciences also build abstractions but their models only serve to describe and explain whereas Computer Science must, in addition, perform actions on and with the data that has been modelled if it is to solve problems. These actions are described by **algorithms** or step-by-step instructions which form what is called the **automation** stage of problem solving. Whilst it is true that automation of tasks existed before Computer Science, their nature involved concrete, real-world objects, e.g. the Jacquard loom, not informational abstractions such as an online book recommendation system.

So far it has not been necessary to mention digital computers. Digital computers are just the current means by which algorithms can be implemented to execute on data. Both algorithms and the models on which they act need to be **implemented**: algorithms in the form of code or instructions that a digital computer can understand, i.e. a computer program; models in data structures in a programming language.

Unit 1 is largely about the fundamentals of programming, data structures, algorithms and their efficiency, i.e. algorithms to run quickly while taking up the minimal amount of resources (e.g. memory, hard disk, electricity), and the limits of computation. Procedural and object-oriented programming paradigms are explored and experience of programming in each is encouraged through a range of practical tasks.

Looking ahead, Unit 2 covers the fundamentals of computing devices, how data is represented and communicated between devices, the logic gate circuits that enable computing devices to perform operations and to store information. Unit 2 also covers the fundamentals of computer organisation and architecture, the structure and role of the processor, the language of the machine, binary (machine code) and how it is used to program the hardware directly.
In addition, Unit 2 covers the fundamentals of networking, data models for storing structured and unstructured data that can be accessed from networked machines. Structured data is covered first using the relational database model. The limitations of this model are exposed for data that lacks structure and which is too big to fit into a single server. Such data is known as "Big Data". Machine learning techniques are needed to discern patterns in this data and to extract useful information. Big Data also requires a different programming paradigm, functional programming, one that facilitates distributed programming. Unit 2 also explores the consequences of uses of computing.

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Understand the concept of a data type*
- *Understand and use the following appropriately:*
  - *integer*
  - *real/float*
  - *Boolean*
  - *character*
  - *string*
  - *date/time*
  - *pointer/reference*
  - *records (or equivalent)*
  - *arrays (or equivalent)*
- *Define and use user-defined data types based on language-defined (built-in) data types*

Information

A bit is a binary digit which is either 0 or 1.

A bit pattern is just a sequence of bits, e.g. 0100100001101001.

The bit pattern

0100100001101001

in memory box 1 can be interpreted as representing an unsigned integer with decimal value 18537.

The bit pattern

0110100001101111

in memory box 2 can be interpreted as representing an unsigned integer with decimal value 26735.

The bit pattern in memory box 3 in Figure 1.1.2

1011000011011000

can be interpreted as representing an unsigned integer with decimal value 45272.

1.1.1 Data types

Introduction to programming

Any system that computes can be described as executing sequences of actions, with an action being any relevant change in the state of the system.

For example, the following orders or commands change the states of register boxes labelled 1, 2 and memory box labelled 3:

Action 1: FETCH the number in memory box 1 and place in register box 1

Action 2: FETCH the number in memory box 2 and place in register box 2

Action 3: ADD contents of register box 1 to contents of register box 2

Action 4: STORE the answer in memory box 3.

Figure 1.1.1 shows the starting state and the impending actions. *Figure 1.1.2* shows the final state after the above actions have been carried out. See the information panel for help in interpreting the bit patterns as unsigned integers.

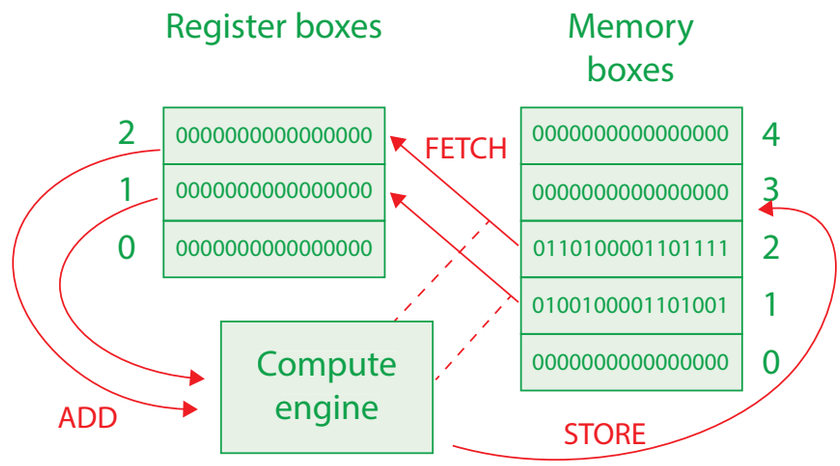


Figure 1.1.1 Starting state of the machine

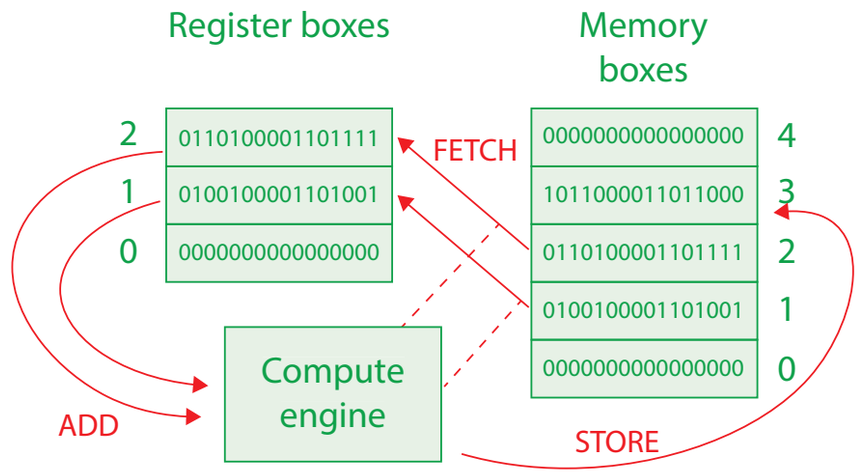


Figure 1.1.2 Final state of the machine

Single licence - Abingdon School

Data types

What is a data type?

Note that in *Figure 1.1.1* and *Figure 1.1.2*, the register and memory boxes contain bit patterns consisting of zeros and ones.

The arithmetic operation ADD performed by the compute engine treats these bit patterns as numbers. The bit patterns have been **data typed** with the meaning number, specifically the number type (positive) **integer**. We say that the **data type** of these bit patterns is (positive) integer.

This isn't the only possible interpretation that can be placed on these bit patterns:

> The text "Hi" and the integer number 18537 can both be represented with the same zeros and ones in a digital computer (0100100001101001).
> One interpretation of this bit pattern is a **string** "Hi" consisting of two characters 'H' and 'i', and another is a **16-bit integer**. See *Chapter 5.5* for the ASCII coding scheme for characters such as 'H'.

Information

Programming languages which enforce data typing - a typed language - are employed because they can prevent an expression from being used which will lead to a data type error when the program is executed, e.g. 4 + 'D'. Data type errors may be detected at compile time or runtime.

Questions

1. What data type do you think the following values belong to
   (a) "Have a nice day" (b) 45

Key term

Sequence:
A sequence is simply an ordered collection of things, e.g. a b c . . . or the sequence of digits in a telephone number, 433004.

What do the bits mean?

This is one role performed by a **data type**: to establish what the bits mean.

Unless we know the meaning the only things that we see in a computer is a datum, i.e. 0s and 1s.

A bit pattern is referred to as a datum.

Formally, a **datum** is a finite sequence of 0s and 1s, e.g. 0100100001101001.

Computations operate on values and produce other values as results,

e.g. 5 + 6 = 11.

The meaning of **value** is a **datum** together with its **meaning** or **interpretation**. Therefore, every value must have a **data type** - datum + meaning.

For example, the data type of *whole number* values such as 5, 6 and 11 is integer.

The following data types are associated with the given example values shown below

- integer, e.g. value 18537
- real number, e.g. value 3.142
- character, e.g. value 'H'
- string, e.g. value "Hi"
- Boolean, e.g. value True

Key point

Meaning of datum:
A datum is a finite sequence of 0s and 1s, e.g. 0100100001101001.

Key point

Meaning of value:
The meaning of value is a datum together with its interpretation.

Single licence - Abingdon School

Specifically, a data type is a correspondence between a mathematical entity such as integer (see *Chapter 5.1.2*) and a set of datums, e.g. all datums of length 16 bits (0000000000000000 to 1111111111111111 in binary or 0 to 65535 in decimal).

Questions

2 Give two possible interpretations of the bit pattern 0100100001101001

What operations can be performed with the data type?

Computations within a digital computer are carried out on data in the form of bit patterns consisting of 0s and 1s.

Another role of data type is **to define which computations/operations may be carried out on the bit patterns**.

For example, if the interpretation of a datum is number then the arithmetic operations ADD, SUBTRACT, MULTIPLY, DIVIDE applied to number values make sense but if the interpretation of a datum is string of characters then arithmetic operations ADD, SUBTRACT, MULTIPLY and DIVIDE don't make sense. ADD would have to be redefined as a string operation called concatenation.

For example, adding one string "Hi" to another "ho" produces the single string "Hiho" which is the bit pattern for "Hi" followed by the bit pattern for "ho." But what does "Hi" divided by "ho" mean?

If the operation is an arithmetic one then adding the number 18537 (0100100001101001), which has the same bit pattern as "Hi", to the number 26735 (0110100001101111), which has the same bit pattern as "ho", produces the expected result: number 45272 (1011000011011000) if we interpret the bit patterns as numbers.

However, the result of adding the number 18537 to the string "ho" is not defined and therefore should be reported as an error.

Enforcing data typing enables outcomes which are undefined to be detected and prevented from occurring.

Questions

3 Is the following addition of two values a valid operation

"two" + 3?

How many bits are allocated to a data type?

When a bit pattern is interpreted as an integer we are mapping this machine integer onto a subset of mathematical integers. A machine integer is a datum whose data type represents some finite subset of the mathematical integers. Mathematical integers are positive and negative whole numbers and zero. However, the set of mathematical integers is infinite but a machine's ability to

Key point

Data type:
A data type is a correspondence between a mathematical entity such as integer (see Unit 2 - Chapter 5.1.2) and a set of datums, e.g. all datums of length 16 bits (0000000000000000 to 1111111111111111 in binary or 0 to 65535 in decimal).
A bit pattern (datum) interpreted as an integer is a mapping from a (digital) machine integer onto a subset of mathematical integers.

Key term

Set:
A set is an unordered collection of different things, e.g. the set of all students in my computer science class: John Smith, Mary Brown, Kamal Patel, Sui Ling, …

Key concept

Data type:
A data type
1. is a method of interpreting a bit pattern
2. defines which computations / operations may be carried out on the bit patterns
3. specifies the number of bits allocated to each datum.

Single licence - Abingdon School

store bit patterns is not. All computing machines have a finite memory. This means that we must specify a finite number of bits for machine integers.

This is the third role of a data type: specifying the number of bits allocated to each datum. A fixed number of bits is therefore allocated to each datum of a data type, e.g. 32 bits.

Questions

4 What is meant by data type?

Integer data type

Signed 32-bit (4-byte) integers range in value from -2,147,483,648 through 2,147,483,647. Values in this range represent mathematical integers exactly. The range is clearly a subset of the range of mathematical integers. Mathematical integers outside this range cannot be represented.

Questions

5 Is $2^{2^{100}}$ in the integer range 0 to 2,147,483,647?

Floating point numbers (real/float data type)

Most computers store and manipulate real numbers such as 3.142 using the IEEE 754 floating point standard.

IEEE floating point numbers can be very large or very small on a scale from the size of galaxies to the size of sub-atomic particles.

Floating-point numbers are little more than scientific notation encoded as bits. Any real number N that can be put in scientific notation can also be put in the form

$$N = s \times m \times 10^{e}$$

where s is −1 or 1, m is a natural number, and e is an integer.

For example, -1.543×10^{-10} represents a real number in scientific form. To put this in the above form, the decimal point is moved 3 places to the right, and 3 is subtracted from the e to get $-1 \times 1543 \times 10^{-13}$.

A number N's significant digits are represented by m either exactly or approximately. Significant digits define the precision of a number.

Digital computers operate with powers of two rather than powers of ten so floating-point numbers that represent real numbers are put in the form

$$N = s \times m \times 2^{e}$$

Key term

Integer data type:
A machine integer is a datum whose data type represents some finite subset of the mathematical integers.
A fixed number of bits are allocated to the datum, e.g. 32 bits.
Signed 32-bit (4-byte) integers range in value from
-2,147,483,648 through
2,147,483,647.
The integer data type is free of rounding errors. Mathematical integers are represented exactly. However, this type has a limited range.

Did you know?

Integer data type:
When the range is exceeded integer types wrap around, e.g. when 1 is added to the integer value 2,147,483,647 it becomes -2,147,483,648 if the data type is signed 32-bit integer.

Key term

Floating point data type:
Computers store and manipulate real numbers such as 3.142 using the IEEE 754 floating point standard (knowledge of its detail is not required for AS or A Level).
Floating-point numbers are little more than scientific notation encoded as bits.
Floating-point numbers that represent real numbers are put in the form

$$N = s \times m \times 2^{e}$$

where s is −1 or 1, m is a natural number, and e is an integer.
A number's significant digits are represented by m either exactly or approximately.
IEEE floating point numbers can be very large or very small on a scale from the size of galaxies to the size of sub-atomic particles.

Single licence - Abingdon School

Floating-point numbers represent the **real numbers of mathematics**. However, floating-point numbers are an approximate representation of these mathematical real numbers because there is

- an infinite number of numbers between any two numbers on the real number line.
- a finite number of values in any floating point data type because we are limited to a finite number of bits for each datum.

Some of the levels of precision (number of bits for each datum[1]) present in IEEE 754 floating point standard are as follows

- 16-bit: Half - half precision
- 32-bit: Single - single precision
- 64-bit: Double - double precision

If single precision (32 bits) is used the

- minimum positive value is $2^{-126} \approx 1.18 \times 10^{-38}$
- maximum positive value is $(1 - 2^{-24}) \times 2^{128} \approx 3.402823 \times 10^{38}$.

All integers with six or fewer significant decimal digits can be converted to a single precision IEEE 754 floating point value without loss of precision. Some integers up to nine significant decimal digits can be converted without loss of precision, but no more than nine significant decimal digits can be stored.
The name **real** or the name **float** is typically used as the data type name for values expressed in floating point form.

Key fact

Significant digits:
123456, 123456000, 0.00123456 all have the same number of significant decimal digits. Significant digits define the precision of a number.

Key term

Real/float:
The name *real* or the name *float* are typically used for the data type name for values expressed in floating point form.

Questions

6 (a) Calculate $(x - y) \times (x + y)$ rounded off to 6 significant digits where x = 293452 and y = 153761 using Windows Calculator or similar.
(b) Calculate x^2 rounded off to 6 significant digits where x = 293452.
(c) Calculate y^2 rounded off to 6 significant digits where y = 153761.
(d) Calculate $x^2 - y^2$ rounded off to 6 significant digits.
If the calculations were done by representing each integer value with a single precision IEEE 754 floating point number, why would the first calculation $(x - y) \times (x + y)$ differ from the second $x^2 - y^2$ even though algebraically $x^2 - y^2 = (x - y) \times (x + y)$?

7 Data types integer (32-bit) and float (32-bit single precision IEEE 754) are available. State which data type could be used for each of the following decimal values without loss of precision and explain why
(a) 45 (b) -2,147,483,648 (c) 3, 124, 560, 000

8 Data types integer (32-bit) and float (32-bit single precision IEEE 754) are available. State which data type could be used for each of the following decimal values and explain why
(a) 3.142 (b) 1.18×10^{-36} (c) 2.46215×10^{38}

1 Precision is determined by how many datum bits are reserved for significant digits, m. One bit is reserved for the sign and several bits are reserved for the exponent, e.

Single licence - Abingdon School

Key term

Boolean data type:
Two values only belong to the data type Boolean:
- the truth value True
- the truth value False.

Boolean data type

The statement "It is raining" is either true or false.
Two values only belong to the data type Boolean:

- the truth value True
- the truth value False.

Only one bit is needed to represent a Boolean truth value but in practice multiple bits are normally used.
Typically, the datum or bit pattern 1111111111111111 is used to represent the truth value True in 16 bits (32 1's in 32 bits, 64 1's in 64 bits and so on).
Typically, the datum or bit pattern 0000000000000000 is used to represent the truth value False in 16 bits (32 1's in 32 bits, 64 1's in 64 bits and so on).

Questions

9 What is the data type of the result of evaluating the following expressions
(a) 5 > 6 (b) 5 < 6 (c) 7 = 7?

Character data type

A character is a letter of the alphabet (upper or lower case) or the digits 0 to 9, or punctuation symbols or some special symbols, e.g. / .

Information

Extended-ASCII:
Extended-ASCII uses 8 bits to encode characters.

A digital machine can only understand sequences of 1's and 0's. Therefore, to denote or represent a character value, e.g. A, inside a computer we must use a bit pattern. One method of allocating bit patterns to characters is the ASCII code system, another is Unicode. ASCII uses seven bits whilst Unicode uses either 16 bits (UTF-16), 32 bits (UTF-32) or one to four bytes (UTF-8) where a byte is eight bits - see *Chapter 5.5*.

The meaning of datum 1000001 is character A if the data type of the datum is **character** (7-bit ASCII). Straight single or double quotation marks are used to indicate a character value, e.g. 'A' or "A".

Questions

10 Shown below are some 7-bit bit patterns along with their decimal equivalents. Use the ASCII look-up table *Table 5.5.1* in *Chapter 5.5* to look up the corresponding character value, i.e. the human readable form.
(a) 1011001 (89) (b) 1001000 (72) (c) 1001100 (76)
(d) 1111001 (121)

11 You may use the ASCII look-up table *Table 5.5.1* in *Chapter 5.5* to help you answer this question.
Do you think that each of the following expressions could be valid, i.e. evaluated successfully? Give your reason for each of your answers.
(a) 'B' > 'A' (b) '5' < '7' (c) '6' = '6'

Single licence - Abingdon School

String data type

Figure 1.1.3 shows beads on a string with each bead labelled with a character value (the space character value corresponds to the unlabelled bead).

Figure 1.1.4 shows beads on a string with each bead labelled with the corresponding character datum (7-bit ASCII), e.g. 1110011 is datum with value 's'.

Borrowing from the beads on a string analogy, a string is just a sequence of characters, e.g. "A string". Straight double or single quotation marks are used to indicate a string value. The corresponding sequence of bit patterns or datum for the string value "A string" is

1000001 0100000 1110011 1110100 1110010 1101001 1101110 1100111

if each character of this string is represented in 7-bit ASCII.

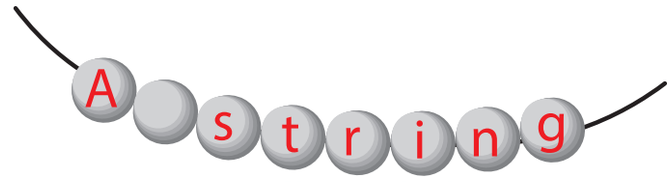


Figure 1.1.3 Character beads on a string

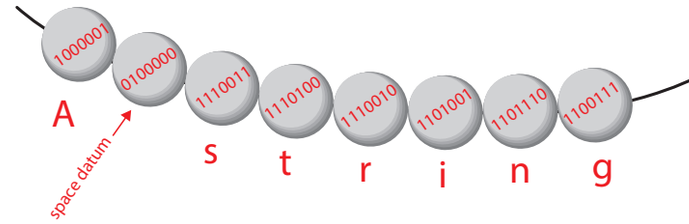


Figure 1.1.4 Character datum beads on a string

Questions

12 Why might a string data type be useful?

Key point

Meaning of value:
Value = datum + its meaning

Datum:
A datum is a finite sequence of 0s and 1s, e.g. 0100100001101001.

Data types Integer, Real/float, Boolean, Character, and String for various programming languages

At the lowest level, as far as computer science is concerned, all data (plural of datum) in digital computers is represented by bits packaged into bytes (8 bits) and words (multiples of 8 bits).

A bit (0 or 1) is a convenient notation that represents which of two observable physical states holds. For example, bit value 1 might mean that the observable output of an electrical circuit making up an element of the computer's memory is 5 volts, and bit value 0 might mean that it is 0 volts. The physical state of this element of the computer's memory is thus either 5 volts or 0 volts.

Information that is stored in a computer system is represented by physical states.

Time and energy are needed to sense (read) or alter (write) these physical states.

This means that a bit is not capable of an existence which can be manipulated until it is recorded in a physical medium where a machine can get at it.

The bits become real objects when recorded in a computer's various memories.

However, any real digital computer has a limited or finite amount of memory in which to record these bits, i.e. it has a finite number of physical states to choose from (the physical states of all memory bits combined). This is the point at which computing deviates from classical mathematics which has no such limitation.

The computer programs that you write in this course will control physical processes involving matter and energy at the level of atoms and electrons but you will rely upon computing abstractions to hide much of this from you (until you hear the fan that keeps a CPU cool switch on).

Programming languages provide abstractions to make life easier for you as a programmer. You have now studied one such abstraction, data typing.

Each programming language that supports explicit data typing has its own names for its **built-in data types**.

For example, C# has a data type with the name **int** which represents a subset of the mathematical integers.

Single licence - Abingdon School

This data type **int** covers the range of integer values from -2,147,483,648 to 2,147,483,647 with each datum in this range represented by 32 bits (physically represented by different energy states of about 10^{24} atoms).
Pascal has a data type with the appropriate name **real** which represents a subset of real numbers (formed from the union of the set of rational numbers and the set of irrational numbers) from mathematics. Python, Java and C# use the name **float** which reflects in the name how real numbers are stored in a digital computer, i.e. as floating point numbers.
Table 1.1.1 shows the names for some of the data types that represent integers, real numbers, truth values, characters and strings in the given languages.

Language	Integer	Real/float	Boolean	Character	String
C#	**int**: 32-bit signed two's complement integer	**float**: 32-bit IEEE 754 floating point **double:** 64-bit IEEE 754 floating point	**bool**: true/false	**char**	**string**
Java	**int**: 32-bit signed two's complement integer	**float**: single-precision 32-bit IEEE 754 floating point **double:** 64-bit IEEE 754 floating point	**boolean**: true/false	**char**: single 16-bit Unicode character	**String**
Pascal	**Integer**	**Real**	**Boolean**: True/False	**Char**	**String**
Delphi	**Integer**: 32-bit two's complement integer -2147483648 to 2147483647	**Double**: 64-bit IEEE 754 floating point supporting approximately 15 decimal digits of precision in a range from 2.23×10^{-308} to 1.79×10^{308} (Real available as well)	**Boolean**: True/False	**Char**: holds a single character in 8 bits. **AnsiChar**: character type guaranteed to be 8 bits in size	**String**
Python	**int**: 32-bit signed two's complement integer	**float**: double-precision 64-bit IEEE 754 floating point	**bool**: True/False	**unicode**	**str**
VB.Net	**Integer**: 32-bit signed two's complement integer	**Double**	**Boolean**: True/False	**Char**: 2 bytes	**String**: 0 to approximately 2 billion Unicode characters (but depends on platform)

Table 1.1.1 Integer, Real/float, Boolean, Character, String data types for the given programming languages

Single licence - Abingdon School

Date/time

Values may be dates, e.g. 28/05/2016, or a time of day, e.g. 15:35:16. Internally, these are just bit patterns. For these bit patterns to have the meaning date or the meaning time of day, the bit patterns must be data typed. This requires a data type **date** and a data type **time** or a combination of the two, a data type **datetime**.

Table 1.1.2 shows the names of these data types[2] in the programming languages, C#, Java, Pascal, Delphi, Python, VB.Net.

Language	
C#	**DateTime**: eight bytes — two four-byte integers. The DateTime value type represents dates and times with values ranging from 00:00:00 (midnight), January 1, 0001 Anno Domini (Common Era) through 11:59:59 P.M., December 31, 9999 A.D. (C.E.) in the Gregorian calendar. Four bytes are allocated to date and four bytes to time. Time values are measured in 100-nanosecond units called ticks, and a particular date is the number of ticks since 12:00 midnight, January 1, 0001 A.D. (C.E.) in the Gregorian Calendar (excluding ticks that would be added by leap seconds).
Java	Java SE8 package java.time **LocalDate**, **LocalTime**, **LocalDateTime**: represent date and time from the context of the observer, e.g. local calendar or clock
Pascal	**TDateTime**: data type double, with the date as the integral part, and time as fractional part. The date is stored as the number of days since 30 December 1899.
Delphi	**TDateTime**: data type double, with the date as the integral part, and time as fractional part. The date is stored as the number of days since 30 December 1899.
Python	**datetime.date:** An idealized naive (not aware of time zone and daylight saving time information) date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day. **datetime.time**: An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of "leap seconds"). Attributes: hour, minute, second, microsecond, and tzinfo.
VB.Net	See C#

Table 1.1.2 Date and Time data types for the given programming languages

Questions

13 What data type could be used for the value of a student's date of birth?

2 C#, VB.Net, Python, Java datetime, date and time data types are actually classes but this requires an understanding of object-oriented programming (OOP) to explain.

Single licence - Abingdon School

Pointer/reference

A pointer is just a memory address.

A pointer type could be declared as follows:

```
Type
  MyPointerType = ^Integer
```

A variable of this type could be declared as follows:

```
Var
  MyPointer : MyPointerType
```

A variable of this type could be used as shown in *Figure 1.1.5*.

In *Figure 1.1.6* the symbolic name `MyPointer` references memory location with address 44382262.

The location at memory address 44382262 contains a reference to another memory location with address 55346782.

It is this address which is obtained when the pointer variable, `MyPointer` is dereferenced.

The dereferencing is done as follows

```
MyPointer^
```

`MyPointer^ ← 6` assigns 6 to the memory location with address 55346782 as shown in *Figure 1.1.6*.

The memory at 55346782 must be grabbed from the heap which is the area of memory that programs may use for this purpose.

In the code in *Figure 1.1.5* this is done with

```
New MyPointer
```

The area of memory occupied by the program is shown in grey in *Figure 1.1.6*.

The memory at 55346782 is grabbed after the program starts executing.

Pointers are of fixed length, e.g. only 32 or 64 bits whereas the "chunk" of memory that they reference could be much larger, e.g. 64 KB.

A copy of a pointer therefore consumes much less memory than a copy of the "chunk" of memory. They are useful for this reason.

```
New MyPointer {Claims storage space from the
                    heap(RAM) for an integer}
              {Places address of this storage
                         location in MyPointer}
MyPointer^ ← 6 {Assigns the integer value 6 to
                         this storage location}
Output MyPointer^ {Displays the stored integer
                                     value 6}
DisposeMyPointer {Frees storage space claimed
                                      by New}
MyPointer ← Nil {Sets pointer to Nil which is
                     the null pointer value}
                  {A null pointer points
                                     nowhere}
```

Figure 1.1.5 Use of a pointer

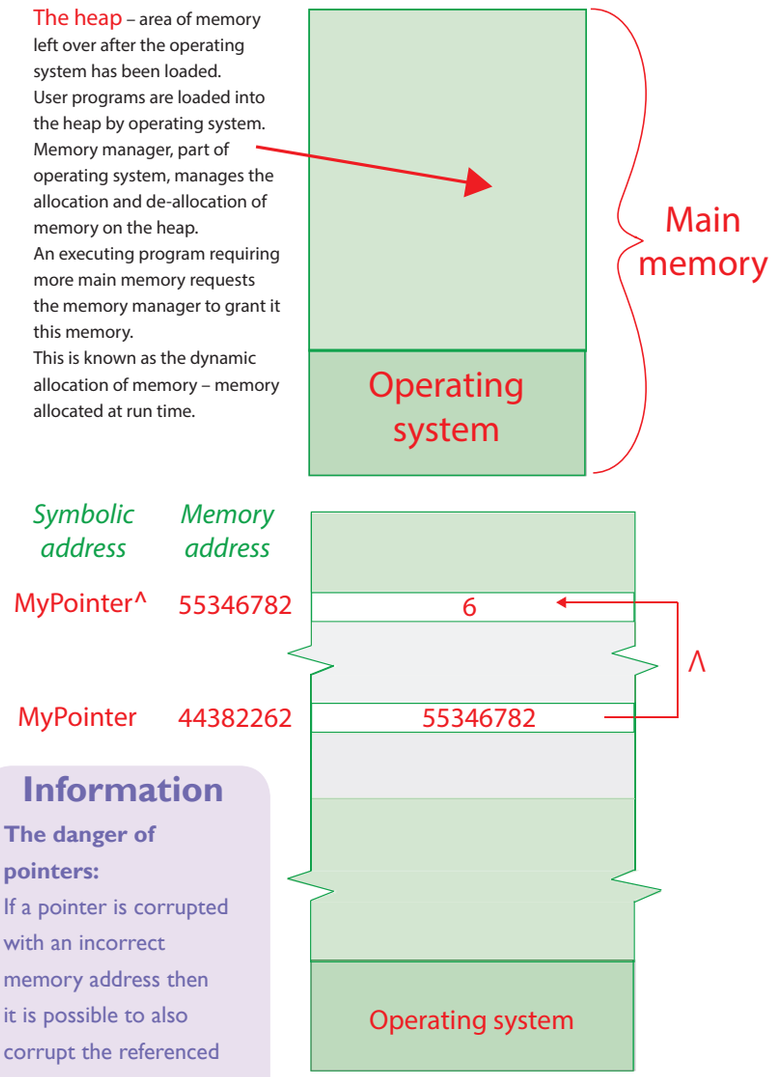


Figure 1.1.6 Area of memory used by pointer

Information

The danger of pointers:

If a pointer is corrupted with an incorrect memory address then it is possible to also corrupt the referenced memory.

Questions

14 What is meant by a pointer?

15 Explain using an example how a pointer variable could be used to reference a memory location so that an integer value may be stored in this location.

Single licence - Abingdon School

Records

Figure 1.1.7 shows a card index consisting of cards on which information has been recorded. Each card contains a record of something. In *Figure 1.1.7*, it is a **record** of a person's name, address, latitude, longitude, no of rooms in their dwelling, whether the dwelling has loft insulation or not, and Council Tax band. It is intended that the card shown will be filed in the section of the card index labelled 'B'. We can say that the card index contains a file of records ordered by initial letter of surname. Notice that the information on the card in *Figure 1.1.7* is subdivided into an area for the surname, an area for the forename, and so on. Each area is called a **field of the record**. Each **field has a name**, e.g. surname, and a value, e.g. BOND.

We have learned that every value has a data type. The data type of each value recorded in the given record is as follows

BOND *String*
KEVIN *String*
THE OAK TREE *String*
THE OAK FOREST *String*
DINGLEY DELL *String*
TR33 OAK *String*
NEVER NEVER LAND *String*
51.410519 *Real*
-1.715651 *Real*
6 *Integer*
TRUE *Boolean*
D *Character*

Figure 1.1.7 Card index showing one record

To record the information about the person with surname BOND in a machine, the human readable values of the record are replaced with their data type equivalent bit patterns. At the machine level the information becomes a sequence of bits as follows

1000010100111110011101000100100101110001011101110100100110011 10 . . .

SURNAME *FORENAME*

If we view the entire bit pattern for the person's record and use the "**method of interpreting a bit pattern definition of data type**", we now have a new data type which has **structure** consisting of **named fields** and their **corresponding data types**. This new data type is called a **record data type** - see *Figure 1.1.8*. This record data type consists of 12 fields.

Key term

Record data type:
A record data type is a data type composed of related data of various data types.
A record data type is divided into named fields.

Single licence - Abingdon School

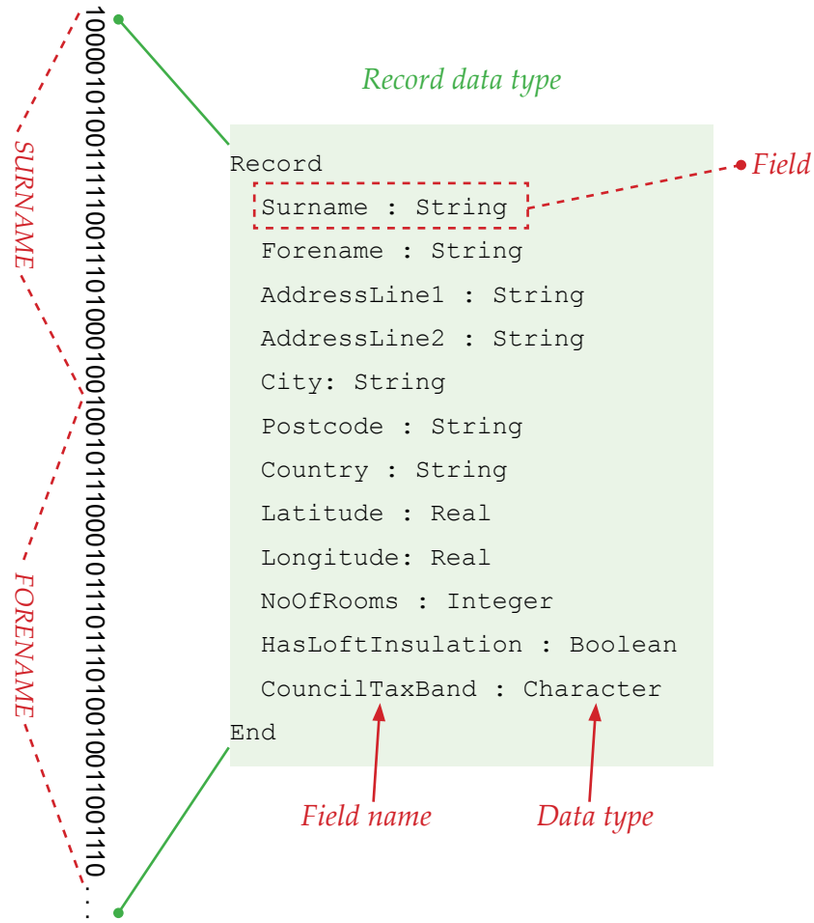


Figure 1.1.8 Record data type which imposes meaning to bit pattern consisting of values surname, forename, etc.

The record data type is a data type in which the constituent parts of a bit pattern may have different meanings, e.g. string, integer, real, etc.

The higher level view is:

a record data type is a data type composed of related data of various data types.

Table 1.1.3 shows how record data types can be defined in the programming languages Pascal/Delphi, C#, Python, Java, and VB.Net.

Questions

16 Using the data types integer, real, character, Boolean, and string, define a record data type which will enable the following details of a book to be recorded:
Title, ISBN, whether in stock or not, number of pages, price, category which may be fiction (F) or nonfiction (N).

Single licence - Abingdon School

Pascal/Delphi	C#

`TExam = Record`
`  Name : String;`
`  ExamScore : Integer;`
`  ExamGrade : Char;`
`  Resit : Boolean;`
`  MeanGCSEScore : Real;`
`  End;`
	`struct Exam`
	`{`
	`  public string name;`
	`  public int ExamScore;`
	`  public char ExamGrade;`
	`  public bool Resit;`
	`  public float MeanGCSEScore;`
	`};`

Python	**Java**

`class exam(object):`
`  def __init__(name,examscore,examgrade,resit,meangcsescore)`
`  self.name = name`
`  self.examscore = examscore`
`  self.examgrade = examgrade`
`  self.resit = resit`
`  self.meangcsescore = meangcsescore`
	`class Exam {`
	`  String name;`
	`  int examScore;`
	`  char examGrade;`
	`  boolean resit;`
	`  float meanGCSEScore;`
	`}`

VB.Net	

`Structure exam`
`  Public name As String`
`  Public examScore As Integer`
`  Public examGrade As Char`
`  Public resit As Boolean`
`  Public meanGCSEScore As Double`
`End Structure`	

Table 1.1.3 Defining the record data type in various programming languages

Arrays

When a number of similar data items are put alongside each other and treated as a unit the result is an **array data structure**. For example, temperature readings recorded over a twelve hour period as shown in *Figure 1.1.9*.

The individual values shown in *Figure 1.1.9* are all the same data type **integer**. As a unit the data type is **array of**

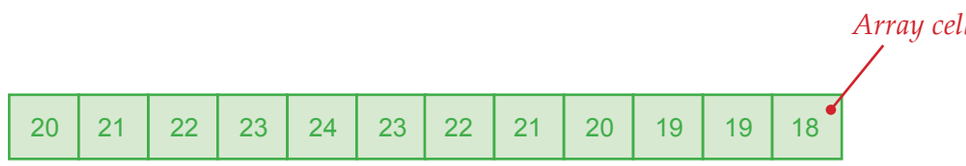


Figure 1.1.9 Temperature readings gathered and recorded over a 12 hour period

integer. The unit shown has 12 cells for 12 integer values, one per cell. The data type of the value in each cell is integer. An **array data type** is a (fixed-size if static) **sequential collection of elements** of the **same type**.

In Pascal/Delphi this data type could be defined as follows

```
TArrayOfTemperature = Array[0..11] Of Integer;
```


Single licence - Abingdon School

`Array[0..11]` means that the array has 12 cells labelled 0, 1, 2, 3, . . ., 9, 10, 11. Cell labelling is important because it is the mechanism by which the value in a particular cell is accessed. The name **array index** is used for a cell label.

The cell data type in the example is integer but cell data types can be any data type supported by the programming language. The only requirement is that every cell must be of the same data type.

Key term

Array data type:
An array data type is a (fixed-size if static) sequential collection of elements of the same type.

Table 1.1.4[3] shows examples of how an array is defined in Pascal/Delphi, C#, Python, Java, and VB.Net. The Python example uses a package called numpy. Numpy is a package used for scientific computing with Python.

Pascal/Delphi	C#
`TArrayOfTemperature = Array[0..11] Of Integer;` `ArrayOfTemperature : TArrayOfTemperature;`	`int [] arrayOfTemperature = new int[12];`
VB.Net	**Java**
`Dim ArrayOfTemperature(12) As Integer`	`int [12] arrayOfTemperature;`
Python	

`import numpy as np`
`>>> arrayoftemperature = np.linspace(0, 11, num=12, dtype=int)`
`>>> arrayoftemperature`
`array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])`	

Table 1.1.4 Defining an array in various programming languages

Questions

17 What is meant by an array data type?

18 Define an array data type that has 5 cells labelled 0, 1, 2, 3, 4 with cell data type Real. Use the syntax shown on page 13.

Built-in data types

Data types which are already predefined in the programming are called **built-in data types**.
For example, the following are the names of some built-in data types for the programming language Pascal:

Integer, Real, Boolean, Char, String.

User-defined data types

The built-in data types are often not enough to do what a programmer wants to do. Programming languages allow programmers to build their own data types. These are called **user-defined data types**.
For example, in Pascal/Delphi an integer subrange data type may be defined as follows

```
Type
  MyFirstSubRangeType = 0..9;
```

Values of this type are restricted to integers in the range 0 to 9.

3 In both Python and C# examples a variable of the desired array type is created

Single licence - Abingdon School

Programming languages that support user-defined data types can prevent expressions with such data typed values from being used if a value occurs outside the permitted range of values when the program is compiled/executed.

The record and array data types that have been covered already are also examples of user-defined data types.

Questions

19 What is meant by a user-defined data type?

In this chapter you have covered:

- the concept of a data type
- the following data types:
  - integer
  - real/float
  - Boolean
  - character
  - string
  - date/time
  - pointer/reference
  - records (or equivalent)
  - arrays (or equivalent)
- user-defined data types based on language-defined (built-in) data types

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Use, understand and know how the following statement types can be combined in programs:*
  - *variable declaration*
  - *constant declaration*
  - *assignment*
  - *iteration*
  - *selection*
  - *subroutine(procedure/function*
- *Use definite and indefinite iteration, including indefinite iteration with the condition at the start or end of the iterative structure. A theoretical understanding of condition(s) at either end of an iterative structure is required, regardless of whether they are supported by the language being used*
- *Use nested selection and nested iteration structures*
- *Use meaningful identifier names and know why it is important to use them.*

Key concept

Variable:
In programming, a variable can be thought of as a container for a value.
Like a physical container a variable may be empty, in which case we say that its value is undefined.

1.1.2 Programming concepts

Variable declaration

Variable

Data which are subject to change are modelled in a program by variables.
A **variable** is simply a container in which a value may be stored for subsequent manipulation within a program - *Figure 1.1.2.1*.

The stored value may be changed by the program as it executes its instructions, hence the use of the term "variable" for the container.

Under the bonnet, the container is realised as one or more reserved memory locations in the RAM of the machine.

Figure 1.1.2.1 shows the value 6 about to be stored in variable x represented by the yellow container

Variable declaration

A **variable declaration** is one way of causing a variable to be created.

For example, in Pascal/Delphi, an integer variable, `x`, may be declared as follows

```
Var
  x : Integer;
```

The amount of reserved memory allocated to a variable and what is allowed to be stored in this reserved memory depend upon its data type.

This declaration reserves a **named space** in memory (RAM) for a value of data type `Integer`. The space is named `x` and the amount of RAM is four bytes because that is what Pascal's `Integer` data type specifies[1].

By assigning different data types to each variable, *integers*, *numbers with a fractional part*, *characters*, *strings*, *arrays*, *records* and *other entities*, may be stored in these variables.

In some languages, e.g. Python, variables do not need an explicit declaration for memory space to be reserved. The declaration happens automatically when a value is assigned to a variable. Values in Python are strongly typed but variables are not.

1 It is implementation dependent but four bytes is typical.

Single licence - Abingdon School

Key concept

Variable declaration:
A variable declaration is one way of causing a variable to be created.

Key point

Modelling data by variables:
Data which are subject to change are modelled by variables.

Key concept

Constant:
In programming, data which never change are modelled in a program by constants. This means stating their values explicitly.
To make a program easier to read, understand and maintain, a constant is given a symbolic name which can be used throughout the program whenever the value of the constant is required.

Questions

1. What is a variable?
2. What is a variable declaration?
3. What is a constant declaration?

Initialising variables

It is common practice to initialise a variable when it is declared.
For example, in Pascal/Delphi

```
Var
  x : Integer = 6;
```

In C#, an integer variable, `x`, may be declared and initialised as follows

```
int x = 6;
```

In Java, an integer variable, `x`, may be declared and initialised as follows

```
int x = 6;
```

In VB.Net, an integer variable, `x`, may be declared and initialised as follows

```
Dim x As Integer = 6
```

Constant declaration

Some of the data used in programs never change. For example, the ratio of the circumference of a circle to its diameter which is approximately 3.142.

Data which never change are modelled in a program by constants.

This means stating their values explicitly. To make a program easier to read, understand and change, a constant is given a **symbolic name** which can be used throughout the program whenever the value of the constant is required.

For example, in Pascal/Delphi a constant for the value 3.142 can be defined using the language keyword `Const` as follows

Language keyword → `Const`

```
Pi = 3.1.42;
```

Symbolic name for constant

In C# the symbolic name by convention is in uppercase and can use underscores for spaces, e.g. `NO_OF_DAYS_IN_WEEK`. The following is an example of a constant definition in C#

```
(public/private) const float PI = 3.142f;
```

or

```
(public/private) const double PI = 3.142;
```

The data type `double` is a double-precision 64-bit IEEE 754 floating point number. The data type `float` is a single-precision 32-bit IEEE 754 floating point number. In the case of `float` the letter `f` must be appended to the value.

In Java the symbolic name by convention is in uppercase and can use underscores for spaces. The following is an example of a constant definition in Java

```
(public/private)static final float PI = 3.142f;
```

or

```
(public/private) static final double PI = 3.142;
```


Single licence - Abingdon School

The data type `double` is a double-precision 64-bit IEEE 754 floating point number. The data type `float` is a single-precision 32-bit IEEE 754 floating point number. In the case of `float` the letter `f` must be appended to the value.

In VB.Net

```
Const pi As Double = 3.142
```

Python has no way to declare constants. Variables with symbolic names written in uppercase letters and with underscores for spaces signify that their values will be used as constants but the language cannot prevent the values from being changed. It is only a convention.

Questions

4 The calculation 2 x 3.141592654 x radius is made in several places in a program. Give **two** reasons why replacing 3.141592654 by the named constant `pi` could improve this program?

Figure 1.1.2.2 shows the value 6 about to be stored in variable x which currently contains the value 4

Assignment

An instruction which alters the value of a variable is called an **assignment statement** and the operation is called **assignment**. A value is copied into a variable when a computer executes an assignment statement. The action replaces the currently stored value in the variable as illustrated in *Figures 1.1.2.2* and *1.1.2.3* for a variable `x`.

For example, in Pascal/Delphi an assignment statement that **assigns** the value `6` to variable `x` is written as follows

```
x := 6;
```

The operand, `x`, to the left of the `:=` operator is the name of the variable and the operand, `6`, to the right of the `:=` operator is the value that will be stored in the variable when assignment is carried out. The value in `x` before assignment occurs is overwritten by the assignment operation. This is shown by example in *Figure 1.1.2.3* where the value 4 is overwritten by the value 6.

Figure 1.1.2.3 shows that value 4 has been replaced by value 6

The `:=` operator is called the **assignment operator**. In pseudo-code, a left-pointing arrow ← is used to represent the assignment operator.

The term "statement" is used universally for historical reasons. It would have been better to have adopted the term "command" or "instruction".

Examples of assignment statements and assignment operators in other programming languages and in pseudo-code are shown below.

In C#,	`x = 6;`
In Java,	`x = 6;`
In Python,	`x = 6`
In VB.Net,	`x = 6`
In pseudo-code,	`x ← 6`

Key term

Assignment statement:
An instruction which alters the value of a variable is called an assignment statement.
The operation is known as assignment.

Questions

5 What is assignment ?

Single licence - Abingdon School

Pseudo-code

Pseudo-code is a programming-like language which is employed to communicate a solution to a problem in a manner which is independent of any particular production programming language such as Pascal, Delphi, C#, Java, Python or VB.Net. The given pseudo-code solution may then be recast in any one of these programming languages, if necessary.

You have seen that Pascal/Delphi uses := for the language construct known as the assignment operator whilst = is used in C#, Java, Python and VB.Net. Pseudo-code provides a common way of expressing programming language constructs, e.g. the assignment operator, which are independent of any actual programming language.

However, it is not usually possible to execute pseudo-code in a machine. If it were, it would become a fully-fledged programming language.

Instead, pseudo-code is executed by hand which means reading it and processing it statement-by-statement in your head. The process is called a **trace** or **hand-trace** execution.

Which authority defines pseudo-code?

You can design and use your own pseudo-code.

AQA uses pseudo-code for its AS and A level Computer Science examinations.

Generally speaking, one person's pseudo-code may differ in some respects from another's but usually the differences are not sufficient to make reading and understanding difficult.

Information

Pseudo-code:
Pseudo-code is a programming-like language which is employed to communicate a solution to a problem in a manner which is independent of any particular production programming language.

Iteration

Suppose we are required to sum the first 10 natural numbers, i.e.

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

One way that this could be done is to **repeat** the addition of each natural number in turn to a running total, initialised to 0, **until** all 10 natural numbers have been added together.

In pseudo-code this might be expressed as shown in *Table 1.1.2.1*.

```
RunningTotal ← 0      Assign the value 0 to variable RunningTotal
NatNo ← 1             Assign the value 1 to variable NatNo
Repeat
  RunningTotal ← RunningTotal + NatNo
  NatNo ← NatNo + 1
Until NatNo = 11
```

Add 1 to current value in NatNo, replace this current value with the value which results from this addition

Add NatNo to current value in RunningTotal, replace this current value with the value which results from the addition

Stop repeating when value in NatNo reaches 11

Table 1.1.2.1 Pseudo-code example which adds the first 10 natural numbers

Single licence - Abingdon School

Figures 1.1.2.4 and *1.1.2.5* show the execution of the assignment statement

```
NatNo ← NatNo + 1
```

split over two stages. `NatNo` contains the value 6 just before this statement is executed and afterwards it contains the value 7.

The pseudo-code in *Table 1.1.2.1* illustrates repetition or **iteration**. The general form of such iteration is

```
Repeat
  instructions
Until condition
```

which means that the instructions between the words `Repeat` and `Until` are executed repeatedly until the condition specified in `Until condition` is met.

The instructions between `Repeat` and `Until` are known as the **loop body**. The iteration performed by `Repeat Until` is called a **loop** because execution loops back and forth between `Repeat` and `Until`.
The condition occurring in `Until condition` is called the **terminating condition** of the loop.

Figure 1.1.2.4 shows the value 6 stored in variable NatNo, being copied then added to 1 and the result 7 about to be stored

Figure 1.1.2.5 shows the value 7 stored in variable NatNo

Questions

6 Trace the execution of the pseudo-code shown in *Table 1.1.2.1*. Record in *Table 1.1.2.2* the value of `NatNo` and `RunningTotal` as they change during the execution. The first value of each variable has been recorded for you in the table.

NatNo	RunningTotal
1	0

Table 1.1.2.2 Trace table for Count and RunningTotal

The final value of `RunningTotal` should be 55.

7 How would you change the pseudo-code in *Table 1.1.2.1* so that when executed, it adds the first 15 natural numbers?
Check that your change is correct by hand-tracing the resulting pseudo-code. The final value of `RunningTotal` should be 120.

Key term

Iteration statements: Instructions are executed repeatedly until the terminating condition is met.

Single licence - Abingdon School

Table 1.1.2.3 shows another way to add the first 10 natural numbers.

```
RunningTotal ← 0
NatNo ← 1
While NatNo < 11        Stop repeating when value in NatNo reaches 11
  RunningTotal ← RunningTotal + NatNo
  NatNo ← NatNo + 1
EndWhile
```

Table 1.1.2.3 Pseudo-code example which adds the first 10 natural numbers

Questions

8 Trace the execution of the pseudo-code shown in *Table 1.1.2.3*. Record in *Table 1.1.2.4* the value of `NatNo` and `RunningTotal` as they change during the execution. The first value of each variable has been recorded for you in the table.

NatNo	RunningTotal
1	0

Table 1.1.2.4 Trace table for NatNo and RunningTotal

9 How would you change the pseudo-code in *Table 1.1.2.3* so that when executed it adds the first 15 natural numbers?
Check that your change is correct by hand-tracing the resulting pseudo-code. The final value of `RunningTotal` should be 120.

10 What is the final value of `RunningTotal` when the following pseudo-code is executed by hand if x in `RunningTotal x Choice` is the multiplication operator?

```
RunningTotal ← 1
Count ← 2
Repeat
  RunningTotal ← RunningTotal x Count
  Count ← Count + 1
Until Count = 11
```


Single licence - Abingdon School

Sequence

The following two instructions (assignment statements) of the pseudo-code in *Table 1.1.2.3* are executed one after the other

```
RunningTotal ← RunningTotal + NatNo

NatNo ← NatNo + 1
```

They form a **sequence** of steps governed by the following rules

1. The steps are executed one at a time
2. The order in which the steps are written is the order in which they are executed
3. Each step is executed exactly once.

We say that this part of the solution to the problem of adding the first 10 natural numbers is constructed using a program design principle called **sequence**.

Key term

Sequence statements: Statements are executed one at a time, one after another in the order in which they are written with each executed exactly once.

Questions

11 A variable `x` contains the value 7 and a variable `y` the value 3.
Using only sequence, write assignment statements that swap the values contained in `x` and `y`. You may use `Temp`, a third variable.

Selection

If statement

If solutions to problems consist only of steps in sequence then there is no possibility of solving problems which need to deviate from the given sequence of execution if circumstances require this.

For example, suppose instead of summing the first 10 natural numbers we are required to sum only the even numbers amongst the first 10 natural numbers, using the pseudo-code in *Table 1.1.2.3*. The solution must test to see if the next natural number is even. Only if it is even will it then be added it to the running total.

The revised sequence section of the pseudo-code solution in *Table1.1.2.3* will take the form

```
If NatNo is even                                   condition
  Then RunningTotal ← RunningTotal + NatNo         step
EndIf

NatNo ← NatNo + 1
```

The conditional part of this pseudo-code fits the general form

```
If condition
  Then step
```

where `condition` specifies the circumstance under which the `step` is to be executed. If `condition` is true then `step` is executed; otherwise it is not.

`If condition Then step` is a statement known as **selection**.

Suppose instead of summing the first 10 natural numbers we are required to sum the even numbers in one running total and the odd numbers in a different running total. We now have two alternative steps to be executed.

The revised sequence section of the pseudo-code solution in *Table 1.1.2.3* will take the form

```
If NatNo is even                                   condition
  Then EvenRunningTotal ← EvenRunningTotal + NatNo   step 1
  Else OddRunningTotal ← OddRunningTotal + NatNo     step 2
EndIf
NatNo ← NatNo + 1
```


Single licence - Abingdon School

The general form of selection in which selection is between one of two alternative steps is

```
If condition
  Then step 1
  Else step 2
```

where `condition` determines whether `step 1` is executed or `step 2` is executed. We thus have two forms of `If` selection statement:

```
If condition
  Then step
```

```
If condition
  Then step 1
  Else step 2
```

Key term

Selection:
A selection statement takes one of two possible forms:

```
If condition
  Then step
```

or

```
If condition
  Then step 1
  Else step 2
```

The condition determines whether or not `step` is executed, `step1` or `step2` is executed.

Key fact

Combining principles:
The three combining principles (sequence, iteration/repetition and selection/choice) are basic to all imperative programming languages.

Questions

12 Rewrite the following If statements as a single If Then Else statement

```
If Age < 37 Then AgeCategory ← 'A'
If Age >=37 Then AgeCategory ← 'B'
```

13 The pseudocode in *Table 1.1.2.5* finds the largest of two numbers, `No1`, `No2`.
The value of the largest of the two numbers is stored in variable `Largest`.
(a) If `No1` stores the value 6, `No2` the value 3 which branch of the selection statement is executed, `Then` or `Else`? Explain your answer.
(b) If `No1` stores the value 6, `No2` the value 7 which branch of the selection statement is executed, `Then` or `Else`? Explain your answer.

```
If No1 > No2
  Then Largest ← No1
  Else Largest ← No2
EndIf
```

Table 1.1.2.5 Pseudo-code to find the largest of two numbers

Case/Switch statement

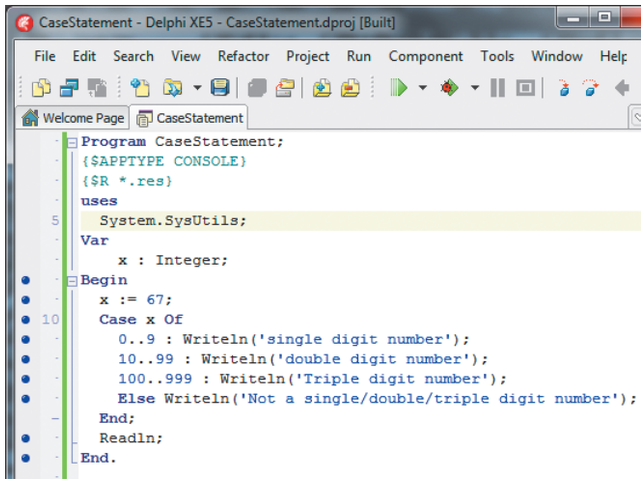
The **Case/Switch statement** is used to select one of several alternatives. In *Table 1.1.2.6* if `x` is in the range `0..9` then the statement `Output 'Single digit number'` is executed and the remaining statements in the *Case* statement are skipped. If `x` is in the range `10..99` then the statement `Output 'Double digit number'` is executed and the other statements in the *Case* statement are skipped. The *Case* statement tries to match the value of `x` with *Case* list `0..9`, `10..99`, `100..999`. If it fails then it executes the statement selected by `Else`, i.e. `Output 'Not a single/double/triple digit number'`. The variable `x` is known as the **case selector**.

```
Case x Of
  0..9 : Output 'Single digit number'
  10..99 : Output 'Double digit number'
  100..999 : Output 'Triple digit number'
  Else Output 'Not a single/double/triple digit number'
EndCase
```

Table 1.1.2.6 Pseudo-code Case statement

Single licence - Abingdon School

The Case/Switch statement is shown in *Figures 1.1.2.6* to *1.1.2.9* for Delphi/Pascal, C#, Java and VB.NET.



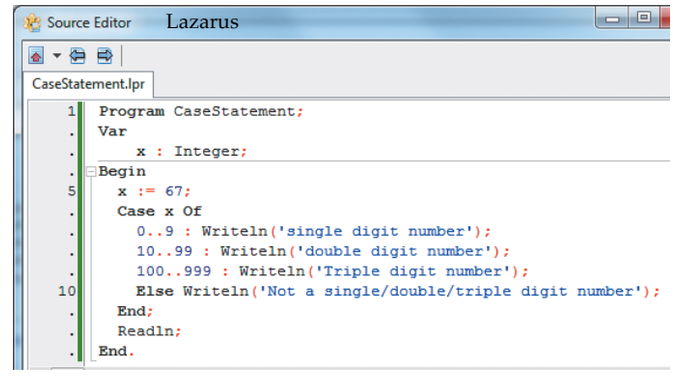


Figure 1.1.2.6 Case statement in Delphi and Lazarus Pascal

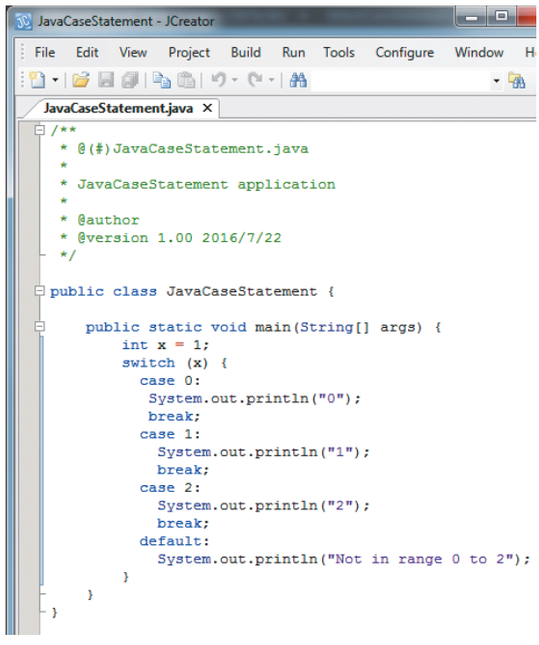


Figure 1.1.2.7 Switch Case statement in Java

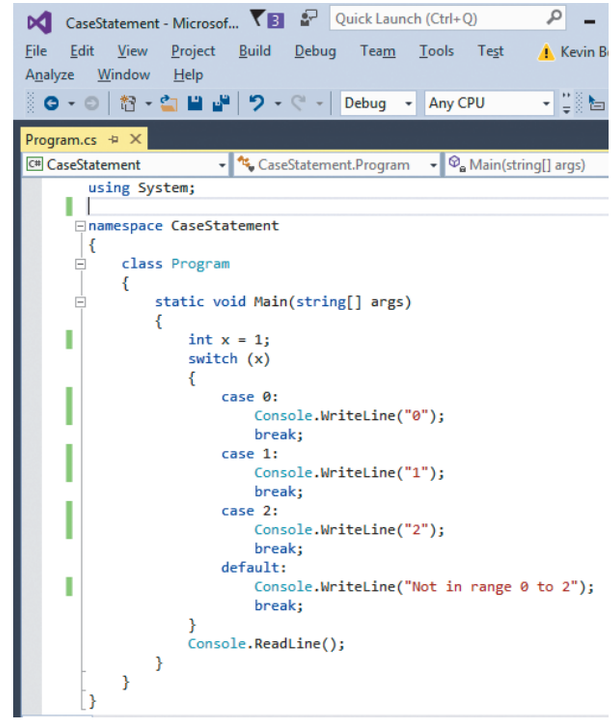


Figure 1.1.2.8 Switch Case statement in C#

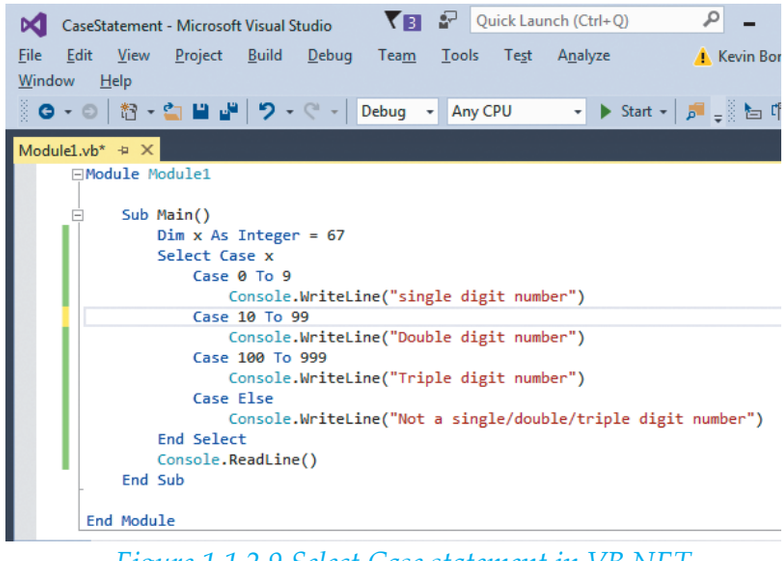


Figure 1.1.2.9 Select Case statement in VB.NET

Java and C# do not support ranges in their Switch Case statement alternatives. The chosen value for x in the Java and C# examples is different from the other languages for this reason.

There is no Case/Switch statement in Python so a combination of *if* and *elifs* have to be used instead.

Single licence - Abingdon School

Subroutine

A subroutine is a **named self-contained block of instructions**, e.g. `drawsquare`.

By encapsulating and naming a block of instructions in a program it becomes possible to call the block from other parts of the program. This is very useful in situations where the same block of instructions or action or calculation needs to be repeated in multiple places in a program.

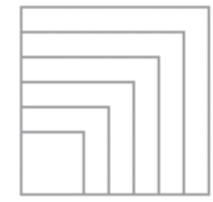


Figure 1.1.2.10 Pattern of squares

Suppose that we wish to draw the pattern containing six squares shown in *Figure 1.1.2.10*. The size (side-length) of the squares increases by the same fixed amount. One way of drawing a square with length of side, `size`, is to use a pen-carrying turtle moving under the guidance of the sequence of instructions or commands shown in *Figure 1.1.2.11*.

```
turtle.forward(size)
turtle.left(90)
turtle.forward(size)
turtle.left(90)
turtle.forward(size)
turtle.left(90)
turtle.forward(size)
turtle.left(90)
```

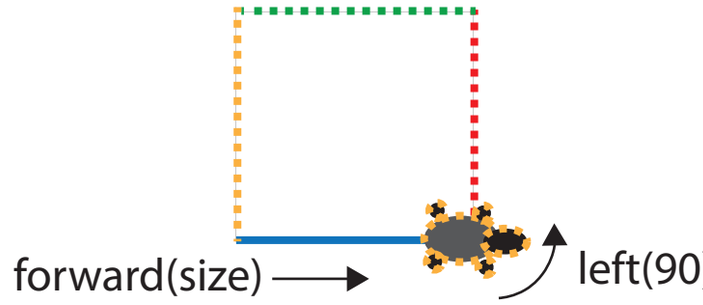


Figure 1.1.2.11 Path followed by a turtle obeying the sequence of commands shown opposite

It is good practice to use active verbs for subroutine names, e.g. Read, Write, Add, DoSomething.

If this sequence of instructions in *Figure 1.1.2.11* is named `drawsquare` then a program could call upon this subroutine to draw a square of a specific size, say, 25, as follows: `drawsquare(25)`

A program that calls upon the subroutine `drawsquare` is said to **call the subroutine** `drawsquare`.

To draw a differently-sized square of size, say, 35, `drawsquare` would need to be called upon as follows

```
drawsquare(35)
```

The square-drawing turtle subroutine itself could be written as shown in *Table 1.1.2.7*.

The variable `size` is called the **formal parameter of the subroutine**: it is used in the **body of the subroutine** to define how big the square will be.

When the subroutine is called, the **actual parameter** 25 or 35, for example, provides a specific value for `size`, and thus determines how big the square will be in each call.

```
subroutine drawsquare(size)
  turtle.forward(size)
  turtle.left(90)
  turtle.forward(size)
  turtle.left(90)
  turtle.forward(size)
  turtle.left(90)
  turtle.forward(size)
  turtle.left(90)
endsubroutine
```

Body

Table 1.1.2.7 Subroutine drawsquare(size) defined

The subroutine `drawsquare` itself calls subroutines `turtle.forward` and `turtle.left`.

Single licence - Abingdon School

```
drawsquare(25)
drawsquare(35)
drawsquare(45)
drawsquare(55)
drawsquare(65)
drawsquare(75)
```

Table 1.1.2.8 Calling subroutine drawsquare(size) with different actual parameters

To draw the pattern of squares in *Figure 1.1.2.10* we need the sequence of calls to subroutine `drawsquare` shown in *Table 1.1.2.8*.

Figure 1.1.2.12 shows how the `drawsquare` subroutine could be defined in Python 3.4 and how it would be called to produce the pattern of squares shown in *Figure 1.1.2.10*.

A subroutine may also contain its own *variable*, *type*, *label*[2] and *const* declarations. In fact, it may also define other subroutines within its declaration section.

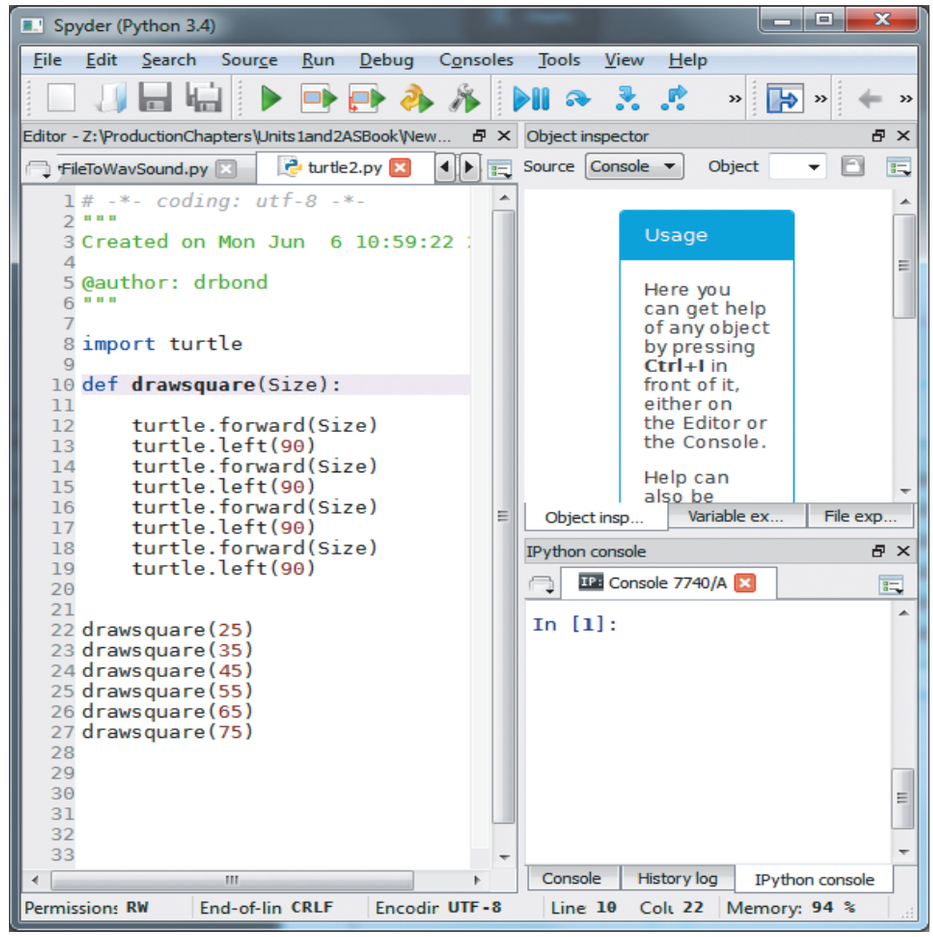


Figure 1.1.2.12 Python 3.4 program to draw a pattern of squares

Key term

Subroutine:
A subroutine is a named self-contained block of instructions, e.g. `drawsquare`, which may be called (i.e. executed) from anywhere within a program where its name appears.

Procedures and functions

There are two types of subroutines:

- **Procedure**
- **Function**

A procedure is a **subroutine consisting of one or more statements or actions which may or may not return a result**. If it returns a result it does so through output parameters in its interface - see page 101. The statements are referred to collectively by a name assigned to the procedure called the **procedure name**. The procedure must be declared/defined – its name must be stated and its statements listed. The procedure's statements are executed wherever its name is encountered in the executable part of a program - see *Figure 1.1.2.12*. This is called **calling a procedure**.

Key term

Procedure:
A procedure is a subroutine consisting of one or more statements or actions which may or may not return a result.

2 Not in AQA specification

Single licence - Abingdon School

In Pascal, a procedure is declared and used as follows:

```
Program ProcedureExample;
Var
  Ch : Char;
Procedure DoExample;
  Begin
    Writeln('The sun ');
    Writeln('has got its hat on, ');
    Writeln('hip hip hooray.');
  End;

Begin
  Repeat
    DoExample;
    DoExample;
    Write('Continue Y/N? ');
    Readln(Ch);
    DoExample;
  Until Ch In ['N','n'];
End.
```

Name of procedure

Procedure heading

Writeln is a procedure which writes a line of text to the screen, e.g. The sun, *moving the cursor to the beginning of the next line*

Procedure body

Write is a procedure which writes text to the screen without moving the cursor to the beginning of the next line

Readln is a procedure which reads a line of text from the keyboard, e.g. N

Procedure call

Another procedure call

Another procedure call

Readln(Ch) reads the first character typed and stores it in variable Ch then skips the rest of the line, if there is more

Another procedure call

Yet another procedure call

Checks to see if the value stored in variable Ch is 'N' or 'n'

Table 1.1.2.9 Pascal program containing calls to procedures DoExample, Writeln, Write, Readln

Table 1.1.2.10 shows a VB.Net program which declares a user-defined procedure called `DoExample`.

The procedure `DoExample` is called three times when the program executes. `Main()` is called to run the program.

In VB.Net, `Main()` is a standard procedure name found in every executable module (Visual Studio supplies a module template containing `Main()`).

VB.Net doesn't use the keyword `Procedure` but instead surrounds the body of the procedure with the keywords `Sub` and `End Sub`.

```
Module ProcedureExample
    Sub DoExample()
        Console.WriteLine("The sun has ")
        Console.WriteLine("got its hat on,  ")
        Console.WriteLine("hip hip hooray.")
    End Sub
    Sub Main()
        DoExample()
        DoExample()
        DoExample()
        Console.ReadLine()
    End Sub
End Module
```

Table 1.1.2.10 VB.Net program containing calls to a procedure DoExample

Instead of the language keyword `Program` as in Pascal, VB.Net uses the keyword `Module`. It has a matching keyword `End Module` placed at the very end of the program.

Single licence - Abingdon School

Figure 1.1.2.13 shows one call to a Python 3.5 procedure with the name do_example which has been defined using Python's keyword `def`.

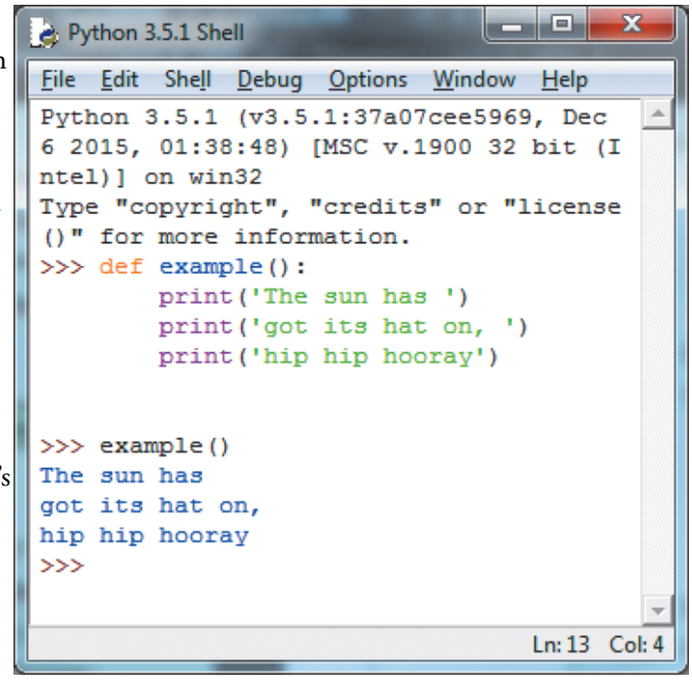


Figure 1.1.2.13 Shows one call to a Python 3.5 procedure with the name do_example which has been defined using Python's keyword `def`

Procedure interface

A procedure interface is **a mechanism for passing data into and out of a procedure**.

In Pascal, for example, `x` is known as a **formal parameter** as shown in *Table 1.1.2.11*. This is a variable used in the body of the procedure and which appears in the interface. When the procedure is called an **actual parameter** is supplied through the procedure's interface, e.g. `DoInterfaceExample(MyName)` where the variable `MyName` is the **actual parameter**. The value in `MyName` is copied into the variable `x` when the procedure is called by `DoInterfaceExample(MyName)`.

Suppose `MyName` stores the string 'Fred Bloggs', then the variable `x` in the body of the procedure will contain the string 'Fred Bloggs'.

Language keyword *Procedure name* *Procedure interface*

```
Procedure DoInterfaceExample (x : String);
  Begin
    Writeln('The value passed to this procedure is ', x:4);
  End;
```

Formal parameter

Table 1.1.2.11 Procedure interface in Pascal

Key concept

Procedure interface: A procedure interface is a mechanism for passing data into and out of a procedure.

Questions

14 What is a procedure interface?

Programming task

1 In a programming language with which you are familiar, write a program which defines a procedure that displays the message passed it through its interface.

Function

A function is a type of subroutine designed to **always return a single result**. The mechanism by which the result is returned is different from that used by a procedure.

Functions may appear in expressions such as `5 x cube(3)` where `cube` is a mathematical function.
This is possible because a function **returns a value**.
So `5 x cube(3)` evaluates to `5 x 27` which evaluates to `135`.

In the first evaluation step, `cube(3)` is replaced by `27`, the result returned from the execution of function `cube(3)`. Any attempt to use a procedure in this manner would fail.

This is a key difference between a function and a procedure.

Single licence - Abingdon School

Procedure calls do not support a mechanism by which a value is substituted for the procedure call when the latter completes and returns control. If the procedure did then it would be a function[3].

For example, the following does not make sense as an expression because `drawsquare` doesn't "evaluate" to a single value which can be multiplied by `5` to produce a numeric result, e.g. `127`:

`5 x drawsquare(3)` where `drawsquare` is a procedure.

Because functions "evaluate" to a single value[4] they may appear on the right-hand side of an assignment statement.

For example,

```
x ← cube(3)
```

After this statement is executed, the variable `x` contains/stores the value `27`, because the result of calling `cube(3)` is `27`.

Attempting the following would fail because `Writeln` is a procedure. It also would not make sense.

```
x := Writeln('The sun has got its hat on');
```

In Pascal, the code `Writeln('The sun has got its hat on')` is a command which writes the string `'The sun has got its hat on'` to the screen then moves the screen's cursor to the beginning of the next line.

Figure 1.1.2.14 shows a function `cube(n)` defined in Python 3.5, and then called with argument 3.

In function `cube(n)`, `n` is called the formal parameter. When functions are called we use a slightly different nomenclature. The actual parameter is now called the **argument**.

For example, in the function call `cube(3)`, `3` is the argument to the function.

Pascal makes a clear distinction between procedure and function by using different keywords for defining each. *Table 1.1.2.12* shows a function `Cube` being defined using the language keyword `Function`. As Pascal is a strongly typed language, the data type of the formal parameter and the data type of the returned value must be specified. It is `Integer` for both in this example.

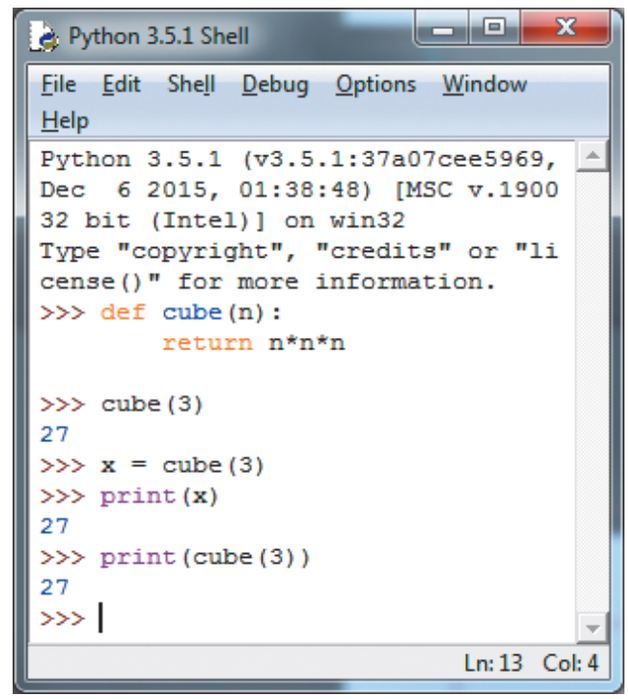


Figure 1.1.2.14 Function cube(n) defined and used in Python 3.5

Key term

Function:
A function is a type of subroutine designed to always return a single result. It may appear in expressions and on the right hand-side of an assignment statement unlike a procedure.

Questions

15 State **one** key difference between a function and a procedure.

3 Impure functions in addition to returning a result, may have side effects which is the collective name for other things that the function may do as well.

4 In some programming languages functions are allowed to return a structured result i.e. a result with more than one value.

Single licence - Abingdon School

Pascal uses the name of the function to assign the value that is to be returned.

Alternatively, the keyword Result can be used in place of Cube.

```
Function Cube (n : Integer) : Integer;
  Begin
    Cube := n * n * n;
  End;
```

Table 1.1.2.12 Function Cube(n) defined in Pascal

C#

In C#, the approach to defining subroutines is different from Pascal. In C#, subroutines are called functions, whether in their behaviour they are procedures or functions. In C#, the subroutine heading takes the following form

<visibility> <modifier ><return type> <subroutine-name>(<parameters>)

The first part is the visibility, and is optional. If not specified then the function will be private.
The second part is the modifier, and is optional. If it is specified then the function belongs to the class otherwise it belongs to a specific object of the class[5]. The modifier is the keyword `static`.

The third part is the return type. If this is `void` then no result is returned via the function mechanism. In this case the subroutine behaves as a procedure. The term "`void`" is meant to indicate that the return value is empty or nonexistent.

If the return type is a valid data type, e.g. `int`, then a result is returned via the function mechanism. In this case the subroutine behaves as a function.

Figure 1.1.2.15 shows a Visual Studio 2015 C# program which defines a procedure `DoExample` which is called twice by `Main`. It is a procedure because the specified return type is `void`. The procedure belongs to the type `class Program` because the modifier is specified with the keyword `static`.

Figure 1.1.2.16 shows a Visual Studio 2015 C# program which defines a function `Add2Numbers` which is called by `Main`.

It is a function because the specified return type is `int`.

The function belongs to the class `Program` because the modifier of `Add2Numbers` is specified with the keyword `static`.

`Add2Numbers` is used as a function in `Main`. It appears on the right-hand side of an assignment statement as follows

```
int answer = Add2Numbers(4, 8);
```

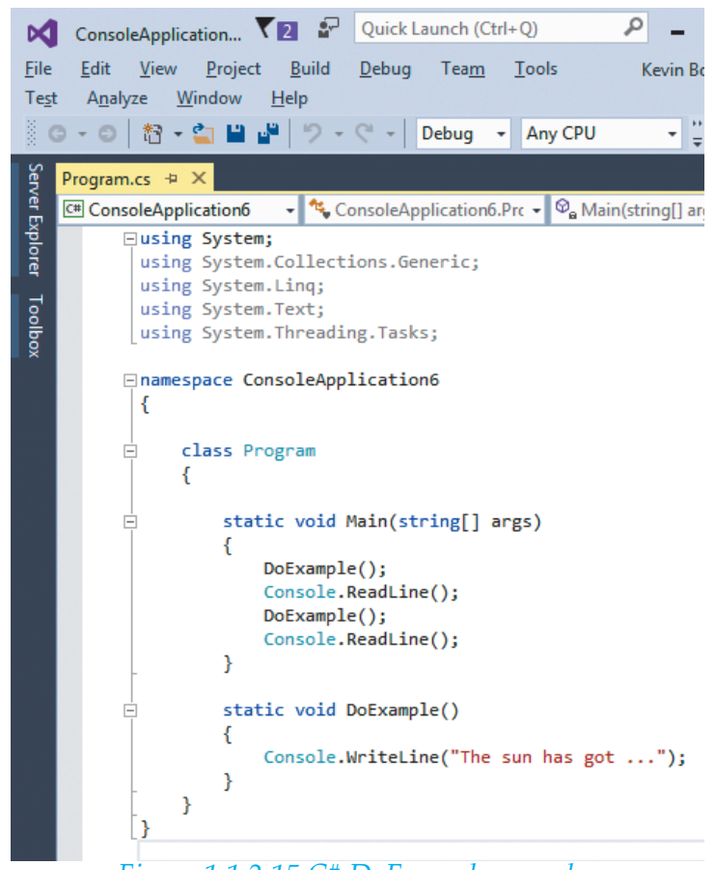


Figure 1.1.2.15 C# DoExample procedure

5 Object-oriented programming not in AS

Single licence - Abingdon School

Function `Add2Numbers` makes it possible to add two numbers from various places in a program, simply by calling this function instead of having to write the calculation code each time.

Java

In Java, the approach to defining subroutines is similar to C#'s approach.

```
<visibility> <modifier> <return-type>
<subroutine-name > ( <parameters >) {
      statements
}
```

The statements between the braces, { and }, in a subroutine definition make up the body of the subroutine.

The first part is the visibility, and is optional. If it isn't specified then the function will be private.

The second part is the modifier, and is optional. If it is specified then the function belongs to the class otherwise it belongs to a specific object of the class[6]. The modifier is the keyword `static`.

The third part is the return type. If this is `void` then no result is returned via the function mechanism. In this case the subroutine behaves as a procedure. The term "`void`" is meant to indicate that the return value is empty or nonexistent.

Figure 1.1.2.17 shows a Java program which defines a procedure `doExample` which is called twice by `Main`. It is a procedure because the specified return type is `void`. The procedure belongs to the class `Procedure` because the modifier of `doExample` is specified with the keyword `static`.

Figure 1.1.2.18 shows a Java program which defines a function `add2Numbers` which is called by `Main`.

It is a function because the specified return type is `int`.

The function belongs to the class `JavaAdd2Numbers` because the modifier of `add2Numbers` is specified with the keyword `static`.

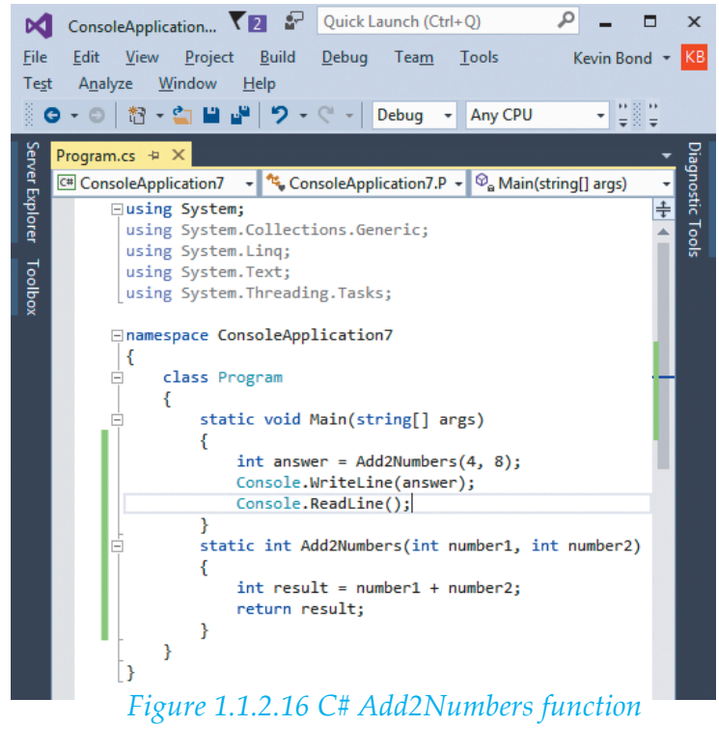


Figure 1.1.2.16 C# Add2Numbers function

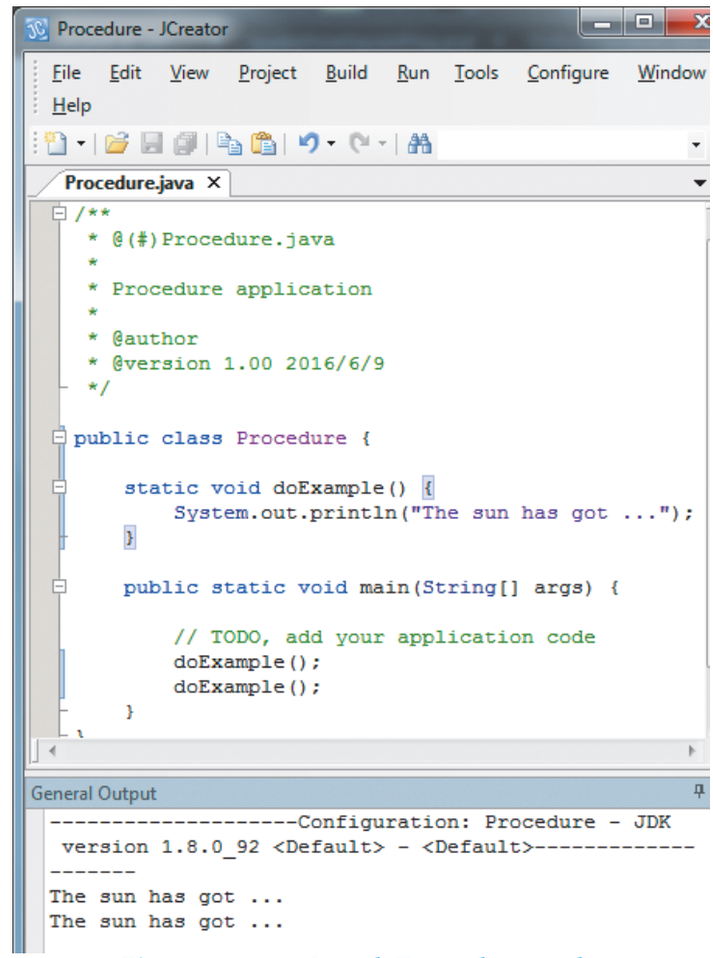


Figure 1.1.2.17 Java doExample procedure

`add2Numbers` is used as a function in `Main`. It appears on the right-hand side of an assignment statement as follows

```
int answer = add2Numbers(4, 8);
```

6 Object-oriented programming is not covered in AS

Single licence - Abingdon School

VB.NET

In VB.Net, the approach to defining a function is as follows

```
<visibility> <modifier> Function <FunctionName>
(<parameters>) As <return type>
        <Statements>
End Function
```

The first part is the visibility, and is optional. If it isn't specified then the function will be private.

The second part is the modifier, and is optional. The third part is the return type, e.g. Integer.

Figure 1.1.2.19 shows a VB.NET program which defines a function `Add2Numbers` which is called by `Main` from two different places in `Main`.

The return type of this function is `Integer`.

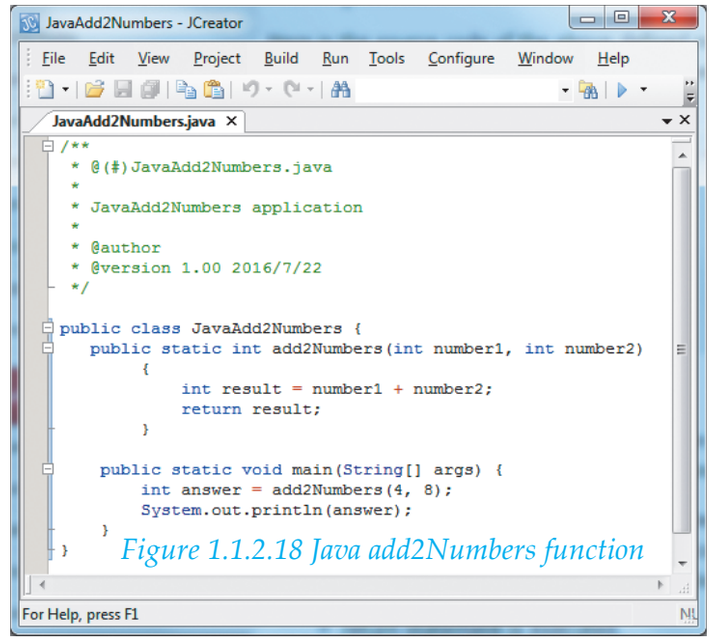


Figure 1.1.2.18 Java add2Numbers function

`Add2Numbers` is used as a function in `Main`. It appears on the right-hand side of an assignment statement as follows

```
answer = Add2Numbers(4, 8)
```

It also appears in a second call as an actual parameter to the procedure `WriteLine` as follows

```
Console.WriteLine(Add2Numbers(6,9))
```

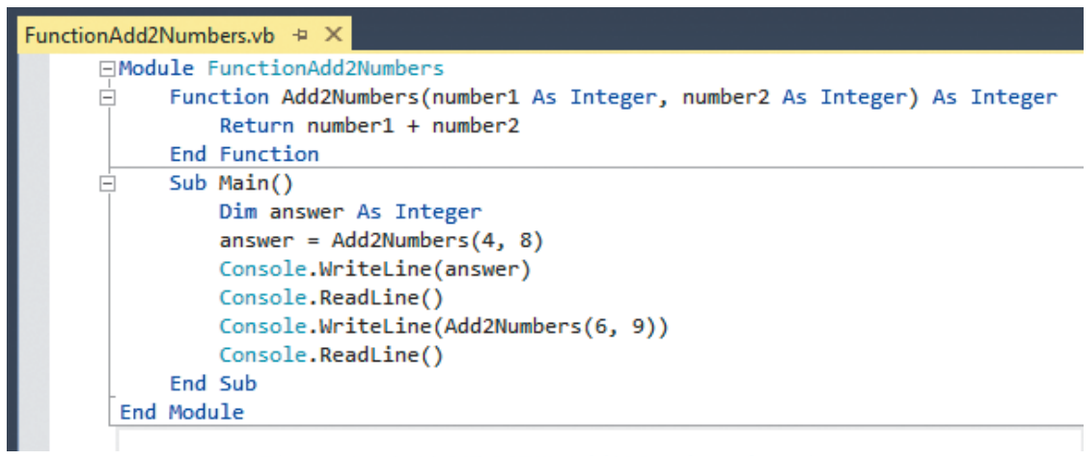


Figure 1.1.2.19 VB.NET Add2Numbers function

Programming task

2 In a programming language with which you are familiar:

(a) Write a program which defines a function that sums the first n natural numbers and returns this sum. The program should display this sum for a given n.

(b) Write a program which defines a function that sums the even numbers amongst the first n natural numbers and returns this sum. The program should display this sum for a given n.
(HINT: `If ((NatNo Mod 2) = 0) Then Even ← True Else Even ← False`)

(c) Write a program which defines a function that finds the largest of two given integers, x and y. The program should display the largest.

Single licence - Abingdon School

Definite and indefinite iteration

We introduced the concept of a loop in the section on iteration. In this section, we explore ways in which the number of iterations (i.e. repetitions of the body of the loop) is determined.

We have two cases to consider, definite and indefinite iteration:

- In **definite iteration**, the **number of iterations is known before the execution of the body of the loop is started**. For example, repeat 5 times writing the string "Hello World!" to the output device.
- In **indefinite iteration**, the **number of iterations is not known before the execution of the body of the loop starts**. The number of iterations depends on when **a specific condition is met** (and this depends on what happens in the body of the loop). For example, repeat printing the string "Hello World"until when asked the user declines to continue.

Definite iteration

Suppose that we wanted to output the value of a variable `i`, first when it is 1, next when it is 2, and so on until it is 5. We could do this with a *repeat until* loop or a *while* loop as shown by the pseudo-code examples in *Table 1.1.2.13*. The number of iterations of each loop is known in advance, it is five, so these are examples of **definite iteration**.

Note that a

- *repeat until <condition>* loop executes at least once
- a *while <condition>* executes zero or more times

```
i ← 0
Repeat
  i ← i + 1
  Output i
Until i = 5
```

```
i ← 1
While i <= 5
  Output i
  i ← i + 1
EndWhile
```

Table 1.1.2.13 Definite iteration with repeat and while loops

Key term

Definite iteration:
In definite iteration the number of iterations is known before the execution of the body of the loop is started.

Key term

Loop terminating condition:
The condition which causes execution of the loop body to terminate is known as the terminating condition.

Loop terminating condition

In both *repeat* and *while* loops quite a lot of work has to be done by the programmer.

In the example, the variable `i` has to be initialised, with 0 for the repeat loop and with 1 for the while loop.

It has to be incremented (`i ← i + 1`), and it has to be tested with the test condition `i = 5` for the repeat loop and `i <= 5` for the while loop.

The *repeat until* loop terminates when its test condition is true.

The *while loop* terminates when its test condition is false.

In each case, the condition which causes execution of the loop body to terminate is known as the **terminating condition**.

In definite iteration, this terminating condition is met after a known and predictable number of iterations.

Single licence - Abingdon School

For loop

There is an easier way for the programmer to program definite iteration called the **for loop**.
This pseudo-code for the *for loop* shown in *Table 1.1.2.14* is executed as follows

Step 1: The initial value of variable `i` is set to 1, "`i ← 1`". This step happens only once, regardless of how many times the loop repeats.

Step 2: The "`To 5`" part evaluates the condition (`i <= 5`) by comparing the value of `i` with 5.
If `i` is less than or equal to 5, the condition evaluates to true, and the statement "`Output i`" is executed. This sends the value of i to the output device, e.g. the VDU.
If `i` is greater than 5, the condition evaluates to false, and **the loop is exited**. The instruction immediately following `EndFor` is then executed next.

Step 3: The value of `i` is incremented by 1.

Step 4: The loop returns to *step 2* to evaluate the condition again.

```
For i ← 1 To 5
  Output i
EndFor
```

Table 1.1.2.14 Definite iteration with a for loop

Note that if the initial value of `i` is greater than the 5 as shown in *Table 1.1.2.15* the body of the loop should not be executed.

```
For i ← 6 To 5
  Output i
EndFor
```

Table 1.1.2.15 Definite iteration with a for loop

Variable `i` in this example is called the **loop control variable**.

A loop control variable must be an ordinal data type (see ordinal number *Unit 2 Chapter 5.1.6* page 18). An ordinal data type is one that consists of an ORDERED set of things in which each member has a single value only, e.g. an integer data type.

Questions

16 What is the output of the following pseudocode?

```
For i ← 1 To 5
  j ← 2 * i
  Output j
EndFor
```

17 What is the output of the following pseudocode?

```
For Ch ← 'A' To 'C'
  Output Ch
EndFor
```

18 How many times is the pseudo-code loop executed in *Table 1.1.2.16*?

```
For i ← 4 To 4
  j ← 2 * i
  Output j
EndFor
```

Table 1.1.2.16

Figure 1.1.2.20 shows an example of a *for loop* in Delphi/Pascal.
Figure 1.1.2.21 shows an example of a *for loop* in VB.NET.
For loops in C# and Java take a different form from Delphi/Pascal and VB.NET. This form is as follows

```
for (initialiser; condition; iterator)
    body
```

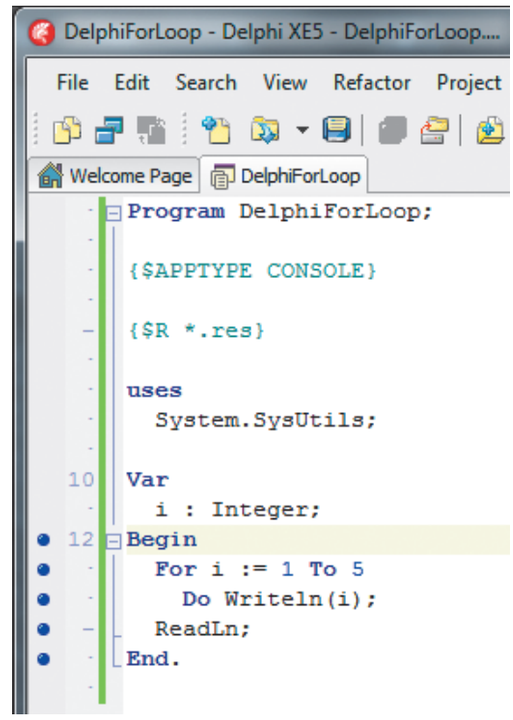


Figure 1.1.2.20 Delphi/Pascal For loop example

Single licence - Abingdon School

Figure 1.1.2.22 shows a C# and *Figure 1.1.2.23* a Java *for loop* example.

The *iterator* `i++` means increment `i` by 1. The *initialiser* `int i = 1` means create a loop control variable `i` and give it an initial value of 1.

The condition `i <= 5` evaluates to true if `i` is less than or equal to 5; false if `i` is greater than 5.

Python's *for loop* is different altogether from that of Delphi/Pascal, VB.Net, C# and Java.

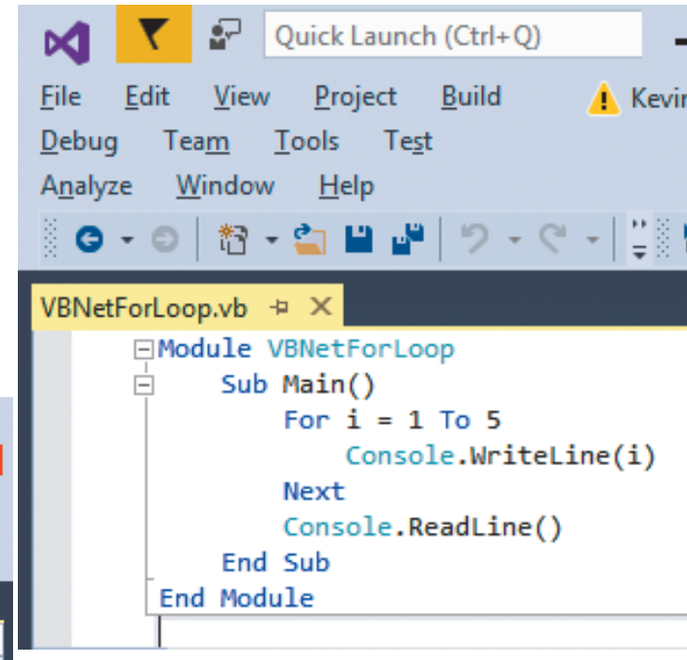


Figure 1.1.2.21 VB.NET For loop example

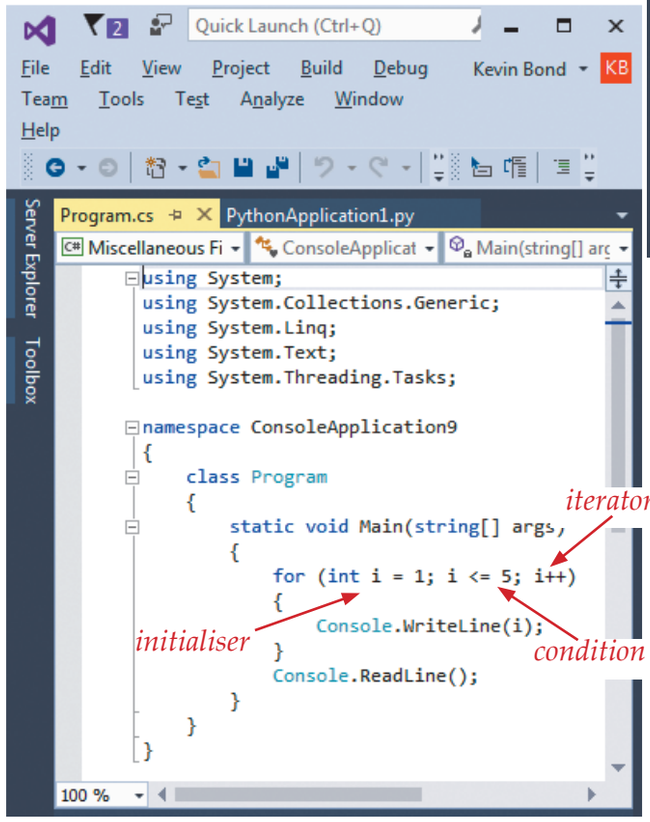


Figure 1.1.2.22 C# for loop example

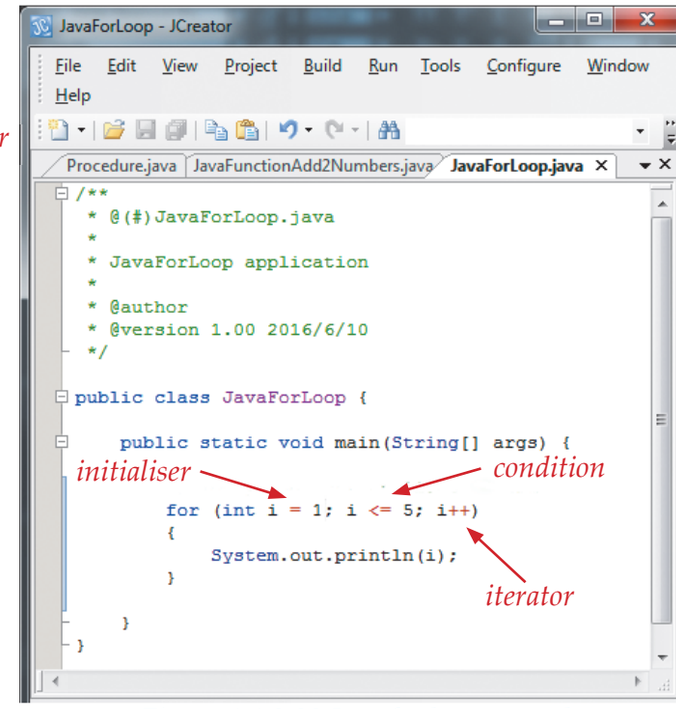


Figure 1.1.2.23 Java for loop example

Figure 1.1.2.24 shows a simple *for loop* example in Python 3.5.

The *range* function call `range(1,6)` generates the sequence of integers 1, 2, 3, 4, 5.

The last value generated is always one less then the upper bound value for the range, e.g. the last value is 5 if upper bound is 6.

Step 1: The loop control variable `i` is given the initial value 1, the first value in the range.

Step 2: The hidden condition (`i <= 5`) is evaluated by comparing the value of `i` with 5.

If `i` is less than or equal to 5, the condition evaluates to true, and the statement `print(i)` is executed.

If `i` is greater than 5, the condition evaluates to false, and the loop is exited. The instruction immediately following the body of the loop is then executed next, if one exists.

Step 3: The value of `i` is incremented by 1.

Step 4: The loop returns to the start of step 2 to evaluate the condition again.

Single licence - Abingdon School

Programming task

3 In a programming language with which you are familiar, write a program that codes the pseudo-code shown in Question 16.

Key term

Indefinite iteration:
In indefinite iteration, the number of iterations is not known before the execution of the body of the loop starts.

Key term

Infinite loop:
If the terminating condition cannot ever be met for some reason then the execution remains within the repeat or the while loop indefinitely. We then have a situation called an infinite loop.

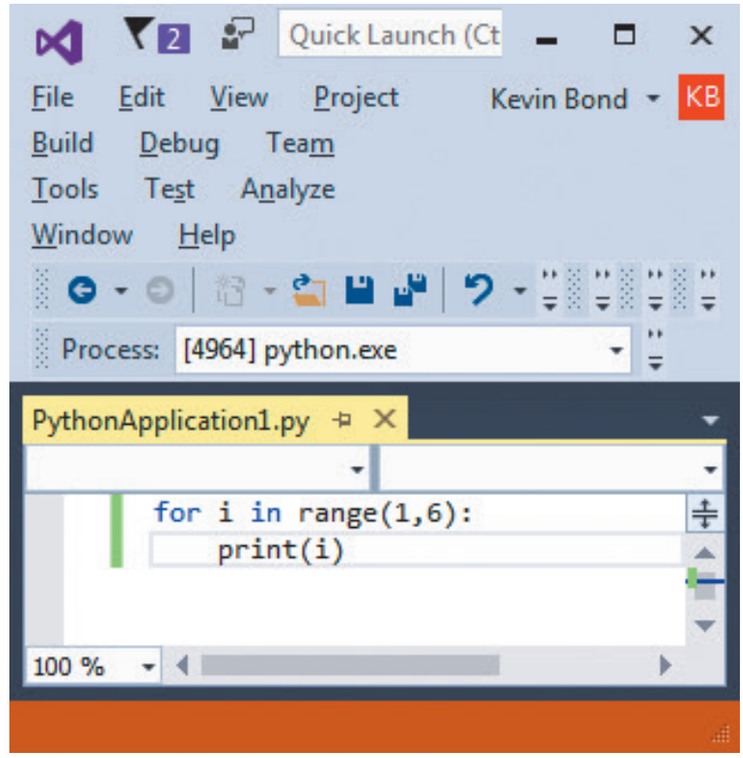


Figure 1.1.2.24 Python 3.5 For loop example in Visual Studio 2015

Indefinite iteration

The two cases of indefinite iteration to consider are *repeat* and *while* loops. "*For loops*" do not support indefinite iteration only definite iteration.
In *Table 1.1.2.17* the value of `i` in both pseudo-code examples is changed inside the loop in a way that cannot be predicted because it is obtained from the user through the input device, e.g. a keyboard, when `Input i` executes.

```
Repeat
  Input i
  Output i
Until i = 5
```

```
Input i
While i <> 5
  Input i
  Output i
EndWhile
```

Table 1.1.2.17 Indefinite iteration with repeat and while loops

The value of `i` before the *while loop* executes is only known at the time of execution when `Input i` executes.

It is therefore not possible to determine in advance the number of iterations of the body of each loop.

The number of iterations depends on when the terminating condition `i = 5` becomes true for the *repeat loop* and when the terminating condition `i <> 5` becomes false for the *while loop*.
If the terminating condition cannot ever be met for some reason then the execution remains within the *repeat* or the *while loop* indefinitely. We then have a situation called an **infinite loop**.
Table 1.1.2.18 shows how each loop could be written so that the iteration is indefinite.

Sometimes an infinite loop condition was not the intention but the loop's programmer has got it wrong. Exiting the loop then becomes impossible by ordinary means because the loop terminating condition can never be met.

```
Repeat
  Input i
  Output i
Until False
```

```
While True Do
  Input i
  Output i
EndWhile
```

Table 1.1.2.18 Indefinite iteration with repeat and while loops showing a situation called an infinite loop

Single licence - Abingdon School

Questions

19 What is the output of the following pseudo-code when the input is 1, 2, 3, 4, 0?

```
(a) Sum ← 0
    Repeat
      Input n
      Sum ← Sum + n
    Until n = 0
    Output Sum
```

```
(b) Product ← 1
    Input n
    While n > 0 Do
      Product ← Product * n
      Input n
    EndWhile
    Output n
```

20 What is the output of the following pseudo-code?

```
(a) j ← 4
    Repeat
      j ← j - 1
      Output j
    Until j < 1
```

```
(b) j ← 4
    Repeat
      j ← j + 1
      Output j
    Until j > 6
```

```
(c) j ← 4
    While j < 5
      j ← j + 1
      Output j
    EndWhile
```

21 What is the essential difference between *definite* and *indefinite iteration*?

22 What is the essential difference between a *repeat until* loop and a *while* loop?

23 What is meant by an *infinite loop*?

24 Write a loop in pseudo-code which demonstrates an *infinite loop*.

Indefinite iteration in Pascal, Delphi, VB.NET, C#, Java, Python <> means not equal to in Pascal/Delphi

Figure 1.1.2.25 shows two simple examples which illustrate how a *repeat* and a *while* loop can be constructed in Pascal/Delphi. The *while loop* is the only indefinite loop supported in Python. An example is shown in *Figure 1.1.2.26.*

Figure 1.1.2.27 shows a simple example which illustrates how a *repeat loop* can be constructed in VB.NET. The syntax of the construct is actually of the form *Do Loop Until <condition>*. *Figure 1.1.2.28* shows a simple example which illustrates how a *while loop* can be constructed in VB.NET using *Do While <condition> Loop*.

In C#, the *repeat loop* construct is implemented as a *do while loop* (*Figure 1.1.2.29*) and therefore the loop terminating condition is expressed as `(i != 5)`, the inverse of what it would be if *repeat until* could be used. "!=" means not equal to.

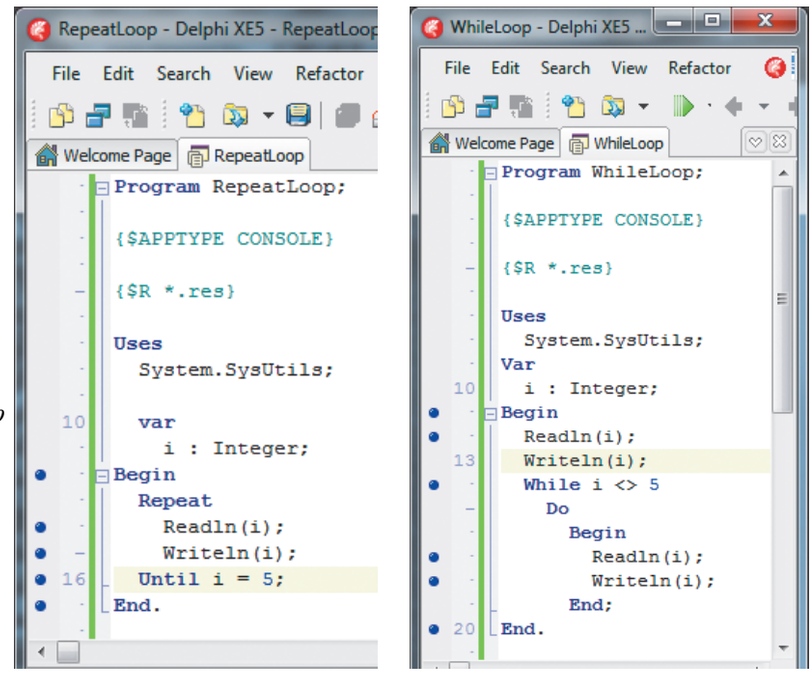


Figure 1.1.2.25 Repeat and while loops in Delphi

Figure 1.1.2.30 shows a simple example which illustrates how a *while loop* can be constructed in C#.

Single licence - Abingdon School

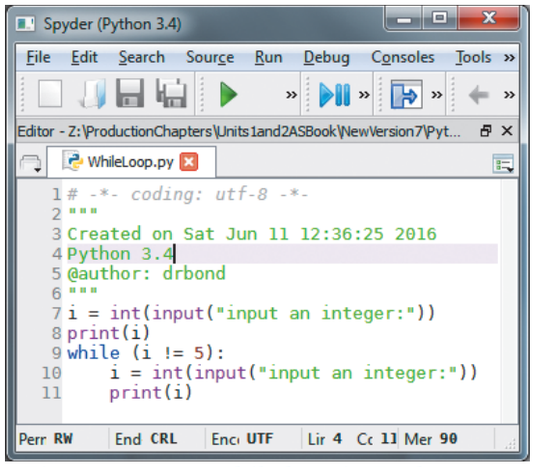


Figure 1.1.2.26 while loop in Python 3.4

!= means not equal to in Python

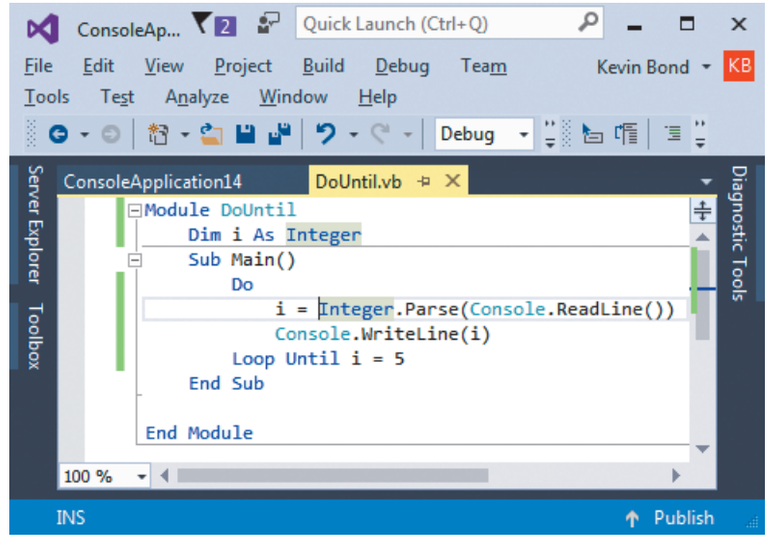


Figure 1.1.2.27 Repeat loop in VB.Net

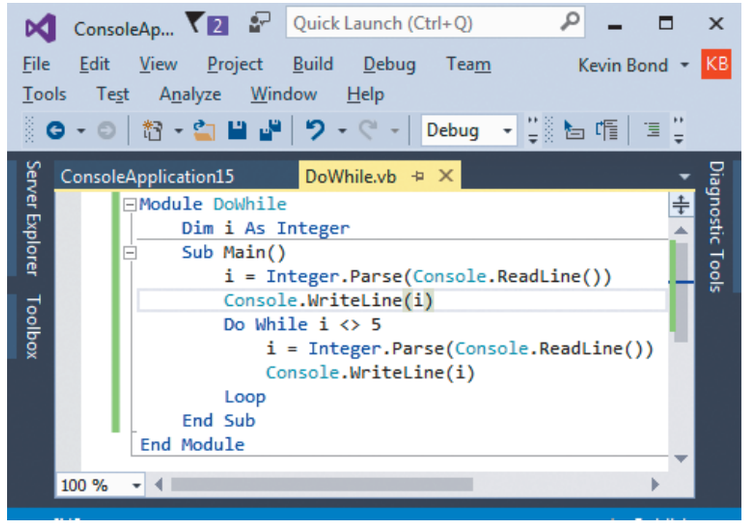


Figure 1.1.2.28 While loop in VB.Net

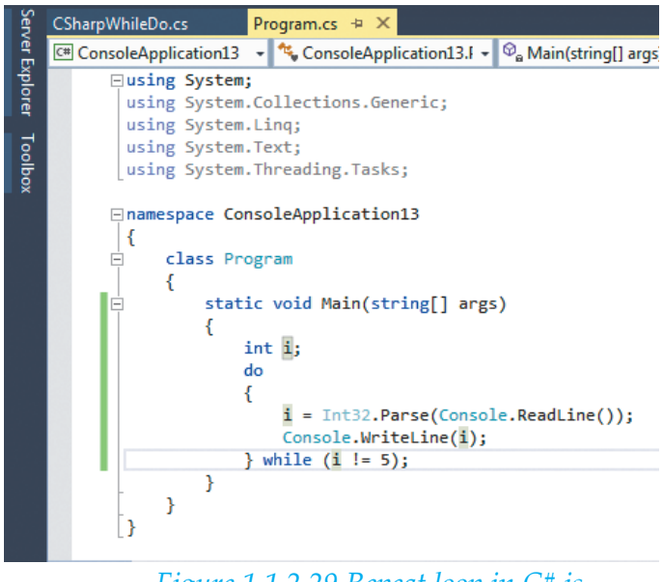


Figure 1.1.2.29 Repeat loop in C# is implemented as a do while loop

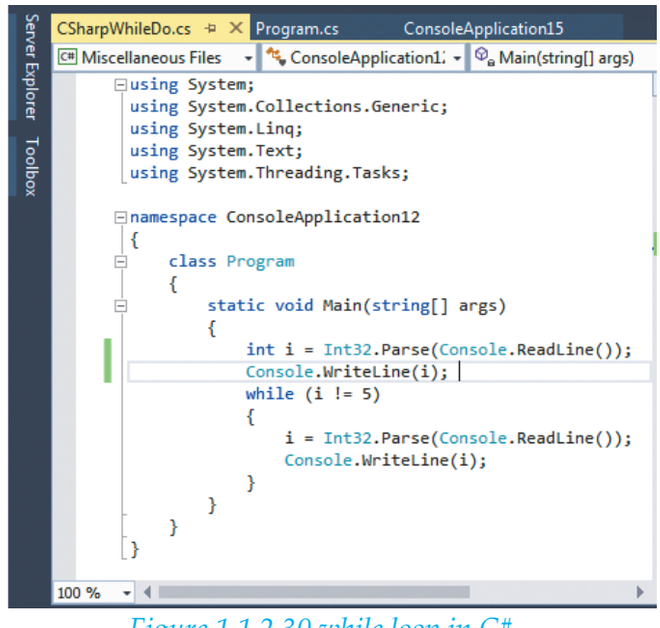


Figure 1.1.2.30 while loop in C#

Single licence - Abingdon School

In Java, the *repeat loop* construct is implemented as a *do while loop* (*Figure 1.1.2.31*) and therefore the loop terminating condition is expressed as `(i != 5)`, the inverse of what it would be if *repeat until* could be used. *Figure 1.1.2.32* shows a simple example which illustrates how a *while loop* can be constructed in Java.

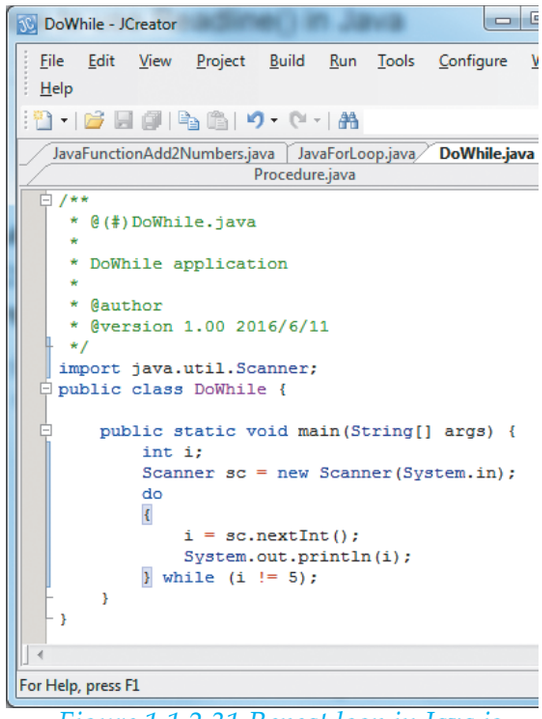


Figure 1.1.2.31 Repeat loop in Java is implemented as a Do While loop

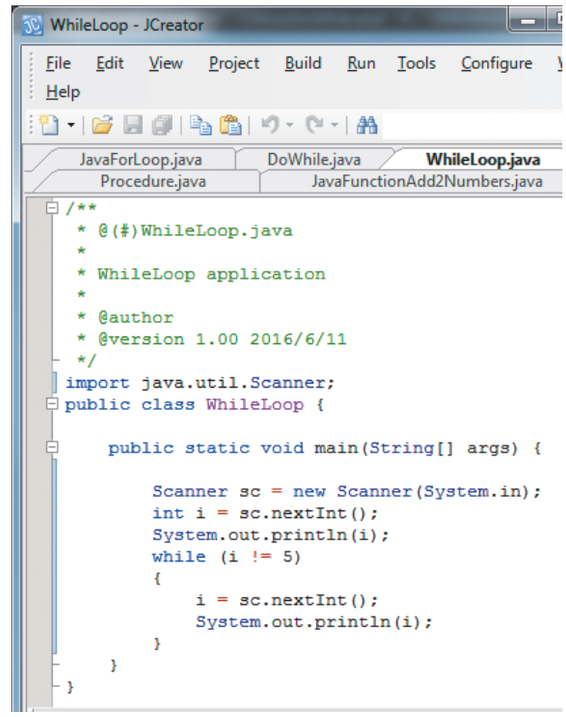


Figure 1.1.2.32 While loop in Java

Programming task

4 In a programming language with which you are familiar:

(a) Write programs that code the pseudo-code shown in Question 19(a) and 19(b).

(b) Write programs that code the pseudo-code shown in Question 20(a), 20(b) and 20(c).

Nested selection statements

The pseudo-code in *Table 1.1.2.19* contains three occurrences of *selection*, one marked in **black**, one in red and one in blue. The red-marked *selection* and the blue-marked *selection statements* are each nested inside the **black**-marked *selection statement*. This pseudo-code finds the largest of three numbers. The value of the first number is stored in variable `No1`. The value of the second number is stored in variable `No2` and the third in `No3`.

The value of the largest of the three numbers is stored in variable `Largest`.

If `No1` stores the value 6, `No2` the value 3 and `No3` the value 8, the result of the comparison `If No1 > No2`

is true because 6 is greater than 3.

The `Then` block of the **black**-coloured *selection* is executed next.

```
If No1 > No2
  Then
    If No1 > No3
      Then Largest ← No1
      Else Largest ← No3
    EndIf
  Else
    If No2 > No3
      Then Largest ← No2
      Else Largest ← No3
    EndIf
EndIf
```

Table 1.1.2.19 Pseudocode to find the largest of three numbers

Single licence - Abingdon School

This block contains another `If Then Else` statement, the one marked in red.

The result of the comparison `If No1 > No3` is false because 6 is less than 8.

The `Else` coloured in red is executed next and the value stored in variable `No3` is assigned to variable `Largest`.

Questions

25 Using the pseudocode in *Table 1.1.2.19*

(a) If `No1` stores the value 6, `No2` the value 7 and `No3` the value 5 which selection statements are executed? Explain your answer.

(b) If `No1` stores the value 6, `No2` the value 6 and `No3` the value 5 which selection statements are executed? Explain your answer.

Programming task

5 Write a program which defines a function that finds the largest of three integers, *x*, *y*, and *z*. The program should display the largest.

Nested iteration statements

We may place a *for loop* inside a *for loop* as shown in *Table 1.1.2.20* The inner *for loop* executes for each value of the loop control variable `i` of the outer *for loop*. The trace table, *Table 1.1.2.21*, shows the output and the values of `i` and `j`, the outer loop and inner loop control variables, respectively, for the first few values of the trace.

```
For i ← 1 To 5
  For j ← 1 To 4
    Output i
  EndFor
EndFor
```

Table 1.1.2.20 Nested for loop

i	j	Output
1	1	1
1	2	1
1	3	1
2	1	2
2	2	2
2	3	2

Table 1.1.2.21 Trace table

Questions

26 What is the output of the following pseudo-code?

(a)
```
For i ← 1 To 2
  For j ← 1 To 3
    Output j
  EndFor
EndFor
```

(b)
```
For i ← 1 To 3
  For j ← 1 To i
    Output j
  EndFor
EndFor
```

27 What is the output of the following pseudo-code?

(a)
```
For Ch1 ← 'A' To 'C'
  For Ch2 ← 'A' To Ch1
    Output Ch2
  EndFor
EndFor
```

(b)
```
For Ch1 ← 'a' To 'c'
  For Ch2 ← 'a' To 'c'
    For Ch3 ← 'a' To 'c'
      Output Ch1, Ch2, Ch3
    EndFor
  EndFor
EndFor
```


Single licence - Abingdon School

Questions

28 Write the following nested *for loop* as a single for loop

```
For i ← 1 To 4
  For j ← 3 To 5
    Output('*')
  EndFor
EndFor
```

29 Complete the trace table, *Table 1.1.2.22*, by hand tracing the following pseudo-code

```
For i ← 0 To 1
  For j ← 0 To 2
    Output j
  EndFor
EndFor
```

i	j	Output
0	0	

Table 1.1.2.22 Trace table

Programming tasks

6 In a programming language with which you are familiar,

(a) Write a program that codes the pseudo-code shown in Question 26(a).

(b) Write a program that codes the pseudo-code shown in Question 26(b).

Using meaningful identifier names

The names that have been used for variables, constants, subroutines such as `RunningTotal`, `NO_OF_DAYS_IN_WEEK`, `drawsquare`, `Cube` are all examples of identifiers. These identifiers describe what they represent, e.g. `RunningTotal`. **When an identifier is descriptive of what it represents or of its purpose, we say that it has a meaningful name.**

The following points are relevant to why programmers should use meaningful identifier names:

- Meaningful identifier names make it easier for the programmer to understand the source code because meaningfully-named identifiers describe what they represent or do
- Programmers spend far longer reading their source code than writing it so it is important that the source code is as descriptive of what it does as possible
- Programmers spend a lot of time reading other programmers' source code as well as their own and so it is important that the source code is as descriptive of what it does as possible
- Program source code needs to make sense when it is read, i.e. it should be possible to understand what the source code has been written to do, otherwise its intention will be unclear
- A programmer may wish/has to use source code that someone else has written. To do this successfully they need at least to understand the source code
- A programmer may be tasked to debug a program because it contains runtime/logical errors, e.g. it doesn't do what it is expected to do. The programmer will need to understand the source code in order to debug it

Single licence - Abingdon School

- A programmer may also be tasked to modify a program because what it is required to do has changed. The programmer will need to understand the source code to change it successfully.

Table 1.1.2.23 shows pseudo-code which sums the first 10 natural numbers.

```
x ← 0          // Initialise running total
y ← 1          // Initialise natural number
Repeat
  x ← x + y    // add natural number to running total
  y ← y + 1    // Increment natural number
Until y = 11   // Terminate loop when natural number is 11
```

Table 1.1.2.23 Pseudo-code to sum the first 10 natural numbers

Comments have been added to the pseudo-code which describe the purpose of each statement because the identifier names alone are not sufficient to make the purpose clear (this may be an oversimplified example but it is done to make a point).

Table 1.1.2.24 shows the pseudo-code rewritten with meaningful/self-describing identifier names.

Key term

Meaningful identifier name: When an identifier is descriptive of what it represents or of its purpose, we say that it has a meaningful name.

```
RunningTotal ← 0
NaturalNo ← 1
Repeat
  RunningTotal ← RunningTotal + NaturalNo
  NaturalNo ← NaturalNo + 1
Until NaturalNo = 11
```

Table 1.1.2.24 Pseudo-code to sum the first 10 natural numbers

The pseudo-code comments in *Table 1.1.2.23* use 148 characters whilst the meaningful/self-describing identifiers in the pseudo-code in *Table 1.1.2.24* use 61 characters.

We say that the pseudo-code in *Table 1.1.2.24* is self-documenting and because of this comments are largely superfluous.

Questions

30 Why is it important to use meaningful identifier names?

Programming tasks

7 In a programming language with which you are familiar,

(a) Write a program to print a " 4 times table " in the form

```
1 x 4 =  4
2 x 4 =  8
3 x 4 =  12
etc.
```

(b) Write a program to read in any integer, represented by the letter *n* say, and print an 'n times table'.

8 Write a program to print all the multiples of 3, starting at 3 and finishing at 90.

Single licence - Abingdon School

Programming tasks

9 In a programming language with which you are familiar,
(a) Write a program to input 6 numbers and display how many of them are zero.
(b) Write a program to input 10 numbers and print the largest. (Hint: assume the first number is the largest, store it in Largest, compare each new number with Largest, store new number in Largest if new number is larger. Alternatively, set Largest to 0 at start of program, then compare each new number with Largest as before).

10 Write a program to determine if a given year is a leap year. A leap year is a year which is exactly divisible by 4 and not a century year unless the century year is exactly divisible by 400. (Hint: (Year Mod 4) = 0 tests for exact division by 4).

11 Write a program that will enable a user to input the day of the week on which a month begins and the number of days in the month. The program should produce as output a calendar for the given month in the format shown below

Sun	Mon	Tues	Wed	Thurs	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

The day on which the month begins should be entered as an integer where 1 corresponds to Sunday, 2 to Monday and so on.

In this chapter you have covered:

- The following statement types in sufficient depth to understand and be able to use them, and to know how they can be combined in programs:
  - variable declaration
  - constant declaration
  - assignment
  - iteration
  - selection
  - subroutine(procedure/function
- Use of definite and indefinite iteration, including indefinite iteration with the condition at the start or end of the iterative structure (a theoretical understanding of condition(s) at either end of an iterative structure is required, regardless of whether they are supported by the language being used)
- The use of nested selection and nested iteration structures
- Meaningful identifier names and why it is important to use them.

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be familiar with and be able to use:*
  - *addition*
  - *subtraction*
  - *multiplication*
  - *real/float division*
  - *integer division, including remainders*
  - *exponentiation*
  - *rounding*
  - *truncation*

1.1.3 Arithmetic operations in a programming language

Addition/Subtraction/Multiplication

Arithmetic expressions

5 + 2

is an example of an arithmetic expression.

This expression has two operands and one operator as shown in *Figure 1.1.3.1*.

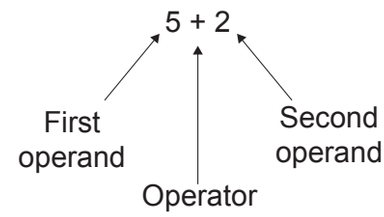


Figure 1.1.3.1 Arithmetic expression

Table 1.1.3.1 shows the arithmetic operators for the arithmetic operations addition, subtraction, multiplication and division in the programming languages C#, Java, Python, Pascal/Delphi and VB.NET.

Arithmetic Operator					Operation	Example
C#	Java	Python	Pascal/ Delphi	VB.NET		
+	+	+	+	+	Addition	3 + 5 is 8 3.0 + 2.0 is 5.0
-	-	-	-	-	Subtraction	6 – 3 is 3 6.0 – 3.0 is 3.0
*	*	*	*	*	Multiplication	3 * 2 is 6 3.0 * 2.0 is 6.0
/	/	/	/	/	Division	5.0 / 2.0 is 2.5

Table 1.1.3.1 Arithmetic operations in C#, Java, Python, Pascal/Delphi and VB.NET

Care needs to be taken with division because two kinds of division exist:

- Real number or float division
- Integer division

Real/floating point division

In real or floating point division, the quotient is a number with a fractional part, e.g. if 3 is divided by 2 the quotient is 1.5 in real/float division

3 / 2 = 1.5 (Floating point quotient)

whereas in integer division the quotient is integer, e.g.

3 / 2 = 1 (Integer quotient)

Single licence - Abingdon School

Questions

1 Express the following mathematical formulae in programming language form using the arithmetic operators from *Table 1.1.3.1*.

(a) $b^2 - 4ac$ (b) $\frac{1}{1 + x^2}$ (c) $\frac{1}{u} + \frac{1}{v}$

Programming task

1 *Table 1.1.3.2* shows two simultaneous linear equations in two variables x and y. The coefficients are a, b, m, n. For example, if the two equations are

5x + 4y = 22 and 3x + 8y = 30 then a = 5, b = 4, c = 22, m = 3, n = 8 and d = 30.

To solve for x and y we can use the following

$$x = \frac{(b*d - n*c)}{(m*b - a*n)} \qquad y = \frac{(a*d - m*c)}{(a*b - m*b)}$$

Write a program to solve for x and y given the coefficients a, b, m, n of two simultaneous linear equations. Test your program with a = 5, b = 4, c = 22, m = 3, n = 8 and d = 30.

a.x + b.y = c
m.x + n.y = d

Table 1.1.3.2

Programming languages differ in how they support the operations of real/float division and integer division. *Table 1.1.3.3* shows examples of both real/float and integer division in C#, Java, Python 2.x, Python 3.x, Pascal/Delphi and VB.NET.

Language	Example	Output
C#	`Console.WriteLine("Integer quotient: {0}", 3/2);` `Console.WriteLine("Float quotient: {0}", 3.0/2);`	`Integer quotient: 1` `Float quotient: 1.5`
Java	`System.out.println("Integer quotient: " + 3/2);` `System.out.println("Float quotient: " + 3.0/2);`	`Integer quotient: 1` `Float quotient: 1.5`
Python 2.x	`print "Integer quotient: ", 3/2` `print "Float quotient: ", 3.0/2`	`Integer quotient: 1` `Float quotient: 1.5`
Python 3.x	`print("Integer quotient: ", 3//2)` `print("Float quotient: ", 3/2)`	`Integer quotient: 1` `Float quotient: 1.5`

Pascal/Delphi	`Writeln('Integer Quotient: ', 3 Div 2);`
	`Writeln('Real Quotient: ', 3/2);`
	`Writeln('Real Quotient: ', 3.0/2);`
		`Integer Quotient: 1`
		`Real Quotient: 1.5`
		`Real Quotient: 1.5`

VB.NET	`Console.WriteLine("Integer Quotient: {0}", 3\2)` `Console.WriteLine("Float Quotient: {0}", 3/2)`	`Integer Quotient: 1` `Float Quotient: 1.5`

Table 1.1.3.3 Comparison of real/float division and integer division in C#, Java, Python, Pascal/Delphi and VB.NET

Integer division

The operation of integer division computes the integral part of the result of dividing the first operand by the second. The integral part is the whole number of times the second operand (the divisor) goes into the first operand (the dividend).

Figure 1.1.3.2 shows a length of rope of integer length L round a capstan of integer circumference C. The number of times that the rope can be wound on the circumference of the capstan is L DIV C where DIV is the **integer division operator** applied to an integer dividend and an integer divisor.

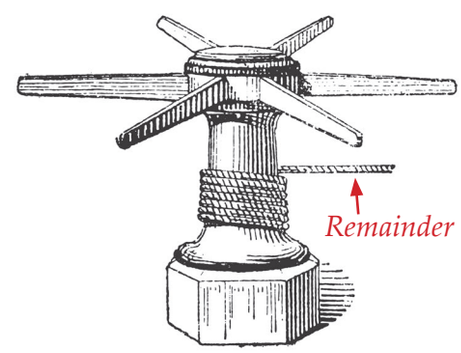


Figure 1.1.3.2 Rope wound round a capstan

Single licence - Abingdon School

The short length of rope left over is called the **remainder** because it is not long enough to fit the circumference. The remainder is given by *L* MOD *C* and is integral (a whole number).

Table 1.1.3.4 shows the integer division and integer remainder operators being used in C#, Java, Python, Pascal/Delphi and VB.NET.

Language	Example	Output
C#	`Console.WriteLine("Integer quotient: {0}", 5 / 2);` `Console.WriteLine("Integer remainder: {0}", 5 % 2);`	`Integer quotient: 2` `Integer remainder: 1`
Java	`System.out.println("Integer quotient: " + 5 / 2);` `System.out.println("Integer remainder: " + 5 % 2);`	`Integer quotient: 2` `Integer remainder: 1`
Python 3.x	`print("Integer quotient: ", 5 // 2)` `print("Integer remainder: ", 5 % 2)`	`Integer quotient: 2` `Integer remainder: 1`
Pascal/Delphi	`Writeln('Integer quotient: ', 5 Div 2);` `Writeln('Integer remainder: ', 5 Mod 2);`	`Integer quotient: 2` `Integer remainder: 1`
VB.NET	`Console.WriteLine("Integer quotient: {0}", 5 \ 2)` `Console.WriteLine("Integer remainder: {0}", 5 Mod 2)`	`Integer quotient: 2` `Integer remainder: 1`

Table 1.1.3.4 Integer division in C#, Java, Python, Pascal/Delphi and VB.NET

Questions

2 (a) How many times can a cotton thread of length 1655 cm be wound around a cotton reel of circumference 13 cm?

(b) How much cotton thread is left over?

3 Convert 4589 minutes into hours and minutes.

4 Explain how DIV and MOD can be used to obtain the number of hundreds, tens and units of a 3-digit integer, N.

5 Dividing an integer x by an integer y using integer division, we obtain quotient q and the remainder r.

The relationship between x, y, q and r is expressed in the following formula $x = y * q + r$

e.g. dividend x = 5, divisor y = 2, quotient q = 2, remainder r = 1, applying the formula 5 = 2 * 2 + 1.

Complete *Table 1.1.3.5*.

Dividend x	Divisor y	Quotient q	Remainder r
5	2	2	1
6	3	2	0
25	4	6	1
36	6		
121	7		
23	3		
1	3		
5	10		

Table 1.1.3.5

Programming task

2 Write a program to determine if a given year is a leap year.

A leap year is a year which is exactly divisible by 4 and not a century year unless the century year is exactly divisible by 400.

3 Write a program to produce a display of the time of day in the form

hours : mins : secs

given the time in seconds that have elapsed since 12:00 midnight (Hint: use DIV and MOD).

4 Write a program to display the number of hundreds, tens and units of a 3-digit integer number, N.

5 Write a program to display the digits, one per line, of an integer, N.

Single licence - Abingdon School

Programming task

6 Write a program to work out the day of the week on which a given date falls using the formula shown below

DayCode = ((13 * Month – 1) DIV 5 + Decade DIV 4 + Century DIV 4 + Decade + Day – 2 * Century) Mod 7

This calculates the day of the week on which any date after 1 January 1583 will fall or has fallen.

In this formula, the year is considered as consisting of two parts neither of which have their usual meaning:

- a century represented by the first two digits of its integer representation, e.g. 20 in 2010, and
- a decade represented by the last two digits, e.g. 10 in 2010.

The date for which the corresponding day of the week is required must be coded in the following way:

- The day of the month an integer between 1 and 31 inclusive
- The year is an integer, e.g. 1996 represents the year 1996.
- The month must be coded as an integer as follows:
  - March is coded 1, April as 2 and so on until December, which is coded as 10
  - January and February are coded as 11 and 12 respectively of the **previous year**. So, for example, 15 February 1996 would be represented as day 15, month 12 of year 1995.

The result of applying this formula is an integer in the range 0 – 6 inclusive.
The integer 0 represents Sunday, 1 represents Monday and so on.
Your program should use the days of the week in its output, i.e. Sunday, Monday, etc.

Questions

6 The following pseudo-code calculates the quotient q and remainder r when an integer x is divided by an integer y using integer division

```
r ← x
q ← 0
While r >= y
   r ← r - y
   q ← q + 1
EndWhile
```

Complete *Table 1.1.3.6* by tracing this pseudo-code by hand.

Iteration	x	y	r	q	r > = y
0	7	2	7	0	True

Table 1.1.3.6 Trace table

Single licence - Abingdon School

Exponentiation

Exponentiation is the operation of raising one quantity to the power of another,

e.g. $6^7 = 6 \times 6 \times 6 \times 6 \times 6 \times 6 \times 6 = 279936$.

C#, Java, Python, Delphi, and Pascal provide library support for the exponentiation operation as follows:

- C# : Math.Pow(Quantity, PowerValue)
- Java : Math.pow(quantity, powerValue)
- Python: math.pow(quantity, power_value) and quantity**power_value
- Delphi: System.Math.Power(Quantity, PowerValue)
- Pascal: Math.Power(Quantity, PowerValue)
- VB.NET provides the ^ operator: Quantity ^ PowerValue and Math.Pow(Quantity, PowerValue).

Operator precedence	Highest
()	Highest (evaluated first)
Exponentiation	↓
*, /, DIV, MOD	↓
+, -	Lowest (evaluated last)

Table 1.1.3.7 Operator precedence

Table 1.1.3.8 shows exponentiation in use in C#, Java, Python, Delphi, Pascal and VB.NET.

Language	Example	Output
C#	`Console.WriteLine("6 raised to the power of 7: {0}", Math.Pow(6, 7));`	6 raised to the power of 7: 279936
Java	`System.out.println("6 raised to the power of 7: " + Math.pow(6, 7));`	6 raised to the power of 7: 279936.0

Python	`import math`
	`print("6 raised to the power 7: ", math.pow(6, 7))`
	`print(6**7)`
		6 raised to the power of 7: 279936.0

Delphi	`Program Exponentiation;`
	`Uses`
	`  System.Math;`
	`Begin`
	`  Writeln('6 raised to the power of 7: ', Power(6, 7): 8:1);`
	`  Readln;`
	`End.`
		6 raised to the power of 7: 279936.0

Pascal (Lazarus)	`Program Exponentiation;`
	`Uses`
	`  Math;`
	`Begin`
	`  Writeln('6 raised to the power of 7: ', Power(6, 7): 8:1);`
	`  Readln;`
	`End.`
		6 raised to the power of 7: 279936.0

VB.NET	`Console.WriteLine("6 raised to the power of 7: {0}", 6 ^ 7)` `Console.WriteLine("6 raised to the power of 7: {0}", Math.Pow(6,7))`	6 raised to the power of 7: 279936

Table 1.1.3.8 Exponentiation in C#, Java, Python, Pascal/Delphi and VB.NET

Programming task

7 The volume of a sphere is $\frac{4\pi}{3}R^3$ where R is the radius of the sphere.
Write a program which calculates and displays the volume of a sphere of a given radius R. Your program should use exponentiation.

8 Write a program which sums the following series and displays the result

$$\frac{1}{2^1} + \frac{1}{3^2} + \frac{1}{5^3} + \frac{1}{7^4} + \frac{1}{9^5} + \frac{1}{11^6}$$

Single licence - Abingdon School

Programming task

9 A mortgage (a loan to purchase a house) is repaid monthly over Y years at an annual rate of interest of R%. The amount borrowed is A. The monthly repayment P is given by the formula

$$P = \frac{A \times \frac{R}{1200} \times \left(1 + \frac{R}{1200}\right)^{12Y}}{\left(1 + \frac{R}{1200}\right)^{12Y} - 1}$$

Write a program to calculate the monthly payment P for a loan A which is repaid over Y years at an annual interest of R%. The program should prompt the user to enter values for A, Y and R.

Rounding

Rounding off

Numbers can be rounded in many ways. *Table 1.1.3.9* shows rounding in C#, Java, Python, Pascal/Delphi and VB.NET in which rounding takes place to the closest integer value to the argument (**rounding off**).

When the fractional component is halfway between two integers, one of which is even and the other odd, then the even number is returned in C#, Python, Pascal/Delphi, and VB.NET. Java is the exception to this.

Lang.	Example	Output

C#	`Console.WriteLine("Rounding a float: {0}", Math.Round(5.667));`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(-5.667));`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(5.5));`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(-5.5));`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(5.4));`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(-5.4));`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(4.5));`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(-4.5));`
		`Rounding a float: 6`
		`Rounding a float: -6`
		`Rounding a float: 6`
		`Rounding a float: -6`
		`Rounding a float: 5`
		`Rounding a float: -5`
		`Rounding a float: 4`
		`Rounding a float: -4`

Java	`System.out.println("Rounding a float: " + Math.round(5.667));`
	`System.out.println("Rounding a float: " + Math.round(-5.667));`
	`System.out.println("Rounding a float: " + Math.round(5.5));`
	`System.out.println("Rounding a float: " + Math.round(-5.5));`
	`System.out.println("Rounding a float: " + Math.round(5.4));`
	`System.out.println("Rounding a float: " + Math.round(-5.4));`
	`System.out.println("Rounding a float: " + Math.round(4.5));`
	`System.out.println("Rounding a float: " + Math.round(-4.5));`
		`Rounding a float: 6`
		`Rounding a float: -6`
		`Rounding a float: 6`
		`Rounding a float: -5`
		`Rounding a float: 5`
		`Rounding a float: -5`
		`Rounding a float: 5`
		`Rounding a float: -4`

Python	`from math import round`
	`print("Rounding a float: ", round(5.667));`
	`print("Rounding a float: ", round(-5.667));`
	`print("Rounding a float: ", round(5.5));`
	`print("Rounding a float: ", round(-5.5));`
	`print("Rounding a float: ", round(5.4));`
	`print("Rounding a float: ", round(-5.4));`
	`print("Rounding a float: ", round(4.5));`
	`print("Rounding a float: ", round(-4.5));`
		
		`Rounding a float: 6`
		`Rounding a float: -6`
		`Rounding a float: 6`
		`Rounding a float: -6`
		`Rounding a float: 5`
		`Rounding a float: -5`
		`Rounding a float: 4`
		`Rounding a float: -4`

Pascal/ Delphi	`Writeln('Rounding a real: ', Round(5.667));`
	`Writeln('Rounding a real: ', Round(-5.667));`
	`Writeln('Rounding a real: ', Round(5.5));`
	`Writeln('Rounding a real: ', Round(-5.5));`
	`Writeln('Rounding a real: ', Round(5.4));`
	`Writeln('Rounding a real: ', Round(-5.4));`
	`Writeln('Rounding a real: ', Round(4.5));`
	`Writeln('Rounding a real: ', Round(-4.5));`
		`Rounding a real: 6`
		`Rounding a real: -6`
		`Rounding a real: 6`
		`Rounding a real: -6`
		`Rounding a real: 5`
		`Rounding a real: -5`
		`Rounding a real: 4`
		`Rounding a real: -4`

VB.NET	`Console.WriteLine("Rounding a float: {0}", Math.Round(5.667))`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(-5.667))`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(5.5))`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(-5.5))`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(5.4))`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(-5.4))`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(4.5))`
	`Console.WriteLine("Rounding a float: {0}", Math.Round(-4.5))`
		`Rounding a float: 6`
		`Rounding a float: -6`
		`Rounding a float: 6`
		`Rounding a float: -6`
		`Rounding a float: 5`
		`Rounding a float: -5`
		`Rounding a float: 4`
		`Rounding a float: -4`

Table 1.1.3.9 Rounding off in C#, Java, Python, Pascal/Delphi and VB.NET

Single licence - Abingdon School

Rounding up and rounding down

The operation of **rounding up** returns the smallest integral value that is greater than or equal to the specified real or floating-point number. This kind of rounding is sometimes called rounding toward positive infinity.

The operation of **rounding down** returns the largest integer less than or equal to the specified real or floating-point number. This kind of rounding is sometimes called rounding toward negative infinity.

Table 1.1.3.10 shows the functions for rounding up and rounding down available in in C#, Java, Python, Pascal/Delphi and VB.NET.

Lang.	Example	Output
C#	`Console.WriteLine("Rounding up a float: {0}", Math.Ceiling(5.667));` `Console.WriteLine("Rounding down a float: {0}", Math.Floor(5.667));`	`Rounding up a float: 6` `Rounding down a float: 5`
Java	`System.out.println("Rounding up a float: " + (int)Math.ceil(5.667));` `System.out.println("Rounding down a float: " + (int)Math.floor(5.667));`	`Rounding up a float: 6` `Rounding down a float: 5`

Python	`from math import ceil, floor`
	`print("Rounding up a float:  ", ceil(5.667))`
	`print("Rounding down a float:  ", floor(5.667))`
		`Rounding up a float: 6`
		`Rounding down a float: 5`

Pascal/ Delphi	`Writeln('Rounding up a real: ', Ceil(5.667));` `Writeln('Rounding down a real: ', Floor(5.667));`	`Rounding up a real: 6` `Rounding down a real: 5`
VB.NET	`Console.WriteLine("Rounding up a float: {0}", Math.Ceiling(5.667))` `Console.WriteLine("Rounding down a float: {0}", Math.Floor(5.667))`	`Rounding up a float: 6` `Rounding down a float: 5`

Table 1.1.3.10 Rounding up and rounding down in C#, Java, Python, Pascal/Delphi and VB.NET

In the case of Java, because the integral part of the argument is returned as a float, it displays as a float. To avoid this, the returned value is converted to integer by a process called casting. To cast the float to integer `(int)` is placed in front of `Math.floor(5.667)`. C# and VB.NET also return the integral value as a float but `WriteLine` formats the output as integer.

Truncation

When a number with a fractional part is truncated the fractional part is removed, e.g. 5.667 becomes 5.

The operation of truncation returns the integral part of the number. *Table 1.1.3.11* shows this operation applied to the argument 5.667 in C#, Java, Python, Pascal/Delphi and VB.NET.

The method `floor` in Java gives the largest integer that is less than or equal to the argument.

In the case of negative values this will give a more negative value than the argument, e.g. -5.667 becomes -6 after the operation `(int) Math.floor` is applied. The method `ceil` in Java gives the smallest integer that is greater than or equal to the argument.

In the case of negative values this will give a less negative value than the argument, e.g. -5.667 becomes -5 after the operation `(int) Math.ceil` is applied.

Lang.	Example	Output
C#	`Console.WriteLine("Truncating a float: {0}", Math.Truncate(5.667));` `Console.WriteLine("Truncating a float: {0}", Math.Truncate(-5.667));`	`Truncating a float: 5` `Truncating a float: -5`
Java	`System.out.println("Truncating a float: " + (int)Math.floor(5.667));` `System.out.println("Truncating a float: " + (int)Math.ceil(-5.667));`	`Truncating a float: 5` `Truncating a float: -5`

Python	`from math import trunc`
	`print("Truncating a float:  ", trunc(5.667))`
	`print("Truncating a float:  ", trunc(-5.667))`
		`Truncating a float: 5`
		`Truncating a float: -5`

Pascal/ Delphi	`Writeln('Truncating a real: ', Trunc(5.667));` `Writeln('Truncating a real: ', Trunc(-5.667));`	`Truncating a real: 5` `Truncating a real: -5`
VB.NET	`Console.WriteLine("Truncating a float: {0}", Math.Truncate(5.667))` `Console.WriteLine("Truncating a float: {0}", Math.Truncate(-5.667))`	`Truncating a float: 5` `Truncating a float: -5`

Table 1.1.3.11 Truncation in C#, Java, Python, Pascal/Delphi and VB.NET

Single licence - Abingdon School

Questions

7 Explain how to obtain the 3 most significant figures of the number 5.6676593, i.e. 5, 6, 6 by using a combination of truncation, subtraction and multiplication by 10.

8 Explain how to round 5.6676593 to 2 decimal places by using a combination of multiplication and division by 10 and a rounding operation which rounds to the closest integer value to the argument

In this chapter you have covered:

- Be familiar with and be able to use:
  - addition
  - subtraction
  - multiplication
  - real/float division
  - integer division, including remainders
  - exponentiation
  - rounding
  - truncation

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be familiar with and be able to use:*
  - *equal to*
  - *not equal to*
  - *less than*
  - *greater than*
  - *less than or equal to*
  - *greater than or equal to*

1.1.4 Relational operators in a programming language

Relational operators

Expressions involving the relational operators shown in *Table 1.1.4.1* produce Boolean results. For example,

2 < 3	True
5 > 6	False

Therefore, such Boolean expressions may be assigned to any Boolean variable, e.g. `FlagIsTrue`.

The following pseudo-code outputs the value True because 2 < 3 evaluates to True :

```
FlagIsTrue ← 2 < 3
Output FlagIsTrue
```

The following pseudo-code outputs the value False because 5 < 6 evaluates to False:

```
FlagIsTrue ← 5 > 6
Output FlagIsTrue
```

Operator					Meaning	Example (Pascal)	Outcome
C#	Java	Python	Pascal/ Delphi	VB.NET			
==	==	==	=	=	Equal To	6 = 6	True
<	<	<	<	<	Less Than	4 < 7	True
<=	<=	<=	<=	<=	Less Than Or Equal To	7 <= 3	False
>	>	>	>	>	Greater Than	34 > 12	True
>=	>=	>=	>=	>=	Greater Than Or Equal To	23 >= 23	True
!=	!=	!=	<>	<>	Not Equal To	6 <> 6	False

Table 1.1.4.1 Relational operators in C#, Java, Python, Pascal/Delphi and VB.NET

Relational operators are more commonly used in selection statements and loops.

For example,

```
If x >= 6
  Then Output "x is greater than or equal to 6"
  Else Output "x is not greater than or equal to 6"
EndIf

While x < 7
  x ← x + 2
  Output x
EndWhile
```


Single licence - Abingdon School

Questions

1. What is the outcome of evaluating each of the following expressions if `x` stores the value 5 and `y` the value 10?

   (a) `x = y` (b) `2*x < y` (c) `2*x <= y` (d) `x > y` (e) `2*x > y` (f) `x <> y`
   (g) `10*x <> 5*y`

2. What value is output by the following pseudo-code if `Flag` is a Boolean variable?

```
Flag ← 6 > 8
Output Flag
```

3. What message is output by the following pseudo-code?

```
If  6 <> 6
  Then Output "Have a nice morning!"
  Else Output "Have a nice evening!"
```

In this chapter you have covered:

- Using and becoming familiar with:
  - equal to
  - not equal to
  - less than
  - greater than
  - less than or equal to
  - greater than or equal to

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be familiar with and be able to use:*
  - *NOT*
  - *AND*
  - *OR*
  - *XOR*

Key term

Boolean operators: Operators which act on Boolean operands or Boolean values and evaluate or return a Boolean value are called Boolean operators.

Most programming languages support the Boolean operators NOT, AND, OR and XOR.

1.1.5 Boolean operations in a programming language

Boolean operators

Operators which act on Boolean operands or Boolean values and evaluate or return a Boolean value are called **Boolean operators**.

Boolean operands or values are those that are either True or False.

Most programming languages support the Boolean operators NOT, AND, OR and XOR so that a programming statement such as the following involving a Boolean operation can be written

```
If (x > 5) AND (y < 3) Then Output "A message"
```

Boolean operators are used to perform the **Boolean operations**

- `x AND y`
- `x OR y`
- `x XOR y`
- `Not x`

where `x` and `y` are Boolean variables or Boolean expressions.

For example if operand `x` is True and operand `y` is False then

- `x AND y` evaluates to False
- `x OR y` evaluates to True
- `x XOR y` evaluates to True
- `Not x` evaluates to False

Table 1.1.5.1 shows the symbols used for these Boolean operators in the programming languages, C#, Java, Python, Pascal/Delphi and VB.NET and their meaning.

Boolean Operator							
C#	Java	Python	Pascal/ Delphi	VB.NET	Meaning	Example (Pascal)	Outcome
!	!	not	NOT	Not	Evaluates to true, if operand false; Evaluates to false if operand is true	NOT True	False
&&	&	and	AND	And	Evaluates to true if both operands are true otherwise evaluates to false	True AND True	True
||	|	or	OR	Or	Evaluates to true if at least one operand is true otherwise false	True OR False	True
^	^	^	XOR	Xor	Evaluates to true if one and only one of the operands is true. Evaluates to false if both operands are true or if both operands are false.	True XOR True	False

Table 1.1.5.1 Boolean or Logical operators operators in C#, Java, Python, Pascal/Delphi and VB.NET

Single licence - Abingdon School

Operator precedence

Operator precedence refers to the order in which operators are applied to operands in an expression.

The NOT operator has the highest precedence, followed by the AND operator, then OR and XOR operators which are at the same level of operator precedence.

Table 1.1.5.2 summarises operator precedence.

Operator precedence	Precedence
NOT	Highest (evaluated first)
AND	
OR, XOR	Lowest (evaluated last)

Table 1.1.5.2 Operator precedence of Boolean operators

For example, in the following logical expression `NOT x` is evaluated first because NOT has a higher precedence than AND.

```
NOT x AND y
```

If Boolean variable `x` stores the value False,

`NOT False` evaluates to True because `x` is False.

Substituting True for `NOT x` in the expression

```
NOT x AND y
```

we obtain

```
True AND y
```

If Boolean variable `y` stores the value True, we obtain

```
True AND True
```

which evaluates to True.

If the expression is

```
NOT (x AND y)
```

then `x AND y` is evaluated first

The result of evaluating `x AND y` is False if `x` is False and `y` is True.

We now have

```
NOT (False)
```

which evaluates to True.

Now consider the logical expression with Boolean variables, `x`, `y` and `z` shown below

```
x AND y OR z
```

The operator precedence of AND is higher than OR.

If `x` and `y` store the value False and `z` the value True the expression evaluates in the following order

1. `x AND y` evaluates to False
2. `False OR z` evaluates to True

To change the order of evaluation we need to bracket the term `y OR z` in the expression `x AND y OR z` as follows

```
x AND (y OR z)
```

If `x` and `y` store the value False and `z` the value True the expression evaluates in the following order

1. `y OR z` evaluates to True
2. `x AND True` evaluates to False

Single licence - Abingdon School

Operators at the same level of precedence evaluate left to right in an expression.

In the following expression, `x` is True, `y` is True and `z` is False

```
x XOR y OR z
```

The expression evaluates in the following order

1. `x XOR y` evaluates to False
2. `False OR False` evaluates to False

If the expression is

```
x OR y XOR z
```

and `x` is True, `y` is True and `z` is False, the expression evaluates in the following order

1. `x OR y` evaluates to True
2. `True XOR False` evaluates to True

Questions

1 What is the outcome of evaluating each of the following expressions if `x` stores the value True and `y` the value False

(a) `NOT x` (b) `x AND y` (c) `x AND NOT y` (d) `x OR y` (e) `NOT x OR y`
(f) `x XOR y` (g) `NOT x XOR y`?

2 What is the outcome of evaluating each of the following expressions if `x` stores the value True, `y` the value False and `z` the value False

(a) `x AND y AND z` (b) `x AND y OR z` (c) `x OR y AND z`
(d) `(x OR y) AND z` (e) `x AND NOT y AND z` (f) `x AND NOT (y AND z)`
(g) `x AND NOT(y XOR z)`?

3 Integer variable `s` stores the value 9 and integer variable `t` stores the value 8.
What is the output of the following pseudo-code?

```
If (s > 3) AND (t < 6)
  Then Output "Have a nice morning!"
  Else Output "Have a nice evening!"
```

In this chapter you have covered:

- Becoming familiar with and using:
  - NOT
  - AND
  - OR
  - XOR

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be able to explain the differences between a variable and a constant*
- *Be able to explain the advantages of using named constants*

1.1.6 Constants and variables in a programming language

The differences between a variable and a constant

Data which are subject to change are modelled in a program by variables. A **variable** is simply a container in which a value may be stored for subsequent manipulation within a program.

The stored value may be changed by the program as it executes its instructions, hence the use of the term "variable" for the container.

Some of the data used in programs never change. For example, the ratio of the circumference of a circle to its diameter which is approximately 3.142.

Data which never change are modelled in a program by constants.

This means stating their values explicitly. To make a program easier to read, understand and change, a constant is given a **symbolic name** which can be used throughout the program whenever the value of the constant is required.

The advantages of using named constants

Named constants make programs easier to understand by replacing data of a fixed nature such as 3.142 with a name which makes the purpose clear, e.g. Pi.

Named constants make it easier to debug programs because the purpose of a constant in an expression in the program is made clear by its name, e.g. to represent the value of π. The literal value will appear in only one place in the program, i.e. where it is associated with its constant identifier, e.g. Const Pi = 3.142. Checking that the desired literal value is being used is easier to check if it is specified in one place only.

Named constants make programs easier to modify.

For example, suppose a program needs to change the value used for π to one with greater precision. Using a named constant Pi in the program for the value of π is a benefit because changing its value where Pi is defined, e.g. Const Pi = 3.142 automatically changes its value wherever else Pi is referenced in the program.

Named constants make it easier to avoid mistakes when writing programs.

For example, suppose a literal value of π is used in several places in a program. In the first place the programmer wrote the value of π as 3.142 but in the second place the digits were transposed and the programmer wrote 3.412. In a third place, it was written 3.144.

Single licence - Abingdon School

These errors could have been avoided by defining a constant Pi = 3.142 and then using the constant's name Pi throughout the program.

Programming task

1. Write a program which calculates the following for a given radius R, use a value of π = 3.142
   - circumference of a circle
   - area of a circle
   - volume of a sphere
   - surface area of a sphere.

   You should write your program so that the value of π in the source code can be easily changed to one of greater or lesser precision.

2. The acceleration due to gravity, g, at the surface of the earth is given by

$$g = \frac{G \times M}{R^2}$$

   where G is the gravitational constant, M is the mass of the earth and R is the radius of the earth, assuming a spherical earth.
   Write a program, using the values $G = 6.672 \times 10^{-11}$, $M = 5.98 \times 10^{24}$, R = 6371030, which calculates the value of g.
   You should write your program so that the value of M can be easily changed in the source code to a different value (e.g. to calculate g on the moon), and easily modified to calculate the value of g at a height h above the surface of the earth.

Questions

1. State **one** difference in between a variable and a constant in a programming language.
2. State **three** advantages of using named constants in programs.

In this chapter you have covered:

- The differences between a variable and a constant
- The advantages of using named constants

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be familiar with and be able to use:*
  - *length*
  - *position*
  - *substring*
  - *concatenation*
  - *character → character code*
  - *character code → character*
  - *string conversion routines*
    - *string to integer*
    - *string to float*
    - *integer to string*
    - *float to string*
    - *date/time to string*
    - *string to date/time*

1.1.7 String-handling operations

ASCII and ANSI

We introduced the string data type in *Chapter 1.1.1* as a sequence of characters.

Long ago, before the Internet and the World Wide Web, the only characters that mattered were unaccented uppercase and lowercase English letters based on a 26 letter alphabet, digits, and a variety of punctuation.

We have learned that computers work with numbers in the form of bit patterns. **Therefore, to store letters and other characters we have to assign each one a number.**

An encoding scheme called **ASCII** does just this. It was invented to encode the limited set of characters mentioned above.

In this scheme, characters are encoded using a number between 32 and 127. For example, in the ASCII character set, space is 32, and the letter "A" is 65. Device-control characters such as line feed and carriage return were added to this set of characters and allocated numbers in the range 0 to 31.

In all, the entire character set is encoded using numbers in the range 0 to 127. To represent this number range in the language of the machine, binary, requires 7 bits.

At the time, most computers worked with 8 bits not 7. Using the eighth bit made it possible to encode a further 128 characters, represented by codes in the range 128 to 255. Different OEM[1] character sets were dreamed up, which all used the top 128 characters for their own purposes.

Eventually, the OEM free-for-all got codified in the **ANSI standard**.

In the ANSI standard, everybody agreed on what to do for numbers below 128, which was in effect to use ASCII, but there were lots of different ways to handle the characters from 128 to 255, depending on which region of the world the computers were being made for. These different systems were called **code pages**. There are more than 220 DOS[2] and Windows code pages.

However, logogram-based languages such as Chinese have characters that number in the tens of thousands. These characters will never fit an 8-bit encoding scheme. It is impossible, therefore, to represent a string such as 你好世界 in any of the 8-bit ANSI encoding schemes.

Information

Code page identifiers:
Each code page is assigned a numeric code page identifier - https://msdn.microsoft.com/en-us/library/windows/desktop/dd317756(v=vs.85).aspx

Information

Character:
The Unicode Glossary defines a character as:
The smallest component of written language that has semantic value.

1 Original Equipment Manufacturer
2 DOS is short for disk operating system and an acronym for several computer operating systems that are operated by using the command line

Single licence - Abingdon School

Key term

Unicode:
Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. Unicode provides a unique number for every character:
no matter what the platform;
no matter what the program;
no matter what the language.

Information

Unicode code charts:
http://www.unicode.org/charts/

Information

The Unicode Standard Version 8.0 - Core Specification:
http://www.unicode.org/versions/Unicode8.0.0/ch02.pdf

Information

Characters and glyphs:
Characters are represented by code points that reside only in a memory representation as strings in memory, on disk, or in data transmission. For example U+0041 is Latin capital letter A. **The Unicode Standard deals only with character codes.**

Glyphs represent the shapes that characters can have when they are rendered or displayed, e.g. @ A A A A are all Unicode U+0041. In contrast to characters, glyphs appear on the screen or paper as particular representations of one or more characters. A repertoire of glyphs makes up a font. **Glyphs are not part of the Unicode Standard.**

Unicode

The answer to this problem is an encoding scheme called **Unicode** (www.unicode.org).

Unicode covers all of the characters in all of the world's writing systems, plus accents and other special marks and symbols, and control codes such as tab and carriage return, and assigns each one a standard number called a **Unicode code point**.

Unicode version 8 defines code points for over 120,000 characters in well over 100 languages and scripts but not Klingon, which was rejected in 2001 by the Unicode Technical Committee.

The Unicode Glossary defines a **character** as:

- The smallest component of written language that has semantic value
- The basic unit of encoding for Unicode character encoding
- The English name for the ideographic written elements of Chinese origin.

UTF-32 is the simplest Unicode encoding form.

Each Unicode code point is represented directly by a single 32-bit code unit. Because of this, UTF-32 has a one-to-one relationship between encoded character and code unit; it is a fixed-width character encoding form.
As for all of the Unicode encoding forms, UTF-32 is restricted to representation of code points in the range $0..10FFFF_{16}$ which is a 21-bit code space.
Whilst UTF-32 provides the simplest mapping, it **uses much more space than is necessary** - 4 bytes for every Unicode code point or character.

Most computer-readable text is in **ASCII**, which requires only 7 bits which can be accommodated in 1 byte. In fact, all the characters in widespread use still number fewer than 65,536, which can be coded in 16 bits or 2 bytes. This gave rise to two other Unicode encoding forms - **UTF-16** and **UTF-8**.

Figure 1.1.7.1 shows the three Unicode encoding forms - UTF-32, UTF-16, UTF-8 and how they are related to Unicode code points.

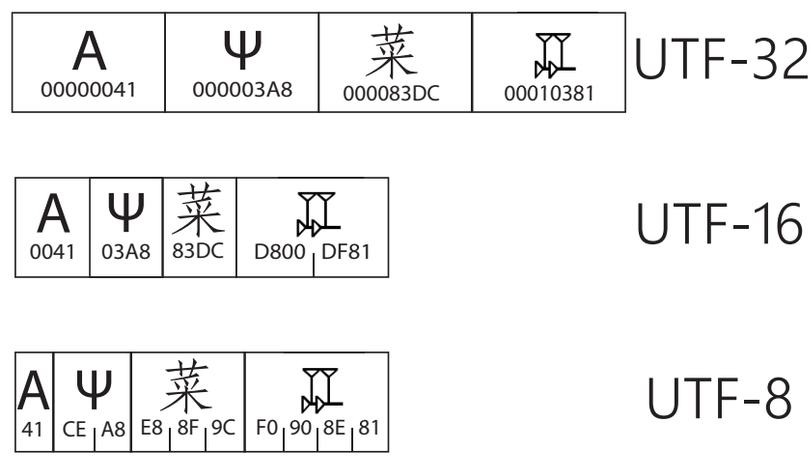


Figure 1.1.7.1 Unicode encoding forms

Single licence - Abingdon School

Note that for **UTF-16** most characters can be expressed with one 16-bit code unit, whose value is the same as the code point for the character, but characters with high code points require a pair of 16-bit code units called surrogate units.

UTF-8 like UTF-16 is a variable-length encoding of Unicode code points.

UTF-8 uses between 1 and 4 bytes to represent a Unicode code point or character but only 1 byte for ASCII characters, and only 2 or 3 bytes for most characters in common use.

The high order bits of the first byte of the encoding indicate how many bytes follow. A high order 0 indicates 7-bit ASCII, where each character takes only 1 byte, so it is identical to conventional ASCII. A high order 110 indicates that each character takes 2 bytes; the second byte begins with 10. Larger code points are also encoded in a similar manner. *Table 1.1.7.1* shows how the encoding indicates the number of bytes.

Information

Unicode code points:
Search Unicode code point and its corresponding character, UTF-32, UTF-16, UTF-8 encoding:
http://www.fileformat.info/info/unicode/char/search.htm

Key point

UTF:
Unicode offers multiple ways of representing the same code point (or Unicode character numerical value) in terms of actual storage, or of physical bytes.
UTF is an acronym for Unicode Transformation Format. These are bidirectional, algorithmic mappings which map each code point (the absolute numeric representation of a character) to a unique sequence of bytes representing the given character.

Strings

A string is a sequence of zero or more characters.

				Code point range	
0xxxxxxx				0 - 127	ASCII
110xxxxx	10xxxxxx			128 - 2047	Values < 128 unused
1110xxxx	10xxxxxx	10xxxxxx		2048 - 65535	Values < 2028 unused
11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	65536 - 0x10FFFF	Other values unused

Table 1.1.7.1 *Variable length encoding in UTF-8*

The encoding used for each character in a string could be ASCII (7-bits), ANSI (8 bits = ASCII/OEM), UTF-32, UTF-16, or UTF-8.
UTF-16 is effectively how characters are maintained internally in .NET[3].
Each character is encoded as a sequence of 2 bytes, other than surrogates which take 4 bytes. Surrogates are needed in those cases where the Unicode code point uses more than 16 bits, e.g. the white touch phone symbol has UTF-32 code point 1F57E. In UTF-16 it is the pair D83D, DD7E.
In UTF-16 the string 'ABC' will require 3 x 2 = 6 bytes of storage for its characters 'A', 'B' and 'C'.

Figure 1.1.7.2 shows the C# console program output for a UTF-16 encoded string '程序样例' consisting of four characters and a UTF-16 encoded string 'ABC', consisting of three characters. The program output also shows the code points in hexadecimal for each character. Note that characters, 'A', 'B' and 'C', each takes up 2 bytes but in each case, the leading byte hexadecimal value is not shown because it is zero.

In versions of Delphi from Delphi 2009 onwards, string characters are encoded in UTF-16. *Figure 1.1.7.3* shows that the string '程序样例' consisting of four characters is encoded in 8 bytes with two bytes per character.

Key point

UTF-8:
UTF-8 transforms characters into a variable-length encoding of 1-4 bytes. UTF-8 is used for HTML and similar protocols because it can be compact, most characters fall within the ASCII subset, e.g. HTML markers.
UTF-16:
In UTF-16 most characters can be expressed with one 16-bit code unit, whose value is the same as the code point for the character, but characters with high code points require a pair of 16-bit code units.
UTF-16 is popular in many operating systems and development environments such as Java and .NET. Most characters used in these scenarios fit in two bytes.
UTF-32:
In UTF-32 all code points encode to the same fixed length code (32 bits) but it is memory consuming and therefore has limited practical usage.

3 .NET Framework is a software framework developed by Microsoft

Single licence - Abingdon School

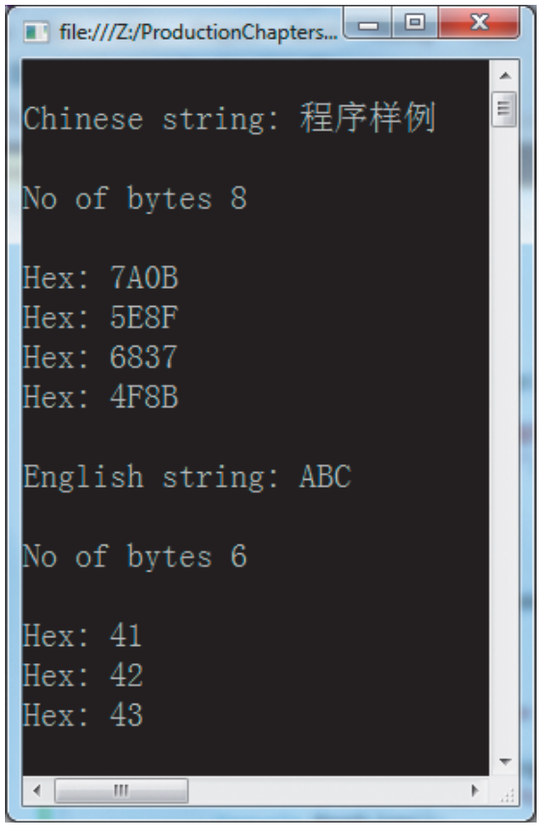


Figure 1.1.7.2 C# console program output for a UTF-16 encoded string '程序样例' *and a UTF-16 encoded string 'ABC'*

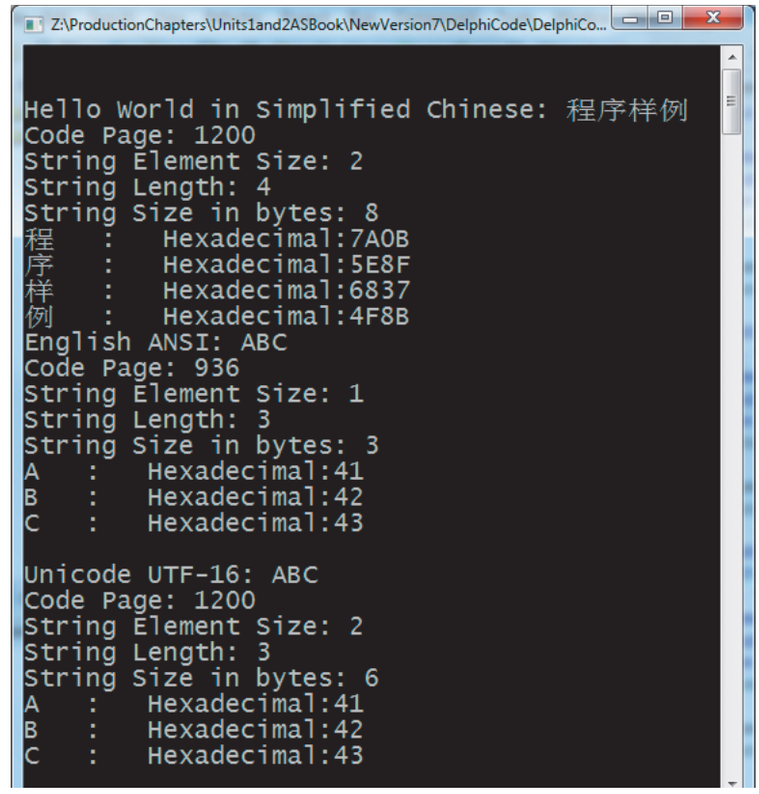


Figure 1.1.7.3 Delphi XE5 console program output for a UTF-16 encoded string '程序样例' *and an ANSI string 'ABC'* and a UTF-16 string *'ABC'*

Information

UTF-16 and .NET:
UTF-16 is effectively how characters are maintained internally in .NET. Each character is encoded as a sequence of 2 bytes, other than surrogates which take 4 bytes.
Surrogates are needed in those cases where the Unicode code point uses more than 16 bits.

Key point

ASCII and Unicode:
ASCII *x* is the same character as Unicode *x* for all characters within ASCII.

Each character is a called a string element, e.g. 程. The string element size is 2 bytes because UTF-16 is used. However, the length of the string is 4 characters.

The code page identifier 1200 indicates that the encoding is UTF-16.

The program variable that contains the string 'ABC' was declared as an ANSI string (8 bits). The string element size is 1 byte and the string length 3 characters.

When the same string is encoded in UTF-16, the element size becomes 2 bytes and the string size in byes 6 bytes.

It does not make sense to have a string without knowing what encoding it uses.

For a string in memory, in a file, or in an email message, to be interpreted correctly or displayed to users correctly, its encoding must be known.

Questions

1. What is Unicode?
2. What problem was solved by Unicode?

Single licence - Abingdon School

String operations

Strings are used to store human-readable text. The literal string, 'Hello World!', consists of twelve characters placed between single quotes. Some programming languages use single quotes, some use double quotes and others allow the use of both, e.g. Python.

The literal string of twelve characters, 'Hello World!', is stored in a container with a capacity for more than twelve bytes because

- each character may need more than one byte, e.g. when UTF-16 is used
- some bytes must be used to store the count of characters or to indicate the end of the sequence of characters
- some bytes will be needed if the string is reference counted (the number of references made to the string may be more than one and the string must not be destroyed if the reference count is greater than zero)
- some bytes will be needed if the code page is stored.

For this and other reasons programming languages provide a library of routines and string operators for programmers to use when working with strings.

String indexing

To access individual characters of a string an indexing scheme is required. *Figure 1.1.7.4* shows a scheme that starts numbering string elements (characters) at 0. *Figure 1.1.7.5* shows a Delphi XE5 program and its output. The program creates a string container (variable) called `s` in a declaration. The program assigns the string value 'Hello World!' to variable `s`. It obtains the index number of the character 'H' when `s.IndexOf('H')` is evaluated. This number is 0 which is written to the console by `Writeln`. It then confirms that the index of 'H' is 0 with `Writeln(s[0])`.

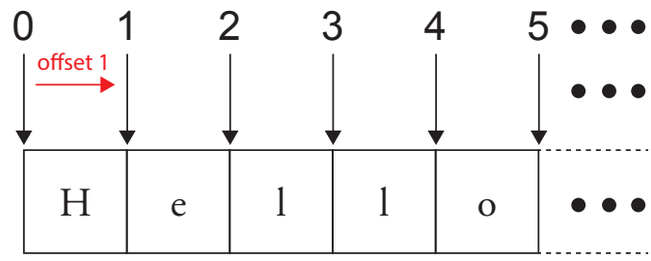


Figure 1.1.7.4 0-based numbering scheme for elements of a string

```
Program StringIndexingExample1;
{$APPTYPE CONSOLE}
{$ZEROBASEDSTRINGS ON}
{$R *.res}
Uses
  System.SysUtils;
Var s : String;
Begin
  s := 'Hello World!';
  Writeln(s.IndexOf('H'));
  Writeln(s[0]);
  Readln;
End.
```

Figure 1.1.7.5 String indexing illustrated by a Delphi XE5 program and its output

Single licence - Abingdon School

The brackets `[]` are one mechanism by which individual characters of a string may be accessed.

In this example, the index is treated as an **offset**. If the offset is 0 then we stay on 'H'.

To test this, we try `s[1]` and `s[4]` - see *Figure 1.1.7.6*. The character one on from the beginning of the string is 'e', four on is 'o'.

```
Program StringIndexingExample2;
{$APPTYPE CONSOLE}
{$ZEROBASEDSTRINGS ON}
{$R *.res}
Uses
  System.SysUtils;
Var s : String;
Begin
  s := 'Hello World!';
  Writeln(s[1]);
  Writeln(s[4]);
 Readln;
End.
```

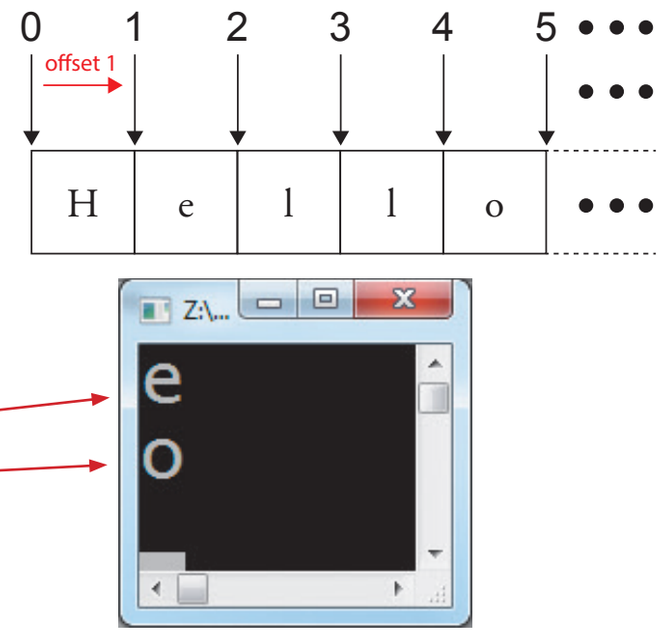


Figure 1.1.7.6 String indexing illustrated by a Delphi XE5 program and its output, 'e' has index 1 because it is offset by 1 from beginning of string, 'o' has index 4 because it is offset by 4.

Figure 1.1.7.7 shows a VB.NET program which uses string indexing. In VB.NET the brackets `()` are used to access individual characters of a string. In this example, the index is treated as an **offset**. If the offset is 0 then we stay on 'H'.

```
Module StringIndexing
    Sub Main()
        Dim s As String
        s = "Hello World!"
        Console.WriteLine(s)
        Console.WriteLine(s.IndexOf("H"))
        Console.WriteLine(s(0))
        Console.WriteLine(s(1))
        Console.WriteLine(s(4))
        Console.ReadLine()
    End Sub
End Module
```

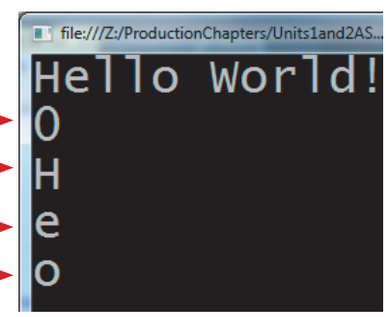


Figure 1.1.7.7 String indexing illustrated by a Visual Basic 2015 program and its output, the index of 'H' is 0, 'e' has index 1 because it is offset by 1 from beginning of string, 'o' has index 4 because it is offset by 4.

Single licence - Abingdon School

Figure 1.1.7.8 shows a C# program which uses string indexing. In C# the brackets `[]` are used to access individual characters of a string.

In this example, the index is treated as an **offset**. If the offset is 0 then we stay on 'H'.

```
using System;
namespace ConsoleStringIndexing
{
    class StringIndexing
    {
        static void Main(string[] args)
        {
            string s;
            s = "Hello World!";
            Console.WriteLine(s);
            Console.WriteLine(s.IndexOf('H'));
            Console.WriteLine(s[0]);
            Console.WriteLine(s[1]);
            Console.WriteLine(s[4]);
            Console.ReadLine();
        }
    }
}
```

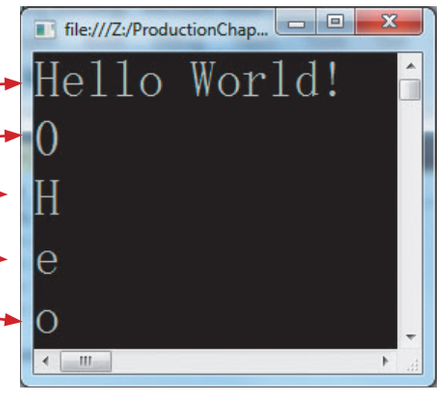


Figure 1.1.7.8 String indexing illustrated by a Visual C# 2015 program and its output, the index of 'H' is 0, 'e' has index 1 because it is offset by 1 from beginning of string, 'o' has index 4 because it is offset by 4.

Figure 1.1.7.9 shows a Java program which uses string indexing. In Java, the function `charAt` is used to access individual characters of a string. In this example, the index is treated as an **offset**. If the offset is 0 then we stay on 'H'.

```
public class StringIndexing {
    public static void main(String[] args) {
        String s;
        s = "Hello World";
        System.out.println("Hello World!");
        System.out.println(s.indexOf("H"));
        System.out.println(s.charAt(0));
        System.out.println(s.charAt(1));
        System.out.println(s.charAt(4));
    }
}
```

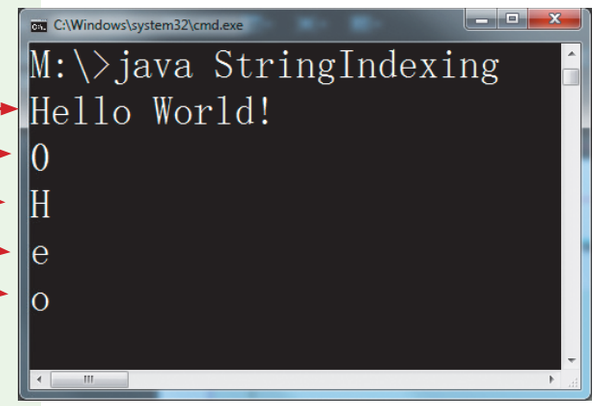


Figure 1.1.7.9 String indexing illustrated by a Java program and its output, the index of 'H' is 0, 'e' has index 1 because it is offset by 1 from beginning of string, 'o' has index 4 because it is offset by 4.

Single licence - Abingdon School

Figure 1.1.7.10 shows string indexing in Interactive Python 3.4.

The brackets `[]` are used to access individual characters of the string "Hello World!".

In this example, the index is treated as an **offset**. If the offset is 0 then we stay on 'H'.

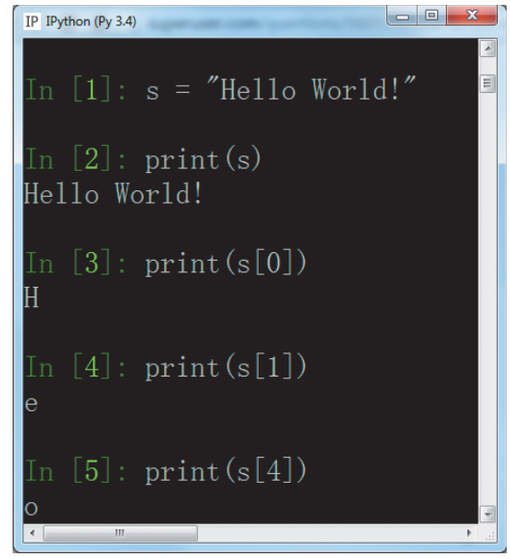


Figure 1.1.7.10 String indexing illustrated in Interactive Python 3.4 and its output, the index of 'H' is 0, 'e' has index 1 because it is offset by 1 from beginning of string, 'o' has index 4 because it is offset by 4.

Information

In Delphi prior to XE2 string indexing is one-based as shown in *Figure 1.1.7.11*. Indexing is treated as an ordinal number not an offset. Delphi XE3 onwards requires the directive ZEROBASEDSTRINGS OFF for one-based string indexing.

```
Program StringIndexingExampleOneBased;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils;
Var s : String;
Begin
  s := 'Hello World!';
  Writeln(s);
  Writeln(s[1]);
  Writeln(s[2]);
  Writeln(s[5]);
  Readln;
End.
```

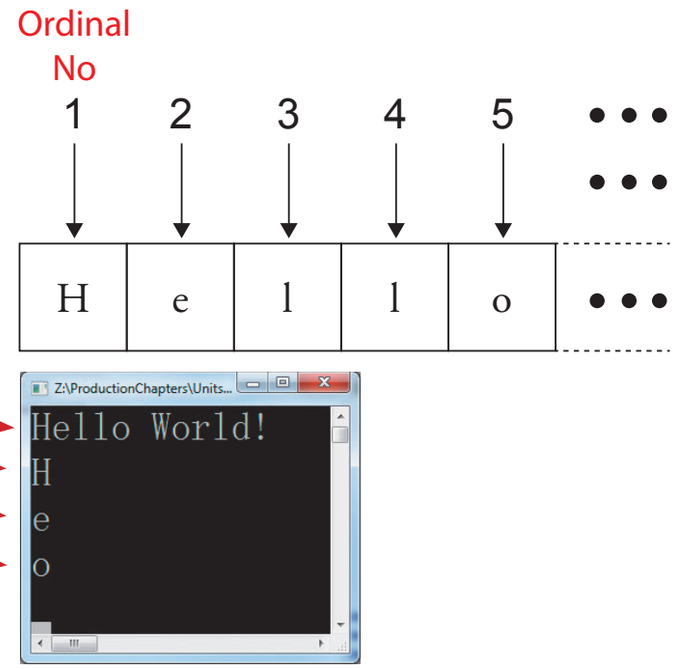


Figure 1.1.7.11 One-based string indexing illustrated by a Pascal/Delphi program and its output, the index of 'H' is 1, 'e' has index 2, 'o' has index 5.

Single licence - Abingdon School

Length of a string

In Pascal and Delphi the **Length** function returns the number of characters in a string.

Figure 1.1.7.12 shows the `Length` function returning 12 for the length of string `s` which contains string value 'Hello World!'. The *For loop* iterates from 0 to 11 (`Length(s) - 1`) to access each character of this zero-based-indexed string and `Writeln(s[i])` then sends a copy of the selected character to the console where it is displayed.

```
Program StringLengthZeroBased;
{$APPTYPE CONSOLE}
{$ZEROBASEDSTRINGS ON}
{$R *.res}
Uses
  System.SysUtils;
Var
  s : String;
  i : Integer;
Begin
  s := 'Hello World!';
  Writeln(s);
  Writeln('No of characters in string s = ', Length(s));
  Writeln('Character at offset 0 = ', s[0]);
  For i := 0 To Length(s) - 1
    Do Writeln(s[i]);
  Readln;
End.
```

Directive to compiler to create an executable that is a console application

Directive to compiler instructing it to use string indexing which starts at 0

String variable

Loop control variable i

String value 'Hello 'World!'

Stops console window closing until return key pressed

```
Hello World!
No of characters in string s = 12
Character at offset 0 = H
H
e
l
l
o

W
o
r
l
d
!
```

Figure 1.1.7.12 Using Pascal/Delphi's Length function to iterate through the characters of a zero-based string.

Figure 1.1.7.13 shows a C# program which uses the **Length** property of a C# string object. In C# a string is an object of type **String** whose value is a sequence of **Char** objects characters. The **Length** property of a string represents the number of Char objects it contains. The alias **string** is used in place of the class type String. Letter case is significant in C#.

String objects are immutable: they cannot be changed after they have been created. All of the **String** methods and C# operators that appear to modify a string actually return the results in a new string object.

In the following source code, **string** is an alias for the **String** class in the .NET framework:

```
string s = "Hello World!"
```

In the source code

```
Console.WriteLine("No of characters in string s = {0}", s.Length);
```

The expression `s.Length` evaluates to the length of the string that `s` contains.

In the literal string value `"No of characters in string s = {0}", s.Length);`

`{0}` is a place holder for the returned string length value 12.

Single licence - Abingdon School

```
using System;
namespace ConsoleStringLength
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "Hello World!";
            Console.WriteLine("No of characters in string s = {0}", s.Length);
            for (int i = 0; i < s.Length; i++)
            {
                Console.WriteLine(s[i]);
            }
            Console.ReadLine();
        }
    }
}
```

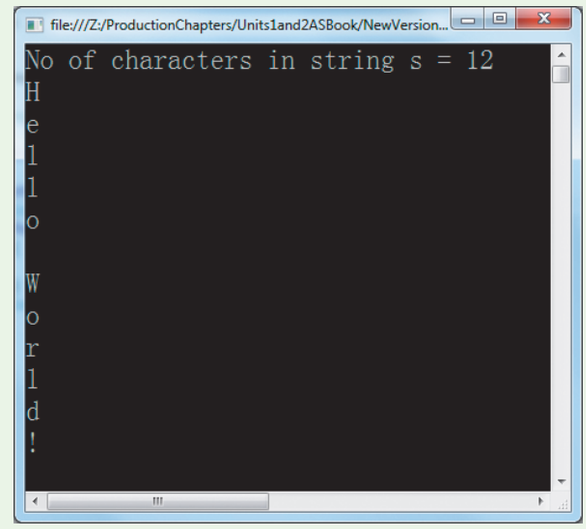
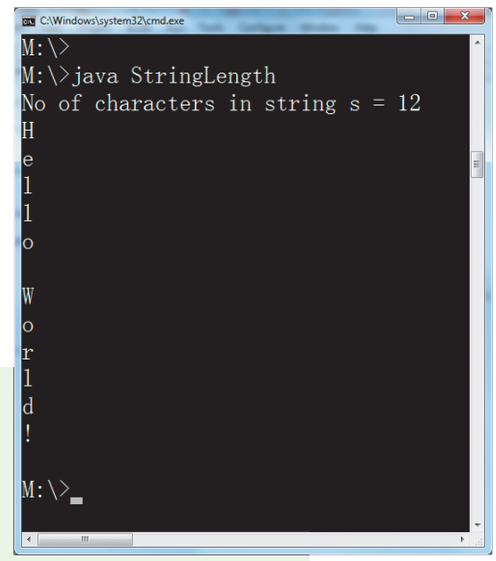


Figure 1.1.7.13 Using C#'s Length property of a string object to iterate through the characters of a zero-based string.

Figure 1.1.7.14 shows a Java program which uses the **length** method of a Java string object. In Java, a string is an object of type **String** whose value is a sequence of characters of data type **char** - a single 16-bit Unicode character. The **length** method of a string object returns the number of characters it contains.

Enclosing a character string within double quotes automatically creates a new String object. String objects are immutable, which means that once created, their values cannot be changed.

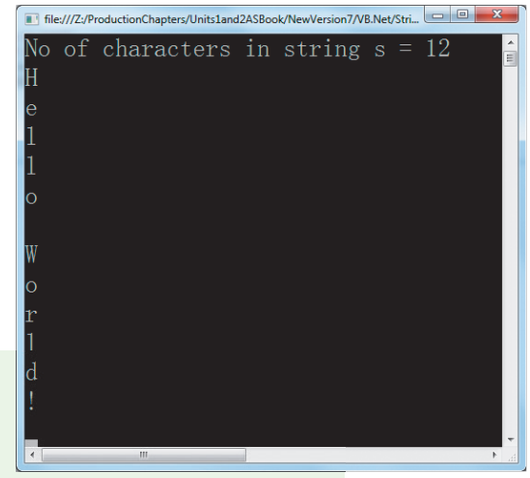



```
public class StringLength {
    public static void main(String[] args) {
        String s;
        s = "Hello World!";
        System.out.println("No of characters in string s = " + s.length());
        for (int i = 0; i < s.length(); i++){
            System.out.println(s.charAt(i));
        }
    }
}
```

Figure 1.1.7.14 Using Java's Length property of a string object to iterate through the characters of a zero-based string.

Single licence - Abingdon School

Figure 1.1.7.15 shows a VB.NET program which uses the **Length** property of a VB.NET string object. In VB.NET, a string is an object of type **String** whose value is a sequence of characters of data type **char** - a single 16-bit Unicode character. The **Length** property of a string object returns the number of characters it contains. Enclosing a character string within double quotes automatically creates a new String object. String objects are immutable, which means that once created, their values cannot be changed.




```
Module StringLength
    Sub Main()
        Dim s As String = "Hello World!"
        Console.WriteLine("No of characters in string s = {0}", s.Length)
        For i As Integer = 0 To s.Length - 1
            Console.WriteLine(s(i))
        Next
        Console.ReadLine()
    End Sub
End Module
```

Figure 1.1.7.15 Using VB.NET's Length property of a string object to iterate through the characters of a zero-based string.

Figure 1.1.7.16 shows that the use of the Python `len` function applied to a string variable `s`, as follows: `len(s)`, returns the number of characters in the string value "Hello World!" which `s` contains.

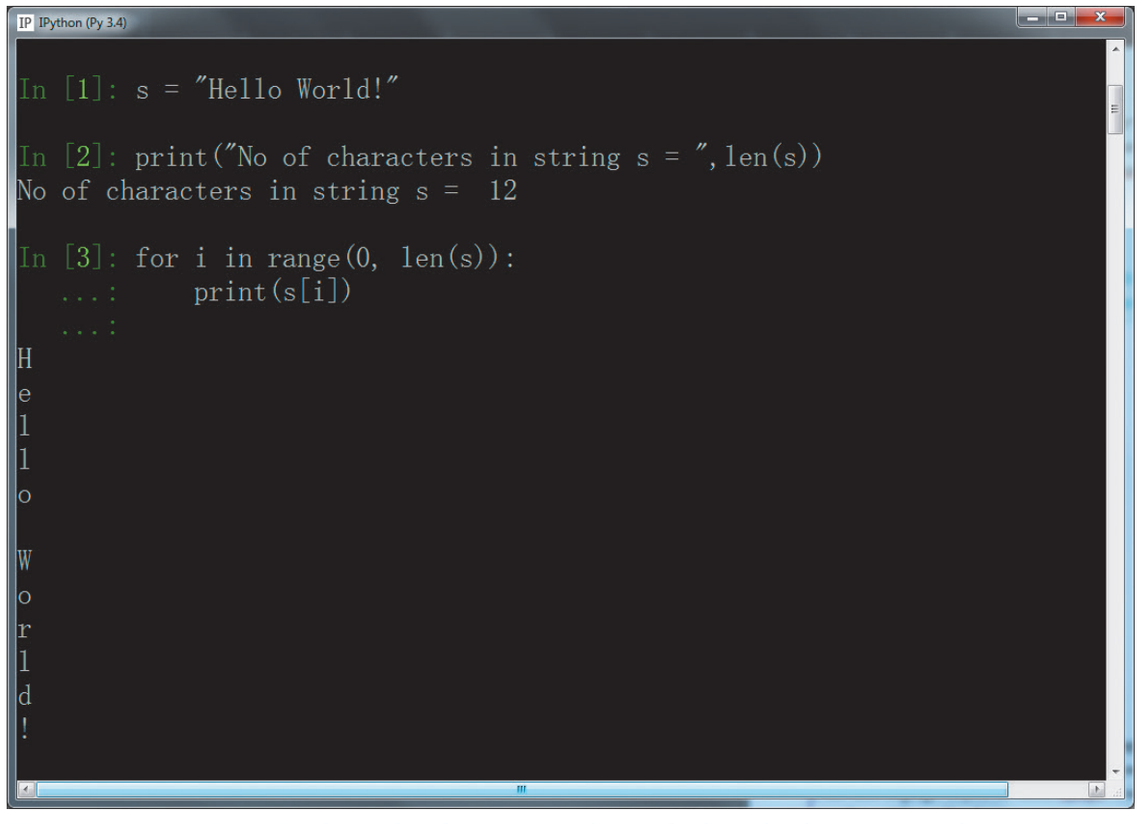


Figure 1.1.7.16 Using Python's len function to obtain the length of a string, s, then iterate through the characters of this zero-based string.

Single licence - Abingdon School

Figure 1.1.7.17 shows that the use of the Pascal/Delphi `Length` function applied to a one-based string variable `s`, as follows: `Length(s)`, returns the number of characters in the string value "Hello World!" which variable `s` contains.

```
Program StringLengthOneBased;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils;
Var
  s : String;
  i : Integer;
Begin
  s := 'Hello World!';
  Writeln(s);
  Writeln('No of characters in string s = ', Length(s));
  Writeln('Character at index 1 = ', s[1]);
  For i := 1 To Length(s)
    Do Writeln(s[i]);
  Readln;
End.
```

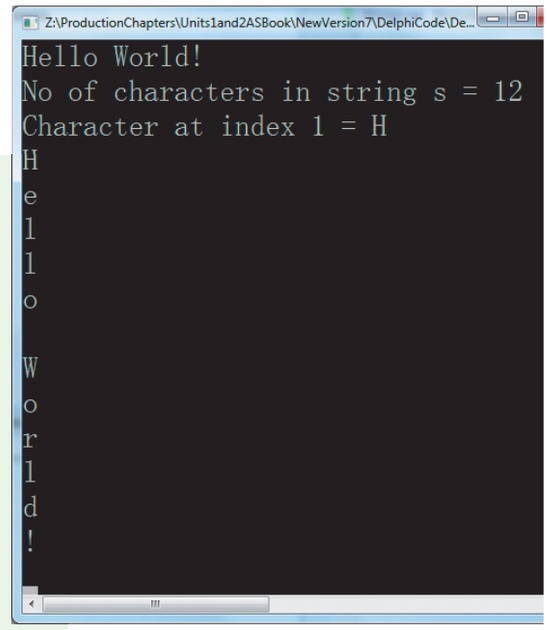


Figure 1.1.7.17 Using Pascal/Delphi's Length function to obtain the length of a string, s, then iterate through the characters of this one-based string.

Character → character code

Sometimes an operation needs to be carried out on a single character value, e.g. 'A', or a variable of character data type, e.g. `Ch`.

In Pascal and Delphi, the `Char` data type is used to create a single character variable and the `Ord` function converts a character value to its character code as shown in *Figure 1.1.7.18*.

```
Program CharacterExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils;
Var
  Ch : Char;
Begin
  Ch := 'A';
  Writeln(Ch);
  Writeln(Ord(Ch));
  Readln;
End.
```

Figure 1.1.7.18 The Char data type and the Ord function in Pascal/Delphi which converts a character to its character code.

In VB.NET, a character variable is declared using the `Char` data type as shown in *Figure 1.1.7.19*. The `Asc` function returns the character code for a given character value.

```
Module CharacterExample
    Sub Main()
        Dim ch As Char = "A"
        Console.WriteLine(ch)
        Console.WriteLine(Asc(ch))
        Console.ReadLine()
    End Sub
End Module
```

Figure 1.1.7.19 The Char data type and the Asc function in VB.NET.

Single licence - Abingdon School

In Java, a character variable is declared using the `char` data type and to convert a character value to its character code in Java, data type casting is used as shown in *Figure 1.1.7.20* where `(int) ch` casts the value contained in `ch` to a value of type `int`.

```
public class CharacterExample {
    public static void main(String[] args) {
        char ch = 'A';
        System.out.println(ch);
        System.out.println((int) ch);
    }
```

causes data type cast from char to int

Figure 1.1.7.20 The char data type in Java and the use of data type casting, (int) to convert a character to its character code.

The `char` keyword is used in C# to declare an instance of the `System.Char` structure that the .NET Framework uses to represent a Unicode character. The value of a Char object is a 16-bit numeric (ordinal) value.

To convert a character to its character code in C#, data type casting is used as shown in *Figure 1.1.7.21*. *Figure 1.1.7.22* shows the use of the `ord` function in Python 3.4. to convert a character to its character code.

```
using System;
namespace CharExample
{
    class Program
    {
        static void Main(string[] args)
        {
            char ch = 'A';
            Console.WriteLine(ch);
            Console.WriteLine((int) ch);
            Console.ReadLine();
        }
    }
}
```

causes data type cast from char to int

Figure 1.1.7.21 The char data type in C# and the use of data type casting, (int) to convert a character to its character code.

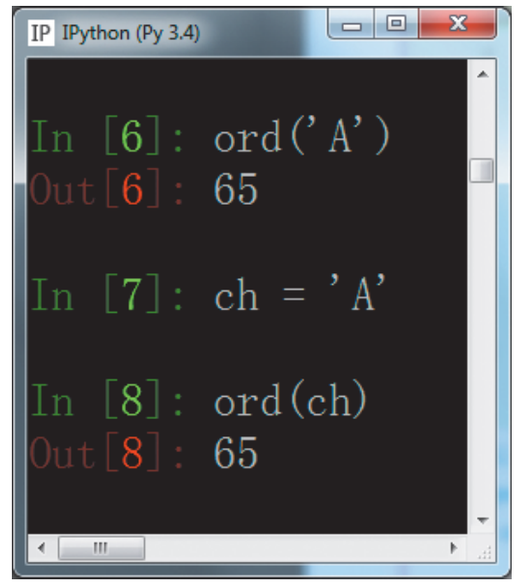


Figure 1.1.7.22 Using the ord function in Python 3.4 to convert a character to its character code.

Character code → character

Table 1.1.7.2 shows how to convert from character code to character in C#, Java, Pascal/Delphi, Python and VB.NET and display the result in the console window. C# and Java use data type casting whilst Pascal/Delphi and VB.NET use a function, `Chr`, and Python a function `chr`.

Language	Code
C#	`Console.WriteLine((char)65);`
Java	`System.out.println((char)65);`
Pascal/Delphi	`Writeln(Chr(65));`
Python	`print(chr(65))`
VB.Net	`Console.WriteLine(Chr(65))`

Table 1.1.7.2 Code to convert character code to character in C#, Java, Pascal/Delphi, Python and VB.NET and display the result in the console window.

Single licence - Abingdon School

String conversion operations

String to integer

C#

A string can be converted to a number using methods in the `Convert` class or by using the `TryParse` method found on the various numeric types (int, long, float, etc). `Convert.ToInt32` converts an integer written in string form, e.g. "-125" to a 32-bit integer value, e.g. `-125`.

```
Console.WriteLine("String -125 has integer value {0}",  Convert.ToInt32("-125"));
```

There are also other methods that may be used when converting a string representing a numeric value:

- `Parse:` If the string is not in a valid format, `Parse` throws an exception. `Int32.Parse("-125")` returns the 32-bit integer value -125. *Table 1.1.7.3* shows an example of this.
- `TryParse`: In the example in *Table 1.1.7.3*, `TryParse` returns true if the conversion succeeded, storing the result in `anotherNumber`, and false if it fails.

Both methods ignore *whitespace* at the beginning and at the end of the string, but all other characters must be characters that form the appropriate numeric type (int, long, ulong, float, decimal, etc). Any *whitespace* within the characters that form the number cause an error.

Java

The `Integer.parseInt(String s)` static method parses the string argument `s` as a signed decimal integer and returns an `int` value as shown in *Table 1.1.7.3*. The resulting value is not an instance of Java's Integer class but just a primitive `int` value.

The `Integer.valueOf(String s)` static method will return an `Integer` object holding the value of the specified `String s` argument.

VB.NET

A string can be converted to a number using methods in the `Convert` class or by using the `TryParse` method found on the various numeric types (int, long, float, etcetera). `Convert.ToInt32` converts an integer written in string form, e.g. "-125" to a 32-bit integer value, e.g. `-125`.

```
Console.WriteLine("String -125 has integer value {0}",  Convert.ToInt32("-125"))
```

There are also other methods that may be used when converting a string representing a numeric value:

- `Parse:` If the string is not in a valid format, `Parse` throws an exception. `Int32.Parse("-125")` returns the 32-bit integer value -125. *Table 1.1.7.3* shows an example of this.
- `TryParse`: In the example in *Table 1.1.7.3*, `TryParse` returns true if the conversion succeeded, storing the result in `anotherNumber`, and false if it fails.

Python

The Python standard built-in function `int()` converts a string into an integer value. It is called with an argument which is the string form of an integer. It returns the integer that corresponds to the string form of the integer.

Table 1.1.7.3 shows an example of the use of `int()`.

Pascal

The `StrToInt` function converts an Integer string such as '-125' to an integer as shown in *Table 1.1.7.3*.

Delphi

The `StrToInt` function converts an Integer string such as '-125' to an integer.

Single licence - Abingdon School

It is also possible to use `Parse` and `TryParse` as follows:

```
AnotherNumber := System.Int32.Parse('-125');
Writeln(AnotherNumber);
If System.Int32.TryParse('-125', YetAnotherNumber)
  Then Writeln(YetAnotherNumber)
  Else Writeln('String could not be parsed');
```

Table 1.1.7.3 shows examples of the use of each.

Language	Code

C#	`Console.WriteLine("String -125 has integer value {0}",`
	`  Convert.ToInt32("-125"));`
	`int number = Int32.Parse("-125");`
	`Console.WriteLine(number);`
	`int anotherNumber;`
	`if (Int32.TryParse("-125", out anotherNumber))`
	`  Console.WriteLine(anotherNumber);`
	`else Console.WriteLine("String could not be parsed.");`

Java	`int number = Integer.parseInt("-125");`
	`System.out.println("The number is: " + number);`
	`int anotherNumber = Integer.valueOf("-125");`
	`System.out.println("The number is: " + anotherNumber);`

Pascal	`Number := StrToInt('-125');` `Writeln(Number);`

Delphi	`Var Result, YetAnotherNumber : Integer;`
	`.....`
	`Number := StrToInt('-125');`
	`Writeln(Number);`
	`AnotherNumber := System.Int32.Parse('-125');`
	`Writeln(AnotherNumber);`
	`If System.Int32.TryParse('-125', YetAnotherNumber)`
	`  Then Writeln(YetAnotherNumber)`
	`  Else Writeln('String could not be parsed');`

Python	`print(int("-125"))`

VB.NET	`Console.WriteLine("String -125 has integer value {0}",`
	`  Convert.ToInt32("-125"))`
	`Dim number As Integer = Int32.Parse("-125")`
	`Console.WriteLine(number)`
	`Dim anotherNumber As Integer`
	`If Int32.TryParse("-125", anotherNumber) Then`
	`  Console.WriteLine(anotherNumber)`
	`Else Console.WriteLine("String could not be parsed")`
	`End If`

Table 1.1.7.3 Code examples in C#, Java, Pascal, Delphi, Python and VB.NET which demonstrate how to convert an integer in string form to an integer value

Single licence - Abingdon School

String to float

C#

A string can be converted to a number using methods in the `Convert` class or by using the `TryParse` method found on the various numeric types (int, long, float, etc). `Convert.ToSingle` converts a number written in string form, e.g. "-125.5" to a single precision floating point value, e.g. `-125.5`. See below and *Table 1.1.7.4*.

```
Console.WriteLine("String -125.5 has float value {0}", Convert.ToSingle("-125.5"));
```

The precision of a single floating point number is 7 decimal digits.

There are also other methods that may be used when converting a string representing a numeric value:

- `Parse:` If the string is not in a valid format, `Parse` throws an exception. `float.Parse("-125.5")` returns a single precision floating point value -125.5. *Table 1.1.7.4* shows an example of this.
- `TryParse`: In the example in *Table 1.1.7.4*, `TryParse` returns true if the conversion succeeded, storing the result in `anotherNumber`, and false if it fails.

Both methods ignore *whitespace* at the beginning and at the end of the string, but all other characters must be characters that form the appropriate numeric type (int, long, ulong, float, decimal, etc). Any *whitespace* within the characters that form the number cause an error.

Java

The `Float.parseFloat(String s)` static method parses the string argument `s` as a signed number and returns a floating point value as shown in *Table 1.1.7.4*.

The `Float.valueOf(String s)` static method will returns a `Float` object holding the float value represented by the argument string `s`.

VB.NET

A string can be converted to a number using methods in the `Convert` class or by using the `TryParse` method found on the various numeric types (int, long, float, etcetera). `Convert.ToSingle` converts a number written in string form, e.g. "-125.5" to a single precision floating point value, e.g. `-125.5`. See below and *Table 1.1.7.4*.

```
Console.WriteLine("String -125.5 has float value {0}", Convert.ToSingle("-125.5"))
```

The precision of a single floating point number is 7 decimal digits.

There are also other methods that may be used when converting a string representing a numeric value:

- `Parse:` If the string is not in a valid format, `Parse` throws an exception. `float.Parse("-125.5")` returns a single precision floating point value -125.5. *Table 1.1.7.4* shows an example of this.
- `TryParse`: In the example in *Table 1.1.7.4*, `TryParse` returns true if the conversion succeeded, storing the result in `anotherNumber`, and false if it fails.

Both methods ignore *whitespace* at the beginning and at the end of the string, but all other characters must be characters that form the appropriate numeric type (int, long, ulong, float, decimal, etc). Any *whitespace* within the characters that form the number cause an error.

Single licence - Abingdon School

Language	Code

C#	`Console.WriteLine("String -125.5 has float value {0}",`
	`  Convert.ToSingle("-125.5"));`
	`float number = float.Parse("-125.5");`
	`Console.WriteLine(number);`
	`float anotherNumber;`
	`if (float.TryParse("-125.5", out anotherNumber))`
	`  Console.WriteLine(anotherNumber);`
	`else Console.WriteLine("String could not be parsed.");`

Java	`float number = Float.parseFloat("-125.5");`
	`System.out.println("The number is: " + number);`
	`float anotherNumber = Float.valueOf("-125.5");`
	`System.out.println("The number is: " + anotherNumber);`

Pascal	`Writeln(StrToFloat('-125.5'):8:2);`

Delphi	`Var Result, YetAnotherNumber : Single;`
	`.....`
	`Writeln(StrToFloat('-125.5'):8:2);`
	`AnotherNumber := System.Single.Parse('-125.5');`
	`Writeln(AnotherNumber:8:2);`
	`If System.Single.TryParse('-125.5', YetAnotherNumber)`
	`  Then Writeln(YetAnotherNumber:8:2)`
	`  Else Writeln('String could not be parsed');`

Python	`print(float("-125.5"))`

VB.NET	`Console.WriteLine("String -125.5 has float value {0}",`
	`  Convert.ToSingle("-125.5"))`
	`Dim Number As Single = Single.Parse("-125.5")`
	`Console.WriteLine(Number)`
	`Dim AnotherNumber As Single`
	`If (Single.TryParse("-125.5", AnotherNumber)) Then`
	`  Console.WriteLine(AnotherNumber)`
	`Else Console.WriteLine("String could not be parsed.")`
	`End If`

Table 1.1.7.4 Code examples in C#, Java, Pascal, Delphi, Python and VB.NET which demonstrate how to convert a number in string form to a floating point value

Python

The Python standard built-in function `float()` converts a string into a floating point value. It is called with an argument which is the string form of a number It returns the floating point value that corresponds to the string form of the number.

Table 1.1.7.4 shows an example of the use of `float()`.

Pascal

The `StrToFloat` function converts a number string such as '-125.5' to a floating point value as shown in *Table 1.1.7.4*.

Single licence - Abingdon School

Delphi

The `StrToFloat` function converts a number string such as '-125.5' to a floating point value.
It is also possible to use `Parse` and `TryParse` as follows:

```
AnotherNumber := System.Single.Parse('-125.5');
Writeln(AnotherNumber);
If System.Single.TryParse('-125.5', YetAnotherNumber)
  Then Writeln(YetAnotherNumber)
  Else Writeln('String could not be parsed');
```

where `AnotherNumber` and `YetAnotherNumber` are single precision floating point variables declared as follows

```
Var
  AnotherNumber, YetAnotherNumber : Single;
```

Table 1.1.7.4 shows examples of the use of each. To use double precision substitute `Double` for `Single` in the code above.

Integer to string

C#

An integer can be converted to its equivalent string form using `Convert.ToString`, e.g. -125 to its string representation, e.g. `"-125"`. See below and *Table 1.1.7.5*.

```
Console.WriteLine(Convert.ToString(-125));
```

Java

The Integer class has a static method that returns a `String` object representing the specified `int` parameter, e.g.

```
Integer.toString(-125));
```

as shown in *Table 1.1.7.5*.

VB.NET

An integer can be converted to its equivalent string form using `Convert.ToString`, e.g. -125 to " -125" as shown below and in *Table 1.1.7.5*.

```
Console.WriteLine(Convert.ToString(-125))
```

Python

The Python standard built-in function `str()` converts a number to its equivalent string form. It is called with a number argument, e.g. -125 and returns the equivalent string form "-125" as shown in *Table 1.1.7.5*.

Pascal/Delphi

The `IntToStr` function converts an integer such as -125. to its equivalent string form '-125' as shown in *Table 1.1.7.5*.

Language	Code
C#	`Console.WriteLine(Convert.ToString(-125));`

Java	`System.out.println(Integer.toString(-125));`
	`String numberString = String.valueOf(-125);`
	`System.out.println(numberString);`

Pascal/Delphi	`Var`
	` NumberString : String;`
	`.....`
	`NumberString := IntToStr(-125);`
	`Writeln(NumberString);`

Python	`print(str(-125))`
VB.NET	`Console.WriteLine(Convert.ToString(-125))`

Table 1.1.7.5 Code examples in C#, Java, Pascal, Delphi, Python and VB.NET which demonstrate how to convert an integer to an equivalent string form

Single licence - Abingdon School

Float to string

C#

A number stored in floating point form can be converted to its equivalent string form using `Convert.ToString`, e.g. -125.5 to its string representation, e.g. `"-125.5"`. See below and *Table 1.1.7.6*.

```
Console.WriteLine(Convert.ToString(-125.5));
```

Java

The `Float` and `Double` classes each have a static method that returns a `String` object representing the specified `Float` or `Double` parameter, e.g.

```
System.out.println(Float.toString(-125.5f));
System.out.println(Double.toString(-125.5));
```

as shown in *Table 1.1.7.6*.

Java has IEEE 754 single and double precision types supported by keywords:

```
float f = -125.5f;  // 32 bit float, note f suffix
double d = -125.5d; // 64 bit float, suffix d is optional
```

Alternatively, we may use `String.valueOf(float f)`. If a float value, e.g. -125.5, is passed to this method as an argument, then the string representation of -125.5 is returned, i.e. "-125.5".

```
float number = -125.5f;
String numberString = String.valueOf(number);
```

VB.NET

A number stored in floating point form can be converted to its equivalent string form using `Convert.ToString`, e.g. -125.5 to " -125.5" as shown below and in *Table 1.1.7.6*.

```
Console.WriteLine(Convert.ToString(-125.5))
```

Python

The Python standard built-in function `str()` converts a number with a fractional part to its equivalent string form. It is called with a number argument, e.g. -125.5 and it returns the equivalent string form "-125.5" as shown in *Table 1.1.7.6*.

Pascal/Delphi

The `FloatToStr` function converts a number stored in floating point form such as -125.5 to its equivalent string form '-125.5' as shown in *Table 1.1.7.6*.

Language	Code
C#	`Console.WriteLine(Convert.ToString(-125.5));`

Java	`System.out.println(Float.toString(-125.5f));`
	`System.out.println(Double.toString(-125.5));`
	`float number1 = -125.5f;`
	`String numberString1 = String.valueOf(number1);`
	`System.out.println(numberString1);`
	`double number2 = -125.5;`
	`String numberString2 = String.valueOf(number2);`
	`System.out.println(numberString2);`

Pascal/Delphi	`Var`
	`  NumberString : String;`
	`.....`
	`NumberString := FloatToStr(-125.5);`
	`Writeln(NumberString);`

Python	`print(str(-125.5))`
VB.NET	`Console.WriteLine(Convert.ToString(-125.5))`

Table 1.1.7.6 Code examples in C#, Java, Pascal, Delphi, Python and VB.NET which demonstrate how to convert a floating point value to an equivalent string form

Single licence - Abingdon School

Date/time to string

Java

Java provides the `Date` class in the `java.util` package. This class encapsulates the current date and time.
The `Date` class is imported with the statement

```
import java.util.Date;
```

An object, `date` of the `Date` class is constructed as follows

```
Date date = new Date();
```

and then used as follows with the Java `toString()` and `println` methods to display the current date and time

```
System.out.println(date.toString());
```

Alternatively, the `java.lang.String.format` method which returns a formatted string, using the specified format string and arguments, can be used, e.g.

```
String dateString = String.format("Current Date and time: %tc", date );
```

`%tc` is a place holder for the `date` argument. A two-letter format is used starting with `t` and ending in a letter that specifies the actual format. For example, the letter "`c`" means display *complete date and time*.

It is possible to invoke the method `printf` on the standard output pipe, `System.out`, to format the output according to a given format string and `date` object argument as follows

```
dateString = String.format("Current Time: %tT", date );
System.out.printf(dateString);
```

where `%tT` sets the format of the string output to be time in 24-hour format.

```
dateString = String.format("%1$tB %1$td, %1$tY", date);
System.out.printf(dateString);
```

In the format string, "`%1$tB %1$td, %1$tY`", `%1` specifies which argument to insert, in this case, `date`.
In the part of the format string, "`$tB $td, $tY`":

- uppercase "`B`" means use the long month name format, e.g. December.
- lowercase "`d`" means use a day number, e.g. 23
- "`,`" means insert a comma
- uppercase "Y" means use long year format, e.g. 2016.

C#

```
new DateTime(1550, 7, 21, 13, 31, 17)
```

initializes a new instance of the `DateTime` structure to the specified year, month, day, hour, minute, and second, e.g. year = 1550, month = 7, day = 21, hour of day = 13, minute of day = 31, second of day = 17.

The instance can be assigned to a variable `dateOld` of type DateTime as follows

```
DateTime dateOld = new DateTime(1550, 7, 21, 13, 31, 17);
```

The date part of this structure can be displayed using the `ToString` method and the format string argument "`d/M/yyyy`" as follows

```
Console.WriteLine(dateOld.ToString("d/M/yyyy"));
```

In the format string argument, "`d/M/yyyy`",

- "`d`" means display day of the month, in numeric form, 1 through 31
- "`m`" means display month, in numeric form, 1 through 12
- "yyyy" means display the year as a four-digit number
- "/" means display /

Table 1.1.7.7 shows examples in the languages Java and C#.

Single licence - Abingdon School

Language	Code

Java	`Date date = new Date();`
	`System.out.println(date.toString());`
	`String dateString = String.format("Current Date and time: %tc", date );`
	`System.out.printf(dateString);`
	`System.out.println();`
	`dateString = String.format("Current Time: %tT", date );`
	`System.out.printf(dateString);`
	`System.out.println();`
	`dateString = String.format("%1$tB %1$td, %1$tY", date);`
	`System.out.printf(dateString);`

C#	`DateTime oldDateTime = new DateTime(1550, 7, 21, 13, 31, 17);`
	`Console.WriteLine(oldDateTime);`
	`Console.WriteLine(oldDateTime.ToString("d/M/yyyy"));`
	`Console.WriteLine(oldDateTime.ToString("H:m:s"));`
	`String oldDate = String.Format("{0:d/M/yyyy}", oldDateTime);`
	`String oldTime = String.Format("{0:H:m:s}", oldDateTime);`
	`Console.WriteLine(oldDate);`
	`Console.WriteLine(oldTime);`
	`DateTime localDateTime = DateTime.Now;`
	`Console.WriteLine(localDateTime.ToString("d/M/yyyy H:m:s"));`

Table 1.1.7.7 Code examples in Java and C# which demonstrate how to convert a date/time value to an equivalent string form

For example, if `DateTime(2016, 11, 3, 15, 21, 45)` is used then the date 3/11/2016 is displayed if the format string argument is `"d/M/yyyy"`.

The time part of this structure can be displayed using the format string argument `"H:m:s"` with the `ToString` method as follows

```
Console.WriteLine(dateOld.ToString("H:m:s"));
```

In the format string, `"H:m:s"`

- `"H"` means display the hour, using a 24-hour clock from 0 to 23.
- `"m"` means display the minute, in numeric form, 0 through 59.
- `"s"` means display the second, from 0 through 59
- `":"` means display :

For example, if `DateTime(2016, 11, 3, 15, 21, 45)` is used then the time 15:21:45 is displayed if the format string argument is `"H:m:s"`.

There is another way to format string output which uses `String.Format` to convert the value of an object to a string based on the format specified, e.g. `"d/M/yyyy"`.
The following shows how a string of the desired format derived from `oldDateTime` may be assigned to variables `oldDate` and `oldTime` of type `String`:

```
String oldDate = String.Format("{0:d/M/yyyy}", oldDateTime);
String oldTime = String.Format("{0:H:m:s}", oldDateTime);
```

`0:` is the first argument and `0:d/M/yyyy` means apply format string `0:d/M/yyyy` to this argument, i.e. `oldDateTime`.
The two strings may then be displayed using

```
Console.WriteLine(oldDate);
Console.WriteLine(oldTime);
```


Single licence - Abingdon School

It is also possible to access the operating system for the current time and date.
For this a static property, `DateTime.Now`, of the `DateTime` structure may be used as follows

```
DateTime localDate = DateTime.Now;
```

and both date and time displayed as follows

```
Console.WriteLine(localDate.ToString("d/M/yyyy H:m:s"));
```

Static property means that no instance of an object needs to be created to use the property `DateTime.Now`, it is available globally because it is actually a property of an existing object called `System.Object`.

VB.NET

Table 1.1.7.8 shows examples for VB.NET which is similar to the C# examples because both C# and VB.NET rely on .NET classes.

Pascal/Delphi

Date and time processing in Pascal and Delphi depend on the `TDateTime` type. A variable of the `TDateTime` type can contain a date and time combination, e.g. `OldDateTime` as shown in *Table 1.1.7.8*.

A date and a time are encoded in a structure using `EncodeDateTime` and the specified year, month, day, hour, minute, second and miilisecond, e.g. year = `1550`, month = `7`, day = `21`, hour of day = `13`, minute of day = `31`, second of day = `17`, millisecond of day = `0` as shown in *Table 1.1.7.8*.

The date part of this structure can be obtained and assigned using

```
OldDate := DateOf(OldDateTime);
```

The time part of this structure can be obtained and assigned using

```
OldTime := TimeOf(OldDateTime);
```

The string form of `OldDateTime`, `OldDate` and `OldTime` are obtained using `DateTimeToStr`, `DateToStr`, and `TimeToStr`, respectively.

To display the result on the console for each case, `Writeln` can be used as follows

```
Writeln(DateTimeToStr(OldDateTime));
Writeln(DateToStr(OldDate));
Writeln(TimeToStr(OldTime));
```

Formatted output can be achieved by applying the procedure `DateTimeToString` to a `TDateTime` variable, e.g. `OldDateTime`, and a format string, e.g. `'d/m/yyyy'`, as follows

```
DateTimeToString(OldDateString, 'd/m/yyyy', OldDateTime);
Writeln(OldDateString);
DateTimeToString(OldTimeString, 'h:n:s', OldDateTime);
Writeln(OldTimeString);
```

The procedure returns a formatted string in the variable in the first position in the parameter list, e.g. `OldDateString`.

It is also possible to access the operating system for the current time and date.
The function `Now` is used to obtain the current date and time as follows

```
LocalDateTime := Now;
Writeln(DateTimeToStr(LocalDateTime));
```

The `datetime` module in Python provides a number of types to deal with dates, times, and time intervals.
To use this module it needs to be imported as follows

```
import datetime
```


Single licence - Abingdon School

Language	Code

VB.NET	`Dim oldDateTime As DateTime = New DateTime(1550, 7, 21, 13, 31, 17)`
	`Console.WriteLine(oldDateTime)`
	`Console.WriteLine(oldDateTime.ToString("d/M/yyyy"))`
	`Console.WriteLine(oldDateTime.ToString("H:m:s"))`
	`Dim oldDate As String = String.Format("{0:d/M/yyyy}", oldDateTime)`
	`Dim oldTime As String = String.Format("{0:H/m/s}", oldDateTime)`
	`Console.WriteLine(oldDate)`
	`Console.WriteLine(oldTime)`
	`Dim localDateTime As DateTime = DateTime.Now`
	`Console.WriteLine(localDateTime.ToString("d/M/yyyy H:m:s"))`

Pascal/ Delphi	`Var`
	`  OldDateTime, LocalDateTime : TDateTime;`
	`  OldDate : TDate;`
	`  OldTime : TTime;`
	`  OldDateString, OldTimeString : String;`
	`Begin`
	`  OldDateTime := EncodeDateTime(1550, 7, 21, 13, 31, 17, 0);`
	`  Writeln(DateTimeToStr(OldDateTime));`
	`  OldDate := DateOf(OldDateTime);`
	`  Writeln(DateToStr(OldDate));`
	`  OldTime :=  TimeOf(OldDateTime);`
	`  Writeln(TimeToStr(OldTime));`
	`  DateTimeToString(OldDateString, 'd/m/yyyy', OldDateTime);`
	`  Writeln(OldDateString);`
	`  DateTimeToString(OldTimeString, 'h:n:s', OldDateTime);`
	`  Writeln(OldTimeString);`
	`  LocalDateTime := Now;`
	`  Writeln(DateTimeToStr(LocalDateTime));`
	`  Readln;`
	`End.`

Python	`import datetime`
	`olddatetime = datetime.datetime(1550, 7, 21, 13, 31, 17)`
	`olddate = olddatetime.date()`
	`oldtime = olddatetime.time()`
	`print(olddatetime)`
	`print(olddate)`
	`print(oldtime)`
	`print(olddatetime.ctime())`
	`print(olddate.strftime("%d/%m/%Y"))`
	`print(oldtime.strftime("%H:%M:%S"))`
	`localdatetime = datetime.datetime.now()`

Table 1.1.7.8 Code examples in VB.NET, Pascal/Delphi, and Python which demonstrate how to convert a date/time value to an equivalent string form

Python

In Python a `datetime` object is a single object containing all the information from a `date` object and a `time` object. As objects are not strings they need to be converted to strings. This can be done with the Python function `str`.

Python's `print` procedure (Python calls it a function) automatically converts `datetime`, `date` and `time` objects to strings before printing to the console window.

The following code creates, initialises and assigns a `datetime` object

```
olddatetime = datetime.datetime(1550, 7, 21, 13, 31, 17)
```

for the specified year, month, day, hour, minute, second, e.g. year = `1550`, month = `7`, day = `21`, hour of day = `13`, minute of day = `31`, second of day = `17`.

Single licence - Abingdon School

The following code extracts and assigns the `date` object within the `datetime` object

```
olddate = olddatetime.date()
```

and the following code extracts and assigns the `date` object within the `datetime` object

```
oldtime = olddatetime.time()
```

These may be displayed on a console using the following code

```
print(olddatetime)
print(olddate)
print(oldtime)
```

The `strftime` method can be used in the following manner to format `olddate` and `oldtime` when these are printed

```
print(olddate.strftime("%d/%m/%Y"))
print(oldtime.strftime("%H:%M:%S"))
```

It is also possible to access the operating system for the current time and date.
The `now` method is used as follows to obtain the current date and time

```
localdatetime = datetime.datetime.now()
```

String to date/time

Table 1.1.7.9 shows how to convert a date and a time in string form to an equivalent date/time type form in Java and C#.

Language	Code

Java	`import java.text.DateFormat;`
	`import java.text.SimpleDateFormat;`
	`import java.util.Date;`
	`public class StringToDateConversion {`
	`  public static void main(String[] args) {`
	`  String dateString = "14/09/2016";`
	`  DateFormat df = new SimpleDateFormat("dd/MM/yyyy");`
	`  Date equivalentDate;`
	`  try`
	`  {equivalentDate = df.parse(dateString);`
	`  System.out.println("Formatted date: " + equivalentDate);`
	`  }`
	`  catch (Exception e)`
	`  {System.out.println("Something has gone wrong: " + e);}`
	`  }`
	`}`
	---------- 	
	`import java.text.DateFormat;`
	`import java.text.SimpleDateFormat;`
	`import java.util.Date;`
	`public class StringToTimeConversion {`
	`  public static void main(String[] args) {`
	`  String timeString = "12:30:45 am";`
	`  DateFormat tf = new SimpleDateFormat("hh:mm:ss a");`
	`  try`
	`  {Date equivalentTime = tf.parse(timeString);`
	`  System.out.println("Formatted time: " + equivalentTime);`
	`  }`
	`  catch (Exception e)`
	`  {System.out.println("Something has gone wrong: " + e);}`
	`  }`
	`}`

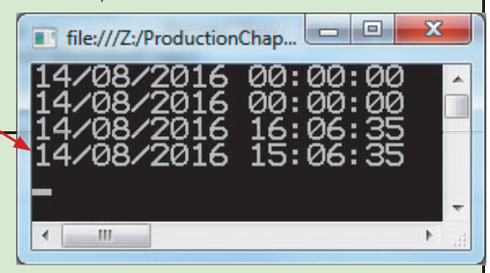
C#	`string date = "14/08/2016";`
	`DateTime d = Convert.ToDateTime(date);`
	`Console.WriteLine("Day: {0}, Month: {1}, Year: {2}", d.Day, d.Month, d.Year);`
	`string dateTime = "14/08/2016 12:30:45.15";`
	`DateTime dt = Convert.ToDateTime(dateTime);`
	`Console.WriteLine("Day: {0}, Month: {1}, Year: {2}, Hour: {3}, Minute: {4},`
	`Second: {5}, Millisecond: {6}",dt.Day, dt.Month, dt.Year, dt.Hour, dt.Minute,`
	`dt.Second, dt.Millisecond);`

Table 1.1.7.9 Code examples in Java and C# which demonstrate how to convert a date/time in string form into an equivalent date/time form

Single licence - Abingdon School

Table 1.1.7.10 shows how to convert a date and a time in string form to an equivalent date/time type form in VB.NET, Pascal/Delphi and Python.

Language	Code

VB.NET	`Module Module1`
	`  Sub Main()`
	`  Dim aDate As DateTime = DateTime.Parse("14/08/2016")`
	`  Console.WriteLine(aDate)`
	`  aDate = DateTime.Parse("Sunday, August 14, 2016")`
	`  Console.WriteLine(aDate)`
	`  Dim aDateAndTime = DateTime.Parse("Sun, 14 Aug 2016 15:06:35 GMT")`
	`  Console.WriteLine(aDateAndTime)`
	`  aDateAndTime = DateTime.Parse("14/08/2016 15:06:35")`
	`  Console.WriteLine(aDateAndTime)`
	`  Console.ReadLine()`
	`  End Sub`
	`End Module`
	 

Pascal/ Delphi	`Program StringToDateExample;`
	`Uses Sysutils;`
	`Var`
	`  DateString : String;`
	`  DateFromString : TDateTIme;`
	`Begin`
	`  DateString := '14/08/2016';`
	`  DateFromString := StrToDate(DateString);`
	`  Writeln(FormatDateTime('DD MM YYYY', DateFromString));`
	`  Readln;`
	`End.`
	`----------`
	`Program StringToTimeExample;`
	`Uses Sysutils;`
	`Var`
	`  TimeString : String;`
	`  TimeFromString : TDateTIme;`
	`Begin`
	`  TimeString := '15:45:23';`
	`  TimeFromString := StrToTime(TimeString);`
	`  Writeln(FormatDateTime('hh:nn:ss', TimeFromString));`
	`  Readln;`
	`End.`

Python	`import datetime`
	`a_date = datetime.datetime.strptime('14/08/2016', '%d/%m/%Y')`
	`a_time = datetime.datetime.strptime('17:15:50', '%H:%M:%S')`
	`print(a_date)`
	`print(a_time)`

Table 1.1.7.10 Code examples in VB.NET, Pascal/Delphi, and Python which demonstrate how to convert a string representing a date/time into an equivalent date/time data type form

Single licence - Abingdon School

Substring

A substring is a subset of a string between one index and another. For example, the substring 'mit' is a subset of the string 'smith'.

Figure 1.1.7.23 shows two ways to identify a substring of the string by

- Index
- Offset

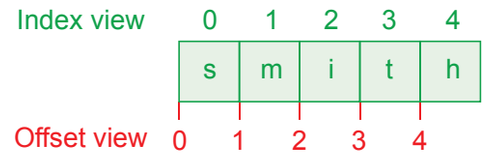


Figure 1.1.7.23 The string 'smith' and two ways to identify a substring of the string, by Index or by Offset

Suppose a subroutine called `Substring` returns a substring of a given string, `s`.

It is called with arguments `s`, `StartIndex` and `EndIndex` as shown in *Table 1.1.7.11* and the substring returned is assigned to another string variable, `SubOfs`.

The `StartIndex` is 1. In the *index view* in *Figure 1.1.7.1*, this is the index of the character 'm' in the string `s`.

The `EndIndex` is 3. In the *index view* in *Figure 1.1.7.1*, this is the index of the character 't' in the string `s`.

The substring returned is between index 1 and 3, inclusive, i.e. substring 'mit'.

```
s ← 'smith'
StartIndex ← 1
EndIndex ← 3
SubOfs ← Substring(s, StartIndex, EndIndex)
Output SubOfs
```

Table 1.1.7.11 Pseudo-code showing a call to Substring

In the *offset view*,

- the first character of the substring, 'm' is reached by an offset of one character from the beginning of the string `s` as shown in *Figure 1.1.7.24* - `StartOffset` is 1.
- the last character of the substring, 't' is included by an offset of four characters from the beginning of the string `s` as shown in *Figure 1.1.7.24* (an offset of 0 is the first character 's' - `EndOffset` is 4. A slice can then occur between 1 and 4 as shown in *Figure 1.1.7.24* to obtain the substring.

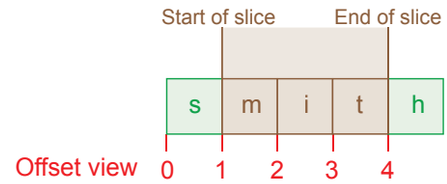


Figure 1.1.7.24 Taking a slice of the string 'smith' between StartOffset = 1 and EndOffset = 4

We can use `StartOfSlice` and `EndOfSlice` instead of `StartOffset` and `EndOffset`, to make it clearer that the selected substring is a slice through the string, i.e. `StartOfSlice = 1`, `EndOfSlice = 4`.

Table 1.1.7.12 shows examples of how a substring can be obtained in Pascal/Delphi, C#, Python, Java and VB.NET.

Python has no substring subroutine. Instead, we use slice syntax to get parts of existing strings.

Single licence - Abingdon School

Pascal/Delphi

```
Program SubstringExample;
{String indexing starts at 1}
Var
  s, SubOfs : String;
  StartIndex, Count : Integer;
Begin
  s := 'smith';
  StartIndex := 2;
  Count := 3;
  SubOfs := Copy(s,StartIndex,Count);
  Writeln(SubOfs);
  Readln;
End.
{Copy treats zero-based strings as if
they are 1-based In Delphi with zero-
based compiler option ON
```

C#

```
using System;
namespace SubstringExample
{
  class Program
  {
    static void Main(string[] args)
    {
      String s = "smith";
      int startIndex = 1;
      int endIndex = 3;
      String subOfs = s.Substring(startIndex,endIndex);
      Console.WriteLine("Substring is {0}",subOfs);
      Console.ReadLine();
    }
  }
}
```

Python

```
s = "smith"
start_offset = 1
end_offset = 4
sub_of_s = s[start_offset:end_offset]
print("Substring is ", sub_of_s)
```

Java

```
public class SubstringExample {
  public static void main(String[] args) {
    String s = new String("smith");
    int startOffset = 1;
    int endOffset = 4;
    String subOfs = s.substring(startOffset,endOffset);
    System.out.println("Substring is " + subOfs);
  }
}
```

VB.Net

```
Module Module1
  Sub Main()
    Dim s As String = "smith"
    Dim startIndex As Integer = 1
    Dim endIndex As Integer = 3
    Dim subOfs = s.Substring(startIndex, endIndex)
    Console.WriteLine("Substring is {0}", subOfs)
    Console.ReadLine()
  End Sub
End Module
```

Table 1.1.7.12 Examples of how a substring can be obtained in Pascal/Delphi, C#, Python, Java and VB.NET

Concatenation

The concatenation of strings is the operation of joining character strings end-to-end.

For example, the concatenation of "john" and "smith" is "johnsmith".

The concatenation of "john" and " " and "smith" is "john smith".

The most common way of concatenating strings is to use the '+' operator.

For example in Java:

```
String a = "Hello";
String b = " World!";
String c = a + b;
System.out.print(c);
```

The concatenation operator '+' is common to all the programming languages covered in this chapter.

Single licence - Abingdon School

Programming tasks

1. Write a program which takes as input a string and prints out the number of characters in it.
2. Write a program which takes as input a word and checks if the first and last letters of the word are the same. The program should print either "The first and last letters are the same" or "The first and last letters are different".
3. Write a program which takes as input 2 words and prints the word made from the last 3 letters of the first word and the first 3 letters of the second word.
4. Write a program which takes as input a sentence and calculates how many words are in the sentence. The program should print this number (with appropriate message). Assume a word ends with a space, a sentence ends with a full stop.
5. Write a program which inputs two words, a master word and a test word. The program should check whether or not the test word appears anywhere in the master word and then print an appropriate message. For example "THE" appears inside the word “STRENGTHEN” but not inside "STEALTH". The program should first check that the test word is shorter than the master word; if not it should print the message "Test word too long" and request a new test word.
6. Write a program which takes as input a pair of words and prints a third word made from the letters at the end of the first word if they are the same as the letters at the beginning of the second word, e.g. IGNORANT and ANTLER are input, the output is ANT, otherwise the program should request another pair of words.
7. Write a program which takes as input a word and prints out a new word made from reversing the order of the letters in the input word, e.g. input word = "BEAR", output word = "RAEB".
8. Write a program which takes as input a pair of words and checks if one is an anagram of the other. The program should output the message "ANAGRAM" if it is and the message "NOT AN ANAGRAM" if it isn't.
9. Write a program which takes as input a word and determines whether or not the word is a palindrome. (A palindrome is a word that reads the same forwards and backwards. E.g. ROTOR.)
   Test your program on the following palindromes DAD, NOON, MADAM, REDDER, ROTOVATOR.
10. Write a program which takes a single word containing only upper case letters of the alphabet as input, and outputs an encrypted version of the word using the following simple encryption algorithm
    Character code for "A" + (((Character code of letter to encrypt - character code for "A") + 13) MOD 26)
11. Write a program to output the current date.
12. Write a program to output the current time.
13. Write a program which takes as input two dates in string form and calculates the number of days between the dates.

Single licence - Abingdon School

Position

Sometimes we may wish to discover if a given substring is present within a given string. For this we use a **position** function which returns the index within the given string of the first occurrence of the substring. If the substring is not present in the given string then a value is returned outside the index range, e.g. -1. *Table 1.1.7.13* shows how this can be done in Pascal/Delphi, C#, Python, Java and VB.NET.

Pascal/ Delphi	`Position := AnsiPos('ello', 'Hello World!');`
	`If Not (Position = 0)`
	`  Then Writeln('String contains ello')`
	`  Else Writeln('Not found');`
		The AnsiPos function finds the position of one string 'ello' within another 'Hello World'.
		If the string is not found, 0 is returned.
		The search is case sensitive.

C#	`string s = "Hello World!";`
	`if (s.IndexOf("ello") != -1)`
	`  {`
	`  Console.WriteLine("String contains ello");`
	`  }`
		Function `IndexOf` returns the index of a substring. First it scans the String. And if the substring is not found, it returns -1.

VB.NET	`Dim s As String = "Hello World!"`
	`If Not s.IndexOf("ello") = -1 Then`
	`  Console.WriteLine("String contains ello")`
		Function `IndexOf` returns the index of a substring. First it scans the String. And if the substring is not found, it returns -1.

Java	`String s = "Hello World!";`
	`if (s.indexOf("ello") >= 0){`
	`  System.out.println(" String contains ello")`
	`}`
		Function indexOf returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.

Python	`str1 = "Hello World!"`
	`str2 = "ello"`
	`if (str1.find(str2) != -1):`
	`  print ("String contains ello")`
		Function find returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.

Table 1.1.7.13 Examples of searching for a substring within a given string in Pascal/Delphi, C#, Python, Java and VB.NET

In this chapter you have covered:

- Using and becoming familiar with:
  - length
  - position
  - substring
  - concatenation
  - character → character code
  - character code → character
  - string conversion routines
    - string to integer
    - string to float
    - integer to string
    - float to string
    - date/time to string
    - string to date/time

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be familiar with, and be able to use, random number generation*

Figure 1.1.8.1 Rolling dice

Did you know?

The history of pseudorandom number generator algorithms began during the Manhattan project in the Second World War when John von Neumann devised the middle-square method of generating pseudorandom number sequences - the method was marked Top Secret initially.
A quick way of generating random numbers was needed for simulations for the nuclear bomb programme.
The middle-square method is flawed because it can go horribly wrong and generate sequences which are not very random but John Neumann was happy to work with this limitation because when things went wrong it was easy to spot.
In 1949, D.H Lehman came up with the linear congruential method, a far superior method still used today in some software systems although many now use the Mersenne Twister PRNG, e.g. Free Pascal and Python.

1.1.8 Random number generation in a programming language

Random number generation

Random numbers

An algorithm which generates a random, or apparently random, sequence of numbers is called a **random number generator**.

For example, suppose we require a method for selecting an integer at random from the set of integers 1, 2, 3, .., N.

For small values of N simple mechanisms exist.
For example:

- for N = 2 we can toss a coin
- for N = 6 we can roll a die - *Figure 1.1.8.1*
- for N = 12 we can roll a 12-sided die
- for N = 36 we can use a roulette wheel.

Every number in the chosen set of integers 1, 2, 3, .., N, is equally likely.

Statistically, each number from the set should appear on average the same number of times in a long sequence generated by coin tossing or dice rolling.

The sequence generated which satisfies these two conditions is called a **random sequence**.

Pseudorandom numbers

Most computer generated random numbers use pseudorandom number generators (PRNGs) which are algorithms that can automatically create long runs of numbers with good random properties but eventually the sequence repeats.

This kind of random number is adequate for many situations, e.g. computer simulations and cryptography.

However, pseudorandom generated number sequences are not as random as coin tosses and dice rolls or random sequences generated from physical phenomena such as electrical noise signals.

Pseudorandom number generator algorithms generate a random sequence which is completely determined by a shorter initial value known as a seed value or key.

As a result, the entire seemingly random sequence can be reproduced if the seed value is known.

Single licence - Abingdon School

For example, in the linear congruential method the next number in the pseudorandom sequence is calculated from its immediate predecessor as follows

next = (multiplier ∗ predecessor + increment) MOD modulus

We can demonstrate this method by hand if we choose small values, e.g.

- multiplier = 13,
- increment = 0,
- modulus = 31

If we start the sequence with predecessor = 1, we get the pseudorandom sequence

1, 13, 14, 27, 10, 6, 16, 22, 7, 29, 5, 3, . . .

The first thirty terms in this sequence are a permutation of the integers from 1 to 30 and then the sequence repeats itself. It has a period equal to modulus − 1.

The starting value is called the **seed**.

If we divide each pseudorandom integer in the sequence by the modulus (31) we obtain numbers which are uniformly distributed between 0 and 1.

The pseudorandom sequence

1, 13, 14, 27, 10, 6, 16, 22, 7, 29, 5, 3, . . .

becomes

0.03226, 0.4194, 0.4516, 0.8710, 0.3226, 0.1935, ...

Information

If we choose the increment to be 0 then the generator is said to be a multiplicative linear congruential generator.
To obtain a maximal cycle with such a generator, we have to ensure the modulus is prime.

Questions

1. Calculate the first seven random numbers using the linear congruential method and multiplier = 7, increment = 0, modulus = 13 and starting value 1.
2. What is the maximum length of sequence in question 1 before the sequence hits 1 again?
3. Explain why, for a given starting value, the sequence of random numbers generated from the linear congruential method in question 1 eventually repeats itself.

Task

1. Set up a spreadsheet to calculate the first 37 random numbers generated by the linear congruential method and multiplier = 7, increment = 0, modulus = 13 and starting value 1.
   What do you observe?

   Change the starting value in the spreadsheet to 6. What do you observe?

Single licence - Abingdon School

Multiplicative linear congruential generator

Stephen Park and Keith Miller proposed a minimal standard random number generator in 1988 in which the increment was 0, the multiplier was 16807 and the modulus was 2147483647 (2^{31} -1).
The modulus was chosen to be as large as possible and at least the word size of operating system e.g. for a 32-bit operating system 32 bits in size.

The multiplier was chosen to be relatively prime to the modulus (two numbers are relatively prime if their greatest common divisor is 1).

Mersenne Twister generator

The Mersenne Twister generator algorithm (1998) based on a Mersenne prime is an alternative to the linear congruential generator algorithm.
The most commonly used version is based on the Mersenne prime $2^{19937}-1$.
It has a period of $2^{19937}-1$, i.e. the maximum sequence length before the sequence repeats is $2^{19937}-1$.
The standard implementation uses a 32-bit word length.

Key term

Mersenne Twister:
The Mersenne Twister generator has a better statistical distribution than a Linear Congruential generator algorithm, but is considerably slower than the latter.

Randomizing the seed

Although it is possible to set the seed manually, using the same seed will generate the same pseudorandom sequence of numbers.
Both linear congruential generators and Mersenne twister generators use algorithms which are **deterministic**, i.e. given the seed value, the algorithm and its parameter values, the sequence is predictable.
Truly random number generators are based on some physical phenomenon which is governed ultimately by quantum mechanics, a nondeterministic mechanics.

However, the choice of seed can be randomized. One method relies upon sampling the computer's system clock another chooses from a small set of truly random numbers.

For example, in Python one would call the procedure `random.seed(None)` to use the system clock to generate the seed for the Mersenne Twister generator. Next, function `random.random()` is called. It returns the next random floating point number in the interval [0.0, 1.0). "[" means 0.0 is included in the range. The symbol ")" means 1.0 is excluded.
For example, in one call to `random.seed(None)` followed by `random.random()`, 0.40488239522745517 was returned.

However, the pseudorandom number generated by the call `random.seed(1)` followed by `random.random()`, produced 0.13436424411240122 as the first number in the sequence every time and the same sequence of numbers on calling `random.random()` again and again.

Random number generators in programming languages

Table 1.1.8.1 shows pseudorandom number generator subroutines for Python, Java, Free Pascal, Delphi, C# and VB.NET. *Table 1.1.8.2* shows how the seed value may be set in Python, Java, Free Pascal, Delphi, C# and VB.NET.

Table 1.1.8.3 shows the pseudorandom generator algorithm which is used for Python, Java, Free Pascal, Delphi, C# and VB.NET.

Single licence - Abingdon School

Language	Pseudorandom number generator subroutine	Explanation

Python	`random.random()`
	
	`random.randint(a, b)`
	
	`random.randint(N)`
		Returns the next random floating point number in the interval [0.0, 1.0)
		Returns a random integer N such that a <= N <= b
		Returns `Random integer` in interval [0,N]

Java	`import java.util.Random`
	`public class PRSG {`
	`  public static void main(String[] args){`
	`  System.out.println(Math.random())`
	`  }`
	`}`
		`Math.random()`
		Returns the next pseudorandom, double value in the interval [0.0, 1.0)

Java	`public class PRSG {`
	`  public static void main(String[] args){`
	`  Random pRSG = new Random();`
	`  System.out.println(pRSG.nextInt(10));`
	`  }`
	`}`
		Returns the next pseudorandom, `int` value in the interval [0, 10)

Free Pascal	`Uses`
	`  SysUtils;`
	`Var`
	`  Hours, Mins, Secs, Millisecs : Word;`
	`Begin`
	`  Randomize;`
	`  WriteLn(Random :20:18);`
	`  DecodeTime(Now, hours, mins, secs, milliSecs);`
	`  RandSeed := milliSecs;`
	`  WriteLn(Random :20:18);`
	`  WriteLn(Random(10));`
		`Randomize` changes the seed used to generate its range of pseudo random numbers.
		The `RandSeed LongInt` variable can be set directly.
		`Random` returns a random `real` number in the interval [0.0, 1.0)
		`Random(N)` returns a random number in `LongInt` integer interval [0, N)

Delphi	`Randomize;`
	`Writeln(Random :20:18);`
	`DecodeTime(Now, hours, mins, secs, milliSecs);`
	`RandSeed := milliSecs;`
	`Writeln(Random :20:18);`
	`Writeln(Random(10));`
		`Random` returns a random real number in range [0.0, 1.0)
		`Random(N)` returns a random number in `LongInt` integer interval [0, N)

C#	`Random n = new Random();`
	`Console.WriteLine("Randomly generated number in range [0.0, 1.0) : {0}", n.NextDouble());`
	`Console.WriteLine("Randomly generated number in range [0, 10) : {0}", n.Next(10));`
		`new Random()` initializes a new instance of the `Random` class, using a time-dependent default seed value.
		`NextDouble()` gets next `double` random no in interval [0.0, 1.0).
		`Next(N)` gets next integer random no in interval [0,N).

VB.NET	`Dim n As New Random()`
	`Console.WriteLine("Randomly generated number in range [0.0, 1.0) : {0}", n.NextDouble())`
	`Console.WriteLine("Randomly generated number in range [0, 10) : {0}", n.Next(10))`
		`new Random()` initializes a new instance of the `Random` class, using a time-dependent default seed value.
		`NextDouble()` gets next `double` random no.
		`Next(N)` gets next integer random no in interval [0,N).

Table 1.1.8.1 shows pseudorandom number generator subroutines for the languages

Single licence - Abingdon School

Language	Seed setting subroutines	Explanantion

Python	`import random`
	`random.seed(None)`
	`random.seed(seedvalue)`
		`random.seed(None)`, None means system clock sets seed
		Seed is `seedvalue`, e.g. 1.

Free Pascal	`Randomize`	Initializes the random number generator by giving a value to Randseed LongInt variable, calculated with the system clock.

Delphi	`Randomize`
	
	`DecodeTime(now, hours, mins, secs, milliSecs);`
	`RandSeed := milliSecs;`
		Randomize changes the seed used to generate its range of 2^{32} pseudo random numbers.
		
		The RandSeed LongInt variable can be set directly.

C#	`Random(N)`	Initializes a new instance of the `Random` class, using the specified seed value N of type `Int32`.
VB.NET	`Random(N)`	Initializes a new instance of the `Random` class, using the specified seed value N of type `Int32`.

Table 1.1.8.2 shows seed setting for Python, Free Pascal, Delphi, C# and VB.NET

Questions

4 What is the role of a seed in the generation of pseudorandom number sequences?

5 A certain online Poker site found that a group of players was winning every time against the computer-generated poker hands. On investigation, it was discovered that the group was familiar with the programming language used to program the poker game and had also worked out how to obtain the uptime of the server.
Explain one way this group could have accurately predicted the computer-generated poker hands?

Language	Pseudorandom number generator algorithm
Python	Mersenne Twister
Free Pascal	Mersenne Twister
Delphi	Multiplicative linear congruential generator
C#	Donald E. Knuth's subtractive random number generator algorithm
VB.NET	Donald E. Knuth's subtractive random number generator algorithm

Table 1.1.8.3 Pseudorandom number generator algorithms used in the languages

Programming tasks

1 Write a program to generate and display 10 random floating point numbers in the interval [0.0, 1.0), i.e. $0.0 \le x < 1.0$.

2 Write a program to generate and display 10 random integers in interval [0, 10), i.e. $0 \le x < 10$.

In this chapter you have covered:

- Using and becoming familiar with random number generation

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be familiar with the concept of exception handling*
- *Know how to use exception handling in a programming language with which students are familiar*

Key term

Exception:
An exception is an unexpected (or at least unusual) condition that arises during program execution, and which cannot be easily handled by the program.

Key concept

Exception handling:
The idea of exception handling is to move error-checking code "out-of-line", so that the source code can express the solution to the normal case without its intent being obscured by error-checking code.
When an exception does occur control branches "out-of-line" to a handler.
A handler performs some operation which, if recovery is possible, allows a program to recover from the exception and continue execution.
If recovery is not possible, a handler can print an informative error message before the program terminates.

1.1.9 Exception handling

The concept of exception handling

What is an exception?

An exception is an unexpected (or at least unusual) condition that arises during program execution, and which cannot be easily handled by the program.

The most common types of exceptions are various kinds of run-time errors such as *arithmetic overflow*, *division by zero*, *end-of-file on input*, *input conversion*, *subscript* and *subrange* errors.

For example, an input routine may find letters in the input when it is expecting digits.

To anticipate every unexpected condition is unrealistic. Attempts to include code to check for multiple error conditions is itself likely to introduce other errors.
It is also likely to obscure the normal control flow of the program, making the program difficult to read, understand and debug.
The resulting source code would resemble a veritable spaghetti junction of nested *If statements*.

Exception handling

Exception handling mechanisms address these issues by moving error-checking code "out-of-line", de-cluttering the source code which expresses the solution to the normal case.
To this source code must be added an arrangement for control to branch to a handler when an exception occurs.

A handler will perform some operation which, if recovery is possible, allows a program to recover from the exception and continue execution.

If recovery is not possible, a handler can print an informative error message before the program terminates.

Single licence - Abingdon School

Using exception handling in a programming language

In most programming languages an exception handler is attached to a statement or to a list of statements.

Delphi/Pascal

Table 1.1.9.1 shows an example in Delphi which catches a division by zero error. The block of statements between *Try* and *Except* are protected.

```
Readln(x);
y := 4 Div x;
Writeln(y);
```

If `yx <> 0` then no exception occurs and the Except block is ignored.

However, if `x = 0` then an exception occurs when execution of `y := 4 Div x` is attempted.

Control is now transferred to the *Except* block to handle the exception. The `Writeln(y)` statement following `y := 4 Div x` is not executed.

Exceptions are usually objects in the object-oriented sense of the word - a value of some class type.

Most languages allow an exception to have parameters so that the code that raises the exception can pass information to the code that handles it.

In *Table 1.1.9.1* when an exception has occurred, an exception object is constructed and passed to the *Except* block.

In the *Except* block, *On E : EDivByZero* checks for a specific class type called *EDivByZero*.

If the class of the exception object matches the class *EDivByZero,* the exception object is assigned to the inline local variable *E*. The exception object has a field called *Message* which contains a pre-defined string, in the case of a divide by zero error it is the string 'Division by zero'.

This field is accessed with *E.Message* in the *Writeln* statement. *Figure 1.1.9.1* shows the output when this program is executed with input 0 for x.

```
Program DivByZeroException;
{$APPTYPE CONSOLE}
Uses
  SysUtils;
Var
  x : Integer;
  y : Integer;
Begin
  Try
    Readln(x);
    y := 4 Div x;
    Writeln(y);
  Except
    On E : EDivByZero
      Do
        Begin
          Writeln('Error: ', E.Message, '; Will recover by setting y to 1');
          y := 1;
        End;
  End;
  Writeln(y);
  Readln;
End.
```

```
Try
  // Do something that may raise an exception
Except
  // Handle any exception
End;
```

Table 1.1.9.1 Example of Exception handling in Delphi

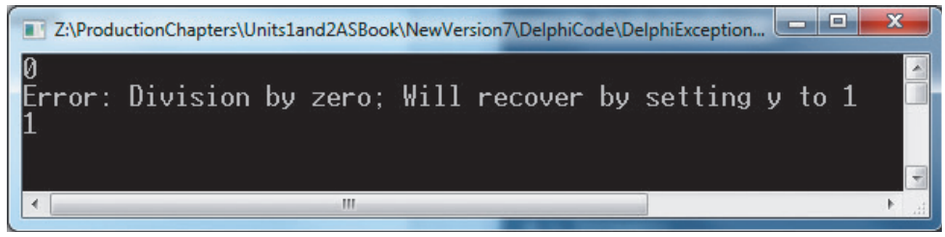


Figure 1.1.9.1 Output from the exception handler when the Delphi program in Table 1.1.9.1 is executed with input 0 for x.

Single licence - Abingdon School

Table 1.1.9.2 shows a Delphi program with an *Except* block which checks for different specific exception types.

```
Program ExceptionHandling;
{$APPTYPE CONSOLE}
Uses SysUtils;
Var
  x, y : Integer;
  Date : TDate;
  DateStr : String;
Begin
  Try
    Readln(x);
    y := 4 Div x;
    Writeln(y);
    Readln(DateStr);
    Date := StrToDate(DateStr);
  Except
    On E:EDivByZero
      Do Writeln('Error 1 ', E.Message);
    On E: EInOutError
      Do Writeln('Error 2 ', E.Message);
    On EConvertError
      Do
        Begin
          DateStr := '01/01/2001';
          Date := StrToDate(DateStr);
        End;
  End;
  Writeln(y);
  Writeln(DateToStr(Date));
  Readln;
End.
```

```
Try
  // Do something that may raise an exception
Except
  On Identifier1 : ExceptionClass1
    Do Begin {Handle ExceptionClass1 exception}; End;
  On Identifier1 : ExceptionClass2
    Do Begin {Handle ExceptionClass2 exception}; End;
  On Identifier1 : ExceptionClass3
    Do Begin {Handle ExceptionClass3 exception}; End;
End;
```

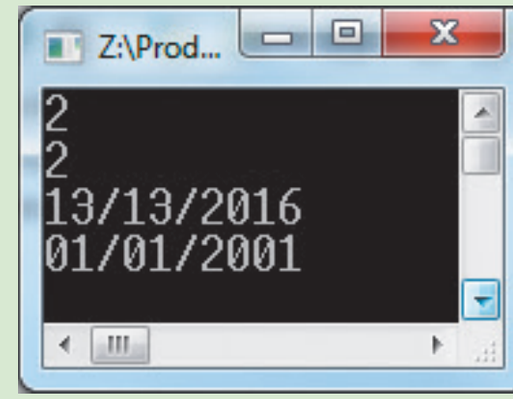


Figure 1.1.9.2 Output from the exception handler when the Delphi program in Table 1.1.9.2 is executed and an EConvertError exception occurs on converting a date string to date format

Table 1.1.9.2 Example of a Delphi program which checks for different specific exception types

VB.NET

Table 1.1.9.3 shows an example of exception handling in VB.NET.

VB.NET uses *Try* ... *Catch* ... *End Try* in a similar way that Delphi uses *Try* ... *Except* ... *End*.

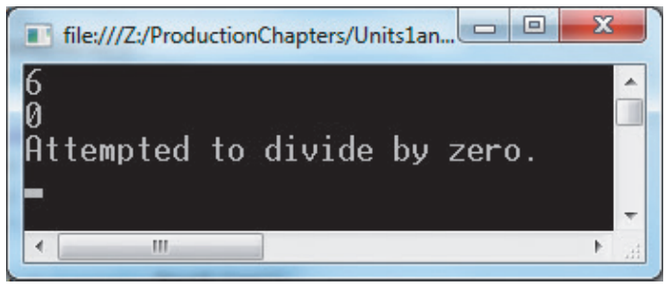


Figure 1.1.9.3 Output from the exception handler when the VB.NET program in Table 1.1.9.3 is executed with input 0 for y.

```
Module Module1
    Sub Main()
        DivideByZero()
        Console.ReadLine()
    End Sub
    Public Sub DivideByZero()
        Dim x As Integer
        Dim y As Integer
        Dim r As Integer = 0
        Try
            x = Console.ReadLine()
            y = Console.ReadLine()
            r = x \ y
            Console.WriteLine(r)
        Catch e As DivideByZeroException
            Console.WriteLine(e.Message)
        End Try
    End Sub
End Module
```

Table 1.1.9.3 Example of exception handling in VB.NET

Single licence - Abingdon School

C#

Table 1.1.9.4 shows an example of exception handling in C#.

C# uses *Try* {...} *Catch* {...} in a similar way that Delphi uses *Try* ... *Except* ... *End*.

```
using System;
namespace DivideByZeroExceptionHandling
{
    class Program
    {
        static void Main(string[] args)
        {
            int x, y, r;
            try
            {
                string input = Console.ReadLine();
                Int32.TryParse(input, out x);
                input = Console.ReadLine();
                Int32.TryParse(input, out y);
                r = x / y;
                Console.WriteLine(r);
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine(e.Message);
            }
            Console.ReadLine();
        }
    }
}
```

Table 1.1.9.4 Example of exception handling in C#

Java

Table 1.1.9.5 shows an example of exception handling in Java.

Java uses *try* {...} *catch* {...} in a similar way that Delphi uses *Try* ... *Except* ... *End*.

```
import java.util.Scanner;
public class DivideByZeroExceptionHandling {
    public static void main(String[] args) {
            int x, y, r;
            try
            {
                Scanner in = new Scanner(System.in);
                x = in.nextInt();
                y = in.nextInt();
                r = x / y;
                System.out.println(r);
            }
            catch (ArithmeticException e)
            {
                System.out.println(e);
            }

    }
}
```

```
6
0
java.lang.ArithmeticException: / by zero
```

Table 1.1.9.5 Example of exception handling in Java

Single licence - Abingdon School

Python

Table 1.1.9.6 shows an example of exception handling in Python.

Python uses *try:* ... *except* ... in a similar way that Delphi uses *Try* ... *Except* ... *End*.

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter a number: "))
    result = x/y
    print(result)
except ZeroDivisionError:
    print("Division by zero detected")
else:
    print("Evaluation of x/y was successful")
```

Table 1.1.9.6 Example of exception handling in Python

Questions

1. What is the meaning of the terms *program exception* and *exception handling* in the context of programming?

Programming Task

1. Investigate exception handling in a programming language with which you are familiar.
   Explore the following:
   (a) Math exceptions
   (b) Input conversion exception caused by input of non-digit characters when character digits expected .

In this chapter you have covered:

- The concept of exception handling
- Using exception handling in a programming language with which students are familiar

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be familiar with subroutines and their uses*
- *Know that a subroutine is a named 'out of line' block of code that may be executed (called) by simply writing its name in a program statement*
- *Be able to explain the advantages of using subroutines in programs*

1.1.10 Subroutines (procedures/functions)

Information

Ord:
Ord applied to a character returns its character code.
Chr:
Chr applied to a character code returns the character.
Length:
Length applied to a string returns the no of characters in the string.

Subroutines and their uses

A subroutine is a named self-contained block of instructions, e.g. drawsquare. By encapsulating and naming a block of instructions in a program it becomes possible to call the block from other parts of the program.
This is very useful in situations where the same block of instructions or action or calculation needs to be repeated in multiple places in a program.
A program that references the subroutine `drawsquare` by name at a particular place in the program flow is said to **call the subroutine** `drawsquare`. It is sufficient to just use its name, `drawsquare`, to cause its block of instructions to execute.
Subroutines have been covered in depth in *Chapter 1.1.2*.
A subroutine may contain its own variable, type, label and const declarations.
A subroutine may also define subroutines which it may use and it may use subroutines defined elsewhere (usually library or language-defined subroutines).
Table 1.1.10.1 shows a pseudo-code procedure definition, `ConvertStringToUpper`, which itself contains a function definition, `Upper`, and a local variable declaration, `i : Integer`.
These procedure and function subroutines use library or language-defined subroutines, `Ord`, `Chr` and `Length`.

```
Procedure ConvertStringToUpper (INOUT Message : String)
  LocalVar
    i : Integer
  EndLocalVar
  Function Upper(Ch : Char) : Char
    If Ch In ['a'..'z']
      Then Ch ← Chr(Ord(Ch) - Ord('a') + Ord('A'))
    EndIf
    Return Ch
  EndFunction
  For i ← 1 To Length(Message)
    Message[i] ← Upper(Message[i])
  EndFor
EndProcedure
```

Table 1.1.10.1 Pseudo-code procedure definition, showing encapsulation of a block of instructions, a function and a local variable

A reference to a string is passed to procedure `ConvertStringToUpper` in the procedure parameter variable `Message`. The reference is to a string which exists in the calling program's variable memory space. Any changes made by `ConvertStringToUpper` are applied to this string. This is what is meant by `Message` being an INOUT parameter.

The function `Upper` is said to be nested inside procedure `ConvertStringToUpper`.

On encountering the name `ConvertStringToUpper`, the executing program transfers control to this subroutine. On finishing its execution `ConvertStringToUpper` transfers control back to the point in the program immediately following where it was called.

Single licence - Abingdon School

A subroutine is a named out of line block of code

Figure 1.1.10.1 shows the control structure of a program block consisting of program statements S1, S2, S3, a selection statement (shown as ?) controlling execution of two statements S4 and S5. This selection statement is followed by a loop which controls a block of statements S.
A procedure T consists of program statements S1, S2, S3 (different ones from the program block statements) and a loop controlling a block of statements S (also different from the program block S).

The flow of control in the program block is **in line**, forwards from the beginning to the end, except when statement S1 is encountered. This statement transfers control '**out of line'** to procedure T.
Flow of control in procedure T is from its beginning to its end.
On reaching the end of T, control is transferred back to the program block.
Execution is resumed in the program block at statement S2, the statement immediately following S1, where the call to T occurred. If T was a function then control would be returned to statement S1 along with the function's result.

Procedure T is a subroutine.
In Pascal, statement S1 which is a call to a procedure T which doesn't use parameters is simply `T;`
In VB.NET, the call to a procedure T which doesn't use parameters would be `T()`
This example, shows that a subroutine is:

a named 'out of line' block of code which may be executed (called) by simply writing its name in a program statement.

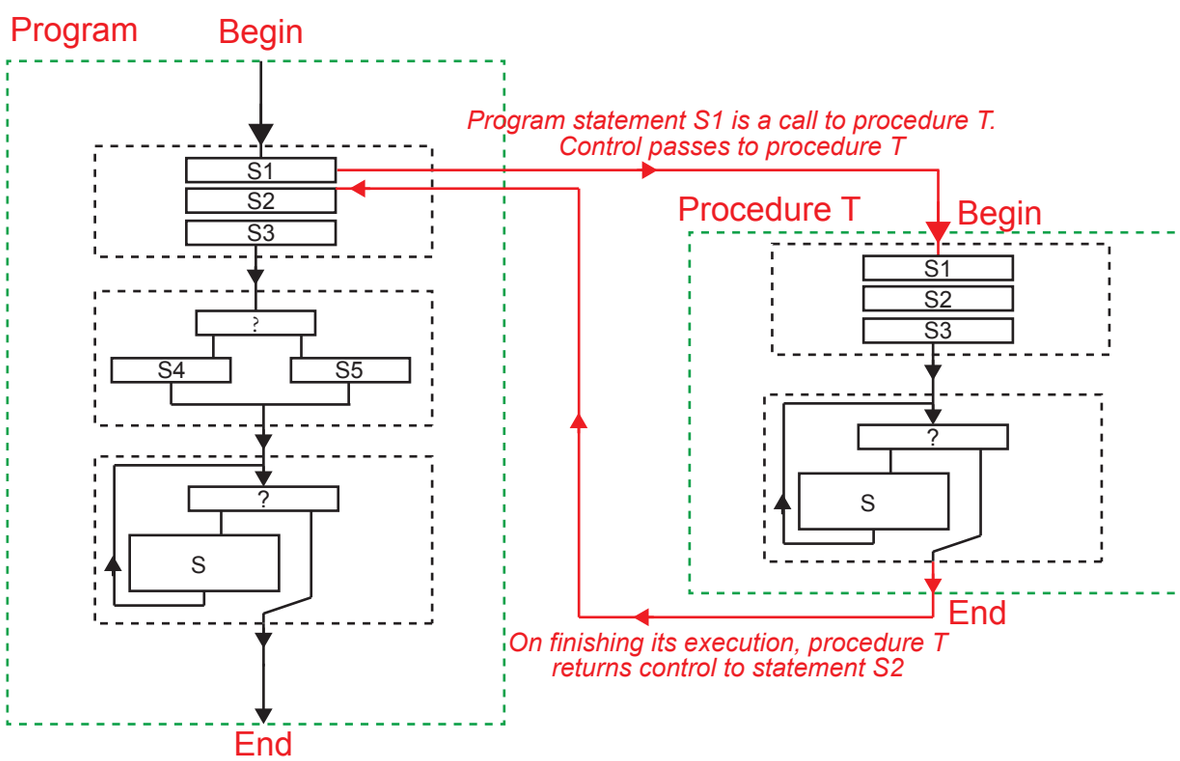


Figure 1.1.10.1 In line flow of control to out of line flow of control to a named out of line block of code

Questions

1. What is meant by "a subroutine is a named 'out of line' block of code that may be executed by simply writing its name in a program statement"?

Single licence - Abingdon School

Advantages of using subroutines in programs

Without programming language support for subroutines (procedures and functions), all programming would consist of inline blocks of program instructions.

This would make even programs of modest size

- difficult to understand
- difficult to debug.

Removing blocks of instructions from the program block and placing these in 'out of line', named subroutine blocks (*Figure 1.1.10.1*) separate from the control flow of the program block, reduces the intellectual demand needed to understand what the program does

The program block is reduced in length because where it relies on the instructions in subroutines these are referenced by a short and descriptive name (ideally).

If subroutines are self-contained they can be worked on separately.
This is useful when writing and debugging software.
In software projects involving a team of developers, different subroutines can be given to different members of the team to write and debug.
The more self-contained (independent) the subroutine the easier it is to write and debug without having to understand the program block in which it is called.

Subroutines written for one program may be reused in a different program. The more self-contained they are the easier it is to do this.

If a subroutine is particularly useful, it may be added to a library of subroutines which can be imported into any program which needs them.

Questions

2 State and explain **three** advantages of using subroutines in programs.

Did you know?

Inline code:
In defence applications such as missile homing heads, program code is often converted to inline code during the software construction stage.
Each subroutine call in the program block is replaced by the subroutine's block of instructions.
One reason this is done is to avoid a potential software crash caused by stack overflow.
A stack is used in subroutine calls.
A stack (or call stack) stores return addresses which are used by each executing subroutine to return to the place from where they were called. Specifically, to the statement following the call statement.
A stack is finite in size and is limited by the memory allocated to it.
A missile homing head is the part of a missile which guides the missile onto the target.
Missiles often cost upwards of £30000.
Once launched there is no opportunity to reboot if the computer crashes from stack overflow.

In this chapter you have covered:

- Subroutines and their uses
- A subroutine as a named 'out of line' block of code that may be executed (called) by simply writing its name in a program statement
- The advantages of using subroutines in programs

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be able to describe the use of parameters to pass data within programs*
- *Be able to use subroutines with interfaces*

1.1.11 Parameters of subroutines

Using parameters to pass data within programs

A subroutine parameter is one way of passing data into and out of a subroutine. When a subroutine is called, any data passed to it via the subroutine parameter mechanism is copied into the memory area reserved for subroutine formal parameters as shown in the memory schematic in *Figure 1.1.11.1*.

```
Program Demo
Var x, y :Integer
Subroutine T(IN r : Integer; INOUT s : Integer)
  s ← s*r
EndSubroutine
BeginProgramBlock
  x ← 25
  y ← 17
  Call T(x,y)
  Output y
EndProgramBlock
```

Table 1.1.11.1 An example of a pseudocode program with a subroutine call involving data associated with two program variables being passed to the subroutine

There are two ways that data may be passed via the subroutine parameter mechanism into a subroutine:

- Call by value
- Call by reference/Call by address

We illustrate both with data stored in two program variables, x and y.

The variable name x maps to value 25 and the variable name y maps to 17.

We can express this as follows where the symbol ↦ means 'maps to'

x ↦ 25 and y ↦ 17

This mapping is set up at the point in time when statements

```
x ← 25
y ← 17
```

are executed in the program shown in *Table 1.1.11.1*.

The subroutine memory area shows two variables, r and s, called **formal parameters** of the subroutine.

Figure 1.1.11.1 shows these mapping to 5674 and 25.

r ↦ 25 and y ↦ 5674

This mapping is set up at the point in time when

```
Call T(x, y)
```

is executed in the program shown in *Table 1.1.11.1*.

Close inspection of the memory map in *Figure 1.1.11.1* indicates that the datum associated with x has been copied into the location in the subroutine memory area associated with r.

Similarly, close inspection shows that the program memory address 5674 of the location in program memory associated with y has been copied into the location in the subroutine memory area associated with s.

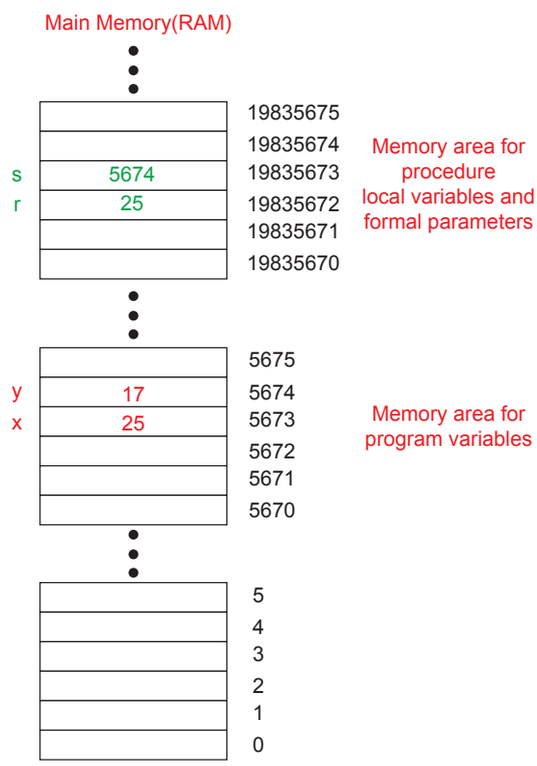


Figure 1.1.11.1 Memory map showing an area reserved for program variables and an area reserved for subroutine variables

Single licence - Abingdon School

Now when we look at the subroutine header we see that the r parameter is labelled an IN parameter and the s parameter is labelled an INOUT parameter:

```
Subroutine T(IN r : Integer; INOUT s : Integer)
```

The interpretation of IN r : Integer is as follows: a value is to be copied into the formal parameter r when the subroutine is called.
This is what is meant by **Call by Value**.
The interpretation of INOUT s : Integer is as follows: an address of where a value can be found in the program memory area is to be copied into the formal parameter s when the subroutine is called.
This is what is meant by **Call by Reference/Call by Address**.

This copying takes place when the subroutine is called with **actual parameters** x and y by the program statement

```
Call(x, y)
```

If a subroutine has the address of a datum in the program memory area then the subroutine may change the value in this area. The subroutine in *Table 1.1.11.1* does this with an assignment statement

```
s ← s*r
```

Call by reference/address thus can have the **side-effect** of changing the value of a variable in another area of memory. This side-effect can be desirable and intended or undesirable.
Call by value cannot change the original value which has been copied.
Call by reference/address should be used as follows

- when a datum is too big to pass by value, i.e. it would take up a lot of space in subroutine memory or it would take too long (relatively speaking) to copy into subroutine memory
- when more than one result of executing the subroutine needs to be returned from the subroutine call
- when the data type of the result to be returned is not supported by the mechanism used by a function to return a result. Function return uses a different mechanism from subroutine parameters to return a result.

If only a single result needs to be returned and the data type of the result is supported by the function mechanism for returning results then a function should normally be used.

When a subroutine calls a subroutine, the subroutine memory area is used for both the calling subroutine and the called subroutine parameters/variables.

Table 1.1.11.2 shows Call by Value and Call by Reference/Address support in the programming languages Python, Java, Pascal/Delphi, C# and VB.NET.

Table 1.1.11.3 shows an example in Java of Call by Value.

Table 1.1.11.4 shows an example in Java of Call by Reference/Address.

Table 1.1.11.5 shows an example in Pascal of Call by Value and Call by Address.

Table 1.1.11.6 shows an example in C# of Call by Value and Call by Reference/Address.

Table 1.1.11.7 shows an example in VB.NET of Call by Value and Call by Reference/Address.

Key term

Subroutine parameter:
A subroutine parameter is a mechanism for passing data into and out of a subroutine.

Key term

Call by value:
Formal parameter of subroutine gets a copy of the datum associated with the actual parameter used in call to subroutine.

Call by address:
Formal parameter of subroutine is assigned the address in memory of the datum associated with the actual parameter used in call to subroutine.

Single licence - Abingdon School

Language	Call by Value	Call by Reference/Address
Python	Python is neither "call by value" nor "call by reference". In Python a variable is not an alias for a location in memory. It is a binding to a Python object.	

Java	`void a(int r){`
	`  r = r +10;`
	`}`
		There is no call by reference/address only call by value but a reference may be passed by value:
		`class Test{`
		`  int x =5;`
		`  void c(Test r){`
		`  r.x = r.x+10;//object variable x is changed`
		`  }`
		`}`

Pascal/ Delphi	`Procedure A(r : Integer);`
	`  Begin`
	`  r := r + 10;`
	`  End;`
	`Function B(s : Integer) : Integer;`
	`  Begin`
	`  B := s + 10;`
	`  End;`
		`Procedure C(Var r : Integer);`
		`  Begin`
		`  r := r + 10;`
		`  End;`

C#	`static void a(int r)`
	`  {`
	`  r = r + 10;`
	`  }`
		`static void b(ref int s)`
		`  {`
		`  s = s + 10;`
		`  }`

VB.NET	`Sub A(ByVal r As Int)`
	`  r = r + 10`
	`End Sub`
		`Sub B(ByRef s As Int)`
		`  s = s + 10`
		`End Sub`

Table 1.1.11.2 Call by Value and Call by Reference/Address support in the programming languages Python, Java, Pascal/Delphi, C# and VB.NET

```
public class CallByValue {
  public static void main(String[] args) {
    int x = 5;
     System.out.println("Before call to c " + x);
     c(x);
     System.out.println("After call to c " + x);
  }
  static void c(int r){
    r = r + 10;
  }
}
```

Table 1.1.11.3 Java example of Call by Value

```
public class CallByReference {
    int x = 5;
    void c(CallByReference s){
      s.x = s.x+10;//object variable x is changed
    }
   public static void main(String[] args) {
     CallByReference exampleCByR = new CallByReference();
     System.out.println("Before Before call to c " + exampleCByR.x);
     exampleCByR.c(exampleCByR);
     System.out.println("After call to c " + exampleCByR.x);
    }
}
```

Table 1.1.11.4 Java example of Call by Reference

Single licence - Abingdon School

```
Program CallByValueAndCallByAddress;
Procedure A(r : Integer);
  Begin
    r := r + 10;
  End;
Function B(s : Integer) : Integer;
  Begin
    B := s + 10;
  End;
Procedure C(Var s : Integer);
  Begin
    s := s + 10;
  End;
Var
  x, y : Integer;
Begin
  x := 5;
  y := 5;
  A(x);
  Writeln(x);
  Writeln(B(y));
  C(y);
  Writeln(y);
  Readln;
End.
```

Table 1.1.11.5 Pascal example of Call by Value and Call by Reference

```
using System;
namespace CallByValueAndCallByReference
{
    class Program
    {
        static void Main(string[] args)
        {
            int x;
            x = 10;
            a(x);
            Console.WriteLine(x);
            b(ref x);
            Console.WriteLine(x);
            Console.ReadLine();
        }

        static void a(int r)
        {
            r = r + 10;
        }

        static void b(ref int s)
        {
            s = s + 10;
        }
    }
}
```

Table 1.1.11.6 C# example of Call by Value and Call by Reference

```
Module Module1
    Sub Main()
        Dim x As Integer = 5
        A(x)
        Console.WriteLine(x)
        B(x)
        Console.WriteLine(x)
        Console.ReadLine()
    End Sub
    Sub A(ByVal r As Integer)
        r = r + 10
    End Sub
    Sub B(ByRef s As Integer)
        s = s + 10
    End Sub
End Module
```

Table 1.1.11.7 VB.NET example of Call by Value and Call by Reference

Questions

1. Explain the role of a subroutine parameter.
2. Explain
   (a) Call by value
   (b) Call by address

Using subroutines with interfaces

```
Subroutine T(IN r : Integer; INOUT s : Integer)
  s ← s*r
EndSubroutine
```

In the subroutine T, the part

```
IN r : Integer; INOUT s : Integer
```

is called the subroutine's **interface**. A subroutine interface is a mechanism by which data may be passed in and out of a subroutine via subroutine parameters. The use of subroutine interfaces is covered in *Chapter 1.1.2* and *Chapter 1.2.1*.

In this chapter you have covered:

- The use of parameters to pass data within programs
- Using subroutines with interfaces

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be able to use subroutines that return values to the calling routine*

1.1.12 Returning a value(s) from a subroutine

Using subroutines that return values to the calling routine

Subroutines

A subroutine allows a block of instructions to be encapsulated, named and called by this name from other parts of a program.

This is particularly useful if the same block of instructions needs to be used in multiple places in a program or in several different programs.

After a subroutine is executed, control returns to the statement calling the subroutine. If the subroutine is a procedure then the calling statement's execution is complete. If the subroutine is a function, then the function returns a result to the calling statement which this statement may or may not deal with before it completes its execution.

For example, if the calling statement is as follows

```
x ← SquareOf(2)
```

the result **4** is returned by the call to function `SquareOf` and this result is then assigned to variable `x`.

Returning a result

If a result needs to be returned from an executing subroutine then a programmer may choose from two subroutine options:

- Procedure with the result returned via an INOUT or OUT parameter (call by address/reference)
- Function with the result returned via the function return mechanism

Procedure option

The procedure option has been covered in *Chapter 1.1.11*. An OUT parameter is similar to an INOUT parameter but some programming languages differentiate between the two by requiring an INOUT parameter to be initialised with a value whereas an OUT parameter can be passed uninitialized or undefined (see C#).

Information

In C# a value must be assigned to an OUT parameter within the subroutine, whereas an INOUT parameter can be left unchanged.

Function option

In some programming languages, e.g. Pascal, a function specifies a return value within the body of the function by executing an assignment statement whose left-hand side is the name of the function - *Table 1.1.12.1*.

```
Function SquareOf (Number : Integer) : Integer;
  Begin
    SquareOf := Number * Number;
  End;
```

Table 1.1.12.1 Pascal function SquareOf which returns an integer result to the calling statement

Single licence - Abingdon School

In Delphi, when a function is called, a variable is automatically created with the name Result of the same type as the return type of the function. It is available for the programmer to use to hold the result value to be returned by the function. Its value is passed back to the calling statement when the function returns.

The implicitly declared variable Result can be seen as equivalent to an OUT type parameter - where the value upon entry to the function is undefined. It is still possible in Delphi to use the Pascal convention of using the name of the function to assign the result. In Pascal, assigning a value to be returned to the name of the function does not automatically cause the function to return. Similarly, in Delphi assigning a value to be returned to the name of the function or to the variable Result does not automatically cause the function to return. In both Pascal and Delphi, usual practice is to use a temporary local variable to hold the result to be returned and assign this to Result or the function name at the end of the function, which is the place where control passes back to the calling statement.

In other programming languages, e.g. C#, functions use an explicit return statement

```
return expression
```

In addition to specifying a value, return causes the immediate termination of the function.

Table 1.1.12.2 shows a rather contrived C# function `Calculate` with three return expression statements.
The program executes when Main is called.
The logic of the program selects

```
return number * number
```

because parameter number is 2.
At this point control passes back to the calling statement

```
Console.WriteLine(Calculate(2));
```

This statement then outputs the value 4.
If control did not pass back at this point then `return 6`, the last statement in Calculate, would also be executed with the outcome that the return value would change to 6.
The fact that the output is 4 and not 6 confirms that return causes the immediate termination of the function.

```
using System;
namespace FunctionReturn
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Calculate(2));
            Console.ReadLine();
        }
        static int Calculate(int number)
        {
            if (number == 2)
            {
                return number * number;
            }
            if (number == 4)
            {
                return number * number * number;
            }
            return 6;
        }
    }
}
```

Table 1.1.12.2 C# function Calculate which returns an integer result to the calling statement

Type of result returned

Many programming languages place restrictions on the type of the result returned by a function.

Table 1.1.12.3 shows a Pascal program which defines and uses a function whose result type is a **composite data type**.
A composite data type or compound data type is any data type which can be constructed in a program using the programming language's primitive data types and other composite types.
It is sometimes called a **structured data type**, but this term is more commonly reserved for arrays and lists.
The act of constructing a composite type is known as composition.

Single licence - Abingdon School

When not to use a function to return a result

Although it is possible to return multiple separate results from a function, it is not normally considered good practice. For example, the following Pascal function returns one result in INOUT parameter s and another result via the function return mechanism:

```
Function D(Var s:Integer) : Integer;
  Begin
    ...
  End;
```

Both results in this example return a value which is a scalar data type, in this case, an integer. A scalar data type is a single value data type. It would be better to use a procedure with two INOUT parameters instead of a function.

```
Program FunctionReturnType;
Type
  TArrayType = Array[1..10] Of Integer;
Function Test (x : TArrayType) : TArrayType;
  Var
    i : Integer;
  Begin
    For i := 1 To 10
      Do x[i] := 2 * x[i];
    Test := x;
  End;
Var
  y : TArrayType = (1,2,3,4,5,6,7,8,9,10);
  z : TArrayType;
Begin
  z := Test(y);
  Writeln(z[2]);
  Readln;
End.
```

Table 1.1.12.3 Pascal function test which returns a result of composite type to the calling statement

Mathematical functions vs functions in a programming language

Loosely speaking, a function in mathematics is a rule that, for each element in some set A of inputs, assigns an output chosen from set B. For example, a function SquareOf takes input integer 2 and outputs its square 4 another integer.

Mathematics is very strict in its definition of a function:

- A function must return a single result
- A function must always return the same value for a given input, e.g. the square of 2 is always 4.
- A function must return a value for every value of input.

In programming languages, functions can break these rules:

- The Random (N) function covered in *Chapter 1.1.8* returns a different value in the interval [0, N) each time it is called. It would be useless as a random number generator if it returned the same value every time
- Multiple separate results may be returned from a function by using INOUT parameters as well as the function return mechanism
- The integer data type, e.g. int in C#, is a subset of the set of integers because computer memory is finite. If we attempt to square an integer which is larger than the square root of the maximum integer for the integer data type then overflow will result.

Programming tasks

1. Write and test a function which accepts a string as input and returns the string reversed.
2. Write and test a function which accepts a string as input and returns the number of words it contains, e.g. "Hello World" contains 2 words.

Single licence - Abingdon School

Programming tasks

3 Write and test a function which on each call returns a random number of data type integer in the interval (0, 12], i.e. 0 < number <= 12.

Use the generating relation and an appropriate value for k

NextRandomNo = Multiplier * PreviousRandomNo MOD k

and multiplier = 21. Seed the random number generator with the value 7, i.e. the first value of PreviousRandomNumber = 7.

4 Write and test a function which on each call returns a random number of data type real/float in the interval (0.0, 1.0].

In this chapter you have covered:

- Using subroutines that return values to the calling routine

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Know that subroutines may declare their own variables called local variables, and that local variables:*
  - *exist only while the subroutine is executing*
  - *are accessible only within the subroutine*
- *Be able to use local variables and explain why it is good practice to do so*

1.1.13 Local variables in subroutines

Declaring local variables

A subroutine may declare its own variables and use these within the body of the subroutine as shown in the subroutine *DoExample* in *Table 1.1.13.1*.

```
Subroutine DoExample(IN x : Integer)
  Var
    s, t : Integer
  Body
    Output x
    s ← 6
    t ← 7
    Output s + t
  EndBody
EndSubroutine
```

Declaration of two local variables, *s* and *t*

Using local variables, *s* and *t* in body of subroutine

Table 1.1.13.1 Subroutine with local variables

The variables *s* and *t* are known as **local variables**. They are only visible inside subroutine *DoExample*, i.e. they cannot be accessed from outside the subroutine. In fact, they do not exist until the subroutine starts executing. They disappear when the subroutine stops executing. Thus any values that they hold are stored temporarily.

Lifetime

The lifetime of a local variable is the lifetime of an execution of the subroutine in which the local variable is declared.

Visibility

The scope of a local variable is the scope of the subroutine in which it is declared – scope means where the local variable is visible and can be used.

Using local variables

Table 1.1.13.2 shows the declaration and use of local variables (called `i` and `j`) in the programming languages Pascal/Delphi, C#, Python, Java and VB.NET.

Key term

Local variable:
These are variables declared inside a subroutine and used within the body of the subroutine. The lifetime of a local variable is the lifetime of an execution of the subroutine in which the local variable is declared.
The scope of a local variable is the scope of the subroutine in which it is declared – scope means where the local variable is visible and can be accessed.

Programming tasks

1. Write and test a function which calculates x^n where $n \geq 1$.
   Use the following algorithm to calculate x^n

```
Power ← x
Count ← 1
While Count < n Do
  Power ← Power * x
  Count ← Count + 1
```

   Your function should make use of local variables.

Single licence - Abingdon School

Pascal/Delphi

```
Procedure DoExample(n : Integer);
  Var
    i, j : Integer;
  Begin
    j := 6;
    For i : = 1 To n
      Do Writeln('Hello World!', i*j)
  End;
```

C#

```
static void DoExample(int n) {
  int j = 6;
  for (int i = 1; i <= n; i++)
    {
      Console.WriteLine("Hello World! {0}", i*j);
    }
}
```

Python

```
def doExample(n):
    j = 6
    for i in range(n):
        print('Hello World!', j*(i + 1))
```

Java

```
public static void doExample(int n) {
  int j = 6;
  for (int i = 1; i <= n; i++) {
    System.out.println("Hello World! " + j*i);
  };
}
```

VB.Net

```
Sub doExample(ByVal n As Integer)
   Dim j As Integer = 6
   For i As Integer = 1 To n
       Console.WriteLine("Hello World! {0}", i*j)
   Next
End Sub
```

Table 1.1.13.2 Declaring and using local variables

Why use local variables?

Support for modularisation

Subroutines enable modularisation of a program.

A solution to a problem can be divided into separate and independent modules. These modules can be implemented as subroutines.

The aim of modularisation is to have subroutines which can be worked on independently of the rest of the program. This requires that each subroutine is **self-contained** and that its interaction with the rest of the program takes place only through the subroutine's interface, i.e. through its formal parameter(s).

Using local variables aids modularisation.

It also enables a subroutine to be reused because if a subroutine is self-contained and independent it may be lifted and used in another program.

Undesirable side-effects

In *Table 1.1.13.3*, the formal parameter of subroutine *DoExample* is *x*. However, instead of declaring local variables, the subroutine manipulates two global variables *s* and *t*.

These global variables come into existence when the program starts executing and only disappear when it finishes executing.

They are visible and therefore accessible inside procedure *DoExample*.

Unfortunately, this means that when *DoExample* is called and it executes the values of *s* and *t* are altered as shown.

This is called a **side-effect** of executing subroutine *DoExample*.

Single licence - Abingdon School

```
Program
  Var
    s, t, y : Integer
  Subroutine DoExample(INOUT x : Integer)
    Body
      x ← 2 * x
      s ← 6
      t ← 7
      Output s + t
    EndBody
  EndSubroutine
Begin
  s ← 3
  t ← 2
  y ← 9
  DoExample(y)
  Output s + t
  Output y
End
```

Declaration of three global variables, *s*, *t* and *y*

Using global variables, *s* and *t* in body of subroutine

s + t = 13

Expect output to be *3 + 2 = 5* but it is *13*. This is a side-effect of calling subroutine *DoExample*

Table 1.1.13.3 Demonstrating the side-effect of using global variables inside a subroutine

Side-effects are undesirable because

- debugging a program is made harder - isolating the source of an error is made more difficult, the whole program may need to be examined
- the independence of the subroutine is reduced
  - it cannot be designed without considering the context in which it will be used
  - it cannot be debugged without considering the context in which it will be used
  - its reusability is reduced because it will be more difficult to use it in another program.

The only variable outside of *DoExample* which should be manipulated by *DoExample* is *y* because this variable is passed into *DoExample* through its INOUT interface, the formal parameter *x*.

Questions

1. What is a local variable?
2. What is the lifetime and scope of a local variable?
3. Explain why the use of local variables is considered good practice.

In this chapter you have covered:

- That subroutines may declare their own variables called local variables, and that local variables:
  - exist only while the subroutine is executing
  - are accessible only within the subroutine
- Using local variables and why it is good practice to do so

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be able to contrast local variables with global variables*

1.1.14 Global variables in a programming language

Key term

Static scoping:
In a programming language with static (lexical) scoping, the scope of a binding is determined at the compile time.

Key term

Binding:
A binding is an association between two things, such as a name and the thing its names.

Key term

Scope:
Scope is the the textual region in which a binding is active.

Key term

Binding time:
Binding time is the time at which a binding is created.

Key term

Compiler:
A compiler translates the source code form of a program into an independently executable object code form.

Contrasting local variables with global variables

Contrasting the scope of global and local variables

In *Chapter 1.1.13* we learned that the scope of a local variable is limited to the subroutine in which it appears; it is not visible elsewhere.

Local variables are created when their subroutine is called and destroyed when it returns.

In a programming language with static scoping, the bindings between names and variables, etc, can be determined at compile time by examining the text of the program without consideration of the flow of control at run time.

Table 1.1.14.1 shows a program which declares two global variables *s* and *t*.

The compiler is able by inspection of the program text to determine at compile time that memory needs to be reserved for two integer variables, *s* and *t*.

The compiler builds this requirement into the executable form of the program which it produces. When the executable form of the program is loaded into memory, space is allocated to the two global variables, *s* and *t*.

A binding exists between global variables, *s* and *t* and the space allocated in memory to each.

The space allocated to *s* and *t* is retained throughout the program's execution.

We call this kind of allocation of memory **static memory allocation** and as a consequence the global variables are called **static variables**.

The space allocated to *s* and *t* is de-allocated on exit from the program.

```
Program GlobalVariables
  Var
    s, t : Integer          ← Declaration of two global variables, s and t
  Subroutine DoExample()
    Var
      s, t : Integer        ← Declaration of two local variables s and t
    Body
      s ← 6
      t ← 7
      Output s + t          ← Output 6 + 7 = 13
    EndBody
  EndSubroutine
Begin
  s ← 3
  t ← 2
  DoExample()
  Output s + t              ← Output 3 + 2 = 5
End
```

Table 1.1.14.1 Contrasting the scope of global variables and local variables

Single licence - Abingdon School

In contrast, the compiler will note that the variables *s* and *t* declared inside subroutine *DoExample* are local to this subroutine.

Memory is allocated for these local variables only when *DoExample* is called and this memory is de-allocated when the subroutine returns (control is passed back to the calling statement).
The compiler will set up the executable form of the program to allocate memory space for local variables *s* and *t* when *DoExample* is called.
Memory is allocated to local variables from an area of memory called stack memory. This is a different area of memory from the area used for global variables.

If the program in *Table 1.1.14.1* is executing and control is currently in the block of the program between *Begin* and *End*, i.e.

```
s ← 3
t ← 2
DoExample()
Output s + t
```

Variables *s* and *t* in this block refer to the global variables *s* and *t*.
We say that the binding between *s* and *t* and the statically allocated memory is active.

If control passes to subroutine *DoExample* i.e. the block of program between *Body* and *EndBody*, i.e.

```
s ← 6
t ← 7
Output s + t
```

Variables *s* and *t* in this block refer to local variables *s* and *t*.

We say that the binding between *s*, and *t* and the allocated stack memory is active.

To avoid two variables *s* and two variables *t* being active at the same time, global *s* and global *t* go out of scope, and local variables *s* and *t* come into scope. We say their bindings are active and the binding of the global variables, temporarily inactive.

On exit from the subroutine, control returns to the *Begin End* block and the global variable bindings become active again.

Global scope

Table 1.1.14.2 shows the creation of two global variables *s* and *t* in various programming languages.

In Pascal/Delphi, variables declared at the outermost level with the language keyword *Var* are both global and static.
In C#, the keyword *static* creates a variable and statically binds it; the keyword *public* turns this variable into a global variable.
In Java, the keyword *static* creates a variable and statically binds it; the keyword *public* turns this variable into a global variable.
In VB.NET, the keyword *Public* in front of *variable name As datatype* creates a global variable.
In Python, two global variables *s* and *t* are created by the statements

```
s = 2
t = 3
```

To access these global variables inside subroutine *doExample*, the modifier *global* must be placed in front of each to indicate that it is global variables *s* and *t* which are to be used. Otherwise, Python will create two local variables of the same names.

Single licence - Abingdon School

Pascal/Delphi

```
Program GlobalVariables;
  Var
    s, t : Integer;
  Procedure DoExample();
    Begin
      s := 6;
      t := 7;
    End;
Begin
  s = 3;
  t = 2;
  Writeln("s = ", s, " t = ", t);
  DoExample();
  Writeln("s = ", s, " t = ", t);
End.
```

Accessing global variables *s* and *t*

C#

```
using System;
namespace GlobalVariables
{
  class Program
  {
    public static int s, t;
    static void DoExample()
    {
      s = 6;
      t = 7;
    }
    static void Main(string[] args)
    {
      s = 3;
      t = 2;
      Console.WriteLine("s = {0} t = {1}", s, t);
      DoExample();
      Console.WriteLine("s = {0} t = {1}", s, t);
      Console.ReadLine();
    }
  }
}
```

Python

```
def doExample():
    global s
    global t
    s = 6
    t = 7
s = 2
t = 3
print("s = ", s, " t = ",t)
doExample()
print("s = ", s, " t = ",t)
```

Java

```
public class GlobalVariables {
  public static int s, t;
  static void doExample() {
    s = 6;
    t = 7;
  }
  public static void main(String[] args) {
    s = 3;
    t = 2;
    System.out.println("s = " + s + " t = " + t);
    doExample();
    System.out.println("s = " + s + " t = " + t);
  }
}
```

VB.Net

```
Module Module1
    Public s As Integer
    Public t As Integer
    Sub DoExample()
        s = 6
        t = 7
    End Sub
    Sub Main()
        s = 3
        t = 2
        Console.WriteLine("s = {0} t = {1}", s, t)
        DoExample()
        Console.WriteLine("s = {0} t = {1}", s, t)
        Console.ReadLine()
    End Sub
End Module
```

Table 1.1.14.2 Demonstration of the use of global variables in various languages

Single licence - Abingdon School

Figure 1.1.14.1 shows the output when the C# program in *Table 1.1.14.2* is executed. Variables ***s*** and ***t*** are global and so are visible within the subroutine *DoExample*, i.e. they have *global scope*.

Variables with global scope are visible throughout the entire program and therefore can be accessed from anywhere in the program unless their binding is made inactive temporarily for the reason explained in the first section in this chapter. When variables binding is temporarily inactive the variable is said to be out of scope.

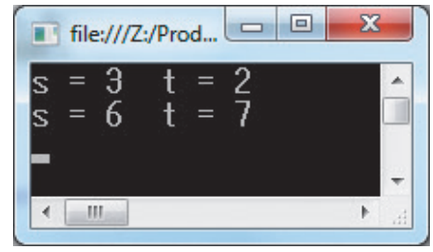


Figure 1.1.14.1 Screenshot of output from the C# program in Table 1.1.14.2 when executed

Questions

1. State **two** differences between global and local variables.
2. What does it mean to say that a global variable is out of scope?
3. What is the output when the program shown in *Table 1.1.14.3* is executed?
   Describe the action of this program in producing this output.

```
Program GlobalVariables
  Var
    s : Integer
  Subroutine DoExample()
    Var
      s: Integer
    Body
      s ← 6
    EndBody
  EndSubroutine
Begin
  s ← 3
  Output s
  DoExample()
  Output s
End
```

Table 1.1.14.3

In this chapter you have covered:

- Contrasting local variables with global variables

Single licence - Abingdon School

1 Fundamentals of programming

1.1 Programming

Learning objectives:

- *Be familiar with the use of recursive techniques in programming languages (general and base cases and the mechanism for implementation)*
- *Be able to solve simple problems using recursion*

1.1.16 Recursive techniques

What is Recursion?

Recursion occurs in situations where an object partially consists of or is defined in terms of itself. *Figure 1.1.16.1* shows a recursive picture. The picture consists of increasingly smaller versions of itself.

The power of recursion lies in its possibility of defining an infinite set of objects by a finite statement.

Information

To understand recursion, you must understand recursion.

For example,

Recursive picture = Picture

Or

Recursive Picture = Recursive Picture + Recursive Picture x Magnification factor of 0.9

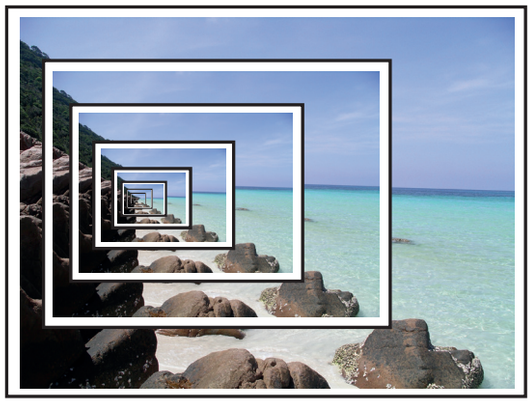


Figure 1.1.16.1 Recursive picture

The original picture, Picture, is shown in *Figure 1.1.16.2*.

Figure 1.1.16.2 Original picture

In computer science, we may define data structures in this way, i.e. in terms of themselves - see *Chapter 2.5.1* on trees for an example of this.

To see how this is possible, consider a simple shopping list which starts out empty. We will call this an empty shopping list or a list with no elements or items.

Now we use a rule to turn this simple case into something slightly less simple:

> *We take the (empty) list and construct a list one bigger by attaching an item to the beginning.*

This seems at first to be a circular definition, i.e. a definition of something in terms of itself. But it isn't because we start somewhere, the **base case** which is the empty list.

In *Figure 1.1.16.1* the base case is the original picture, Picture. The next largest (0.9 x base case) is then painted/stuck on top and the next largest on top of this and so on using the recursive definition shown above.

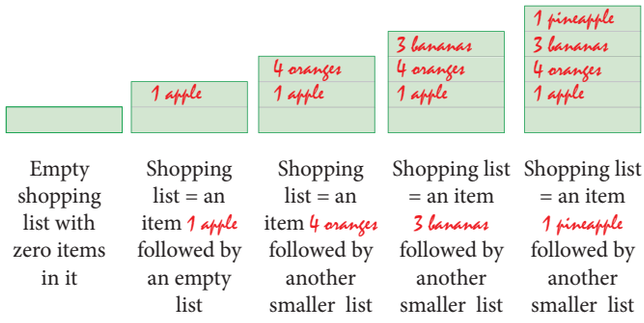


Figure 1.1.16.3 Shopping list consisting of an item and another list

Figure 1.1.16.3 shows a list example which illustrates how it is possible to view a structure as being recursively defined.

The recursive definition of a list is

- the empty list with zero items in it, or
- an item followed by a list

Single licence - Abingdon School

The true usefulness of a recursive definition comes from going the other way:

> If what you are dealing with meets the recursive definition, then there is usually a way to move back towards the base case, getting simpler and simpler along the way.

This is what recursive programming does.

All recursive functions/proceduress follow a similar pattern:

- Am I at the base case? If so, return the easy solution if a function, and stop recursing; if a procedure stop recursing.
- Otherwise, think in terms of solving the current problem by moving closer to the base case with a slightly simpler problem and solving this simpler problem.

Key concept

Recursive definition:
A recursive definition is one that is defined in terms of itself.

Recursively-defined function/ procedure:
A recursively-defined function/ procedure is a function/procedure which calls itself or which is defined in terms of itself.

Recursive object:
An object is said to be recursive if it partially consists of or is defined in terms of itself.

For example, suppose we wanted to calculate how many billiard balls fit an equilateral triangle frame where the length of side is measured as the number of billiard balls that fit along the side, e.g. see *Figure 1.1.16.4*.

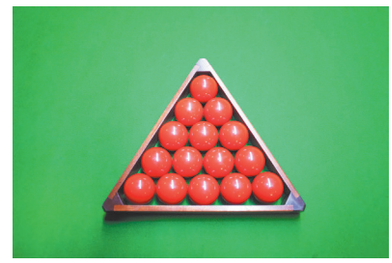


Figure 1.1.16.4 Billiard balls in frame

We can use the recursive definition

```
If NoAlongSide = 1
    NoOfBilliardBallsInTriangle(1) = 1    Base case
If NoAlongSide > 1    General case
    NoOfBilliardBallsInTriangle(NoAlongSide) = NoAlongSide + NoOfBilliardBallsInTriangle(NoAlongSide – 1)
```

So, for example, if the calculation is for a triangle with number of balls along the side 4 then we find a simpler case to solve first, the number of balls in a triangle with number of balls along the side 3 - *see Figure 1.1.16.5*.

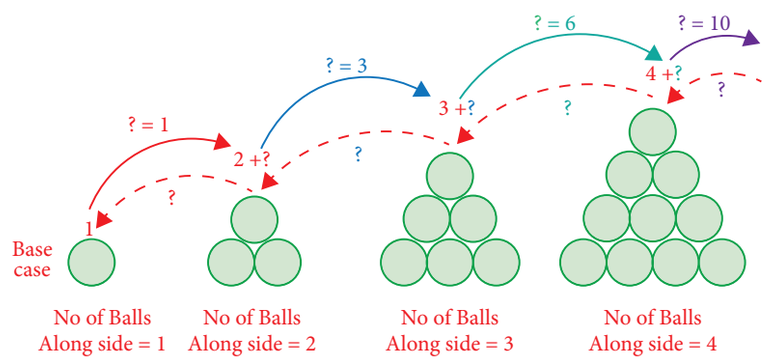


Figure 1.1.16.5 Billiard ball in equilateral triangles

This simpler problem can be solved by first solving a an even simpler problem, the number of balls in a triangle with number of balls along the side 2 which in turn can be solved by first solving for one ball, the **base case**.
This has the solution 1.
We can now solve for side 2, it is 2 + 1 = 3.
Next we solve for side 3, it is 3 + 3 = 6 and finally we solve for side 4, it is 4 + 6 = 10.

We could have drawn a triangle of side 4 and simple counted the total number of billiard balls.
So why express the solution to this problem recursively?

Single licence - Abingdon School

Well, the side 4 example is trivial but suppose we want to calculate the number of balls in a triangle of side 1000. This is not a trivial problem to solve. However, it can be done by recursion easily.

Figure 1.1.16.6 shows a pseudo-code function which returns 1 if the side is 1 else it returns the result of the calculation

NoAlongSide + NoOfBilliardBallsInTriangle(NoAlongSide - 1)

```
Function NoOfBilliardBallsInTriangle(NoAlongSide : Integer) : Integer
  If NoAlongSide = 1
    Then Return 1
    Else Return NoAlongSide + NoOfBilliardBallsInTriangle(NoAlongSide - 1)
  EndIf
End Function
```

Figure 1.1.16.6 Pseudo-code function which calculates no of billiard balls in a triangle of given size

Figure 1.1.16.7 shows this pseudo-code implemented in the functional programming language F# as a recursive function `NoOfBilliardBallsInTriangle` with parameter `NoAlongSide`.

Functional programming is covered in *Unit 2 section 12*.

When this function is called with 1000 substituted for `NoAlongSide`, i.e `NoOfBilliardBallsInTriangle 1000` the result returned is 500500.

Figure 1.1.16.7 shows this program running in an F# interactive console window.

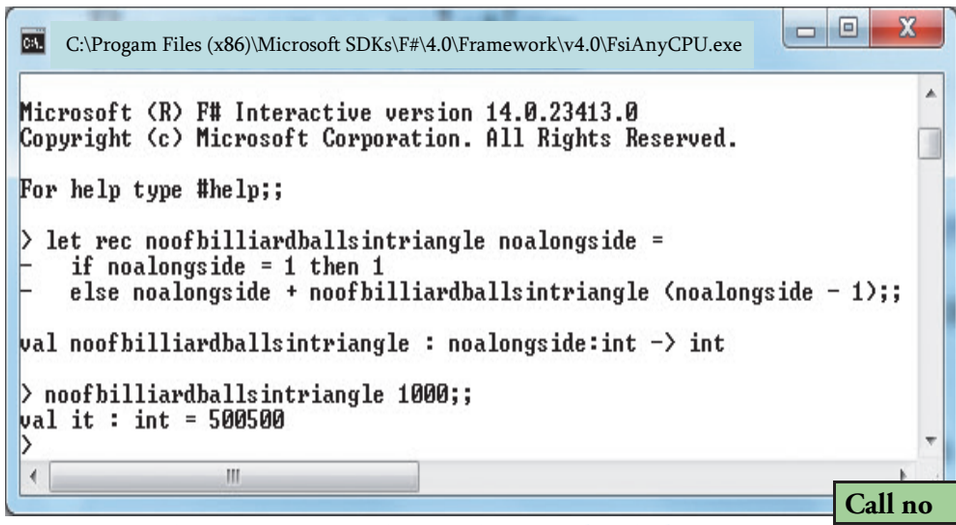


Figure 1.1.16.7 F# interactive console window

Information

An **FSharp Interactive (FSI)** window can be opened from Visual Studio by selecting **View | Other Windows | F# Interactive**.

Alternatively, launch C:\Progam Files (x86)\Microsoft SDKs\F#\<version>\Framework\<version>\FsiAnyCPU.exe - see Figure 1.1.16.7.

Key principle

The power of recursion lies in the fact that an infinite number of computations can be described by a finite recursive program which may contain no explicit repetitions.

Worked example

The task is to calculate the sum of the first *n* natural numbers recursively,

e.g. if n = 5 then the Sum = 1 + 2 + 3 + 4 + 5.

First find the **base case**:

If n = 1 then Sum = 1.

Next find the **general case**:

If n > 1 Then Sum = n + Sum(n -1).

Table 1.1.16.1 shows a trace table of recursive calls. The last call is to the base case.

Figure 1.1.16.8 shows a pseudo-code function which calculates the sum of the first *n* natural numbers recursively.

Call no	Sum(n)	Return
1	Sum(5)	5 + Sum(4)
2	Sum(4)	4 + Sum(3)
3	Sum(3)	3 + Sum(2)
4	Sum(2)	2 + Sum(1)
5	Sum(1)	1

Table 1.1.16.1 Calculating Sum(5) recursively

```
Function Sum(n : Integer) : Integer
  If n = 1
    Then Return 1
    Else Return n + Sum(n-1)
  EndIf
End Function
```

Figure 1.1.16.8 Pseudo-code function which calculates sum of first n natural numbers

Single licence - Abingdon School

Questions

1 Write in pseudo-code a recursively-defined function Product which calculates the product of the first *n* natural numbers, e.g. n = 5, product = 1 x 2 x 3 x 4 x 5.

Complete the trace table shown in *Table 1.1.16.2* for this function.

Call no	Product(n)	Return
1	Product(5)	
2		
3		
4		
5		

Table 1.1.16.2 Calculating Product(5) recursively

2 Write in pseudo-code a recursively-defined function CountOfNoOfBallsInPyramid which calculates the number of billiard balls that can be stacked on top of each other if the base layer is contained in an equilateral triangle of side *n* balls as illustrated in *Figure 1.1.16.9* with a pyramid of side 4 balls for its base. (Hint: You may use the function in *Figure 1.1.16.6* in your answer).

Figure 1.1.16.9 Pyramid with triangular base

3 A recursive definition is one that is defined in terms of itself.

For example,

digit string is a digit or a digit followed by a digit string

This can be expressed in the following notation:

<digit string> ::= <digit> | <digit> <digit string>

where ::= means "consists of", | means "Or",

<digit> means 0 or 1 or 2 or 3 or 4 ... or 9, i.e. <digit> ::= 0|1|2|3|4|5|6|7|8|9

(a)

(i) Is the number 234 a valid digit string?

(ii) State the reasoning which you used in arriving at your answer to (a)(ii).

(b)

(i) Explain why the number 2·34 is not a valid digit string.

(ii) How would you modify the recursive definition of digit string to make 2·34 a valid digit string?

4 *Figure 1.1.16.10* shows a recursively-defined function Power(y, n) which calculates y^n for positive integers.

Complete the trace table shown in *Table 1.1.16.3* for the function call Power(2, 4).

Call no	Power(y, n)	Return
1	Power(2, 4)	
2		
3		
4		
5		

Table 1.1.16.3 Calculating Power(2, 4)

```
Function Power(y, n : Integer) : Integer
  If n >= 0
    Then
      If n = 0
        Then Return 1
        Else Return y * Power(y, n-1)
      EndIf
    Else Output 'n must be greater than or equal to 0'
  EndIf
End Function
```

Figure 1.1.16.10 Pseudo-code function Power which calculates y^n

Single licence - Abingdon School

Role of stack in recursion

It is possible to associate a set of local objects with a function/procedure – local variables, constants, types and other functions/procedures.

These local objects have no existence outside this function/procedure.

Each time such a function/procedure is activated recursively, a new set of local variables is created.

Although this new set will have the same names as any previous sets that are created during the recursion, their values are distinct.
Any conflict in naming is avoided by the rules of scope of identifiers: the identifiers always refer to the most recently created set of identifiers.
The same rule holds for function/procedure parameters which by definition are bound to the function/procedure.
Each new set of local variables is allocated space in an area of main memory called a **stack**.

The space allocated to each new set of variables is referred to as a **stack frame**.
The function/procedure's parameters also go into this stack frame.
In addition, each stack frame

- stores a return address – used by the computer to resume execution of the program at the correct place after executing a procedure
- stores the contents of registers needed at different stages in the execution of a program so that the values may be restored to their correct values after a function/procedure call. The function/procedure can write its own values into these registers whilst it is executing.

Stack frames are covered in *Chapter 2.1.4*.

Provoking stack overflow

Many recursive calls mean many stack frames. There is a limit on the amount of main memory set aside as a stack. There is always a danger of running out of stack space during recursion. When this happens **STACK OVERFLOW** is said to occur (or a stack full error occurs).
It is possible to get the computer to detect when stack overflow is about to occur. The recursion can then be aborted and the error message STACK OVERFLOW reported - see *Figure 1.1.16.11*.
If stack overflow is not detected and prevented then the consequences can be quite disastrous. The recursion will ignore the fact that stack space is exhausted. It will carry on creating new stack frames using main memory beyond the region of the stack. Usually this is main memory allocated to the program.
The consequence is that the program is overwritten and the computer "crashes". A reboot is then necessary.
The detection and prevention of stack overflow must be switched on at the time of compiling the recursive program.

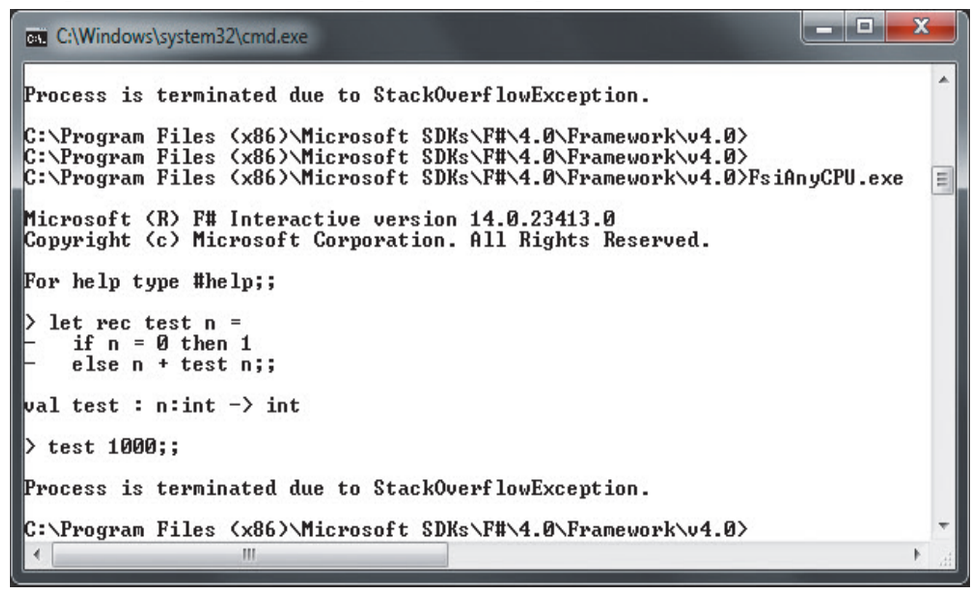


Figure 1.1.16.11 F# function which consumes stack frames when called with parameter n = 1000

Single licence - Abingdon School

When to use recursion and when not to use recursion

Recursive definitions such as shown in *Figure 1.1.16.12* do not guarantee that a recursive algorithm is the best way to solve a problem.

For example, Fibonacci (4) calls Fibonacci (3) and Fibonacci (2). But Fibonacci (3) also calls Fibonacci (2). Thus, the algorithm repeats computations unnecessarily.

```
Function Fibonacci(n : NonNegativeInteger) : NonNegativeInteger
  If n <= 1
    Then Return n
    Else Return Fibonacci(n - 1) + Fibonacci(n - 2)
  EndIf
End Function
```

Figure 1.1.16.12 Pseudo-code function Fibonacci

This is a strong case for using a non-recursive solution.
Non-recursive solutions rely on iteration to perform the computation.
For example, the recursive picture in *Figure 1.1.16.1* could be generated by iteration[1] as shown in *Figure 1.1.16.13*.

```
MagnificationFactor ← 1
Repeat
  Draw Picture * MagnificationFactor
  MagnificationFactor ← MagnificationFactor * 0.9
Until False
```

Figure 1.1.16.13 Non-recursive pseudo-code to generate recursive picture

The Fibonacci number sequence is defined mathematically by a recurrence relation as shown in *Figure 1.1.16.14*.
A recurrence relation is an equation that recursively defines a sequence of values, once one or more initial terms are given.

The Fibonacci sequence is defined using the recurrence

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0 \text{ and } F_1 = 1$$

Figure 1.1.16.14 Recurrence relation which generates the Fibonacci number sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

A non-recursive solution which uses iteration to generate the Fibonacci series is shown in *Figure 1.1.16.15*.
The non-recursive iterative solution generates Fibonacci numbers more efficiently in both time and space than its recursive equivalent because

- it involves fewer computations
- it only uses one stack frame

Recursive algorithms are particularly appropriate when the underlying problem or the data to be processed are defined in recursive terms and the definition does not involve simple recurrence relations such as illustrated by the Fibonacci sequence.

1 The Repeat Until *condition* loop repeats until the condition is True but it will never be True as we have used False for the value of the condition. We have created an infinite loop.

Single licence - Abingdon School

```
Function Fibonacci (n : NonNegativeInteger) : NonNegativeInteger
  Var
    First, Second, i : Integer
    First ← 0
    Second ← 1
    If n <= 1
      Then Return n
      Else
        For i ← 1 To (n - 1)
          Next ← First + Second
          First ← Second
          Second ← Next
        EndFor
        Return First
     EndIf
End Function
```

Figure 1.1.16.15 Non-recursive pseudo-code function Fibonacci

Questions

5 Explain why there is always a potential to run out of stack space when a recursive function is called.

6 *Figure 1.1.16.16* shows a recursively-defined function Factorial(n) which calculates n! for non-negative integers.
Complete the trace table shown in *Table 1.1.16.4* for the function call Factorial(4).

```
Function Factorial(n : PositiveInteger) : PositiveInteger
  If (n = 0) Or (n = 1)
    Then Return 1
    Else Return n * Factorial(n-1)
  EndIf
End Function
```

Figure 1.1.16.16 Pseudo-code function Factorial

7 Write the non-recursively-defined equivalent of the Factorial function using pseudo-code.

8 How many function calls are required when the non-recursive Factorial function is called with argument 4?

Call no	Factorial(n)	Return
1	Factorial(4)	
2		
3		
4		

Table 1.1.16.4 Calculating Factorial(4)

Programming tasks

1 In a programming language with which you are familiar, write a program which defines and uses the following recursively-defined functions which take positive integer arguments:

(a) Sum(n) (b) Factorial(n) (c) NoOfBilliardBallsInTriangle(NoAlongSide)
(d) Power(y, n) which calculates y^n for positive integers (e) Fibonacci(n)
(f) CountOfNoOfBallsInPyramid(NoAlongBaseSide)

2 For at least one of the recursively-defined functions in Task 1, try to cause stack overflow.

Single licence - Abingdon School

In this chapter you have covered:

- *The use of recursive techniques in programming languages (general and base cases and the mechanism for implementation)*
- *Solving simple problems using recursion*

Key concept

Base case:
When designing a recursive algorithm/procedure/function, it is important to avoid generating an infinite sequence of calls on itself else the recursion will never terminate. This requires that there must be a "way out" of the sequence of recursive calls. This is called the base case. The base case stops the recursion.
A recursive function/procedure F must be defined for at least one parameter/argument or group of parameters/arguments in terms that do not involve F.
This is usually a single non-recursive statement.

General case:
The general case is defined by the statement or statements in a recursive function/procedure F which reference F. The general case is responsible for the recursion.

Single licence - Abingdon School

1 Fundamentals of programming

1.2 Programming paradigms

Learning objectives:

- *Understand the structured approach to program design and construction*
- *Be able to construct hierarchy charts when designing programs*
- *Be able to explain the advantages of the structured approach*

Key term

Design:
To plan or mark out the form and method of solution.

Key term

Structured design:
Structured design is a disciplined process of deciding which components interconnected in which way will solve some well-specified problem.
A problem is divided into smaller and smaller sub problems, so that each sub problem will eventually correspond to a component of the system which solves the problem.
The partitioning process into smaller and smaller sub problems is done until the sub problems are

- manageably small, and
- solvable separately, i.e. relatively independent of one another.

Key principle

Good design:
Good design is an exercise in partitioning and organising the pieces of the system so that each is

- cohesive, and
- loosely coupled

1.2.2 Procedural-oriented programming

The structured approach to program design and construction

Software artefact

The end result of program design and construction is an artefact, a piece of software that when executed solves some given problem, hopefully the one required to be solved.

Design

"Design" means to plan or mark out the form and method of solution.

All forms of engineering including software engineering are characterised by the engineer's dissatisfaction with the achievement of just any old solution. Engineering seeks the best solution (judged by factors such as efficiency, maintainability, reliability), within recognised constraints, while making compromises required by working in the real world.

Software design takes a real-world problem and produces a plan for a computer-based solution.

Structured design

Structured design is a disciplined process of deciding which components interconnected in which way will solve some well-specified problem.

Structured design relies on a principle known since the days of Julius Caesar:

DIVIDE and CONQUER

A problem is divided into smaller and smaller sub problems, so that each sub problem will eventually correspond to a **component of the system** which solves the problem.

The partitioning process into smaller and smaller sub problems is done until the sub problems are

- manageably small, and
- solvable separately, i.e. relatively independent of one another.

Good design

Good design is an exercise in partitioning and organising the pieces of the system so that each is

- cohesive, and
- loosely coupled

We call the pieces of the system, modules or components or units.

The modules/components/units are plugged together to create the system.Modules/components/units can be implemented as subroutines (procedures and functions).

Single licence - Abingdon School

Cohesive/Cohesion

Cohesion measures the strength of the interconnection between elements within a module.

To achieve cohesion, highly interrelated parts of the real-world **problem** should be in the same piece of the **software system**, i.e. things that belong together should go together.

Figure 1.2.2.1 shows a schematic mapping from a problem in a real-world system to a highly-cohesive software architecture of a structured design for a computer-based system to solve the problem.

Figure 1.2.2.2 shows a specific example of a simple calculator system. Note the one-to-one mapping between real-world system and the computer-based system.

To achieve a solution in which modules/components are highly-cohesive the software system needs to be broken down into modules

- which are highly independent and
- which accomplish a single objective.

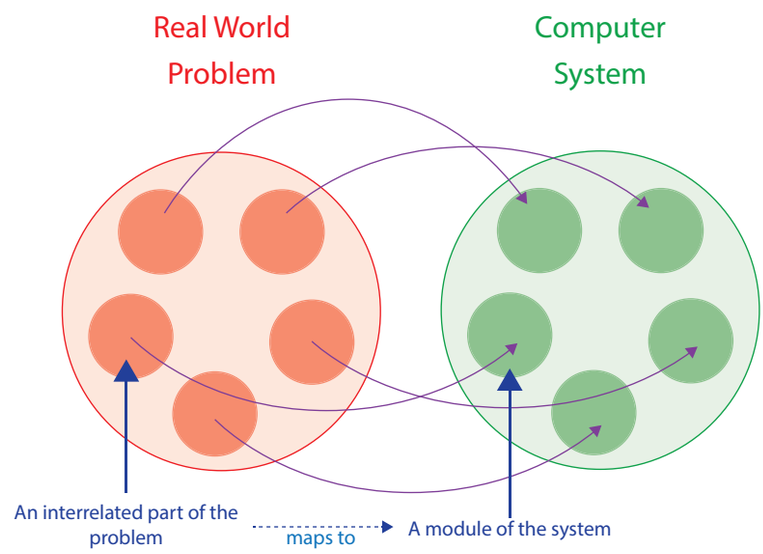


Figure 1.2.2.1 Mapping of problem parts to system modules

The aim is to achieve **functional cohesion**, i.e. modules which contain only one function or perform only one logically-related task.

For example,

- CalculateSquare
- GetDate

One good way to determine if modules are (functionally) cohesive is to write a phrase fully describing what the module does.

Analysis of this phrase can indicate whether or not the module is (functionally) cohesive. The module is not (functionally) cohesive if the result is one of the following:

- A compound sentence such as
  - Edit Student Name AND Test Scores
  - Get two operands AND Add these
- A lack of a specific object
  - Edit all Data
- Words relating to time such as
  - Initialisation
- Words such as "house-keeping" or "clean-up" because these imply more than one task.

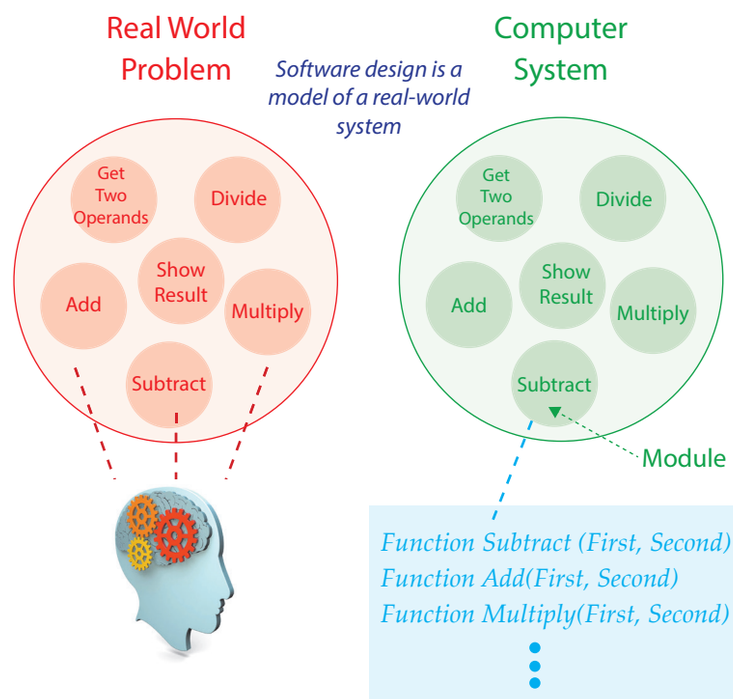


Figure 1.2.2.2 Highly interrelated parts of the problem should be in the same piece of the system

Single licence - Abingdon School

The functional description should not define the code within the module but how the module appears to the coder, e.g. `Subtract`.

The highest level of cohesion, referred to as **functional cohesion**, is a module in which

- every element is an integral part of a single function (task), and
- every element is essential to the performance of a single function (task)

Some advantages of using modules with high cohesion:

- The module can easily be replaced by any other serving the same purpose since what it does is localised within the module, i.e. it has no side-effects when it executes
- In the event of program failure,
  - it should be easier to locate the error as being in the module
  - it should be easier to discount the module as a source of error
- Different developers can work on individual modules because each is an independent unit.

Key terms

Cohesion:
Cohesion measures the strength of the interconnection between elements within a module.

Functional cohesion:
Functional cohesion is the highest level of cohesion. A functionally cohesive module is one in which

- every element is an integral part of a single function (task), and
- every element is essential to the performance of a single function (task)

Questions

1. What is the process known as design?
2. What is structured design?
3. What is meant by the term *cohesive* when applied to a module or component of a system?
4. A procedure in a calculator program is described as "Adding two operands and displaying the result".
   (a) Explain why this procedure is not considered functionally cohesive.
   (b) What can be done to achieve functional cohesion in this case?
5. State **two** advantages of using modules with high cohesion.

Loosely coupled

Coupling measures the strength of relationships between modules.

The objective of structured design is to minimise the coupling between modules, so that they will be as independent as possible.

The lower the coupling, the less likely that other modules will have to be considered in order to

- create a module
- understand a given module
- debug a given module
- change a given module.

Coupling results from connections.

A connection exists when an element of code references a location in memory defined outside the module.

Some connection must exist among modules in a program, or else they would not be part of the same program.

The objective is to **minimise the coupling among modules**.

Key term

Coupling:
Coupling measures the strength of relationships between modules. The objective of structured design is to minimise the coupling between modules, so that they will be as independent as possible. Coupling results from connections. A connection exists when an element of code references a location in memory defined outside the module.

Single licence - Abingdon School

To minimise coupling among modules:

- Subroutines or modules should only be allowed to access that data which they needs to perform their assigned task
- All data transfer between modules is visible in the module parameters
- There must be no hidden flows of data via global variables or shared data areas
- There should be no control information passing between modules, e.g. Boolean flags
- The number of module parameters should be minimal.

Loose coupling is achieved when a module's data interface with other modules is its module parameter list. Here we interpret module to mean a subroutine (procedure/function).

In VB.NET, a procedure `DisplayMessage` is defined with the language keywords `Sub` and `End Sub` as shown in the example in *Figure 1.2.2.3*. In VB.NET, the term module can be interpreted as equivalent to program.

```
Module ProcedureInterface
    Sub Main()
        DisplayMessage("Hello World")
        Console.ReadLine()
    End Sub
    Sub DisplayMessage(ByVal Message As String)
        Console.WriteLine(Message)
    End Sub
End Module
```

Figure 1.2.2.3 VB.NET program with procedure DisplayMessage

Figure 1.2.2.4 shows a procedure `DisplayMessage` defined with the language keywords `Procedure` and `Begin` and `End`.

```
Program DisplayMessageExample;
Procedure DisplayMessage (Message : String);
  Begin
    Writeln(Message);
  End;
Begin
  DisplayMessage('Hello World');
  Readln;
End.
```

Figure 1.2.2.4 Pascal program with procedure DisplayMessage

The data interface for VB.NET `DisplayMessage` is the parameter list `ByVal Message As String`.

The data interface for Pascal `DisplayMessage` is the parameter list `Message : String`.

The procedure `DisplayMessage` is called with actual parameter value "Hello World" and 'Hello World', respectively, in *Figure 1.2.2.3* and *Figure 1.2.2.4*.

The Pascal program in *Figure 1.2.2.5* has declared a global variable `Message` which in Pascal is visible within procedure `DisplayMessage`. The procedure `DisplayMessage` has dispensed with a procedure parameter list and instead relies on accessing the global variable `Message`.

```
Program DisplayMessageExampleGlobal;
Var
  Message : String;
Procedure DisplayMessage;
  Begin
    Writeln(Message);
  End;
Begin
Message := 'Hello World';
  DisplayMessage;
  Readln;
End.
```

Figure 1.2.2.5 Pascal program with procedure DisplayMessage without a parameter list

The procedure `DisplayMessage` in *Figure 1.2.2.5* has a higher coupling with its program than `DisplayMessage` does in *Figure 1.2.2.4*.

Single licence - Abingdon School

Questions

6 What is *coupling* when applied to modules or components of a system?

7 Why should coupling between modules be minimised?

8 How does coupling between modules arise?

9 How can coupling between modules be minimised?

10 Give **two** reasons why the pseudo-code in *Figure 1.2.2.6* could be considered a poor design by structured design standards.

```
Program Exercise1
Var
  No1, No2 : Integer
Procedure AddTwoNumbers
  Output No1 + No2
End Procedure
Begin
  No1 ← 4
  No2 ← 5
  AddTwoNumbers
End
```

Figure 1.2.2.6 Pseudo-code to add two numbers

11 Suggest **one** reason why the procedure `DisplayMessage` in *Figure 1.2.2.7* could be considered poorly designed by structured design standards.

12 How could the design of procedure `DisplayMessage` and the program be changed to reflect good structured design?

```
Module Module1
   Sub Main()
       DisplayMessage("hello world", False)
       DisplayMessage("hello world", True)
       Console.ReadLine()
   End Sub
   Sub DisplayMessage(ByVal Message As String, ByVal UpperCase As Boolean)
         If UpperCase Then
             Console.WriteLine(StrConv(Message, vbUpperCase))
         Else Console.WriteLine(Message)
         End If
     End Sub
 End Module
```

Figure 1.2.2.7 VB.NET program that uses a procedure DisplayMessage with a flag parameter

13 A procedure `Initialise` is defined with a long parameter list of reference formal parameters, i.e. each formal parameter is a pointer type which when replaced by an actual parameter will point to a variable outside the procedure:

```
Procedure Initialise(Var a, b, c, d, e, f, g, h, i, j, k, l, m, n, p : Integer)
```

Why is this considered a poor design by the standards of structured design?

14 A program contains a module A, module B and other modules. Module A writes to a disk file and module B reads from a disk file. When module A is engaged in writing to a disk file no other module may access the same disk file. When module A has finished writing to a disk file it closes it. The program calls module A on file Z followed by module B on file Z. Would you describe the coupling between module A and module B as low or high? Explain.

Single licence - Abingdon School

Simple calculator example which illustrates cohesion and loose coupling

Suppose that we are required to demonstrate structured design in a simple way.

We choose to do this by designing and creating a very simple calculator that is limited to performing integer arithmetic on two given operands.

The arithmetic operations that must be supported are

1. Add
2. Subtract
3. Multiply
4. Divide (Integer division)

The major "pieces" of the system are

1. The system must display to the user the choices which are available for arithmetic operation
2. The system must obtain the user's choice of arithmetic operation
3. The system must obtain two operands from the user
4. The system must carry out the chosen on the two operands
5. The system must have a "piece" to do each of the following
   5.1 Add
   5.2 Subtract
   5.3 Multiply
   5.4 Divide (Integer division)
6. The system must display the result to the user.

Key term

Hierarchy chart:
In structured design of software, the partition of a software system into its component parts can expressed as a hierarchy chart which shows the components (subroutines) and how they are interconnected.

Structured design is then used to decide which modules/components to use to solve this problem.

Structured design tells us that each piece identified above in 1, 2, 3, 4, 5.1, 5.2, 5.3, 5.4, 6 above should be a module of the system - *Figure 1.2.2.2*.

In this simple example, these modules/components can then be implemented in a programming language as subroutines.

Hierarchy charts

We show the **software architecture** of the simple calculator system resulting from the structured design approach in *Figure 1.2.2.8*. We call this a **hierarchy chart**.

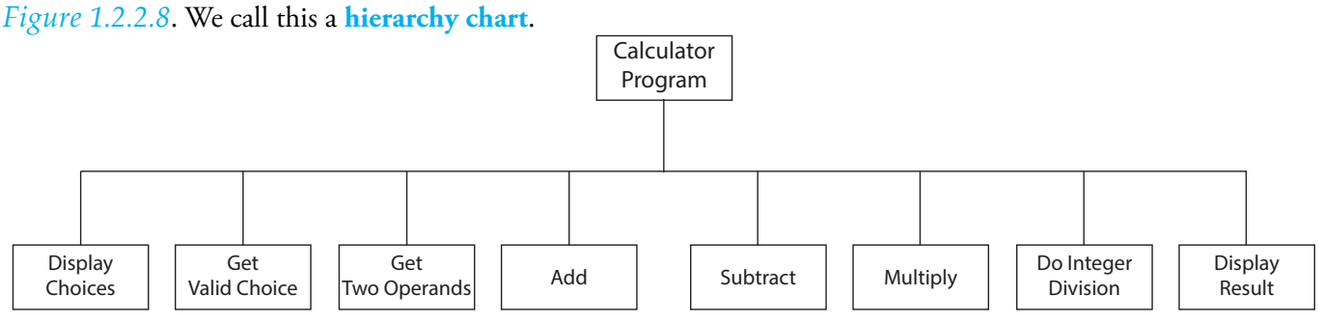


Figure 1.2.2.8 Hierarchy chart showing the software architecture of the simple calculator system

This doesn't tell the whole story because the chart doesn't show the coupling between the modules/components.
Loose coupling is achieved when a module's data interface with other modules is its module parameter list.
By adding the parameter list for each module to the chart we can show that its data interface is indeed that required for low coupling - *Figure 1.2.2.9*.
The meaning of the symbols used in *Figure 1.2.2.9* are shown in *Figure 1.2.2.10*.

Single licence - Abingdon School

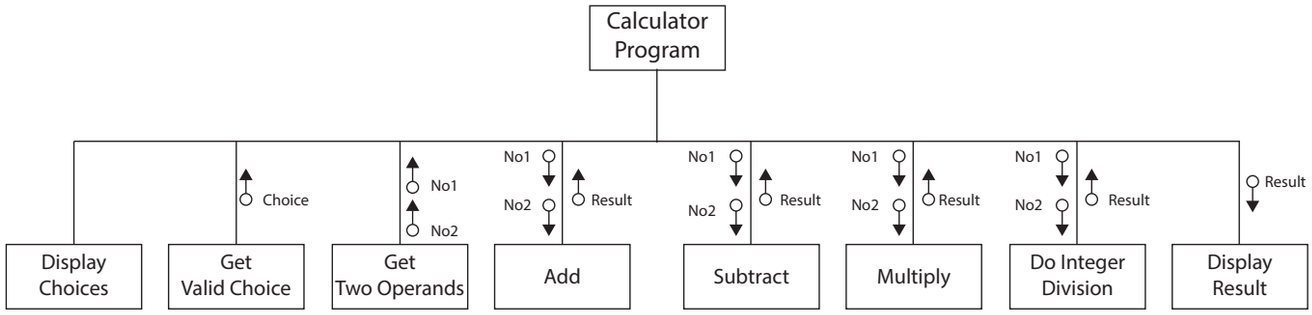


Figure 1.2.2.9 Hierarchy chart showing the software architecture of the simple calculator system and the parameter list/data interface of each module

The parameter `Choice` is the only parameter of module `GetChoice`. It is an OUT parameter. The value returned by `GetChoice` is used by the Calculator Program to choose from among modules `Add`, `Subtract`, `Multiply` and `DoIntegerDivision`.

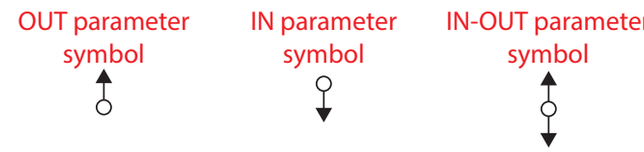


Figure 1.2.2.10 The meaning of the symbols shown in Figure 1.2.1.9

Module `Add` has three parameters in its data interface: `No1`, `No2` and `Result`. `No1` and `No2` are IN parameters. `Result` is an OUT parameter.

Sometimes a module needs to use an IN-OUT parameter because it processes the value passed in by an IN-OUT parameter and then exports a new value from the module in the same IN-OUT parameter.

The chart in *Figure 1.2.2.9* just shows the architectural breakdown of the system into its software components and the data interfaces between these components. If we add control to the hierarchy chart as shown in *Figure 1.2.2.11* we will get the program's structure, e.g. if the value of Choice is, say 'S' or 's', the program should choose to execute the `Subtract` module and not `Add`, `Multiply` or `DoIntegerDivision`. This is indicated with the option symbol o placed in the top-right corner of the module rectangle.

The **control hierarchy chart** shown in *Figure 1.2.2.11* is read from left to right, i.e. module `DisplayChoice` is executed first followed by module `GetChoice`, and so on. A `Repeat Until Choice In ['Q', 'q']` has been added to indicate that the left to right sequence is repeated until the user chooses to quit.

Figure 1.2.2.12 shows the program structure in Pascal using procedures. The program structure could also have been written using functions if required. You can see that the control hierarchy chart maps one-to-one to the structure of its program equivalent.

The option symbol o placed in the top-right corner of the module rectangle for `Add`, `Subtract`, `Multiply` and `DoIntegerDivision` has been mapped to a Case statement (Select Case, if / elif in other languages) in *Figure 1.2.2.12*.

Key term

Control hierarchy chart:
A control hierarchy chart shows the structure of the program in chart form.
It can also show the parameter list for each module/subroutine in the chart.

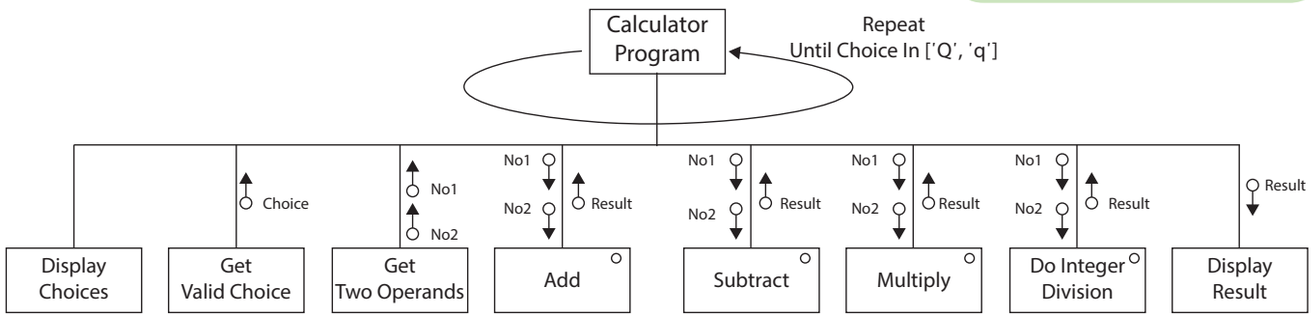


Figure 1.2.2.11 Control hierarchy chart showing the software architecture of the simple calculator system, the parameter list/data interface of each module and the program's structure

Single licence - Abingdon School

```
Program SimpleCalculator;
  Procedure DisplayChoices;
    Begin
    End;
  Procedure GetValidChoice(Var Choice : Char);
    Begin
    End;
  Procedure GetTwoOperands(Var No1, No2 : Integer);
    Begin
    End;
  Procedure Add(No1, No2 : Integer; Var Result : Integer);
    Begin
    End;
  Procedure Subtract(No1, No2 : Integer; Var Result : Integer);
    Begin
    End;
  Procedure Multiply(No1, No2 : Integer; Var Result : Integer);
    Begin
    End;
  Procedure DoIntegerDivision(No1, No2 : Integer; Var Result : Integer);
    Begin
    End;
  Procedure DisplayResult(Result : Integer);
    Begin
    End;
Var
  UsersChoice : Char = 'A';
  FirstNo : Integer = 0;
  SecondNo : Integer = 0;
  Answer : Integer = 0;
Begin
  Repeat
    DisplayChoices;
    GetValidChoice(UsersChoice);
    GetTwoOperands(FirstNo, SecondNo);
    Case UsersChoice of
      'A', 'a' : Add(FirstNo, SecondNo, Answer);
      'S', 's' : Subtract(FirstNo, SecondNo, Answer);
      'M', 'm' : Multiply(FirstNo, SecondNo, Answer);
      'D', 'd' : DoIntegerDivision(FirstNo, SecondNo, Answer);
      'Q', 'q' : ;
    End;
    DisplayResult(Answer);
  Until UsersChoice In ['Q', 'q'];
End.
```

Var denotes that parameter is IN-OUT but in this example it is used as an OUT parameter

Formal parameter

IN only parameters

Actual parameters

Figure 1.2.2.12 Program structure in Pascal of simple calculator written as procedures

Programming task

1. Create a simple calculator program based on the program structure shown in *Figure 1.2.2.12* in a programming language of your choice.

Single licence - Abingdon School

Structured design does not address the issue of how to write the program for the body of each procedure only the division of a system into its components and how those components fit together to produce a solution.

To write the program for the body of the procedures we use the **principles of structured programming**.

Structured programming

The principles

Structured programming advocates a *disciplined approach* to the construction of programs in order to avoid problems which can arise if the approach is not disciplined.

At the lowest level, the main principles of structured programming are concerned with the **flow of control** through a program unit such as shown in *Figure 1.2.2.13*. The most fundamental idea is that the **main flow** of control through a program unit should be from **top to bottom** of the program.

This translates to **every block** (sequence, selection and iteration) should have **one entry point and one exit point**.

In *Figure 1.2.2.13* this means that the flow of control enters at the top of a dotted block and exits at the bottom. Within a block, the flow doesn't have to be forward, e.g. iteration.

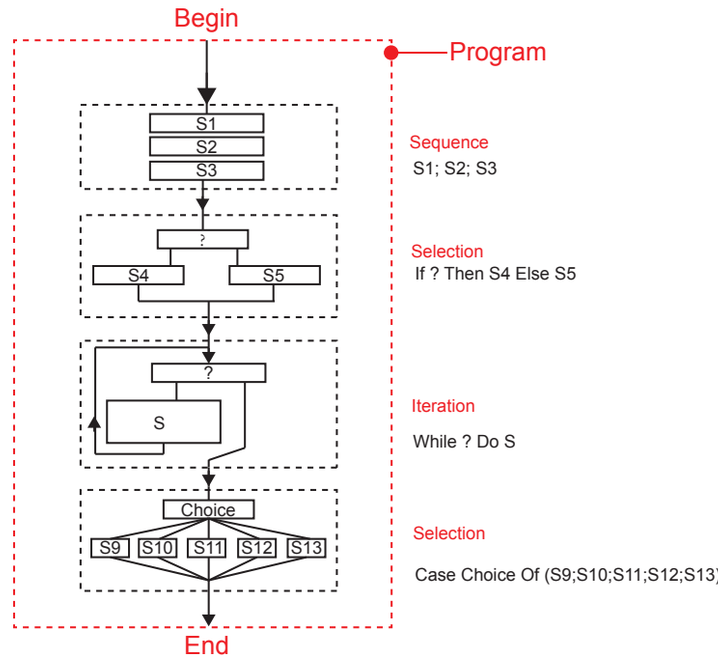


Figure 1.2.2.13 Program unit showing flow of control from top to bottom through the three basic control constructs, sequence, selection and iteration

There are three basic control constructs necessary to build any program:

- Sequence - a list of program statements which are executed one after another (i.e. in sequence),
  e.g. `x := x + 1; y := y + 2; z := 6;`
- Selection - a means of choosing between two or more sequences of statements depending on the value of some condition(s),
  e.g. `If Then Else`
- Iteration - a construct to allow controlled repetition of a sequence of statements, e.g. `While x < 5 Do Something`.

It can be shown that these **three constructs** are sufficient to implement the control structure of any algorithm.

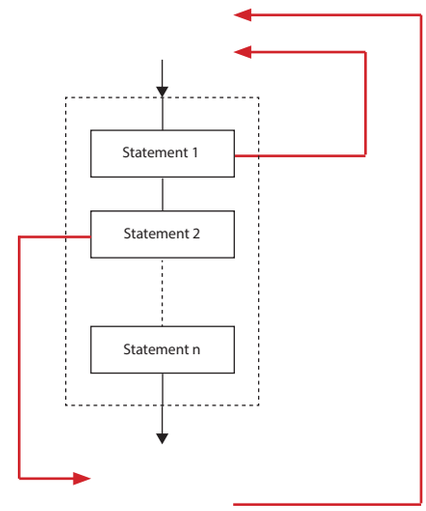


Figure 1.2.2.14 Part of a program unit showing flow of control jumping backwards and forwards between blocks

Structured programming is also known as *gotoless* programming because it avoids the use of the **Go To** control construct. If a label (e.g. `step1`) is attached to a statement in a program then the flow of control can jump to the labelled statement from anywhere in the program which references the label in a `Go To step1`. *Figure 1.2.2.14* shows how the flow of control can jump backwards and forwards when Go To statements are used; each red line is a transfer of control caused by a Go To statement.

Single licence - Abingdon School

In Pascal, a numeric label is declared using the language keyword `Label`. In *Figure 1.2.2.15* a numeric label 100 is declared with `Label 100` and then used by `GoTo 100` in the `Begin End` program block.
Structured programming doesn't mean that Go To should be avoided at all cost. There is one situation where it is the preferred solution. This is when something has gone wrong within the program, e.g. an attempt is made to open a nonexistent file. In this circumstance, it is often better to jump to the end of the program because to avoid use of Go To in this circumstance can often result in a difficult to understand and convoluted program.
The flow is still from top to bottom in this particular use of Go To and the structured programming principle, "*The main flow of control through a program unit should be from top to bottom of the program*" is not violated.

```
Program GoToExample;
Label 100;
Begin
  100:
  Writeln('Hello World');
  Writeln('Hello World');
  Writeln('Hello World');
  GoTo 100;
End.
```

Figure 1.2.2.15 Example of the use of Go To

We use structured programming because it leads to programs which are **easier to understand, maintain and reason about**.

In "Notes on Structured Programming" Edgar Dijkstra wrote:

> "*A program is never a goal in itself; the purpose of a program is to evoke computations and the purpose of the computations is to establish a desired effect. Although the program is the final product made by the programmer, the possible computations evoked by it-the "making" of which is left to the machine! - are the true subject matter of his trade. For instance, whenever a programmer states that his program is correct, he really makes an assertion about the computations it may evoke.*
>
> *The fact that the last stage of the total activity, viz. the transition from the (static) program text to the (dynamic) computation, is essentially left to the machine is an added complication. In a sense the making of a program is therefore more difficult than the making of a mathematical theory: both program and theory are structured, timeless objects. But while the mathematical theory makes sense as it stands, the program only makes sense via its execution."*
>
> "*In vague terms we may state the desirability that the structure of the program text reflects the structure of the computation. Or, in other terms, "What can we do to shorten the conceptual gap between the static program text (spread out in "text space") and the corresponding computations (evolving in time) ?"*"
>
> Page 16, http://www.informatik.uni-bremen.de/agbkb/lehre/programmiersprachen/artikel/EWD-notes-structured.pdf

A goal of structured programming is **to reduce the conceptual gap between the program text and the corresponding computations**. In other words, how can you be sure that your program does what it is required to do, i.e. meets its specification?
Writing programs that are easy to understand contributes to achieving this goal even if it is done at the expense of efficiency, e.g. execution time and memory requirements.
Structured programming requires that

1. The main flow of control through a program unit should be from top to bottom of the program.
2. Program blocks should have one entry point and one exit point
3. Meaningful identifiers should be used for variables, subroutines (procedures and functions), etc, to aid readability and understanding
4. Indentation should be used that reflects the structure of the program and which aids readability and understanding
5. The following control constructs should be used: sequence, selection and iteration
6. Go Tos should be avoided in all but the one case of a major error in the computation which would require, if handled in a structured programming way, convoluted and difficult-to-understand program source code to be written
7. The use of global variables which are used in a global way should be avoided
8. Data should be passed to subroutines in subroutine parameters and results returned through subroutine parameters or preferable as a function return data type
9. Local variables should be used for handling data within subroutines.

Single licence - Abingdon School

Questions

15 What is meant by structured programming?

16 What does structured programming have to say about the flow of control in programs?

17 Give **two** benefits of using structured programming.

Stepwise refinement

The focus of structured design is the identification of the components of the system and how they interact, i.e. the software architecture of the computerised system, and not on the internal design of the components.

For the **internal design of the components**, i.e. the program source code for subroutines and the program which calls these subroutines, we use structured programming.

We take as our starting point the control hierarchy chart for the two-player game *noughts and crosses* - *Figure 1.2.2.16*.

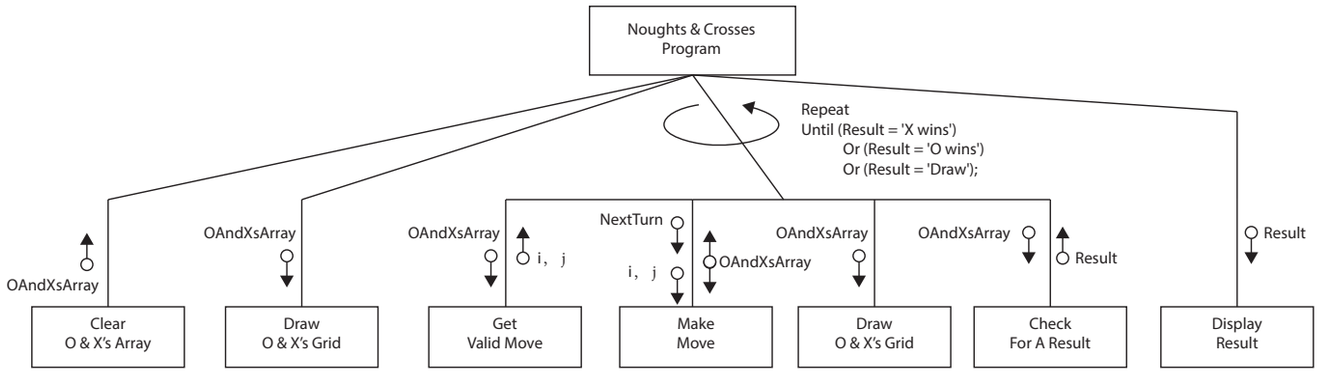
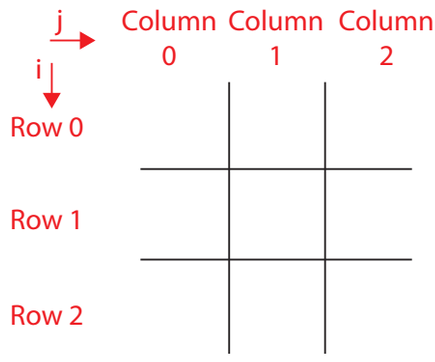


Figure 1.2.2.16 Control hierarchy chart for the game noughts and crosses

We model the two-dimensional grid of the noughts and crosses game in a two-dimensional array type `GridType` - *Figure 1.2.2.17*.




```
Type
   GridType = Array[0..2,0..2] Of Char;
Var
  OAndXsArray : GridType;
  Result : String;
  NextTurn : Char;
  i, j : Integer;
```

Figure 1.2.2.17 Modelling the game noughts and crosses

To reference a cell of this grid we will use integer variables `i` and `j`, e.g. the top-left cell will have `i`, `j` coordinates

`i = 0` and `j = 0`.

Each player of the two-player game takes it in turn to choose an unoccupied cell in which to place their symbol.
One player uses the symbol X and the other the symbol O.
In this game, the player with symbol X starts.
The symbol that is played next is stored in the character variable `NextTurn`.

Single licence - Abingdon School

For symbol X we use character value 'X', and for symbol O we use character value 'O'.

The result of a player's move is stored in string variable `Result`. The value of result could be one of

'X wins', 'O wins', 'Draw' or 'No result so far'.

Once we have designed data structures, we can design the program to process the data stored in these data structures.
The control hierarchy chart *Figure 1.2.2.16* already reflects the program structure so it is a relatively straightforward mapping task to produce the textual form of the program structure as shown in *Figure 1.2.2.18*.
The next task is to design the internal structure of each subroutine and then construct this structure in a programming language. We do this using the control structures of structured programming, sequence, selection and iteration and a technique called **stepwise refinement**.

```
ClearArray(OAndXsArray)
DrawGrid(OAndXsArray)
NextTurn ← 'X'
Repeat
  GetValidMove(i, j, OAndXsArray)
  MakeMove(i, j, OAndXsArray, NextTurn)
  DrawGrid(OAndXsArray)
  Result ← CheckForAResult(OAndXsArray)
  If Result = 'Win'
    Then
      If NextTurn = 'X'
        Then Result ← 'X wins'
        Else Result ← 'O wins'
  If Result = 'No result so far'
    Then
      If NextTurn = 'X'
       Then NextTurn ← 'O'
       Else NextTurn ← 'X'
Until (Result = 'X wins')
        Or (Result = 'O wins')
        Or (Result = 'Draw')
DisplayResult(Result)
```

Figure 1.2.2.18 Program structure for the game noughts and crosses

Stepwise refinement starts with the **major steps**.
Each major step is then refined into a **more detailed sequence of steps**.
Each one of these more detailed steps is then refined and so on until a stage is reached where the steps can be replaced by programming language statements.
For example, the possible stages of stepwise refinement for the subroutine `CheckForAResult` are shown in *Figure 1.2.2.19*, *Figure 1.2.2.20* and *Figure 1.2.2.21*.

Major steps:

```
1. Check rows
2. Check columns
3. Check diagonals
4. If No result
     Then check for a draw
```

Figure 1.2.2.19 Major steps

First level of refinement:

```
1. Check rows
   1.1 For row 0 To 2 Do check row
2. Check columns
   2.1 For column 0 To 2 Do check column
3. Check diagonals
   3.1 Check top left to bottom right diagonal
   3.2 Check top right to bottom left diagonal
4. If No result
     Then check for a draw
   4.1 If no result
         Then
           If grid full
             Then result a draw
           EndIf
       EndIf
```

Figure 1.2.2.20 First level of refinement

There are no hard and fast rules to apply to the process of refinement other than each level should be a step closer to the solution, i.e. the actual programming language statements. Rather than document each stage as has been done here, it is often better to do the refinement inside the actual subroutine in the program development environment. Applying the discipline of using successive refinements in constructing a program is more important than formally generating refinements on paper.

Single licence - Abingdon School

However, paper is a good place to start until you become more skilled and able to iterate the refinement process mentally as you apply the skill directly inside a program development environment.

Second level of refinement:

```
1. Check rows
   1.1 For row 0 To 2 Do check row
     1.1.1 For i ← 0 To 2 Do
             If (OAndXsArray[i,0] = OAndXsArray[i,1])
                  And (OAndXsArray[i,1] = OAndXsArray[i,2])
               Then
                 If (OAndXsArray[i,0] <> ' ') Then Outcome ← 'Win'
             EndIf
           EndFor
2.  Check columns
   2.1 For column 0 To 2 Do check column
     2.1.2 For j ← 0 To 2 Do
             If (OAndXsArray[0,j] = OAndXsArray[1,j])
                  And (OAndXsArray[1,j] = OAndXsArray[2,j])
               Then
                 If (OAndXsArray[0,j] <> ' ') Then Outcome ← 'Win'
           EndFor
3. Check diagonals
   3.1 Check top left to bottom right diagonal
     3.1.1 If (OAndXsArray[0,0] = OAndXsArray[1,1])
                And (OAndXsArray[1,1] = OAndXsArray[2,2])
             Then
               If (OAndXsArray[0,0] <> ' ') Then Outcome ← 'Win'
            EndIf
   3.2 Check top right to bottom left diagonal
     3.2.1 If (OAndXsArray[0,2] = OAndXsArray[1,1])
                And (OAndXsArray[1,1] = OAndXsArray[2,0])
             Then
               If (OAndXsArray[0,2] <> ' ') Then Outcome ← 'Win'
           EndIf
4. If No result Then check for a draw
   4.1 If no result
         Then
           If grid full Then result a draw
         EndIf
     4.1.1 If Outcome = 'No result so far'
            Then
              NoOfMovesSofar ← 0
              For i ← 0 To 2 Do
                For j ← 0 To 2 Do
                  If OAndXsArray[i,j] <> ' ' Then NoOfMovesSoFar ← NoOfMovesSoFar + 1
                EndFor
              EndFor
              If NoOfMovesSoFar = 9 Then Outcome ← 'Draw'
           EndIf
```

Figure 1.2.2.21 Second level of refinement

Single licence - Abingdon School

Programming task

2 Using stepwise refinement create a noughts and crosses program based on the program structure shown in *Figure 1.2.2.18* in a programming language of your choice.

Questions

18 Assuming that the sub tasks in the following are to be implemented as subroutines, create a hierarchy chart with control information and subroutine parameter lists for a program which

(a) collects three integers from the keyboard and calculates and displays their average.
(b) collects a letter from the keyboard, then tests and displays whether the letter is uppercase or not.
(c) collects an integer from the keyboard then calculates and displays its square, cube and fourth power.

Advantages of the structured approach

You have been introduced to the structured approach via very simple systems involving a small number of tasks where the "size" of the computation, i.e. the amount of information and the number of operations involved in it are relatively small.

When the problem to solve is really small, it would be easier not to use the computer at all and instead to do it by hand. The automatic computer is better employed on large computations which humans cannot do.

However, in order to manage such scale of computation successfully a disciplined approach to developing software is required.

The structured approach breaks the problem/system into manageable modules which have

- functional cohesion, i.e. modules which contain only one function or perform only one logically-related task. This enables a module to be easily replaced by another serving the same purpose.
- low coupling so that the modules are as independent as possible. The lower the coupling, the less likely that other modules will have to be considered in order to
  - create a module
  - understand a given module
  - debug a given module
  - change a given module.

Modules which are functionally cohesive, loosely coupled and for which all data transfer in and out of the modules is visible in the module parameters, enable programmers to work on individual modules without needing knowledge of the rest of the system.

The degree of intellectual effort (measured in some loose sense) to understand well-structured programs tends to be proportional to their program length (measured in some equally loose sense). This is important when reasoning about a program to check that it meets its specification. It is a more challenging task to do this with programs which are not well-structured.

Psychology research tells us that humans manage information in "chunks" and that the human brain is able to handle up to five pieces of information, maybe seven at a pinch. Stepwise refinement fits this limitation of the human brain well.

Single licence - Abingdon School

Questions

19 State **three** advantages of the structured approach and explain why these are advantages.

Information

There are two other software design paradigms:

- Object-oriented
- Function programming

In object-oriented design the nature of objects leads to the creation of loosely coupled systems.

In object-oriented design the representation of an object is concealed within that object and is not visible to external components.

The system does not have a shared state and therefore any object can be replaced by another object with the same interface.

Some of the skills you develop following a structured approach to constructing software and the principles of structured design and programming you acquire in AS Computer Science overlap with Object-Oriented Design (OOD) and Object-Oriented Programming (OOP) which is part of the A Level Computer Science course. This course will also introduce you to the functional programming paradigm and will lay the foundations for reasoning about program correctness.

In this chapter you have covered:

- The structured approach to program design and construction
- Constructing hierarchy charts when designing programs
- The advantages of the structured approach

Single licence - Abingdon School

1 Fundamentals of programming

1.2 Programming paradigms

Learning objectives:

- *Be familiar with the concept:*
  - *class*
  - *object*
  - *instantiation*
  - *encapsulation*
  - *inheritance*
  - *polymorphism*
  - *overriding*

1.2.3(1) Object-oriented programming

Objects

One dictionary definition of an object is
"*A material thing that can be seen and touched.*"

Figure 1.2.3.1 shows an image of a stack of books. A book is an example of an object, a material thing which can be seen and touched. Likewise, the stack itself is an object.

Figure 1.2.3.1 Stack of books

Another dictionary definition of an object is
"*A person or thing to which a specified action or feeling is directed.*"

This definition also applies to a book because a book can be picked up and read. The specified action or two actions are "*pick up*" and "*read*".
In the case of the stack object, a book can be "pulled" from the stack or a book can be "pushed" onto the stack.

In object-oriented programming objects in the real world and the actions which can be applied to these objects are modelled in software.

For example, the height of the stack of books can be recorded in a software equivalent of the stack object. This software object equivalent also includes code (or, more accurately, access to this code) to get the height of the stack - *Figure 1.2.3.2*.
We can model many stacks of books with each represented by its own equivalent software object, e.g. *Stack1*, *Stack2*, *Stack3*, ..., *Stackn*, as shown in *Figure 1.2.3.2*.

Key principle

Object-oriented programming:
All aspects of a software system or subsystem are represented as a collection of interacting objects.
The goal of the interacting objects is to model some person, place, thing or idea.

Key concept

Object:
Data plus the operations that act on the data are called an object.

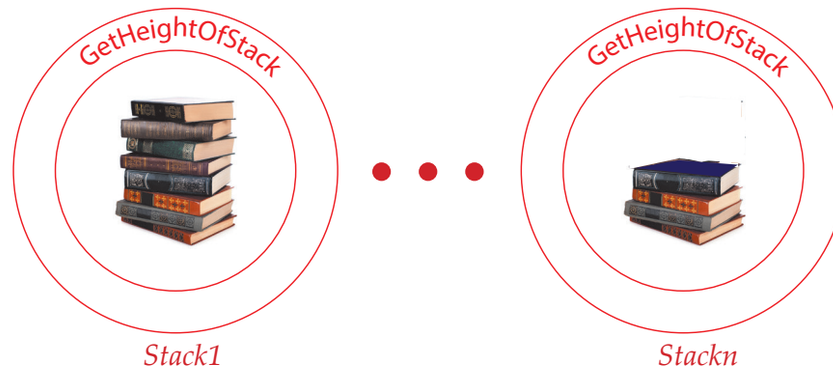


Figure 1.2.3.2 Multiple stack objects

The specific objects shown in *Figure 1.2.3.2* share a common pattern which we call their class or the class they belong to.

Single licence - Abingdon School

Abstraction in object-oriented programming

Abstraction is the ability to engage with a concept while ignoring some of its details, e.g. when considering the concept of a house, we ignore the individual details of its doors, that is, whether they are made of wood, metal, plastic, and where the letter box is located, etc.

A class in object-oriented programming is similar.

At the class-level of abstraction the focus is on properties (also called attributes) and the command/query operations that are of interest.

For example, some properties of a house are

- NumberOfBedrooms
- Address
- NameOwner
- RateableValue

and some query operations are

- GetNumberOfBedrooms
- GetAddress
- GetNameOfOwner
- GetRateableValue

and some command operations are

- ChangeNameOfOwner
- ChangeRateableValue

Properties

NumberOf Bedrooms
Address
Name OfOwner
RateableValue

Operations

GetNumberOfBedrooms
GetAddress
GetNameOfOwner
GetRateablevalue
ChangeNameOfOwner
ChangeRateableValue

Key concept

Class:

A class is a template from which actual objects may be created. The class definition lists the characteristics of the object that are to be recorded, e.g. for a car object, the number of seats, size of engine, etc. The class will also define how the object can behave, e.g. accelerate forwards, slow down when brakes are applied, etc. An actual instance of an object – my car - will have a state that it is currently in, e.g. parked and locked and a set of behaviours.

Key point

Objects share behaviour but not state:

Objects share method code but not attribute values. Each object maintains its own attribute values.

There is only ever one copy of each method code.

Once we have a class abstraction of a house, let's say called *House*, it is then possible to create specific instances of this class.

Each instance is an object of the class *House*. So we could create *House* objects, *House1*, *House2*, etc.

The properties of each would be populated with specific values, e.g. object *House1* property *NameOfOwner* might have value 'Fred Bloggs' whilst object *House2* property *NameOfOwner* might have value 'Mary Poppins'.

Both would **share access to the code** for each of the operations so that each object could be queried and commanded.

These operations are referred to as **methods of the object** and are coded in software as functions and procedures.

When a function or procedure of an object is called we say that a **method is invoked**.

Object-oriented programming takes abstraction a little further with something called **encapsulation**.

Whilst abstraction presents a view of a real-world object which is a high level one, encapsulation says you are not allowed to look at an object at any other level of detail than this high level one.

Stack example

Figure 1.2.3.3 shows an abstraction of a stack, a Last In First Out (LIFO) abstract data type, and a design which identifies attributes and methods for an object which implements an integer stack with space to store 100 integers.

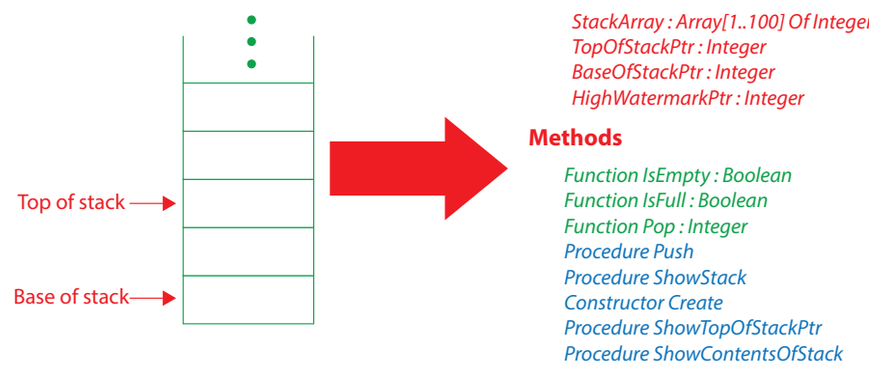


Figure 1.2.3.3 Stack abstraction: query operations in green, command operations in blue

Single licence - Abingdon School

In Object Pascal, a stack class would be defined as shown in *Table 1.2.3.1*.

```
Type
  TStackType = Class
                 Private
                   StackArray : Array[1..100] Of Integer;
                   TopOfStackPtr : Integer;
                   BaseOfStackPtr : Integer;
                   HighWatermarkPtr : Integer;
                 Public
                   Function IsEmpty : Boolean;
                   Function IsFull : Boolean;
                   Function Pop : Integer;
                   Procedure Push;
                   Procedure ShowStack;
                   Constructor Create;
                   Procedure ShowTopOfStackPtr;
               End;
```

Private means these attributes can only be accessed via the methods in the public section.

Public means available to be called from an application.

Table 1.2.3.1 Definition of class TStackType in Object Pascal

Key concept

Object instantiation:
An object is an instance of a class.
An object requires memory for its data and a mechanism for linking to the code for its methods. This code is shared with other objects belonging to the same class. Attribute data is not shared however. Each object maintains its own state, i.e. the values of its attributes.

When the focus is on objects which belong to the class `TStackType` we say that `StackArray`, `TopOfStackPtr`, `BaseOfStackPtr` and `HighWatermarkPtr` are called **attributes**.
Objects in a program often represents things in the real world, and then an object's attributes are properties of the thing.
A programming language object is a kind of container for data and this container requires computer memory to store the object's data.
In the case of a stack object, memory is needed for the `StackArray`, `TopOfStackPtr`, `BaseOfStackPtr` and `HighWatermarkPtr` attribute data plus memory for pointer(s) to the code for the object's methods `IsEmpty`, `IsFull`, `Pop`, `Push`, `ShowStack` and `ShowTopOfStackPtr`.
The code which implements these methods will be shared amongst all class `TStackType` objects created in memory.

`TStackType`'s `Create` method is special. It is a **constructor** that is used to grab the memory needed by an object when it is **instantiated**. The name given to the process of creating an object is called **instantiation**.
A constructor is also able to initialise the attributes of every object it creates, e.g. `TopOfStackPtr ← 0`.

Table 1.2.3.2 shows a program which creates a single stack object.

`TStackType.Create` creates the stack object in memory and `MyStack` stores a reference to this object.

To show the value of the top of stack pointer, `TopOfStackPtr`, we reference the object and invoke the method, `ShowTopOfStackPtr` as follows

```
MyStack.ShowTopOfStackPtr;
```

```
Program StackConsoleProj;
{$AppType Console}
Uses SysUtils,StackUnit;
Var
  MyStack : TStackType;
  i : Integer;
Begin
  MyStack := TStackType.Create;
  MyStack.ShowTopOfStackPointer;
  For i := 1 To 5
    Do MyStack.Push;
  MyStack.ShowStack;
  Repeat
    Writeln(MyStack.Pop);
    MyStack.ShowStack;
  Until MyStack.IsEmpty;
End.
```

No memory is allocated for a stack object at this stage

Table 1.2.3.2 Object Pascal program which instantiates a TStackType object

Single licence - Abingdon School

When the program is executed it behaves as shown in *Figure 1.2.3.4*.
The *For loop* calls `MyStack.Push` five times. Each time the method `Push` requests input from the user.

The method `ShowStack` is then invoked to show the state of the stack.

```
MyStack.ShowTopOfStackPointer;
  For i := 1 To 5
    Do MyStack.Push;
  MyStack.ShowStack;
  Repeat
    Writeln(MyStack.Pop);
    MyStack.ShowStack;
  Until MyStack.IsEmpty;
```

The body of the *Repeat Until loop* invokes the `Pop` method and writes the value popped from the top of stack to the console window.
The state of the stack is then shown by invoking `ShowStack`. The size of the stack decreases by one item in each iteration of the loop until eventually it is empty. The method `IsEmpty` returns `True` when the stack is empty.

```
Z:\ProductionChapters\Unit1Book\NewChaptersForVersion3\...
Top of stack pointer = 0
Input an Item: 1
Input an Item: 2
Input an Item: 3
Input an Item: 4
Input an Item: 5
------Stack------
        5
        4
        3
        2
        1
------Stack------
Press Return Key to Continue

5
------Stack------
        4
        3
        2
        1
------Stack------
Press Return Key to Continue

4
------Stack------
        3
        2
        1
------Stack------
Press Return Key to Continue

3
------Stack------
        2
        1
------Stack------
Press Return Key to Continue

2
------Stack------
        1
------Stack------
Press Return Key to Continue

1
------Stack------
------Stack------
Press Return Key to Continue
```

Figure 1.2.3.4 Output from the execution of the Object Pascal program shown in Table 1.2.3.2

Information

Class:
There are two interpretations of a class in OOP. The first views the class as an Abstract Data Type (ADT) – a mathematical concept. The second views a class as a model of some person, place, thing or idea. An example of an ADT is a stack data type that encapsulates the behaviour and state of a stack.

Questions

1. What is meant by the following in object-oriented programming
   (a) an object (b) object instantiation (c) a constructor (d) an instance of a class?
2. Explain, with an example, the meaning of the following statement
   "Objects share behaviour but not state"
3. Complete the following sentence: "When a function or procedure of an object is called we say that ....."
4. Write a class for a circular queue abstract data type with 100 slots each capable of storing an integer. You need to specify attributes, query operations and command operations

Single licence - Abingdon School

Encapsulation and information hiding

Information hiding underpins both structured design and object-oriented design. *Chapter 1.2.2* covers aspects of structured design, information hiding in the form of the black box, and the concept of modularity.

In object-oriented programming information hiding gives rise to the concepts of **encapsulation** and **modularity** as well.

The concept of information hiding first came to public attention in a paper published by David Parnas in 1972 called "On the Criteria to be Used in Decomposing Systems Into Modules."

Information hiding is characterised by the idea of "secrets": design and implementation decisions that a software developer hides in one place from the rest of the program.

One key task in designing a class is deciding which features should be known outside the class and which should remain secret. This aspect of class design is known as **visibility** since it has to do with which features of the class are visible or exposed outside the class.

The parts exposed are called the class' **interface**. The interface of a class should reveal as little as possible about the inner workings of the class.

The mechanism used in Object Pascal to hide information about a class is the *unit*. The unit is a way of enforcing **modularity**. The unit in *Table 1.2.3.3* defines the class `TStackType` in the unit's *Interface* section.

The *Implementation* section is where things are placed which are to be kept "secret".

In this example, the code which implements the methods of objects of the `TStackType` class is kept "secret", i.e. not visible to any program or other class outside the unit.

This is one aspect of encapsulation, i.e. **implementation hiding**.

```
Unit StackUnit;
Interface
  Type
    TStackType = Class
                   Private
                     StackArray : Array[1..100] Of Integer;
                     TopOfStackPtr : Integer;
                     BaseOfStackPtr : Integer;
                     HighWatermarkPtr : Integer;
                   Public
                     Function IsEmpty : Boolean;
                     Function IsFull : Boolean;
                     Function Pop : Integer;
                     Procedure Push;
                     Procedure ShowStack;
                     Constructor Create;
                     Procedure ShowTopOfStackPtr;
                 End;
Implementation
  Constructor TStackType.Create;
     Begin
       TopOfStackPtr := 0;
       BaseOfStackPtr := TopOfStackPtr + 1;
       HighWatermarkPtr := 100
     End;
  Function TStackType.IsEmpty : Boolean;
    Begin
      If TopOfStackPtr = 0
        Then Result := True
        Else Result := False;
    End;
  Function TStackType.IsFull : Boolean;
    Begin
      If TopOfStackPtr = HighwaterMarkPtr
        Then Result := True
        Else Result := False;
    End;
  Function TStackType.Pop : Integer;
    Begin
      If IsEmpty
        Then Writeln('Stack Empty')
        Else
          Begin
            Result := StackArray[TopOfStackPtr];
            TopOfStackPtr := TopOfStackPtr - 1;
          End;
    End;
  Procedure TStackType.Push;
    Begin
      If IsFull
        Then Writeln('Stack Full')
        Else
          Begin
            TopOfStackPtr := TopOfStackPtr + 1;
            Write('Input an Item: ');
            Readln(StackArray[TopOfStackPtr]);
          End;
    End;
  Procedure TStackType.ShowStack;
    Var
      j : Integer;
    Begin
      Writeln('------Stack------');
      For j := TopOfStackPtr DownTo BaseOfStackPtr
        Do Writeln(StackArray[j]:9);
      Writeln('------Stack------');
      Writeln('Press Return Key to Continue');
      Readln;
    End;
  Procedure TStackType.ShowTopOfStackPtr;
    Begin
      Writeln('Top of stack pointer = ', TopOfStackPtr);
    End;
End.
```

Table 1.2.3.3 Pascal unit which defines TStackType, its interface and its implementation

Single licence - Abingdon School

Key concept

Implementation encapsulation:
Implementation encapsulation involves hiding the implementation of a structure (object) from the rest of the application. The application is then forced to use the structure's abstraction.

Another aspect of encapsulation is **data hiding**. This is achieved by using the **Private** keyword as shown in *Table 1.2.3.4*.

```
Type
  TStackType = Class
                 Private
                   StackArray : Array[1..100] Of Integer;
                   TopOfStackPtr : Integer;
                   BaseOfStackPtr : Integer;
                   HighWatermarkPtr : Integer;
                 Public
                   Function IsEmpty : Boolean;
                   Function IsFull : Boolean;
                   Function Pop : Integer;
                   Procedure Push;
                   Procedure ShowStack;
                   Constructor Create;
                   Procedure ShowTopOfStackPtr;
               End;
```

Table 1.2.3.4 Data hiding using the Private keyword

The attributes `StackArray`, `TopOfStackPtr`, `BaseOfStackPtr` and `HighWatermarkPtr` will not be visible to any program which uses `TStackType` objects because these attributes are specified to be private.

For example, if the program shown in *Table 1.2.3.2* were to be modified as shown in *Table 1.2.3.5* it would fail to compile because of the statement

```
MyStack.TopOfStackPtr
```

The compiler would report that `TopOfStackPtr` was not recognised.

```
Program StackConsoleProj;
{$AppType Console}
Uses SysUtils,StackUnit;
Var
  MyStack : TStackType;
  i : Integer;
Begin
  MyStack := TStackType.Create;
  Writeln(MyStack.TopOfStackPtr);
  MyStack.ShowTopOfStackPointer;
  For i := 1 To 5
    Do MyStack.Push;
  MyStack.ShowStack;
  Repeat
    Writeln(MyStack.Pop);
    MyStack.ShowStack;
  Until MyStack.IsEmpty;
End.
```

Table 1.2.3.5 Object Pascal program which fails to compile because of the statement in red

The class' interface uses the keyword **Public** to make the methods `IsEmpty`, `IsFull`, `Pop`, `Push`, `ShowStack` and `ShowTopOfStackPtr` visible to programs which wish to invoke these methods on objects of this class.

Invoking an object method such as `ShowStack` can also be viewed as **sending a message to the object** as shown in *Figure 1.2.3.5*.

Figure 1.2.3.5 shows two objects of class `TStackType`, `Object1` and `Object2`. Each object is sent a message commanding each to show the state of their stack and a message to push an item onto their stack.

Key concept

Data encapsulation:
Data encapsulation means restricting access to the data of an object.

However, **encapsulation** is at work here because the internal workings of each object are concealed from the program sending these messages. The program communicates with each object via the object's public interface only.

Key concept

Information hiding:
Information hiding =
Data encapsulation +
Implementation encapsulation

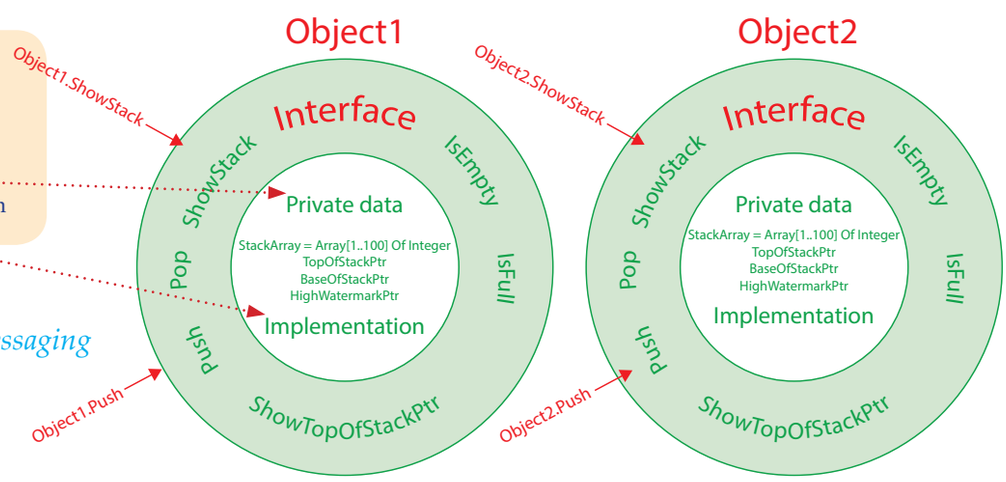


Figure 1.2.3.5 Object messaging

Single licence - Abingdon School

The **behaviour** of each object in response to these messages is similar because they belong to the same class but their **state** can differ because state depends on what has happened to each, i.e. the behaviour that each has engaged in. *Figure 1.2.3.4* shows an example of how the state of the stack can change and therefore the object `MyStack`.

Questions

5 Consider the following statement and then explain what it means.

"An object is a specific entity that exists in a program at run time whereas the class of the object is an entity to be found in a listing of the program, i.e. its source code"

6 What is meant in object-oriented programming by

(a) implementation encapsulation (b) data encapsulation?

Inheritance

What is inheritance?

In designing a software system, it is often the case that objects occur that are much like other objects except for a few differences in their type. For example, a college system which records data on students, may have to deal with both part-time and full-time students. Most of the attributes associated with both kinds of students are the same, but some attributes are different.

The solution in object-oriented programming is to define a general type of student, `TStudent`, and then define full-time students as a type, `TFullTime`, which is a general type of student except for a few differences, and part-time students as a type, `TPartime`, which is also a general type of student except for a few differences.

Operations on student data that don't depend on whether the student is full- or part-time, e.g. `ChangeAddress`, will be common to both full- and part-time students .
However, there will be some operations which apply only to full-time students only, e.g. `AssignTutor` and some operations which apply only to part-time students, `AssignMentor`.

Similarly, there are attributes which both full-time and part-time students share, e.g. `Surname`, `Forename`, `Address`, and attributes which are full-time student only, e.g. `TutorStaffID`, and attributes which are part-time student only, e.g. `MentorStaffID`.

Those **behaviours** (operations) and **characteristics** (attributes) which are common to all students can be made accessible to the classes `TFullTime` and `TPartime` by inheriting them from class `TStudent` in a mechanism called **inheritance**.

The class `TFullTime` can then **extend** the class `TStudent` with those operations and attributes which are special to full-time students.
Similarly, the class `TPartTime` can **extend** the class `TStudent` with those operations and attributes which are special to part-time students.
Inheritance is the concept that one class is a specialisation of another class, e.g. `TFullTime` **is a** specialised version of `TStudent`. The "*is a*" phrase is highlighted because it is a good test of whether a relationship between two classes is one of inheritance.

Key concept

Inheritance:
Inheritance allows a class to be derived from an existing class without the need to modify the existing class.
The derived class has all the attributes and methods of the parent class, but adds new ones of its own.
Another name for a derived class is a subclass.
Inheritance is a type of relationship between classes called an "is-a" relationship.

Single licence - Abingdon School

What is the purpose of inheritance?

The purpose of inheritance is to simplify the writing of code by defining a **base class**, e.g. `TStudent`, which specifies the elements common to two or more **derived classes**, e.g. `TFullTime` and `TPartTime`.

Inheritance helps avoid the need to repeat common operation code and attributes in multiple classes by centralising it within a base class (or in ancestor classes if one class inherits from another which itself inherits from another, etc). In practice, it results in a use of memory similar to the example shown in *Figure 1.2.3.6*[1] with two objects:

- `FullTime1` which belongs to the class `TFullTime`
- `PartTime1` which belongs to the class `TPartTime`.

The surname, forename and address attributes come from the `TStudent` class.
Each object adds its own specialised attribute(s).

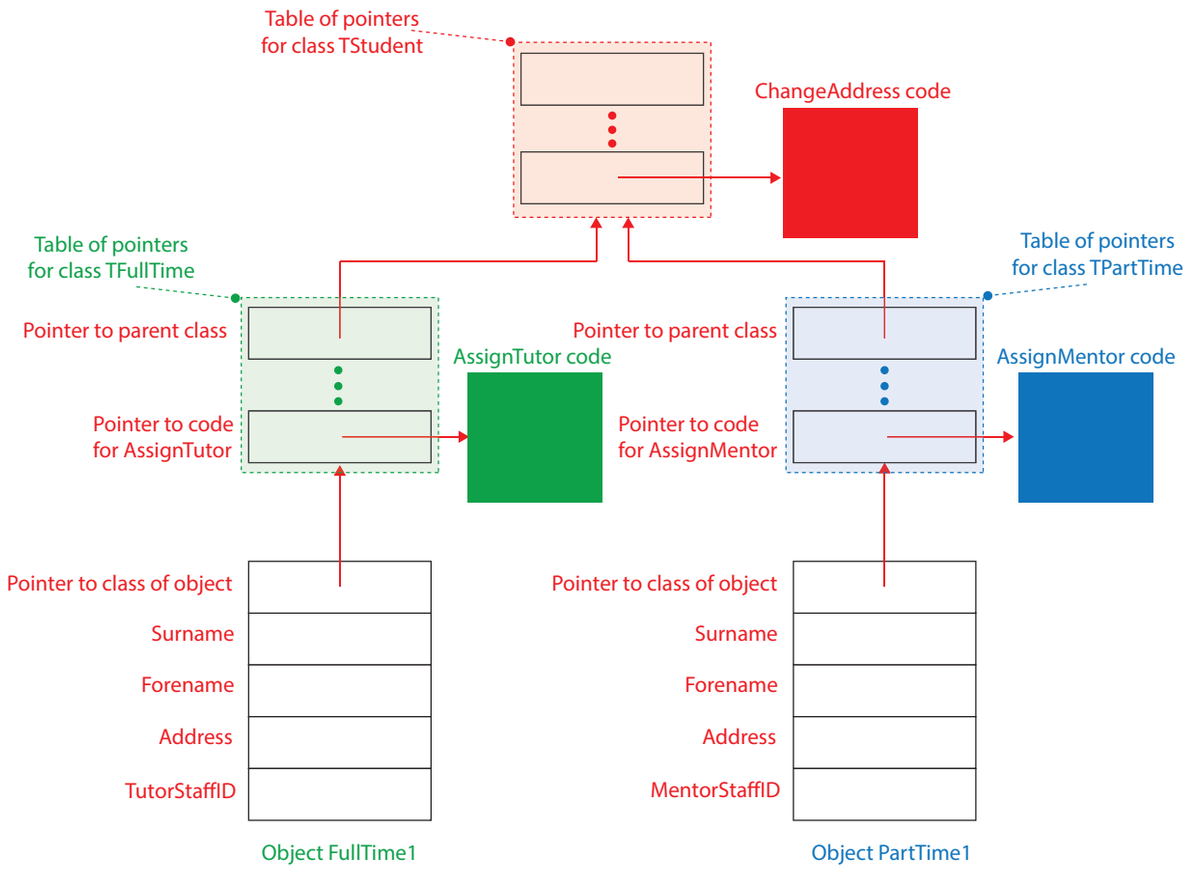


Figure 1.2.3.6 Partial memory model for two objects, FullTime1 and PartTime1

As the code is shared amongst objects, each object has a pointer field which is used to locate the code defined for its class and any code which it inherits from its parent class, `TStudent`.
A table of pointers is shown loaded into memory for each class. For object `FullTime1` to find the code for `ChangeAddress`, it uses its pointer to the pointer table for class `TFullTime` then it uses this table to find the table of pointers for the parent class `TStudent`. The class `TStudent` points to the code for the method `ChangeAddress`.

1 There are other ways it can be done.

Single licence - Abingdon School

The pseudo-code class definitions in *Table 1.2.3.6*, *Table 1.2.3.7* and *Table 1.2.3.8* show how inheritance can be indicated. Different programming languages indicate inheritance in different ways.

```
Type
  TStudent = Class
                Private
                  Surname : String
                  Forename : String
                  Address : String
                  ...
                Public
                  Procedure ChangeAddress
                  Constructor Create
                  ...
              End
```

Table 1.2.3.6 Class TStudent

```
Type
  TFullTime = Class(TStudent)
                Private
                  TutorStaffID : Integer
                  ...
                Public
                  Procedure AssignTutor
                  ...
              End
```

Table 1.2.3.7 Class TFullTime

```
Type
  TPartTime = Class(TStudent)
                Private
                  MentorStaffID : Integer
                  ...
                Public
                  Procedure AssignMentor
                  ...
              End
```

Table 1.2.3.8 Class TPartTime

In the pseudo-code, inheritance from a parent class, for example, is shown in the derived class `TFullTime` as follows

```
TPartTime = Class(TStudent)
```

The parent class `TStudent` is placed in brackets after the keyword class. A derived class may itself be derived from another derived class, and so on, creating a hierarchy of classes.

A derived class may therefore inherit from more than **one ancestor class**, i.e from a parent class, a grandparent class, and so on.

The term **base class** is reserved for the highest class in this hierarchy. The base class may then have **descendent classes** which are also known as **derived classes** or **subclasses**.

Questions

7 A tennis club has both junior and senior members. Each member has a unique membership number, a name and address recorded. Three classes have been identified:

Member

JuniorMember

SeniorMember

The classes JuniorMember and SeniorMember are related (by single inheritance) to the class Member.

(a) In object-oriented programming what is meant by

(i) class (ii) inheritance?

(b) Write a class definition for the class Member to allow programs that use objects of this class to add a new member's details, amend a member's details, show a member's details. No other form of access is to be allowed.

(c) Each junior member has a date of birth recorded. Write a class definition for the class JuniorMember to allow programs that use objects of this class to add a date of birth, amend a date of birth, show a junior member's age. No other form of access is to be allowed.

Single licence - Abingdon School

Polymorphism

Sometimes a derived class needs to do more than just extend a parent class. Sometimes it needs to override a method so that the action carried out when the method executes is the one that matches how the object behaves.

For example, if we have a crow object and a duck object we expect the sound made by a crow to be different from the sound made by a duck. One squawks and one quacks.

However, if the only *make sound* method available to both crow and duck objects is the inherited one, `MakeSound`, defined in their parent class `TBird`, the sound we get from each object will be the same, a screeching one if that is the sound the parent class method has been implemented to make.

The first solution is to add a `MakeSound` method to the `TCrow` class and a `MakeSound` method to the `TDuck` class, with each method implementing the correct sound when invoked.

For example, suppose we have a `TBird` object, `Bird1`, a `TCrow` object, `Crow1`, and a `TDuck` object, `Duck1`.

These classes are shown in *Table 1.2.3.9*, *Table 1.2.3.10* and *Table 1.2.3.11*.

```
Type
  TBird = Class
            Private
              Name : String
            Public
              Procedure MakeSound
              Constructor Create
              ...
            End
```

Table 1.2.3.9 Class TBird

```
Type
  TCrow = Class(TBird)
            Public
              Procedure MakeSound
              ...
            End
```

Table 1.2.3.10 Class TCrow

```
Type
  TDuck = Class(TBird)
            Public
              Procedure MakeSound
              ...
            End
```

Table 1.2.3.11 Class TDuck

The objects are created by first creating reference variables `Bird1`, `Crow1`, and `Duck1` as shown in *Table 1.2.3.12*.

```
Var
  Bird1 : TBird
  Crow1 : TCrow
  Duck1 : TDuck
```

Table 1.2.3.12 Creating reference variables

An instance of each class is then created and a reference assigned to the corresponding reference variable as shown in *Table 1.2.3.13*. The classes `TCrow` and `TDuck` use the inherited constructor `Create`.

```
Bird1 : TBird.Create
Crow1 : TCrow.Create
Duck1 : TDuck.Create
```

Table 1.2.3.13 Creating an instance of each class and assigning it to its corresponding reference variable

Figure 1.2.3.7 shows the correct bird sound being emitted when `MakeSound` is invoked as shown.

Suppose now we have only one reference variable `Bird1` and this is of type `TBird`. Is it possible to assign a `TCrow` object to `Bird1`?

The answer is yes because a `TCrow` object is a type of bird.

What sound is made if we invoke `Bird1.MakeSound`?
The answer is "Screech" because the implementation of `MakeSound` invoked is that belonging to type `TBird` since `Bird1` is of type `TBird` and the method link is made at compile time to the methods of `TBird`.
To get the correct sound we need to alter the class definitions as shown in *Table 1.2.3.16*, *Table 1.2.3.17* and *Table 1.2.3.18*.

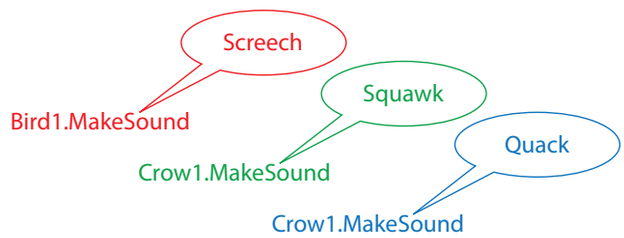


Figure 1.2.3.7 Invoking MakeSound on each object

Single licence - Abingdon School

The parent class `TBird`'s `MakeSound` method is labelled as **virtual** and the corresponding method in each derived class is labelled with **override**.
Now *Table 1.2.3.14* and *Table 1.2.3.15* create a crow object for which `Bird1.MakeSound` emits "Squawk". The bird object that `Bird1` references knows it is a crow object and that the sound to make is the crow sound (the link to method `MakeSound` is the one that corresponds to the current object).

```
Var
  Bird1 : TBird
```

Table 1.2.3.14 Creating a reference variable

```
Bird1 ← TCrow.Create
```

Table 1.2.3.15 Creating an instance of class TCrow and assigning it to reference variable of TBird

If the following code is now executed the sound emitted is "Quack"

```
Bird1.Destroy
Bird1 ← TDuck.Create
Bird1.MakeSound
```

The `TCrow` object assigned to `Bird1` is first destroyed (memory released). A `TDuck` object is then created and assigned to `Bird1`. The method `MakeSound` is then called on this object.

The correct sound, "Quack" is made because the binding of `MakeSound` is the one that is correct for a duck object.

This is **late binding**. Late binding occurs at runtime. This allows methods from the class of the created object to be chosen even though the reference variable is of a parent or ancestor class.

If we follow this with the code shown below we get the sound "Squawk" because `Bird1` now references a crow object

```
Bird1.Destroy
Bird1 ← TCrow.Create
Bird1.MakeSound
```

This is an example of **polymorphism**.
In polymorphism, a method invoked on an object referenced by a variable of a more general type results in behaviour expected of the specific object, e.g. invoking `MakeSound` via a `TBird` variable elicits "Quack" if the object assigned is a duck and"Squawk" if a crow.

```
Type
  TBird = Class
            Private
              Name : String
            Public
              Procedure MakeSound Virtual
              Constructor Create
              ...
            End
```

Table 1.2.3.16 Class TBird

```
Type
  TCrow = Class(TBird)
            Public
              Procedure MakeSound Override
              ...
            End
```

Table 1.2.3.17 Class TCrow

```
Type
  TDuck = Class(TBird)
            Public
              Procedure MakeSound Override
              ...
            End
```

Table 1.2.3.18 Class TDuck

The non-polymorphic example shown in *Table 1.2.3.14*, and *Table 1.2.3.15* uses early binding, i.e. objects are linked to code for operations at compile time. This means that even though the object created at run time is a crow, the method call to `MakeSound` is set up at compile time to link to `MakeSound` method code for `TBird` because `Bird1` is declared as belonging to the `TBird` class.

Key concept

Polymorphism:
Giving an action one name, e.g. MakeNoise, that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.

Late binding delays this linking until run time so that the relevant linkage can be made. For this to happen, the methods of the base class which need to be polymorphic must be declared as **virtual**. The corresponding methods in subclasses of the base/parent/ancestor class must be labelled with the keyword **override**, or its equivalent depending on programming language, for these to override their equivalent parent/ancestor class methods in a polymorphic way.

Single licence - Abingdon School

Tables 1.2.3.19,20,21, 22 show how this can be done in Object Pascal.

```
Program BirdExample;
{$APPTYPE CONSOLE}
Uses
  System.SysUtils, BirdClass,
  DuckClass, CrowClass;
Var
  Bird1 : TBird;
Begin
  Bird1 := TCrow.Create('Crow');
  Writeln(Bird1.Name);
  Bird1.MakeSound;
  Bird1.Destroy;
  Bird1 := TDuck.Create('Duck');
  Writeln(Bird1.Name);
  Bird1.MakeSound;
  Bird1.Destroy;
  Bird1 := TBird.Create('Bird');
  Writeln(Bird1.Name);
  Bird1.MakeSound;
  Readln;
End.
```

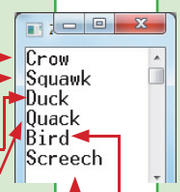


Table 1.2.3.19 Object Pascal BirdExample program

```
Unit BirdClass;
Interface
  Type
    TBird = Class
      Public
        Name : String;
      Public
        Procedure MakeSound; Virtual;
        Constructor Create(BirdName : String);
      End;
Implementation
  Procedure TBird.MakeSound;
    Begin
      Writeln('Screech');
    End;
  Constructor TBird.Create(BirdName : String);
    Begin
      Name := BirdName;
    End;
End.
```

Base class

Virtual methods can be overridden at run time

Create inherited by subclasses

Table 1.2.3.20 Object Pascal Bird class unit

```
Unit DuckClass;
Interface
  Uses BirdClass;
  Type
    TDuck = Class (TBird)
      Public
        Procedure MakeSound; Override;
    End;
Implementation
  Procedure TDuck.MakeSound;
    Begin
      Writeln('Quack');
    End;
End.
```

Inheritance mechanism TBird is parent class of TDuck

Derived class or subclass

Table 1.2.3.21 Object Pascal Duck class unit

```
Unit CrowClass;
Interface
  Uses BirdClass;
  Type
    TCrow = Class(TBird)
      Public
        Procedure MakeSound; Override;
    End;
Implementation
  Procedure TCrow.MakeSound;
    Begin
      Writeln('Squawk');
    End;
End.
```

Inheritance mechanism TBird is parent class of TDuck

Derived class or subclass

Table 1.2.3.22 Object Pascal Crow class unit

Tables 1.2.3.23,24,25, 26 show how this can be done in C#.

```
using System;
namespace BirdExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Bird bird1;
            bird1 = new Crow("Crow");
            bird1.MakeSound();
            Console.WriteLine(bird1.Name);
            bird1 = new Duck("Duck");
            bird1.MakeSound();
            Console.WriteLine(bird1.Name);
            bird1 = new Bird("Bird");
            bird1.MakeSound();
            Console.WriteLine(bird1.Name);
            Console.ReadLine();
        }
    }
}
```

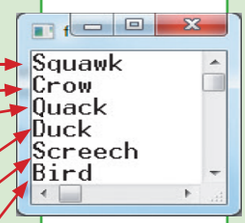


Table 1.2.3.23 C# BirdExample program

```
using System;
namespace BirdExample
{
    class Bird
    {
        public string Name;
        public Bird(string name)
        {
            Name = name;
        }
        virtual public void MakeSound()
        {
            Console.WriteLine("Screech");
        }
    }
}
```

Base class

Virtual method can be overridden at run time

Table 1.2.3.24 C# Bird class file

In C# an object is created with the language keyword `new` followed by the name of the class constructor. The name of this constructor is the same as the class name, e.g. `Bird`. The object created is located somewhere in memory. The reference assigned to the reference variable, `bird1`, is the mechanism used to access this object at its location in memory.

Single licence - Abingdon School

```
using System;
namespace BirdExample
{
    class Duck : Bird
    {
        public Duck(string name) : base(name)
        {
            this.Name = name;
        }

        override public void MakeSound()
        {
            Console.WriteLine("Quack");
        }
    }
}
```

Inheritance

Use *name* from base class

Derived class or subclass

Table 1.2.3.25 C# Duck class file

```
using System;
namespace BirdExample
{
    class Crow : Bird
    {
        public Crow(string name) : base(name)
        {
            this.Name = name;
        }

        override public void MakeSound()
        {
            Console.WriteLine("Squawk");
        }
    }
}
```

Inheritance

Derived class or subclass

Table 1.2.3.26 C# Crow class file

Tables 1.2.3.27,28,29, 30 show how this can be done in VB.NET.

```
Module Module1
    Sub Main()
        Dim bird As Bird = New Crow("Crow")
        bird.MakeSound()
        Console.WriteLine(bird.NameOfBird)
        bird = New Duck("Duck")
        bird.MakeSound()
        Console.WriteLine(bird.NameOfBird)
        bird = New Bird("Bird")
        bird.MakeSound()
        Console.WriteLine(bird.NameOfBird)
        Console.ReadLine()
    End Sub
End Module
```

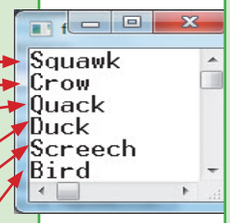


Table 1.2.3.27 VB.NET BirdExample program

```
Public Class Bird
    Public Property NameOfBird As String
    Public Sub New(ByVal name As String)
        NameOfBird = name
    End Sub
    Public Overridable Sub MakeSound()
        Console.WriteLine("Screech")
    End Sub
End Class
```

Base class

Virtual method

Keyword Overridable is equivalent to the keyword Virtual in other OOP languages

Table 1.2.3.28 VB.NET Bird class file

Derived class or subclass

Inheritance

```
Public Class Duck : Inherits Bird
    Public Sub New(ByVal name As String)
        MyBase.New(name)
        NameOfBird = name
    End Sub
    Public Overrides Sub MakeSound()
        Console.WriteLine("Quack")
    End Sub
End Class
```

Table 1.2.3.29 VB.NET Duck class file

Derived class or subclass

Inheritance

```
Public Class Crow : Inherits Bird
    Public Sub New(ByVal name As String)
        MyBase.New(name)
        NameOfBird = name
    End Sub
    Public Overrides Sub MakeSound()
        Console.WriteLine("Squawk")
    End Sub
End Class
```

Table 1.2.3.30 VB.NET Crow class file

VB.NET like C# uses the language keyword `new` followed by the name of the class constructor to create an object The name of this constructor is the same as the class name, e.g. `Bird`.

VB.NET uses the keyword `Overridable` to make a method virtual - *Figure 1.2.3.28*. It uses the keyword `Overrides` applied to a method that will ovveride a virtual method in the parent class.

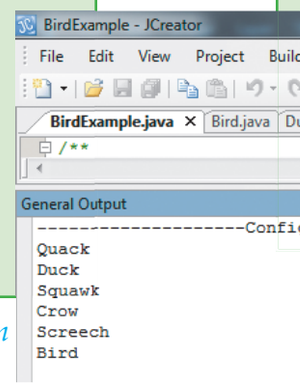
In Java, variables declared to be of a type which defines a class (another name for a class is *object type*) are polymorphic variables, e.g. *Table 1.2.3.31* shows a declaration of a variable `bird1` of type `Bird`, a class defined in *Table 1.2.3.32*.

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behaviour is referred to as virtual method invocation. *Table 1.2.3.33* and *Table 1.2.3.34* show derived classes `Duck` and `Crow` which inherit the attribute `Name` from the class `Bird` and the constructor `Bird` which is referenced as `super` in the `Duck` and `Crow` subclasses. It is the constructor which grabs memory for the attribute `Name`.

Single licence - Abingdon School

```
public class BirdExample {
    public static void main(String[] args) {
        Bird bird1;
        bird1 = new Duck("Duck");
        bird1.MakeSound();
        System.out.println(bird1.Name);
        bird1 = new Crow("Crow");
        bird1.MakeSound();
        System.out.println(bird1.Name);
        bird1 = new Bird("Bird");
        bird1.MakeSound();
        System.out.println(bird1.Name);
    }
}
```

Table 1.2.3.31 Java BirdExample program



Base class

```
public class Bird {
    public String Name;
    public Bird(String name)
    {
        Name = name;
    }
    public void MakeSound()
    {
        System.out.println("Screech");
    }
}
```

Table 1.2.3.32 Java Bird class file

Derived class or subclass — Inheritance

```
public class Duck extends Bird{
    public Duck(String name)
    {
        super(name);// use constructor of superclass
    }
    public void MakeSound()
    {
        System.out.println("Quack");
    }
}
```

Table 1.2.3.33 Java Duck class file

Derived class or subclass — Inheritance

```
public class Crow extends Bird{
    public Crow(String name)
    {
        super(name);
    }
    public void MakeSound()
    {
        System.out.println("Squawk");
    }
}
```

Table 1.2.3.34 Java Crow class file

Tables 1.2.3.35,36,37, 38 show how this can be done in Python 3.4. Variables are polymorphic in Python so there is no need to make use of keywords virtual and override. Python automatically chooses the appropriate `MakeSound` method.

```
from BirdClass import Bird
from DuckClass import Duck
from CrowClass import Crow

bird1 = Duck("Duck")
print(bird1.Name)
bird1.MakeSound()
bird1 = Crow("Crow")
print(bird1.Name)
bird1.MakeSound()
bird1 = Bird("Bird")
print(bird1.Name)
bird1.MakeSound()
print(bird1)
```

```
Python 3.4.1 |Anaconda 2
4 2014, 18:34:57) [MSC v
Type "help", "copyright"
ore information.
runfile('Z:/ProductionCl
rVersion3/Python/BirdExa
py', wdir=r'Z:/Productic
sForVersion3/Python/Birc
>>> Duck
Quack
Crow
Squawk
Bird
Screech
Bird
>>>
```

__repr__(self) turns object attribute value(s) into a string which is returned when object referenced, e.g. bird1

Table 1.2.3.35 Python BirdExample program

Base class — Constructor — Return nothing — *self* references the current object

```
class Bird:
    def __init__(self, nameOfBird) -> None:
        self.Name = nameOfBird

    def MakeSound(self) -> None:
        print("Screech")
    def __repr__(self) -> str:
        return str(self.Name)
```

Table 1.2.3.36 Python Bird class file BirdClass.py

Class — Inheritance — Derived class or subclass

```
from BirdClass import Bird
class Duck(Bird):

    def MakeSound(self) -> None:
        print("Quack")
```

Table 1.2.3.37 Python Duck class file DuckClass.py

File BirdClass.py — Inheritance

```
from BirdClass import Bird
class Crow(Bird):
    def MakeSound(self) -> None:
        print("Squawk")
```

Table 1.2.3.38 Python Crow class file CrowClass.py

Information

The Pascal/Delphi, C#, VB.NET, Java and Python Bird example programs are available to download from www.educational-computing.co.uk/aqacs/alevelcs.html.

All these languages except Python enforce encapsulation.

Single licence - Abingdon School

Programming task

1 Write a program that stores a student's examination score out of 60, the percentage that this score represents, the grade awarded for this score ('A' >= 50, 'B' < 50) and the student's name. The program will show/display the student's name, examination score, percentage and grade.

The methods to be provided are

An object constructor which initialises the object with student name and examination score
CalculatePercentage
CalculateGrade
ShowStudentData

Use a class with 4 attributes – Name, Score, Percentage, Grade - and the 4 methods listed above.

2 A bank account has associated with it the name and address of the customer and the current balance. To be able to use this as a data object, procedures and functions (known as methods) must be provided to initialise an account, change the name of the customer, get the current balance, credit an amount and print the details of the account.

Suppose the bank were to introduce a new class of account for special customers who deposited an initial balance of over £500. This account is to be known as a Gold Account and is to pay interest to customers on their daily balance. The Gold account, as well as having all the characteristics and behaviours of an ordinary account, Account, needs to store the following extra data: interest rate and date interest added, and two extra functions/procedures: AddInterest to account, WithdrawMoney from account. Object-oriented languages allow the programmer to create an object for the Gold Account which inherits all the characteristics and behaviours of the ordinary account, but also has additional characteristics and behaviours of its own. Write a program to implement the banking system.

Questions

8 A company has two types of employees, salaried employee and contract employee. Contract employees are paid an amount each month which is determined by the number of hours worked each month.

A general class Employee contains the following attributes

PayrollNo
Surname
Forename
Address
MonthlyPaymentRecordsForCurrentYear

Employee has four methods AddEmployeeDetails, AmendEmployeeDetails, CalculateMonthlyPay, PrintPaySlip.

A class ContractEmployee exists which inherits from Employee.

ContractEmployee has an additional attribute, HourlyRate and methods AddEmployeeDetails, AmendEmployeeDetails, CalculateMonthlyPay, PrintPaySlip.

When the payroll program runs it creates an object of the appropriate type, i.e. SalariedEmployee or ContractEmployee, for each employee and assigns a reference to a reference variable of type Employee.

(a) Explain why if the reference variable is not polymorphic, CalculateMonthlyPay will need to be polymorphic.

(b) Using the keywords Virtual and Override where appropriate, write class definitions for the ContractEmployee class. You may omit the data type of each attribute.

Single licence - Abingdon School

Questions

9 Consider the class Person. List some attributes that a person could have in each of the following systems
(a) library system (b) healthcare system

10 Identify some of the classes that you would expect to find in each of the following systems
(a) library system (b) airline booking system (c) system which manages hotel bookings

In this chapter you have covered:

- The concept:
  - class - a template listing characteristics and defining behaviours (functions and procedures) from which actual objects may be created. An object is characterized by a state and an interface specifying a collection of state-changing operations. Object-oriented models of computations are expressed in terms of a collection of objects which exchange messages by using interface operations.
  - instantiation is the process of creating an instance of a class, an object. It creates memory for its data (object state) and links to the method code that implement operations that may be performed on the object.
  - encapsulation is a mechanism to conceal the internal workings of an object from the program which interacts with it. The program is forced to use the public interface of the object.
  - inheritance allows a class to be derived from an existing class without the need to modify the existing class. The derived class has all the attributes and methods of the parent class, but adds new ones of its own.
  - polymorphism is when an action is given one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself. The action takes the form of a polymorphic method which when invoked on an object referenced by a variable of a more general type results in behaviour expected of the specific object.
  - overriding is a part of the process which enables polymorphism to take place. Methods of the base class which need to be polymorphic are declared as **virtual** if the language does not support polymorphic variables or equivalent. The corresponding methods in subclasses of the base/parent/ancestor class are labelled with the keyword **override** in such languages for these to override their equivalent parent/ancestor class methods in a polymorphic way.

Single licence - Abingdon School

1 Fundamentals of programming

1.2 Programming paradigms

Learning objectives:

- *Be familiar with the concept:*
  - *aggregation*
  - *composition*

1.2.3(2) Object-oriented programming

Aggregation

Aggregation is not a concept unique to object-oriented programming languages. Any language that supports record-like structures supports aggregation.

Aggregation is a special form of association with the meaning of a whole/part hierarchy together with the ability to navigate from the whole or aggregate to its parts.

For example, a book has pages. We can think of a book as the whole and the pages as the parts.

A book identified with the ID 36 could be made up of 500 pages with each page uniquely numbered.

The book's title might be '*Intro to OOP by L. P. Partley*'.

The pages are bound together and a cover attached to make the book.

In a simplified scenario, we might define a class `TBook` for all book objects in pseudo-code as shown in *Table 1.2.3.39*.

Methods have been omitted from `TBook` for convenience.

Similarly, we might define a class `TPage` for all the page objects which make up a book as shown in *Table 1.2.3.40*.

Key concept

Aggregation:
Aggregation is a special form of association between objects with the meaning of a whole/part hierarchy together with the ability to navigate from the whole or aggregate to its parts. If and only if there exists a whole/part relationship between two objects, e.g. book and page, can an aggregate relationship exist between their corresponding classes, TBookClass and TPageClass.

```
TBook = Class
   BookID : 1..2000
   Title : String
   PageRef : Array[1..500] Of TPage
 End
```

Table 1.2.3.39 TBook class definition in pseudocode

```
TPage = Class
          PageNo : Integer
          PageContent : String
        End
```

Table 1.2.3.40 TPage class definition in pseudocode

```
aBook ← TBook.Create
aBook.BookID ← 36
For i ←1 To 500
  aBook.PageRef[i] ← TPage.Create
  aBook.PageRef[i].PageNo ← i
EndFor
```

Table 1.2.3.41 Creating book and page objects

Methods have been omitted from `TPage` for convenience.

We use an object constructor method `Create` to create an instance of the `TBook` class and assign the reference to a variable `aBook` of type `TBook`.

We can then create 500 page objects and assign their references as shown in pseudo-code in *Table 1.2.3.41*.

We navigate from the whole to the part as follows

```
Output aBook.PageRef[2].PageNo
```

This pseudo-code output statement accesses `aBook`, the *whole*, then an attribute of `aBook`, called `PageRef`. This attribute is an array which can contain up to 500 references to `TPage` objects. The indexed reference `PageRef[2]` references the second page object, a *part* of the *whole*. This has an attribute `PageNo`. It is this which is displayed by `Output`.

Single licence - Abingdon School

The book object may be destroyed, i.e. the memory occupied by the object released, by using a **destructor**, `Destroy`, as shown in the pseudo-code in *Table 1.2.3.42.*

```
For i ← 1 To 500
  aBook.PageRef[i].Destroy
EndFor
aBook.Destroy
```

Table 1.2.3.42 Releasing the memory occupied by the book and page objects

The pseudo-code first destroys all the page objects before finally destroying the book object. The destructor `Destroy` frees the memory of the object to which it is applied.

If the book object was a tree in a forest then its leaves are destroyed first followed by the tree itself.

The detail shown in *Table 1.2.3.42* can be hidden within the destructor so that calling `aBook.Destroy` destroys everything. By analogy, the whole tree is destroyed, leaves, trunk, branches, etc, by the action of destruction.

Questions

1. Assuming that a book object has been created with 500 pages, each numbered consecutively starting from 1, what is displayed by `Output` if the following pseudocode could be executed?

```
Output aBook.PageRef[36].PageNo
```

Composition

Suppose we changed the `TBook` class so that its attributes are private and control access to these attributes via methods such as `TurnPage`, `ShowPage`. Our class definition might now look like as shown in *Table 1.2.3.43*.

```
TBook = Class
          Private
            BookID : 1..2000
            Title : String
            PageRef : Array[1..500] Of TPage
          Public
            Procedure TurnPage
            Procedure ShowPage
            ...
        End
```

Table 1.2.3.43 Applying data encapsulation to TBook

All page objects are now hidden behind access methods. The consequence is that each page is **associated** with just one book object because there is no easy way to copy a page reference so the object it references can be *associated* at the same time with another book object. We call this restricted form of aggregation by a special name **composition**.

Key concept

Composition:
A whole/part relationship or association in which the parts of the whole are not shareable.

It is characterised by the following

1. It is a whole/part relationship or association, i.e. an aggregation
2. The parts of the whole are not shareable

It is not a defining characteristic of composition to say that the runtime lifetime of the part is the same as the whole, i.e. destroy the whole and the parts are destroyed too; it is usually the case.

The only necessary characteristic is that a part object can only be part of **one relationship** or **association**.

Composition:

Whole/part association in which the parts of a whole cannot be shared with any other whole

Information

The example programs in this chapter are available to download from www.educational-computing.co.uk/aqacs/alevelcs.html.

Single licence - Abingdon School

Aggregation

Suppose we use two book variables:

`aBook` and `AnotherBook` of type `TBook`

where `TBook` is the class defined in *Table 1.2.3.39* and a variable

`Pages : Array [1..500] Of TPage`

to hold references to 500 page objects.

To create two books of 500 pages each, we could reuse the 500 pages of the first for the second book shown in *Table 1.2.3.44*.

The page objects, i.e. the parts of the whole, are now shared by two books, i.e. two "wholes", `aBook` and `AnotherBook`.

This is not composition because composition rules out sharing of a part object. This example is a different kind of aggregation in which a part can be associated with more than one whole.

```
aBook ← TBook.Create
AnotherBook ← TBook.Create
For i ← 1 To 500
  Pages[i] ← TPage.Create
  Pages[i].PageNo ← i
  aBook.PageRef[i] ← Pages[i]
  AnotherBook.PageRef[501-i] ← Pages[i]
EndFor
```

Table 1.2.3.44 Sharing the part with two "wholes"

This is aggregation with shareable parts. It is known simply as **aggregation** (or **association aggregation** or **non-composition aggregation**).

However, the concept of an aggregation with shareable parts doesn't mean much in practice so it is often just called an association.

A simple test can be applied to discover if the association or relationship between two objects is aggregation. It applies to both composition and (non-composition) aggregation.

The test is called the "*has-a*" test, e.g. **a book** *has a* **number of pages** indicates that the relationship is aggregation.

Key concept

Aggregation:
It is a whole/part relationship or association in which a part can be associated with more than one whole. This is aggregation with shareable parts.

For discussion

Association

Suppose we want to link a current Ofsted report with the school to which it applies. We could define two classes as follows for a simplified scenario (no methods are shown)

```
TSchool = Class
            SchoolID : 1..5000
            SchoolName : String
          End
TOfstedReport = Class
                  OfstedReportID : 1..5000
                  Report : TBook;
                  School : TSchool
                End
```

Using two variables, `aSchool` of type `TSchool` and `anOfstedReport` of type `TOfstedReport`, we could create two objects as follows

```
aSchool ← TSchool.Create
aSchool.SchoolID ← 1
aSchool.SchoolName ← 'AGS'
anOfstedReport ← TOfstedReport.Create
anOfstedReport.school ← aSchool
anOfstedReport.OfstedReportID ← 1
anOfstedReport.Report ← "......"
```

Is this correct?

In general, a school could exist which has not received a visit from the Ofsted inspectors and an Ofsted report could exist after a school has closed and its corresponding object deleted.

What kind of association/relationship exists between the school object and the Ofsted report object? Is it a whole/part relationship? Can a school also be linked to another entity?

Single licence - Abingdon School

Figure 1.2.3.8 shows a Windows form application which adds two integers and displays the result.

The form object `FormAdd` is of type `TFormAdd`. `TFormAdd` is a class with seven attributes and one method which performs the addition. *Table 1.2.3.45* shows the Delphi unit which contains the class definition `TFormAdd`, and *Table 1.2.3.46* shows the program which uses the class `TFormAdd` to create a form object which is referenced by the reference variable `FormAdd`. This form object **has a** `TEdit` attribute called `EditFirst`. `EditFirst` is an object of type `TEdit`. `TEdit` is a class supplied by the Delphi programming language.

The "*has-a*" test tells us that the relationship or association is aggregation, and since the `EditFirst` object cannot be contained within another form at the same time, this is composition aggregation. The same test is passed by the other objects, `AddBtn` and so on.

```
Unit AddUnit;
Interface
Uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls;

Type
  TFormAdd = Class(TForm)
               AddBtn: TButton;
               EditFirst: TEdit;
               LabelEnterFirst: TLabel;
               EditSecond: TEdit;
               LabelEnterSecond: TLabel;
               Result: TEdit;
               LabelResult: TLabel;
               Procedure AddBtnClick(Sender: TObject);
             End;
Var
  FormAdd: TFormAdd;
Implementation
{$R *.dfm}
  Procedure TFormAdd.AddBtnClick(Sender: TObject);
  Begin
    Result.Text := IntToStr(StrToInt(EditFirst.Text) + StrToInt(EditSecond.Text));
  End;
End.
```

Table 1.2.3.45 Delphi unit for a windows form containing seven objects

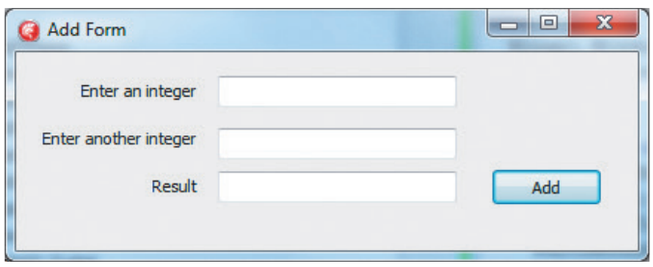


Figure 1.2.3.8 Windows form object which contains other objects

```
Program AddProject;
Uses
  Vcl.Forms, AddUnit in 'AddUnit.pas' {FormAdd};
{$R *.res}
Begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TFormAdd, FormAdd);
  Application.Run;
End.
```

Table 1.2.3.46 Delphi program which creates form

Single licence - Abingdon School

Further information

There are relationships or associations which are not aggregation or inheritance. These associations denote semantic dependency among otherwise unrelated classes, such as between "snakes" and "long grass" – snakes hide in long grass. These are just referred to as associations. For example, the type of association called *dependency* is a "uses" relationship. This is used to specify the classes "used" by the contents of the class concerned. This is usually taken to mean that the object class acts upon another class, or needs information contained within another class.
For example, a "snake" can move on the"ground" and therefore can be said to have a dependence relationship with "ground". However, since a "snake" is not "ground", it is not an inheritance relationship, and a "snake"does not contain "ground" (i.e. a snake has a ground does not make sense), a "snake' does not have an aggregation relationship with "ground".

Questions

2 A bicycle consists of a frame, wheels, tyres, a gear train, pedals, saddle, handlebars, brakes and brake levers, and a gear lever.
What is the relationship between a bicycle object and its parts?

3 A game of chess uses a chess board and chess pieces. A chess board consists of black and white squares.
What is the relationship between
(a) Chess board and its black and white squares? (b) A game of chess and a chess board used to play the game?

4 A digital watch is a kind of clock and an analogue watch is also a kind of clock. An analogue watch consists of a clock face, clock hands, a clock mechanism and a strap. A person wears an analogue watch on their wrist.
What is the relationship between
(a) Person and analogue watch? (b) Clock and digital watch? (c) Clock and analogue watch?
(d) An analogue watch and clock face, clock hands, clock mechanism and strap?

5 A piece of software enables an architect to draw architectural plans for the design of buildings.
Rooms within buildings share walls, they are called partition walls.
Is the relationship between the walls of a room and a room that they define composition or aggregation ?

5 An academic course has a name and is taught by a teacher assigned to teach the course.
A teacher has a name, a teacher ID and a subject specialism.
What is the relationship between course and teacher?

In this chapter you have covered:

- The concept of
  - aggregation in general, i.e. non-composition aggregation and composition - a whole/part hierarchical relationship or association together with the ability to navigate from the whole or aggregate to its parts. Aggregation can be discovered with the "has-a" test
  - aggregation - a whole/part relationship or association in which a part can be associated with more than one whole. This is aggregation with shareable parts.
  - composition - a whole/part relationship or association in which the parts of the whole are not shareable.

Single licence - Abingdon School

1.2 Programming paradigms

Learning objectives:

- *Know why the object-oriented paradigm is used*

1.2.3(3) Object-oriented programming

Why is the object-oriented paradigm used?

The claim is that object-oriented programming is better at modelling the real world than structured programming.

The central concept of object-oriented programming is the object, which is a kind of module containing data and subroutines. An object is a kind of self-sufficient entity that has an internal state (the data it contains) and that can respond to messages (calls to its subroutines).

Key concept

Module:
A module is a component of a larger system that interacts with the rest of the system in a simple and well-defined manner.

This modular approach means that

- **Data is protected**, since it can be manipulated only in known, well-defined ways
- **It is easier to write programs** to use a module because the details of how the data is represented and stored do not need to be known
- The **storage structure of the data and the code** for the subroutines in a module **may be altered without affecting programs** that make use of the module as long as the published interfaces and the module's functionality remain the same.

Key term

Reusability:
Reusability is the ability of software elements to be reused in many different applications.

The popularity of OOP stems from the support it gives to a software development process that relies upon preexisting reusable software components.

The spirit of reusability requires that, amongst programmers, a culture prevails in which software is developed under the assumption that it will be reused.

Code reusability

A main benefit/advantage of object-oriented programming is code reusability.

> **Reusability is the ability of software elements to be reused in many different applications.**

Rather than starting from scratch with each new application, a programmer will consult libraries of existing components to see if any are appropriate as starting points for the design of a new application.

These components will exist in libraries as class definitions. A programmer will select an appropriate class definition from a library and then create a subclass for the application. The subclass will inherit the methods and properties of the library class, add some new ones of its own and possibly redefine the actions of others.

In visual programming languages such as C#, Delphi, Visual C++ and Visual Basic the libraries store the class definitions for components which allow Graphical User Interfaces (GUI) to be built.

Single licence - Abingdon School

Reliability

Components built by specialists in their field are more likely to be designed correctly and reliably. The reuse of these components across many applications will have given the developers of the components ample feedback to deal with any bugs and deficiencies. For example, widgets for graphical user interfaces.

Efficiency (time and space)

The component developers are likely to be experts in their field and will have used the best possible algorithms and data structures.

Time Saving

By relying upon existing components there is less software to develop and hence applications can be built quicker.

The user interface for Skype® was built in Delphi quickly using Borland's Rapid Application Development® (RAD) system. The Skype developers made use of the many pre-built components available in the RAD system to save development time.

Decreased maintenance effort

Using someone else's components decreases the amount of maintenance effort that the application developer needs to expend. The maintenance of the reused components is normally the responsibility of the component supplier.

Consistency

Relying on a library of standard components will tend to spread a consistency of design message throughout a team of programmers working on an application. The library is the basis of a standard that will lend coherency and conformity to the design process, e.g. by using visual components from the Microsoft® Foundation Class library.

Investment

Reusing software will save the cost of developing similar software from scratch. The investment in the original development is preserved if the developed software can be used in another project. The most reusable software tends to be that produced by the best developers. Reusing software is thus a way of preserving the knowledge and creations of the best developers.

Reduced error checking code

The nature of a module/object is that access to its state and internal code is highly controlled. The code surrounding an object may only interact with the object in carefully controlled ways through the object's interface. This reduces the number of possible error situations that may arise and therefore the need for code to check for errors.

Questions

1. State and justify **four** reasons why the object-oriented paradigm is used.

In this chapter you have covered:

- Why the object-oriented paradigm is used

Single licence - Abingdon School

1 Fundamentals of programming

1.2 Programming paradigms

Learning objectives:

- *Be aware of the following object-oriented design principles:*
  - *encapsulate what varies*
  - *favour composition over inheritance*
  - *program to interfaces, not implementation*

1.2.3(4) Object-oriented programming

Object-oriented design principles

Encapsulate what varies

A key design principle is to

separate the parts that may be subject to change from the parts that will stay the same

For example, an encryption algorithm which is used by a part of the system may need to be replaced by a better one.

```
Unit Person;
Interface
  Uses DietInterface;
  Type
    TPerson = Class
                Private
                  Diet : IDiet;
                  PersonsName : String;
                Public
                  Constructor Create(Name : String; Kind : Integer);
                  Procedure ShowDiet;
              End;
Implementation
  Uses MeatDiet, VegetarianDiet, DontKnowDiet;
  Constructor TPerson.Create(Name : string; Kind : Integer);
    Begin
      PersonsName := Name;
      Case Kind Of
        0 : Diet := TMeatDiet.Create;
        1 : Diet := TVegetarianDiet.Create;
        Else Diet := TUnknownDiet.Create;
      End;
    End;
  Procedure TPerson.ShowDiet;
    Begin
      Writeln(PersonsName, ' is a ', Diet.GetDiet);
    End;
End.
```

Table 1.2.3.46 Encapsulation in class TPerson

Following this principle will allow the parts of the system that are subject to change to vary independently of all other parts.

The class is the way that we encapsulate.

So we need to create separate classes for the parts that may be subject to change because of new requirements or because they are required to exhibit different behaviour.

Suppose we need a software system to model people's diets.

We can declare a class `Person` as shown in *Table 1.2.3.46.*

We need to record a person's name, and what kind of eater they are, e.g. meat eater, vegetarian. However, we may be required later to cater for other categories of eater as yet not specified.

The class has been created with a private attribute `Diet` with type `IDiet`.

Type `IDiet` is an **interface type** which declares a method `GetDiet` without specifying how it will be implemented.

Table 1.2.3.47 shows the unit `DietInterface` which contains the interface declaration, `IDiet`.

```
Unit DietInterface;
Interface
  Type
    IDiet = Interface
              Function GetDiet : String;
            End;
Implementation
End.
```

Table 1.2.3.47 Encapsulating diet in an interface

A diet is the part of the application that varies so we have separated this part from the rest of the application and encapsulated this part in an *Interface unit*, `DietInterface`.

Single licence - Abingdon School

We can now implement this interface as a vegetarian diet, a meat diet and an unknown diet as shown in *Tables 1.2.3.48*, *1.2.3.49* and *1.2.3.50*.

If we require more implementations then we just provide more implementations of `IDiet`, e.g. a vegan diet.

```
Unit VegetarianDiet;
Interface
  Uses DietInterface;
  Type
    TVegetarianDiet = Class (TInterfacedObject, IDiet)
                        Function GetDiet : String;
                      End;
Implementation
  Function TVegetarianDiet.GetDiet : String;
    Begin
      Result := 'Pulses eater';
    End;
End.
```

Table 1.2.3.48 Encapsulating class TVegetarianDiet

```
Unit MeatDiet;
Interface
  Uses DietInterface;
  Type
    TMeatDiet = Class (TInterfacedObject, IDiet)
                  Function GetDiet : String;
                End;
Implementation
  Function TMeatDiet.GetDiet : String;
    Begin
      Result := 'Beef eater';
    End;
End.
```

Table 1.2.3.49 Encapsulating class TMeatDiet

```
Unit DontKnowDiet;
Interface
  Uses DietInterface;
  Type
    TUnknownDiet = Class (TInterfacedObject, IDiet)
                     Function GetDiet : String;
                   End;
Implementation
  Function TUnknownDiet.GetDiet : String;
    Begin
      Result := 'Do not know';
    End;
End.
```

Table 1.2.3.50 Encapsulating class TUnknownDiet

Table 1.2.3.51 shows an application which uses an array of `TPerson` to store four person objects.

Each object contains a diet object whose type may be a meat diet, a vegetarian diet or an unknown diet.

The design allows for new diet types to be added as needed because "what varies has been encapsulated" in class `TPerson` in a private attribute `Diet` of type `IDiet`.

Figure 1.2.3.8 shows the output when the application is run.

```
Program PersonsDietExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils, Person;
Type
  TPersonArray = Array[1..4] Of TPerson;
Var
  PersonArray : TPersonArray;
Begin
  PersonArray[1] := TPerson.Create('Fred', 0);
  PersonArray[1].ShowDiet;
  PersonArray[2] := TPerson.Create('Mary', 1);
  PersonArray[2].ShowDiet;
  PersonArray[3] := TPerson.Create('Aslam', 0);
  PersonArray[3].ShowDiet;
  PersonArray[4] := TPerson.Create('Isla', 20);
  PersonArray[4].ShowDiet;
  Readln;
End.
```

Table 1.2.3.51 Program that uses class TPerson

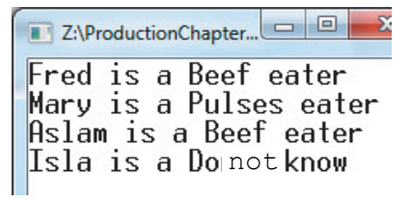


Figure 1.2.3.8 Output from running program PersonsDietExample

Key point

Difference between a class and an interface:

A class describes the attributes and behaviours of an object whereas an interface contains behaviours that a class implements.

An interface cannot be instantiated.

An interface does not contain any constructors.

All the methods in an interface are abstract.

```
TPerson = Class
                Private
                  Diet : IDiet;
                  PersonsName : String;
                Public
                  Constructor Create(Name : String; Kind : Integer);
                  Procedure ShowDiet;
              End;
```

Composition

Table 1.2.3.52 TPerson class uses composition

An important point to make is that the class `TPerson` uses **composition**. A person object of type `TPerson` *has a* diet object of type `IDiet` - *Table 1.2.3.52*.

Single licence - Abingdon School

Could the solution have been achieved using inheritance?

Table 1.2.3.53 shows a redefined class `TPerson` which assumes, in general, that a person is a meat eater.

Table 1.2.3.54 shows the declaration of a class `TXPerson` which inherits from `TPerson`.
In general, a person classified as an X person is a vegetarian. Therefore, the class `TXPerson` overrides the inherited `ShowDiet` method.

The software can still access the inherited method `ShowDiet` for any X person who is a meat eater.

It can do this with `inherited ShowDiet`.

The problem arises when we have a lot of different classes of people, e.g. 26 classes

`TAPerson ... TZPerson`

If all 26 classes inherit from `TPerson`, then each will need to provide an implementation of `ShowDiet` which outputs the diet of the majority.

This is a lot of duplication of code!

Although this example is a little contrived and very simplified, it is meant to make the following point

```
Unit Person;
Interface
  Type
    TPerson = Class
                Private
                  PersonsName : String;
                Public
                  Constructor Create(Name : String);
                  Procedure ShowDiet; Virtual;
              End;
Implementation
   Constructor TPerson.Create(Name : string);
    Begin
      PersonsName := Name;
    End;
  Procedure TPerson.ShowDiet;
    Begin
      Writeln(PersonsName, ' is a meat eater');
    End;
End.
```

Table 1.2.3.53 Class TPerson

```
Unit XPerson;
Interface
  Type
    TXPerson = Class(TPerson)
                 ...
                Public
                  Procedure ShowDiet; Override;
                  ...
              End;
Implementation
  Procedure TPerson.ShowDiet;
    Begin
      Writeln(PersonsName, ' is a pulses eater');
    End;
End.
```

Table 1.2.3.54 Class TXPerson

To build software so that when it needs to be changed it can be done with the least impact on the existing code the following design principles need to be applied

- **encapsulate what varies**
- **favour composition over inheritance**

If `TPerson`'s class uses composition as shown in *Table 1.2.3.52* and not inheritance as shown in *Table 1.2.3.53* then duplication of code for `ShowDiet` can be eliminated.

Program to interfaces, not implementation

Chapter 1.2.2 covered the concept of coupling in connection with structured programming and design.

Coupling measures the strength of relationships between modules.

It was stated in *Chapter 1.2.2* that the objective of structured design is to minimise the coupling between modules so that the modules are as independent as possible.

The same objective applies in object-oriented design and programming.

In the object-oriented programming paradigm, **the object is the fundamental unit of modularity**.

Key concept

Unit of modularity:
The object is the fundamental unit of modularity.

In structured programming, the subroutine is the fundamental unit of modularity.

Loose coupling in structured programming is achieved when a subroutine's parameter list is its (data) interface with other subroutines and the main program.

Single licence - Abingdon School

Loose coupling in object-oriented programming is achieved when an object's interface with other objects is via methods and properties whose identifiers (names) but not implementation are made public.

Some object-oriented programming languages allow a declaration of methods and properties without writing their implementation.

This type of declaration without an implementation is called an **interface**.

In Delphi, the keyword **Interface** is used for this purpose.

Table 1.2.3.55 shows an interface `ISum` declared in a unit `ISumUnit`.
It contains one method `SumToN` which returns an integer.
The number string below the `Interface` keyword is a Globally Unique Identifier (GUID) which is used by the compiler to identify this interface. It is generated in the IDE by typing CTRL+SHIFT+G.

Note there is no implementation code.

Interfaces consist of a name (in this case, `ISum`) and a declaration of methods and properties.

It is conventional to start the name given to the interface with the letter "I".

Generally, an interface does not allow object attributes (fields), variables or constants to be declared but check what is possible in the language that you use.

An interface is purely a declaration of capability.

An interface cannot use access modifiers such as private, protected, etc.

Thus, every member of an interface is by default public.

An interface can't do anything without an implementing class.

The implementing class must declare that it is implementing the interface (it does this by including the interface identifier in brackets) and then must make sure that it implements all the methods declared in the interface.

Table 1.2.3.56 shows one possible implementation of the interface `ISum` by class `TSum`.

Information

Object property:
We have called the data fields that store an object's state the attributes of the object. The object property is a mechanism of allowing indirect access to an attribute via an alias, and getter and setter methods which create the impression that the attribute is being accessed directly. The alias is used in assignment statements to set the value of its underlying attribute and copied from to read the value of its underlying attribute.

```
Unit ISumUnit;
Interface
  Type
    ISum = Interface
             ['{66642D9A-C0BC-4DC1-9F8D-308704207A1E}']
             Function SumToN : Integer;
           End;
Implementation
End.
```

Table 1.2.3.55 Declaring an interface ISum

```
Unit TSumUnit;
Interface
  Uses ISumUnit;
  Type
    TSum = Class(TInterfacedObject, ISum)
             Public
               Function SumTo(n : Integer) : Integer;
           End;
Implementation
  Function TSum.SumTo(n : Integer) : Integer;
    Var
      Sum, i : Integer;
    Begin
      Sum := 0;
      For i := 1 To n
        Do Sum := Sum + i;
      Result := Sum;
    End;
End.
```

Table 1.2.3.56 TSumUnit showing the implementing class, TSum

Class `TSum` inherits from a base class, `TInterfacedObject`, supplied by Delphi. This base class supports the management of interfaces and must be used. Class `TSum` declares that it is implementing `ISum` by including a reference to `ISum` in brackets.

Single licence - Abingdon School

Table 1.2.3.57 shows a program which uses an object reference of type `ISum` and an object of type `TSum`. The `TSum` object is sent the message `SumTo(n)` where the value of `n` is read from the keyboard.

This is a highly contrived example to illustrate a point that interfaces make possible something that is critical to writing good code:

They allow programming to an abstraction and not an implementation.

Interfaces are abstractions.

Programming against an abstraction, means that the program is not coupled to a specific implementation.

For example, the alternative implementation of class `TSum` shown in *Table 1.2.3.58* can be substituted for the original shown in *Table 1.2.3.56* without breaking the program shown in *Table 1.2.3.57* as long as `TSum` implements `ISum`.

```
Program SumToNExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils,
  ISumUnit in 'ISumUnit.pas',
  TSumUnit in 'TSumUnit.pas';
Var
  n : Integer;
  SumObjectReference : ISum;
Begin
  SumObjectReference := TSum.Create;
  Write('Input n: ');
  Readln(n);
  Writeln(SumObjectReference.SumTo(n));
  Readln;
End.
```

Table 1.2.3.57 Program which uses an object reference of type ISum and an object of type TSum.

```
Unit TSumUnit;
Interface
  Uses ISumUnit;
  Type
    TSum = Class(TInterfacedObject, ISum)
             Public
               Function SumTo(n  : Integer) : Integer;
           End;
Implementation
  Function TSum.SumTo(n : Integer) : Integer;
    Begin
      Result := ((n + 1) * n) Div 2;
    End;
End.
```

Table 1.2.3.58 TSumUnit showing an alternative implementation of class, TSum

Key point

More than one source of behaviour:

By using interfaces, you can, for example, include behaviour from multiple sources in a class.

For example,

Interface

```
TSomeClass = Class((TInterfacedObject, IEx1, IEx2,IEx3)
                ...
             End
```

Information

The example programs in this chapter are available to download from www.educational-computing.co.uk/aqacs/alevelcs.html.

Information

Keyword Interface:

Interfaces are also supported in C#,VB.NET and Java.

Python does not support interfaces but it is possible to use an abstract class in which all methods are abstract and multiple inheritance to achieve a similar outcome

C#:

```
interface IDiet
{
    method declarations
}
```

VB.NET:

```
Interface IDiet
    method declarations
End Interface
```

Java:

```
public interface IDiet
{
  method declarations
}
```


Single licence - Abingdon School

Questions

1. What **three** design principles enable software to be built so that when it needs to be changed it can be done with the least impact on the existing code?

2. A computer activity game has game characters which acquire specific tools during the game.
Each character object can make use of one tool object at a time, but can change tools at any time during the game. The following tools are available
   - *saw*
   - *lockpick*
   - *screwdriver*
   - *hammer*

   Each tool has a different behaviour.
   A tool object has a method `UseTool` which implements the appropriate behaviour.
   A character class `Character` has an attribute `Tool` of type `ToolBehaviour`.
   A tool object is of a type which describes the object's behaviour and may be one of
   - *SawBehaviour*
   - *LockPickBehaviour*
   - *ScrewdriverBehaviour*
   - *HammerBehaviour*

   A character may be one of the following
   - *master craftsman*
   - *apprentice*
   - *journeyman*

   The activity game uses an interface `ToolBehaviour` and eight classes in this activity game.
   (a) Explain why declaring `ToolBehaviour` as an interface is a good design decision.
   (b) Identify eight classes in this scenario.

3. The following are statements related to interfaces:
   - *"An interface allows programming to an abstraction not an implementation."*
   - *"An interface cannot declare a constructor."*
   - *"An interface cannot have attributes for object state."*
   - *"An interface can't do anything without an implementing class."*

   Explain these statements.

4. Explain why it is a good idea to favour composition over inheritance.

In this chapter you have covered:

- Been made aware of the following object-oriented design principles:
  - encapsulate what varies
  - favour composition over inheritance
  - program to interfaces, not implementation

Single licence - Abingdon School

1 Fundamentals of programming

1.2 Programming paradigms

Learning objectives:

- *Be able to write object-oriented programs*

1.2.3(5) Object-oriented programming

Introduction

In structured programming computations are expressed as shown in *Table 1.2.3.59* in terms of states (e.g. `n` has value 1 initially) and sequences of potentially state-changing operations (e.g. `n ← n + 1`).

```
n ← 1
Sum ← 0
While i <= 10
  Sum ← Sum + n
  n ← n + 1
EndWhile
```

Table 1.2.3.59 While loop to count the first 10 natural numbers

In object-oriented programming, computations are expressed in terms of a collection of objects which exchange messages by using interface operations (functions and procedures, collectively known as methods).

Figure 1.2.3.9 Rolling a pair of six-sided dice

An object is characterized by a state and an interface specifying a collection of potentially state-changing operations.

For example, suppose we want to model a six-sided die in an object-oriented program. *Figure 1.2.3.9* shows a pair of six-sided dice being rolled.

The potentially state-changing operations that we might wish our die object to respond to are

- roll die - randomly selects a number between 1 and 6
- get number - returns this number
- set number - sets this number

The state that we need our die object to retain after we have rolled it or set its number is the equivalent of which face of the real die it models is upwards.

The operation *get number* reads the number on this face and returns this number. The operation *roll die* rolls the die to produce a randomly chosen number in range 1 to 6 which becomes the new state of the object.

The operation *set number* sets the die's state to a given number, e.g. `SetNumber(5)` sets it to 5. This is the equivalent of manually choosing which face of the die is upwards.

The message `GetNumber` to the object now returns the number `5`. The key fact to note is that the die object retains this state of being 5 after `SetNumber(5)` has finished its operation. **Objects retain state**.

Single licence - Abingdon School

Figure 1.2.3.10 shows an instance of the `Die` class, i.e. a `Die` object, `Die1`. It has one attribute, `Number`, and can respond to three interface methods, `SetNumber(n)`, `Roll` and `GetNumber`.

How do we create a class from which die objects with state can be created?

The following sections show how this can be done in Python, Object Pascal, C#, VB.NET, and Java.

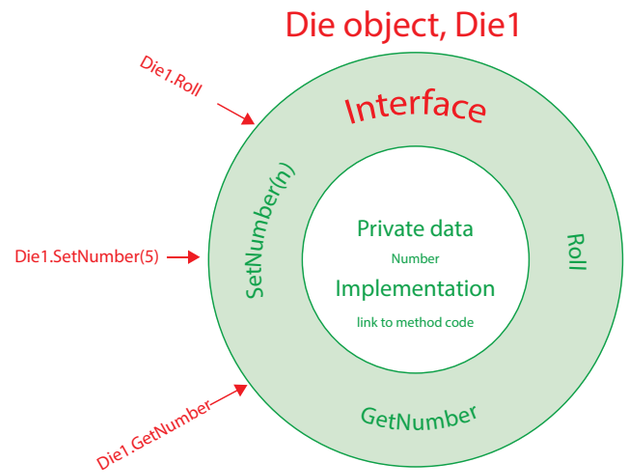


Figure 1.2.3.10 Die Object messaging

Python

Table 1.2.3.60 shows how a `Die` class can be written in Python.

The first parameter of a method in Python is special. It always contains a reference to the object on which the method is acting.

For example if we have an object called `dieObject` and we send it a message `dieObject.roll` then the `roll` method will be called with parameter `self` initialised to point to `dieObject`. The random number generator will run and the number it generates will be assigned to the `__number` attribute of `dieObject`.

Information

In Python, the convention is to use the following to indicate private and protected attributes.
Two underscores = private.
One underscore = protected.
Python does not enforce these specifiers.

Mechanism for creating attribute, *__number*

Saved in DieClass.py

constructor

```
from random import randrange
class Die:
    def __init__(self):
        self.__number = 1
    def roll(self):
        self.__number = randrange(1, 7)
    def getNumber(self):
        return self.__number
    def setNumber(self, n):
        self.__number = n
```

Table 1.2.3.60 Die Class in Python

When `roll` is called we don't include `self` as a parameter, e.g. it is `roll()` and not `roll(self)`.
You will see that in other OOP languages the `self` parameter is implicit and not as in Python where it is explicit.

So far we have not mentioned how a die object comes into existence, i.e. acquires memory space for its attribute, `__number`, and a link to the methods which can act on the object. This is the role of `Die`'s constructor.

Python has built-in methods called "magic methods". The Python language uses its own methods because everything in the Python language is an object. Variables in Python are reference variables to objects.
Each Python variable holds the memory address of the object which it references, e.g. in the statement `x = 6`, `x` is a reference variable which points to an object whose state is currently the integer value 6.

Single licence - Abingdon School

Suppose we have another Python variable, `y`, to which we assign the integer value **7**, i.e. `y = 7`, and we want `x + y`.

When we sum two objects `x` and `y` in Python, what really happens is that the `__add__` method of the `x` object is called as follows

```
x.__add__(y)
```

This is an example of the application of a magic method.

Magic methods are special methods that you can use to add "magic" to your classes. They are always surrounded by double underscores (e.g. `__init__` or `__add__`).

The method `__init__` is a magic method. It can be called to create and initialise an object. Under the hood it actually calls another magic method called `__new__` which gets memory space before `__init__` initilialises attributes.

DieClass.py DieExample.py Class

```
from DieClass import Die
die1 = Die()
die1.roll()
print(die1.getNumber())
print(die1.getNumber())
die1.roll()
print(die1.getNumber())
print(die1.getNumber())
die1.setNumber(5)
print(die1.getNumber())
```

Table 1.2.3.61 DieExample program in Python

Table 1.2.3.61 shows how the `Die` class may be used in a program, DieExample.py.
First the `Die` class is imported from the file DieClass.py. A `Die` object is then created and its memory address assigned to reference variable `die1`.
This object is then sent the message `roll()`.
Next the number just rolled is fetched by sending the object the message `getNumber()`.
In fact `getNumber()` is sent twice to verify that the object's state has not changed between number requests.
The `die1` object is rolled again and its state examined twice.
Finally its state is set to **5** by sending it the message `setNumber(5)` and then the state is examined.

constructor

```
from random import randrange
class Die:
    def __init__(self, dieNoOfFaces):
        self.__number = 1
        self.__noOfFaces = dieNoOfFaces
    def roll(self):
        self.__number = randrange(1, self.__noOfFaces + 1)
    def getNumber(self):
        return self.__number
    def setNumber(self, n):
        self.__number = n
```

Table 1.2.3.62 Die Class in Python which allows the number of faces to be set when creating the object

Table 1.2.3.62 shows a modified class `Die` with a constructor which creates a die with a given number of faces. The constructor has a parameter `dieNoOfFaces` which is used to specify the number of faces. The object `die1` now has a second attribute, `__noOfFaces`.
In the constructor call `Die(8)`, in *Table 1.2.3.63*, the parameter `dieNoOfFaces` is assigned the integer value **8** which the constructor then uses to initialise `die1` object's attribute `__noOfFaces`. This attribute value is used as an argument to `randrange` in the `roll` method, but to access this attribute value on object `die1` requires use of the `self` reference parameter/variable, i.e. `self.__noOfFaces`.

DieClass2.py Class

```
from DieClass2 import Die
die1 = Die(8)
die1.roll()
print(die1.getNumber())
print(die1.getNumber())
die1.roll()
print(die1.getNumber())
print(die1.getNumber())
die1.setNumber(5)
print(die1.getNumber())
```

Table 1.2.3.63 Die program in Python

Single licence - Abingdon School

Programming task

1. In the Python programming environment which you use, create the six-sided Die class DieClass.py and the DieExample program DieExample.py. Run this program and check that its rolls a six-sided die as expected. Modify this program so that it generates and displays the outcome of rolling the die 10 times.
2. Change the DieClass so that it supports the creation of Die objects with a given number of faces. Test that your modified Die class works for a twelve-side die object.

Delphi/Pascal

Table 1.2.3.64 shows the die class, `TDie`, in Delphi/Pascal. In Delphi the convention is to use the prefix `T` if the identifier identifies a type.

The `Self` variable is a hidden parameter of every method in an object. It allows the method to refer to the object on which it is acting.

The language keyword `Result` which is used in `TDie.GetNumber` is a variable used to hold the return value of the function.
In Pascal, the return value would be assigned as follows using the name of the function in place of `Result`

```
GetNumber := Self.Number;
```

The unit contains a declaration of a constructor called `Create` which, in addition to acquiring memory for an object of type `TDie`, initialises the attribute, `Number`.

The program which uses `TDie` to create an object `Die1` is shown in *Table 1.2.3.65*.
Delphi allows a program to destroy an object explicitly by invoking `Destroy` on the object (and `Free`). The last statement in the DieExample program invokes `Die1.Destroy` to release the memory occupied by the `Die1` object.

Mechanism for creating attribute, *Number*

Constructor

Self can be omitted because in this context Delphi assumes *self* applies.

Saved in TDieClassUnit.pas

Generates a random number in range 0 to 5

```
Unit TDieClassUnit;
Interface
Type
  TDie = Class
    Private
      Number : Integer;
    Public
      Procedure Roll;
      Function GetNumber : Integer;
      Procedure SetNumber(n : Integer);
      Constructor Create;
    End;
Implementation
  Constructor TDie.Create;
    Begin
      Self.Number := 1;
    End;
  Procedure TDie.Roll;
    Begin
      Randomize;
      Self.Number := Random(6) + 1;
    End;
  Function TDie.GetNumber : Integer;
    Begin
      Result := Self.Number;
    End;
  Procedure TDie.SetNumber(n : Integer);
    Begin
      Self.Number := n;
    End;
End.
```

Table 1.2.3.64 TDie Class Unit in Delphi/Pascal

DieExample.dproj

Creates a TDie object

Releases memory allocated to a TDie object

Destructor

```
Program DieExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils,
  TDieClassUnit in 'TDieClassUnit.pas';
Var
  Die1 : TDie;
Begin
  Die1 := TDie.Create;
  Die1.Roll;
  Writeln(Die1.GetNumber);
  Writeln(Die1.GetNumber);
  Die1.Roll;
  Writeln(Die1.GetNumber);
  Writeln(Die1.GetNumber);
  Die1.SetNumber(5);
  Writeln(Die1.GetNumber);
  Readln;
  Die1.Destroy;
End.
```

Table 1.2.3.65 DieExample Program in Delphi

Single licence - Abingdon School

Most object-oriented languages have an automatic mechanism for releasing memory allocated to objects which are no longer required. The mechanism is called *garbage collection* and the software that does this, a *garbage collector*. Delphi operates a garbage collector but it also allows a programmer to control object destruction as well.

`Readln` is included to stop the console window closing automatically. `Readln` pauses the program execution until the Enter key is pressed.

Programming task

3. In the Delphi/Pascal programming environment which you use, create the six-sided Die class in a unit file DieClass.pas, and the DieExample program. Run this program and check that it rolls a six-sided die as expected.
   Modify this program so that it generates and displays the outcome of rolling the die 10 times.
4. Change the DieClass so that it supports the creation of Die objects with a given number of faces. Test that your modified Die class works for a twelve-sided die object.

C#

Table 1.2.3.66 shows the die class, `Die`, in C#.

The variable `this` is a hidden parameter of every method in an object. It allows a method to refer to the object on which it is acting.

The random number generator used by a `Die` object is itself an object created with

```
new Random()
```

Therefore each `Die` object needs to contain its own pseudorandom number generator object so that it can select the next random number.

The reference to the random number generator object is stored in attribute `n` of each `Die` object.

In C# the constructor must be given the same name as the class, i.e. `Die`.

Table 1.2.3.67 shows program `DieExample` which creates a `Die` object, `die1`, as follows

```
Die die1 = new Die();
```

The data type of reference variable `die1` is `Die` and a `Die` object is created with `new Die()` where the latter is the constructor.

`Console.ReadLine()` is called at the end of the program to stop the console window closing until the Enter key is pressed.

```
using System;
namespace DieExample
{
    class Die
    {
        private int Number;
        private Random n;
        public Die()
        {
            this.Number = 1;
            n = new Random();
        }
        public void Roll()
        {
            this.Number = n.Next(6) + 1;
        }
        public int GetNumber()
        {
            return this.Number;
        }
        public void SetNumber(int num)
        {
            this.Number = num;
        }
    }
}
```

Mechanism for creating attribute, *Number*

Each object needs a pseudorandom number sequence generator object

Constructor

"this" is a language keyword which is used to reference the object on which the method is acting. It performs a similar role to *self* in Python

Saved in DieClass.cs

Table 1.2.3.66 Die Class in C#

```
class Program
    {
        static void Main(string[] args)
        {
            Die die1 = new Die();
            die1.Roll();
            Console.WriteLine(die1.GetNumber());
            Console.WriteLine(die1.GetNumber());
            die1.Roll();
            Console.WriteLine(die1.GetNumber());
            Console.WriteLine(die1.GetNumber());
            die1.SetNumber(5);
            Console.WriteLine(die1.GetNumber());
            Console.ReadLine();
        }
    }
}
```

Creates a Die object

DieExample.cs

Table 1.2.3.67 DieExample Program in C#

Single licence - Abingdon School

Programming task

5 In the C# programming environment which you use, create the six-sided Die class, in a file DieClass.cs, and the DieExample program. Run this program and check that its rolls a six-sided die as expected. Modify this program so that it generates and displays the outcome of rolling the die 10 times.

6 Change DieClass so that it supports the creation of Die objects with a given number of faces. Test that your modified Die class works for a twelve-sided die object.

VB.NET

Table 1.2.3.68 shows the die class, `Die`, in VB.NET.

The random number generator used by a `Die` object is itself an object created with

```
New Random()
```

In VB.NET, a constructor for a class is named `New`. The constructor for the `Die` class is defined as follows

```
Public Sub New()
  number = 1
End Sub
```

As well as acquiring memory for the attributes of a `Die` class object, it initialises attribute `number` setting it to 1.

Object `die1` is created in *Table 1.2.3.69* with the following statement

```
Dim die1 As Die = New Die()
```

`Console.ReadLine()` is called at the end of the program to stop the console window closing until the Enter key is pressed.

Saved in DieClass.vb

```
Public Class Die
    Private number As Integer
    Private n As New Random()
    Public Sub New()
        number = 1
    End Sub
    Public Sub Roll()
        Randomize()
        number = n.Next(6) + 1
    End Sub
    Public Function GetNumber() As Integer
        Return number
    End Function
    Public Sub SetNumber(ByVal num As Integer)
        number = num
    End Sub
End Class
```

Mechanism for creating attribute, *number*

Each object needs a pseudorandom number sequence generator object

Constructor

Table 1.2.3.68 Die Class in VB.NET

DieExample.vb

```
Module Module1
    Sub Main()
        Dim die1 As Die = New Die()
        die1.Roll()
        Console.WriteLine(die1.GetNumber())
        Console.WriteLine(die1.GetNumber())
        die1.Roll()
        Console.WriteLine(die1.GetNumber())
        Console.WriteLine(die1.GetNumber())
        die1.SetNumber(5)
        Console.WriteLine(die1.GetNumber())
        Console.ReadLine()
    End Sub
End Module
```

Creates a Die object

Table 1.2.3.69 DieExample Program in VB.NET

Programming task

7 In the VB.NET programming environment which you use, create the six-sided Die class, in a file DieClass.vb, and the DieExample program. Run this program and check that its rolls a six-sided die as expected.
Modify this program so that it generates and displays the outcome of rolling the die 10 times.

8 Change DieClass so that it supports the creation of Die objects with a given number of faces. Test that your modified Die class works for a twelve-sided die object.

Single licence - Abingdon School

Java

Table 1.2.3.70 shows the die class, `Die`, in Java.

The random number generator used by a `Die` object is itself an object created with

```
new Random()
```

Therefore each `Die` object needs to contain a pseudorandom number generator object so that it can select the next random number.

The reference to the random number generator object is stored in attribute `n` of each `Die` object.

In Java the constructor must be given the same name as the class, i.e. `Die`.

Table 1.2.3.71 shows program `DieExample.java` which creates a `Die` object, `die1`, as follows

```
Die die1;
die1 = new Die();
```

Programming task

9 In the Java programming environment which you use, create the six-sided Die class, in a file DieClass.java, and the DieExample program. Run this program and check that its rolls a six-sided die as expected. Modify this program so that it generates and displays the outcome of rolling the die 10 times.

10 Change DieClass so that it supports the creation of Die objects with a given number of faces. Test that your modified Die class works for a twelve-sided die object.

```
import java.util.Random;
public class Die {
      private int Number;
      private Random n;
      public Die()
      {
            Number = 1;
            n = new Random();
      }
      public void Roll()
      {
          Number = n.nextInt(6)+ 1;
      }
      public int getNumber()
      {
            return Number;
      }
      public void setNumber(int num)
      {
            Number = num;
      }
}
```

Mechanism for creating attribute, *Number*

Constructor

Saved in DieClass.java

Table 1.2.3.70 Die Class in Java

```
public class DieExample {
    public static void main(String[] args) {
     Die die1;
     die1 = new Die();
     die1.Roll();
     System.out.println(die1.getNumber());
     System.out.println(die1.getNumber());
     die1.Roll();
     System.out.println(die1.getNumber());
     System.out.println(die1.getNumber());
     die1.setNumber(5);
     System.out.println(die1.getNumber());
    }
}
```

Creates a Die object

DieExample.java

Table 1.2.3.71 DieExample Program in Java

Abstract methods

Some classes should never be instantiated. For example, creating an instance of the Bird class covered in *Chapter 1.2.3(1)* doesn't make any sense when we don't know what type of bird it is.

Object Pascal/Delphi

Table 1.2.3.72 shows the Object Pascal declaration of the Bird class `TBird` from *Chapter 1.2.3(1)*.

```
Unit BirdClass;
Interface
  Type
    TBird = Class
              Public
                Name : String;
              Public
                Procedure MakeSound; Virtual;
                Constructor Create(BirdName : String);
              End;
Implementation
  Procedure TBird.MakeSound;
    Begin
      Writeln('Screech');
    End;
  Constructor TBird.Create(BirdName : String);
    Begin
      Name := BirdName;
    End;
End.
```

Base class

Virtual methods can be overridden at run time

Create inherited by subclasses

Table 1.2.3.72 Object Pascal Bird class unit

If we make `TBird` method `MakeSound` **abstract** we can eliminate its implementation. We still retain the constructor `Create(BirdName : String)` because we need this to acquire memory for a bird object's attribute `Name` whether it is a crow or a duck and to assign to it the bird's name.

Single licence - Abingdon School

```
Unit BirdClass;
Interface
  Type
    TBird = Class
              Public
                Name : String;
              Public
                Procedure MakeSound; Virtual; Abstract;
                Constructor Create(BirdName : String);
              End;
Implementation
    Constructor TBird.Create(BirdName : String);
    Begin
      Name := BirdName;
    End;
End.
```

Table 1.2.3.73 Pascal/Delphi Bird class unit with abstract method

Table 1.2.3.73 shows the new Bird class declaration with abstract method `MakeSound`.

Having at least one abstract method makes the class an abstract class.

The abstract method `MakeSound` is implemented in both `TDuck` and `TCrow`. Each inherits `TBird` but implements `MakeSound` in a way which is relevant to their class. *Table 1.2.3.74* shows the declaration of `TDuck` class.

```
Unit DuckClass;
Interface
  Uses BirdClassAbstract;
  Type
    TDuck = Class (TBird)
              Public
                Procedure MakeSound; Override;
              End;
Implementation
  Procedure TDuck.MakeSound;
    Begin
      Writeln('Quack');
    End;
End.
```

Table 1.2.3.74 Pascal/Delphi Duck class

```
Program BirdExampleAbstract;
{$APPTYPE CONSOLE}
Uses
  System.SysUtils, DuckClass,
  CrowClass, BirdClassAbstract;
Var
  Bird1 : TBird;
Begin
  Bird1 := TCrow.Create('Crow');
  Writeln(Bird1.Name);
  Bird1.MakeSound;
  Bird1.Destroy;
  Bird1 := TDuck.Create('Duck');
  Writeln(Bird1.Name);
  Bird1.MakeSound;
  Bird1.Destroy;
  Readln;
End.
```

Table 1.2.3.75 Delphi BirdExample which uses abstract Bird class

Table 1.2.3.75 shows the use of both `TDuck` and `TCrow` classes to create a crow object and then a duck object.

`MakeSound` behaves **polymorphically**.

The use of abstract methods and polymorphism go hand-in-hand.

The `TDuck` class *inherits* from class `TBird`.

Key term

Abstract method:

An abstract method is a method that has a declaration but no method body. Inheriting classes must implement the abstract method.

C#

Table 1.2.3.76 shows a declaration of an abstract class in C#.

Table 1.2.3.77 shows a program which uses this abstract Bird class.

```
namespace BirdAbstract
{
    abstract class BirdClassAbstract
    {
        public string Name;
        public BirdClassAbstract(string name)
        {
            Name = name;
        }
        public abstract void MakeSound();
    }
}
```

Table 1.2.3.76 C# Bird abstract class with abstract method MakeSound

```
using System;
namespace BirdAbstract
{
    class Program
    {
        static void Main(string[] args)
        {
            BirdClassAbstract bird1;
            bird1 = new Crow("Crow");
            bird1.MakeSound();
            Console.WriteLine(bird1.Name);
            bird1 = new Duck("Duck");
            bird1.MakeSound();
            Console.WriteLine(bird1.Name);
            Console.ReadLine();
        }
    }
}
```

Table 1.2.3.77 C# Bird example program which uses an abstract Bird class

Key term

Abstract class:

An abstract method in a class makes the class an abstract class.

Single licence - Abingdon School

Table 1.2.3.78 shows the Duck class which implements the abstract method `MakeSound`.
Using a common superclass makes it possible to avoid code duplication in its subclasses as well as simplifying the code in applications which use the classes.

```
namespace BirdAbstract
{
    class Duck : BirdClassAbstract
    {
        public Duck(string name) : base(name)
        {
            this.Name = name;
        }
        override public void MakeSound()
        {
            Console.WriteLine("Quack");
        }
    }
}
```

Table 1.2.3.78 C# Duck class which implements abstract method MakeSound

VB.NET

Table 1.2.3.79 shows a declaration of an abstract class in VB.NET. *Table 1.2.3.81* shows a program which uses this abstract Bird class. *Table 1.2.3.80* shows the Duck class which implements the abstract method `MakeSound`.

```
Public MustInherit Class BirdClass
    Public Property NameOfBird As String
    Public Sub New(ByRef name As String) 'constructor
        NameOfBird = name
    End Sub
    MustOverride Sub MakeSound()
End Class
```

Table 1.2.3.79 VB.NET abstract Bird class with abstract method MakeSound

```
Public Class DuckClass : Inherits BirdClass
    Public Sub New(ByRef name As String)
        MyBase.New(name)
        NameOfBird = name
    End Sub
    Public Overrides Sub MakeSound()
        Console.WriteLine("Quack")
    End Sub
End Class
```

Table 1.2.3.80 VB.NET Duck class which implements abstract method MakeSound

```
Module Module1
    Sub Main()
        Dim bird As BirdClass = New CrowClass("Crow")
        bird.MakeSound()
        Console.WriteLine(bird.NameOfBird)
        bird = New DuckClass("Duck")
        bird.MakeSound()
        Console.WriteLine(bird.NameOfBird)
        Console.ReadLine()
    End Sub
End Module
```

Table 1.2.3.81 VB.NET Bird example program which uses an abstract Bird class

Python

Table 1.2.3.82 shows a declaration of an abstract class in Python. *Table 1.2.3.83* shows a program which uses this abstract Bird class. *Table 1.2.3.84* shows the Duck class which implements the abstract method `MakeSound`.

```
from abc import ABC, abstractmethod
class BirdAbstractClass(ABC):
    @abstractmethod
    def MakeSound(self):
        pass
    def __init__(self, nameOfBird) -> None:
        self.Name = nameOfBird
```

Table 1.2.3.82 Python abstract Bird class with abstract method MakeSound

```
from DuckClass import Duck
from CrowClass import Crow
bird1 = Duck('Duck')
bird1.MakeSound()
print(bird1.Name)
bird1 = Crow('Crow')
bird1.MakeSound()
print(bird1.Name)
```

Table 1.2.3.83 Python Bird example which uses an abstract Bird class

```
from BirdAbstractClass import BirdAbstractClass
class Duck(BirdAbstractClass):
    def MakeSound(self):
        print('Quack')
```

Table 1.2.3.84 Python Duck class which implements abstract method MakeSound

Single licence - Abingdon School

Java

Table 1.2.3.85 shows a declaration of an abstract class in Java. *Table 1.2.3.86* shows a program which uses this abstract Bird class. *Table 1.2.3.87* shows the Duck class which implements the abstract method `MakeSound`.

```
public abstract class BirdAbstractClass {
       public String Name;
       public BirdAbstractClass(String name)
       {
           Name = name;
       }
       abstract public void MakeSound();
}
```

Table 1.2.3.85 Java abstract Bird class with abstract method MakeSound

```
public class BirdAbstract {
    public static void main(String[] args) {
       BirdAbstractClass bird1;
       bird1 = new Duck("Duck");
       bird1.MakeSound();
       System.out.println(bird1.Name);
       bird1 = new Crow("Crow");
       bird1.MakeSound();
       System.out.println(bird1.Name);
    }
}
```

Table 1.2.3.86 Java Bird example program which uses an abstract Bird class

```
public class Duck extends BirdAbstractClass{
       public Duck(String name)
       {
           super(name);// use constructor of superclass
       }
       public void MakeSound()
       {
           System.out.println("Quack");
       }
}
```

Table 1.2.3.87 Java Duck class which implements the abstract method MakeSound

Constructors and destructors

Constructors and destructors are special kinds of methods. These specialised methods control how objects are created prior to use and disposed when no longer needed.

The constructor ensures, beside all other things that constructors may do, that the virtual methods appropriate for a specific object are called.

Destructors release the memory occupied by an object.

Virtual methods

A virtual method is declared with the keyword *virtual*.

Virtual method calls are not resolved immediately by the compiler. The actual method to be executed is determined at runtime, through a process known as **late binding**. Any method can be made virtual by including a **virtual directive** at the end of the method declaration in the class type - *Table 1.2.3.72*.

When a virtual method is referenced, the actual class type of the object referenced by the variable is used to determine which method (procedure/function) will be called.

Very importantly, a descendent class type may override any of the virtual methods that descendent class type inherits from its parent class type - *Table 1.2.3.74*. This override change in the virtual method will also be inherited by descendents of the class with the override.

Single licence - Abingdon School

Static methods

A **static method** is a method attached to the class itself and not any particular instance of the class. This means that a static method can be accessed without an object reference because no object is required.

Creating a class consisting of just static methods creates a **static class**. A static class is a convenient container for methods that just operate on input parameters and do not have to get or set any internal instance attributes, e.g. a set of maths routines (subroutines).

Key term

Static method:
A static method is a method attached to the class itself and not any particular instance of the class. This means that a static method can be accessed without an object reference because no object is required.

For example, suppose there is a need to provide a routine to calculate the Fibonacci sequence for a given input parameter, *n*. The Fibonacci sequence is a sequence of numbers where a number is found by adding up the two numbers before it. The first two numbers are 0 and 1. The third number is therefore 1 (0 + 1) and the fourth 2 (1 + 1).

Table 1.2.3.88 shows the declaration and implementation in Object Pascal/Delphi of a class `TFibonacci` with static method `Fibonacci(n)`.

Table *1.2.3.89* shows an Delphi program `FibonacciExample` which displays the first six Fibonacci numbers in the sequence - *Figure 1.2.3.11*.

Figure 1.2.3.11 Output from FibonacciExample

FibonacciClass.pas

```
Unit FibonacciClass;
Interface
  Type
    TFibonacci = Class
                   Public
                     Class Function Fibonacci (n : Integer) : Integer; Static;
                 End;
Implementation
  Class Function TFibonacci.Fibonacci(n : Integer) : Integer;
    Var
      x, y, i, Temp : Integer;
    Begin
      x := 0; y := 1;
      If n > 0
        Then
          For i := 0 To n - 1
            Do
              Begin
                Temp := x; x := y; y := Temp + y;
              End;
      Result := x;
    End;
End.
```

Table 1.2.3.88 Object Pascal TFibonacci class with static method Fibonacci(n)

```
Program FibonacciExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils,
  FibonacciClass in 'FibonacciClass.pas';
Var
  i : Integer;
Begin
  For i := 0 To 5
    Do Writeln(TFibonacci.Fibonacci(i));
  Readln;
End.
```

Table 1.2.3.89 Delphi program FibonacciExample which uses static class TFibonacci and static class method Fibonacci(n)

Single licence - Abingdon School

Table 1.2.3.90 shows the declaration and implementation of a C# class `TFibonacci` with static method `Fibonacci(n)`.

Table 1.2.3.91 shows a C# program `FibonacciExample` which displays the first six Fibonacci numbers in the sequence.

FibonacciClass.cs

```
using System;
namespace FibonacciExample
{
  class TFibonacci
  {
    public static int Fibonacci(int n)
    { int x = 0; int y = 1;
      for (int i = 0; i < n; i++)
        {
          int temp = x; x = y; y = temp + y;
        }
      return x;
    }
  }
}
```

Table 1.2.3.90 C# TFibonacci class with static method Fibonacci(n)

FibonacciExample.cs

```
using System;
namespace FibonacciExample
{
  class Program
  {
    static void Main(string[] args)
    {
      for (int i = 0; i < 6; i++)
        {
          Console.WriteLine(TFibonacci.Fibonacci(i));
        }
      Console.ReadLine();
    }
  }
}
```

Table 1.2.3.91 C# program FibonacciExample which uses static class TFibonacci and static class method Fibonacci(n)

Table 1.2.3.92 shows the declaration and implementation of a VB.NET class `TFibonacci` with static method `Fibonacci(n)`.

Table 1.2.3.93 shows a VB.NET program `FibonacciExample` which displays the first six Fibonacci numbers in the sequence. VB.NET uses the keyword `Shared` to indicate that a method is static.

FibonacciClass.vb

```
Public Class TFibonacci
  Public Shared Function Fibonacci(ByVal n As Integer)
    Dim x As Integer = 0
    Dim y As Integer = 1
    For i As Integer = 0 To n - 1
      Dim Temp As Integer = x
      x = y
      y = Temp + y
    Next
    Return x
  End Function
End Class
```

Table 1.2.3.92 VB.NET TFibonacci class with static method Fibonacci(n)

Fibonacci.vb

```
Module Module1
  Sub Main()
    For i As Integer = 0 To 5
      Console.WriteLine(TFibonacci.Fibonacci(i))
    Next
    Console.ReadLine()
  End Sub
End Module
```

Table 1.2.3.93 VB.NET program Fibonacci.vb which uses static class TFibonacci and static class method Fibonacci(n)

Table 1.2.3.94 shows the declaration and implementation of a Java class `TFibonacci` with static method `Fibonacci(n)`.

TFibonacci.java

```
public class TFibonacci{
    public static int Fibonacci(int n)
    {
        int x = 0; int y = 1;
        for (int i = 0; i < n; i++)
        {
            int temp = x; x = y; y = temp + y;
        }
        return x;
      }
}
```

Table 1.2.3.94 Java TFibonacci class with static method Fibonacci(n)

Single licence - Abingdon School

Table 1.2.3.95 shows a Java program `FibonacciExample` which displays the first six Fibonacci numbers in the sequence.

Table 1.2.3.96 shows the declaration and implementation of a Python class `TFibonacci` with static method `fibonacci(n)`.

Table 1.2.3.97 shows a Python program `FibonacciExample` which displays the first six Fibonacci numbers in the sequence.

FibonacciExample.java

```
public class FibonacciExample {
    public static void main(String[] args) {
       for (int i = 0; i < 6; i++)
       {
           System.out.println(TFibonacci.Fibonacci(i));
       }
    }
}
```

Table 1.2.3.95 Java program FibonacciExample.java which uses static class TFibonacci and static class method Fibonacci(n)

```
class TFibonacci(object):
    def fibonacci(n):
        x = 0
        y = 1
        for i in range(0, n):
            temp = x
            x = y
            y = temp + y
        return y
    fibonacci = staticmethod(fibonacci)
```

FibonacciClass.py

Table 1.2.3.96 Python TFibonacci class with static method fibonacci(n)

```
from FibonacciClass import TFibonacci
for i in range(0, 5):
    print(TFibonacci.fibonacci(i))
```

FibonacciExample.py

Table 1.2.3.97Python program FibonacciExample.java which uses static class TFibonacci and static class method fibonacci(n)

Public, private and protected specifiers

Access to the **attribute fields** and **methods** of an object may be controlled with the access modifiers/specifiers **public**, **private** and **protected**.

Public

Table 1.2.3.98 shows the declaration of `TPerson` class in Object Pascal. The details of the implementation of its methods are omitted because they are not visible outside the unit. However, the identifiers `AddPersonDetails`, `EditPersonDetails`, `ShowPersonDetails` are visible outside the unit because they have been declared within the `TPerson` class in the `Interface` section as public using the keyword **Public**.

Private

The attributes `Name`, `Address` and `Age` are not visible outside of the unit because they are declared as private with the keyword **Private** in the `Interface` section.

They can be accessed from outside the unit indirectly but only through the public methods of the class, `TPerson`. Inside the unit they may be accessed directly in the implementation section.

PersonClass.pas

```
Unit PersonClass;
Interface
Type
  TPerson = Class
              Private
                Name : String[30];
                Address : String[40];
                Age : Integer;
              Public
                Procedure AddPersonDetails;
                Procedure EditPersonDetails;
                Procedure ShowPersonDetails;
              End;
Implementation
  Procedure TPerson.AddPersonDetails;
    Begin
      ...
    End;
  Procedure TPerson.EditPersonDetails;
    Begin
      ...
    End;
  Procedure TPerson.ShowPersonDetails;
    Begin
      ....
    End;
End.
```

Visible inside unit

Not visible outside unit

Visible outside unit

Hidden

Table 1.2.3.98 TPerson class in Object Pascal showing the use of Private and Public access modifiers

Key term

Access modifiers or specifiers:

It is possible to specify whether access to an attribute or method from outside the unit/module in which declared, is either public, private or protected.

Single licence - Abingdon School

Table 1.2.3.99 shows in red, the error message emitted by the compiler (dcc32 - **D**elphi **C**ommand line **C**ompiler - **32** bit version) on encountering an unsuccessful attempt to access the `Name` attribute outside of the `PersonClass` unit.

```
Program PersonExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils,
  PersonClass in 'PersonClass.pas';
Var
  Person : TPerson;
Begin
  Person := TPerson.Create;
  Person.Name := 'Fred';
End.
```

[dcc32 Error] PersonExample.dpr(13): E2361 Cannot access private symbol TPerson.Name

Table 1.2.3.99 TPersonExample program fails to compile because Name is private

Table 1.2.3.100 shows a modified version of the program which does compile because the method `AddPersonDetails` is labelled public and therefore visible outside the unit `PersonClass`. As both `AddPersonDetails` and `Name` belong to the same class, `TPerson`, method `AddPersonDetails` is allowed to update attribute `Name`.

```
Program PersonExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils,
  PersonClass in 'PersonClass.pas';
Var
  Person : TPerson;
Begin
  Person := TPerson.Create;
  Person.AddPersonDetails;
End.
```

Table 1.2.3.100 Delphi TPersonExample program which does compile because AddPersonDetails is visible outside PersonClass.pas

Protected

Suppose we need to model a special person (this is a very contrived example) in a class `TSpecialPerson`.
A special person will have all the attributes and behaviours of a person plus some additional methods and attributes of its own.
It therefore needs to inherit from class `TPerson`.

```
Unit SpecialPerson;
Interface
  Uses PersonClass;
  Type
    TSpecialPerson = Class (TPerson)
                       Public
                         Procedure AddTitle;
                     End;
Implementation
  Procedure TSpecialPerson.AddTitle;
    Begin
      Name := 'Sir ' + Name;
    End;
End.
```

Table 1.2.3.101 TSpecialPerson class which inherits from TPerson

Table 1.2.3.101 shows the declaration of the class `TSpecialPerson` with one public method, other methods and attributes have been omitted for brevity.

The method procedure `TSpecialPerson.AddTitle` needs to access the attribute `Name` declared in `TPerson`.

Unfortunately, `Name` is a private attribute and therefore not visible outside the unit `PersonClass`.

It is not enough for unit `SpecialPerson` to state `Uses PersonClass` to gain access to `Name`.

The solution is to modify `TPerson` as shown in *Table 1.2.3.102* and declare attribute `Name` as **protected**.

Protected attributes are visible in a class's subclasses, but not otherwise.

```
Unit PersonClass;
Interface
Type
  TPerson = Class
              Protected
                Name : String[30];
              Private
                Address : String[40];
                Age : Integer;
              Public
                Procedure AddPersonDetails;
                Procedure EditPersonDetails;
                Procedure ShowPersonDetails;
              End;
Implementation
  Procedure TPerson.AddPersonDetails;
    Begin
    End;
  Procedure TPerson.EditPersonDetails;
    Begin
    End;
  Procedure TPerson.ShowPersonDetails;
    Begin
    End;
End.
```

Visible in subclasses

Not visible in subclasses

Table 1.2.3.102 Modified TPerson class which allows its subclasses to directly access protected attributes but not private attributes

Single licence - Abingdon School

Programming task

11 Using a language you are familiar with, write an object-oriented program which declares a TPerson class and a TSpecialPerson class in separate files based on *Table 1.2.3.101* and *Table 1.2.3.102*. Provide implementations for AddPersonDetails, EditPersonDetails, ShowPersonDetails and a modified AddTitle which prompts the user for the specific title to use.

Write a main program which

- declares two object reference variables of type TPerson
- creates two objects, one of type TPerson and one of type TSpecialPerson
- populates each object with data
- allows the state of each object to be edited
- shows the state of each object

Key term

Public:
Attribute or method is visible/accessible outside its unit/module.

Private:
Attribute or method is not visible/accessible outside its unit/module.

Protected:
Attribute or method is not visible/accessible outside its unit/module except in its subclasses.

Information

Note that Java interprets protected slightly differently, as a protected attribute can be accessed anywhere within the same package (https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html).

Aggregation

We have already encountered aggregation in *Chapter 1.2.3(2)*. In aggregation an object contains other objects.

Table 1.2.3.103 shows the use of composition, a specialised form of aggregation, in an Object Pascal declaration of class `TGroup`. Any object of type `TGroup` will be able to contain up to 40 `TPerson` objects. The class `TPerson` is declared in *Figure 1.2.3.104*.

The method `AddPersons` adds `TPerson` objects to a group object. *Table 1.2.3.105* shows an example program that declares a `MyGroup` object reference variable of type `TGroup`. `TGroup.CreateGroup` creates a `TGroup` object and assigns its reference to `MyGroup`. `MyGroup.AddPersons` adds a number of `TPerson` objects (*Table 1.2.3.105*) to the `MyGroup` referenced object. `MyGroup.ShowPersons` iterates through the `TPerson` objects stored in `MyGroup` object displaying the attribute values of each in turn.

```
Unit GroupClass;
Interface
Uses PersonClass;
Type
  TGroup = Class
             Private
               Group : Array[1..40] Of TPerson;
               GroupSize : Integer;
             Public
               Constructor CreateGroup(Size : Integer);
               Function GetGroupSize : Integer;
               Procedure AddPersons;
               Procedure ShowPersons;
             End;
Implementation
  Constructor TGroup.CreateGroup(Size : Integer);
    Begin
      GroupSize := Size;
    End;
  Function TGroup.GetGroupSize : Integer;
    Begin
      Result := GroupSize;
    End;
  Procedure TGroup.AddPersons;
    Var
      i : Integer;
    Begin
      For i := 1 To GroupSize
        Do
          Begin
            Self.Group[i] :=  TPerson.Create;
            Self.Group[i].AddPersonDetails;
          End;
    End;
  Procedure TGroup.ShowPersons;
    Var
      i : Integer;
    Begin
      For i := 1 To GroupSize
        Do
          Begin
            Self.Group[i].ShowPersonDetails;
          End;
    End;
End.
```

Table 1.2.3.103 TGroup which uses composition

Single licence - Abingdon School

```
Unit PersonClass;
Interface
  Type
    TPerson = Class
                Private
                  Name : String[30];
                  Address : String[40];
                  Age : Integer;
                Public
                  Procedure AddPersonDetails;
                  Procedure EditPersonDetails;
                  Procedure ShowPersonDetails;
               End;

Implementation
  Procedure TPerson.AddPersonDetails;
    Begin
      Write('Input name: ');
      Readln(Name);
      Write('Input address: ');
      Readln(Address);
      Write('Input age in years: ');
      Readln(Age);
    End;
  Procedure TPerson.EditPersonDetails;
    Begin
    End;
  Procedure TPerson.ShowPersonDetails;
    Begin
      Writeln('Name: ', Name);
      Writeln('Address: ', Address);
      Writeln('Age in years: ', Age);
    End;
End.
```

Table 1.2.3.104 TPerson class

```
Program PersonGroupExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils,
  PersonClass in 'PersonClass.pas',
Var
  MyGroup : TGroup;
  NoOfPersons : Integer;
Begin
  Write('How many persons in the group? ');
  Readln(NoOfPersons);
  MyGroup := TGroup.CreateGroup(NoOfPersons);
  MyGroup.AddPersons;
  MyGroup.ShowPersons;
  Readln;
End.
```

Table 1.2.3.105 PersonGroupExample program

Programming task

12 Using a language you are familiar with, create an object-oriented program based on the Delphi program and units shown in *Tables 1.2.3.103*, *1.2.3.104* and *1.2.3.105*.

13 Add a method ShowPerson to TGroup's declaration. This method shows the details of the person object at a given array index. Adapt your program created in question 12 to use this method.

14 Add a method FindPerson to TGroup's declaration. This method searches for a given name and returns the index in the array of the object with this name if found otherwise it returns -1. Adapt your program created in question 12 to use this method.

15 Add a method EditPerson to TGroup's declaration. This method enables any attribute of a person object to be edited. It should use EditPersonDetails in TPerson for this. Adapt your program created in question 12 to use this method.

Single licence - Abingdon School

Questions

1. A class declaration uses the access modifiers/specifiers *public*, *private* and *protected*. Explain each of these terms.
2. Distinguish method types *static*, *abstract* and *virtual*.

In this chapter you have covered:

- Writing object-oriented programs involving
  - abstract, virtual and static methods
  - inheritance
  - aggregation
  - composition - a specialised form of aggregation
  - polymorphism
  - public, private and protected specifiers

Single licence - Abingdon School

1 Fundamentals of programming

1.2 Programming paradigms

Learning objectives:

- *Be able to draw and interpret class diagrams involving*
  - *single inheritance*
  - *aggregation*
  - *composition*
  - *access modifiers/specifiers*
    - *public (+)*
    - *private (–)*
    - *protected (#)*

1.2.3(6) Object-oriented programming

Class diagrams

Single inheritance, association, aggregation and composition were covered in a practical way in *Chapter 1.2.3(1)* and *Chapter 1.2.3(5)*.

A class diagram can be used to show the relationships and associations between classes.

Figure 1.2.3.12 shows how single inheritance is represented in a class diagram consisting of two classes A and B. Class B is the derived class or subclass which inherits from class A. Each class is drawn as a labelled rectangle.

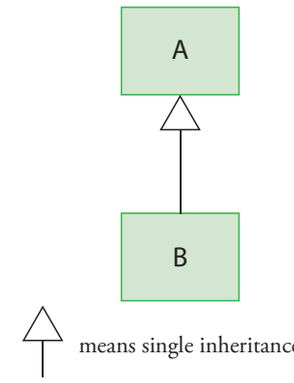


Figure 1.2.3.12 Shows how single inheritance is represented in a class diagram

Figure 1.2.3.13 shows how aggregation (shared association) is represented in a class diagram consisting of two classes C and D. In the whole/part association known as aggregation, Class C is the *whole* and class D is the *part*.

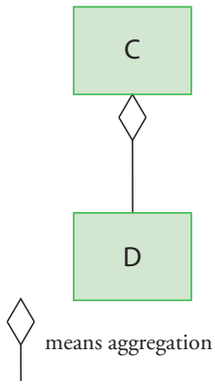


Figure 1.2.3.13 Shows how aggregation is represented in a class diagram

Figure 1.2.3.14 shows how composition (not-shared association) is represented in a class diagram consisting of two classes E and F. In the whole/part association known as composition, Class E is the *whole* and class F is the *part*.

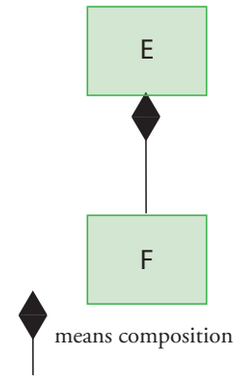


Figure 1.2.3.14 Shows how composition is represented in a class diagram

Figure 1.2.3.15 shows how association[1] is represented in a class diagram consisting of two classes G and H. The direction of the arrow indicates that the association is from G to H, e.g. "snake *uses* ground" where "snake" corresponds to G and "ground" corresponds to H. The association is "uses" of the dependency kind (see *Chapter 1.2.3(2)*).

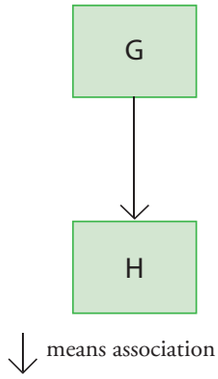


Figure 1.2.3.15 Shows how association is represented in a class diagram

Questions

1 What is the symbol used on a class diagram to indicate

(a) single inheritance (b) composition (c) aggregation (d) association?

1 Association class diagram not in AQA specification

Single licence - Abingdon School

The class diagrams shown so far have omitted the detail of each class. This is fine if all that is of interest is the kind of relationship or association, e.g. single inheritance. However, sometimes a more informative and detailed class diagram is required. Each class in this more detailed class diagram would show details of its attributes and methods and their access level specifiers as shown *Figure 1.2.3.16*. The bubble shows an example of the contents of a class rectangle. *Figure 1.2.3.17* shows the interpretation of the access level specifiers.

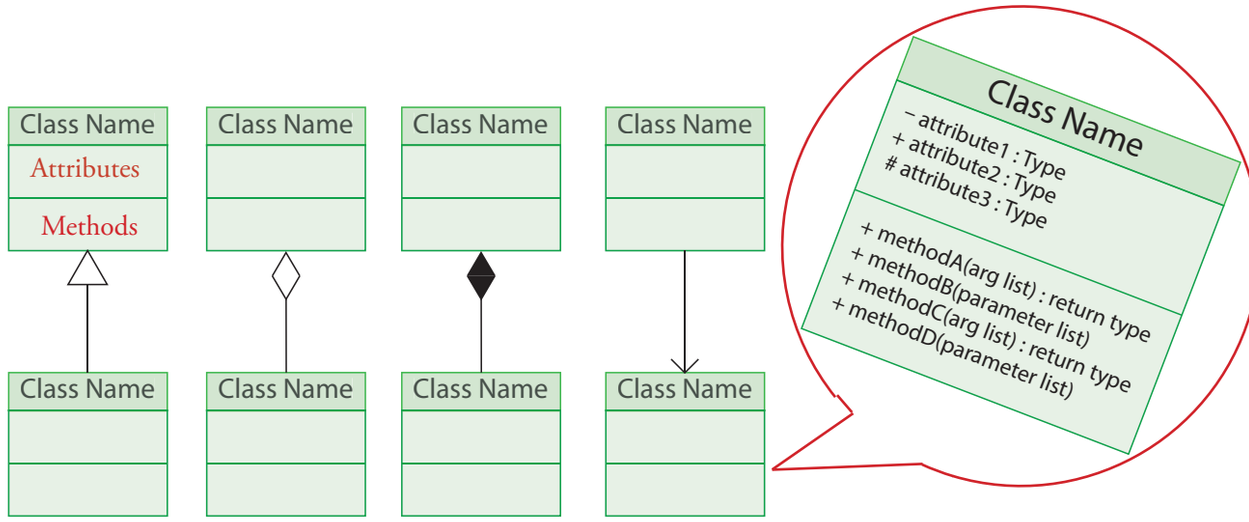


Figure 1.2.3.16 Each class diagram is shown in more detail by splitting the rectangle into an attributes part and a methods part

Access level specifiers

+ means public – means private # means protected

Figure 1.2.3.17 Access level specifiers for attributes and methods

Single inheritance class diagram

Figure 1.2.3.18 shows an example based on an exercise used in *Chapter 1.2.3(4)*.

Figure 1.2.3.19 class diagram is a more detailed version of *Figure 1.2.3.18.* The attributes section is omitted in *Figure 1.2.3.19* from classes `Duck` and `Crow` because the exercise in *Chapter 1.2.3(4)* did not supply these.

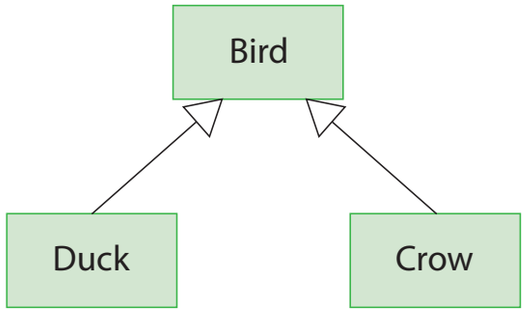


Figure 1.2.3.18 Class diagram demonstrating single inheritance

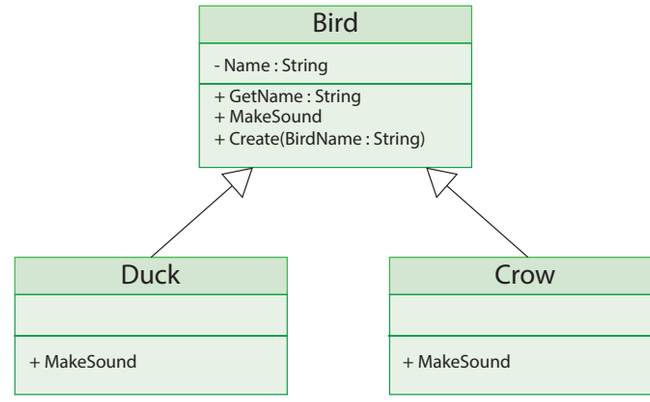


Figure 1.2.3.19 Class diagram demonstrating single inheritance and showing attributes and methods

Figure 1.2.3.19 shows the visibility levels of the attribute `Name` and the three methods `GetName`, `MakeSound` and `Create`. Visibility level is indicated by placing an access modifier or access/visibility level specifier, one of '–', ' +', or ' #', before the attribute or method identifier. If an attribute or method is private then the symbol used is ' –' . If an attribute or method is public then the symbol used is '+' . If an attribute or method is protected then the symbol used is '#'.

Single licence - Abingdon School

Questions

2 Pythons and cobras are types of snake. Draw a single inheritance class diagram which shows the relationship between these three types if each is modelled as a class.

Composition class diagram

Figure 1.2.3.20 shows a composition class diagram for a noughts and crosses grid such as that shown in *Figure 1.2.3.21*. Composition on this class diagram is indicated by a filled diamond and line.

The degree of the association is shown as one grid is associated with nine squares by labelling the diamond with 1 and the other end of the composition symbol with 9. If the degree of association is variable then the 9-end is replaced by *n*.

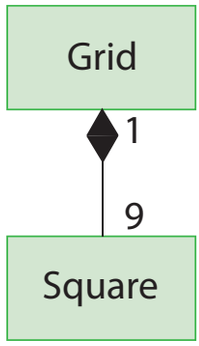


Figure 1.2.3.20 Class diagram demonstrating composition

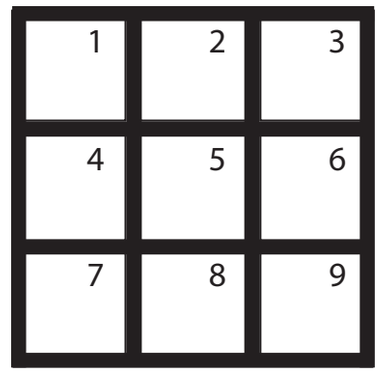


Figure 1.2.3.21 Noughts and crosses grid

Key concept

Composition:
A whole/part relationship or association in which the parts of the whole are not shareable.

Questions

3 A snake has a fang. Draw a class diagram to show the association between snakes and fangs if each is modelled as a class.

Key concept

Association:
Whenever two or more things come together for some purpose the relationship between them is called association.
Association is modelled as a relationship between two classes in object-oriented analysis. It allows one object instance to cause another to perform an action on its behalf.
Aggregation and composition are special forms of association.

Aggregation class diagram

Figure 1.2.3.22 shows a class diagram for aggregation. An unfilled diamond and line are used to indicate aggregation.

Aggregation is a shared association in which an object, e.g. an object of type Student class, is associated with more than one object, e.g. an object of type TutorGroup class and an object of type SportsTeam class.

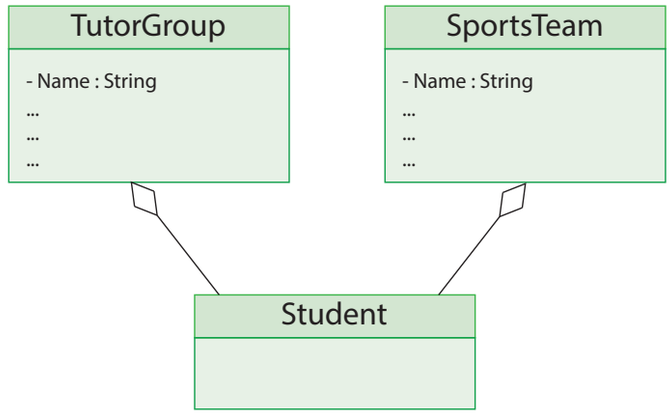


Figure 1.2.3.22 Class diagram demonstrating aggregation

Key concept

Aggregation:
It is a whole/part relationship or association in which a part can be associated with more than one whole. This is aggregation with shareable parts.

Single licence - Abingdon School

Questions

4 A universal remote control is a part of home entertainment systems which consist of televisions, a cable TV set top box, blu-ray players and other electronic devices.

Draw a class diagram that shows the type of association between a universal remote controller and two types of electronic device, a TV and a blu-ray player if these are modelled as classes.

Association

It is likely that you will encounter association which isn't a whole/part association(aggregation association or composition association).

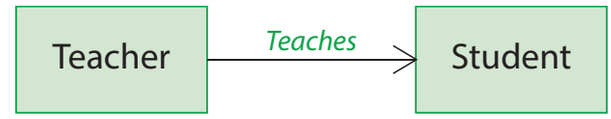


Figure 1.2.3.23 Class diagram demonstrating aggregation

Figure 1.2.3.23 shows an association class diagram for a Teacher class and a Student class. The association is one in which a teacher teaches a student. Clearly, this is not a whole/part association, i.e. a student is not a part of a teacher. Contrast this with the example in *Figure 1.2.3.22* in which an engine is a part of a car, and an engine is a part on a parts list for a car.

The grid for noughts and crosses shown in *Figure 1.2.3.21* does have an association relationship in addition to its aggregation one, it is a *uses* association: a noughts and crosses game *uses* a grid. This uses association is shown in *Figure 1.2.3.24*.

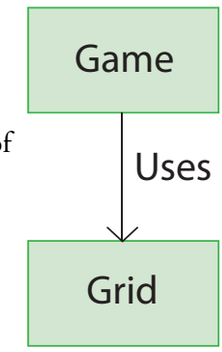


Figure 1.2.3.24 Association class diagram for noughts and crosses game

Questions

5 A class diagram for a simulator to assess the effect of antibiotics and white blood cells on bacteria cells and antibiotic-resistant bacteria cells is shown in *Figure 1.2.3.25*. There are three types of cell. Antibiotics are complex chemical molecules. The system models the three types of cell and antibiotics as particles in the system. The simulation environment supports modelling with up to 10000 particles. Assign the following class types to the correct boxes on a copy of the class diagram shown in *Figure 1.2.3.25*.

Justify your assignments.

Class types: Environment, Particle, Cell, Bacteria, Resistant bacteria, White blood cells, Antibiotics.

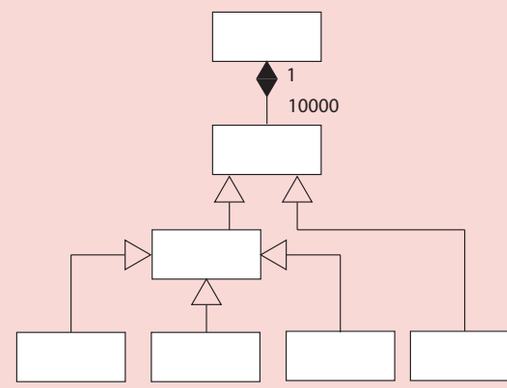


Figure 1.2.3.25 Class diagram of a simulation system

Single licence - Abingdon School

Questions

6 A school has a number of subject departments. Maths is a subject department. Computer Science is a subject department. A subject department has a number of teachers who teach the department's subject. A teacher may teach one or more subjects. A department has teachers who teach in other departments as well.

A software system models this scenario.

Figure 1.2.3.26 shows a class diagram which models a part of the system. Subject departments other than Maths and Computer Science have been omitted.

Assign the following class types to the correct boxes on a copy of the class diagram shown in *Figure 1.2.3.26*.

Justify your assignments.

Class types: School, Department, Teacher, Maths, Computer Science.

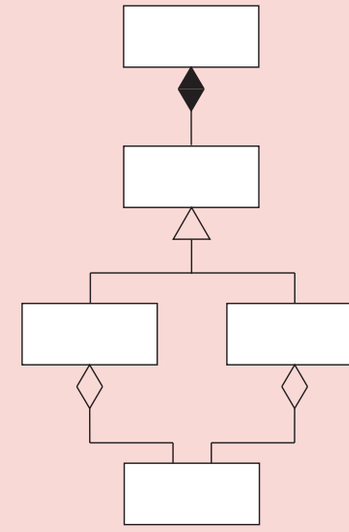


Figure 1.2.3.26 Class diagram of school system

7 An application with a graphical user interface (GUI) has one button and three edit boxes. These are placed on a form, Form1 of type TForm1. The button is of type TButton and the edit boxes of type TEdit. TForm1 is derived from the form type TForm. The application is of type TApplication.

Assign the following class types to the correct boxes on a copy of the class diagram shown in *Figure 1.2.3.27*.

Justify our assignments.

Class types: TApplication, TForm, TForm1, TButton, TEdit.

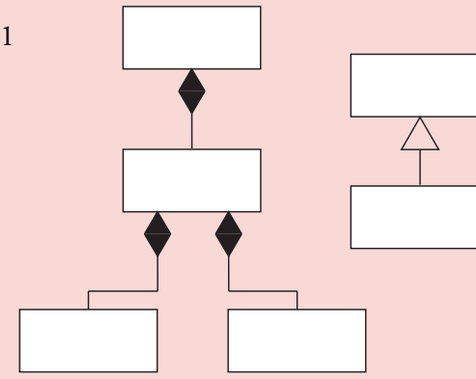


Figure 1.2.3.27 GUI applicaton

Information

Is using an aggregation link worthwhile?

Some people (e.g. Jim Rumbaugh) argue that an aggregation link adds no value and should not be used at all. They argue that an association link can replace the aggregation link in every situation. However, the reverse is not always possible: aggregation cannot replace association in some situations, e.g. when the association is 'uses' such as in 'a snake uses the ground to get around'.

On the other hand, composition is strong association and is useful. The 'owner' object in composition will contain a reference or references to 'owned' objects.

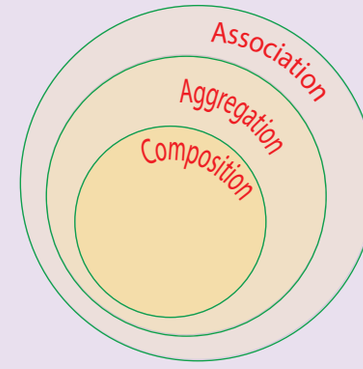


Figure 1.2.3.28 shows the relationship between association, aggregation and composition

Single licence - Abingdon School

Public, private and protected access level specifiers

Chapter 1.2.3(5) covered an example which required the use of access level specifiers *public*, *private* and *protected*.

This example modelled a special person in a class `TSpecialPerson`. A special person has all the attributes and behaviours of a person plus some additional methods and attributes of its own. It therefore inherits from class TPerson. In order to make the attribute `Name` visible outside the unit `PersonClass` it is marked as protected (#).
Protected attributes are visible in a class's subclasses, but not otherwise.

Figure 1.2.3.29 shows a class diagram for the classes, `TPerson` and `TSpecialPerson`. Attributes `Address` and `Age` are private (–). Methods `AddPersonDetails`, `EditPersonDetails`, `ShowPersonDetails` and `AddTitle` are public (+).
The type of method has been made explicit by the use of the keyword `Procedure`.

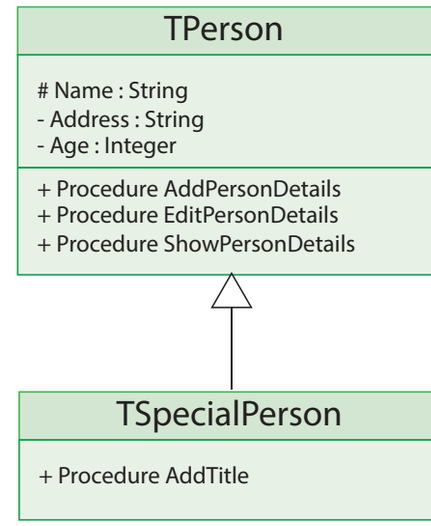


Figure 1.2.3.29 Class diagram showing the use of access specifiers +, – and #

Questions

8 An automobile class, TVehicle, contains private attributes Make, ModelName, EngineCapacity, FuelLevel and a public constructor which initialises these private attributes. A class named TCar is derived from TVehicle. It has public methods Steer, Accelerate, Brake and UpdateFuelLevel. UpdateFuelLevel needs to update the inherited attribute FuelLevel but is prevented from doing so because this attribute is private.

Explain what change needs to be made to make attribute FuelLevel accessible to UpdateFuelLevel and methods in other classes which inherit from TVehicle whilst preventing access from classes which don't.

Draw the class diagram for classes TVehicle and TCar.

In this chapter you have covered:

- Drawing and interpreting class diagrams involving
  - single inheritance
  - aggregation (white diamond line)
  - composition (black diamond line)
  - association (single arrow)
  - public (+), private (-) and protected (#) specifiers.

Single licence - Abingdon School

2 Fundamentals of data structures

2.1 Data structures and abstract data types

Learning objectives:

- *Be familiar with the concept of data structures*

Figure 2.1.1.1 Queue Abstract Data Type

Key term

Composite data type:
In computer science, a composite data type or compound data type is any data type which can be constructed in a program using the programming language's primitive data types and other composite types.

Primitive data type:
In imperative languages, a primitive data type is one in which the set of values of the type are scalar values meaning "single" values e.g. integer, Boolean, character.

Key concept

Data structure:
A data structure is a named collections of variables, possibly of different data types, which are connected in various ways.
The collection consists of an aggregation of cells representing some abstract data type, e.g. a queue which models a real world entity, e.g. a supermarket queue.
The cell is the basic building block of data structures.

2.1.1 Data structures

The concept of data structures

Learning how to program is a process of developing the ability to model problems in such a way that a computer can solve them.

Array data structure

Suppose we needed to solve a problem which involved queueing. *Figure 2.1.1.1* shows an abstraction of a queue of people.

To represent this model in a computer programming language we use a data structure.

A data structure is a named collections of variables, possibly of different data types, which are connected in various ways. In the case of the queue, the person at the front of the queue is connected by relative position to the person next in the queue, this person is connected by relative position to the next and so on.

The cell is the basic building block of data structures. We can picture a cell as a box that is capable of holding a value drawn from some basic or composite data type. *Figure 2.1.1.2* shows a collection of or aggregation of cells representing a queue. This data structure is given a name, *Queue*, that refers to the aggregate of cells.

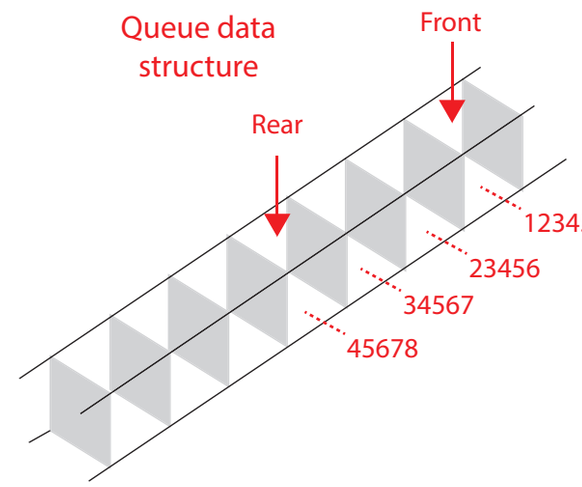


Figure 2.1.1.2 Queue data structure consisting of an aggregation of cells storing integers

Each cell of *Queue* is designed to store a value of an integer data type.

The simplest aggregating mechanism available in many programming languages is the array which is a sequence of cells of a given type.

Record data structure

Another common mechanism for grouping cells in programming languages is the record data structure. A record is a cell that is made up of a collection of other cells, called fields, that may be of different data types. The record data structure can model a customer record from the real world and store such values in its fields such as customer name, address, age, salary, job position.

Single licence - Abingdon School

File data structure

A third grouping method is the file. A file is a collection of data of some particular data type stored on mass storage, e.g. magnetic disk.
In a sequential file, elements of the file can be accessed only in the order of their appearance in the file. The number of elements in the file data structure and the size of the file can vary in time and this size is not constrained by the available RAM in the way that array data structures are constrained by main memory (RAM) capacity.

The structure of the file refers to the way that data stored in the file is subdivided or structured, e.g. as records.

The array and file data structures are covered in depth in *Chapter 2.1.2* and *Chapter 2.1.3*.

Questions

1. What is meant by a data structure?

2. Customer orders in a fish and chip shop are queued in the order that they arrive by an assistant at the counter using a till receipt as a record of an order. Other assistants are responsible for retrieving till receipts from the queue and fulfilling each order.
   Suggest a suitable a data structure to use to model this scenario by computer.

In this chapter you have covered:

- The concept of data structures

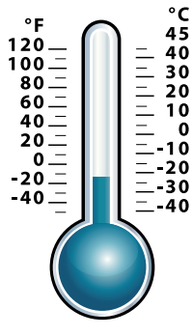
Single licence - Abingdon School

2 Fundamentals of data structures

2.1 Data structures and abstract data types

Learning objectives:

- *Use arrays (or equivalent) in the design of solutions to simple problems*



2.1.2 Single- and multi-dimensional arrays (or equivalent)

One-dimensional array

The problem

Suppose you are tasked with writing a computer program to keep a set of one hundred temperature readings in memory so that calculations may be performed on these readings in a uniform manner. A sample of the first 8 readings is shown in *Table 2.1.2.1*.

20.5	22.7	24.3	26.8	25.1	23.9	21.4	20.0

Table 2.1.2.1 Sample of temperature readings

We could store the first of the readings in a variable, `Temperature1`, the second in a variable, `Temperature2`, and so on as shown in *Table 2.1.2.2*. The data type chosen for `Temperature1` would be one that allows storage of a value possessing a fractional part, e.g. `float`. The same data type would be chosen for `Temperature2`, and each of the other 98 variables.

Variable	Value
`Temperature1`	20.5
`Temperature2`	22.7
`Temperature3`	24.3
`Temperature4`	26.8
`Temperature5`	25.1
`Temperature6`	23.9
`Temperature7`	21.4
`Temperature8`	20.0

Table 2.1.2.2 Variables for first 8 readings

Using 100 separate variables, each of which is only able to store a single value, is not ideal.

The solution

What is needed is a *single variable* which can store many values.

For this we use an **array** variable, `Temperature`, as shown in *Figure 2.1.2.1*.

Variable	Value
`Temperature`	20.5
	22.7
	24.3
	26.8
	25.1
	23.9
	21.4
	20.0
	•
	•
	•
	17.6

Figure 2.1.2.1 Array variable with storage for 100 temperature readings

1. An array is a data structure which is capable of storing many values, e.g. 100 temperature readings: 20.5, 22.7, ..., 17.6
2. An array is also a single entity, referred to by a single name, e.g. `Temperature`.
3. The values stored in an array are all of the same data type, e.g. `Float`.
4. The stored values are arranged in an order so that there is a first, a second, and so on.

Information

When we want to indicate that we have omitted numbers, etc, in a sequence of numbers or other entities we use a series of dots called an ellipsis • • • or •
•
•

The omission is used where it would be unnecessary to include more numbers because these can be inferred from the context.

Single licence - Abingdon School

Questions

1 A program is required to keep 10000 temperature readings in memory and to reference all the items in a uniform manner.
Explain why it would be preferable to use a one-dimensional array to store the value of each reading instead of 10000 separate variables.

Separate variables

In the separate variable approach, to extract, say, the value of the seventh temperature reading from among 100 separate variables, we need the name of the corresponding variable. We can construct this name by combining the name `Temperature` with the numeric name `7` to produce the name, `Temperature7`.

We then use this name to refer to the variable which contains the value of the seventh reading, i.e. 21.4 as shown in *Table 2.1.2.2*. Perhaps we might want to output the value of this variable to the console.
Pseudo-code that does this is as follows

```
Output Temperature7
```

Array index

Using an array to store 100 temperature readings, we retain the use of numeric naming 1, 2, 3, 4, ..., 100 and view the array as consisting of 100 elements with each element storing the value of a temperature reading.

To extract, say, the value of the seventh temperature reading, we construct a name by combining the name of the array, `Temperature`, with the numeric name `7` as follows `Temperature[7]` as shown in *Figure 2.1.2.2*.

We then use this name to refer to the **element** of the array which contains the seventh reading, i.e. 21.4 (see *Table 2.1.2.2*).

Variable	Index	Value
`Temperature`	`[1]`	20.5
	`[2]`	22.7
	`[3]`	24.3
	`[4]`	26.8
	`[5]`	25.1
	`[6]`	23.9
	`[7]`	21.4
	`[8]`	20.0
	⋮	⋮
	`[100]`	17.6

Figure 2.1.2.2 Use of an index in array variable Temperature

Perhaps we might want to output the value of this variable to the console. Pseudo-code that does this is as follows

```
Output Temperature[7]
```

Pseudo-code to output the first, second and third values of array `Temperature` is as follows

```
Output Temperature[1]
Output Temperature[2]
Output Temperature[3]
```

Information

Square bracket notation:
Square bracket notation [ ] is used to access values in an array.

The name that we use inside the square brackets is called the **array index**, e.g. 3 in `[3]`.

Pseudo-code to output all 100 values is as follows

```
For i ← 1 To 100 Output Temperature[i]
```

Here the array index is a loop control variable `i` which ranges from 1 to 100.

We are not restricted to starting array indexing at 1. We can index from 0. In fact, many programming languages only offer array indexing from 0. Pascal and Delphi are exceptions. In fact, Pascal and Delphi allow indexing from any starting value of ordinal data type.

Single licence - Abingdon School

Questions

2 A one-dimensional array `Height` consists of 10 elements. Array indexing starts at 0 and ends at 9. The array stores the heights of 10 people. Write pseudocode using a *for loop* to output all 10 heights in the same order that they are stored in the array.

Numeric names and non-numeric names for array index

We used numeric names for the index of an array, e.g. 1, 2, 3, etc, in the previous section. We don't have to. Instead, we can use non-numeric names, e.g. A, B, C, D, etc. For example, to extract the first value stored in array `LetterCount` we would use `LetterCount[A]`, the second value, `LetterCount[B]`, and so on.

Note, however, that whatever names we use, they must be capable of being ordered and the next name must be the natural successor to the previous, e.g. B is the successor to A, 2 is the successor to 1. This is very important.

If this wasn't the case then the *for loop* on the previous page, which assumes an increment of 1, wouldn't be able to access every one of the 100 values.

Choosing to use as an array index a range of numbers drawn from the set of integers is thus a natural choice to make. For other scenarios, the array index may be chosen from values in a different data type, e.g. character.

Questions

3 A one-dimensional array `LetterCount` consists of 26 elements. Array indexing starts at letter A and ends at letter Z. The array stores the count for each uppercase letter of the alphabet occurring in a piece of text, e.g. `LetterCount[D]` stores the value 36. Write pseudocode using a *for loop* to output the count stored for each letter in the same order that they are stored in the array. Use a loop control variable, `Letter`, which uses values chosen from A, B, C, ..., X, Y, Z.

What kind of identifier is an array index?

There are three kinds of non-negative integral number[1] usage:

Quantities: 0, 1, 2, ...

Quantities are used to give the quantity (a count) of items. They include zero, because zero is a possible quantity. Quantities are **cardinal numbers**.

Position words: 1st, 2nd, 3rd, ...

Position words give the position of an item within a sequence with a beginning. The item at the beginning of the sequence is called the "first" item, the next the "second", and so on. "First" is written as 1st, "second" as 2nd, and so on. Position words are **ordinal numbers**.

Identification numbers: 0, 1, 2, ..., or 1, 2, 3, ..., or any other ordered sequence.

Identification numbers are just a methodical way to give numerical names to items such as the sequence of 100 temperature values. Identification numbers are therefore a way of numbering the temperature values.

Information

Identification numbers are used extensively in the world. For example, all vehicles registered in the UK must have a unique, stamped-in vehicle identification number (VIN). It is usually stamped into the chassis of the vehicle.

1 Integral numbers are whole numbers such as the integers

Single licence - Abingdon School

Figure 2.1.2.3 shows array `Temperature` together with the identification numbers which identify each value stored in the array.

1	2	3	4	5	6	7	8	• • •	100
20.5	22.7	24.3	26.8	25.1	23.9	21.4	20.0	• • •	17.6

Figure 2.1.2.3 Use of identification numbers in range 1 to 100 to index array `Temperature`

Questions

4 What kind of identifier is the array index in the Temperature array?

Basic operations that access arrays

The two basic operations that access the one-dimensional array `Temperature` are

- Extraction: this is achieved by evaluating `Temperature[i]` to identify a particular stored value in the array `Temperature`, e.g. if `i` = 5, then `Temperature[i]` evaluates to 25.1 - see *Figure 2.1.2.3*.
- Storing: this is achieved by executing the assignment `Temperature[i]` ← `x` which results in a copy of the value stored in `x` being assigned to the array `Temperature` at the location within this array identified by identification number `i`. Variable `x` holds the temperature reading to be stored in the array `Temperature`, e.g. if `i` = 5 and `x` = 25.1, then `Temperature[5]` ← 25.1 results in 25.1 being stored in the array location with identification number 5.

Questions

5 Describe with examples the **two** basic operations used to access elements of a one-dimensional array.

Addresses

Numbers as names are also called **addresses**. We are familiar with numbers being assigned to houses: "Such and such lives at no 42".

We can therefore think of an array index as an address. Using an address also implies a location. For example, `Temperature[5]` is a **location** with address 5 within array `Temperature`.

Figure 2.1.2.4 shows storage bays for Russian doll objects. We can think of this structure as an array used for storing objects. Each bay within this array is a **location**. The structure is divided into cells, one above another. So we can also use the synonym **cell** for location.

Each location/cell is assigned an address. In this numbering scheme we have chosen to start the numbering from 0 and to label the lowermost cell 0. We could have started the numbering from 1 and we could have chosen the uppermost cell as the first cell. The choice is arbitrary. All that we need to ensure is that the numbering is ordered, i.e. 0, 1, 2 , 3 or 1, 2, 3, 4, and each cell is assigned, in order, a unique number drawn from the range of numbers whether 0, 1, 2, 3 is chosen or 1, 2, 3, 4.

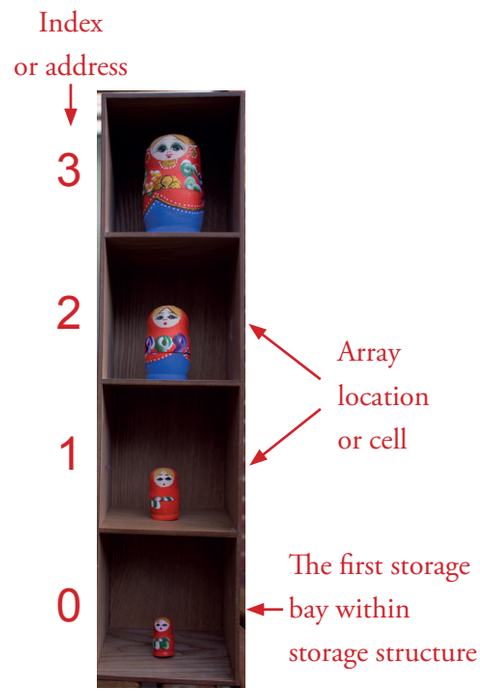


Figure 2.1.2.4 Location/cell view of an array with array index as a location address

Single licence - Abingdon School

We have now encountered three different terms for an array part that stores a value:

- element
- location
- cell

Questions

6 Explain why we can think of an array index as an address.

Why choose to start numbering array locations with the number zero?

Figure 2.1.2.5 shows array `Temperature` together with the identification numbers which identify each value stored in the array. This time the identification numbering/index starts at 0.

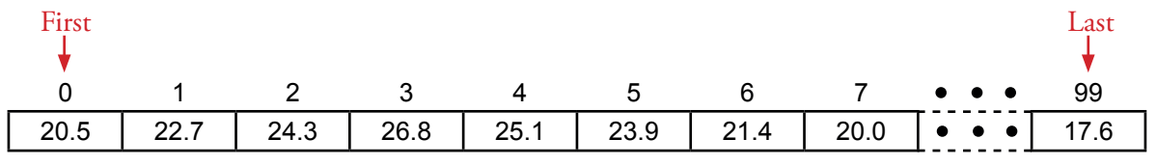


Figure 2.1.2.5 Use of identification numbers from 0 to 99 to index array `Temperature`

The meaning in English of the ordinal-number word "first" is the first entry within a sequence, even if this first entry is labelled with identification number 0, as unnatural as it may seem. We are conditioned by everyday experience to use identification number 1 when referring to the first entity in a sequence, e.g. the first house in a street has address 1 not 0. However, if the house number was 0, it would still be the first - see *Figure 2.1.2.6.*

Figure 2.1.2.6 House with house number 0

0 Acacia Avenue
Dingley Dell
NeverNeverLand
ZX99 9ZZ

Questions

7 Array indexing starts at 0 for an array `Row` with 10 elements.
(a) What is the index of the last element?
(b)Write pseudo-code to output the value stored in the first cell of this array.

8 (a) Explain why the following pseudo-code will output the 10 values stored in the array `Row`.

```
i ← 0
While i < 10
  Output Row[i]
  i ← i + 1
EndWhile
```

(b) What is the value stored in `i` when the loop terminates?

9 Rewrite the *While loop* pseudo-code in Question 8 so that it uses a *Repeat loop* with the terminating condition at the end of the loop.

Single licence - Abingdon School

Offset

By starting array indexing at 0, the index takes on the role of an **offset**. If the physical address in memory of the base of the array is, say, 1500 then the first cell of the array has address 1500 + 0, the second 1500 + 1, and so on.

Questions

10 An array `Vector` has 10 elements with an index that starts at 0. The physical address in memory of the base of this array is 2300.
What is the physical address of the fifth element?

11 A collection of ropes of various lengths are wrapped in turn around a cylinder of circumference 10 units. All the ropes wrap around the cylinder at least once but never exactly. The total length of each rope is measured and the measurement truncated so that it is integral. The integral measurement `L` is stored in a one-dimensional 10 element array `ExcessRopeLength` according to whether the integral excess length is 0, 1, 2, ..., 8, 9 units.
The integral excess length can be calculated using the formula

L mod 10

For example, if the length L is 36 units then the excess length is 6 units.
You are required to choose between using array indexing that starts at 0 and array indexing that starts at 1. Justify your choice.

A one-dimensional array is a useful way of representing a vector

Most data processing and numerical simulations are nothing more than a succession of elementary operations, such as *add* and *multiply,* applied to large amounts of numerical data. Computers are extremely good at this provided that the data is structured in a way that facilitates computation. The typical structure of numerical data is that of **vectors** and **matrices** (plural of matrix), and more generally **multi-dimensional arrays**.

A vector represents a grouping of data values, e.g. 0, 0, 100, 100.

The group of values may be stored in a one-dimensional array, `exampleVector`, expressed in some languages as

[0, 0, 100, 100]

The array, `exampleVector` could represent a Euclidean vector (sometimes called a geometric or spatial vector, or simply a vector).
A Euclidean vector is a geometric object that has magnitude (or length) and direction and can be added to other vectors according to vector algebra.
Figure 2.1.2.7 shows a Python program created in Processing 3.1 (processing.org). The program sets the window used for output to 200 x 200 pixels. It then creates a list [0, 199, 100, 100] which groups 4 values. The first pair, 0, 199, represents the start coordinate (bottom left in *Figure 2.1.2.8*) and the second pair, 100, 100, the end coordinate of a vector. A list is not an array but in this example it is being used in a way that is equivalent to an array.
Figure 2.1.2.8 shows the geometric vector, a line, displayed in the output window of the executing program.
Figure 2.1.2.9 shows a Java program created in Processing 3.1. The program sets the window used for output to 200 x 200 pixels. It then creates an array of four integer elements using `new int[4]`, before initialising this array with the group of values 0, 199, 100, 100 representing a vector. The first pair, 0, 199, represents the start coordinate and the second pair, 100, 100, the end coordinate of a vector. *Figure 2.1.2.10* shows the geometric vector, a line, displayed in the output window of the executing program.

Single licence - Abingdon School

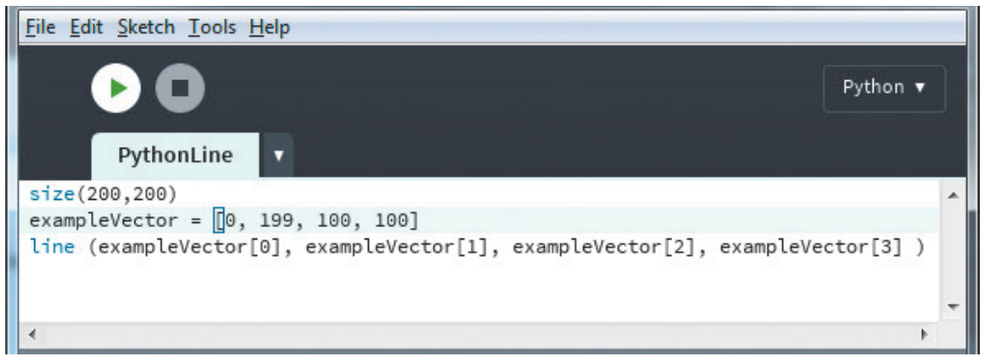


Figure 2.1.2.7 Python program in Processing 3.1 to draw a line for a given vector

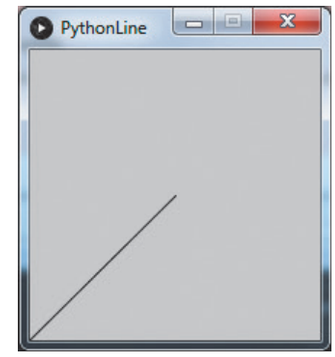


Figure 2.1.2.8 Processing 3.1 graphical output for Python program

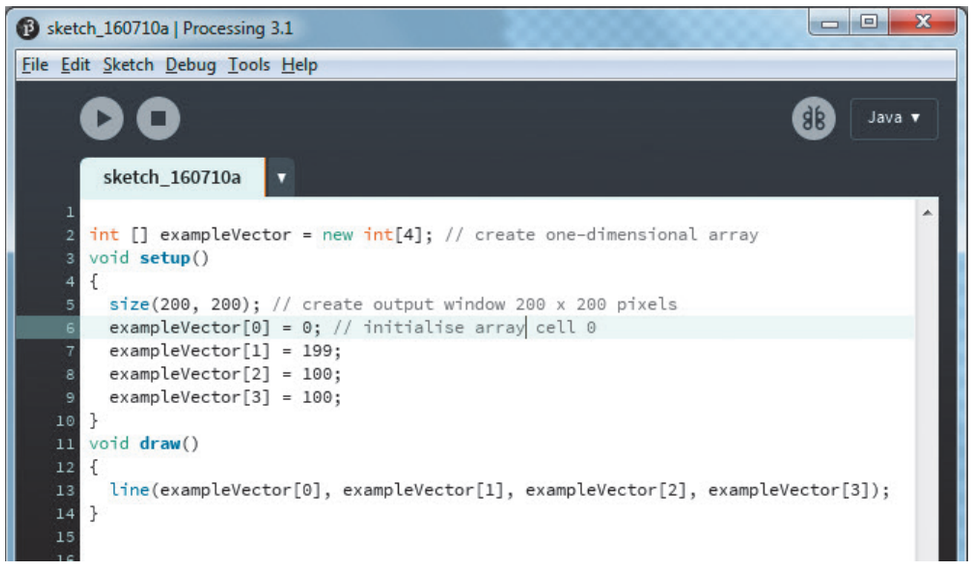


Figure 2.1.2.9 Java program in Processing 3.1 to draw a line from a given vector

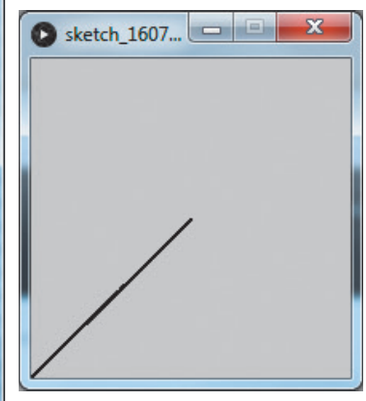


Figure 2.1.2.10 Processing 3.1 graphical output for Java program

Operations on vectors

It is useful to think of a vector in geometric terms. *Figure 2.1.2.11* shows two vectors realised as two lines in green. The vectors are expressed as two one-dimensional arrays as follows

[0, 0, 25, 75] and [0, 0, 100, 100]

If we add these two vectors element-wise we get a third vector expressed as a one-dimensional array as follows

[0, 0, 125, 75]

This vector is realised as the line in blue in *Figure 2.1.2.11*. Element-wise means that the first element of vector [0, 0, 25, 75] is added to the first element of vector [0, 0, 100, 100], and so on.

The red line drawn parallel to the shorter green line and started from the endpoint of the longer green line closes the triangle (of vectors). If we label this point A, then we have two ways to get to A: the first is along the long green line then along the red line; the second is along the blue line.

Figure 2.1.2.12 shows the Java program in Processing 3.1 which performs the vector addition and line drawing described above.

The term vector refers to a one-dimensional array of values to which operations such as add, subtract, multiple, divide can be applied to the array as a whole. The operations are applied in element-wise fashion so that the result is an array of the same size as the original array(s).

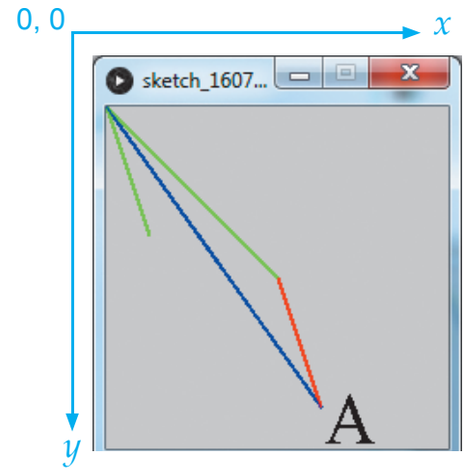


Figure 2.1.2.11 Processing 3.1 graphical output for Java program

Single licence - Abingdon School

The vector addition can be performed in parallel if a single ADD instruction can be applied to the four elements of each array simultaneously. Modern CPUs are capable of executing such operations by using **SIMD**[2] instructions, i.e. **S**ingle **I**nstruction **M**ultiple **D**ata stream instructions.

By creating and manipulating arrays, efficient vector computations can be performed with them.

The four elementary operations ADD, SUBTRACT, MULTIPLY, and DIVIDE can be applied to data in arrays provided they have compatible shapes, i.e. the same dimensions.

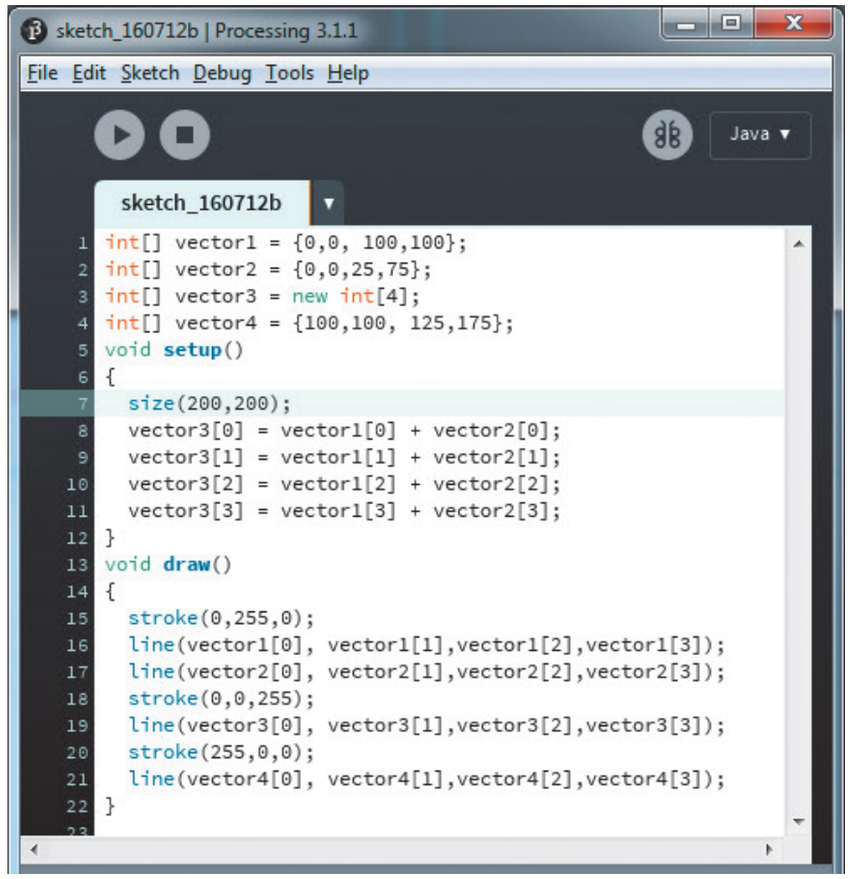


Figure 2.1.2.12 Java program in Processing 3.1 which performs vector addition

Questions

12 A one-dimensional array is a useful way of representing a vector. Use the multiplication of a one-dimensional array of integers, `ArrayOneD`, by **5** to explain what this means. Assume the array is initialised as follows `ArrayOneD = [3,6,7,2]`.

Creating a one-dimensional array

Table 2.1.2.3 shows examples of how integer and real/floating point one-dimensional arrays may be created in C#, Java, Pascal, Delphi, Python[3] and VB.NET. For Python two approaches are used. The first uses Python lists and the second arrays from the Numpy library.

Python lacks native support for creating an array data structure within the language. The closest that Python gets to a native data structure that can be used like a one-dimensional array is a list (a list is actually a dynamic array). A list is just a sequence of values (think of a shopping list).

However, it is possible to create an array in Python with the `NumPy` package if this package is loaded with the command `import numpy`.

`NumPy` is short for Numeric Python.

With `numpy` installed we can use the `array` function to create an array.

Information

Numpy:
NumPy is the fundamental package for scientific computing with Python.

Anaconda:
Installing some of the larger Python libraries, particularly those such as NumPy which depend on complex low-level C and Fortran packages, is made easy with Anaconda. Anaconda will do all the dependency checking and binary installs required.
Anaconda is free to download and install from
https://www.continuum.io/downloads

2 Not in the AQA specification
3 Spyder 2.3.1 the **S**cientific **PY**thon **D**evelopment **E**nvi**R**onment and Python 3.4

Single licence - Abingdon School

Language	Integer	Real/float
C#	`int[] vector = new int[4];` `int[] vector = {0,1,2,3};`	`float[] vector = new float[4];` `float[] vector = {0.0,1.5,2.3,3.4};`
Java	`int[] vector = new int[4];` `int[] vector = {0,1,2,3};`	`float[] vector= new float[4];` `float[] vector = {0.0,1.5,2.3,3.4};`
Pascal	`Vector : Array[0..3] Of Integer;` `Vector : Array[0..3] Of Integer = (0,1,2,3);`	`Vector : Array[0..3] Of Real;` `Vector : Array[0..3] Of Real = (0.0,1.5,2.3,3.4);`
Delphi	`Vector : Array[0..3] Of Integer;` `Vector : Array[0..3] Of Integer = (0,1,2,3);`	`Vector : Array[0..3] Of Real;` `Vector : Array[0..3] Of Real = (0.0,1.5,2.3,3.4);`

Python	`vector = [0,1,2,3,4] # list equivalent`
	`import numpy as np`
	`vector = np.array([0, 1, 2, 3])`
	`vector = np.empty([4],dtype=int)#empty array`
	`vector = np.zeros(4, dtype=int)`
		`vector = [0.0,1.5,2.3,3.4]`
		`import numpy as np`
		`vector = np.array([0, 1, 2, 3], dtype = float)`
		`vector = np.empty([4])#empty float array`
		`vector = np.zeros(4)`

VB.Net	`Dim vector(3) As Integer` `Dim vector = New Integer() {0,1,2,3}`	`Dim vector(3) As Double` `Dim vector = New Double() {0.0,1.5,2.3,3.4}`

Table 2.1.2.3 Creating one-dimensional arrays of different array element data type

In *Table 2.1.2.3*, the Python list [0, 1, 2, 3] is supplied as argument to the function `array` to create array `x` as follows

```
x = np.array([0,1, 2, 3])
```

In *Table 2.1.2.3*, the program code `x = np.zeros(4, dtype=int)` uses the `numpy` function `zeros` to create a four-element array with each element initialised to 0. The default array element data type is `float`.

In IPython (ipython.org), `%pylab` will install `numpy` as well as `matplotlib.pyplot`, a plotting library for displaying data visually - see *Table 2.1.2.5*.

IPython is an interactive python useful for trying out ideas.

Python lists are unsatisfactory as proxies for arrays for the following reasons:

1. Elements of a list can be of different data types, e.g. `[3, 4.5, 'hello', True, [34,56]]`
2. Lists may grow or shrink, e.g. `x = [0,1,2] x.append('a') x.pop()`
3. Processing of Python lists is considerably slower than processing arrays from Numpy

Tables 2.1.2.4, *2.1.2.5*, and *2.1.2.6* show examples of one-dimensional array creation in C#, Java, Pascal, Delphi, Python and VB.NET. The Python example uses IPython. IPython already has `numpy` in its namespace so use of the prefix `np` is unnecessary.

Language	Integer

Python 3.4
(IPython)
	`In [1]: %pylab`
	`In [2]: vector2 = array([0,1,2,3]), dtype=int)`
	`In [3]: vector4 = array([0.0,1.5,2.3,3.4], dtype=float)`
	`In [4]: print(vector2[0])`
	`In [5]: print(vector4[3])`

Python 3.4
(Scipy)
	`vector = [1,2,3,4] # list equivalent of an array`
	`print (vector[2])`
	`import numpy as np`
	`vector2 = np.array([1,2,3,4])`
	`print (vector2[2])`

Table 2.1.2.4 Creating one-dimensional arrays of different array element data type

Single licence - Abingdon School

Language	Integer

C#	`using System;`
	`namespace Arrays`
	`{`
	`  class Program`
	`  {`
	`  static void Main(string[] args)`
	`  {`
	`  int[] vector1 = new int[4];`
	`  int[] vector2 = { 0, 1, 2, 3 };`
	`  float[] vector3 = new float[4];`
	`  float[] vector4 = { 0.0f, 1.5f, 2.3f, 3.4f };`
	`  double[] vector5 =  { 0.0, 1.5, 2.3, 3.4 };`
	`  vector1[0] = 45;`
	`  vector3[0] = 45.8f;`
	`  Console.WriteLine(vector1[0]);`
	`  Console.ReadLine();`
	`  }`
	`  }`
	`}`

Java	`public class Arrays {`
	`  public static void main(String[] args) {`
	`  int[] vector1 = new int[4];`
	`  int[] vector2 = {0,1,2,3};`
	`  float[] vector3 = new float[4];`
	`  float[] vector4 = { 0.0f, 1.5f, 2.3f, 3.4f };`
	`  double[] vector5 = { 0.0, 1.5, 2.3, 3.4 };`
	`  vector1[0] = 45;`
	`  vector3[0] = 45.8f;`
	`  System.out.println(vector1[0]);`
	`  }`
	`}`

Delphi	`Program ArrayVector;`
	`{$APPTYPE CONSOLE}`
	`{$R *.res}`
	`Uses`
	`  System.SysUtils;`
	`Var`
	`  Vector1 : Array[0..3] Of Integer;`
	`  Vector2 : Array[0..3] Of Integer = (0,1,2,3);`
	`  Vector3 : Array[0..3] Of Real;`
	`  Vector4 : Array[0..3] Of Real = (0.0,1.5,2.3,3.4);`
	`Begin`
	`  Vector1[0] := 45;`
	`  Vector3[0] := 45.8;`
	`  Writeln(Vector1[0]);`
	`  Writeln(Vector4[3] :6:1);`
	`  Readln;`
	`End.`

Pascal	`Program ArrayVector;`
	`Var`
	`  Vector1 : Array[0..3] Of Integer;`
	`  Vector2 : Array[0..3] Of Integer = (0,1,2,3);`
	`  Vector3 : Array[0..3] Of Real;`
	`  Vector4 : Array[0..3] Of Real = (0.0,1.5,2.3,3.4);`
	`Begin`
	`  Vector1[0] := 45;`
	`  Vector3[0] := 45.8;`
	`  Writeln(Vector1[0]);`
	`  Writeln(Vector4[3] :6:1);`
	`  Readln;`
	`End.`

Table 2.1.2.5 Creating one-dimensional arrays of different array element data type

Single licence - Abingdon School

Language	Integer

VB.NET	`Module Arrays`
	`  Sub Main()`
	`  Dim vector1(3) As Integer`
	`  Dim vector2 = {0, 1, 2, 3}`
	`  Dim vector3(3) As Single`
	`  Dim vector4(3) As Double`
	`  Dim vector5 = {0.0, 1.5, 2.3, 3.4}`
	`  vector1(0) = 45`
	`  vector3(0) = 45.8F`
	`  vector4(3) = 45.8`
	`  Console.WriteLine(vector1(0))`
	`  Console.WriteLine(vector5(3))`
	`  Console.ReadLine()`
	`  End Sub`
	`End Module`

Table 2.1.2.6 Creating one-dimensional arrays of different array element data type

Iterating through the elements of a one-dimensional array

Iterating through the elements of an array is a common operation on arrays. *Figure 2.1.2.13* shows the use of a *For loop* to access each element of array `Vector` starting with the first element `Vector[0]`. The index used is `i`, the loop control variable. *Table 2.1.2.7* shows the value of `i` and the value of array `Vector[i]` in each iteration of the *For loop*.

The one-dimensional array `Vector` contains 4 elements with the following values 0.0, 1.5, 2.3, 3.4, respectively.

```
For i ← 0 To 3
  Output Vector[i]
EndFor
```

Figure 2.1.2.13 Pseudo-code For loop

Table 2.1.2.8 shows the loop pseudo-code expressed in the programming languages C#, Java, Delphi/Pascal, Python and VB.NET.

Value of i	Element accessed	Value of element
0	`Vector[0]`	0.0
1	`Vector[1]`	1.5
2	`Vector[2]`	2.3
3	`Vector[3]`	3.4

Table 2.1.2.7 Value of loop control variable and element of array Vector on each iteration

Language	Integer

C#	`int i;`
	`for (i = 0; i < 4; i++)`
	`  {`
	`  Console.WriteLine(vector[i]);`
	`  }`

Java	`int i;`
	`for (i = 0; i < 4; i++)`
	`  {`
	`  System.out.println(vector[i]);`
	`  }`

Delphi/ Pascal	`Var i : Integer;`
	`.....`
	`For i := 0 To 3`
	`  Do Writeln(Vector[i]);`

Python	`for i in range(4):` `  print(vector[i])`

VB.NET	`Dim i As Integer`
	`For i = 0 To 3`
	`  Console.WriteLine(vector(i))`
	`Next`

Table 2.1.2.8 Accessing the elements of a one-dimensional array by iteration with a For loop

Programming task

1. Write a program which iterates through ArrayOneD outputting each value in turn starting at the first value contained in this array. Assume the array is initialised as follows `ArrayOneD = [3,6,7,2]`.

Single licence - Abingdon School

Computing a single value from the contents of a one-dimensional array

To sum all the elements of a one-dimensional array, `Vector`, we must access each element of this array in turn adding its value to a running total `Sum`. The variable `Sum` is initialised to contain zero. *Figure 2.1.2.14* shows pseudo-code that computes the sum of the elements of array `Vector`. Array `Vector` contains 4 elements with the following values 1, 3, 8, 9.

Table 2.1.2.9 shows the state of `Sum` before iteration begins and at the end of each iteration, e.g. after the first iteration variable `Sum` contains 1, after the second, 5.

```
Sum ← 0
For i ← 0 To 3
  Sum ← Sum + Vector[i]
EndFor
Output Sum
```

Figure 2.1.2.14 For loop

Value of i	Element accessed	Value of element	Sum
			0
0	`Vector[0]`	1	1
1	`Vector[1]`	3	4
2	`Vector[2]`	8	12
3	`Vector[3]`	9	21

Table 2.1.2.9 Summing the elements of a one-dimensional array

Programming tasks

2. Write a program which iterates through ArrayOneD and sums the values contained in this array. Assume the array is initialised as follows `ArrayOneD = [3,6,7,2]`.
3. Write a program which collects 6 integer values from the keyboard and stores them in an array called `IntegerArray`. The program is to display the contents of each cell of the array in turn, starting at the first cell, after all the values have been entered.
4. Using the same program structure, add code which calculates the average of the first three numbers stored in the array and the second three numbers.
5. Using the same program structure, add another array to the program of the same size and type as the first. Now add code to copy the contents of the first array to the second array so that the second array holds the contents of the first array in reverse order.

When to use a one-dimensional array?

When it is necessary to keep a large number of items in memory and reference all the items in a uniform manner.

Multi-dimensional arrays

Two-dimensional arrays

We have so far considered how data may be stored in a one-dimensional array. A one-dimensional array has, of course, one dimension. We can visualise the items of data in a one-dimensional array arranged along a single axis as shown by the example in *Figure 2.1.2.15*.

Figure 2.1.2.15 Visualising items of numerical data in a one-dimensional array arranged along a single axis

Single licence - Abingdon School

If we have two dimensions for storing data then we can visualise the data stored in a grid-like fashion defined by two axes as shown in *Figure 2.1.2.16*.

The **structure** of 5 rows and 5 columns of data shown in *Figure 2.1.2.16* is called a **two-dimensional array** or **matrix.** The matrix contains 5 x 5 = 25 numbers.

In order to refer to a specific number, we need to specify both the **row** and **column** because the matrix has two dimensions. For example, the third value along in the second row in *Figure 2.1.2.16* is the number 27. This number has been marked by a red square in the figure.

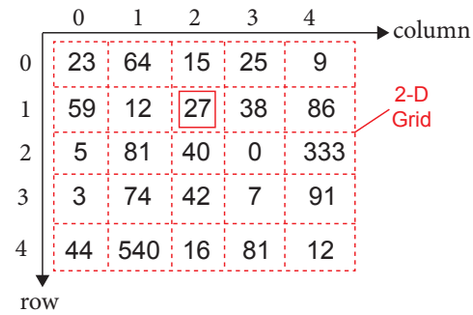


Figure 2.1.2.16 Visualising items of numerical data stored in a two-dimensional array laid out in two dimensions

The axes in *Figure 2.1.2.16* have been labelled, `row` and `column` so that the first row has value 0 and the first column has value 0. Therefore, the `row` and `column` values of the number 27 marked by a red square are 1 and 2, respectively.

Loading a greyscale image into a two-dimensional array

A greyscale image can be loaded into a two-dimensional array of the same dimensions, e.g. if the image is 450 x 600 then its height is 450 and its width is 600 - see *Figure 2.1.2.17*.

A value representing the intensity of a pixel at a particular position in the image is stored in the array cell of the same row and column as the image pixel..

The Python 3.4 program shown in *Figure 2.1.2.17* loads a single channel greyscale image file, `PlaneBWSingleChannel.png`, into a two dimensional array `im` using the command

```
im = imread('C:\\Users\\drbond\\
PlaneBWSingleChannel.png')
```

(\\ is used and not \ because \ is used in Python strings as an escape character).

The values stored in array `im` are rendered by `imshow` to display in shades of grey the 450 x 600 greyscale image shown in *Figure 2.1.2.17*.

The size and shape of the array `im` is shown with the command

```
im.shape
```

Figure 2.1.2.17 shows that the array `im` has the dimensions `450` x `600`. This is a two-dimensional array.

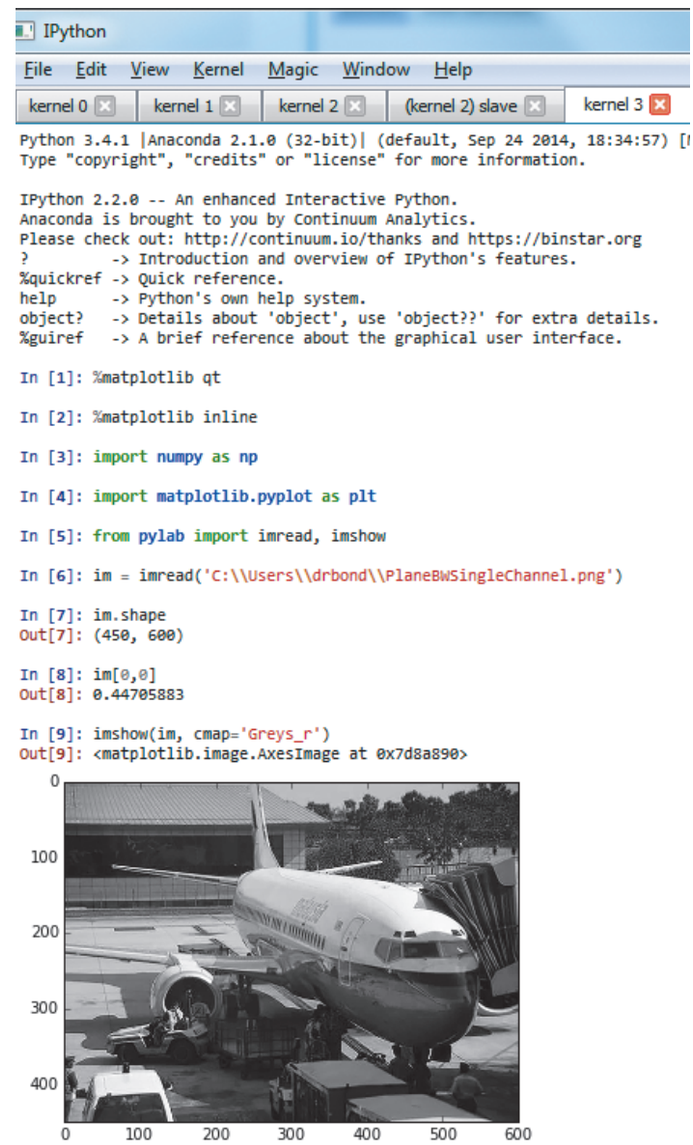


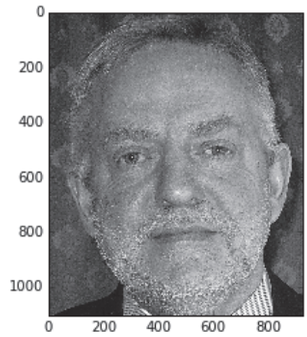
Figure 2.1.2.17 Using Python 3.4. and library routines to load a greyscale image into a two-dimensional array, im

Information

Download: www.educational-computing.co.uk/Unit2/Images/Images2.rar to obtain PlaneBWSingleChannel.png

Single licence - Abingdon School

```
In [38]: matrix1 = np.ones([1104, 927])

In [39]: matrix2 = imread('C:/Users/drbond/KRB.png')

In [40]: matrix3 = matrix1 - matrix2

In [41]: matrix1.shape
Out[41]: (1104, 927)

In [42]: matrix2.shape
Out[42]: (1104, 927)

In [43]: imshow(matrix2, cmap='Greys_r')
Out[43]: <matplotlib.image.AxesImage at 0x78f5950>
```




```
In [44]: imshow(matrix3, cmap='Greys_r')
Out[44]: <matplotlib.image.AxesImage at 0x792a730>
```

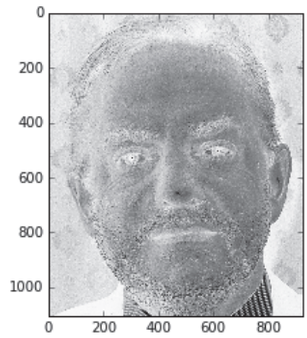


Figure 2.1.2.19 Subtracting an image matrix from a ones matrix to invert the image

The value stored in this array at `0, 0` is revealed with the command `im[0, 0]`. The `imread` function normalises each pixel intensity read from the image file to a value between 0 and 1.

The procedure `imshow` converts the array values to numbers in the range 0 to 255 when it displays an image. The library `matplotlib` handles the plotting of the image and the `inline` directive causes the image to be displayed in the IPython window in line with the Python code. `imshow(im, cmap='Greys_r')` renders the values in array `im` using a mapping which produces a greyscale image display.

Matrix algebra

We reserve the term two-dimensional array for a data structure which stores data in rows and columns in two dimensions.

The term matrix refers to a two-dimensional array of values to which operations such as add, subtract, multiple, divide can be applied to the array as a whole. For example, suppose that we have a 5 x 4 array `matrix1` consisting of all ones and another 5 x 4 array `matrix2` consisting of values between 0 and 1 as shown in *Figure 2.1.2.18* and we subtract `matrix2` from `matrix1`. We get `matrix3` as shown in *Figure 2.1.2.18*.

In effect we have done `matrix3 = matrix1 - matrix2`.

Figure 2.1.2.19 shows the result of subtracting an image matrix from a ones matrix: the image is inverted.

Two image matrices (plural of matrix), `image1` and `image2`, are shown in *Figure 2.1.2.20* and *Figure 2.1.2.21*. We can add these in different proportions so that we can fade one into the other as shown in *Figures 2.1.2.22, 2.1.2.23, 2.1.2.24, 2.1.2.25*.

matrix1[1,3] − matrix2[1,3] = matrix3[1,3]

$$\begin{bmatrix} 1 & 1 & \boxed{1} & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 0.5 & 0.3 & \boxed{0.9} & 0.75 \\ 0.0 & 0.1 & 0.15 & 0.4 \\ 0.8 & 0.6 & 0.05 & 0.23 \\ 0.2 & 0.6 & 0.5 & 0.3 \\ 0.94 & 0.86 & 0.3 & 0.7 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.7 & \boxed{0.1} & 0.25 \\ 1.0 & 0.9 & 0.85 & 0.6 \\ 0.2 & 0.4 & 0.95 & 0.77 \\ 0.8 & 0.4 & 0.5 & 0.7 \\ 0.06 & 0.14 & 0.7 & 0.3 \end{bmatrix}$$

matrix1 matrix2 matrix3

Figure 2.1.2.18 Subtracting one matrix from another

Questions

13 A two-dimensional array is a useful way of representing a matrix. Use the multiplication of a two-dimensional array of integers, `ArrayTwoD`, by 5 to explain what this means. Assume the array is initialised as follows `ArrayTwoD` $= \begin{bmatrix} 3,6,7 \\ 4,8,1 \\ 9,3,8 \end{bmatrix}$

Information

Download: www.educational-computing.co.uk/Unit2/Images/Images2.rar to obtain images used above.

Single licence - Abingdon School

```
In [6]: image1 = imread('C:/Users/drbond/Poh2017P1Cropped.png')

In [7]: image1.shape
Out[7]: (1104, 927)

In [8]: imshow(image1, cmap='Greys_r')
Out[8]: <matplotlib.image.AxesImage at 0x73bc930>
```

1104 x 927 pixels

1104 x 927 array

=1104 rows &

927 columns

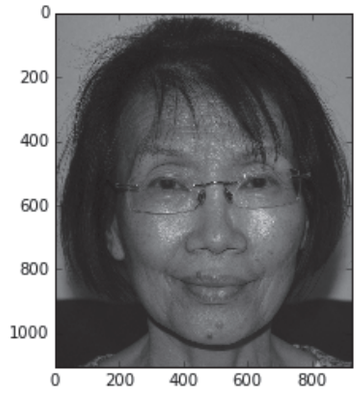


Figure 2.1.2.20 Loading array image1 with a greyscale image

```
In [8]: image2 = imread('C:/Users/drbond/KRB.png')

In [9]: imshow(image2, cmap='Greys_r')
Out[9]: <matplotlib.image.AxesImage at 0x545abf0>
```

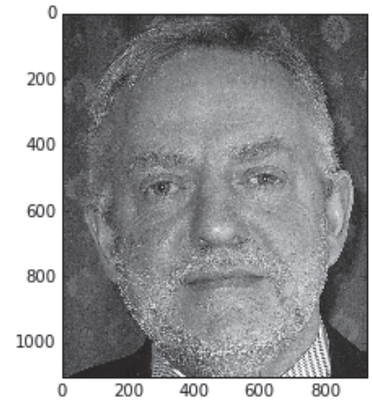


Figure 2.1.2.21 Loading array image2 with another greyscale image

```
In [16]: image = 0.7*image1 + 0.3*image2

In [17]: imshow(image, cmap='Greys_r')
Out[17]: <matplotlib.image.AxesImage at 0x557b350>
```

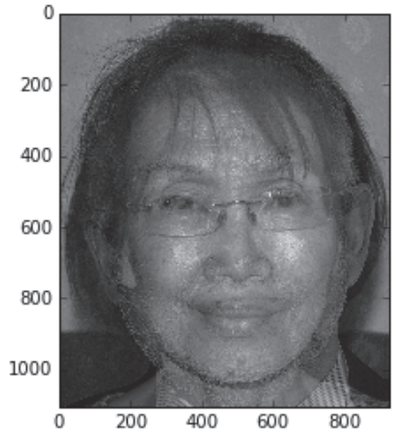


*Figure 2.1.2.22 0.7*image1 + 0.3*image2*

```
In [10]: image = 0.5*image1 + 0.5*image2

In [11]: imshow(image, cmap='Greys_r')
Out[11]: <matplotlib.image.AxesImage at 0x549a430>
```

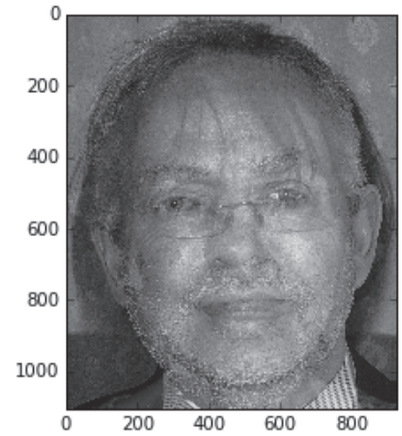


*Figure 2.1.2.23 0.5*image1 + 0.5*image2*

```
In [18]: image = 0.3*image1 + 0.7*image2

In [19]: imshow(image, cmap='Greys_r')
Out[19]: <matplotlib.image.AxesImage at 0x55b30f0>
```

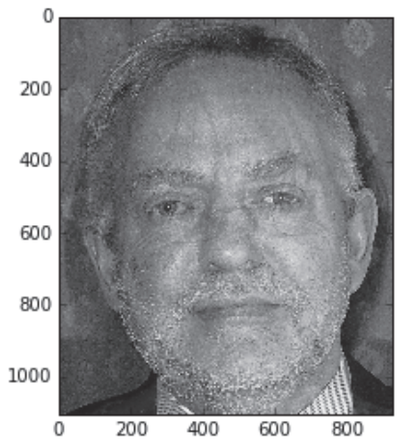


*Figure 2.1.2.24 0.3*image1 + 0.7*image2*

```
In [45]: image = 0.0*image1 + 1.0*image2

In [46]: imshow(image, cmap='Greys_r')
Out[46]: <matplotlib.image.AxesImage at 0x79634d0>
```

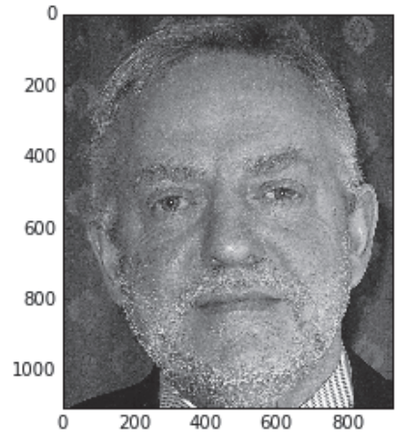


*Figure 2.1.2.25 0.0*image1 + 1.0*image2*

Single licence - Abingdon School

Creating two-dimensional arrays in some programming languages

Table 2.1.2.10 shows the creation of two-dimensional arrays in C#, Java, Pascal, Delphi, Python 3.4 and VB.NET.

Lang-uage	Integer	Real/float
C#	`int[,] array2D = new int[4,2];` `int[,] array2D = {{1,2},{3,4},{5,6},{7,8}};`	`float[,] array2D = new float[4,2];`
Java	`int[][] array2D = new int[4][2];` `int[][] array2D = {{1,2},{3,4},{5,6},{7,8}};`	`float[][] array2D=new float[4][2];`
Pascal	`Array2D : Array[0..3,0..1] Of Integer;` `Array2D : Array[0..3,0..1] Of Integer = ((1,2),(3,4),(5,6),(7,8));`	`Array2D: Array[0..3,0..1] Of Real;`
Delphi	`Array2D : Array[0..3,0..1] Of Integer;` `Array2D : Array[0..3,0..1] Of Integer = ((1,2),(3,4),(5,6),(7,8));`	`Array2D : Array[0..3,0..1] Of Real;`

Python	`array2D = []`
	`for row in range(4):`
	`  array2D.append([])`
	`  for column in range(2):`
	`  array2D[row].append(0)#creating a 4 x 2 list equivalent`
	`import numpy as np`
	`array2D = np.array([[1,2], [3,4], [5,6], [7,8]])`
	`array2D = np.empty([4,2],dtype=int)# creates a 4 x 2 empty array`
		`array2D = []`
		`for row in range(4):`
		`  array2D.append([])`
		`  for column in range(2):`
		`  array2D[row].append(0.0)`
		
		`import numpy as np`
		`array2D =`
		`np.empty([4,2],dtype=float)`

VB. NET	`Dim array2D(3, 1) As Integer` `Dim array2D = New Integer(3, 1) {{1, 2}, {3, 4}, {5, 6}, {7, 8}}`	`Dim array2D(3,1) As Double`

Table 2.1.2.10 Creating two-dimensional arrays of different array element data type

C#

The following creates a two-dimensional integer array of 4 rows and 2 columns

```
int[,] array2D = new int[4,2];
```

The following creates and initialises a two-dimensional integer array of 4 rows and 2 columns

```
int[,] array2D = new int[4,2] {{1,2},{3,4},{5,6},{7,8}};
```

If the array is being initialised, `new int[4,2]` can be omitted. Array indexing is zero-based.

Java

The following creates a two-dimensional integer array of 4 rows and 2 columns

```
int[][] array2D = new int[4][2];
```

The following creates and initialises a two-dimensional integer array of 4 rows and 2 columns

```
int[][] array2D = {{1,2},{3,4},{5,6},{7,8}};
```

Array indexing is zero-based.

Pascal/Delphi

The following creates a two-dimensional integer array of 4 rows and 2 columns

```
Array2D : Array[0..3,0..1] Of Integer;
```

The following creates and initialises a two-dimensional integer array of 4 rows and 2 columns

```
Array2D : Array[0..3, 0..1] Of Integer = ((1,2),(3,4),(5,6),(7,8));
```

Array indexing is more flexible in Pascal and Delphi. It may begin at any value, e.g. -1, 1. Array is a keyword in the Pascal and Delphi.

Single licence - Abingdon School

Python

The following creates an equivalent of a two-dimensional integer array of 4 rows and 2 columns by using lists within a list and initialises it with zeros

```
array2D = []
for row in range(4):
    array2D.append([])
    for column in range(2):
      array2D[row].append(0)
```

List indexing is zero-based.

The following creates an empty two-dimensional integer array of 4 rows and 2 columns by using the numpy library

```
import numpy as np
array2D = np.empty([4,2],dtype=int)
```

The following creates and initialises a two-dimensional integer array of 4 rows and 2 columns

```
array2D = np.array([[1,2], [3,4], [5,6], [7,8]])
```

VB.NET

The following creates a two-dimensional integer array of 4 rows and 2 columns

```
Dim array2D(3, 1) As Integer
```

Array indexing is zero-based. 3 and 1 are the upper bound index values.

The following creates and initialises a two-dimensional integer array of 4 rows and 2 columns

```
Dim array2D = New Integer(3, 1) {{1, 2}, {3, 4}, {5, 6}, {7, 8}}
```

Using two-dimensional arrays

Tables 2.1.2.11, 2.1.2.12 and *2.1.2.13* show the creation and use of two-dimensional arrays in C#, Java, Pascal, Delphi, Python 3.4 and VB.NET.

Language	Integer

C#	`using System;`
	`namespace TwoDimensionalArrays`
	`{`
	`  class Program`
	`  {`
	`  static void Main(string[] args)`
	`  {`
	`  int[,] array2D = new int[4, 2];`
	`  int[,] array2D1 = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};`
	`  Console.WriteLine(array2D1[1,1]);`
	`  array2D[0,0] = 1;`
	`  Console.ReadLine();`
	`  }`
	`  }`
	`}`

Java	`public class TwoDArray {`
	`  public static void main(String[] args) {`
	`  float[][] array2D = new float[4][2];`
	`  int[][] array2D1 = {{1,2},{3,4},{5,6},{7,8}};`
	`  System.out.println(array2D1[1][1]);`
	`  array2D[0][0] = 1;`
	`  System.out.println(array2D[0][0]);`
	`  }`
	`}`

Table 2.1.2.11 Creating and using two-dimensional arrays

Single licence - Abingdon School

Language	Integer

Delphi	`Program TwoDimensionalArrays;`
	`{$APPTYPE CONSOLE}`
	`{$R *.res}`
	`Uses`
	`  System.SysUtils;`
	`Var`
	`  Array2D : Array[0..3,0..1] Of Integer;`
	`  Array2D1 : Array[0..3,0..1] Of Integer = ((1,2),(3,4),(5,6),(7,8));`
	`Begin`
	`  Array2D[0,0] := 1;`
	`  Writeln(Array2D[0,0]);`
	`  Writeln(Array2D1[1,1]);`
	`  Readln;`
	`End.`

Pascal
(Lazarus)
	`Program TwoDimensionalArrays;`
	`Var`
	`  Array2D : Array[0..3,0..1] Of Integer;`
	`  Array2D1 : Array[0..3,0..1] Of Integer = ((1,2),(3,4),(5,6),(7,8));`
	`Begin`
	`  Array2D[0,0] := 1;`
	`  Writeln(Array2D[0,0]);`
	`  Writeln(Array2D1[1,1]);`
	`  Readln;`
	`End.`

Python
3.4
(Scipy)
	`import numpy as np`
	`array2D = np.array([[1,2], [3,4], [5,6], [7,8]])`
	`for row in range(4):`
	`  for column in range(2):`
	`  print( array2D[row,column], end= '')`
	`  print(end='\n')`
	`array2D1 = np.empty([4,2],dtype=int)`
	`array2D1[0,0] = 1`
	`array2D1[0,1] = 2`
	`array2D1[1,0] = 3`
	`array2D1[1,1] = 4`
	`array2D1[2,0] = 5`
	`array2D1[2,1] = 6`
	`array2D1[3,0] = 7`
	`array2D1[3,1] = 8`
	`for row in range(4):`
	`  for column in range(2):`
	`  print( array2D1[row,column], end= '')`
	`  print(end='\n')`

Table 2.1.2.12 Creating and using two-dimensional arrays

Single licence - Abingdon School

Language	Integer

VB.NET	`Dim array2D(3, 1) As Integer`
	`Dim array2D1 = New Integer(3, 1) {{1, 2}, {3, 4}, {5, 6}, {7, 8}}`
	`Dim array2D2(3, 1) As Double`
	`array2D(0, 0) = 1`
	`array2D(0, 1) = 2`
	`array2D(1, 0) = 3`
	`array2D(1, 1) = 4`
	`array2D2(0, 0) = 1.5`
	`array2D2(0, 1) = 2.3`
	`array2D2(1, 0) = 3.6`
	`array2D2(1, 1) = 4.8`
	`Console.WriteLine(array2D(1, 0))`
	`Console.WriteLine(array2D1(1, 0))`
	`Console.WriteLine(array2D2(1, 0))`

Table 2.1.2.13 Creating and using two-dimensional arrays

Programming task

6 Write a program which creates a two-dimensional array `ArrayTwoD` and displays the values contained in the second row of this array. The array `ArrayTwoD` should be created with values arranged into rows and columns as follows

$$\texttt{ArrayTwoD} = \begin{bmatrix} 3,6,7 \\ 4,8,1 \\ 9,3,8 \end{bmatrix}$$

Iterating through the elements of a two-dimensional array

Iterating through the elements of an array is a common operation on arrays. *Figure 2.1.2.26* shows the use of nested *For loops* to access each element of array `Array2D` starting with the first element `Array2D[0,0]`. The outer *For loop* uses the loop control variable, `i`. The inner *For loop* uses the loop control variable, `j`.

The two-dimensional array `Array2D` contains 3 x 3 = 9 elements with values as shown in *Table 2.1.2.14*.

```
For i ← 0 To 2
  For j ← 0 To 2
    Output Array2D[i,j]
  EndFor
EndFor
```

Figure 2.1.2.26 Pseudo-code that iterates through the elements of array Array2D

	0	1	2
0	12	45	6
1	9	23	65
2	2	18	33

Table 2.1.2.14 Array2D

The **trace table**, *Table 2.1.2.15*, shows the value of array `Array2D[i,j]` and the output for each value of `i` and `j`. A trace table is a table which shows the current values of variables used by the pseudo-code and its output whilst the pseudo-code is executed (traced) by hand. This execution by hand is called a **hand-trace**.

i	j	Array2D[i,j]	Output
0	0	12	12
0	1	45	45
0	2	6	6
1	0	9	9
1	1	23	23
1	2	65	65
2	0	2	2
2	1	18	18
2	2	33	33

Table 2.1.2.15 Trace table for pseudo-code

Single licence - Abingdon School

Table 2.1.2.16 shows the nested loop pseudo-code expressed in the programming languages C#, Java, Delphi/Pascal, Python and VB.NET.

Language	Integer

C#	`int[,] array2D = new int[3, 3] {{12,45,6},{9,23,65},{2,18,33}};`
	`int i, j;`
	`for (i = 0; i < 3; i++)`
	`{`
	`for (j = 0; j < 3; j++)`
	`{`
	`Console.Write(array2D[i,j]);`
	`Console.Write(" ");`
	`}`
	`Console.WriteLine();`
	`}`
	`Console.ReadLine();`

Java	`int [][] array2D = {{12,45,6},{9,23,65},{2,18,33}};`
	`int i, j;`
	`for (i = 0; i < 3; i++)`
	`{`
	`for (j = 0; j < 3; j++)`
	`{`
	`System.out.print(array2D[i][j]);`
	`System.out.print(' ');`
	`}`
	`System.out.println();`
	`}`

Delphi/ Pascal	`Var`
	`i, j : Integer;`
	`Array2D : Array [0..2, 0..2] Of Integer = ((12,45,6),(9,23,65),(2,18,33));`
	`.....`
	`For i := 0 To 2`
	`Do`
	`Begin`
	`For j := 0 To 2`
	`Do Write(Array2D[i, j], ' ');`
	`Writeln;`
	`End;`
	`Readln;`

Python	`import numpy as np`
	`array2D = np.array([[12,45,6], [9,23,65], [2,18,33])`
	`for i in range(3):`
	`for j in range(3):`
	`print( array2D[row,column], end= ' ')`
	`print(end='\n')`

VB.NET	`Dim array2D = New Integer(2, 2) {{12, 45, 6}, {9, 23, 65}, {2, 18, 33}}`
	`For i = 0 To 2`
	`For j As Integer = 0 To 2`
	`Console.Write(array2D(i, j))`
	`Console.Write(" ")`
	`Next`
	`Console.WriteLine()`
	`Next`
	`Console.ReadLine()`

Table 2.1.2.16 Accessing the elements of a two-dimensional array by iteration with a nested For loop

Programming task

7 Write a program which iterates through `ArrayTwoD` and sums the values contained in this array. The array `ArrayTwoD` should be created with values arranged into rows and columns as follows

$$\texttt{ArrayTwoD} = \begin{bmatrix} 3,6,7 \\ 4,8,1 \\ 9,3,8 \end{bmatrix}$$

Single licence - Abingdon School

Programming tasks

8 Write a program which iterates through `ArrayTwoD` multiplying each value by 5. The result of each multiplication should replace the original value.
The array `ArrayTwoD` should be created with values arranged into rows and columns as follows

$$\texttt{ArrayTwoD} = \begin{bmatrix} 3,6,7 \\ 4,8,1 \\ 9,3,8 \end{bmatrix}$$

A particular black and white image contains a single simple convex shape, such as a circle, represented as a dark area on a light background. Such images are commonly represented in computers as two-dimensional arrays, where each element has the value 1 or 0. A dark area on a light background becomes a group of 1's surrounded by 0's. *Figure 2.1.2.27* shows a two-dimensional array representation of a lower resolution image of a black circle on a white background.

```
0,0,0,0,0,0,0
0,0,0,1,0,0,0
0,0,1,1,1,0,0
0,1,1,1,1,1,0
0,0,1,1,1,0,0
0,0,0,1,0,0,0
0,0,0,0,0,0,0
```

Figure 2.1.2.27

9 Write a program with a two-dimensional array initialised with the values shown in *Figure 2.1.2.27*.
The program should display the shape of the dark region by displaying the points of the image which represent dark regions as 1's and the background as spaces.
Hint: Scan the image row-wise or column-wise.

10 Write a program with a two-dimensional array initialised with the values shown in *Figure 2.1.2.27*.
The program should display the shape of the dark region by displaying the points in the image on the boundary between the dark region and the surrounding white region as 1's and the background and interior points as spaces.
Hint: Scan the image row-wise or column-wise. For a given 1, if **any** of the surrounding points are 0's then that 1 represents a boundary point.

11 Write a program with a two-dimensional array initialised with the values shown in *Figure 2.1.2.27*.
The program should calculate the ratio of the area of the dark region to its perimeter. As a measure of the area, count the number of ones. A (rather inaccurate) measure of the perimeter can be obtained by counting the number of boundary points.

When to use a two-dimensional array?

When it is necessary to keep a large number of items in memory and reference all the items in a uniform manner.

When the structure of the data is two-dimensional.

N-dimensional arrays

If we add a third axis at right-angles to the two axes in *Figure 2.1.2.16* we obtain a three dimensional storage grid, i.e. a three-dimensional array.
In Pascal, a three-dimensional array variable, `Array3D`, is created as follows

```
Array3D : Array[0..3,0..4,0..3] Of Integer;
```


Single licence - Abingdon School

We can store the pixel intensity of each pixel of an RGB image in a three-dimensional array as illustrated by the schematic in *Figure 2.1.2.28*.

Figure 2.1.2.29 shows an RGB image 'plane.png' loaded into a three-dimensional array `im1`. This array has 450 rows, 600 columns and in the third dimension, 3 colour planes, one for the red intensity, one for the blue intensity and one for the green intensity of a pixel.

We can continue to add dimensions but we will not be able to visualise the array geometrically because we live in a three-dimensional world. It is quite rare, however, to work on arrays with more than four dimensions.

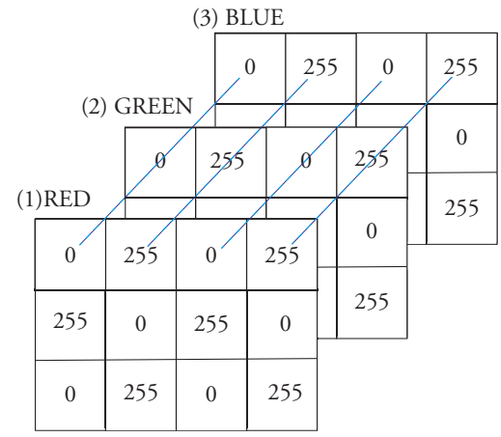


Figure 2.1.2.28 Colour planes of an RGB image, each plane is a two-dimensional array

A n-dimensional array is a multi-dimensional array of n dimensions where n could be, say 4.

All the elements of the n-dimensional array are of the same data type which is why it is an array. The index consists of a tuple of *n* integers in a zero-based indexed array, e.g. in C#, a four-dimensional array, `array4D`, could be declared as follows

```
int[,,,] array4D = new int[4,4,4,4];
```

The index part of the four-dimensional array declaration is the tuple of 4 integers 4, 4, 4, 4.

```
In [5]: im1 = imread('C:/Users/drbond/Plane.png')

In [6]: im1.shape
Out[6]: (450, 600, 3)

In [7]: imshow(im1)
Out[7]: <matplotlib.image.AxesImage at 0x4c64b70>
```

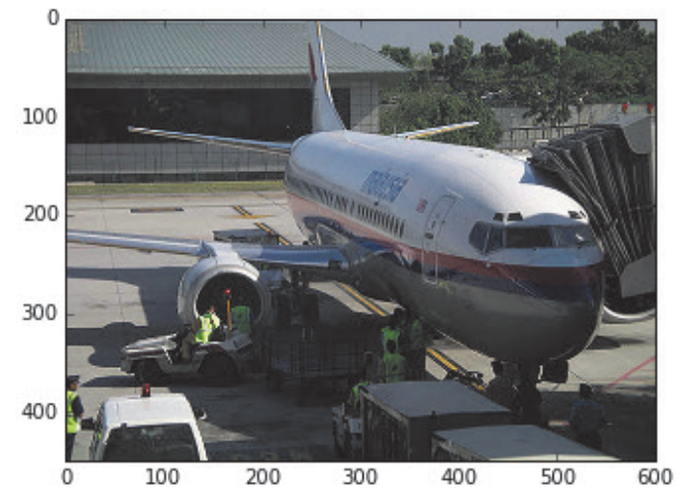


Figure 2.1.2.29 The RGB image 'plane.png' is loaded into a three dimensional array `im1` of dimensions 450 x 600 x 3

Questions

14 What is an n-dimensional array?

Key concept

Array:
An array is a container for a fixed number of values of a single data type. An array may be organized into several dimensions.

Length of an array:
The length of an array is established when the array is created. After creation, its length is fixed.

One-dimensional array:
A one-dimensional array is a data structure containing an ordered sequence of elements (e.g. numbers) that are indexed with a single integer.

Two-dimensional array:
A two-dimensional array is a data structure containing elements indexed by a pair of integers, that is, the row index and the column index.

n-dimensional array:
More generally, an n-dimensional array is a set of elements with the same data type that are indexed by a tuple of n integers.

Key concept

Element data type:
All elements in an array must have the same data type.

Storage:
Elements in an array are stored internally in a contiguous block of memory.
For example, the elements in a vector of size 5 occupy 5 consecutive memory addresses. When the dimension of the array is two or more, there is more than one way of ordering of the elements in memory but this is hidden from user.

Single licence - Abingdon School

In this chapter you have covered:

- What is an array?
  - An array is a container for a fixed number of values of a single data type
  - An array may be organized into several dimensions.
- Square bracket notation
  - Square bracket notation [ ] is used to access values in an array.
- Array index
  - The name that we use inside the square brackets is called the **array index**, e.g. 3 in [3].
- Length of an array
  - The length of an array (static array) is established when the array is created. After creation, its length is fixed
  - All elements in an array must have the same data type
  - Elements in an array are stored internally in a contiguous block of memory. For example, the elements in an array of size 5 occupy 5 consecutive memory addresses.
- One-dimensional arrays
  - A one-dimensional array is a data structure containing an ordered sequence of elements (e.g. numbers) that are indexed with a single integer
  - It is a container for a fixed number of values of a single data type
  - A one-dimensional array is a useful way of representing a vector
  - The term vector refers to a one-dimensional array of values to which operations such as add, subtract, multiple, divide can be applied to the array as a whole in an element-wise way. The result is a vector with the same dimension.
- Two-dimensional arrays
  - A two-dimensional array is a data structure containing elements indexed by a pair of integers, that is, the row index and the column index.
  - The term matrix refers to a two-dimensional array of values to which operations such as add, subtract, multiple, divide can be applied to the array as a whole in an element-wise way. The result is a matrix with the same dimensions.
- Higher dimensional arrays: n-dimensional arrays
  - An n-dimensional array is a set of elements with the same data type that is indexed by a tuple of n integers (a tuple is a sequence of data that does not change. A sequence is simply an ordered collection of things, e.g. 0 1 2 . . .).

Key concept

Vector:
The term vector refers to a one-dimensional array of values to which operations such as add, subtract, multiple, divide can be applied to the array as a whole. The operations are applied in element-wise fashion so that the result is an array of the same size as the original array(s).

Matrix:
The term matrix refers to a two-dimensional array of values to which operations such as add, subtract, multiple, divide can be applied to the array as a whole. The operations are applied in element-wise fashion so that the result is an array of the same size as the original array(s).

Single licence - Abingdon School

2 Fundamentals of data structures

2.1 Data structures and abstract data types

Learning objectives:

- *Be able to read/write from/to a text file*
- *Be able to read/write data from/to a binary (non-text) file*

2.1.3 Fields, records and files

Files

A file is a data structure for storing data. The number of items of data stored in the file can vary in time and the amount of data is not limited in the way that other data structures are because files rely on secondary storage such as magnetic disk for their storage unlike arrays which rely on RAM storage.

By using secondary storage, files persist in time because secondary storage is non-volatile whereas RAM is volatile.

A file is assigned a name called its *filename* so that it can be referenced by name.

Files have structure which is either defined by the programmer at creation time, a file of some record type or is a commonly used structure such as **text**.

Text files

Text files are files whose contents are sequences of characters organised on a line by line basis. For example, Shakespeare's Sonnet 116 shown in *Table 2.1.3.1* consists of 14 lines as do almost all of Shakespeare's sonnets.

> *Let me not to the marriage of true minds*
> *Admit impediments. Love is not love*
> *Which alters when it alteration finds,*
> *Or bends with the remover to remove:*
> *O, no! it is an ever-fixed mark,*
> *That looks on tempests and is never shaken;*
> *It is the star to every wandering bark,*
> *Whose worth's unknown, although his height be taken.*
> *Love's not Time's fool, though rosy lips and cheeks*
> *Within his bending sickle's compass come;*
> *Love alters not with his brief hours and weeks,*
> *But bears it out even to the edge of doom.*
> *If this be error and upon me prov'd,*
> *I never writ, nor no man ever lov'd.*

Table 2.1.3.1 Shakespeare's Sonnet 116

Text files may be opened and read in a text editor such as Microsoft's WordPad. Text editors expect files to be organised on a line-by-line basis and to consist of characters that can read when displayed with the exception of some specific control characters.

Each line ends with a special control character called the end of line or *newline* character (character code 10 or character codes 10 and 13).

Key term

File:
A file is a data structure for storing data. The number of items of data stored in the file can vary in time and the amount of data is not limited in the way that other data structures are because files rely on secondary storage such as magnetic disk for their storage unlike arrays which rely on RAM storage.

Key term

Text file:
Text files are files whose contents are sequences of characters organised on a line by line basis.

Questions

1. What is a file?
2. What is a text file?

Single licence - Abingdon School

The only control characters that text editors are able to handle are characters called *whitespace* characters, i.e. characters with ASCII or UTF-8 character codes 32 (space), 10 (line feed), 13 (carriage return) and 9 (tab). Other control characters have an effect on a text editor which is unpredictable and usually render the display unreadable.

Non-text files do not display well in text editors because they often contain control character codes which text editors are unable to handle because they are not control character codes 32, 10, 13 or 9.

Not only can text files be read by text editors but they may also be created and edited in a text editor.

We turn to computer programs when we wish to manipulate text files in ways that text editors do not support.

Reading from a text file

Table 2.1.3.2 shows a Python program which opens a text file with the filename 'sowpods.txt' for reading. The contents of this file are read one line at a time from the beginning of this file. Each line which is read is stored temporarily in the string variable `line`. The line of characters stored in this variable is displayed on the console by `print(line)`. Finally the file is closed.

```
f = open("sowpods.txt", "r")
for line in f:
    print(line)
f.close()
```

Table 2.1.3.2 Python program which opens a text file 'sowpods.txt' for reading and displaying

```
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
aals
aardvark
aardvarks
```

Table 2.1.3.3 The first few lines of the text file 'sowpods.txt'

The first thing that this Python program does is open the specified file in a particular mode, in this case, for reading.

The call to `open("sowpods.txt", "r")` returns a file handle which is assigned to file handle variable `f`. `"r"` specifies that the mode is reading.

The contents of file `"sowpods.txt"` are now accessed through file handle `f`.

Next, `for line in f` iterates through the file line-by-line. The file handle `f` is aware of the file structure, and is able to keep track of which line in the file is currently selected so that string variable `line` is able to receive the next line of the file.

The print statement `print(line)` outputs the line it currently stores followed by a newline character. A *newline* character is the line feed character (character code 10)

Unfortunately, this results in a blank line between lines appearing on the output because each line ends with a *newline* character as well.

```
>>>line = line.readline()
>>>line
'aa\n'
```

The solution is to delete the *newline* character ('\n') from the end of the string in variable `line` before `print(line)` is reached. *Table 2.1.3.4* shows the revised Python program.
The action of strip() removes the *whitespace* at the end of the line, i.e. the *newline* character.

```
f = open("sowpods.txt", "r")
for line in f:
    line = line.strip()
    print(line)
f.close()
```

Table 2.1.3.4 Python program which opens a text file 'sowpods.txt' for reading and displaying

Summarising, reading from a text file takes the following form:

```
open the text file for reading
read the text file line-by-line
do something with each line
close the file
```

We can also read from the file in the following ways:
`line = f.readline()` will read one line of the file into variable `line`.
`all_lines = f.read()` will read the entire contents of a file into variable `all_lines`.

Single licence - Abingdon School

Table 2.1.3.5 and *Table 2.1.3.6* show how to read from a text file in Pascal/Delphi, Java, VB.NET and C#.

Java requires try catch around the code because the methods which are called under the hood are designed to throw exceptions which have to be trapped (or caught).

Two ways of reading a file are shown for Java, VB.NET and C#.

Pascal/Delphi

```
Program ReadingATextFile;
Var
  f : TextFile;
  Line : String;
Begin
  AssignFile(f, 'Sonnet116.txt');
  Reset(f);
  While Not Eof(f)
    Do
      Begin
        Readln(f, Line); {Read line of text from file into string variable Line}
        Writeln(Line); {Write line of text in string variable Line to output}
      End;
  CloseFile(f);
  Readln;
End.
```

Java

```
import java.io.FileReader;
import java.io.BufferedReader;
public class ReadATextFile {
    public static void main(String[] args) {
       try{
               FileReader f = new FileReader("Sonnet116.txt");
               BufferedReader textReader = new BufferedReader(f);
               String line;
               while (( line = textReader.readLine()) != null ) {
                     System.out.println(line);
               }
           }
        catch (IOException e)
        {
              System.out.println(e);
        }

        FileReader inputStream = null;

        try {
                inputStream = new FileReader("Sonnet116.txt");
                int ch;
                System.in.read();
                while ((ch = inputStream.read()) != -1) {
                    System.out.print((char)ch);
                }
            }
        catch (IOException e)
        {
              System.out.println(e);
        }
    }
}
```

Table 2.1.3.5 Reading from a text file "Sonnet116.txt" in Pascal/Delphi and Java

Single licence - Abingdon School

VB.NET

```
Imports System
Imports System.IO
Module Module1
    Sub Main()
      ' Reads the entire file at once
        ' Open the file using a stream reader.
        Using f As New StreamReader("Sonnet116.txt")
            Dim line As String
            ' Read the stream string variable line and write the string to the console.
            line = f.ReadToEnd() ' Reads to end of file
            Console.WriteLine(line)
        End Using ' Dispose of all the resources
        Console.ReadLine()
      ' Reads the file line-by-line.
        ' Open the file using a stream reader.
        Using f As New StreamReader("Sonnet116.txt")
            Dim line As String
            line = f.ReadLine()
            While Not (line Is Nothing)
                Console.WriteLine(line)
                line = f.ReadLine()
            End While
        End Using
        Console.ReadLine()
    End Sub
End Module
```

C#

```
using System;
namespace ReadingATextFile
{
    class Program
    {
        static void Main(string[] args)
        {
         // Read the whole file into a string array lines.
            string[] lines = System.IO.File.ReadAllLines("Sonnet116.txt");
            foreach (string line in lines)
            {
                Console.WriteLine(line);
            }
            Console.ReadLine();
            string nextLine;
            // Read the file and display it line by line.
            System.IO.StreamReader f = new System.IO.StreamReader("Sonnet116.txt");
            while ((nextLine = f.ReadLine()) != null)
            {
                System.Console.WriteLine(nextLine);
            }

            f.Close();
            Console.ReadLine();
        }
    }
}
```

Table 2.1.3.6 Reading from a text file "Sonnet116.txt" in VB.NET and C#

Writing to a text file

In Python, if we want to write to a file with filename `"studentresults.txt"`, we open this file in write mode (`"w"`) with

```
open("studentresults.txt", "w")
```

This will create a new file `"studentresults.txt"` or overwrite this file if it exists already.

The program in *Table 2.1.3.7* collects a student name and the student's exam score typed at the keyboard and then using the file handle `f` to the opened file, writes student name then a comma then exam score on the same line to the opened text file

```
f.write(student_name + "," + exam_mark + "\n")
```

`"\n"` is the special control character called the end of line or *newline* character.

Single licence - Abingdon School

This ensures that the file handle `f` is ready to write the next student name, comma, exam score combination on the next line.

Entering the student name "quit" causes the program to exit the w*hile loop* but not before printing "Quitting...". Finally the file is closed.

```
f = open("studentresults.txt", "w")
while True:
    student_name = input("Name: ")
    if student_name == "quit":
        print("Quitting...")
        break
    exam_mark = input("Exam score for " + student_name + " : ")
    f.write(student_name + "," + exam_mark + "\n")
f.close()
```

Table 2.1.3.7 Python program which creates and writes lines of text to a text file 'studentresults.txt'

Table 2.1.3.8 shows the contents of `"studentresults.txt"` produced by executing the program in *Table 2.1.3.7*.

```
Bond K, 95
Cheadle P, 85
Gunawardena P, 90
Khan M, 88
De Silva S, 75
Smith E,55
Teng P, 85
Tipp S,30
```

Table 2.1.3.8 Contents of "studentresults.txt" created by program in Table 2.1.3.7

Appending to a text file

Changing the file mode to `"a"` allows new lines to be appended to the end of

`"studentresults.txt"` if it exists

or to an empty newly created `"studentresults.txt"` - *Table 2.1.3.9*.

```
f = open("studentresults.txt", "a")
while True:
    student_name = input("Name: ")
    if student_name == "quit":
        print("Quitting...")
        break
    exam_mark = input("Exam score for " + student_name + " : ")
    f.write(student_name + "," + exam_mark + "\n")
f.close()
```

Table 2.1.3.9 Python program which appends lines of text to an existing text file 'studentresults.txt'

Table 2.1.3.10 shows two ways of writing to text files in Pascal/Delphi, Java and one way in VB.NET.

Pascal:

1. Line-by-line:

```
For Line in Lines
  Do Writeln(f, Line);
```

2. Writing all the lines in one go

```
Lines.LoadFromFile('Sonnet116.txt');
Lines.SaveToFile('AnotherNewFile.txt');
```

Java:

1. Using java.io.PrintWriter
2. Using java.io.PrintWriter and java.io.File

VB.NET: Using StreamWriter and WriteLine.

Table 2.1.3.11 shows one way of writing to text files in C#.

Single licence - Abingdon School

Pascal/Delphi

```
Program WriteToATextFile;
Uses Classes;
Var
  f : TextFile;
  Lines : TStringList;
  Line : String;
Begin
  AssignFile(f, 'NewFile.txt');
  Rewrite(f);
  Lines := TStringList.Create;
  Lines.Add('Let me not to the marriage of true minds');
  Lines.Add('Admit impediments. Love is not love');
  Writeln(Lines.text);
  For Line in Lines
    Do Writeln(f, Line); {Write a line of text to text file}
  CloseFile(f);
  Lines.Clear;
  Lines.LoadFromFile('Sonnet116.txt');
  Lines.SaveToFile('AnotherNewFile.txt');
  Readln;
End.
```

Java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.io.File;
public class WriteToATextFile {
  public static void main(String[] args) {
    try{
         PrintWriter printLine = new PrintWriter("Z:/NewFile.txt");
         String line = "Let me not to the marriage of true minds";
         printLine.println(line); // Write a line of text to the file
         printLine.close(); // Close the file
         File f = new File("Z:/AnotherNewFile.txt");
         if (!f.exists()) {
           if (f.createNewFile()) {
             PrintWriter newPrintLine = new PrintWriter(f);
             newPrintLine.println("Let me not to the marriage of true minds");
             newPrintLine.println("Admit impediments. Love is not love");
             newPrintLine.close();
            }
         }
    }
    catch (IOException e){
         System.out.println(e);
    }
  }
}
```

VB.NET

```
Imports System
Imports System.IO
Imports System.Text
Module Module1
    Sub Main()
        Try
            Dim w As StreamWriter = New StreamWriter("NewFile.txt")
            w.WriteLine("Let me not to the marriage of true minds")
            w.WriteLine("Admit impediments. Love is not love")
            w.Close()
        Catch e As Exception
            Console.WriteLine("The process failed: {0}", e.ToString())
        End Try
    End Sub
End Module
```

Table 2.1.3.10 Writing to a text file in Pascal/Delphi, Java and VB.NET

Single licence - Abingdon School

C#

```
using System;
using System.Text;
using System.IO;
namespace WriteToATextFile
{
    class Program
    {
        public static void Main()
        {
            try
            {
                using (StreamWriter w = new StreamWriter("NewFile.txt"))
                {
                    w.WriteLine("Let me not to the marriage of true minds");
                    w.WriteLine("Admit impediments. Love is not love");
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("The process failed: {0}", e.ToString());
            }
        }
    }
}
```

Table 2.1.3.11 Writing to a text file in C#

Programming tasks

1. Write a program which makes a copy of a text file. Your program should prompt the user to enter the names of the input and output text files.

2. Write a program that reads a text file and displays it with the corresponding line number at the beginning of each line. Start line numbering from 1.

3. The text file "Dict5LetterWords.txt" (download from www.educational-computing.co.uk/CS/Textfiles.html) contains 5 letter words. Write a program which finds all 5 letter words in this file which contain the substring 'oe'.

4. The text file "sowpods.txt" is an official Scrabble dictionary (download from www.educational-computing.co.uk/CS/Textfiles.html). Write a program to find all words containing a particular substring in the text file "sowpods.txt". The program should prompt the user to enter a substring to search for.

5. Write a program to create a Geography quiz which tests a user's knowledge of country capitals.
   Use the text file "countriescapitals.txt" (download from www.educational-computing.co.uk/CS/Textfiles.html) which is a comma separated file of countries and their capitals.
   The program should display the name of a country chosen at random from this text file and prompt the user to supply the name of the capital of this country. The program should then check the user's answer.
   If the user's answer is correct the program should respond "Well done, you got it right!". If the user's answer is incorrect the program should respond "Incorrect answer, the correct answer is ???????" where the correct answer is substituted for the string "???????" when the program executes.

Single licence - Abingdon School

Binary files

A binary file is considered to be any file that is not a text file.

Binary files can only be processed by applications that know about the file's structure.

For example, the Pascal program shown in *Table 2.1.3.12* creates a file with the structure *file of integer*.

If we want to write to a file with filename 'FileOfIntegers.int', we open this file in write mode with `Rewrite(f);`

This will create a new file 'FileOfIntegers.int' or overwrite this file if it exists already.

The file handle, f, to the file 'FileOfIntegers.int' is created with

```
AssignFile(f, 'FileOfIntegers.int');
```

Each integer is written to this file by the statement

```
Write(f, NextInteger);
```

The file 'FileOfIntegers.int' is closed with `CloseFile(f);`

```
Program FileOfIntegers;
Var
  f : File Of Integer;
  n, i : Integer;
  NextInteger : Integer;
Begin
  AssignFile(f, 'FileOfIntegers.int');
  Rewrite(f);
  Write('How many integers? ');
  Readln(n);
  For i := 1 To n
    Do
      Begin
        Write('Input next integer: ');
        Readln(NextInteger);
        Write(f, NextInteger);
      End;
  CloseFile(f);
  Writeln('File of ', n, ' integers created');
End.
```

Table 2.1.3.12 Pascal program which creates a file with the structure file of integer

The program in *Table 2.1.3.12* creates a *file of integer* containing `n` integers.

If we want to read the contents of the file with filename 'FileOfIntegers.int' we open this file in reading mode as shown in *Table 2.1.3.13* with

```
Reset(f);
```

The file handle, f, to the file 'FileOfIntegers.int' is created as before with

```
AssignFile(f, 'FileOfIntegers.int');
```

Each integer is read from this file by the statement

```
Read(f, NextInteger);
```

The end of file is checked for by `Eof(f)` which returns True when the end of file is reached otherwise it returns False.

We can create a file with a different structure by changing `File Of Integer` to a `File Of` *a different type*, e.g. `File Of Real` or `File Of Byte` or `File Of SomeArrayType` or `File of SomeRecordType`.

```
Program ReadingFileOfIntegers;
Var
  f : File Of Integer;
  NextInteger : Integer;
Begin
  AssignFile(f, 'FileOfIntegers.int');
  Reset(f);
  While Not Eof(f)
    Do
      Begin
        Read(f, NextInteger);
        Writeln(NextInteger);
      End;
  CloseFile(f);
  Writeln('End of file reached, press return to close
DOS window');
  Readln;
End.
```

Table 2.1.3.13 Pascal program which reads a file with the structure file of integer

`File of Byte` is interesting because every file consists of bytes.

Viewing the structure of a file as other than `File Of Byte` applies an **abstraction** to the file contents. This abstraction occurs when the file is created.

The file type declaration sets the unit of data that will be written to a file of this type, e.g. for a file type `File Of`

Single licence - Abingdon School

`Integer`, the unit of data is *integer* which typically is four bytes.

However, we can open any file with a handle which has been declared as `File Of Byte` and read the contents of the file byte by byte.

This is very useful when the file's data structure is unknown or when we might want to manipulate its contents, e.g. in an application of steganography. Steganography is the science of hiding messages in pictures and other media.

The program in *Table 2.1.3.14* is written to treat any file as a `File Of Byte`, in particular bitmap files.

```
Program ReadABitMap;
Type
  BitMapFilePipeType = File Of Byte;
Var
  BitMapFilePipe: BitMapFilePipeType;
  Filename: String;
  NextByte : Byte;
Begin
  Write('Input name of file to be opened for reading: ');
  Readln(Filename);
  Assign(BitMapFilePipe, Filename);
  Reset(BitMapFilePipe);
  While Not Eof(BitMapFilePipe)
    Do
      Begin
        Read(BitMapFilePipe, NextByte);
        If NextByte In [32..126]
          Then Write(NextByte : 1, ' ');
          {Note the single space character}
      End;
End.
```

Table 2.1.3.14 Pascal program which intended for reading a bitmap files, byte by byte (it can read any file type)

Table 2.1.3.15 shows a program for hiding a message in the pixel area of a bitmap file by replacing a pixel byte with a message byte.

```
Program HideMessageInABitMap;
Type
  BitMapFilePipeType = File Of Byte;
Var
  BitMapFilePipe, BitMapFileCodedPipe : BitMapFilePipeType;
  Filename, Message : String;
  NextByte : Byte;
  n, i : Integer;
Begin
  Write('Input name of file to be opened for reading: ');
  Readln(Filename);
  Assign(BitMapFilePipe, Filename);
  Reset(BitMapFilePipe);
  Assign(BitMapFileCodedPipe, 'c' + Filename);
  Rewrite(BitMapFileCodedPipe);
  Write('Input message up to 30 characters in length: ');
  Readln(Message);
  If Length(Message) < 30
    Then
      For i := 1 To 30 - Length(Message)
        Do Message := Message + ' ';
  {Pads message with spaces so that it is always 30 characters in length}
  n := 0;
  i := 0;
  While Not Eof(BitMapFilePipe)
    Do
      Begin
        Read(BitMapFilePipe, NextByte);
        If (n >= 1078) And (n <= 1078 + Length(Message)-1)
          Then
            Begin
              i := i + 1;
              NextByte := Ord(Message[i]);
            End;
        n := n + 1;
        Write(BitMapFileCodedPipe, NextByte);
      End;
End.
```

Table 2.1.3.15 Pascal steganography program which reads a bitmap file and a 30 character message which hides in the bitmap

Single licence - Abingdon School

Table 2.1.3.16 shows samples of code in C#, VB.NET, Java and Python for writing data to and reading data from non-text files, i.e. binary files.

C#

```
using (BinaryWriter writer = new BinaryWriter(File.Open(fileName, FileMode.Create)))
{
    writer.Write(3.142F);
    writer.Write("Hello World!");
    writer.Write(10);
    writer.Write(true);
}
If (File.Exists(fileName)) Then
{
    using (BinaryReader reader = new BinaryReader(File.Open(fileName, FileMode.Open)))
    {
        pi = reader.ReadSingle();
        greeting = reader.ReadString();
        score = reader.ReadInt32();
        flag = reader.ReadBoolean();
    }
}
```

VB.NET

```
Using writer As BinaryWriter = New BinaryWriter(File.Open(fileName, FileMode.Create))
    writer.Write(3.142F)
    writer.Write("Hello World!")
    writer.Write(10)
    writer.Write(true);
End Using
If (File.Exists(fileName)) Then
    Using reader As BinaryReader = New BinaryReader(File.Open(fileName, FileMode.Open))
        pi = reader.ReadSingle()
        greeting = reader.ReadString()
        score = reader.ReadInt32()
        flag = reader.ReadBoolean()
    End Using
End If
```

Java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
public class WritingToABinaryFile {
    public static void main(String[] args) throws IOException {
        WritingToABinaryFile binary = new WritingToABinaryFile();
        byte[] bytes = binary.readFromBinaryFile("C:/Javacode/SmallJPG.jpg");
        binary.writeToBinaryFile(bytes,"C:/Javacode/SmallJPG2.jpg");
    }
    byte[] readFromBinaryFile(String aFileName) throws IOException {
        Path path = Paths.get(aFileName);
       return Files.readAllBytes(path);
    }

  void writeToBinaryFile(byte[] aBytes, String aFileName) throws IOException {
    Path path = Paths.get(aFileName);
    Files.write(path, aBytes); //creates, overwrites
  }
}
```

Python

```
from struct import pack
with open("c:/Javacode/test.bin", "wb") as file:
  file.write(pack("IIIII", *bytearray([120, 3, 255, 0, 100])))
with open("c:/Javacode/foo.bin", "rb") as f:
    byte = f.read(1)
    while byte:
        byte = f.read(1)
        print(byte)
```

Table 2.1.3.16 Writing data to and reading data from a binary file in C#, VB.NET, Java and Python

Single licence - Abingdon School

Questions

3 What is a binary file?

Programming tasks

6 Write a program which creates a file of decimal numbers (e.g. decimal number 12.67) with filename "FileOfDecimals.int". Your program should prompt the user to enter n, the number of decimal numbers to store in the file. The program should then collect n decimal numbers from the user.

7 Investigate steganography with the programs in *Table 2.1.3.14* and *Table 2.1.3.15* recreated in a programming language with which you are familiar.

In this chapter you have covered:

- Reading/writing from/to a text file
- Reading/writing data from/to a binary (non-text) file

Single licence - Abingdon School

2 Fundamentals of data structures

2.1 Data structures and abstract data types

Learning objectives:

- *Be familiar with the concept and uses of a:*
  - *queue*
  - *stack*
  - *graph*
  - *tree*
  - *hash table*
  - *dictionary*
  - *vector*

2.1.4a Abstract data types/data structures

Abstract data type

An **abstract data type (ADT)** is a logical description of how a user views the data and the operations that are allowed on this data without regard to how both data and operations will be implemented.

An abstract data type is described logically as a collection of data items together with a set of operations defined on the collection.

For example, a queue is an abstract data type in which data items are added at the rear of the queue and removed from the front - *Figure 2.1.4.1*.

Data structures

The implementation of an abstract data type, often referred to as a **data structure**, will require that a physical view of the data is provided using some collection of programming constructs and primitive data types.

The physical view is a named collection of variables, possibly of different data types, which are connected in various ways.

The collection consists of an aggregation of cells, e.g. characters typed at a keyboard are stored in the collection, one character per cell.
The cell is the basic building block of data structures.

Data structures allow us to store data in different ways, e.g. a linear queue may be implemented as a static array, or a dynamic array, or a linear list or a linked list.

Why use abstract data types and why use data structures?

The separation of the abstract data type perspective from the data structure perspective allows complex data models for problems to be defined without having to give any indication as to the details of how the model will actually be built. This provides an **implementation-independent view** of the data.

Since there will usually be many different ways to implement an abstract data type, this implementation independence allows a programmer to change the details of an implementation without changing the way a user of the data interacts with it.

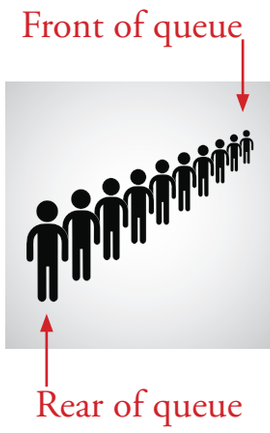


Figure 2.1.4.1 The concept of a queue

Questions

1. What is an abstract data type?
2. What is a data structure?
3. Give **one** reason why abstract data types are used.

Single licence - Abingdon School

Queue

We are all used to queueing in a line. For example, we wait in a queue at a supermarket check-out, and we wait in a queue at a bus stop. Queues have only one way in (the rear) and only one way out (the front).

A **queue** is thus an ordered collection of items (ordered by time of arrival) for which new items are added at one end, called the *rear* or *tail*, and existing items are removed at the other end, commonly called the *front* or *head* as shown in *Figure 2.1.4.2*.

The item that has been in the collection the longest is at the front. The ordering principle is sometimes called First In First Out or just FIFO.

A queue is therefore a **First In First Out (FIFO) abstract data type**.

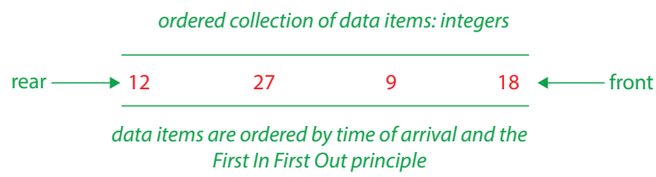


Figure 2.1.4.2 A queue showing the collection of data items 18, 9, 27, 12

Key concept

Queue:
A queue is an ordered collection of items (ordered on the first-in, first-out principle) and a First In First Out abstract data type.
One end is called the front or head, the other end is called the rear or tail.
Items are added to the rear and removed from the front.

Queues are used in computing. For example, in a network of computers served by a print server, printing jobs are sent by different computers to the print server and queued at the print server ready for printing. The print server picks the next job to print from the front of the queue. New print jobs join the queue at the rear.

Linear queue

Figure 2.1.4.3 shows a **linear queue** data structure that defines a storage structure in which character data items are stored one after another in the separate cells of the structure.

Characters are added at the end labelled *rear* and removed from the other end labelled *front*.

The item at the front of the queue has been in the queue the longest. The item at the rear of the queue is the most recently added one. We call *rear* and *front* pointer variables.

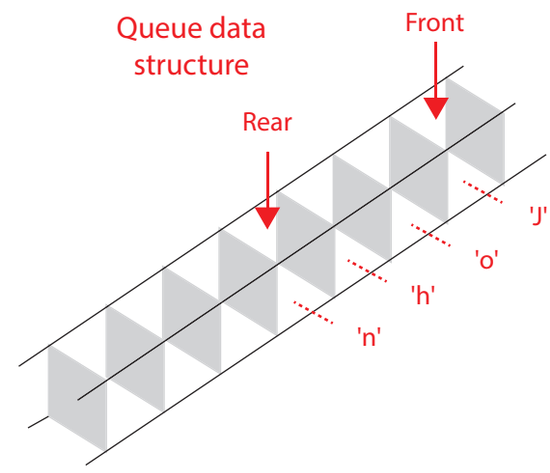


Figure 2.1.4.3 A linear queue showing the data items 'J', 'o', 'h', 'n' stored in separate cells

Figure 2.1.4.4 shows a queue containing four integers and pointers to the *front* and *rear*. The cells have been labelled 0, 1, 2, 3, 4, 5, 6. The pointer variables, *front* and *rear*, have values 0 and 3, respectively

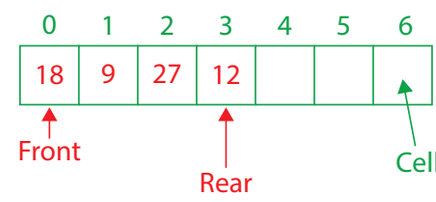


Figure 2.1.4.4 Linear queue storing the integer data items 18, 9, 27, 12

Figure 2.1.4.5 shows the queue after one item, the integer 18, has been removed. Removal has a strange meaning. Removal actually means taking a copy of the item at the front of the queue and then changing the *front* pointer so that it points to the next cell, i.e. *front* = 1.

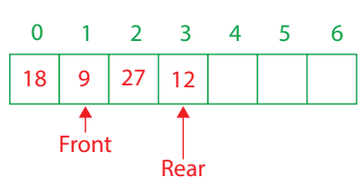


Figure 2.1.4.5 Linear queue after the item at the front, integer 18, has been "removed"

Single licence - Abingdon School

Figure 2.1.4.6 shows the queue after a single item, integer 4, has been added. This data item is added to the rear of the queue. The *rear* pointer is changed so that it points to the new item, integer 4, in the queue.

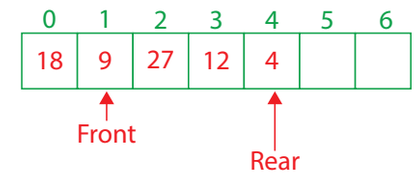


Figure 2.1.4.6 Linear queue after a single item, integer 4, has been added.

Questions

4. Sketch the linear queue shown in *Figure 2.1.4.6* after two items have been removed and one item added, the integer 35.
5. Sketch the linear queue for your answer in Question 4 after a further three more items have been removed. What value have you chosen for the front pointer and what value have you chosen for the rear pointer for the resulting queue? Explain your answer.
6. Why is a queue called a First In First Out abstract data type?

What if a programming language doesn't have a queue built-in data type?

If the programming language used to create a queue does not have a built-in data type for creating a queue then a queue must be built from primitives that the programming language does possess.

For example, a linear queue of 200 integer cells queue can be defined using a one-dimensional array primitive in Pascal as follows:

```
QueueType = Array[1..200] of Integer;
```

We need to indicate the front and rear of this queue. We can do this with the integer variables `F` and `R`.

If the queue is empty, then we use an integer value for `F` and `R` which is outside the index range for the array, 1..200, say `F = R = 0`.

Figure 2.1.4.7 shows this implementation with `F = 1`, and `R = 4`.

The integer values stored in this queue are 18, 9, 27, and 12.

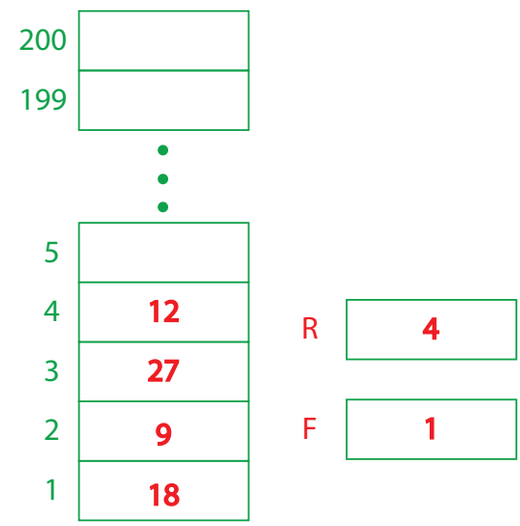


Figure 2.1.4.7 A linear queue implemented as an array of integer

Uses of queues

In addition to printing queues, operating systems use a number of different queues to control processes within a computer. For example, a queue-like buffer is used to store characters typed at a keyboard so that it is possible to type ahead of the characters that appear on the screen. This is important because sometimes the computer is busy doing something else and therefore cannot do more than just collect the characters typed as keystrokes at the keyboard. Eventually these characters will be displayed on the screen in the correct order.

A queue-like keyboard buffer can also support lazy I/O. Lazy I/O is where a user can correct what is typed using the backspace key before it is processed by an application that reads the keyboard buffer. A line of characters typed at the keyboard is not available for processing until the return key is pressed.

Single licence - Abingdon School

Figure 2.1.4.8 A stack of books

Stack

A **stack** is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the *top*. The end opposite the *top* is known as the *base* or *stack base*. *Figure 2.1.4.8* shows a stack of books.

Items stored in the stack that are closer to the base represent those that have been in the stack the longest. The most recently added item is the one that is on top and is the one to be removed first.

This ordering principle is sometimes called **Last In First Out (LIFO)** or **First In Last Out (FILO)**. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base.

For this reason a stack is a Last In First Out (LIFO) abstract data type:

- Items can only be added to a stack at the top. This is known as pushing (an item onto the stack).
- Items can only be taken off a stack at the top. This is known as popping (an item from or off the stack).

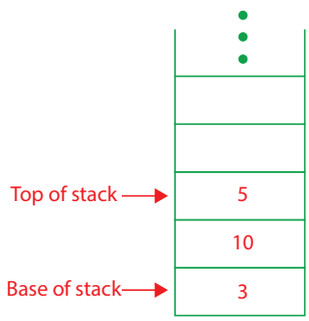


Figure 2.1.4.9 A stack of integers

Figure 2.1.4.9 shows a stack of integers. Integer 3 has been on the stack the longest and integer 5 is the most recently added.

Figure 2.1.4.10 shows the stack after the removal of one item, integer *5*. Removal has a strange meaning in stack programming. Removal means a copy is taken and the stack pointer is decremented by one so that it now points to the item immediately below the removed item. An item is said to be popped from the stack when it is removed from the stack.

Figure 2.1.4.11 shows the stack after the removal of another item, integer 10.

Figure 2.1.4.12 shows the stack after a new item, integer 22, has been added.

It is said that an item is pushed onto the stack when it is added to the stack.

In all three cases, it is only the item pointed at by the top of stack pointer that can be accessed.

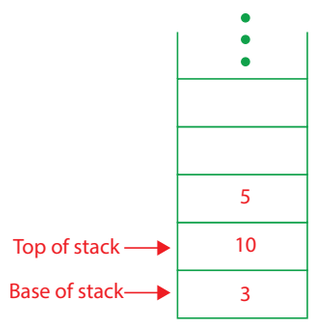


Figure 2.1.4.10 Stack after one item, integer 5, popped

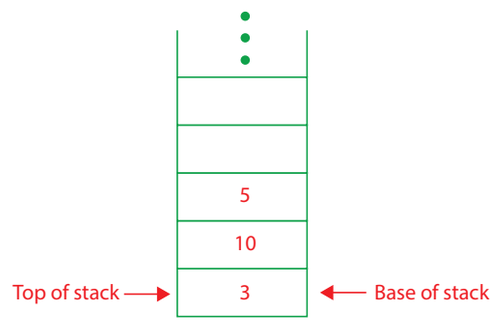


Figure 2.1.4.11 Stack after another item, integer 10, popped

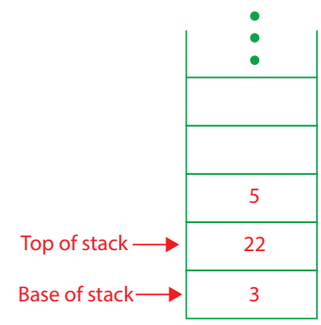


Figure 2.1.4.12 Stack after integer 22 pushed onto stack

Single licence - Abingdon School

Questions

7. Sketch the stack shown in *Figure 2.1.4.10* after one item has been removed and two items have been added, 16 and 67.

8. Why is a stack called a Last In First Out abstract data type?

9. *Figure 2.1.4.13* shows a stack of labelled books. Show with a series of sketches using a second stack how the order of the books can be reversed so that A is on top and H on the bottom.

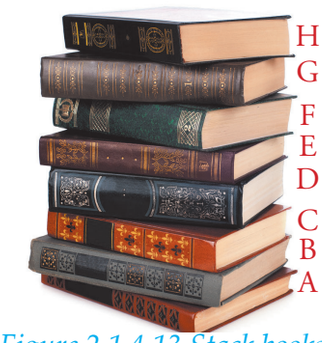


Figure 2.1.4.13 Stack books labelled A, B, C, D, E, F, G, H

10. A bookshelf contains six books, labelled A, B, C, D, E, F. They are arranged on the bookshelf in the following order C, A, F, E, D, B. If we treat this arrangement of books as a linear queue with book B at the front and book C at the rear, we can reverse the order of the books on the bookshelf using a stack so that they are stored back on the bookshelf in the arrangement B, D, E, F, A, C.

    Show with a series of sketches how this can be done.

What if a programming language doesn't have a stack built-in data type?

If the programming language used to create a stack does not have a native data type for creating stacks then a stack must be built from primitives which the programming language does possess (support).

Here is one example: A stack data type can be defined using a one-dimensional array primitive in the programming language Pascal as follows:

```
StackType = Array[1..200] of Integer;
```

The following variable declarations in Pascal create a variable `Stack` of data type `StackType`, a top of stack pointer variable `S` of data type `Integer` and a variable `Item` also of data type `Integer`:

```
Var
   Stack : StackType;
   S : Integer;
   Item : Integer;
```

The variable `Item` will be used for the value pushed to the stack and the value popped from the stack.

The top of stack pointer (sometimes this is abbreviated to just stack pointer), `S`, is an integer variable.

Figure 2.1.4.14 shows the stack after the integer values 31, 10 and 5 have been pushed to the stack. Note that the top of stack pointer, `S`, has the array subscript value 3. The base of stack has the array subscript 1.

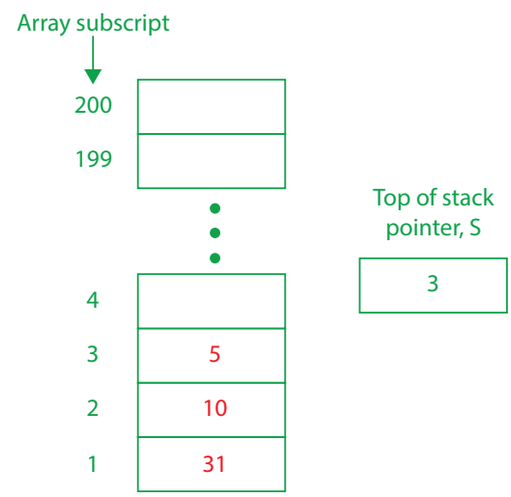


Figure 2.1.4.14 Stack created in Pascal

To indicate when the stack is empty, a value outside the range of array subscripts is assigned to `S`.

In this example the empty stack is chosen to be when `S = 0`.

Single licence - Abingdon School

Questions

11 A stack data type is defined using a one-dimensional array primitive l as follows:

```
StackType = Array[0..199] of Integer;
```

(a) What is the maximum numbers of integers that can be stored in a stack of this data type?

(b) When the stack is full what is the value of variable `S` which is used to indicate the top of stack?

(c) What is the value of variable `B` which is used to indicate the base of this stack?

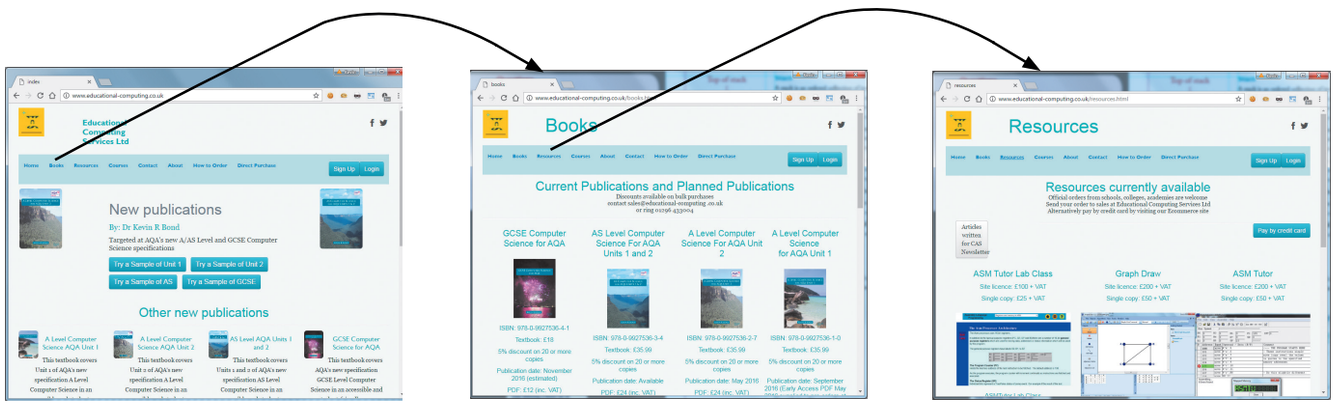
(d) Suggest a suitable value for `S` when the stack is empty.

Uses of stacks

Back button

One of the most useful ideas related to stacks comes from the simple observation that the order of insertion is the reverse of the order of removal.

This reversal property is applied, for example, in every web browser which has a Back button. As you navigate from web page to web page as shown in *Figure 2.1.4.15*, these pages are placed on a stack, or rather the URLs are placed on a stack as shown in *Figure 2.1.4.16*. The current page that is displayed is on the top and the first page that was visited is at the base. If you click on the Back button, you begin to move in reverse order through the pages.



www.educational-computing.co.uk *www.educational-computing.co.uk/books.html* *www.educational-computing.co.uk/resources.html*

Figure 2.1.4.15 Web pages visited in left to right order

Key concept

Stack:
A stack is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end called the top of stack.
For this reason a stack is a Last In First Out (LIFO) abstract data type.

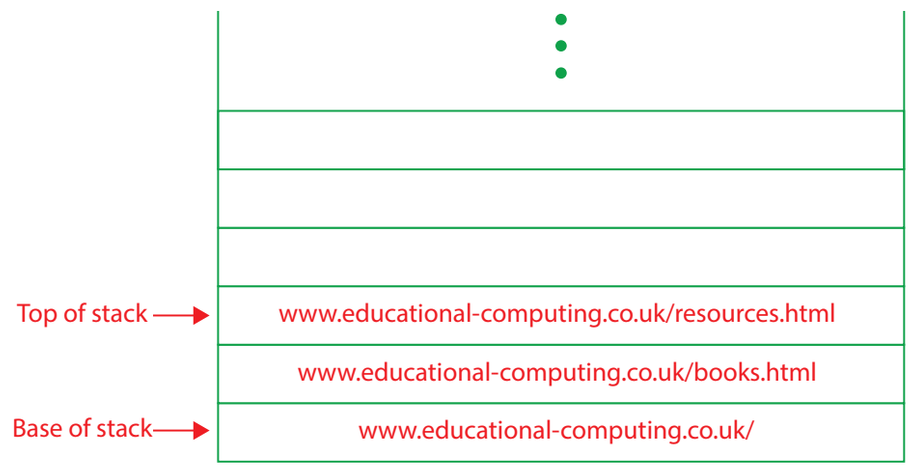


Figure 2.1.4.16 The stack of URLs with the URL of the current page on top and the first page visited on the bottom

Single licence - Abingdon School

Subroutine calls and stack frames

Very importantly, a stack is used to keep track of subroutine calls in stack frames.

A stack frame consists of

1. Return address (program counter)
2. Contents of registers (e.g. stack pointer register)
3. Actual parameters/arguments passed to subroutine
4. Local variables

For example, suppose a subroutine `DoExample` is called within the main program block of code

```
  ⋮
s ← 6
t ← 8
DoExample(s, t)
Output "Another go (Y/N)?"
  ⋮
```

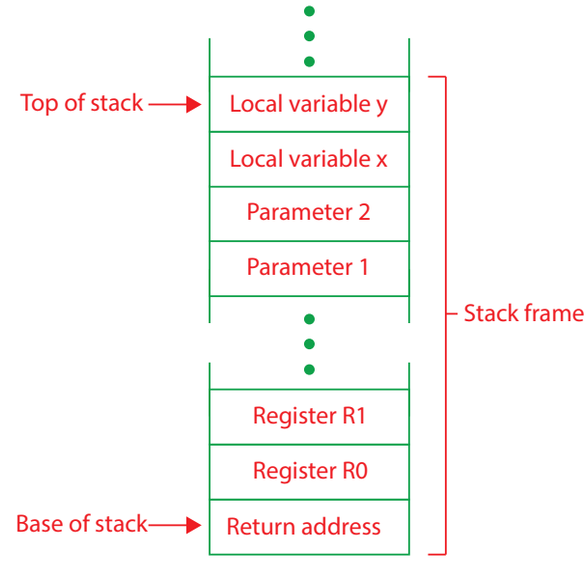


Figure 2.1.4.17 Stack frame resulting from a call to subroutine DoExample

And subroutine `DoExample` is defined as follows

```
Subroutine DoExample(Parameter1, Parameter2 : Integer)
Var
  x, y : Integer
BeginBodyOfSubroutine
  x ← 2*Parameter1
  y ← x - Parameter2
  Output(y)
EndBodyOfSubroutine
```

Then a stack frame is created on the stack as shown in *Figure 2.1.4.17* with space to store:

- the address to return to (Output "`Another go (Y/N)?`") after `DoExample` finishes executing
- the volatile environment, i.e. the processor's registers, because their contents may be overwritten by `DoExample`
- the values of the two parameters, `6` and `8`, so that these are accessible to `DoExample`
- values assigned to the two local variables, `x` and `y`

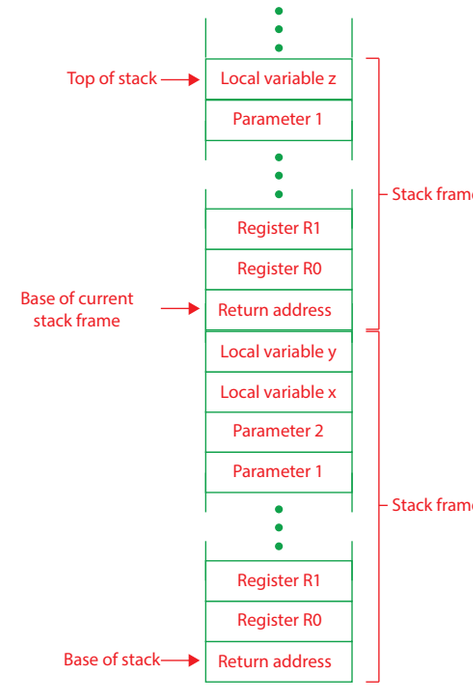


Figure 2.1.4.18 New stack frame resulting from a call to subroutine Output

Figure 2.1.4.18 shows that a second stack frame is created when `Output(y)` is called from `DoExample`. A current stack frame pointer shown as base of current stack frame in the *Figure 2.1.4.18* is required for this situation. This pointer is used to access the value of the parameter and the value of the local variable `z` of subroutine `Output`.

Single licence - Abingdon School

Questions

12 Suppose subroutine `DoExample` was modified as follows

```
Subroutine DoExample(Parameter1, Parameter2 : Integer)
  Var
    x, y : Integer
  BeginBodyOfSubroutine
    x ← 2*Parameter1
    y ← x - Parameter2
    DoExample(1,1)
    Output(y)
  EndBodyOfSubroutine
```

It now contains a call to itself. The stack for subroutine calls is limited to using 600 storage cells. What is likely to happen when this subroutine is called as follows `DoExample(6,8)`?

13 Why is it not a good idea to pass the contents of a large array in a subroutine parameter?

Evaluating an expression

An arithmetic expression such as 5 + 3, is interpreted as 5 is added to 3 since the addition operator + appears between the 5 and the 3 in the expression. This type of notation is referred to as infix since the operator + is in between the two operands, 5 and 3, which it is working on.

Consider another infix example, 5 + 3 x 6. The operators + and x still appear between the operands, but there is a potential ambiguity. Which operands do they work on? Is it 5 plus 3 then times 6 or is it 3 times 6 then add 5?

Fortunately, most people know about operator precedence:

Operators of higher precedence are used before operators of lower precedence.

The only thing that can change that order is the presence of brackets.
The precedence order for arithmetic operators places multiplication and division above addition and subtraction.
If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

Applying operator precedence to the expression 5 + 3 x 6

3 and 6 are multiplied first, and
5 is then added to that result.

The use of brackets as follows in (5 + 3) x 6 would force the addition of 5 and 3 to be done first before the multiplication.

Computers need a little guidance to know exactly in which order to evaluate an arithmetic expression.
It helps if the arithmetic expression is presented for evaluation in a form called postfix.

For example, the postfix form of the infix expression 5 + 3 is 5 3 +.
The postfix form of an expression makes it possible to use a stack to evaluate the expression as follows

Working from left to right, push each operand onto stack. On encountering an operator, pop top two items and apply operator. Push result onto stack.

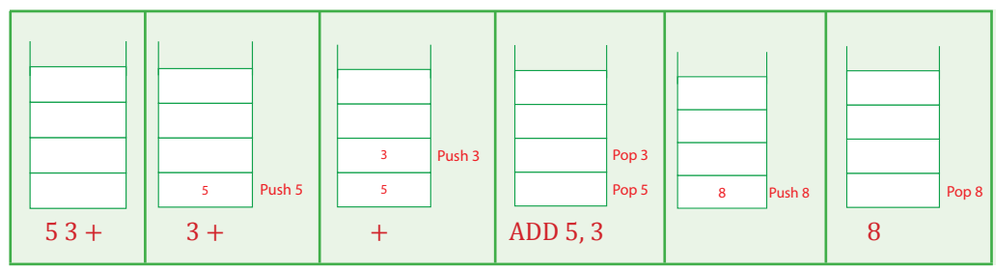


Figure 2.1.4.19 Using a stack to evaluate a postfix expression

Single licence - Abingdon School

Questions

14 The infix expression 5 + 3 x 6 is in postfix 5 3 6 x +.
Show how this would be evaluated using a stack and the operations Push and Pop.

15 Show how a stack and the operations Push and Pop would be used to evaluate the following postfix expressions

(a) 5 3 + 6 4 + x (b) 5 2 ↑ (c) 3 5 2 ↑ + (d) 5 3 + 6 4 + /

(↑ is the exponentiation operator)

Graph

A **graph** is a way of showing connections between things such as Facebook friends or towns and cities.

Figure 2.1.4.20 is a graph showing the road connections between Aylesbury, Oxford, High Wycombe and St Albans.
The red lines are called **edges** or **arcs** and correspond to roads.
The twists, turns and junctions of the road connections have been abstracted away and replaced by straight lines.
Each edge is labelled with the distance between the towns/cities it connects, e.g. the edge connecting Aylesbury to St Albans is labelled 30 because the distance by road is 30 miles.

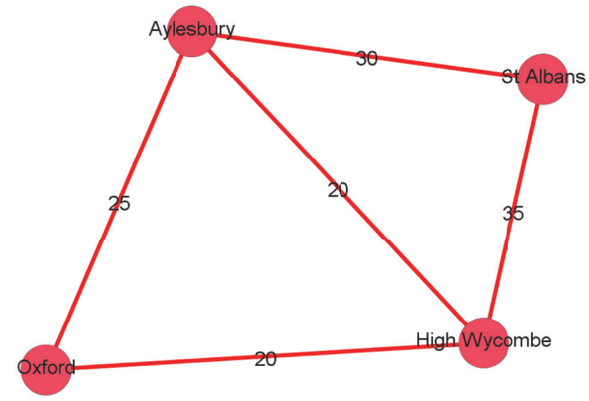


Figure 2.1.4.20 Graph showing road links between some towns/cities drawn with Gephi - https://gephi.org

Each town/city is a **vertex** or **node**, shown in *Figure 2.1.4.20* as a filled red circle. *Figure 2.1.4.20* was created in Gephi, a free and open-source tool for graph visualisation of data and structures.

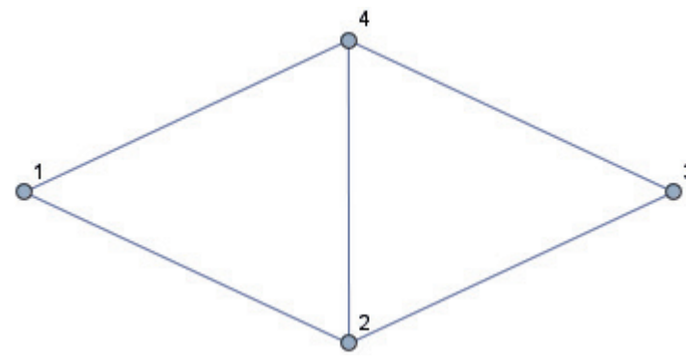


Figure 2.1.4.21 Graph produced in the Wolfram programming language - https://develop.wolframcloud.com

Figure 2.1.4.21 shows a graph created in the Wolfram programming language with the following programming statement:

```
UndirectedGraph[{1->2,2->3,3->4,4->1,2->4}, VertexLabels->All]
```

This makes a bidirectional connection between vertices/nodes 1 and 2, `1->2`, 2 and 3, `2->3`, etc.
The graph abstract data type is at the heart of GPS-based route planners such as TomTom™.

Figure 2.1.4.22 shows a typical route finding task, finding a route through a maze from entrance to exit, e.g. house to exit from the "maze garden".

How could the junctions, entrance, exit and pathways between these be represented if the task was delegated to a computer program?

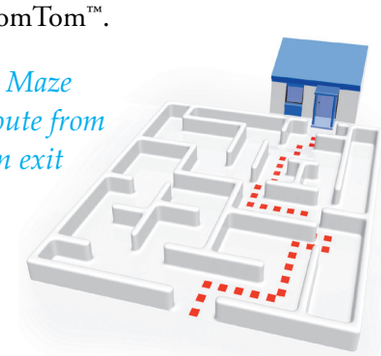


Figure 2.1.4.22 Maze garden showing route from house to garden exit

Single licence - Abingdon School

First let's abstract away unnecessary detail and record the maze in a representation called a graph using the rules for vertices and edges shown in *Table 2.1.4.1*.

To keep the description as simple as possible consider the "maze" shown in *Figure 2.1.4.23* which has one entrance labelled 1, one dead end labelled 3, one exit labelled 4 and one junction labelled 2.

The graph that models this is shown in *Figure 2.1.4.24*. The circles represent vertices or nodes and the interconnecting lines, edges.

Vertices

1. Vertex for a starting point, i.e. entrance
2. Vertex for a finishing point, i.e. exit
3. Vertices for all dead ends
4. Vertices for all the points in the maze where more than one path can be taken, i.e. junction

Edges

1. Connect the vertices according to the paths in the maze.

Table 2.1.4.1 Table showing which vertices are connected

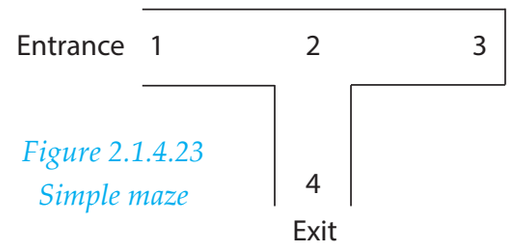


Figure 2.1.4.23 Simple maze

What if programming language doesn't have a graph built-in data type?

How can this graph be represented for processing by route finding software?

We can use a table as shown in *Table 2.1.4.2*.

Vertex 1 is connected to vertex 2. Vertex 2 is connected to vertices 1, 3 and 4, etc. This table can be represented by a one-dimensional array of records with each record storing the corresponding adjacent vertices, the first record 2, the second record 1, 3, 4 and so on as shown in *Table 2.1.4.3*.

Table 2.1.4.4 shows how a `Graph` array of the type `GraphType` would be initialized for the graph in *Figure 2.1.4.24*.

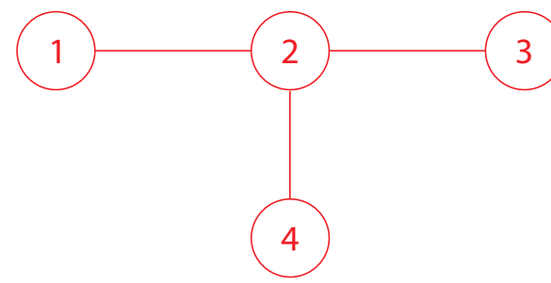


Figure 2.1.4.24 Graph model of simple maze

Vertex	Adjacent vertices
1	2
2	1, 3, 4
3	2
4	2

Table 2.1.4.2 Table showing which vertices are connected

3

Table 2.1.4.3 One-dimensional array of records with each record storing the corresponding adjacent vertices

```
ListType = Array[1..MaxNoOfAdjacentVertices] Of Integer;
VertexType = Record
                 List : ListType;
                 NoOfListVertices : Integer;
             End;
GraphType = Array[1..MaxNoOfVertices] Of VertexType;
```

```
Graph[1].List[1] ← 2; Graph[1].NoOfListVertices ← 1;
Graph[2].List[1] ← 1; Graph[2].List[2] ← 3;
Graph[2].List[3] ← 4; Graph[2].NoOfListVertices ← 3;
Graph[3].List[1] ← 2; Graph[3].NoOfListVertices ← 1;
Graph[4].List[1] ← 2;
Graph[4].NoOfListVertices ← 1;
```

Table 2.1.4.4 Initialisation of Graph array of the type GraphType

Key concept

Graph:
A graph is a series of vertices or nodes joined by connections known as edges or arcs.

Single licence - Abingdon School

Figure 2.1.4.25 and *Figure 2.1.4.26* show examples of more complex mazes.

Did you know?

Facebook graph:
The set of Facebook users and their interconnections make up a graph with approximately 1.5 billion vertices or nodes in 2016.

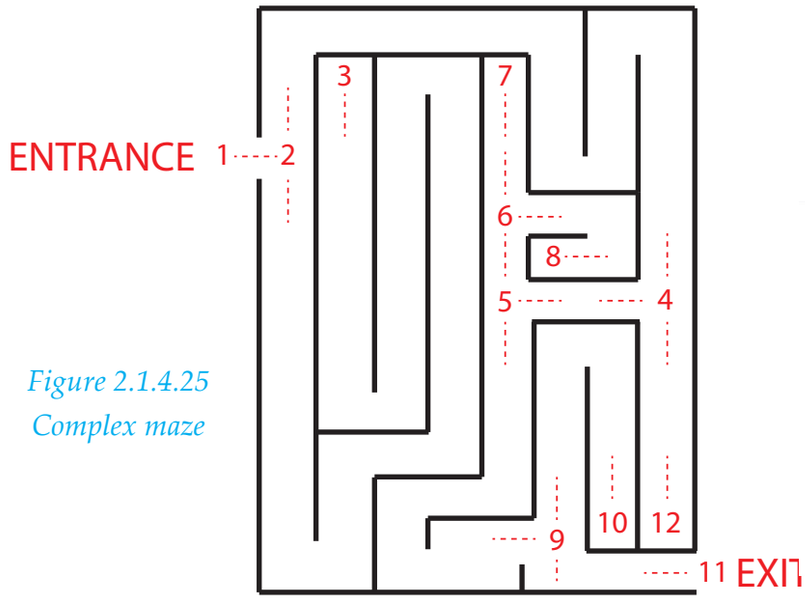


Figure 2.1.4.25 Complex maze

Figure 2.1.4.26 Complex maze

Figure 2.1.4.27 shows the equivalent graph for the maze in *Figure 2.1.4.25*.
Vertices 2, 4, 5, 6 and 9 in *Figure 2.1.4.27* represent junctions. For example, a person navigating this maze has a choice on entering the maze of turning left or right from the point labelled 2 in *Figure 2.1.4.25*.

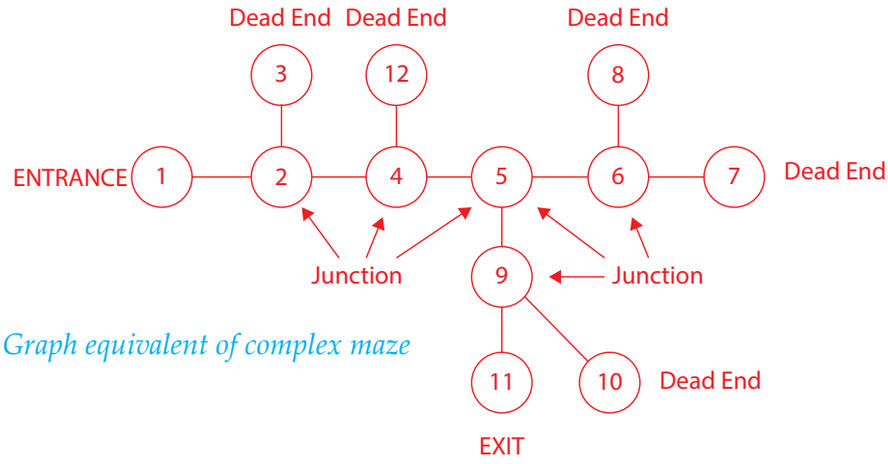


Figure 2.1.4.27 Graph equivalent of complex maze

Questions

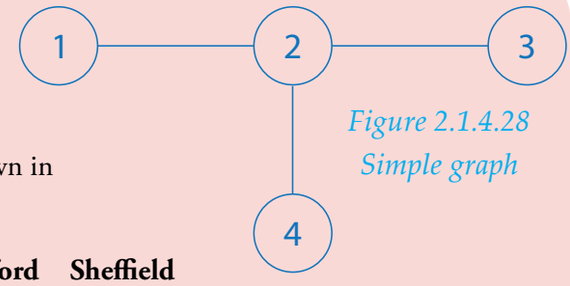


Figure 2.1.4.28 Simple graph

16 On a copy of *Figure 2.1.4.28*, label **one** example of
(a) a vertex or node (b) an edge

17 Draw a labelled graph that represents the information shown in *Table 2.1.4.4* which shows the distances in miles between the given cities.

Table 2.1.4.4 Table of distances between some cities

	London	Manchester	Oxford	Sheffield
London		184	56	160
Manchester	184		144	38
Oxford	56	144		130
Sheffield	160	38	130	

Single licence - Abingdon School

Questions

18 *Figure 2.1.4.29* shows some villages and towns in Wales:

Cardigan, New Quay, Aberaeron, Synod Inn, Lampeter, Newcastle Emlyn.

Draw a graph which shows how these towns/cities are connected by the road network.

Figure 2.1.4.29 Road map showing some Welsh villages/towns (Anquet Technology Ltd © Crown copyright 2016 OS GB map)

19 The following are friends:

- John and Janet
- Janet and Mary
- Mary and Michael
- Michael and John.
- John and Mary

Draw the graph which shows who is connected to whom by friendship.

20 Draw a graph which represents the maze shown in *Figure 2.1.4.30*.

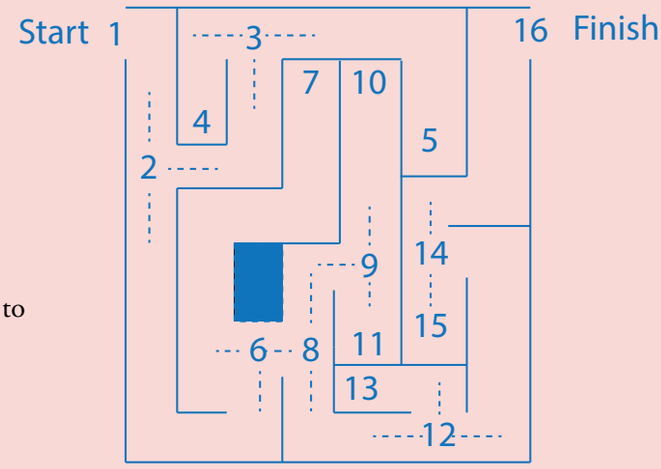


Figure 2.1.4.30 Maze

21 Draw the graph equivalent of the graph data structure, `Graph` shown in *Figure 2.1.4.31*. The data type of `Graph` is `GraphType` which is shown in *Table 2.1.4.2*. The data type of the field `List` is `ListType` which is also shown in *Table 2.1.4.2*.

```
Graph[1].List[1] ← 2;
Graph[1].List[2] ← 3; Graph[1].NoOfListVertices ← 2;
Graph[2].List[1] ← 1;
Graph[2].List[2] ← 3;
Graph[2].List[3] ← 4; Graph[2].NoOfListVertices ← 3;
Graph[3].List[1] ← 1;
Graph[3].List[2] ← 2;
Graph[3].List[3] ← 4; Graph[3].NoOfListVertices ← 3;
Graph[4].List[1] ← 2;
Graph[4].List[2] ← 3; Graph[4].NoOfListVertices ← 2;
```

Figure 2.1.4.31 Initialising graph array

Single licence - Abingdon School

Tree

Linear lists are useful for serially ordered data such as shopping lists but not good for hierarchically ordered data such as the directory structure of a disk. *Figure 2.1.4.32* shows one such directory structure.
Computer Scientists call this hierarchical structure a rooted tree because it resembles the hierarchical structure of trees found in nature, i.e. it has a root, branches and leaves.

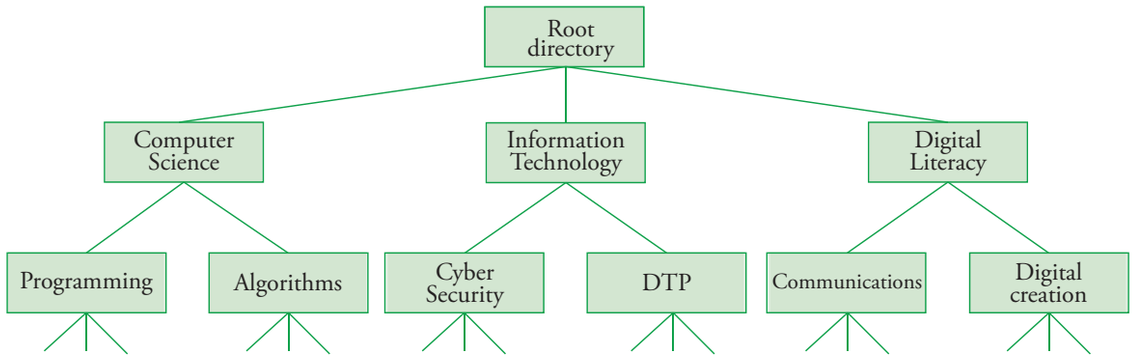


Figure 2.1.4.32 Directory structure

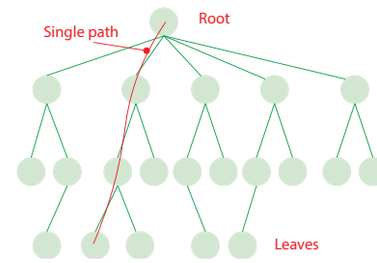


Figure 2.1.4.33 Single path between root element and a leaf element

Rooted trees have the following characteristics

- The element at the top of the hierarchy is the **root**.
- Elements next in the hierarchy are the children of the root.
- Elements next in the hierarchy are the grandchildren of the root, and so on
- Elements at the lowest level of the hierarchy are the **leaves**
- The elements of the tree are connected by **branches**
- There is a single unique path along the branches from the root to any particular element, e.g. *Figure 2.1.4.33*.

Each green circle representing an element in *Figure 2.1.4.33* is called a **node**. We say that a rooted tree is a collection of nodes and branches arranged in hierarchical fashion.
Nodes form "parent-child" relationships:

- each node has exactly one parent (node 1-level up) except the root node which has no parent
- a parent can have several children (nodes 1-level down).

Key concept

Tree:
A rooted tree is a collection of nodes and branches arranged in hierarchical fashion.
Nodes form "parent-child" relationships:

1. Each node has exactly one parent (node 1-level up) except the root node which has no parent
2. A parent can have several children (nodes 1-level down).

A rooted tree is a special case of a general tree defined as a connected undirected graph with no cycles. In the general case, the tree does not have a root.

In fact, a rooted tree is a special case of a general tree defined as a connected undirected graph with no cycles. In the general case, the tree does not have a root.

Trees and rooted trees are covered in depth in *Chapter 2.5.1*.

What if a programming language doesn't have a tree built-in data type?

Processing a tree by computer is not so straightforward when the programming language does not have a tree data type. The programmer is thus required to build the tree from what is available in the language. The building blocks that are used are called the primitives of the language.

Single licence - Abingdon School

If the tree is a rooted tree of a kind in which each node is the parent of at most two other nodes as shown in *Figure 2.1.4.34* then we may use three one-dimensional arrays, as shown in *Figure 2.1.4.35*, to implement a data structure equivalent of this tree abstract data type.

This mapping is achieved by noting that the rooted tree can be structured as shown in *Figure 2.1.4.36*. Each node consists of a left pointer, an element or item and a right pointer. The nodes are numbered left to right from the top down starting with 1 for the root node. A null value of 0 is used for the left and right pointers of the leaf nodes.

A separate root node pointer is required, initialised to 1, in order to find the root entry in each of the three one-dimensional arrays.

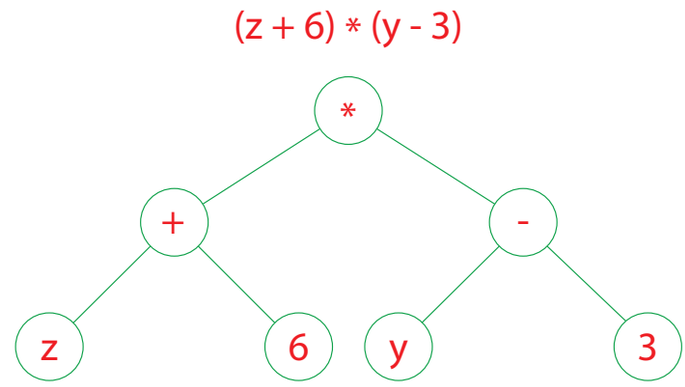


*Figure 2.1.4.34 Rooted tree for the expression (z + 6) * (y - 3)*

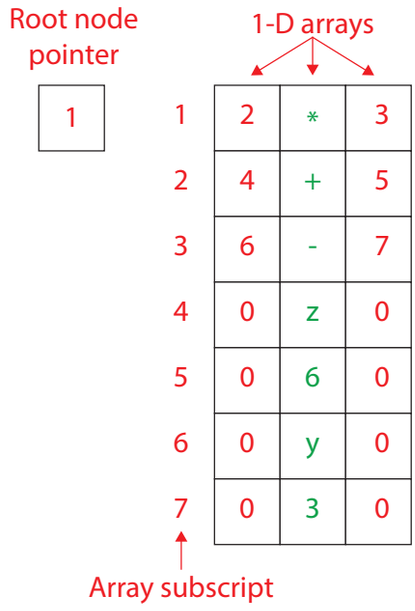


Figure 2.1.4.35 The use of three one-dimensional arrays to represent a rooted tree

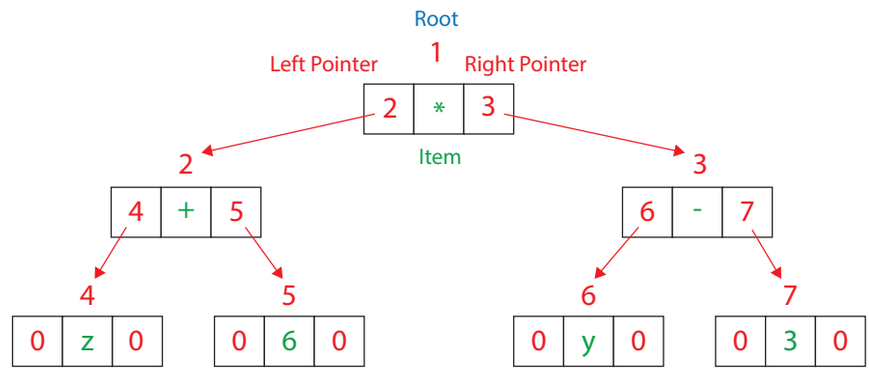


Figure 2.1.4.36 Structuring of each node into a left pointer, item and right pointer

Questions

22 Draw the rooted tree for the expression 5 * y + 4 * z. Multiplication has a higher precedence than addition. Use three one-dimensional arrays to represent this rooted tree.

Single licence - Abingdon School

Hash table

The abstract data type **hash table** is a table of informational elements organised on a row-by-row basis together with a *hash function* and table access procedures. Each row consists of more than one element, one of which is a key which uniquely identifies the row. Collectively, we will refer to the elements making up one row as a *record*.
The *hash function* (or *index calculator*) is applied to a record's key to generate the position to insert the record or the position to find the record when the objective is to retrieve or delete an existing record.
Some things to note:

- The *hash function* takes a record's key and maps it to an integer index which is used to select a specific row of the table
- The table access procedures allow *insertion*, *deletion*, and *retrieval* (looking up) of a record by means of the hash function and the record's key
- A perfect hash function is an ideal function that maps each record's key to a unique table index
- The hash function is based on a randomising algorithm whose goal is to make each generated row index equally likely
- A hash table will start empty, i.e. with no records stored in the table, and be of a predetermined size.

The technique used for locating (and inserting or deleting) a record extremely quickly is called **hashing**.
Tables that use this technique are called **Hash Tables**.

Figure 2.1.4.37 shows a table of 10000 rows, with each row capable of holding a single record, e.g. English word and its French equivalent, and an *index calculator* (hash function).
Whenever it is required to insert a new record in the table, the *index calculator* is used to calculate where it should be placed in the table.
To do this the *index calculator* returns a row index value of the location in which the new record should be stored.

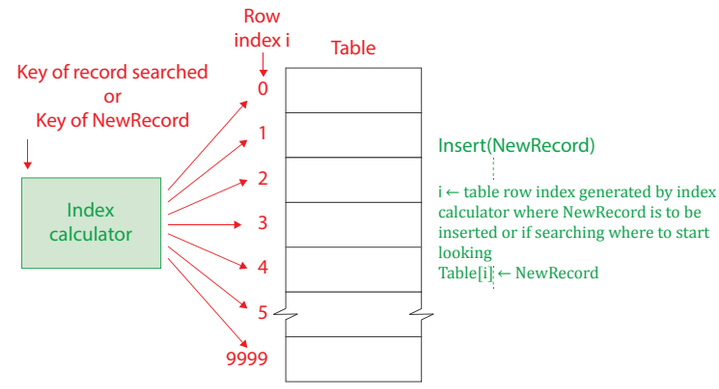


Figure 2.1.4.37 Hash table and the index calculator (hash function)

For example, suppose the English word is 'pen' then the *index calculator* (hashing function) might return value 4.
The insertion operation, `Insert(NewRecord)`, would then store 'pen' and its French equivalent 'plume' in row 4 (assuming the row is not already occupied).
The advantage of this technique is that the insertion operation will be of the order O(1)[1], assuming that the time spent calculating the index is small. This means that it will require constant time to insert, retrieve or delete a record i.e. the time taken will be independent of the size of the table.
Hash tables are covered in depth in *Chapter 2.6.1*.

What if a programming language doesn't have a hash table built-in data type?

The tabular nature of hash tables means that they can be implemented using arrays and records.

Key concept

Hash table:
A hash table is a table of informational elements organised on a row-by-row basis together with a *hash function* and table access procedures.
The *hash function* is applied to a record's key to generate the position to insert the record or to look up the record when the objective is to retrieve or delete an existing record.
The technique used for locating (and inserting or deleting) a record extremely quickly is called hashing.
Tables that use this technique are called hash tables.

1 See Chapter 4.4.3

Single licence - Abingdon School

Questions

23 What is a hash table?
Why would a hash table which stores English, French word pairs be preferred to a linear list of the same word pairs when retrieving the French equivalent of an English word?

Dictionary

We use dictionaries in everyday life when we want to look up the meaning of a word. For example, the word "davit" means "crane". In a word dictionary this would be recorded as "davit - crane" which has the form

key - definition

A word dictionary is a collection of these entries.

The **dictionary abstract data type** is a collection of items in which each item has both a key and a value called a **key, value pair.**

The collection is then

a set of (key, value) pairs

The key is unique and is used to look up its corresponding value.

The data for the dictionary abstract data type takes the form

key : value

for which

- keys are mapped to values, e.g. "France" → "Paris" is the key : value pair "France" : "Paris"
- keys must be comparable, e.g. "Germany" ≠ "France"
- keys must be unique, e.g. only one key "France"

and standard operations are

- insert(key, value), e.g. insert ("Lithuania", "Vilnius") inserts the key : value pair "Lithuania":"Vilnius" into the dictionary (see *Figure 2.1.4.38*)
- find(key), e.g. find("Azerbaijan") - returns the value with key "Azerbaijan" i.e. "Baku", the capital of Azerbaijan, if the dictionary contains such a key, otherwise returns a special value e.g. `No_Such_Key`
- delete(key), e.g. delete("Croatia") removes the key-value pair "Croatia":"Zagreb" whose key is "Croatia"

Items in a dictionary are not stored in any particular order. A dictionary supports fast key lookups.

Table 2.1.4.5 shows a Delphi program which uses the Generics.Collections library's TDictionary data type.

The insert(key, value) operation is called `Add`, e.g.

```
CapitalsOfCountriesDictionary.Add('Lithuania', 'Vilnius')
```

The find(key) operation is implemented in two stages by first checking for the key's presence with `ContainsKey` and then, if present, accessing its value with [key] as follows

```
If CapitalsOfCountriesDictionary.ContainsKey('Azerbaijan')
    Then Writeln(CapitalsOfCountriesDictionary['Azerbaijan'])
    Else Writeln('No key with value Azerbaijan');
```

Key concept

Dictionary:
A dictionary is a collection of items in which each item has both a key and a value and for which
- keys are mapped to values
- keys must be comparable
- keys must be unique

and standard operations are
- insert(key, value)
- find(key)
- delete(key)

Dictionary that maps country to capital, i.e. country → capital:

{"Nigeria":"Abuja",
"Ghana":"Accra", ...,
"Azerbaijan":"Baku", ...,
"Lithuania":"Vilnius", ...,
"Croatia":"Zagreb"}

Figure 2.1.4.38 Dictionary that maps country to its capital

Single licence - Abingdon School

Delphi program	Commentary
`Program DictionaryExample;`	
`{$APPTYPE CONSOLE}`	
`{$R *.res}`	
`Uses`	
`  Generics.Collections;`	
`Type`	
`  TCapitalsOfCountriesDictionary = TDictionary<String, String>;`	Defines a dictionary data type
`Var`	
`  CapitalsOfCountriesDictionary : TCapitalsOfCountriesDictionary;`	Declares a variable of this type
`Begin`	
`  CapitalsOfCountriesDictionary := TCapitalsOfCountriesDictionary.Create;`	Creates a dictionary object
`  CapitalsOfCountriesDictionary.Add('Nigeria','Abuja');`	
`  CapitalsOfCountriesDictionary.Add('Azerbaijan','Baku');`	Inserts key, value pairs
`  CapitalsOfCountriesDictionary.Add('Lithuania','Vilnius');`	
`  CapitalsOfCountriesDictionary.Add('Croatia','Zagreb');`	
`  If CapitalsOfCountriesDictionary.ContainsKey('Azerbaijan')`	Searches for entry with key Azerbaijan
`    Then Writeln(CapitalsOfCountriesDictionary['Azerbaijan'])`	Extracts value of the key : value pair and displays on console
`    Else Writeln('No key with value Azerbaijan');`	
`  Readln;`	
`  CapitalsOfCountriesDictionary.Free;`	Releases memory occupied by object
`End.`	

Table 2.1.4.5 Delphi program that uses the Generics Collections to create and populate a dictionary of country, capital key,value pairs

Table 2.1.4.6 shows a Python program which uses {} to create a dictionary of key, value pairs.

The find(key) operation is implemented in two stages by first checking for the key's presence with `"if "Azerbaijan" in CapitalsOfCountries"` and then, if present, accessing its value with `CapitalsOfCountriesDictionary["Azerbaijan"]` as shown in *Table 2.1.4.6*.

Python program	Commentary
`CapitalsOfCountriesDictionary = {"Nigeria":"Abuja","Ghana":"Accra",` `"Azerbaijan":"Baku","Lithuania":"Vilnius","Croatia":"Zagreb"}`	Creates a dictionary object and inserts key, value pairs
`if "Azerbaijan" in CapitalsOfCountriesDictionary:`	Searches for entry with key Azerbaijan
`    print(CapitalsOfCountriesDictionary["Azerbaijan"])`	Extracts value of the key : value pair and displays on console
`else:`	
`    print("No key with value Azerbaijan")`	

Table 2.1.4.6 Python program that creates and populates a dictionary of country, capital key,value pairs

Table 2.1.4.7 shows a VB.NET program which creates a dictionary of country, capital key, value pairs. The find(key) operation is implemented in two stages by first checking for the key's presence with `"If dictionary.ContainsKey("Azerbaijan") Then".`
If present its value is accessed with `dictionary.Item["Azerbaijan"]` as shown in *Table 2.1.4.7*.

VB.NET program	Commentary
`Sub Main()`	
`        Dim dictionary As New Dictionary(Of String, String)`	Creates a dictionary
`        dictionary.Add("Nigeria","Abuja")`	
`        dictionary.Add("Ghana", "Accra")`	
`        dictionary.Add("Azerbaijan", "Baku")`	Inserts key, value pairs
`        dictionary.Add("Lithuania", "Vilnius")`	
`        dictionary.Add("Croatia", "Zagreb")`	
`        If dictionary.ContainsKey("Azerbaijan") Then`	Searches for entry with key Azerbaijan
`            Console.WriteLine(dictionary.Item("Azerbaijan"))`	Extracts value of the key : value pair and displays on console
`        Else Console.WriteLine("No key with value Azerbaijan")`	
`        End If`	
`        Console.ReadLine()`	
`End Sub`	

Table 2.1.4.7 VB.NET program to create and populate a dictionary of country, capital key,value pairs

For Java and C# versions of the above programs see http://www.educational-computing.co.uk/aqacs/alevelcs.html.

Single licence - Abingdon School

Uses of dictionaries

A dictionary is a searchable collection of key-value items which can be searched by key. This means that it can be used when the data is similarly structured, e.g. a telephone directory. Telephone directories consist of *name, telephone number* pairs and so can be modelled with a dictionary abstract data type and built with a corresponding dictionary data structure.

A dictionary abstract data type or data structure can be used to count the occurrences of words in a story:

- Each dictionary entry is keyed by the word
- The related value is the count
- When entering words into dictionary

  Check if word is already there

  If not, enter it with a value of 1

  If yes, increment its value

The data type of the value part in the *key, value* pair can be any type including a list. This can be useful when creating an index for a book, for example. The list is a list of page numbers on which the key, a string, appears. Routers in networks use look up tables to determine which link to use when forwarding packets - see *Chapter 9.4.3* in Unit 2, page 442.

Figure 2.1.4.39 shows the use of the `LetterCounts` function in the Wolfram programming language to count the occurrences of letters in the Wikipedia article on "Artificial Intelligence". `LetterCounts` uses a dictionary data type (Wolfram calls it an association) of *letter, count* pairs to record the result.

```
In[2]:= LetterCounts[WikipediaData["Artificial intelligence"]]

Out[2]= <|e → 6166, t → 4115, i → 4110, a → 4044, n → 3988, o → 3687, s → 3382, r → 3061, l → 2469, c → 2109, h → 2087, d → 1544,
  m → 1481, u → 1395, g → 1216, p → 1201, f → 1000, b → 795, y → 723, w → 561, v → 469, k → 276, A → 270, I → 214, T → 159,
  x → 120, S → 97, C → 95, M → 94, q → 68, z → 67, R → 60, F → 53, G → 48, H → 47, L → 45, P → 44, N → 43, D → 43, j → 41, B →
  41, E → 36, W → 31, J → 25, O → 24, U → 21, K → 21, V → 7, Y → 4, ö → 4, X → 3, Č → 2, Z → 1|>
```

Figure 2.1.4.39 Wolfram language: Counts how many times each letter occurs in the Wikipedia article on Artificial Intelligence - https://develop.wolframcloud.com

Questions

24 Create a dictionary in the form `{key:value, key:value, ...}` to enable the ASCII code for the character digits '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' to be looked up.

What if a programming language doesn't have a dictionary built-in data type?

It is possible to implement a dictionary abstract data type as a one-dimensional array of records in which the record structure consist of a key field and a value field. Faster insertion and look-up is achieved, however, by using a hash table.

Single licence - Abingdon School

Vector

In mathematics a **vector** is a list of numbers, e.g. (5, 3, 6, 7, 9, 8). The numbers of a vector are called its components, entries or elements. A vector with six elements, each of the number type integer is called a 6-vector over $\mathbb{Z}$.
The vector (2.6, 2.78, 3.4, 1.9) is called a 4-vector over $\mathbb{R}$ because it has four elements, each of the number type real.
The elements of a vector are in some sequence so that we can associate each element with a sequence number as shown in *Figure 2.1.4.40*.
The integers 0, 1, 2, 3 can be interpreted as the number of elements before the current element, e.g. there is just one (1) element before 2.78.
Each element is "ranked" with a rank which is an integer that specifies the number of elements, in the given sequence, before the current element.
If data consist of a list of numbers then we may store these numbers in a one-dimensional array and rank becomes an abstraction of the concept of "index" in arrays. Hence, the name sometimes given to a one-dimensional array is a vector. A one-dimensional array is one way of representing the abstract data type, vector.
We can also use a dictionary to represent a vector,
e.g. the vector (2.6, 2.78, 3.4, 1.9) can be represented by the dictionary

{0 : 2.6, 1 : 2.78, 2: 3.4, 3 : 1.9}

$0 \mapsto 2.6$

$1 \mapsto 2.78$

$2 \mapsto 3.4$

$3 \mapsto 1.9$

Figure 2.1.4.40 Vector and its rank

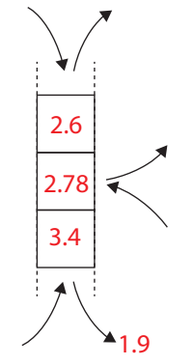


Figure 2.1.4.41 Random access/ insertion/removal of a vector's elements by their "rank"

A vector is a sequence that supports random access/insertion/removal of its elements by their "rank" - see *Figure 2.1.4.41*.

The rank of an element may change whenever the sequence is updated, e.g. when an element is added or removed. This means that the size of the vector may change.
Figure 2.1.4.42 shows four basic vector operations where rank is k and element is a letter of the alphabet

- insertAtRank(k, element)
- removeAtRank(k)
- elementAtRank(k)
- replaceAtRank(k, element)

Unlike arrays, vectors

- automatically increase their capacity, should they need room for more elements
- have built-in methods to accomplish many of the common tasks that otherwise would have to be designed if using arrays.

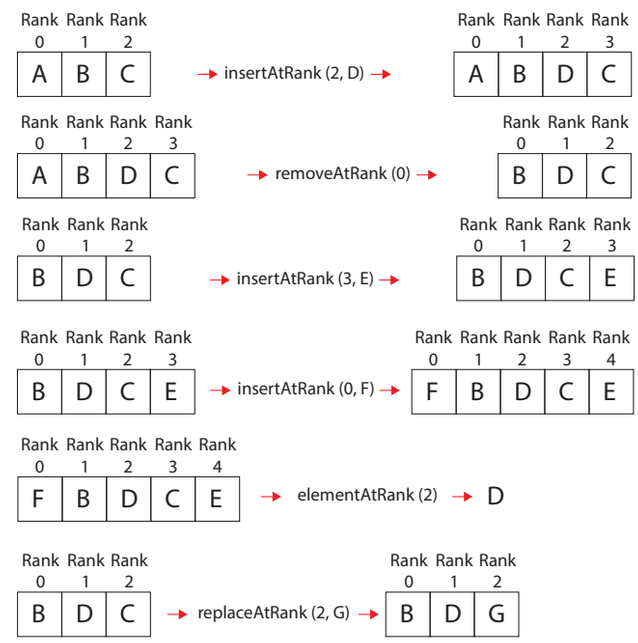


Figure 2.1.4.42 Some vector operations

Remember:

An abstract data type (ADT) is a logical description of how a user views the data and the operations that are allowed on this data without regard to how both data and operations will be implemented.

Single licence - Abingdon School

Figure 2.1.4.43 shows a vector abstract data type (ADT) implemented as a one-dimensional array, `A`.
However, the one-dimensional array implementation is not the only way that a vector ADT may be implemented.

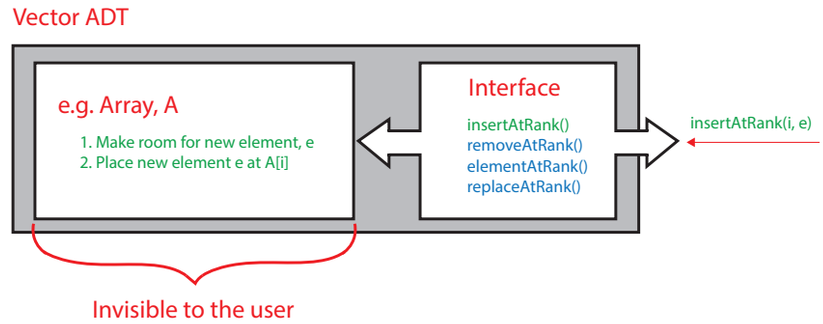


Figure 2.1.4.43 One dimensional array implementation of a vector

Some programming languages have a built-in vector data type or class.
Figure 2.1.4.44 shows a Java program which uses the class Vector to create an instance of this class and then to populate it with four integer objects.

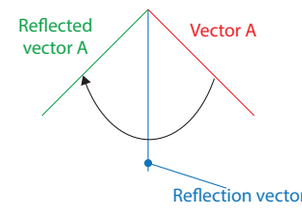
```
import java.util.*;
public class VectorExample {
   public static void main(String[] args) {
      Vector v = new Vector(0);
      System.out.println("Initial size: " + v.size());
      v.addElement(new Integer(1));
      v.addElement(new Integer(2));
      v.addElement(new Integer(3));
      v.addElement(new Integer(4));
      System.out.println("Size after four additions: " + v.size());
    }
}
```

Figure 2.1.4.44 Java program which creates a zero-size vector v initially but then adds four integer elements

Vectors and geometry

Vectors in computer science are useful for representing geometric points. For example, a three-dimensional vector (*x, y, z*) can represent the *x, y, z* coordinates of a point in space.
The concept of a vector originated in geometry and for this reason it is from geometry that the following basic vector operations are derived

- *Add two vectors*: If $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$ are two n-vectors then their sum $X + Y$ is defined as a new vector $(x_1 + y_1, x_2 + y_2, ..., x_n + y_n)$
- *Scale a single vector*: The product of a scalar k and a vector X, written kX, is the vector obtained by multiplying each component of X by k, i.e. $kX = (kx_1, kx_2, .., kx_n)$
- *Dot or scalar product of two vectors*: The dot product of two vectors X and Y denoted by $X \bullet Y$, is defined as $x_1 \bullet y_1 + x_2 \bullet y_2 + \cdots + x_n \bullet y_n$ where $\bullet$ means multiplication. It results in a scalar value.
- *Rotate a single vector*: Defines a new vector rotated through a given angle.
- *Reflect a single vector*: Defines a new vector reflected against another vector.



A *point* in geometry represents a fixed position, but a *vector* in geometry represents a direction and a magnitude (for example, velocity or acceleration). Thus, the endpoints of a line segment are points but their difference is a vector; that is, the direction and length of that line segment.
Figure 2.1.4.45 shows three vectors:

- a vector `aVector1` with endpoints 0,0 and 2,3,
- a vector `aVector2` with endpoints 0,0 and 4,4
- a vector `aVector3` which is the result of adding `aVector1` and `aVector2`.

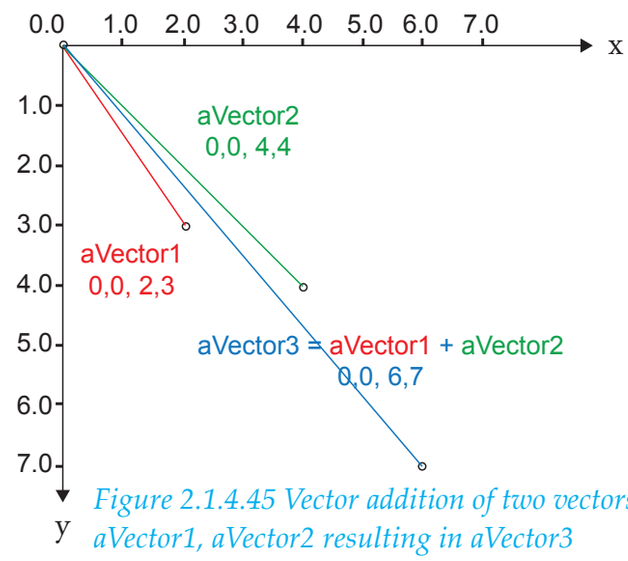


Figure 2.1.4.45 Vector addition of two vectors aVector1, aVector2 resulting in aVector3

Single licence - Abingdon School

When two vectors are added their elements at rank 0 are added and the result placed at rank 0 in the result vector, then their elements at rank 1, and so on as shown below

aVector1 + aVector2 = aVector3

(0,0,2,3) + (0,0,4,4) = (0,0,6,7)

Figure 2.1.4.46 shows a Delphi program which adds two vectors.

Delphi has a point class called `TPointF` and a vector class called `TVector`. The program creates a point at 2, 3 and then a vector, `aVector1` with endpoints 0, 0 and 2, 3.

`TVector.Create` always uses the endpoint 0, 0 and so this does not need to be specified in the method call

`TVector.Create(2,3).`

The user just has to supply the other endpoint coordinates.

Figure 2.1.4.47 shows a VB.NET program which performs a similar addition of two vectors.

```
Program VectorExample;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils, System.Types;
Var
  aVector1, aVector2, aVector3: TVector;
  aPoint1, aPoint2: TPointF;
  aLength1, aLength2: Single;
Begin
  aPoint1 := TPointF.Create(2,3);
  aVector1 := TVector.Create(aPoint1);
  aPoint2 := TPointF.Create(4,4);
  aVector2 := TVector.Create(aPoint2);
  writeln(aVector1.X, aVector1.Y);
  writeln(aVector2.X, aVector2.Y);
  aLength1:=aVector1.Length;
  aLength2:=aVector2.Length;
  writeln(Format('%2.2f',[aLength1]));
  writeln(Format('%2.2f',[aLength2]));
  aVector3 := aVector1 + aVector2;
  writeln(Format('X = %2.2f Y = %2.2f', [aVector3.X, aVector3.Y]));
  Readln;
End.
```

Figure 2.1.4.46 Delphi program which demonstrates vector addition of two vectors aVector1, aVector2 resulting in aVector3

```
Imports System.Windows
Module Module1
    Sub Main()
        Dim vector1 As New Vector(2, 3)
        Dim vector2 As New Vector(4, 4)
        Dim vectorResult As New Vector()
        Dim x As Double = vector1.X
        Dim y As Double = vector1.Y
        x = vector2.X
        y = vector2.Y
        vectorResult = vector1 + vector2
        Console.WriteLine(vectorResult.X)
        Console.WriteLine(vectorResult.Y)
        Console.ReadLine()
    End Sub
End Module
```

Figure 2.1.4.47 VB.NET program which demonstrates vector addition of two vectors vector1, vector2 resulting in vectorResult

Key concept

Vector:

In mathematics a vector is a list of numbers, e.g. (5, 3, 6, 7, 9, 8).

In computer science, a vector is a sequence that supports random access/insertion/removal of its elements by their "rank".

Each element is "ranked" with a rank which is an integer that specifies the number of elements, in the given sequence, before the current element.

Unlike static arrays, vectors automatically increase their capacity, should they need room for more elements.

Vectors have built-in methods to accomplish many of the common tasks that otherwise would have to be designed if using arrays.

Vectors are useful for representing geometric points and geometric vectors. The endpoints of a line segment are points but their difference is a vector; that is, the direction and length of that line segment.

Questions

25 What is the vector that results from adding vectors (1, 2, 4, 9) and (16, 25, 36, 49)?

26 What is the integer value that results from the dot product of these vectors?

Single licence - Abingdon School

What if a programming language doesn't have a vector built-in data type?

It is possible to use a one-dimensional array, a list or a dictionary to implement a vector abstract data type.

For example in Python, the vector v may be defined using the Python dictionary data type as follows

```
v = {0 : 0, 1 : 1, 2 : 4, 3 : 9}
```

and used as follows `print(v[3])` to print 9 to the screen.

Questions

27 The vector (0,0,0, 4,5,6) records the endpoint x, y, z coordinates of a line segment in 3-dimensional space with respect to an origin with coordinates 0,0,0, i.e. the line segment has end coordinates 0,0,0 and 4,5,6. Another line segment is also recorded in a vector (0,0,0, 7,8,9).

(a) The vector addition of these two vectors is itself a 6-vector of coordinates defining a new line segment. What is this new vector?

(b) The vector (0,0,0, 4,5,6) is increased in length by a scaling factor of 1.5. What is the new vector created by this scaling?

(c) The dot or scalar product of two n-vectors, a and b, is defined as

$$a \bullet b = a_0 \times b_0 + a_1 \times b_1 + a_2 \times b_2 + a_3 \times b_3 + \cdots + a_n \times b_n$$

If a = (0,0,0, 4,5,6) and b = (0,0,0, 7, 8, 9), what is the 6-vector dot product a • b?

28 An 8-bit secret key 10110001 is the combination number of a safe. By representing this as the 8-vector over *GF*(2), v = (1, 0, 1, 1, 0, 0, 0, 1), it is possible using vector addition and a randomly generated 8-vector over *GF*(2) to produce two new vectors neither of which on their own are sufficient to derive the combination number.

GF(2) means binary values and the following binary operations only apply, e.g. 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 0 - see the GF(2) calculator at http://www.ee.unb.ca/cgi-bin/tervo/calc.pl.

Another 8-bit vector over GF(2), v_A is generated randomly according to a uniform distribution by tossing a coin. A new vector v_B is obtained by adding v to the vector produced by scaling v_A, i.e.

$$v_B = v + v_A$$

If v_A is (1, 0, 0, 1, 1, 1, 1, 0) what is v_B?

The vector v_A is given to Alice and the vector v_B is given to Bob.

Explain how to recover the original vector v and therefore the secret key from a knowledge of Alice's and Bob's vectors.

29 Explain how to split an n-bit secret among three people, Alice, Bob and Mary, so that each receives an n-vector which requires the other two person's n-vectors for successful regeneration of the n-bit secret key.

30 Calculate the dot product of the the following 7-vectors over *GF*(2) where *GF*(2) means binary values only for which the following operations apply 0 • 0 = 0, 0 • 1 = 0, 1 • 0 = 0, 1 • 1 = 1 - see the GF(2) calculator at http://www.ee.unb.ca/cgi-bin/tervo/calc.pl.

(a) (1, 1, 1, 1, 1, 1, 1) and (1, 0, 0, 1, 1, 1, 0)

(b) (1, 1, 1, 1, 1, 1, 1) and (1, 0, 0, 1, 1, 1, 1)

(c) Suppose that the second vectors in (a) and (b) represent 7-bit data to be sent via a telecommunications link to a receiver. An eighth bit is appended to each 7-bit datum to make the number of total 1's an even number. Explain how a vector operation on the 7-bit datum could assist in this task.

Single licence - Abingdon School

In this chapter you have covered:

- The concept and uses of a:
  - queue
  - stack
  - graph
  - tree
  - hash table
  - dictionary
  - vector
- Using these abstract data types and their equivalent data structures in simple contexts
- Some methods for representing these when a programming language does not support these as built-in types.

Single licence - Abingdon School

2 Fundamentals of data structures

2.1 Data structures and abstract data types

Learning objectives:

- *Be able to distinguish between static and dynamic structures and compare their uses, as well as explaining the advantages and disadvantages of each.*

2.1.4b Static and dynamic structures

Static data structures

Static data structures are of fixed size. The amount of memory, once allocated, cannot change at runtime.

An example of a static data structure is a static array. The size of a static array is fixed at **design time** when the array is declared to be of so many elements.

For example,

```
Var
  MyArray : Array [0..100] Of Integer
```

defines (using Pascal/Delphi) a variable, `MyArray`, which is a fixed size array for storing integers, 101 in total.

The space for this array is obtained when the object form of the program is loaded into main memory ready for execution.

Key concept

Static data structure:
Static data structures are of fixed size. The amount of memory, once allocated, cannot change at runtime.

Dynamic data structures

A dynamic data structure, on the other hand, is one that can grow or shrink at runtime as needed to contain the data that is required to be stored. That is, new storage can be allocated in main memory when it's needed and discarded when it is no longer required.

The allocation and de-allocation of storage takes place **at runtime** after the program has been loaded and has started executing.

Dynamic data structures generally consist of at least some simple data storage along with a link to the next element in the structure. These links are often called pointers, or references. Pointers were introduced in *Chapter 1.1.1*.

The links may be exposed so that a programmer can directly program with them to insert an element or remove a data element. Alternatively, they may be hidden from the programmer. In which case, the programmer grows and shrinks the structure by using insert/add and delete primitives.

A list is an example of a dynamic data structure. List processing is covered in *Chapter 12.3.1* in *Unit 2*.

A dynamic array is another example of a dynamic data structure. Dynamic arrays are usually implemented as lists.

Key concept

Dynamic data structure:
A dynamic data structure is one that can grow or shrink at runtime as needed to contain the data that is required to be stored.

Single licence - Abingdon School

Uses of static data structures, advantages and disadvantages

Static data structures are used when the number of data elements that will need to be stored is known. Static arrays fit the bill well. For example, static arrays are used for look-up tables of data whose number of elements change infrequently and therefore this number is known at design/compile time.

Advantages

Access time to each array element is time complexity O(1) because a static array is a random access data structure. It takes the same constant amount of time to access any element of the array. Random access means that access time is independent of the order in which elements are accessed.

Disadvantages

They are essentially fixed-size so once the structure is full it is not possible to find memory space for any more data without resorting to deleting or overwriting existing stored data.

To avoid the potential of this occurring the size of the structure is made generously large in size. The consequence of this strategy is that some memory space in the structure is inevitably wasted because it will not get used.

Uses of dynamic data structures, advantages and disadvantages

Dynamic data structures are used where the number of data elements will change at runtime. Nonlinear structures such as trees used in parsing expressions, syntax analysis, sorting, or indexing can grow in unpredictable ways and therefore benefit from using a dynamic data structure. List processing by its very nature involves lists of data that grow and shrink in unpredictable ways such as in simulations or the processing of streamed data.

Advantages

This flexibility to grow and shrink in size at runtime is an advantage over static data structures. Dynamic data structures avoid wasting memory space in the case of static data structures described above. Dynamic data structures take only as much memory space as they need at runtime.

Disadvantages

Each data element in a dynamic data structure has the storage overhead of at least one link field, a pointer which may take 32 or 64 bits of storage. If the dynamic data structure is nonlinear (e.g. binary tree) then more than one pointer may be required. Access time to its elements will also take longer than a static data structure such as an array because pointers have to be dereferenced, i.e. followed to locate the next element. This means one memory read access to get the address of the datum and then another to access the datum. If a chain of links have to be followed to get to the required datum then more memory accesses will be necessary. Insertions and deletions by definition will grow or shrink the structure. Both operations consume time because links have to be adjusted. In a static data structure, the pre-allocated storage space is used at a cost of O(1) time when a new datum needs to be inserted and allocated space freed by simply overwriting it with a null value, again at O(1) time cost.

Questions

1. Data structures may be either static or dynamic. What is meant by this and give **one** example of each?
2. When would it be appropriate to use (a) a static data structure (b) a dynamic data structure?
3. State **one** advantage and **one** disadvantage of each of the following over the other
   static data structure dynamic data structure

Single licence - Abingdon School

In this chapter you have covered:

- *Static and dynamic structures sufficiently to be able to distinguish one from the other*
- *Comparing their uses, as well as explaining the advantages and disadvantages of each*

Single licence - Abingdon School

2 Fundamentals of data structures

2.2 Queues

Learning objectives:

- *Be able to describe and apply the following to linear queues, circular queues and priority queues:*
  - *add an item*
  - *remove an item*
  - *test for an empty queue*
  - *test for a full queue*

2.2.1 Queues

Linear queue

Figure 2.2.1.1 shows a queue of eight cells. We say, therefore, that this queue has queue size 8.

The figure also shows two pointers, F and R.

F points to the cell containing the item which has been longest in the queue.

R points to the cell containing the item which has been in the queue for the least amount of time.

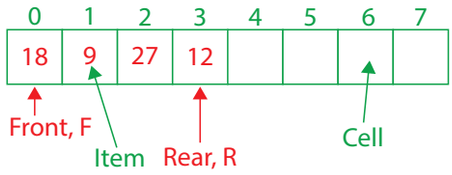


Figure 2.2.1.1 Linear queue storing integer data items 18, 9, 27, 12

Adding an item to a non-empty linear queue

The cells have been labelled 0, 1, 2, ..., 5, 6, 7.

Therefore, for the given state of the queue, F = 0 and R = 3.

An item is added to the queue's rear.

The first free slot in the queue is the cell labelled 4.

Therefore, before an item can be added to the queue shown in *Figure 2.2.1.1*, the rear pointer R must be incremented by 1.

Treating the queue as an array, `Queue`, the item to be added, `Item`, may be assigned to queue cell given by `Queue[R]` as shown in pseudo-code in *Figure 2.2.1.3*.

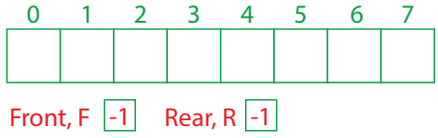


Figure 2.2.1.2 Am empty linear queue

Figure 2.2.1.2 shows the state of the queue when it is empty.

The pointers F and R are set to -1 because, in this example, -1 has been chosen as the value of the null pointer, i.e. the pointer value that doesn't point to anything.

Testing for an empty linear queue

With -1 as the null pointer value, a test for the empty linear queue consists of checking if F = -1 and R = -1 are both true[1].

Adding an item to an empty linear queue

Adding an item to an empty queue (F = -1, R = -1) requires a slightly different approach.

Both F and R must first be set to 0, the label of the first cell, then the item, `Item`, may be assigned to queue cell `Queue[R]` as shown in *Figure 2.2.1.3*.

Testing for a full linear queue

The rear pointer grows rightward along the queue as items are added. Eventually it will reach the value 7 for the given example queue. The label 7 labels the last cell in the queue and if this is now occupied any attempt to add another item will fail because there is no cell labelled 8 to receive the item. A simple test for a full queue is thus to check if R + 1 = QueueSize is true - see *Figure 2.2.1.3*.

Initialise empty queue:

```
F ← -1
R ← -1
```

Add an item to linear queue:

```
If (F = -1) And (R = -1)
  Then
    F ← 0
    R ← 0
    Queue[R] ← Item
  Else
    If R + 1 = QueueSize
      Then Output 'Queue full'
      Else
        R ← R + 1
        Queue[R] ← Item
    EndIf
EndIf
```

Figure 2.2.1.3 Pseudo-code to initialise and add an item to a linear queue

1 Checking either (F = -1) or (R = -1) is true is sufficient.

Single licence - Abingdon School

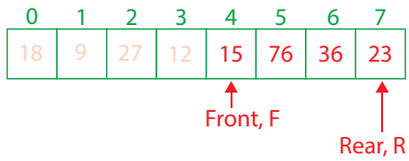


Figure 2.2.1.4 State of linear queue after some items added and existing items removed

Conventional linear queues suffer a problem when the rear pointer reaches its limit. No more items can be added to the queue until it is emptied. *Figure 2.2.1.4* shows the queue after integer items 15, 76, 36 and 23 have been added in that order and integer items 18, 9, 27 and 12 have been removed (shown feint). The queue has unoccupied cells but these are currently inaccessible.

Removing an item from a non-empty linear queue

Items are removed from the front of the queue. The front pointer points to the front of the queue. Therefore, the operation `Item ← Queue[F]` will copy the item which is currently at the front of the queue to the variable Item. This item must now be made inaccessible in the queue.

To do this we need to consider two cases:

- The item copied was the last item in the queue (F = R)
- The item copied was not the last item in the queue

In the first case, F = R, F and R must be set to -1 as shown in *Figure 2.2.1.5*.

In the second case, F is incremented by 1 as shown in *Figure 2.2.1.5*.

Clearly if the queue is already empty (F = -1 and R = -1) it is not possible to remove an item and the message 'Queue empty' is output as shown in *Figure 2.2.1.5*.

```
Remove an item from linear queue:
  If (F = -1) And (R = -1)
    Then Output 'Queue empty'
    Else
      Item ← Queue[F]
      If F = R
        Then
          F ← -1
          R ← -1
        Else F ← F + 1
      EndIf
  EndIf
```

Figure 2.2.1.5 Removing an item from a linear queue

Programming task

1. Implement a linear queue in a programming language with which you are familiar. Your linear queue should be capable of storing up to 8 integers and displaying the state of the queue including its front and rear pointer values. Your program should support the following queue operations:
   - add an item
   - remove an item
   - test for an empty queue
   - test for a full queue.

Questions

1. *Figure 2.2.1.6* shows a linear queue with an alternative cell labelling. The pseudo-code in *Figure 2.2.1.3* will need changing to reflect this new labelling.
   State **three** necessary changes to this pseudo-code.

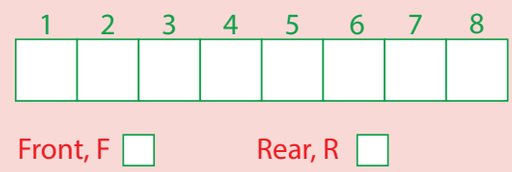


Figure 2.2.1.6 Linear queue with cell labelling starting at 1

Single licence - Abingdon School

Circular queue

The linear queue problem described above when the rear pointer reaches its limit can be solved by allowing the rear pointer to wraparound the end of the queue so that when an item is added, the rear pointer value is changed as follows

$$0 \rightarrow 1,\ 1 \rightarrow 2,\ 2 \rightarrow 3,\ 3 \rightarrow 4,\ 4 \rightarrow 5,\ 5 \rightarrow 6,\ 6 \rightarrow 7,\ 7 \rightarrow 0$$

The transition 7 → 0 is called **wraparound** and the queue is called a **circular queue**. The following expression will generate these transitions

```
(R + 1) Mod QueueSize
```

The operator Mod calculates the remainder after dividing (R + 1) by QueueSize , e.g. R = 7, QueueSize = 8, (7 + 1) Mod 8 = 0.

The front pointer is changed in a similar manner when an item is removed, i.e.

$$0 \rightarrow 1,\ 1 \rightarrow 2,\ 2 \rightarrow 3,\ 3 \rightarrow 4,\ 4 \rightarrow 5,\ 5 \rightarrow 6,\ 6 \rightarrow 7,\ 7 \rightarrow 0$$

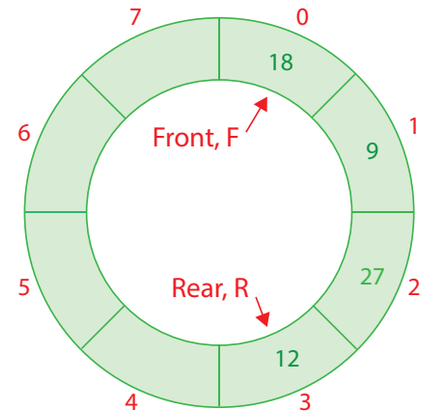


Figure 2.2.1.7 Circular queue containing 8 cells

Figure 2.2.1.7 shows a circular queue containing eight cells arranged for wraparound. Its queue size is therefore 8. This circular queue has gone from the empty state (F = -1 and R = -1) to the state F = 0 and R = 3 after four integer items 18, 9, 27 and 12 were added, in that order.

Figure 2.2.1.8 shows the state of the circular queue after integer items 15, 76, 36 and 23 were added, in that order, and integer items 18, 9, 27 and 12 were removed (shown feint).

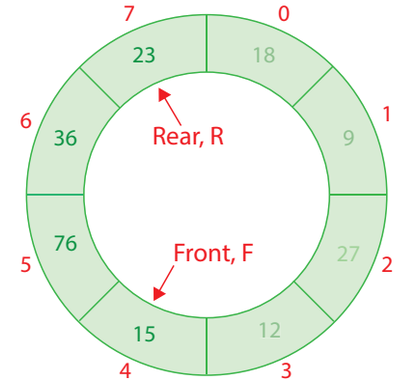


Figure 2.2.1.8 Circular queue after items are added and items are removed

Figure 2.2.1.9 shows the state of the queue after integer item 34 was added. The rear pointer R has wrapped around from 7 to 0, i.e. its value is now 0.

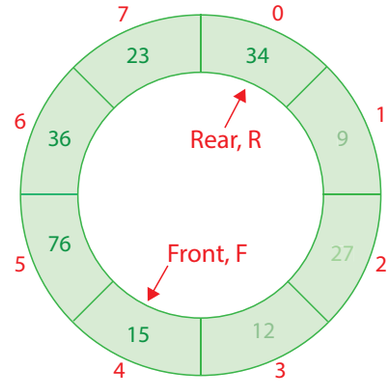


Figure 2.2.1.9 Circular queue showing wraparound

Figure 2.2.1.10 shows the state of the circular queue after both front and rear pointers have wrapped around. The front pointer F is now 0, brought about by removing 15, 76, 36 and 23 (shown feint), in turn, from the front of the queue. The rear pointer R is 2, brought about by adding 56 and 98, in turn, to the rear of the queue. No more items can be added to the circular queue when all cells are occupied, i.e. the following is true

```
(R + 1) Mod QueueSize = F
```

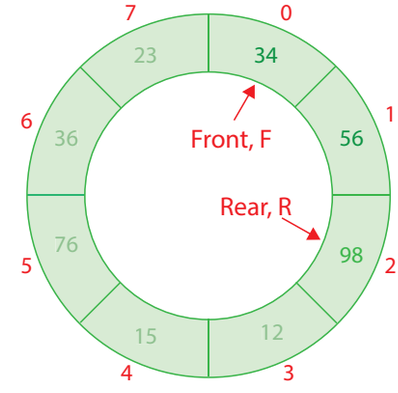


Figure 2.2.1.10 Circular queue showing wraparound of both front and rear pointers

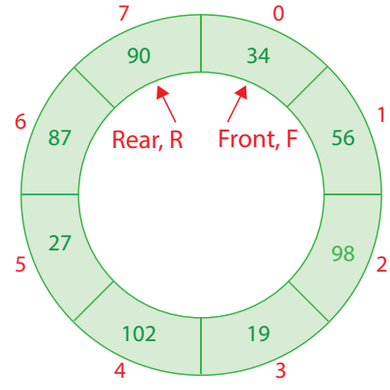


Figure 2.2.1.11 Full circular queue

Figure 2.2.1.11 shows an example of a full circular queue brought about by adding, in turn, 19, 102, 27, 87, 90 to the rear of the circular queue shown in *Figure 2.2.1.10*.

Information

Another solution in the case of a linear queue which is not full but the rear pointer has reached its limit is to shuffle all the items leftward. However, this requires a substantial amount of processing when the list size is large.

Single licence - Abingdon School

Testing for a full circular queue

We need to check if (R + 1) Mod QueueSize = F is true - see *Figure 2.2.1.12*.

Testing for an empty circular queue

With -1 as the null pointer value, a test for the empty circular queue just consists of checking if F = -1 and R = -1 are both true - see *Figure 2.2.1.12*.

Adding an item to a circular queue

Figure 2.2.1.12 shows pseudo-code to initialise an empty circular queue and to add an item to a circular queue. If the circular queue is empty we set F and R to 0 and assign the item to Queue[R]. If the circular queue is non-empty and not full we change R to (R + 1) Mod QueueSize and assign the item to Queue[R].

```
Initialise:
    F ← -1
    R ← -1
Add an item to circular queue:
  If (F = -1) And (R = -1)
    Then
      F ← 0
      R ← 0
      Queue[R] ← Item
    Else
      If (R + 1) Mod QueueSize = F
        Then Output 'Queue full'
        Else
          R ← (R + 1) Mod QueueSize
          Queue[R] ← Item
      EndIf
  EndIf
```

Figure 2.2.1.12 Pseudo-code to add an item to a circular queue and initialise an empty queue

Removing an item from a non-empty circular queue

Items are removed from the front of the queue.

The front pointer points to the front of the queue.

Therefore, the operation `Item ← Queue[F]` in *Figure 2.2.1.13* will copy the item which is currently at the front of the queue. This item must now be made inaccessible in the queue.

To do this we need to consider two cases:

- The item copied was the last item in the queue (F = R)
- The item copied was not the last item in the queue

In the first case, F = R, F and R must be set to -1 as shown in *Figure 2.2.1.13*.

In the second case, F is incremented by 1 and Modded with QueueSize as shown in *Figure 2.2.1.13*.

```
Remove an item from circular queue:
If (F = -1) And (R = -1)
  Then Output 'Queue empty'
  Else
    Item ← Queue[F]
    If F = R
      Then
        F ← -1
        R ← -1
      Else F ← (F + 1) Mod QueueSize
    EndIf
EndIf
```

Figure 2.2.1.13 Removing an item from a circular queue

Clearly if the circular queue is already empty (F = -1 and R = -1) it is not possible to remove an item and the message 'Queue empty' is output as shown in *Figure 2.2.1.13*.

Programming task

2 Implement a circular queue in a programming language with which you are familiar. Your circular queue should be capable of storing up to 8 integers and displaying the state of the queue including its front and rear pointer values. Your program should support the following queue operations:

- add an item
- remove an item
- test for an empty queue
- test for a full queue.

Single licence - Abingdon School

Questions

2 *Figure 2.2.1.14* shows an empty circular queue which can store up to 8 integers.

Five integers are added to the circular queue.

(a) What values are now stored in the F and R?

Three integers are now removed from the circular queue

(b) What values are now stored in the F and R?

Five more integers are added to the circular queue.

(c) What values are now stored in the F and R?

Six integers are now removed from the circular queue.

(d) What values are now stored in the F and R?

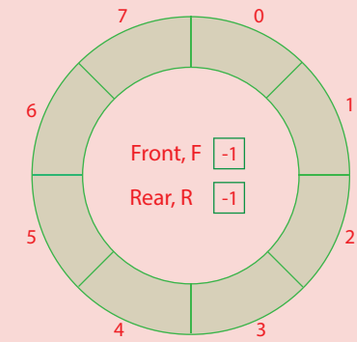


Figure 2.2.1.14 Circular queue

Priority queue

A priority queue is not a queue as defined in *Chapter 2.1.4*.

In a conventional queue, the item which is removed is the one which has been in the queue the longest.

In a priority queue, the item with the highest priority is always removed first and this item may not be the item which has been in the queue the longest. If more than one item at this priority exists, the item chosen is the one that has been longest in the queue.

In a priority queue each item has a priority value and a data value.

One way that a priority queue could be implemented is with an ordered list. This is not the most efficient way time-wise but it is adequate for priority queues of small size.

If the priority queue is modelled by an ordered list, an item is added to the queue by inserting it after any other item in the list with the same priority but before items of a lesser priority.

An item is removed from the list by removing the first item in the list. This first item has the highest priority in the list by design. If it is not the only item with the highest priority it will be the oldest.

Figure 2.2.1.15 shows a priority queue before and after removal of the item with priority 10 and data value 'Smith'.

The figure next shows a datum 'Brown' with priority 4 inserted into the priority queue followed by a datum 'Minn' with priority 5.

Key concept

Priority queue:

In a priority queue each item has a priority value and a data value.

A priority queue is one in which the items are arranged in order of priority.

An item is added to the queue by inserting it after any other item with the same priority but before items of a lesser priority.

The removal operation selects the item with the highest priority. If more than one at this priority exists, the item chosen is the one that has been longest in the queue.

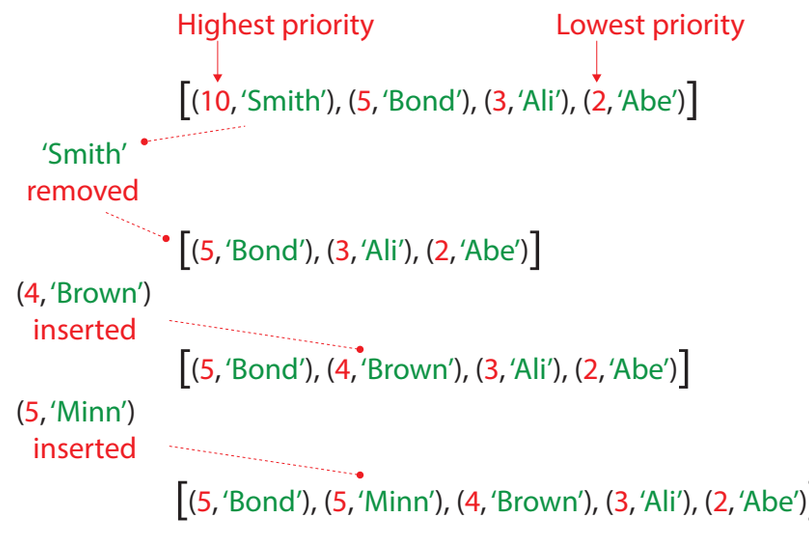


Figure 2.2.1.15 Adding and removing an item from a priority queue

Single licence - Abingdon School

Information

A heap is a much better way of modelling a priority queue.
A heap is a balanced tree in which both children of the root are also heaps and their roots each have key values less than or equal to their parent. If the value used for organising the heap is the priority of an element, the heap guarantees that a highest priority node is always at the root. With a little extra work this structure can be made to act as a priority queue with a time efficiency which is much better than an ordered list.

A list is implemented as a dynamic array usually. A dynamic array is one that automatically acquires the storage space it needs. If it needs another cell, it simply adds this cell to the existing collection of cells. A dynamic array's size is only limited by the available memory in the machine. If the priority queue is implemented as an ordered list then the same is true for the priority queue.

The empty priority queue is an empty list [ ]. testing for an empty list simply consists of testing for [ ].

Questions

3 People seeking medical attention at a hospital accident and emergency department have their name and details placed in a queue in the department's computer system at a position determined by the seriousness of their case. Cases of equal seriousness are treated in their order of arrival.
What type of queue would the computer system use to handle the cases in the manner described?

In this chapter you have covered:

- Describing and applying the following to linear queues, circular queues and priority queues:
  - add an item
  - remove an item
  - test for an empty queue
  - test for a full queue

Single licence - Abingdon School

2 Fundamentals of data structures

2.3 Stacks

Learning objectives:

- *Be able to describe and apply the following operations:*
  - *push*
  - *pop*
  - *peek or top*
  - *test for empty stack*
  - *test for full stack*

2.3.1 Stacks

A stack is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the **top** or **top of stack**. The end opposite the top is known as the base or stack base.

Figure 2.3.1.1 shows a stack with 200 storage cells. Each cell may store an integer and the cells are labelled 0, 1, 2, ..., 199.

Three integers 3, 10, and 5 are currently stored.

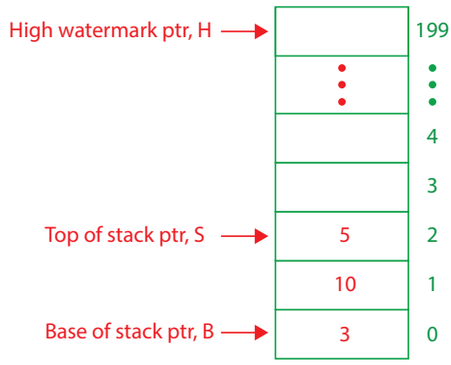


Figure 2.3.1.1 Stack with 200 cells

The integer 3 has been on the stack the longest, 10 the next longest and 5 the least longest.

The base of stack pointer B has value 0.

The top of stack pointer S has the value 2, currently.

The high watermark pointer H has the value 199.

Integers stored in the stack that are closer to the base represent those that have been in the stack the longest.

The most recently added integer is the one that is on top. It is the integer on top that is removed first.

In general, items can only be added to a stack at the top. This is known as **pushing** (an item onto the stack).

Items can only be taken off a stack at the top. This is known as **popping** (an item from or off the stack) and the operation is called **pop**.

Push operation

Figure 2.3.1.2 shows pseudo-code to initialise an empty stack by setting S, its top of stack pointer to -1 and pseudo-code for the operation push.

As long as the value of the top of stack pointer S is less than the value of the high watermark pointer H (199 in this example), then S may be incremented and the item's value assigned to Stack[S].

Initialise empty stack:

```
S ← -1
```

Push an item onto the stack:

```
If S < H
  Then
    Begin
      S ← S + 1
      Stack[S] ← Item
    End
  Else Output 'Stack Full'
EndIf
```

Figure 2.3.1.2 Pseudo-code to initialise an empty stack and to push an item onto a stack

Questions

1. Given the state of the stack shown in *Figure 2.3.1.1*, what is the value of S, the top of stack pointer, after 7, 8 and 9 are pushed on the stack in that order?

Single licence - Abingdon School

To pop an item off a stack:

```
If S >= B
  Then
    Begin
      Item ← Stack[S]
      S ← S - 1
    End
  Else Output 'Stack Empty'
```

Figure 2.3.1.3 Pseudo-code to pop an item off a stack

Pop operation

Figure 2.3.1.3 shows pseudo-code for the operation pop.

As long as the value of the top of stack pointer S is greater than or equal to the value of the base of stack pointer B (0 in this example), the datum stored at Stack[S] may be copied to Item and then S may be decremented.

Questions

2 Given the state of the stack shown in *Figure 2.3.1.1*, what is the value of S, the top of stack pointer, after the stack is popped three times?

Test for empty stack

The empty stack state can be indicated by choosing a value for the top of stack pointer which is less than the lowest valid value for labelling stack cells/locations, e.g. -1. The test for empty stack then becomes an examination of the current value of the stack pointer to check if it is in the valid range of values. In the pseudo-code in *Figure 2.3.1.3* this is done by an If statement with condition S >= B, i.e. if S's value is greater or equal to the lowest valid value then the stack is not empty otherwise it is empty. The lowest valid value is stored in the stack base pointer B.

Test for full stack

The full stack state can be detected by comparing the current value of the top of stack pointer with the highest valid value. In the pseudo-code in *Figure 2.3.1.2* this is done by an If statement with condition S < H, i.e. if S's value is less than the highest valid value then the stack is not full otherwise it is full. The highest valid value is stored in the stack high watermark pointer H.

Peek or top operation

Peek or top returns the value of the top element without removing it. In the example stack this is given by Stack[S].

Programming task

1 Implement a stack in a programming language with which you are familiar. Your stack should be capable of storing up to 10 integers and displaying the state of the stack including its top of stack pointer value. Your program should support the following stack operations:

- push
- pop
- peek
- test for empty stack
- test for full stack.

Single licence - Abingdon School

Questions

3 A stack consists of seven storage cells, a top of stack pointer S, a base of stack pointer B, and a high watermark pointer, H.

The following stack operations are supported

- Procedure Push (Item) - pushes the Item onto the stack
- Function Pop - removes and returns the item which is currently on the top of stack
- Function IsEmpty - returns True if the stack is empty, otherwise False
- Function IsFull - returns True if the stack is full
- Function Peek - returns the value which is currently on the top of stack

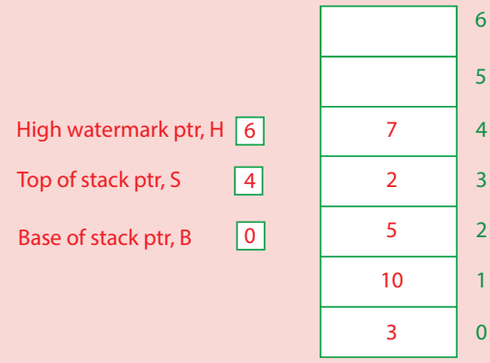


Figure 2.3.1.4 Simple stack with six storage cells

The current state of the stack is shown in *Figure 2.3.1.4*.

(a) (i) What is the value returned when the stack operation Peek is executed on the stack?

(ii) What is the value of S after this operation is executed?

(b) Assuming the current state of the stack is as shown in *Figure 2.3.1.4*, what is the value of S after the following sequence of stack operations are executed

Push(14), Push(27)

(c) After the operations in (b) have been executed, what value is returned when the operation IsFull is now executed on the stack?

(d) Assuming the current state of the stack is as shown in *Figure 2.3.1.4*, what is the value of S after the following sequence of stack operations are executed

Pop, Pop, Pop, Push(8)

(e) Assuming the current state of the stack is as shown in *Figure 2.3.1.4*,

(i) What value is returned when the operation IsEmpty is executed on the stack?

(ii) What value is returned when the operation IsEmpty is executed on the stack after five Pop operations?

In this chapter you have covered:

- Describing and applying the following stack operations:
  - push
  - pop
  - peek or top
  - test for empty stack
  - test for full stack

Single licence - Abingdon School

2 Fundamentals of data structures

2.4 Graphs

Learning objectives:

- *Be aware of a graph as a data structure used to represent more 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.*

Key term

Graph:
A graph is a collection of vertices and a collection of edges connecting these vertices.

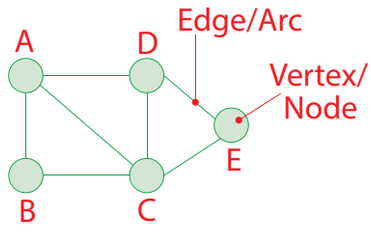


Figure 2.4.1.2 Undirected graph

2.4.1 Graphs

Introduction

Connections between stations on the London Underground network, and connections between friends on Facebook are both examples of complex relationships that may be modelled by a representation and data structure known as a graph.

The objects – tube stations, people– are modelled in a graph by circles called vertices or nodes. The interconnections between pairs of objects are represented by lines, called edges or arcs, joining the circles.
Figure 2.4.1.1 shows a graph of mutual friendships within a Facebook social network. The vertices are labelled with people's names and the edges represent friendship relationships.

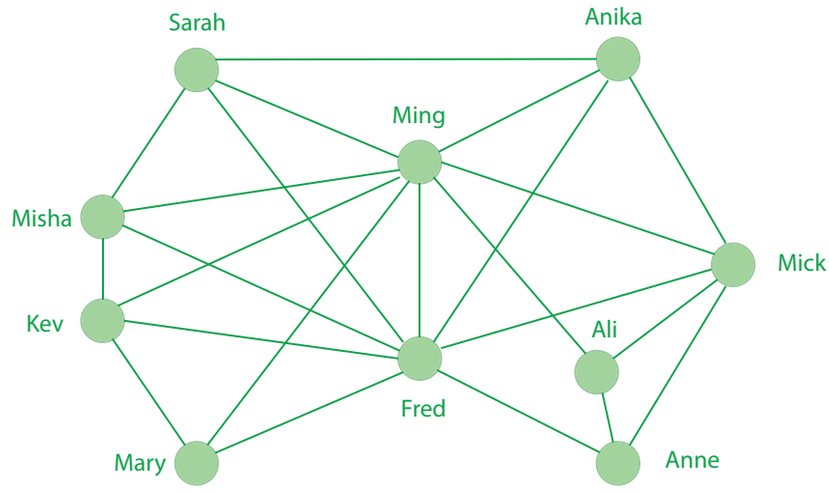


Figure 2.4.1.1 Modelling mutual friendships with a graph

Graph

A graph G = (V, E) consists of two parts

- a nonempty set V of vertices (or nodes), e.g. {A, B, C, D, E} in *Figure 2.4.1.2* (vertices is the plural of vertex)
- a collection E of edges expressed as pairs of vertices, e.g. (A, B), (A, C), (A, D), (B, C), (C, D), (C, E), (D, E) in *Figure 2.4.1.2*.
  (A, B) means there is an edge between the vertices A and B.

Single licence - Abingdon School

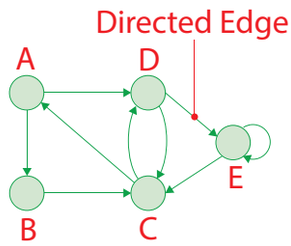


Figure 2.4.1.3 Directed graph

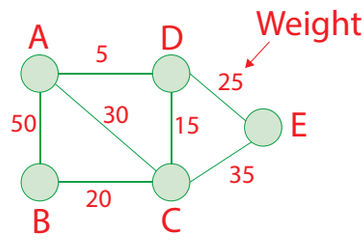


Figure 2.4.1.4 Weighted graph

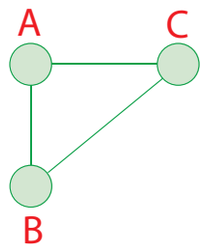


Figure 2.4.1.5 Undirected graph

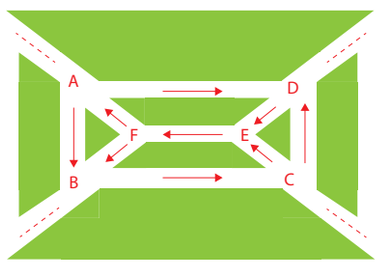


Figure 2.4.1.6 One-way street system

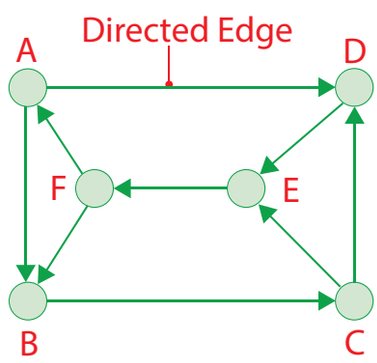


Figure 2.4.1.7 Directed graph modelling one-way street system

Each edge has either one or two vertices associated with it (see *Figure 2.4.1.3* for an edge associated with just one vertex), called its endpoints. An edge is said to connect its endpoints.

In a simple graph each edge connects two different vertices and no two edges connect the same pair of vertices (see *Figure 2.4.1.2*).

Weighted graph

A graph with a weight associated with each edge as shown in *Figure 2.4.1.4* is a weighted graph. A weight is a number associated with an edge representing, say, distance between neighbouring towns/cities.

Undirected graph

Figure 2.4.1.5 is an example of an undirected graph G(V, E), its edges are undirected.

The set of vertices for this graph G is

$$V = \{A, B, C\}$$

The edges of G can be traversed in both directions, e.g. from A to B and from B to A, i.e. (A, B) and (B, A).

The collection of edges of G can be modelled as a set of set of *ordered pairs* as follows

$$E = \{\{(A, B), (B, A)\}, \{(A, C), (C, A)\}, \{(B, C), (C, B)\}\}$$

or as a set of unordered ordered pairs as follows

$$E = \{(A, B), (A, C), (B, C)\}$$

Directed graph

Figure 2.4.1.6 shows a one-way street system. If we model this as a graph with six vertices, A, B, C, D, E, F then we need edges that can model the one-way nature of the roads. For this we use a directed graph as shown in *Figure 2.4.1.7*.

A directed graph G = (V, E) consists of V, a nonempty set of vertices (or nodes), and E, a set of directed edges or arcs.

Each edge is an *ordered pair* of vertices.

The directed edge (u,v) is said to start at u and end at v.

For example, in *Figure 2.4.1.7* the edge connecting vertex A with vertex B is represented by the ordered pair (A, B). A directed graph is also called a digraph.

Single licence - Abingdon School

Questions

1. Draw a graph that represents the following friendships among four people:
   Mary is friends with Amy and Misha, but not Jim;
   Jim is friends with Amy;
   Misha is friends with Amy.

2. A builder of two houses A and B wishes to connect these houses to the three utilities, water (W), gas (G), and electricity (E) by pipework/cable that is buried at the same depth in such a way that the nine connections do not cross each other. Model this problem as a graph and find a solution.

3. The vertices of a graph G belong to the set V = {A, B, C, D, E} and its edges to the set of ordered pairs E = {(A, C), (B, A), (D, A), (E, D), (C, E), (C, B)}. Draw the graph G.

4. Represent the London area one-way street system shown in *Figure 2.4.1.8* by a directed graph.

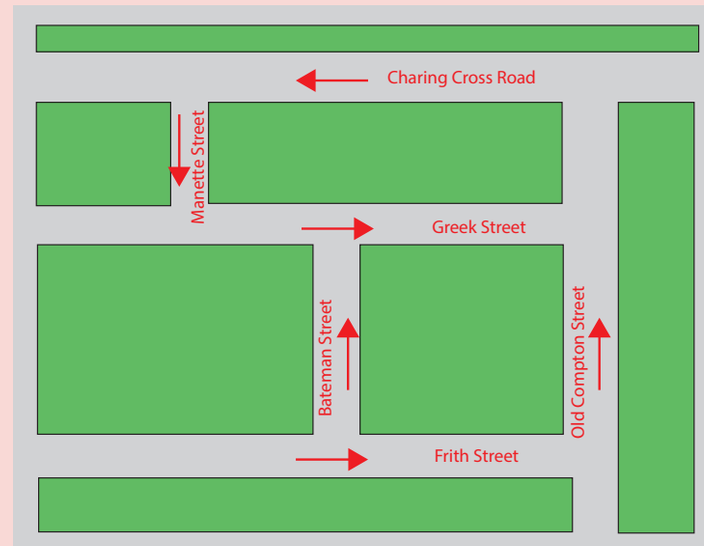


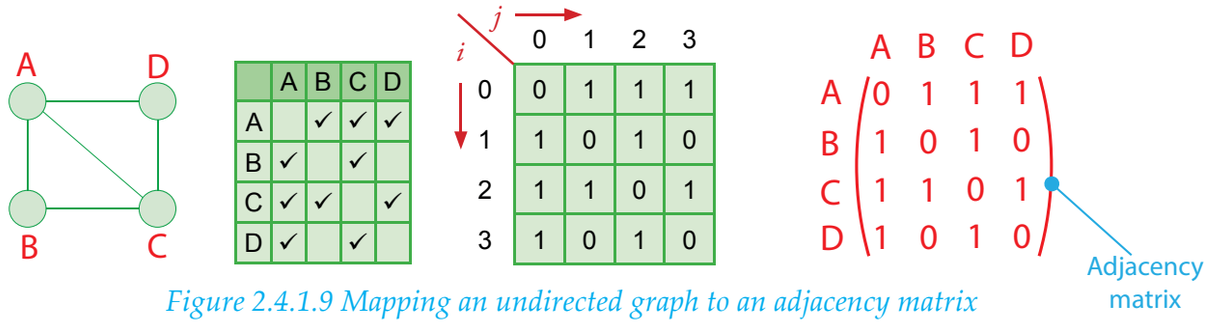
Figure 2.4.1.8 One-way street system in an area of London

5. Different species of animal A, B, C, D, E need to be assigned holding cages so that the following rules are observed:
   A must not be in same cage as B and C.
   B must not be in same cage as D and E.
   C must not be in same cage as A and D.
   D must not be in the same cage as A, B, and C.
   E must not be in the same cage as A, B, and D.
   (a) Model this problem as a graph.
   (b) What is the minimum number of cages that will be required?

Single licence - Abingdon School

Adjacency matrix

A graph with n vertices can be represented as a two-dimensional table called a matrix. We make the table with n rows and n columns and put a tick in the intersection of the i-th row and j-th column if the i-th vertex is connected to the j-th vertex as shown in *Figure 2.4.1.9*. In computer science we use 1s and 0s instead of ticks and no ticks: 1 indicates an edge, and 0, no edge. The table is then called the **adjacency matrix** of the graph.



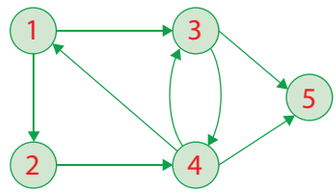
	A	B	C	D
A		✓	✓	✓
B	✓		✓	
C	✓	✓		✓
D	✓		✓	

i \ j	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 1 & 0 & 1 & 0 \\ C & 1 & 1 & 0 & 1 \\ D & 1 & 0 & 1 & 0 \end{array}$$

Figure 2.4.1.9 Mapping an undirected graph to an adjacency matrix

Figure 2.4.1.10 shows the adjacency matrix for a directed graph.



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 & 1 \\ 4 & 1 & 0 & 1 & 0 & 1 \\ 5 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Figure 2.4.1.10 Mapping an undirected graph to an adjacency matrix

If the graph's edges are weighted (*Figure 2.4.1.11*), then the value of each edge weight is substituted for each 1 in the adjacency matrix, as shown in *Figure 2.4.1.12*. Note that in some applications, 0 is a valid weight. In this case it can't be used as a no-edge value. Instead, a value must be chosen that is not a possible weight. To indicate this, the infinity symbol (∞) may be used.

Key term

Adjacency matrix:
A graph with n vertices can be represented as a two-dimensional table called a matrix.
We make the table with n rows and n columns and put a 0 or a 1 in the intersection of the *i-th* row and *j-th* column if the *i-th* vertex is connected to the *j-th* vertex.
1 indicates an edge, and 0, no edge.

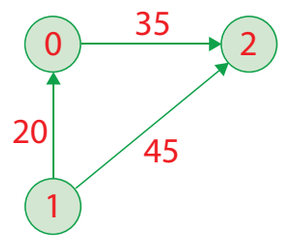


Figure 2.4.1.11 Directed graph with weights

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & \infty & 35 \\ 1 & 20 & 0 & 45 \\ 2 & \infty & \infty & 0 \end{array}$$

Figure 2.4.1.12 Adjacency matrix for directed graph with weights

Single licence - Abingdon School

Questions

6 Write the adjacency matrix for the graph in *Figure 2.4.1.13*.

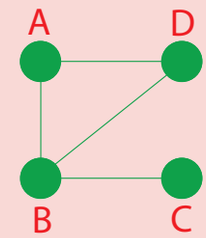


Figure 2.4.1.13

7 A graph can be drawn to represent a maze.

In such a graph, each graph vertex represents one of the following:

- *the entrance to or exit from the maze*
- *a place where more than one path can be taken*
- *a dead end.*

Edges connect the vertices according to the paths in the maze.

Figure 2.4.1.14 shows a maze and *Figure 2.4.1.15* shows one possible representation of this maze.

Position 1 in *Figure 2.4.1.14* corresponds to vertex 1 in *Figure 2.4.1.15* and is the entrance to the maze.

Position 4 in *Figure 2.4.1.14* is the exit to the maze and corresponds to vertex 4.

Complete a copy of *Table 2.4.1.1* to show how the graph in *Figure 2.4.1.14* would be stored using an adjacency matrix.

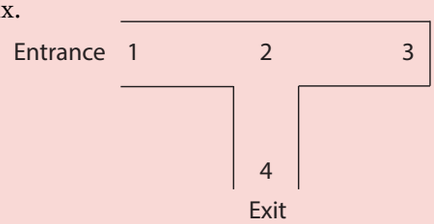


Figure 2.4.1.14

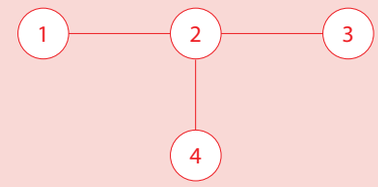


Figure 2.4.1.15

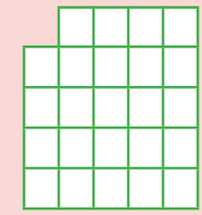


Table 2.4.1.1

8 *Figure 2.4.1.16* shows a maze and *Figure 2.4.1.17* shows one possible representation of this maze.

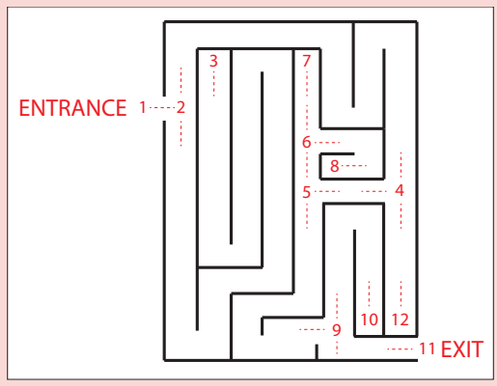


Figure 2.4.1.16

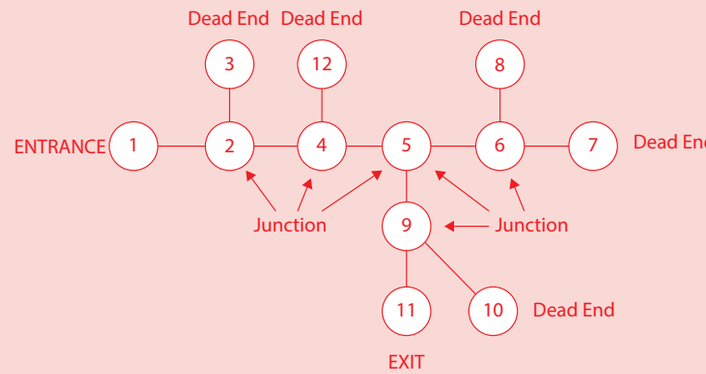


Figure 2.4.1.17

Complete a copy of *Table 2.4.1.2* below to show how the graph in *Figure 2.4.1.17* would be stored using an adjacency matrix.

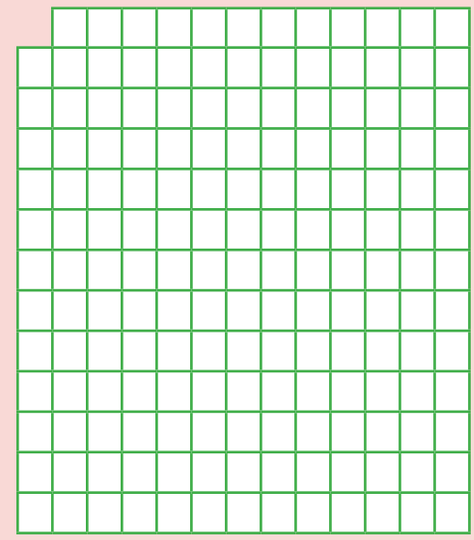


Table 2.4.1.2

Single licence - Abingdon School

Adjacency list

An adjacency list can be used to represent a graph with no multiple edges by specifying the vertices that are adjacent to each vertex of the graph.

Undirected graph

Table 2.4.1.3 is an adjacency list that represents the undirected graph in *Figure 2.4.1.18*. It shows that vertex 1 is connected to vertices 2, 3 and 4; vertex 2 is connected to vertices 1 and 4; vertex 3 is connected to vertices 1, 4 and 5; vertex 4 is connected to vertices 1, 2, 3, and 5; and vertex 5 is connected to vertices 3 and 4.

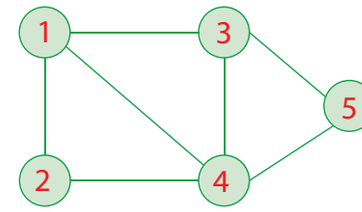


Figure 2.4.1.18 Undirected graph

Vertex	Adjacent vertices
1	2, 3, 4
2	1, 4
3	1, 4, 5
4	1, 2, 3, 5
5	3, 4

Table 2.4.1.3 Adjacency list

Directed graph

With a directed graph, the adjacency list must reflect the direction of edges. *Table 2.4.1.4* shows the adjacency list for the directed graph of *Figure 2.4.1.19*. A pair of adjacent vertices consists of an initial vertex and a terminal vertex. *Table 2.4.1.4* shows that there are directed edges from vertex 1 to vertices 2 and 3, a directed edge from vertex 2 to vertex 4, directed edges from vertex 3 to vertices 4 and 5, directed edges from vertex 4 to vertices 1 and 5, and no directed edges from vertex 5.

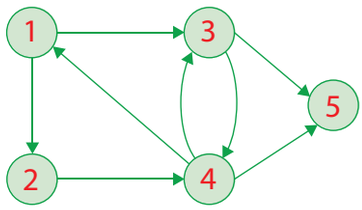


Figure 2.4.1.19 Directed graph

Vertex	Adjacent vertices
1	2, 3
2	4
3	4, 5
4	1, 5
5	

Table 2.4.1.4 Adjacency list

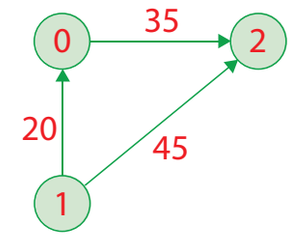


Figure 2.4.1.20 Directed graph with weights

In an adjacency list, we can add a new field to each list element to store the weight of that edge; they are shown in red in *Table 2.4.1.5* for the graph in *Figure 2.4.1.20*.

Vertex	Adjacent vertices
0	2, 35;
1	0, 20; 2, 45
2	

Table 2.4.1.5 Adjacency list for weighted graph

Key term

Adjacency list: Represents a graph by specifying the vertices that are adjacent to each vertex of the graph.

Questions

9 Represent the undirected graph in *Figure 2.4.1.13* as an adjacency list.

10 Represent the directed graph in *Figure 2.4.1.3* an adjacency list.

11 Represent the weighted directed graph in *Figure 2.4.1.4* an adjacency list.

Task

1 Watch the mycodeschool.com video which introduces graphs and considers some applications.
Video URL: https://www.youtube.com/watch?v=gXgEDyodOJU&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=38

Single licence - Abingdon School

Typical uses for graphs

A diverse range of systems may be represented by graphs (*Table 2.4.1.6*). Graph models can be used to solve problems in many different fields.

Used to model	Graph example	Vertices	Edges
Information networks	World Wide Web	Web pages	Hyperlinks
Computer networks	LAN	Computers and switches	Links
	Internet	Routers	Links
Communication networks	Telephone networks	Telephone exchanges	Links
	Packet-switched networks	Packet switches	Links
Social networks	Facebook	People	Friendships
	Twitter	People	Following
	Hollywood graph	Actors	Acted in same film as
Biological networks	Neural network	Neurons	Synapses
Electrical networks	Circuits	Resistors, inductors, capacitors, power supplies, transistors, logic gates	Wires/tracks on printed circuit boards/ interconnections in integrated circuits
Transportation networks	Flight networks	Airports	Air routes
	Road network	Locations, e.g. towns	Roads
Chemical compounds	Molecules/polymers	Molecules	Chemical bonds

Table 2.4.1.6 Some examples in which graphs are used to model the application

The Hollywood graph is an example of a social network system. It represents actors by vertices and connects two vertices when the actors represented by these vertices have worked together on a film. This graph is a simple graph. In January 2006 it contained 20 million edges and 637 099 vertices representing actors who had appeared in 339896 films.

Figure 2.4.1.21 shows a very small part of this graph. Bruce Willis appeared with Sam Moses and Bradley Paterson in the film 16 Blocks (2006) and with Robert Sedgwick and Samuel L. Jackson in the film Die Hard: With a Vengeance (1995).

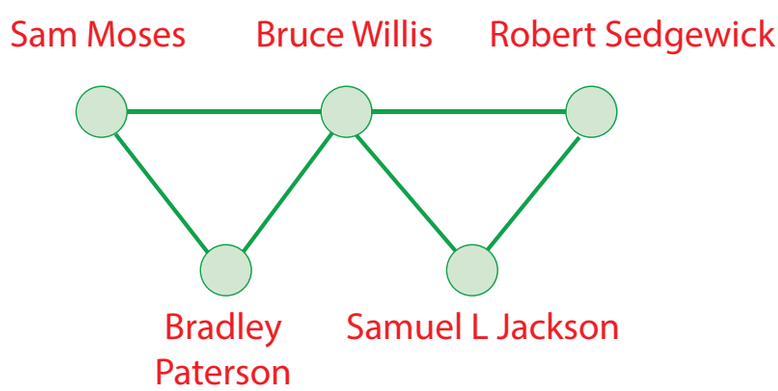


Figure 2.4.1.21 A part of the Hollywood graph

Task

2 Watch the video on cell graphs that reveal a new application of graphs: Image-driven modelling of structure-function relationship in human cells. Video URL: https://www.youtube.com/watch?v=cLGNJuri4pg&index=1&list=PLn0nrSd4xjjbIHhktZoVlZuj2MbrBBC_f

Single licence - Abingdon School

Connectivity

Many applications of graphs involve getting from one vertex to another. For example, it may be required to find the shortest route between one place and another or to find a route out of a maze. Many problems like these can be modelled with paths formed by travelling along the edges of graphs.

A graph is called connected if there is a path between each pair of vertices and is disconnected otherwise.

Path

Informally, a path is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of a graph. In *Figure 2.4.1.21* two vertices, A and B, are linked when there is a chain of actors linking A to B. For example, Sam Moses is linked to Robert Sedgwick by the path Sam Moses–Bruce Willis–Robert Sedgwick. Sam Moses is adjacent to Bruce Willis because they acted together in the film 16 Blocks and Bruce Willis is adjacent to Robert Sedgwick because they acted together in the film Die Hard: With a Vengeance.

In the full Hollywood graph of 637099 vertices, the Bacon number of an actor C is defined to be the length of the shortest path connecting C and the well-known actor Kevin Bacon. An actor who has acted with Kevin Bacon has a Bacon number of 1. Kevin Bacon has a Bacon number of 0.

Which is better, adjacency matrix or adjacency list?

If many vertex pairs are connected by edges, then the adjacency matrix doesn't waste much space and it indicates whether an edge exists with one access (rather than following a list). However, if the graph is sparse, so not many of its vertex pairs have edges between them, then an adjacency list becomes preferable. For example, if the graph has 1000 vertices and only about 2000 edges, its adjacency matrix representation will need 1 million (10^6) entries – 4 million bytes if each entry is a 4-byte word. Representing the same graph with an adjacency list might require, say, 1000 words of memory to identify each node plus 2000 words of memory for the list of neighbours, or 12000 bytes. The difference may make one representation acceptable and the other not acceptable.

Whilst an adjacency list can be more space efficient than a matrix as described above, it may not be as time efficiency with regard to determining if two nodes are connected by an edge or updating edges or obtaining the weight of an edge between two nodes. Access to an element of an array is O(1), i.e. of the order of constant time. An adjacency matrix is a two-dimensional array. However, for the adjacency list as described here, searching the list and modifying the list may be considerably more time consuming than accessing an element of an array.

Task

3 Watch the mycodeschool.com video which compares the use of an adjacency matrix with an adjacency list for a social network application.
Video URL: https://www.youtube.com/watch?v=9C2cpQZVRBA

Questions

12 Explain why an adjacency list is preferred to an adjacency matrix for a social network application.

13 Under what circumstances would the use of an adjacency matrix be considered appropriate for a graph application?

14 Explain why using an adjacency matrix to access edge information can be considered to be more time efficient than using an adjacency list in which edge information is stored in a list.

Single licence - Abingdon School

In this chapter you have covered:

- A graph as a data structure used to represent more complex relationships
- Typical uses for graphs
- The terms:
  - graph
  - weighted graph
  - vertex/node
  - edge/arc
  - undirected graph
  - directed graph
- How an adjacency matrix and an adjacency list may be used to represent a graph
- Compared the use of adjacency matrices and adjacency lists.

Single licence - Abingdon School

2 Fundamentals of data structures

2.5 Trees

Learning objectives:

- *Know that a tree is a connected, undirected graph with no cycles*
- *Know that a rooted tree is*
  - *a tree in which one vertex/node has been designed as the root and*
  - *a parent-child relationship exists between nodes and*
  - *the root is the only node without a parent, all other nodes are descendants of the root.*
- *Know that a binary tree is a rooted tree in which each node has at most two children*
- *Be familiar with typical uses for rooted trees.*

2.5.1 Trees (including binary trees)

Tree

A tree is a connected undirected graph with no cycles.

The graph in *Figure 2.5.1.1* for example, is a tree. Each edge connects two different vertices (no self-loops) and no two edges connect the same pair of vertices (no looping).

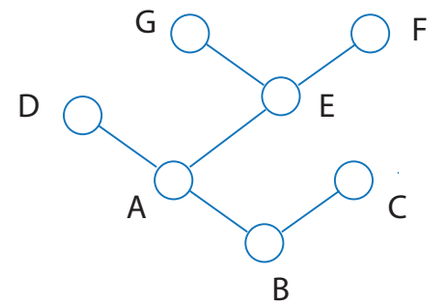


Figure 2.5.1.1 Tree

A path that begins and ends at the same vertex is called a cycle or circuit.

Informally, a path is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of a graph.

Unique path

For a graph to be a tree there must be no cycles or circuits. Therefore, in a tree there is a **unique path** between any two vertices.

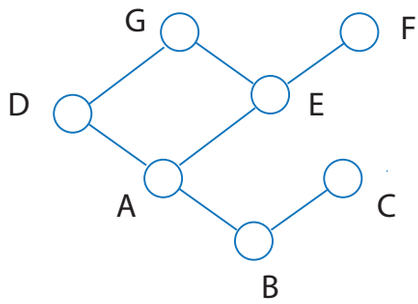


Figure 2.5.1.2 Graph which isn't a tree

The graph in *Figure 2.5.1.2* isn't a tree because more than one path exists between any two given vertices. For example, vertex F can be reached from vertex A along paths A-E-F, A-D-G-E-F, A-D-G-E-A-D-G-E-F, and so on because this graph contains a cycle or circuit (loop) A-E-G-D or A-D-G-E.

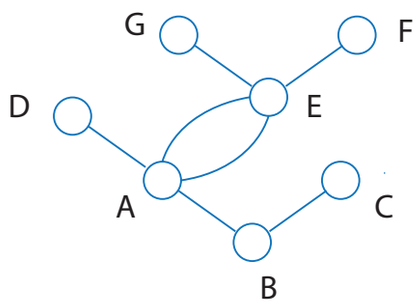


Figure 2.5.1.3 Graph which isn't a tree

The graph in *Figure 2.5.1.3* isn't a tree because more than one path exists between any two given vertices, e.g. vertex B and vertex E have paths B-A-E, B-A-E-A-E, and so on between them. The cause of multiple paths is a loop or cycle within the graph.

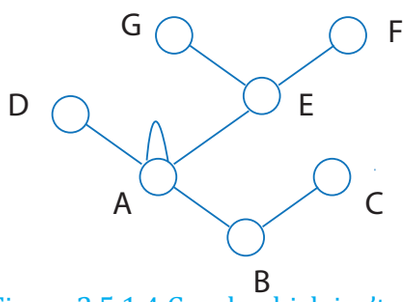


Figure 2.5.1.4 Graph which isn't a tree

The graph in *Figure 2.5.1.4* isn't a tree because more than one path exists between any two given vertices, e.g. B and E has paths B-A-E, B-A-A-E, and so on. The cause of multiple paths is a self-loop or cycle within the graph.

A tree does not contain multiple edges or loops.

Connectedness

An undirected graph is called connected if there is a path between every pair of distinct vertices of the graph.

Single licence - Abingdon School

Key concept

Tree:
A tree is a connected undirected graph with no cycles. Such a tree does not contain multiple edges or loops. There is a unique path between any two vertices. The number of edges is one less than the number of vertices. Other names for a tree are a free tree or an unrooted tree.

Questions

1. Identify **three** different paths (different sequences of edges) between vertex C and vertex F in the graph shown in *Figure 2.5.1.2*.

2. *Figure 2.5.1.5* shows some graphs. Which of these are trees and which are not?

3. Using your answers to question 2, test the theorem that for a connected graph which is a tree the following is true:
   The number of edges is one less than the number of vertices.

4. In the graph model of the chemical butane, C_4H_{10}, each carbon atom (C) is represented by a vertex of degree 4. Degree 4 means that it is connected to 4 edges. Each hydrogen atom (H) is represented by a vertex of degree 1. *Figure 2.5.1.6* shows that the graph for butane is a tree. C_4H_{10} exists in another form called isobutane with a different graph model. Suggest a tree for isobutane which is different from the one for butane. The degree of each carbon vertex is 4 and that of the hydrogen vertex is 1 as for butane.

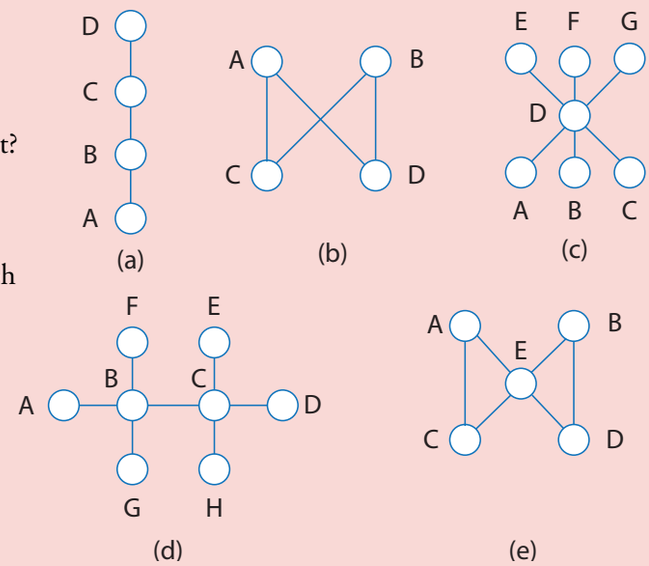


Figure 2.5.1.5 Some graphs which may or may not be trees.

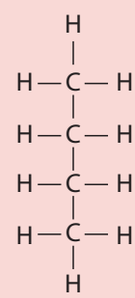


Figure 2.5.1.6 The saturated hydrocarbon butane C_4H_{10}

Rooted tree

A rooted tree is a tree in which one vertex is designated the root and every edge is directed away from the root. *Figure 2.5.1.7* shows the tree from *Figure 2.5.1.1* as a rooted tree with vertex B as the root.

Figure 2.5.1.8 shows the tree from *Figure 2.5.1.1* as a rooted tree with vertex A as the root.

A rooted tree is usually drawn with its root at the top of the graph. The arrows indicating the directions of the edges in a rooted tree may be omitted as shown in *Figure 2.5.1.9*, because the choice of root determines the directions of the edges.

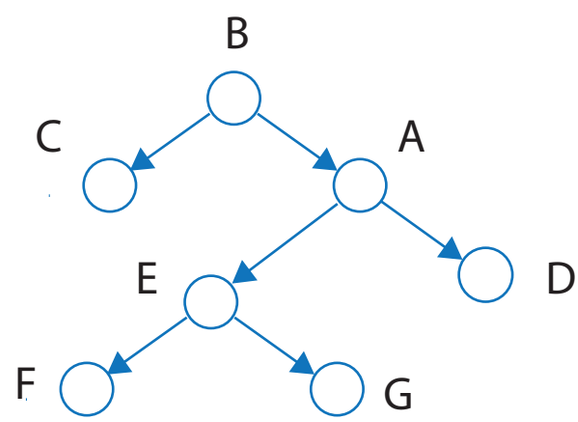


Figure 2.5.1.7 Rooted tree with vertex B as root

Single licence - Abingdon School

Another name for a vertex is a node.

In *Figure 2.5.1.9* the root is labelled the **root node**. Every node except the root node has a **parent**. For convenience, each node has been labelled with a letter of the alphabet. So, for example, the parent of internal node C is node A. Notice that **each node has one and only one parent with the exception of the root node which has no parent**. This allows the path between any two nodes to be unique, e.g. between node A and node H, there is only one path A-B-H.

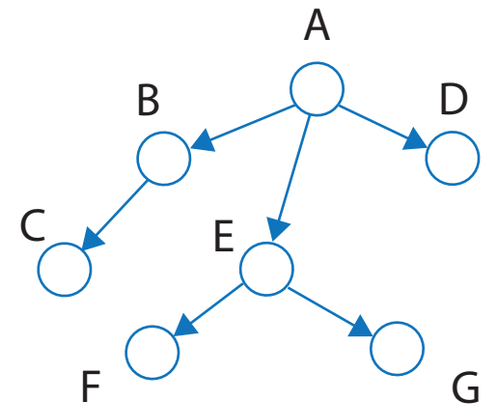


Figure 2.5.1.8 Rooted tree with vertex A as root

If we consider, say node D, its parent is node A and we can also say that node D is a **child** of node A.

Nodes with the same parent are called **siblings**.

The edges are also sometimes called **branches**.

The **ancestors** of a node other than the root are the nodes in the path from the root to this node. For example the ancestors of node L are nodes D and A. The **descendants** of a node, say node A, are those nodes that have node A as an ancestor.

A node of a tree is called a **leaf** if it has no children, e.g. node K in *Figure 2.5.1.9*.

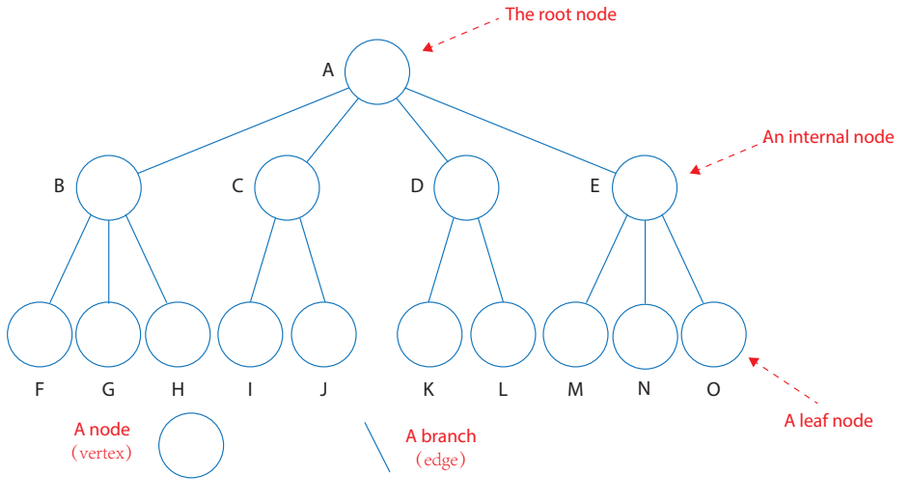


Figure 2.5.1.9 Rooted multi-way tree

Nodes that have children are called **internal nodes** (by this definition, the root node is also an internal node unless it is the only node in the tree). An alternative definition defines an internal node as a node which has children (at least one) and a parent. This definition would exclude the root from being an internal node.

Key concept

Rooted tree:
A rooted tree is a tree in which one vertex is designated the root and every edge is directed away from the root.
A rooted tree has parent-child relationships between nodes.

1. Each node has exactly one parent (node 1-level up) except the root node which has no parent
2. A parent can have several children (nodes 1-level down).

Single licence - Abingdon School

Questions

5 Label a copy of the rooted tree shown in *Figure 2.5.1.10* with the following

- the *root*

an example of a

- *leaf node*
- *internal node*
- *branch*

6 State

(a) the parent of node F

(b) the child of node B

(c) the ancestors of node F

(d) the descendants of node A

(e) the sibling of node B

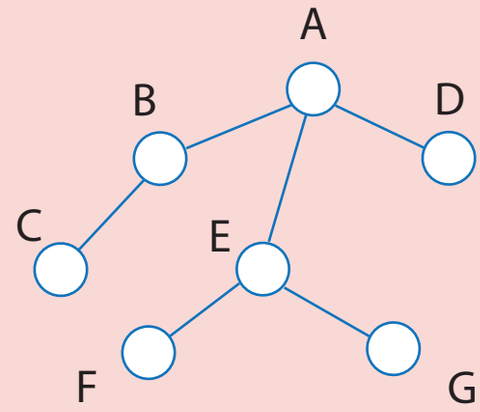


Figure 2.5.1.10 Rooted tree.

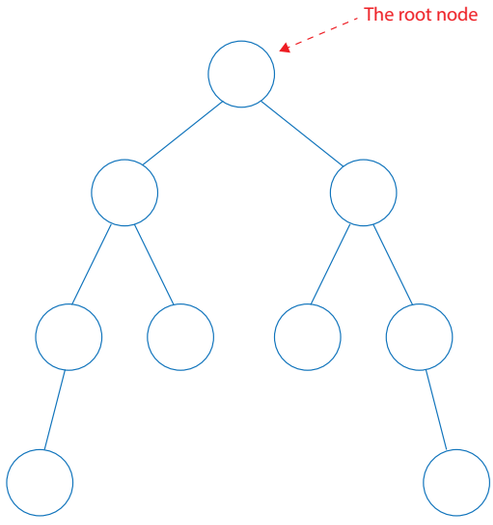


Figure 2.5.1.11 Binary tree

Binary Tree

A **rooted tree in which each node is the parent of at most two other nodes** is known as a **binary tree** as shown in *Figure 2.5.1.11*.

In a binary tree, there are never more than two branches descending from a node.

An alternative definition of a binary tree is as follows:

> **A binary tree is an abstract data type defining a finite set of elements. The set is either empty or is partitioned into three subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees, called the left and right sub-trees of the original tree.**

This alternative view of a tree is shown in *Figure 2.5.1.12*.

Key concept

Binary tree:
A binary tree is a rooted tree in which each node is the parent of at most two other nodes.

Alternative definition:
A binary tree is an abstract data type defining a finite set of elements. The set is either empty or is partitioned into three subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees, called the left and right sub-trees of the original tree.

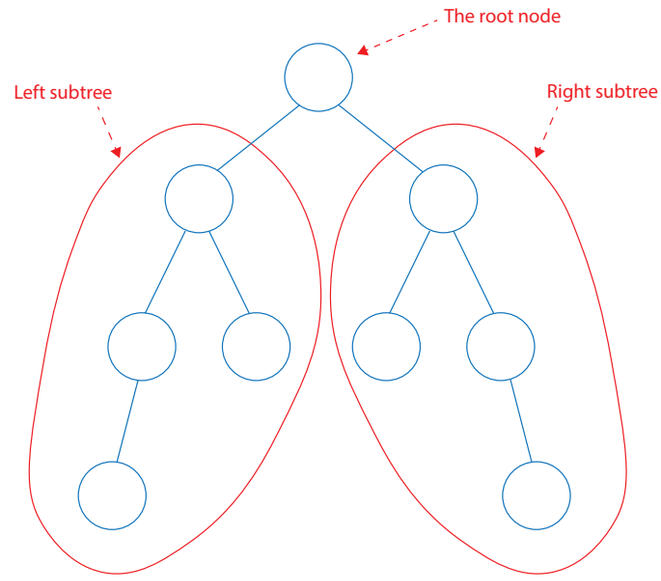


Figure 2.5.1.12 Root, left sub-tree and right sub-tree view of a binary tree

Single licence - Abingdon School

This view of the original binary tree can be applied to each sub-tree. For example the left sub-tree consists of a root and two sub-trees as shown in *Figure 2.5.1.13*.

In turn each of these new sub-trees can be defined in terms of their root, left and right sub-trees, et cetera. Eventually, the sub-trees so defined satisfy the empty case. *Figure 2.5.1.13* illustrates this in a simpler example than for the binary tree in *Figure 2.5.1.12*.

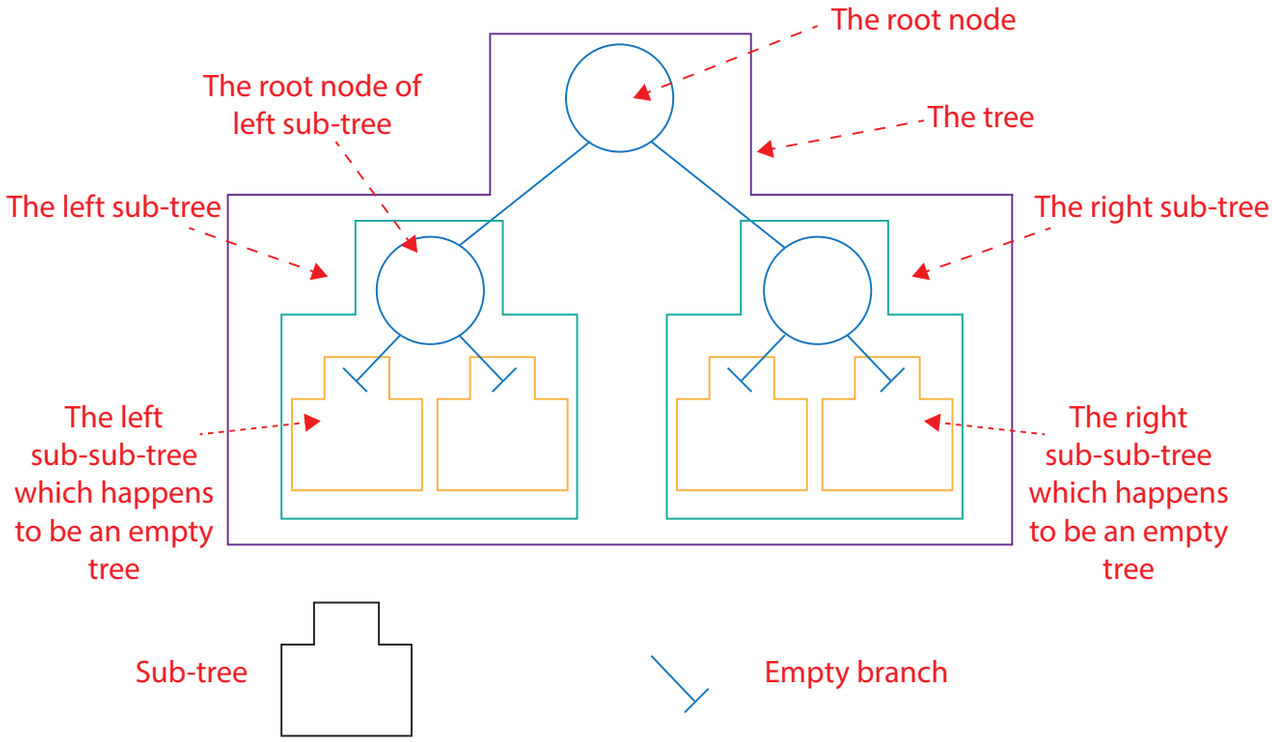


Figure 2.5.1.13 Division of a binary tree into a root, left sub-tree and right sub-tree

What we have done by defining a tree in terms of a root and two sub-trees which, in turn, consist of a root and two sub-sub-trees is define a tree in terms of itself. This is called a recursive definition.

Sometimes it is useful to indicate that a branch is empty, i.e. the node attached to this empty branch is a leaf node. We do this with the symbol $\perp$ placed at an angle as shown in *Figure 2.5.1.13*.

Questions

7 What is a binary tree?

8 Assuming that the top node is designated the root, which figures in *Figure 2.5.1.14* are not binary trees and which are?

9 Identify the left and right sub-trees in the tree shown in *Figure 2.5.1.14(d)*.

10 Identify the left sub-sub-tree of the left sub-tree in the tree shown in *Figure 2.5.1.14(d)*.

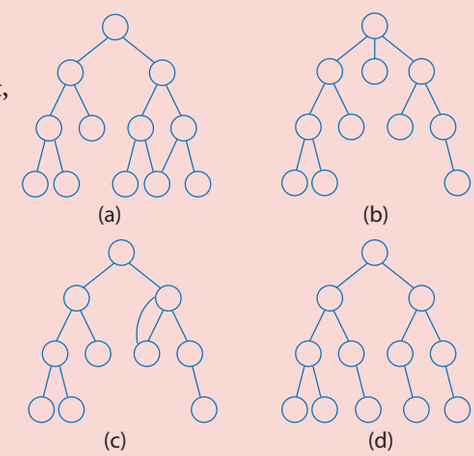


Figure 2.5.1.14 Some graphs only some of which are binary trees

Single licence - Abingdon School

Typical uses for rooted trees

Ordered binary tree

It is possible using a binary tree to sort an unordered list of items into a particular order. For example, consider the following unordered list of integers:

30 20 41 25 37 17 56

An ordered binary tree is created as follows. Using the leftmost integer in the list, 30, the root node is created - *Figure 2.5.1.15 (a)*.

The next integer is taken, 20, and placed in the left sub-tree because it is numerically less than the root node integer 30 - *Figure 2.5.1.15 (b)*.

The next integer in the list, 41, is numerically greater than the root node integer and so goes into the right sub-tree - *Figure 2.5.1.15 (c)*.

The next integer, 25, is numerically less than the root node integer, 30, but greater than the integer, 20, at the root of the left sub-tree so it is placed to the right of node 20- *Figure 2.5.1.15 (d)*.

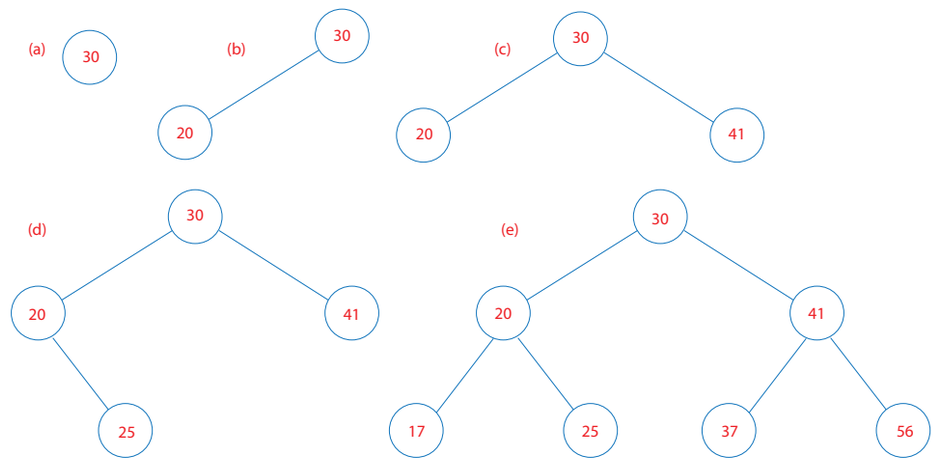


Figure 2.5.1.15 Ordered binary tree

The remainder of the list is processed in a similar way to produce the finished ordered binary tree shown in *Figure 2.5.1.15 (e)*.

By visiting the nodes of this tree in left to right, bottom to top order, an ordered equivalent of the original list is produced. This ordered list places the integers in ascending numerical order as follows:

17 20 25 30 37 41 56

Questions

11 Create an ordered binary tree from the list of integers 67 43 86 97 17 58 81 77 79 12 15 112 93

12 Create an ordered binary tree from the list of strings: *meat cow pig dog cat fish seal terrapin owl*

Binary search tree

The ordered binary tree in *Figure 2.5.1.15* is also known as a **binary search tree** for a reason that will become obvious when set the task of finding if a particular integer is present in the tree. For example, the integer 25 is found in three comparisons (*Figure 2.5.1.16*) by applying the following binary search algorithm whereas searching the original unordered list takes four comparisons:

Compare 25 with the integer at the root,

if it matches then it has been found otherwise,

if 25 less than the root value then binary search the left sub-tree, if greater binary search the right sub-tree

Continue until a match is found or the left or right sub-tree due to be searched is empty.

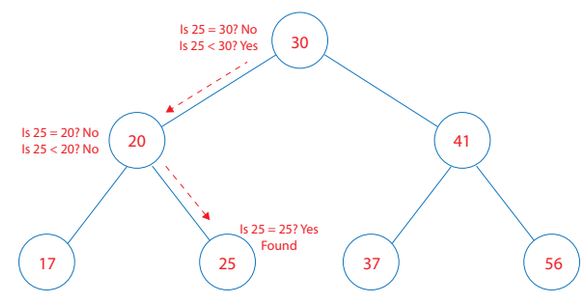


Figure 2.5.1.16 Successful binary search for 25

Single licence - Abingdon School

Figure 2.5.1.17 shows an unsuccessful binary search for the integer 38. The integer 38 is not present in the tree. The searching stops when the empty right sub-tree is encountered.

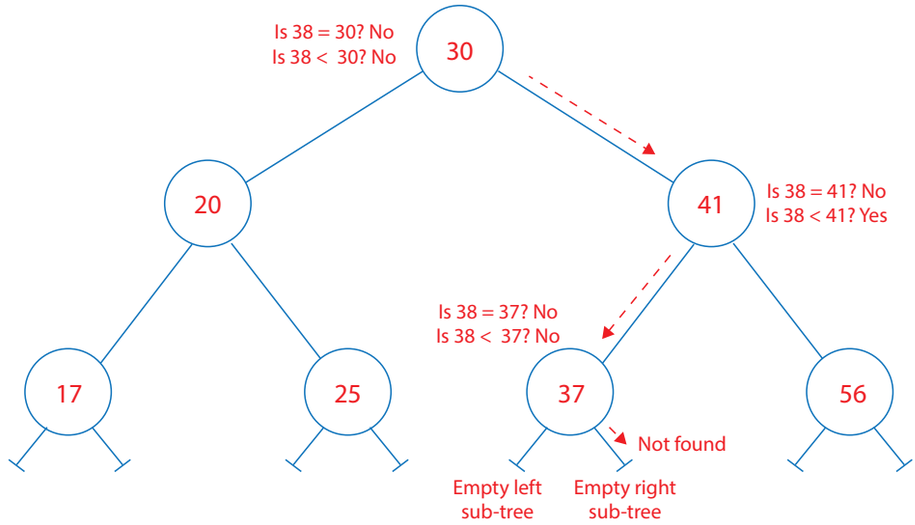


Figure 2.5.1.17 Unsuccessful binary search for 38

Questions

13 (a) How many comparisons are required to locate 25 in the ordered binary tree shown in *Figure 2.5.17*?
(b) List these comparisons.

Syntax trees

Sentences in the English language are constructed according to the rules of English grammar. This grammar specifies valid sentence structures. For example, a valid sentence may consist of a noun phrase followed by a verb phrase. A noun phrase may consist of an adjective followed by a noun. A verb phrase may consist of a verb followed by an adverb.

This valid structure can be expressed in a tree called a **syntax tree**, an example is shown in *Figure 2.5.1.18*.

A given sentence is checked for grammatically correctness by attempting to place its words as leaf nodes as shown in *Figure 2.5.1.18* for the sentence:

Conscientious students study hard

This sentence does fit the syntax tree and so is grammatically valid.

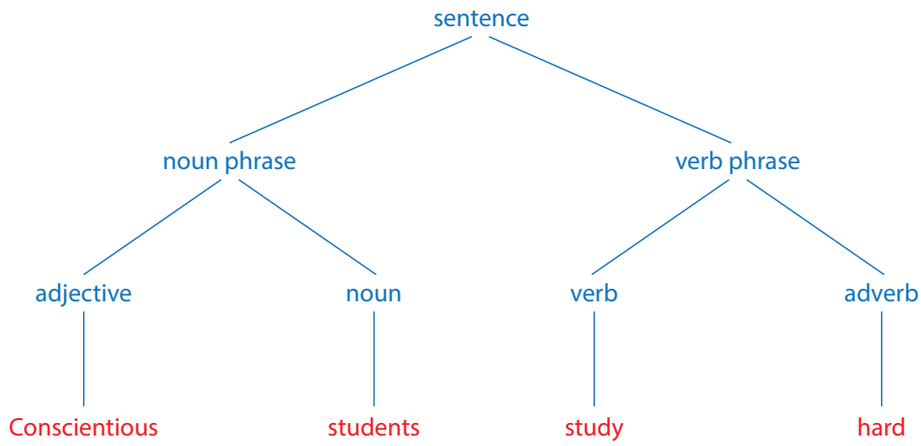


Figure 2.5.1.18 Syntax tree for an English language sentence

Single licence - Abingdon School

Syntax trees are also used by programming language compilers to check that a program statement is syntactically correct. For example, the assignment statement

```
y := 6
```

which assigns 6 to the program variable y has the syntax tree shown in *Figure 2.5.1.19*.

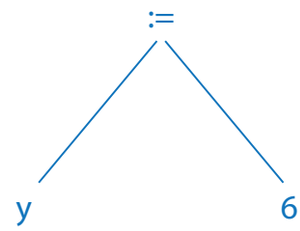


Figure 2.5.1.19 Syntax tree for an assignment statement y := 6

This syntax tree can be used to guide the generation of an equivalent assembly language code[1] form of this assignment statement - *Figure 2.5.1.20*.

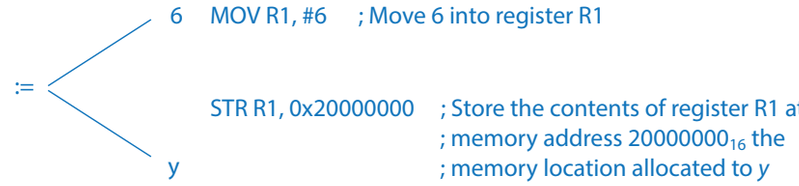


Figure 2.5.1.20 Syntax tree and assembly language equivalent

Figure 2.5.1.21 shows the syntax tree for the program statement

```
y := 3 + y
```

This syntax tree can be used to guide the generation of an equivalent assembly language code form of this assignment statement - *Figure 2.5.1.22*.

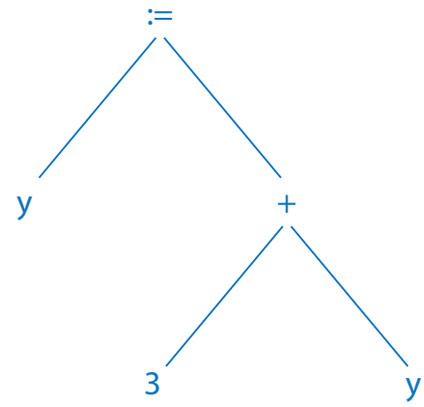


Figure 2.5.1.21 Syntax tree for y := 3 + y

If a programming statement is ill-formed, i.e. invalid syntactically, then the result is in an incomplete syntax tree, e.g.

```
y := 3 +
```

is missing a second operand. The compiler will detect this and report a syntax error.

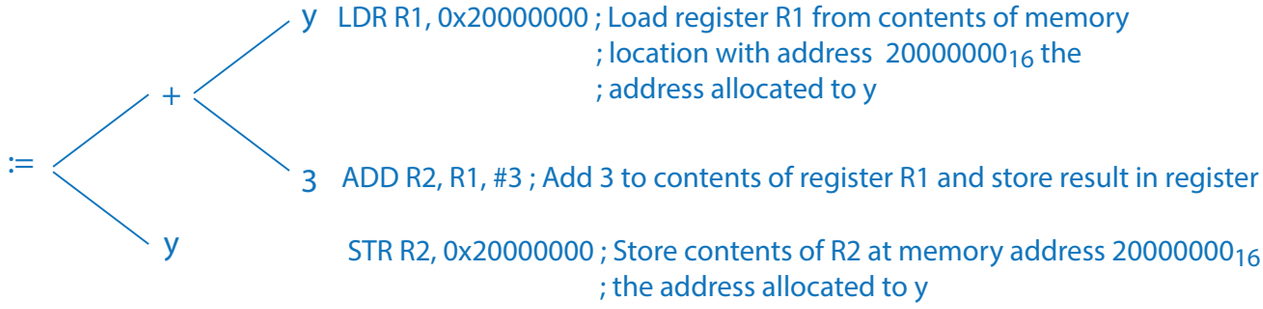


Figure 2.5.1.22 Syntax tree and assembly language equivalent for y := 3 + y

Questions

14 Draw a syntax tree for the sentence Strong oxen pull well.

15 Draw a syntax tree for the program statement `y := 3*y`.

1 See Unit 2 page 288

Single licence - Abingdon School

Decision trees

Rooted trees can be used to model problems in which a series of decisions leads to a solution. These are called decision trees.

We have already encountered one such tree, the binary search tree which may be used to locate an item in a rooted tree based on a series of comparisons.

We may also use a rooted tree to find all possible combinations of heads and tails when a coin is tossed a given number of times.

If H represents a head and T represents a tail then the outcome of tossing a coin twice in succession can be a head followed by a tail which is represented by HT.

Figure 2.5.1.23 shows using a binary tree how all possible outcomes of tossing a coin three times in succession can be generated. Each branch of this tree is labelled H or T representing the two possible outcomes of tossing a coin. If the result of tossing the coin is Head then the H branch is followed and H is appended to the node value.

Each leaf node then shows one possible outcome.

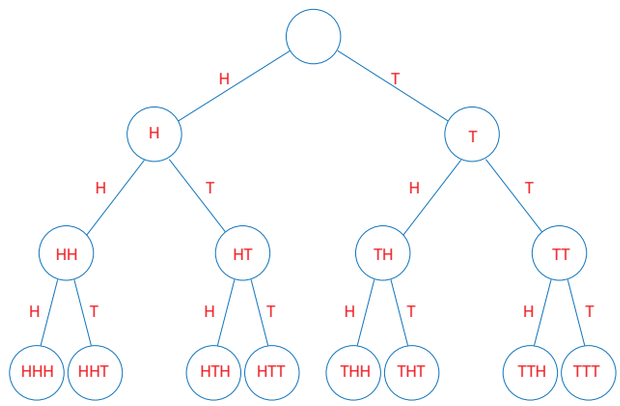


Figure 2.5.1.23 Decision tree which generates all possible outcomes of tossing a coin three times

A decision tree can be used to discover a counterfeit coin in a collection of coins that are all supposed to be of the same weight. Suppose we have two coins, both with the same weight, and a counterfeit coin that weighs less than the others. Given a balance scale as shown in *Figure 2.5.1.24* it is possible to determine the counterfeit coin in just one weighing. We label the coins 1, 2, and 3 respectively.

If coin 1 is placed in the left pan of the scales and coin 3 in the right pan there are three possible outcomes:

1. the pans are balanced because coins 1 and 2 are of equal weight
2. the left side of the balance is lighter than the right
3. the right side is lighter

We can conclude from the weighing evidence that in case 1 when there is balance, coin 2 is the counterfeit coin. In case 2, coin 1 is the counterfeit coin and in case 3, coin 3 is the counterfeit coin.

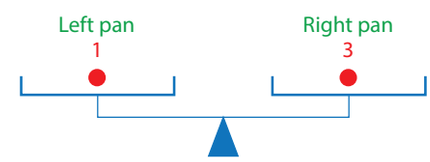


Figure 2.5.1.24 Balance scale

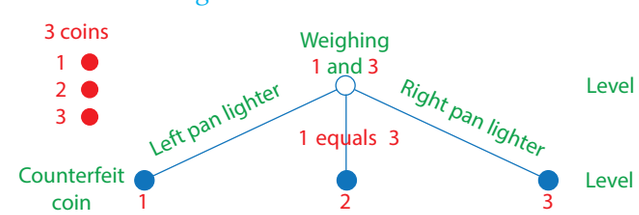


Figure 2.5.1.25 Decision tree which identifies a counterfeit coin amongst three coins

Did you know

In 1699 Isaac Newton became Master of the Royal Mint. One of his many major achievements was to deal successfully with the rampant counterfeiting of the time. The punishment for counterfeiting of coinage was death. Newton was responsible for sending many counterfeiters to the gallows. This was a time when the value of an English coin was meant to match its bullion value - gold coins were really made of gold.

Western Europe was rather behind the times in relying on forms of money whose value was determined by what it was made from. China on the other hand had been using paper for money throughout its empire long before the arrival of Marco Polo in China in the 13th century. Experts put this down to the fact that the Chinese had a system of uniform government across the whole of China whereas Europe was a collection of city states with limited authority beyond the city. The Chinese required everyone to accept the use of paper money. To refuse meant summary execution.

Single licence - Abingdon School

The decision tree drawn in *Figure 2.5.1.25* is a 3-ary tree meaning that each node is the parent of at most three other nodes. This is a consequence of the fact that there are three possible outcomes of a weighing. If there were four then the tree would be a 4-ary tree. This decision tree has 3 leaves because there are three possible outcomes (the counterfeit coin could be any of the three coins).
For convenience, the root of this tree has been labelled level 0, and the level of its immediate descendents, i.e. children, level 1.

Now consider the slightly harder problem of identifying a lighter counterfeit coin amongst 9 coins, eight of which are of the same weight. The trick is to realise that the decision tree which solves this problem is a 3-ary tree (remember in the case of binary search, the search tree was a 2-ary or binary tree because the search branches left or right until a match is found or the search reaches a leaf node).

The 3-ary nature of the tree leads us to divide the set of labelled coins into three subsets as shown in *Figure 2.5.1.26*. After the first weighing, the problem is reduced to finding the counterfeit coin amongst a set of three coins. We need a second weighing to do this, and another level in the tree, level 2, as shown. The figure shows a trace which locates the counterfeit coin 2.

The height, h, of a rooted tree is the maximum of the levels of vertices (nodes). So the height of the rooted tree in *Figure 2.5.1.25* is 1 and the height for the tree in *Figure 2.5.1.26* is 2.

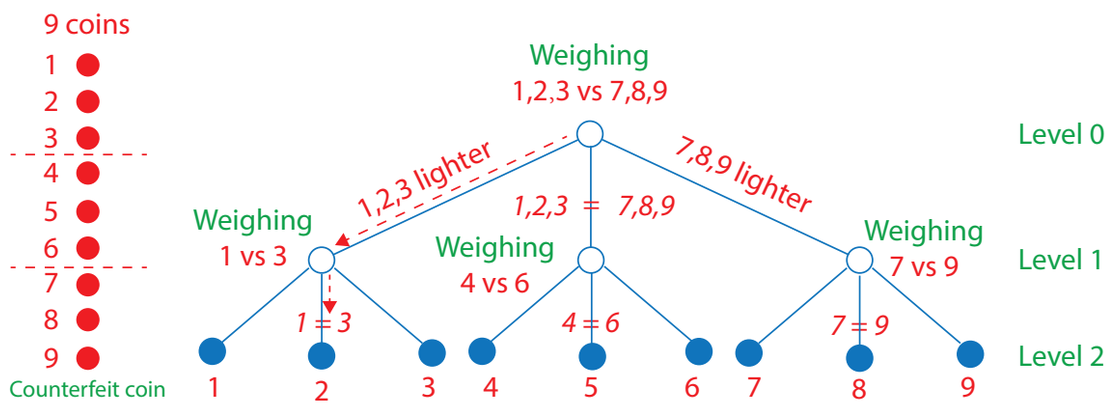


Figure 2.5.1.26 Decision tree which identifies a counterfeit coin amongst nine coins

Generally,

There are at most m^h leaves in an m-ary tree of height h.

Table 2.5.1.1 shows the number of weighings required to find a counterfeit coin amongst a given number of coins of equal weight except for one lighter coin.

Table 2.5.1.1 illustrates something very important about the counterfeit coin tree. Whilst the number of coins (= maximum number of leaf nodes) increases as a power of 3, the height of the tree grows linearly as does the number of weighings. Suppose the number of coins amongst which there is a single counterfeit coin is 14348907 then the tree approach enables this coin to be found after 15 weighings. This number of weighings is the power to which 3 must be raised to equal 14348907, i.e. 3^{15}. If we had to weigh each coin separately we might be very unlucky and only discover the counterfeit coin on the last weighing, i.e. the 14348906th (coins are weighed in pairs).

Figure 2.5.1.27 shows the decision tree when the number of coins is 27. The height of this 3-ary tree is 3 (level of leaf nodes is 3). We divide the set of 27 coins into three equal-sized subsets.
The tree in *Figure 2.5.1.27* is a **full 3-ary tree** because every internal vertex has 3 children. It is also a **complete 3-ary tree** because every leaf is at the same level.

Single licence - Abingdon School

No of coins	No of weighings	Height of tree, h	Maximum no of leaf nodes	Maximum no of leaf nodes expressed as 3^{height}
1	0	0	0	3^0
2 to 3	1	1	3	3^1
4 to 9	2	2	9	3^2
10 to 27	3	3	27	3^3
11 to 81	4	4	81	3^4
82 to 243	5	5	243	3^5

Table 2.5.1.1 No of weighings to find a counterfeit coin included in a given number of coins containing a single counterfeit coin of lighter weight

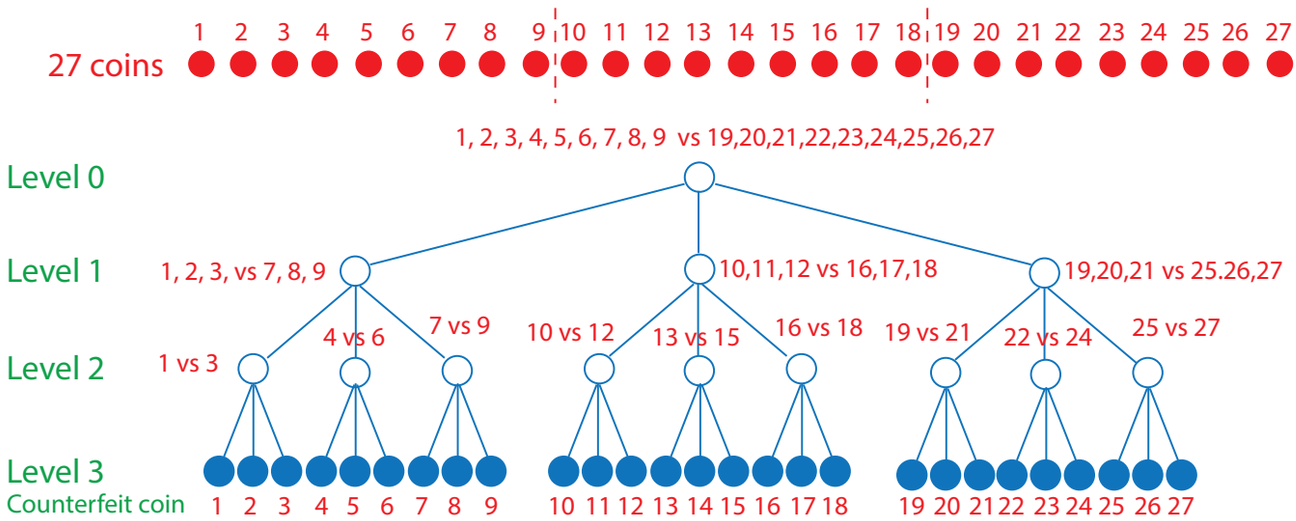


Figure 2.5.1.27 Decision tree which identifies a counterfeit coin amongst twenty seven coins

Figure 2.5.1.28 shows the decision tree when the number of coins is 6. The height of this 3-ary tree is 2 (level of leaf nodes is 2). Two weighings are required to determine the counterfeit coin as illustrated by the trace shown in the figure to counterfeit coin 5. This fits with row 4 of *Table 2.5.1.1* which states that the number of weighings is 2 when the number of coins is between 4 and 9 coins, inclusive.

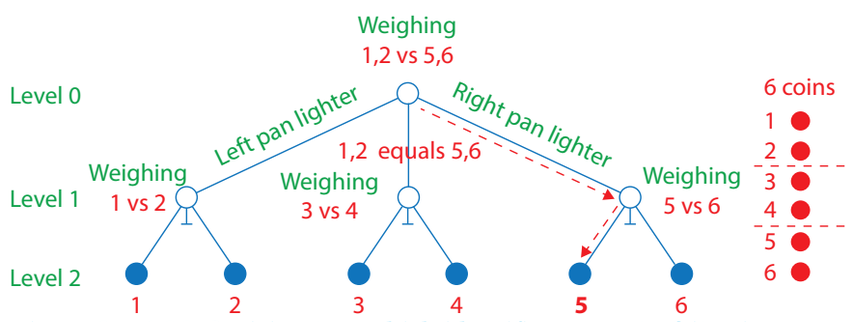


Figure 2.5.1.28 Decision tree which identifies a counterfeit coin amongst 6 coins

Task

1 View the "Santa's Dirty Socks" video at http://csunplugged.org/divideAndConquer/
Can you suggest a quicker way of finding Santa's socks that uses fewer weighings?

Questions

16 Draw a binary tree which shows all possible outcomes of tossing a coin four times in succession. Use H to represent a head and T to represent a tail.

Single licence - Abingdon School

Questions

17 Amongst 19683 pound coins is a single lighter counterfeit pound coin. How many weighings on a balance weighing scale are required to discover the counterfeit pound coin?

18 Amongst 8 pound coins is a single lighter counterfeit pound coin. Draw a 3-ary rooted tree which shows how to weigh these eight pound coins to discover which among them is the counterfeit pound coin. Label the pound coins 1,2,3,4,5,6,7,8, respectively.

19 The height of a full, complete binary tree is 4. How many leaf nodes are there?

Expression trees

Binary trees are useful because they can represent the order in which arithmetic expressions are to be evaluated.

For example, the arithmetic expression

$$(z + 6) * (y - 3)$$

is represented in a binary tree as shown in *Figure 2.5.1.29*.

The order of evaluation is reflected in the tree:

1. Evaluate the left sub-tree: $z + 6 \rightarrow S$
2. Evaluate the right sub-tree: $y - 3 \rightarrow T$
3. Evaluate the tree: $S * T$

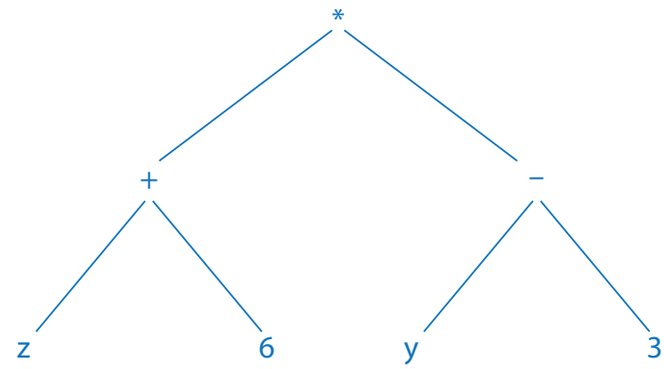


Figure 2.5.1.29 Expression tree reflecting the order of evaluation

The arithmetic expression

$$z + (6 * (y - 3))$$

is represented by the binary tree shown in *Figure 2.5.1.30*.

The order of evaluation is reflected in the tree:

1. Evaluate the right sub-sub-tree: $y - 3 \rightarrow S$
2. Evaluate the right sub-tree: $6 * S \rightarrow T$
3. Evaluate the tree: $z + T$

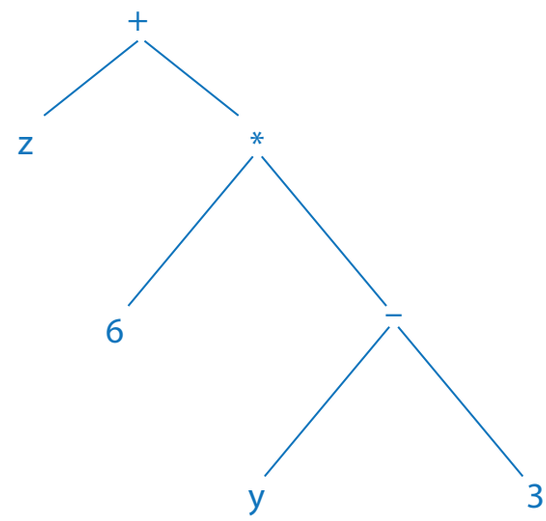


Figure 2.5.1.30 Expression tree reflecting the order of evaluation

Questions

20 Draw an expression tree for the following arithmetic expressions

(a) $(x - 6) * (y + 8)$ (b) $(z + (5/(x + 4))) * (y + 2)$

21 A report is organised into nested sections and the level of each section is indicated by indentation and by the length of the decimal number in the heading as shown in *Figure 2.5.1.31*. Use the information in *Figure 2.5.1.31* to draw a rooted tree to show the organisation of the report.

```
1
  1.1
    1.1.1
      1.1.1.1
      1.1.1.2
    1.1.2
  1.2
  1.3
    1.3.1
    1.3.2
```

Figure 2.5.1.31 Organisation of the sections of a report

Single licence - Abingdon School

Tournament trees

Figure 2.5.1.32 shows a complete rooted tree drawn on its side. It records the result of a knockout tournament between eight players with three rounds played to decide the winner. For example, player 1 plays player 2 in the first round and player 2 wins so goes through to the second round and plays player 3. The eventual winner is player 8.

The height h of this tree is 3. There are $2^4 - 1$ nodes in this tree.

If the number of players in this tournament was halved, the height of the tournament tree would be 2 and the number of nodes $2^3 - 1$. Check this for yourself.

In general, the number of nodes in a knockout tournament tree is

$$2^{h+1} - 1$$

In fact, the number of nodes in a complete binary tree is also

$$2^{h+1} - 1$$

The number of branches is one less than the number of nodes. The number of rounds to be played is the height of the tree.

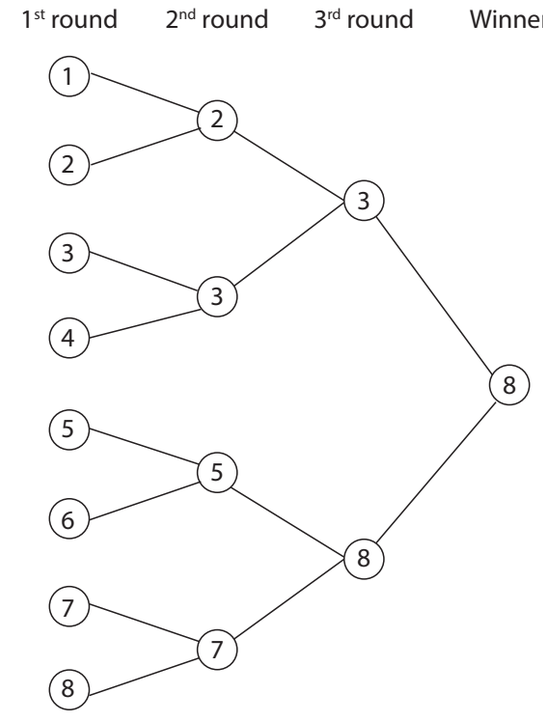


Figure 2.5.1.32 Tournament tree for 8 competitors

Questions

22 How many matches need to be played in a knockout tournament consisting of 64 competitors?

23 A chain letter starts when a person sends a letter to 2 others. Each person who receives the letter sends it to two others. A binary tree is used to record the sending of letters. Each node represents a person in the chain and each branch a letter. Suppose that the chain ends when 65534 people have received a letter and the binary tree is complete.

(a) How many branches does the tree have?

(b) How many nodes does the tree have?

(c) What is the height of the tree?

Game tree

Trees can be used to analyse certain types of games such as noughts and crosses and chess. In each of these games, two players take turns making moves. Each player knows the moves made by the other player and no element of chance enters into the game. Such games are modelled with **game trees**; the nodes of these trees represent the positions that a game can be in as it is played. The edges represent the legal moves between these positions.

Figure 2.5.1.33 shows a part of the game tree for noughts and crosses. The game tree for noughts and crosses is extremely large, too large to draw but a computer can easily build such a tree. Usually some strategy is adopted that tells a player how to select moves to win a game.

The game tree for chess has been estimated to have as many as 10^{100} nodes.

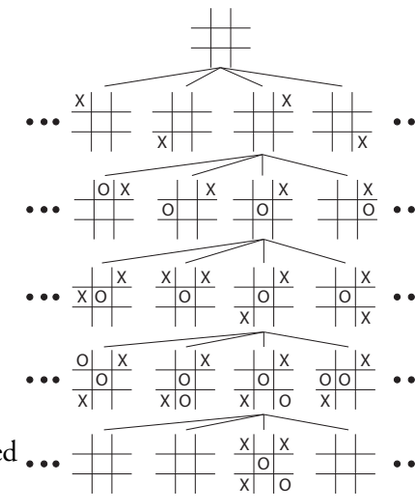


Figure 2.5.1.33 Some of the game tree for noughts and crosses

Single licence - Abingdon School

In order for a computer to play a human at chess the computer must be programmed to use a strategy which reduces the size of the game tree to manageable proportions.

Task

2 Read the "The intelligent piece of paper by Peter McOwan and Paul Curzon of Queen Mary, University of London: http://www.cs4fn.org/teachers/activities/intelligentpaper/intelligentpaper.pdf" which describes a strategy for avoiding losing at noughts and crosses.

In this chapter you have covered:

- That a tree is a connected, undirected graph with no cycles
- That a rooted tree is a tree in which one vertex has been designed as the root and every edge is directed away from the root.
- That a binary tree is a rooted tree in which each node has at most two children
- Typical uses for rooted trees
  - Ordered binary tree
  - Binary search tree
  - Syntax trees
  - Decision trees
  - Expression trees
  - Tournament trees
  - Game trees

Single licence - Abingdon School

2 Fundamentals of data structures

2.6 Hash tables

Learning objectives:

- *Be familiar with the concept of a hash table and its uses.*
- *Be able to apply simple hashing algorithms.*
- *Know what is meant by a collision and how collisions are handled using rehashing.*

2.6.1 Hash tables

Tables

Using a table to store records

A table in computer science is a data structure of rows and columns, an example is shown in *Table 2.6.1.1*. This table consists of 4 rows of data in three columns, labelled `ULN, Forename, Surname`. Each row stores a single record of three fields containing data for an individual student as follows:

- student's unique learner number `(ULN)` consisting of eight digits, e.g. 34567890
- Forename
- Surname

An individual record is uniquely identified by its key field, `ULN`.

The rows of this table are indexed with the first row that contains a student record being labelled with index 0, the second with index 1, and so on.

	ULN	Forename	Surname
0	34567890	Fred	Bloggs
1	90002789	Mary	Smith
2	74432167	Ahmed	Khan
3	24567813	Sarah	White

Table 2.6.1.1 Student records stored in a table

This table will occupy a part of the computer's RAM (main memory). It can also be stored permanently in backing store or secondary storage, e.g. magnetic disk. However, to be searched or manipulated, it must first be copied from secondary store to RAM.

Searching the table for a record

The table could be searched for a particular record by starting at the row labelled with index 0 and scanning the entries in turn until the record is found if it is present, or the end of the table is reached. This is known as linear search which is one of several ways that an existing record can be 'looked up'.

Inserting a new record into the table

Table 2.6.1.2 shows a table similar to *Table 2.6.1.1* but this table has three empty rows following the four rows of data. A new record could be inserted in the first empty row, a second new record in the next row and so on until the table is full.

Single licence - Abingdon School

	ULN	Forename	Surname
0	34567890	Fred	Bloggs
1	90002789	Mary	Smith
2	74432167	Ahmed	Khan
3	24567813	Sarah	White
4			
5			
6			

Table 2.6.1.2 Student records stored in a table with room for new records

Deleting a record in the table

The row containing the student record to be deleted is located by searching from row 0. Once found, the data in the row is deleted. To avoid gaps appearing in the table, the occupied rows following this row are moved up to remove the gap.

Limitations of this type of table and table access

A problem surfaces with the operations of searching, inserting and deleting described above when the table contains a large number of records, e.g. 10,000. It just takes too much of the computer's time to perform these operations. One solution is to use a hash table based on a well-chosen hash function.

Hash table

A hash table resembles an ordinary table as described above but differs in the method used to access the rows of the table.

A row of a hash table is accessed directly when looking up, inserting and deleting a record, i.e. it does not start from row 0 every time but instead goes directly to the required row. Movement of records when deleting a record is also eliminated.

Table 2.6.1.3 shows a hash table that has gone from being empty to containing 3 records located in three different rows with indices, 2, 5, and 6, respectively.

as input data.

24567813 Sarah White

74432167 Ahmed Khan

90002789 Mary Smith

	ULN	Forename	Surname
0			
1			
2	90002789	Mary	Smith
3			
4			
5	74432167	Ahmed	Khan
6	24567815	Sarah	White

Table 2.6.1.3 Hash table storing three student records

Key concept

Hash table:
The table gets its name from the method used to determine the row to use.
The hash value generated by applying a hash function to a key is the table index where the record with this key should be stored if the row is free.

Single licence - Abingdon School

The table gets the name **hash** because of the method used to generate the address or row number. A randomising function called a **hash function** is applied to the record's key, in this case the 8-digit unique learner number or ULN, to map the possible 8-digit ULN values into a much smaller range of values, the possible row numbers. This process is known as hashing.

If the ULN values were used directly as specifiers of row addresses we would have to accommodate addresses covering all possible values of an 8-digit number, 10^8 addresses in total, even though only a small subset of ULNs might be required, e.g. those used in a particular school.

For ease of understanding, the number of rows for the table in *Table 2.6.1.3* has been made small intentionally at seven, and labelled 0, 1, 2, ..., 5, 6.

Hash function

The hash function takes as input the record's key (hash key) and outputs the row address of the row for this record. The output is called the hash value or hash.

In our example, the hash value ranges from 0 to 6 for the seven rows of the given table. A hash function, H, that will map 8-digit ULNs to the set {0, 1, 2, ..., 5, 6} is shown below

```
H(ULN) = ULN Mod 7
```

Mod is the modulo arithmetic operator which calculates the remainder after integer division (see *Chapter 1.1.3*).

Table 2.6.1.4 shows three possible values of ULN being mapped to 2, 5 and 6 respectively e.g. 90002789 when divided by 7 gives 12857541 with a remainder of 2.

ULN	H(ULN)
90002789	2
74432167	5
24567815	6

Table 2.6.1.4 Some hash values produced by hash function H applied to ULN keys.

Key concept

Hash function:
Is a function H, applied to a key k, which generates a hash value H(k) of range smaller than the domain of values of k,
e.g.
H : {00000000..99999999}
→ {0..6}

Key concept

Hash key:
Is the key that the hash function is applied to.

Key concept

Hashing:
The process of applying a hash function to a key to generate a hash value.

Questions

1 Calculate H(ULN) for the following ULNs
(a) 31258745 (b) 62517493 (c) 49981627
Hint: The scientific mode of Microsoft Windows calculator has a Mod operator.

Single licence - Abingdon School

Information

The term "hash" originates by analogy with its non-technical meaning, to "chop and mix". Hash functions often "chop" the input domain into many sub-domains that get "mixed" e.g. add the first three digits of the key, add the last three digits, concatenate the two resulting digit strings then map into the output range by applying modulo N.

Simple hashing functions

Hashing and hash tables are a way that memory locations for records can be assigned so that records can be retrieved quickly.

A hashing function must be relatively quick to compute whilst at the same time generating an even spread of values for the given inputs, the keys.

Another way that the latter can be expressed is that each hash value generated by the hashing function should be equally probable.

Achieving this depends on both the particular key values being hashed, and the particular hash function employed.

The value of N in modulo N is chosen to be prime because this can contribute to producing an even spread of hash values.

One simple hash function that attempts to achieve these objectives, sums the squares of the ASCII codes of each character of `Key`, as shown in *Figure 2.6.1.1* in pseudo-code.

The `Ord` function returns the ASCII code of a given character,

e.g. `Ord('A') = 65`.

The individual characters of `Key` are accessed using array indexing starting at 0, e.g. `Key[0]` accesses the first character in the string.

The algorithm generates hash values in the range 0 ... 522 because `Sum` is Modded with 523, a prime number.

Suppose that `Key` stores a string, then the steps to convert `Key` into a storage-address returned in Hash is as follows:

```
Sum ← 0
For i ← 1 To Length(Key)
  Sum ← Sum + Ord(Key[i]) * Ord(Key[i])
Endfor
Hash ← Sum Mod 523
```

Figure 2.6.1.1 Hashing algorithm that calculates a storage address in range 0 to 522

Looking up a record in a hash table

A record with a given key can be looked up by just calculating the hash of its key and checking the associated storage location.

English-French dictionary example

In this example, English words and their French equivalents are stored in records in a hash table, `HashTable`, using a hashing function, `H`, based on the

Single licence - Abingdon School

hashing algorithm shown in *Figure 2.6.1.1*. Each record must have a key field which uniquely identifies the record. In this case, the key is the English word.

The hashing function, `H`, assigns hash table memory location `H(k)` to the record with key, `k`.

In our English-French dictionary example, `H(k)` could be `H('BEACH')` where `k = 'BEACH'` for the record containing the English word `'BEACH'` and the equivalent French word `'PLAGE'`.

The storage structure, `HashTable`, that will be used with this address has the following data structure:

```
THashTableArray = Array[0..522] Of TRecord
```

Where the data structure `TRecord` is defined as follows

```
TRecord = Record
            EnglishWord : String
            FrenchWord : String
          End
```

Questions

2 Calculate `H(k)` for the following keys, `k`

(a) PEN (b) CAT (c) NOW (d) WON

(ASCII codes for the characters `'A'...'Z'` map to the range 65 ... 90 - see Unit 2 Chapter 5.5)

Programming tasks

1 Write a program to store English words and their French equivalents in a hash table which is an array or its equivalent with addresses in range 0 to 522. The English word and its French equivalent should be stored together in a record or equivalent data structure at an address which is calculated by the hashing function, `H`, described above. The table should be initialised so that every key field stores the string `'-1'` to indicate that this field's record has yet to be used to store an English-French word pair. Use your program to temporarily store the English words, PEN, CAT, NOW and their French equivalents.
(English word with its French equivalent:
PEN – PLUME, CAT – CHAT, NOW – MAINTENANT)

2 Extend your program so that after storing the English-French word pairs for PEN, CAT and NOW, the program uses the hashing function, `H`, to retrieve the French equivalent when the user enters PEN, CAT or NOW. Use a loop to enable the user to continue to look up the French equivalent until the user decides otherwise.

Single licence - Abingdon School

Key concept

Collision:
A collision occurs when two or more different keys hash to the same hash value. For the hash table this means a hash value of a location in the hash table which is already occupied.

Collisions

The hash values calculated in Questions 2(c) and 2(d) are identical because the English words contain the same letters, but arranged in a different order (NOW and WON). So both words hash to the same address. This situation is known as a **collision**. Clearly, there is only space at this address for one English-French word pair.

Collisions can be resolved in two ways:

1. Store the record in the "next available" empty location in the table, or
2. Store a pointer in each table location that points to a list of records that have all collided at this table location, otherwise set the pointer value to null.

Key concept

Closed hashing or open addressing:
Method in which a collision is resolved by storing the record in the "next available" location.

Method 1 – closed hashing or open addressing

The first way of resolving a collision is to *rehash* which means to generate a new table row address at which to store the English-French word pair.
One rehash method, called **linear rehash**, calculates a new address by adding one to the original address before testing that the location with this new address is empty, e.g. indicated by `'-1'` in the `EnglishWord` field.

The rehash step may have to be repeated until an empty slot is found.

To avoid going off the end of the table, the new address is made to wrap around to the beginning of the hash table, if necessary and assuming there is an empty slot, by using modular arithmetic as follows:

```
Repeat
  Address ← (Address + 1) Mod 523
Until HashTable[Address].EnglishWord = '-1'
```

This method is an example of **closed hashing** or **open addressing** because other row addresses of the hash table are open to being used but access to addresses outside the hash table are closed off.

The table, `HashTable`, is an array whose addresses run from 0 to 522.

The table is initialised with 523 empty English-French word pair records in which every EnglishWord field has the string `'–1'` stored in it to indicate that this field is unoccupied and the whole record is empty.

Figure 2.6.1.2 shows an algorithm expressed in pseudo-code for inserting an English-French word pair into an initialised `HashTable`. The English word to insert is supplied in `WordInE` and its French equivalent in `WordInF`. Each row of the hash table has space for a record with two fields, `EnglishWord` and `FrenchWord`.

Single licence - Abingdon School

```
Address ← Hash(WordInE)
If HashTable[Address].Key = '-1'
                 {-1 indicates field is empty}
  Then
    Begin
      HashTable[Address].EnglishWord ← WordInE
      HashTable[Address].FrenchWord ← WordInF
    End
  Else
    If Not(HashTable[Address].EnglishWord = WordInE)
                              {not already stored}
      Then
        Begin
          {find empty slot)
          Repeat
            Address ← (Address + 1) Mod 523
          Until (HashTable[Address].EnglishWord = '-1')
            Or (HashTable[Address].EnglishWord = WordInE)
                                  {already stored}
          If (HashTable[Address].EnglishWord = '-1')
              Then
                Begin
                  HashTable[Address].EnglishWord ← WordInE
                  HashTable[Address].FrenchWord ← WordInF
                End
        End
```

Figure 2.6.1.2 Hashing algorithm incorporating a linear rehash that inserts an English-French word pair into a hash table

Clearly for this algorithm to work the hash table must have at least one empty row.

Single licence - Abingdon School

Searching for a specific record in a hash table accommodating collisions

Figure 2.6.1.3 shows an algorithm expressed in pseudo-code that can be used to search for an English-French word pair in a hash table, `HashTable`, given an English word stored in the variable `WordInE`.

The English word may or may not be present in the hash table.

If it is, then its French equivalent is returned in variable, `Retrieve` otherwise message `'Not found'` is returned in `Retrieve`.

Clearly for this algorithm to work the hash table must have at least one empty row.

```
Address ←  Hash(WordInE)
Found ←  False
Repeat
  If HashTable[Address].EnglishWord = WordInE
    Then Found ←  True
    Else Address ←  (Address + 1) Mod 523
Until Found Or (HashTable[Address].EnglishWord = '-1')
If Found
  Then Retrieve ←  HashTable[Address].FrenchWord
  Else Retrieve ←  'Not found'
```

Figure 2.6.1.3 Hashing algorithm incorporating a linear rehash method that is used to search a hash table for the French equivalent of a given English word

Setting up a hash table that uses closed hashing

Method 1 (closed hashing or open addressing) requires that the number of rows in the table exceeds by about a third the maximum number of records that will ever be stored in the table. When every record has been stored in the table the table should still contain empty rows (i.e. table should never be more than roughly two thirds full). If this isn't the case then search times will be extended as will the time to insert new records.

Although this might seem a waste of storage space, there is a very good reason for working in this way. Studies have shown that the number of collisions depends on

- the hash keys
- the hash function
- the ratio of total number of records to total number of possible locations available to these records in the hash table.

A perfect hash function hashes all the hash keys to hash values without the occurrence of a single collision. That is why it is called perfect.

However, finding a perfect hash function is extremely difficult.

Single licence - Abingdon School

The effectiveness of a hash function is very sensitive to the hash key values. These are not always fully known in advance.

Using a ratio of roughly two thirds for total number of records to total number of hash table locations seems to minimise collisions for hash functions that are close to perfect. The hash table shown in *Table 2.6.1.5* has six student records and seven rows. One improvement could be to change the number of rows to 9 or even better, 11, a prime number. **Using a prime number for modulo arithmetic helps to minimise collisions**.

However, the hash function could be further improved as well as it is far from being perfect.

The aim is to make each hash value generated by the hash function equally likely when the function is applied to any of the possible hash keys, i.e. no one particular hash value should be more favoured than any other.

	ULN	Forename	Surname
0	34567876	Fred	Bloggs
1			
2	90002789	Mary	Smith
3	64156906	Alex	Black
4	24567805	Visha	Baal
5	74432167	Ben	Brown
6	90002985	Shena	Patel

Table 2.6.1.5 Hash table with not enough rows to minimise collisions

Questions

3 Copy and complete *Table 2.6.1.6*.

ULN	ULN Mod 7	ULN Mod 11
24567805		
34567876		
64156906		
74432167		
90002789		
90002985		

Table 2.6.1.6

Single licence - Abingdon School

Questions

4 Insert the ULNs from *Table 2.6.1.6* into a copy of the hash table shown in *Table 2.6.1.7* using the hashing function,

```
H(ULN) = ULN Mod 7
```

The student `Forename` and `Surname` fields do not need to be completed.

You should deal with any collision by performing a linear rehash until an empty slot is found.

	ULN	Forename	Surname
0			
1			
2			
3			
4			
5			
6			

Table 2.6.1.7 Hash table

5 Insert the ULNs from *Table 2.6.1.6* into a copy of the hash table shown in *Table 2.6.1.8* using the hashing function,

```
H(ULN) = ULN Mod 11
```

The student `Forename` and `Surname` fields may be omitted for convenience.

	ULN	Forename	Surname
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

Table 2.6.1.8 Hash table

Single licence - Abingdon School

Questions

6 Explain how the hash table in *Table 2.6.1.8* when populated with student records would be used to look up the forename and surname of student with ULN = 24567805.

7 Explain how the hash table in *Table 2.6.1.7* when populated with student records would be used to look up the forename and surname of student with ULN = 24567805.

8 Why is it necessary to store the key field in a hash table even when an application using this hash table must already know the value of the key field?

Investigation

1 Devise an experiment to investigate collisions on a hash table that is to store 6000 student records. Use a random number generator to generate unique student ID numbers. Try different ratios of total number of records to total number of table rows in the hash table.

Investigation

2 The hash function H that we have used so far is far from perfect for many data sets that we wish to store in a hash table. Investigate other hashing functions.

Method 2 - open hashing or closed addressing

In this alternative method of dealing with collisions, the hash table is extended to include a pointer field. The pointer field for each row is initialised to the null pointer value when the table is set up (¬).

When a collision occurs the colliding record is linked to the corresponding table row by changing the pointer field of this row to point to the colliding record as shown in *Figure 2.6.1.4*.

Key concept

Open hashing or closed addressing:
In a collision, the other rows of the hash table are closed to the colliding record which must, instead, be attached to the addressed table row in a chain or linked list of other colliding records. The table row uses a pointer field to point to the linked list.

Single licence - Abingdon School

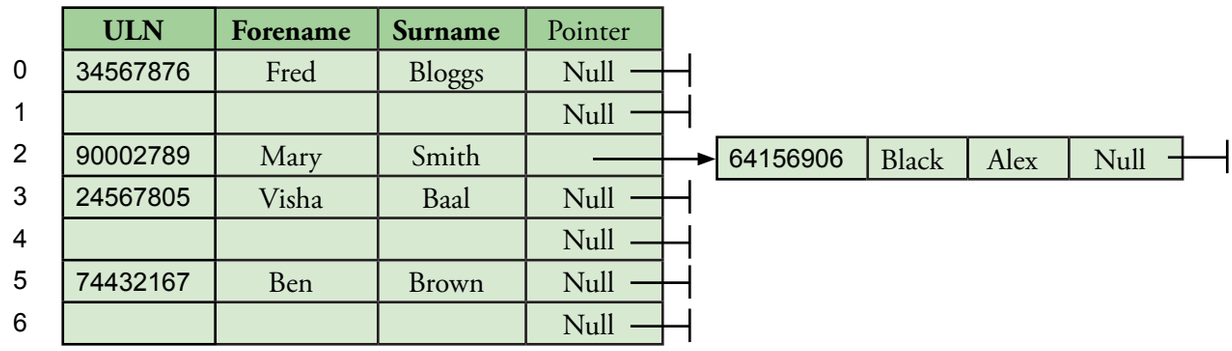


Figure 2.6.1.4 Hash table that uses open hashing

Information

The definitions assigned to the terms closed hashing and open hashing have been interchanged over the years so care needs to be exercised when interpreting them. The key is to focus on concept/method not name and to make sure that you understand the former.

Another record colliding with row 2 will be linked or chained to the record of `'Mary Smith'` by changing the pointer field of Mary's record to point to this record and so on, thus forming a chain of linked records or linked list. Method 2 is called **open hashing** or **closed addressing** because locations outside the table are open for use by the hashing algorithm, i.e. the linked list locations, whilst other row addresses are closed off.

Deleting a record

Care must be exercised when an entry in a hash table is deleted.

Closed hashing

In closed hashing, collisions are resolved by rehashing and storing the colliding record in another row whose table index is the rehash value.

However, if the entry at the original hash value table index or any of the rehash value table entries are deleted and the deleted entry remains empty, searching can be stopped prematurely before all potential matching entries have been examined.

Therefore, a deleted entry must be distinguishable from an entry that has never been used. This requires a special marker to be present in the key field part of the hash table entry when the entry is not in use. The special marker will use one value to indicate that this entry has never been used and a different value to indicate that it has been used but the entry has been deleted.

The special marker values should not use any value that potentially could occur in the key fields of the data set to be stored in the hash table.

Open hashing

In open hashing, collisions are resolved by chaining the colliding record to the table entry slot whose index is the hash. Care must be taken when deleting the record in the table row when the row has a nonempty chain.

A special marker can be left in the key field to signal that there is at least one record in a chain (linked list) attached to the row so that a search does not fail to look at the chain when seeking a match.

There are at least two alternatives that do not rely on a special marker.

Single licence - Abingdon School

In alternative one, the search examines the pointer field of an empty slot to see if a chain is attached.

In alternative two, the first record in the chain is moved into the table slot whilst preserving its link to the rest of the chain.

Questions

9 An empty hash table is set up for open hashing. The following hashing function is to be used to store variable names beginning with an uppercase letter in range A...Z, as well as other information.

`H(VariableName)` = (code for first letter of `VariableName x 11) Mod M`

Where `M` is the number of rows in the hash table.

Using `M = 5` and coding letters of the alphabet as follows, A=1, B=2, ..., Z=26 show the contents of the hash table after inserting the following variable names:

CHECK, OVERTIME, MAIN, P, URL, TAXRATE, INDEX, N, GENDER

You may ignore in your answer the other information associated with each variable name.

10 (a) Using the hashing algorithm expressed in pseudo-code below, calculate the hash value for the hash key `'PEN'` stored in string variable `Key`. You will need access to an ASCII code table to map characters to their equivalent ASCII codes. This is performed in the pseudo-code by the function `Ord`. The `Length` function returns the number of characters in the string. The symbol `'*'` means multiply.

```
Sum ← 0
For i ← 0 To Length(Key)- 1
  Sum ← Sum + Ord(Key[i]) * Ord(Key[i])
EndFor
HashValue ← Sum Mod 523
```

(b) Now repeat the exercise with the made-up word `'NEP'`.

(c) Can you see that there is a problem? What is the problem?

(d) Describe **two** ways that could be used to overcome this problem.

11 Explain why care must be exercised when deleting an entry in a hash table that uses closed hashing and on which searching occurs after deletion.

12 A person owns *n* distinct pairs of socks, which are kept in an unmatched pile in a drawer.
Individual socks are pulled from the drawer blindly, then identified and placed in a separate pile according to identity.

(a) How many individual socks must the person pull from the drawer to ensure that two are pulled that match?

(b) In what respect does this process resemble a hash table and open hashing?

Single licence - Abingdon School

Questions

13 In an application, student records are identified by their key field, the student's unique learner number (`ULN`) consisting of eight digits, e.g. 34567890. The application has to process a `ULN` allocated in the range 1000000 to 99999999 but it will never have to deal with more than 500 ULNs.

(a) Explain why when storing student records in a table in memory it would not be sensible to use the `ULN` as the row address for the record, e.g. 34128496.

(b) Explain why the use of a hash table would be a better option for this application.

14 (a) State **two** advantages of using hashing and the hash table approach over the alternative approach which just stores records in an ordinary table starting from the first row.

(b) It is noticed that after inserting many records into a hash table that uses closed hashing, searches are taking much longer than they did.

(i) Explain why this may be the case

(ii) Suggest a solution that could potentially restore searching times to what they were.

15 Explain why it is necessary to store the hash key in a hash table.

In this chapter you have covered:

- The concept of a hash table and its uses.
- Applying simple hashing algorithms.
- What is meant by a collision and how collisions are handled using rehashing.

Single licence - Abingdon School

2 Fundamentals of data structures

2.7 Dictionaries

Learning objectives:

- *Be familiar with the concept of a dictionary*
- *Be familiar with simple applications of dictionaries, for example information retrieval, and have experience of using a dictionary data structure in a programming language.*

Key concept

Dictionary:
A dictionary is a collection of items in which each item has both a key and a value and for which

- keys are mapped to values
- keys must be comparable
- keys must be unique

and standard operations are

- insert(key, value)
- find(key)
- delete(key)

2.7.1 Dictionaries

Concept

A dictionary is a collection of key-value pairs in which the value is accessed via its associated key. This key is unique.

Dictionaries are covered extensively in *Chapter 2.1.4*.

Information retrieval

Figure 2.7.1.1 shows an example of a dictionary that maps country to capital.

This dictionary may be used to look up the capital of a given country.
The key is country and the value the capital, e.g. country = "Lithuania", capital = "Vilnius".

The operation to find the capital of "Lithuania" would be `find("Lithuania")`.

Dictionary that maps country to capital, i.e. country → capital:

{"Nigeria":"Abuja", "Ghana":"Accra", ..., "Azerbaijan":"Baku", ..., "Lithuania":"Vilnius", ..., "Croatia":"Zagreb"}

Figure 2.7.1.1 Dictionary that maps country to its capital

Questions

1. Explain how to create a dictionary to enable the ASCII code for the characters in the ASCII set of characters which are not control characters to be looked up.

In this chapter you have covered:

- The concept of a dictionary
- Simple applications of dictionaries, for example information retrieval

Single licence - Abingdon School

2 Fundamentals of data structures

2.8 Vectors

Learning objectives:

- *Be familiar with the concept of a vector and the following notations for specifying a vector*
- *Dictionary representation of a vector*
- *List representation of a vector*
- *1-D array representation of a vector*
- *Visualising a vector as an arrow*
- *Vector addition and scalar-vector multiplication*
- *Convex combination of two vectors, u and v*
- *Dot or scalar product of two vectors*
- *Applications of dot product.*

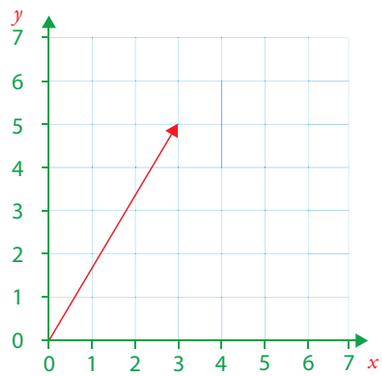


Figure 2.8.1.1 Vector (3, 5) as an arrow

Key concept

Vector:
A vector in an algebraic sense is a collection of objects which may be added together and multiplied (scaled) by numbers called scalars.

2.8.1 Vectors

Concept

A vector in an algebraic sense is a collection of objects which may be added together and multiplied (scaled) by numbers called scalars.
A scalar is a number that is used to multiply each component or element of a vector by, to produce a new vector.
The components of a vector, the objects, can be real numbers or integers or subsets of both of these, e.g. nonnegative integers.

We use a notation to express the fact that a vector is a collection.

The convention in mathematics is to bracket the components as shown below with vector A and vector B.

$$A = (2, 5, 7) \qquad B = (7, 9, 11)$$

Vectors which have the same number of components may be added to create a new vector with the same number of components.

The first components are added together, then the second, and so on, e.g.

$$\begin{aligned} A + B &= (2, 5, 7) + (7, 9, 11) \\ &= (2 + 7, 5 + 9, 7 + 11) \\ &= (9, 14, 18) \end{aligned}$$

A vector may be scaled by multiplying each of its components by a scalar, e.g.

$$5 \bullet A = 5 \bullet (2, 5, 7) = (10, 25, 35))$$

The scalar used in this example is the number 5. Vector A has been scaled by scalar 5.
A vector in a geometric sense is still a mathematical object but one which in some cases can be visualized as an arrow with a direction and a magnitude as shown in *Figure 2.8.1.1*.
The arrow interpretation maps onto physical quantities which possess both magnitude and size such as forces. Adding two vectors which represent forces yields another vector representing the resultant of the two forces.

The vector (3, 5) in *Figure 2.8.1.1* can be interpreted as representing a geometric point in space, the point with x coordinate 3 and y coordinate 5.

For more examples of vectors and their uses see *Chapter 2.1.2*.

Single licence - Abingdon School

Key concept

Vector:
A vector in a geometric sense is a mathematical object which in some cases can be visualized as an arrow with a direction and a magnitude.
The arrow interpretation maps onto physical quantities which possess both magnitude and size such as forces.

Key concept

Vector:
The vector (a, b) can be interpreted as representing a geometric point in space, the point with x coordinate a and y coordinate b.

Key concept

Algebraic dot product:
$u \bullet v = u_1v_1 + u_2v_2 + \ldots + u_nv_n$.
where $u = [u_1, \ldots, u_n]$
and $v = [v_1, \ldots, v_n]$.

We can describe a vector by the number and type of its elements or components.
For example,

the vector C = (2.0, 5.0, 7.0), is described as a 3-vector over ℝ consisting of three elements of type ℝ.

We represent this vector in computer science as follows

- as a dictionary of key:value pairs, e.g. {0 : 2.0, 1 : 5.0, 2: 7.0}
- as a list, e.g. [2.0, 5.0, 7.0]
- as a function f: S → ℝ where for example, the domain could be the set S = {0, 1, 2} and the co-domain, ℝ, the set of real numbers; the input is drawn from the domain and the output from the co-domain in a mapping as shown below (↦ means maps to)

  0 ↦ 2.0

  1 ↦ 5.0

  2 ↦ 7.0

- as a one-dimensional array, `Array[0..2] Of Real.`

Visualising a vector as an arrow

A 2-vector [2.0, 5.0] over ℝ can be represented by an arrow with its tail at the origin of an x-y graph (*Figure 2.8.1.1*) and its head at (2.0, 5.0).

If the vector represents a quantity with direction and magnitude, we write a vector u visualised this way as

$$\vec{u} = (2.0, 5.0)$$

Convex combination of two vectors, u and v

Convex combination of two vectors u and v is an expression of the form

$\alpha u + \beta v$ where $\alpha, \beta \geq 0$ and $\alpha + \beta = 1$.

This applies to both algebraic and the geometric vector interpretations.

See *Chapter 2.1.2* for an application of the algebraic interpretation to images.

Dot or scalar product of two vectors, u and v

The algebraic dot product of two vectors, u and v,

$$u \bullet v = u_1v_1 + u_2v_2 + \ldots + u_nv_n.$$

where $u = [u_1, \ldots, u_n]$ and $v = [v_1, \ldots, v_n]$

This is a scaling of one vector by another. The notation [ ] is used in place of ( ).

See *Chapter 2.1.2* for an example of its use.

Also see Page 248 for another example of its use.

Single licence - Abingdon School

Other Applications of dot product

Graphics programmers often need to find the angle between two given vectors. This can be done using the geometric dot product of the two vectors. The two vectors are treated as having magnitude and direction.

Figure 2.8.1.2 shows a triangle of vectors, $\vec{u}$, $\vec{v}$, and $\vec{w}$, with angle θ between $\vec{u}$ and $\vec{v}$.

By the rules of vector addition, $\vec{u} = \vec{v} + \vec{w}$

and the geometric dot product of two vectors, $\vec{u}$ and $\vec{v}$, is given by

$$\vec{u} \cdot \vec{v} = ||\vec{u}||\ ||\vec{v}|| \cos(\theta)$$

where $||\vec{u}||$ is the length or magnitude of vector $\vec{u}$ and $||\vec{v}||$ is the length of vector $\vec{v}$.

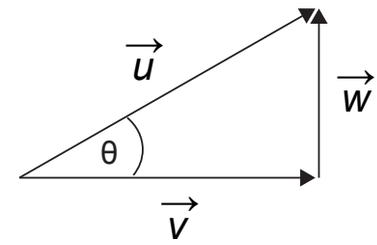


Figure 2.8.1.2 Triangle of vectors

The length of a vector $\vec{u}$ is given by $\sqrt{(\vec{u} \cdot \vec{u})}$.

Figure 2.8.1.3 shows an example where $\vec{u} = (3.0, 5.0)$ and $\vec{v} = (3.0, 0)$.

$$||\vec{u}|| = \sqrt{(3.0^2 + 5.0^2)},\ ||\vec{v}|| = 3.0$$

$$\vec{u} \cdot \vec{v} = (3.0, 5.0) \cdot (3.0, 0) = 3.0 \times 3.0 + 5.0 \times 0 = 9.0$$

$$9.0 = \sqrt{(3.0^2 + 5.0^2)} \times 3.0 \times \cos(\theta)$$

$$\cos(\theta) = \frac{9.0}{\sqrt{(3.0^2 + 5.0^2)} \times 3.0}$$

$$= 0.5145 \text{ to 4 decimal places}$$

$$\theta = 59.0° \text{ to 1 decimal place}$$

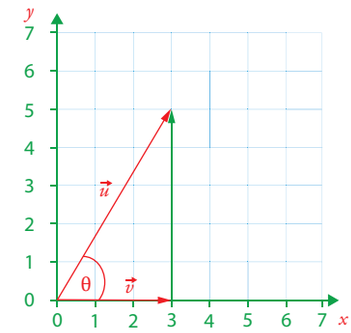


Figure 2.8.1.3 Triangle of vectors

Questions

1. Add the two vectors A = (4, 8, 12) and B = (14, 16, 19).
2. Multiply the vector A = (1, 3, 5, 9) by the scalar 3.
3. Calculate the dot product of the vectors A = (3.0, 5.0, -1.0) and B = (2.0, 6.0, 7.0)
4. Calculate the convex combination of the two vectors A = (4, 8) and B = (14, 16)

   0.2A + 0.8B
5. What is the angle between the two vectors A = (1.0, 1.7321) and B = (1.0, 0) ?

Key concept

Geometric dot product:
The geometric dot product of two vectors, $\vec{u}$ and $\vec{v}$, is given by

$$\vec{u} \cdot \vec{v} = ||\vec{u}||\ ||\vec{v}|| \cos(\theta)$$

In this chapter you have covered:

- The concept of a vector and the following notations for specifying a vector
- Dictionary representation of a vector
- List representation of a vector
- 1-D array representation of a vector
- Visualising a vector as an arrow

Single licence - Abingdon School

- Vector addition and scalar-vector multiplication
- Convex combination of two vectors, u and v
- Dot or scalar product of two vectors
- Applications of dot product
  - finding the angle between two vectors.

Single licence - Abingdon School

3 Fundamentals of algorithms

3.1 Graph-traversal

Learning objectives:

- *Be able to trace breadth-first and depth-first search algorithms and describe typical applications of both.*

Key concept

Graph traversal:
Graph traversal means to visit its vertices in some sequence.

3.1.1 Simple graph-traversal algorithms

Breadth-first search

In breadth-first search, we visit the starting vertex and then on the first pass visit all the vertices directly connected to it.

In the second pass, we visit vertices that are two edges away from the starting node. When presented with a choice of vertices, the one whose label is alphabetically or numerically smaller is chosen.

With each new pass, we visit vertices that are one more edge away - *Figure 3.1.1.1*.

To avoid visiting a vertex more than once which is possible if there are cycles in the graph, it is marked as visited - see *Figure 3.1.1.2* which shows that our graph has a cycle, its path is marked in blue.

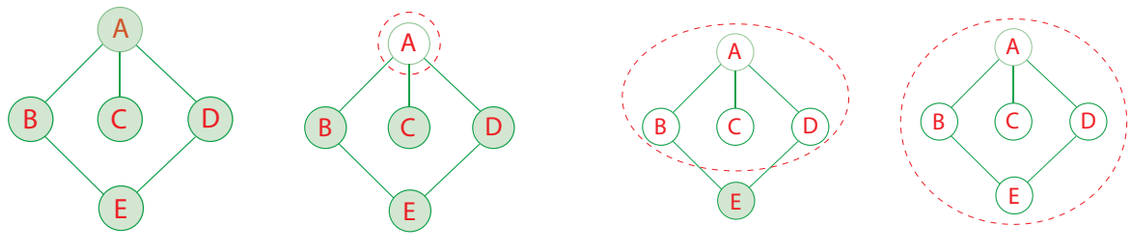


Figure 3.1.1.1 Each step of the search expands to include vertices one more edge away until all vertices are visited

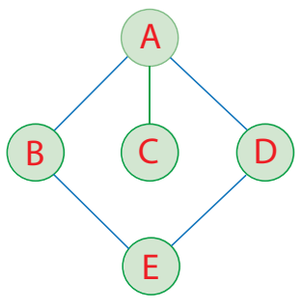


Figure 3.1.1.2 Cycle marked in blue

The steps of the breadth-first algorithm are shown in *Figure 3.1.1.3*. (See *Chapter 2.2.1* if you need to remind yourself how queues work).

```
Mark all vertices as unvisited
Choose a starting vertex StartingVertex
Visit StartingVertex and mark as visited
Enqueue StartingVertex in queue Q
While Q not empty
  CurrentVertex ← Remove vertex from front of Q
  For each neighbouring vertex Neighbour of CurrentVertex
    If Neighbour not visited
      Then
        Visit Neighbour and mark as visited
        Enqueue Neighbour in Q
    EndIf
  EndFor
EndWhile
```

Figure 3.1.1.3 Breadth-first search algorithm

Task

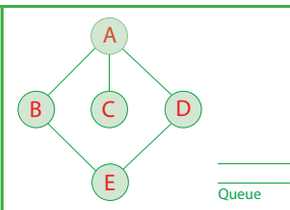
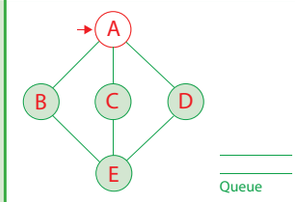
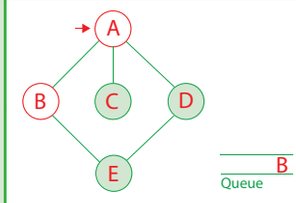
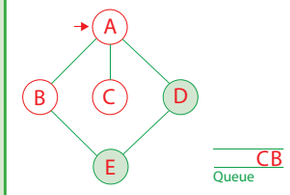
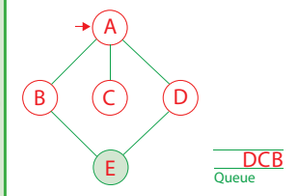
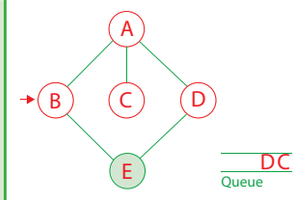
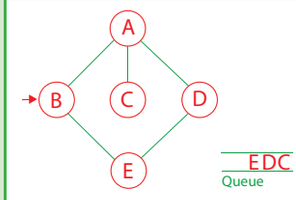
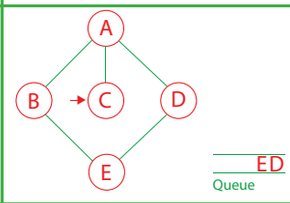
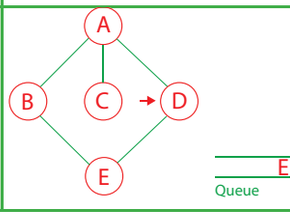
1. Watch video on breadth-first search algorithm, URL: https://www.youtube.com/watch?v=QRq6p9s8NVg

Single licence - Abingdon School

Figure 3.1.1.4 Trace of breadth-first search algorithm

Task

2 Hand trace the algorithm shown in *Figure 3.1.1.3* and confirm the steps shown in *Figure 3.1.1.4*.

Step 1		Initialise queue.
Step 2		We choose to start at vertex A and mark as visited. It is enqueued and then dequeued immediately. Red arrow marks current vertex.
Step 3		Find an unvisited vertex which A is connected to, B, mark it as visited, and enqueue it. Vertex B is chosen alphabetically from amongst B, C, and D.
Step 4		The next vertex that A is connected to and unvisited is C, it is marked as visited and enqueued.
Step 5		The next vertex that A is connected to and unvisited is D, it is marked as visited and enqueued.
Step 6		All vertex A's adjacent vertices are now visited, so we dequeue and find B (adjacent means connected to directly). Red arrow indicates current vertex is B
Step 7		We now visit all B's adjacent unvisited nodes. There is only one, E. It is marked as visited and enqueued. No unvisited vertices of B remain.
Step 8		All vertex B's adjacent vertices are now visited, so we dequeue and find C. Red arrow indicates current vertex is C. C has no adjacent unvisited vertices.
Step 9		Next we dequeue and find D. Red arrow indicates current vertex is D. D has no adjacent unvisited vertices. Next we dequeue again (not shown), E becomes current vertex(not shown). We are done because E has no adjacent unvisited vertices.

Single licence - Abingdon School

Questions

1 Hand trace by completing a copy of *Table 3.1.1.1* the breadth-first search algorithm shown in *Figure 3.1.1.6* for the graph in *Figure 3.1.1.5*. The starting vertex is A. Assume that all the vertices except A and B have been marked unvisited. When presented with a choice of vertices, the one whose label is alphabetically smaller is chosen. The first two entries have been completed for you.

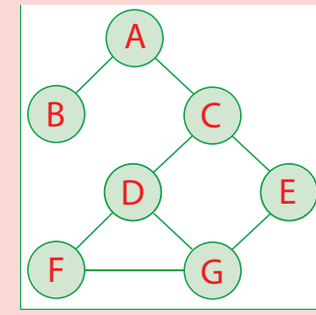


Figure 3.1.1.5

```
Visit StartingVertex and mark as visited
Enqueue StartingVertex in queue Q
While Q not empty
  CurrentVertex ← Remove vertex from front of Q
  For each neighbouring vertex Neighbour of CurrentVertex
    If Neighbour not visited
      Then
        Visit Neighbour and mark as visited
        Enqueue Neighbour in Q
    EndIf
  EndFor
EndWhile
```

Figure 3.1.1.6

CurrentVertex	Neighbours	Visited vertices	Queue Q (Rear Front)
A	B, C	A	A
A	B, C	A, B	B

Table 3.1.1.1

2 If the algorithm is altered to output the current vertex, what is the sequence of letters that is output for the graph in *Figure 3.1.1.5*?

Depth-first search

In depth-first search, the starting vertex is visited and then the algorithm proceeds to follow edges through the graph until a dead end is reached. In an undirected graph, a vertex is a dead end if all the vertices adjacent to it have already been visited. In a directed graph, if a vertex has no outgoing edges, we also have a dead end.

When a dead end is reached, the algorithm backtracks or backs up along the path until an unvisited adjacent vertex is found and it then continues in that new direction. The algorithm is finished searching when it backtracks to the starting vertex and all the vertices adjacent to it have been visited. When presented with a choice of vertices, the one whose label is alphabetically or numerically smaller is chosen.
To avoid visiting a vertex more than once which is possible if there are cycles in the graph, it is marked as visited.

Figure 3.1.1.7 Depth-first searching of a maze to identify a path between entrance and exit

Single licence - Abingdon School

Figure 3.1.1.8 shows an algorithm for depth-first search.
It is a recursively defined algorithm because it contains a call to itself.
The recursion relies on the program's call stack to keep track of where it has been in the graph so that it can properly backtrack when it reaches dead ends. (See *Chapter 2.3.1* if you need to remind yourself how stacks work).

Information

GraphDraw may be obtained from
www.educational-computing.co.uk

```
Subroutine DepthFirstSearch(G, CurrentVertex)
  {G is the graph to be searched}
  (CurrentVertex is the current vertex}
  Visit CurrentVertex
  Mark CurrentVertex as visited
  For each neighbouring vertex Neighbour of CurrentVertex
    If Neighbour not visited
      Then DepthFirstSearch(G, Neighbour)
    EndIf
  EndFor
EndSubroutine
```

Figure 3.1.1.8 Depth-first search algorithm

Task

3 Watch video on depth-first search algorithm, URL: https://www.youtube.com/watch?v=iaBEKo5sM7w

Vertex States

ID	Label	Discovered?	Completely Explored?
1	A	True	False
2	B	False	False
3	C	False	False
4	E	False	False
5	I	False	False
6	G	False	False
7	D	False	False
8	F	False	False
9	H	False	False

Figure 3.1.1.9 Vertex states for the graph in Figure 3.1.1.10

Figure 3.1.1.11 shows a trace of depth-first search obtained using GraphDraw.

Figure 3.1.1.9 shows the vertex states for the first screen shot in Figure *3.1.1.11*. The algorithm uses the vertex IDs shown in *Figure 3.1.1.9* choosing the smaller numerical ID when there is a choice of ID. The column "Discovered?" corresponds to "visited" in the algorithm in *Figure 3.1.1.8*.
Figure 3.1.10 shows the stack for the first four calls to DepthFirstSearch

The Ready List in GraphDraw corresponds to the stack.

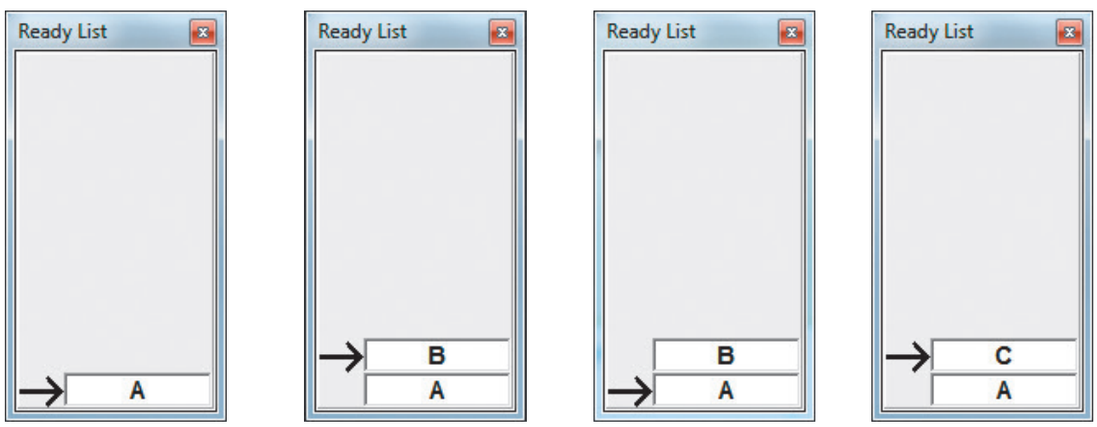


Figure 3.1.1.10 The ready list in GraphDraw corresponds to the stack

Single licence - Abingdon School

1 - A	2 - B	3 - C	4 - E	5 - I	6 - G	7 - D	8 - F	9 - H

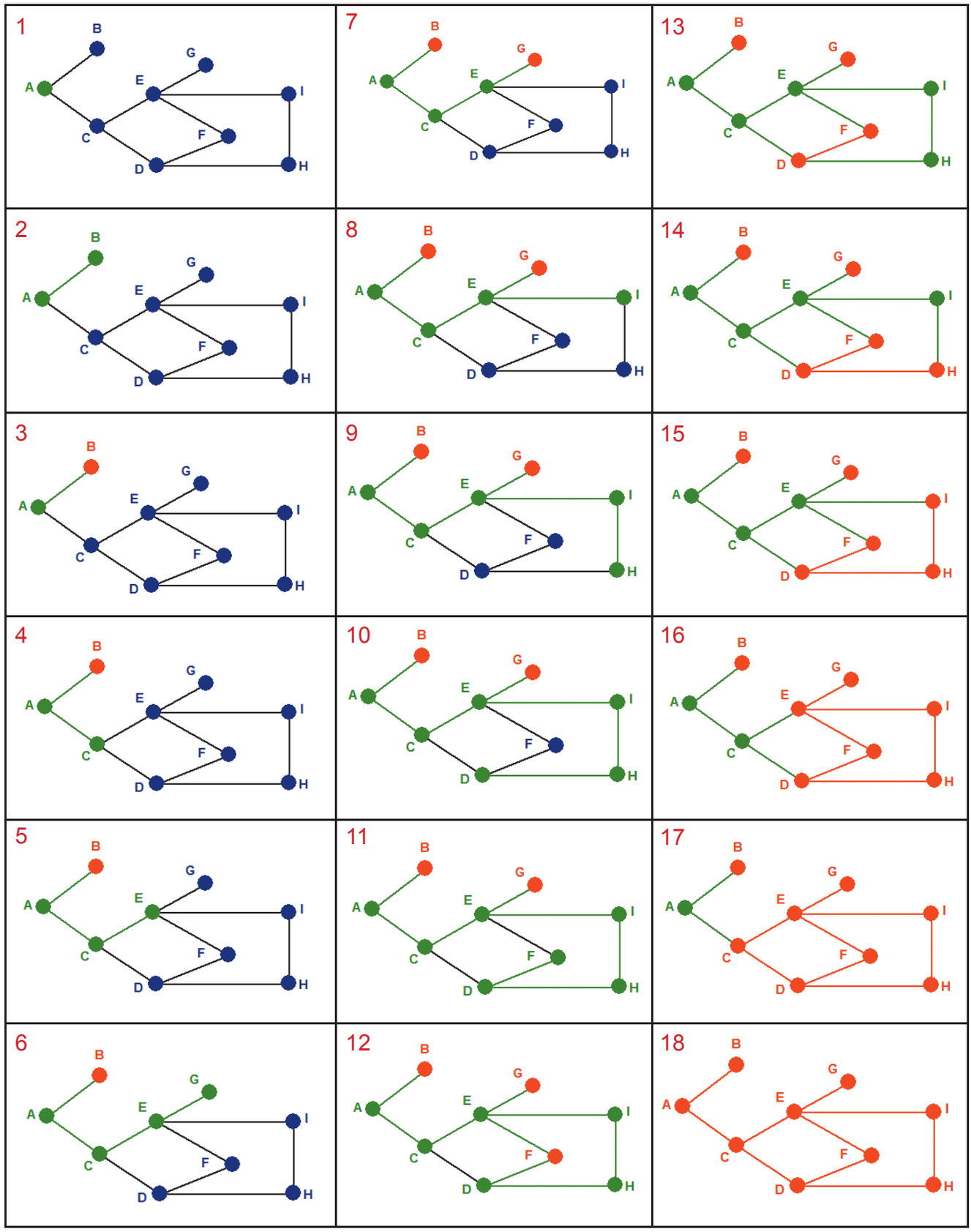


Figure 3.1.1.11 Depth-first search trace using GraphDraw, visited vertices are marked in green, and vertices that are a dead end or are being returned to in order to try another path are marked in red. The numerical vertex ID which is used when making a choice of next vertex is shown at the top

Single licence - Abingdon School

Questions

3 Hand trace by completing a copy of *Table 3.1.1.2* the depth-first search algorithm shown in *Figure 3.1.1.13* for the graph in *Figure 3.1.1.12*. The starting vertex is A. Assume that all the vertices except A and B have been marked unvisited. When presented with a choice of vertices, the one whose label is alphabetically smaller is chosen. The first four entries have been completed for you.

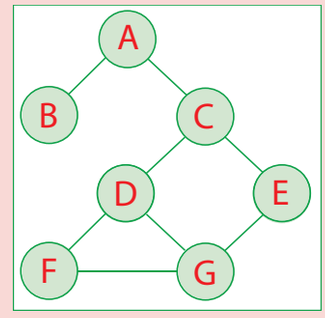


Figure 3.1.1.12

```
Subroutine DepthFirstSearch(G, CurrentVertex)
  {G is the graph to be searched}
  (CurrentVertex is the current vertex}
  Visit CurrentVertex
  Mark CurrentVertex as visited
  For each neighbouring vertex Neighbour of CurrentVertex
    If Neighbour not visited
      Then DepthFirstSearch(G, Neighbour)
    EndIf
  EndFor
EndSubroutine
```

Figure 3.1.1.13

CurrentVertex	Neighbours	Visited vertices	Stack S Base Top
A	B, C	A	A
B	A	A, B	A B
A	B, C	A, B	A
C	A, D, E	A, B, C	A C

Table 3.1.1.2

4 If the algorithm is altered to output the current vertex, what is the sequence of letters that is output for the graph in *Figure 3.1.1.12*?

Single licence - Abingdon School

Applications

Breadth-first search

In breadth-first search vertices are visited in order of their distance from the start vertex. This implies that breadth-first search always finds a shortest possible path (with respect to the number of edges) to the goal, and does not consider longer paths in between.

Crawling websites consists of fetching a page, extracting the hyperlinks in the page, and then systematically fetching all of the pages that are hyperlinked. The process is repeated to an arbitrary depth, depending on what is required.

This is the way that most search engines that index the Web work.

The basic algorithm for a web crawl can be expressed as a breadth-first search.

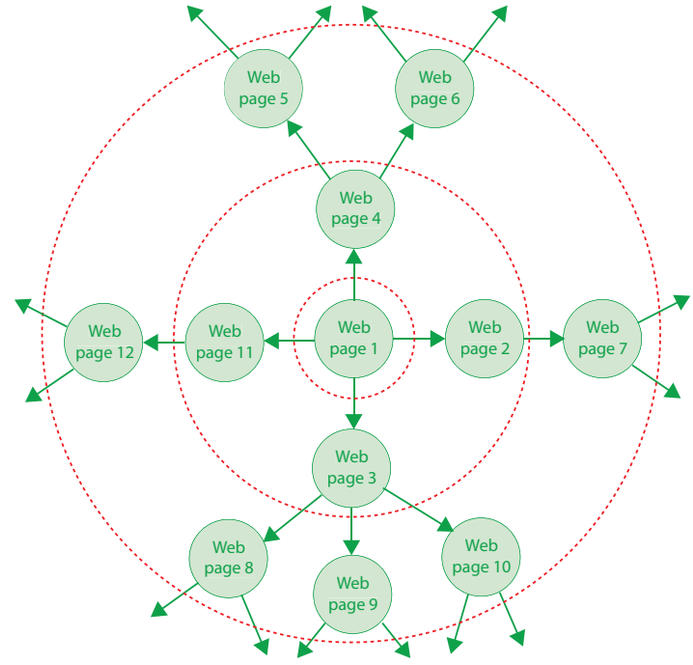


Figure 3.1.1.14 Web crawling using breadth-first search

The starting vertex is the initial web page and the set of neighbouring vertices, are those linked to by a hyperlink - *Figure 3.1.1.14*.

Breadth-first search is a reasonable approach to exploring a small part of the Web.

Breadth-first search is very fast when the graph is large but the goal is close to the starting vertex. In this scenario depth-first search can get lost very easily in more distant regions of the graph.

Information

Web crawling:
See Amazon's Common Crawl Corpus which features more than 5 billion web pages
http://amzn.to/1a1mXXb

Depth-first search

A maze is a series of corridors, dead ends, and junctions.

The goal for a given maze is to navigate from some start position, e.g. the entrance to the maze, to some destination which could be an exit or in *Figure 3.1.1.15* a particular location within the maze. We also desire that each corridor should be traversed no more than once in each direction.

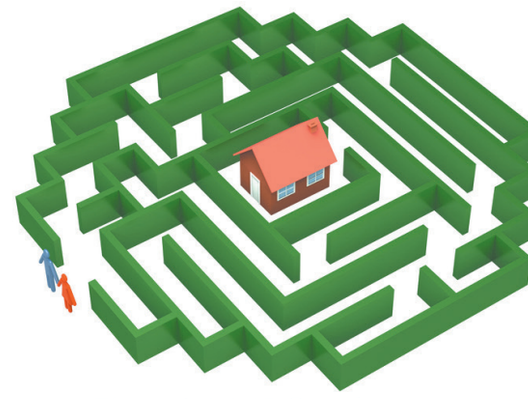


Figure 3.1.1.15 Complex maze

Depth-first search satisfies this goal and in the process avoids traversing a path more than once in each direction.

Figure 3.1.1.16 shows the stage that is reached by a depth-first search of the graph starting from vertex A and seeking destination vertex I. The edge A-B was traversed once in each direction as was the edge E-G before vertex I was found.

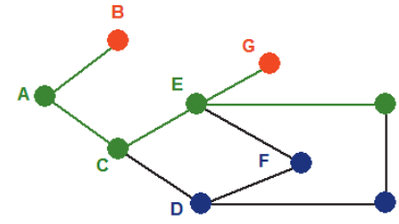


Figure 3.1.1.16 Starting from vertex A with goal of reaching vertex I

Table 3.1.1.3 shows how a graph can represent a maze.

Figure 3.1.1.17 shows a modified depth-first search algorithm for navigating a maze represented by graph G.

DepthFirstSearch is called initially with parameters G, StartVertex and GoalVertex set to the given maze graph, the entrance to the maze, and the exit from the maze.

Single licence - Abingdon School

Vertices	Edges
1. Vertex for a starting point, i.e. entrance 2. Vertex for a finishing point, i.e. exit 3. Vertices for all dead ends 4. Vertices for all the points in the maze where more than one path can be taken, i.e. junction.	1. Connect the vertices according to the paths in the maze.

Table 3.1.1.3 Table showing how to map vertices and edges onto a maze to create a graph representation of the maze

```
Subroutine DepthFirstSearch(G, CurrentVertex, GoalVertex)
  {G is the graph to be searched}
  (CurrentVertex is the current vertex}
  If CurrentVertex = GoalVertex
    Then Exit "Goal found!"
  EndIf
  Visit CurrentVertex
  Mark CurrentVertex as visited
  For each neighbouring vertex Neighbour of CurrentVertex
    If Neighbour not visited
      Then DepthFirstSearch(G, Neighbour)
    EndIf
  EndFor
EndSubroutine
```

Figure 3.1.1.17 Modified depth-first search algorithm for navigating a maze

Questions

5 Explain one application of each of the following
(a) breadth-first search (b) depth-first search

In this chapter you have covered:

- Tracing
  - breadth-first search algorithm
  - depth-first search algorithm
  - typical applications of both.

Single licence - Abingdon School

3 Fundamentals of algorithms

3.2 Tree-traversal

Learning objectives:

- *Be able to trace the tree-traversal algorithms:*
  - *pre-order*
  - *post-order*
  - *in-order*
- *Be able to describe uses of tree-traversal algorithms*

3.2.1 Simple tree-traversal algorithms

Tree-traversal

Traversal of a tree means to visit each node of the tree in some order.

Usually, the visit means do something with the node, e.g. output its data value.

For **binary trees**, there are **three** principle ways of traversing the tree:

1. Pre-order traversal
2. In-order traversal
3. Post-order traversal

These traversals can be specified recursively. The trick is to realise that a binary tree can be defined recursively.

The recursive definition states:

A binary tree is either

1. Empty

Or

2. Consists of
   a. A root node
   b. A left sub-tree
   c. A right- sub-tree.

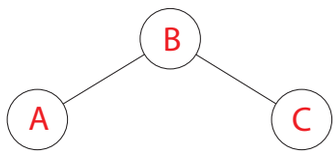


Figure 3.2.1.1 Three node binary tree

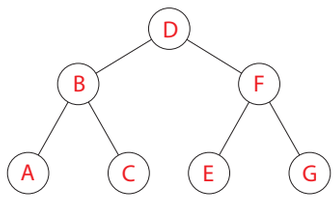


Figure 3.2.1.2 Seven node binary tree

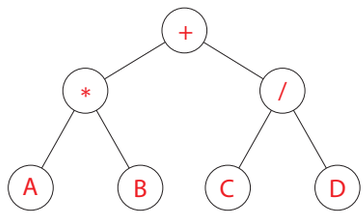


Figure 3.2.1.3 Expression tree

Pre-order Traversal

The pre-order traversal algorithm may be expressed in structured English as follows

1. If binary tree empty do nothing
2. Otherwise
   a. Visit the root (e.g. output the value of the data item at the root)
   b. Traverse the left sub-tree in pre-order
   c. Traverse the right sub-tree in pre-order

If the binary tree in *Figure 3.2.1.1* is traversed in pre-order then the output of the pre-order traversal algorithm is BAC.

If the binary tree in *Figure 3.2.1.2* is traversed in pre-order then the output of the pre-order traversal algorithm is DBACFEG.

If the binary tree in *Figure 3.2.1.3* is traversed in pre-order then the output of the pre-order traversal algorithm is +*AB/CD.

In-order Traversal

The in-order traversal algorithm may be expressed in structured English as follows

1. If binary tree empty do nothing
2. Otherwise
   a. Traverse the left sub-tree in in-order
   b. Visit the root (e.g. output the value of the data item at the root)
   c. Traverse the right sub-tree in in-order

Single licence - Abingdon School

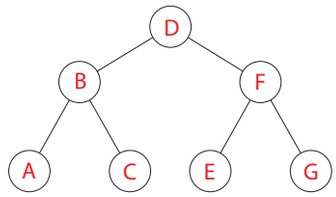


Figure 3.2.1.2 Seven node binary tree (shown again)

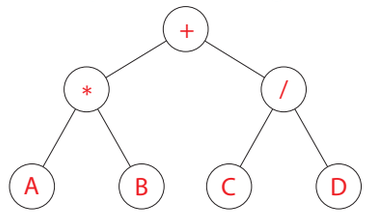


Figure 3.2.1.3 Expression tree (shown again)

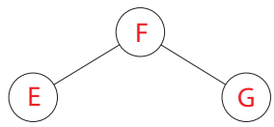


Figure 3.2.1.4 Three node binary tree

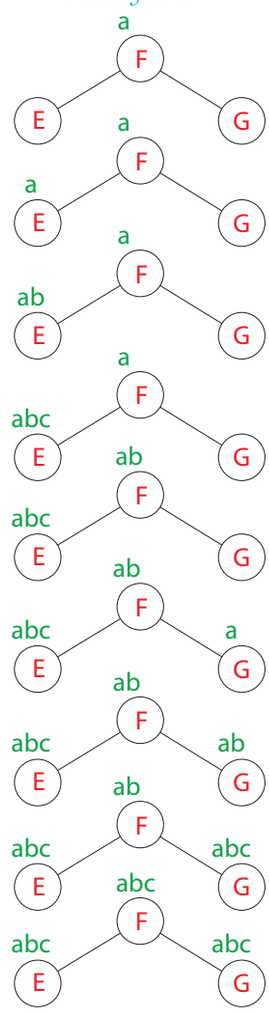


Figure 3.2.1.5 A trace of pre-order traversal of a binary tree

If the binary tree in *Figure 3.2.1.1* is traversed in in-order then the output is ABC.

If the tree in *Figure 3.2.1.2* is traversed in in-order then the output is ABCDEFG.

If the binary tree in *Figure 3.2.1.3* is traversed in in-order then the output is A*B+C/D.

Post-order Traversal

The post-order traversal algorithm may be expressed in structured English as follows

1. If tree empty do nothing
2. Otherwise
    a. Traverse the left sub-tree in post-order
    b. Traverse the right sub-tree in post-order
    c. Visit the root (e.g. output the value of the data item at the root)

If the tree in *Figure 3.2.1.1* is traversed in post-order then the output is ACB.

If the tree in *Figure 3.2.1.2* is traversed in post-order then the output is ACBEGFD.

If the tree in *Figure 3.2.1.3* is traversed in post-order then the output is AB*CD/+ .

Tracing a particular traversal of a binary tree

There are two ways to help you trace a particular traversal of a binary tree.

First method

The first places labels on nodes to aid remembering how far the trace has reached.

Figure 3.2.1.5 shows a trace for a pre-order traversal through the binary tree shown in *Figure 3.2.1.4*. For example, the first tree in the trace shows root node F labelled a. This corresponds to stage a of the algorithm shown in *Figure 3.2.1.6*. The next tree shows that the pre-order traversal of the left sub-tree has begun, its root node E has been visited and is therefore labelled a. When this traversal of F's left sub-tree is done the node E is labelled abc. Now the root F can be labelled ab because the first two stages of the algorithm are complete. This continues until each node is labelled abc. In the process the output FEG is produced.

1. If tree empty do nothing
2. Otherwise
    a. Visit the root (e.g. output the value of the data item at the root)
    b. Traverse the left sub-tree in pre-order
    c. Traverse the right sub-tree in pre-order

Figure 3.2.1.6 Pre-order tree traversal algorithm

Single licence - Abingdon School

Second method

This involves drawing an outline around the binary tree as shown in *Figure 3.2.1.7*, for example, and marking each node with a dot according to the traversal algorithm as shown in each of *Figure 3.2.1.7*, *Figure 3.2.1.8* and *Figure 3.2.1.9*.

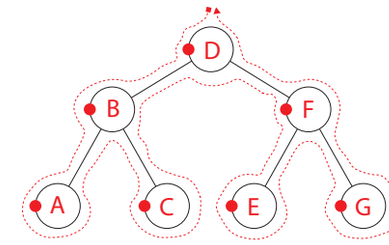


Figure 3.2.1.7 Pre-order traversal of a binary tree using an outline around the tree as a guide

Pre-order traversal

Place a dot on each node as shown in *Figure 3.2.1.7* then starting at the root, draw an outline around the binary tree in an anti-clockwise direction as shown. As you pass to the left of a node (where the red dot is marked), output the data in that node, i.e. DBACFEG.

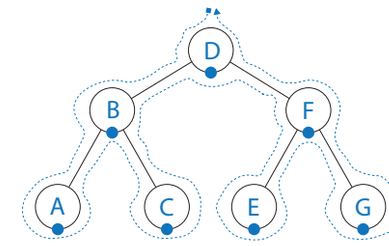


Figure 3.2.1.8 In-order traversal of a binary tree using an outline around the tree as a guide

In-order traversal

Place a dot on each node as shown in *Figure 3.2.1.8* then starting at the root, draw an outline around the binary tree in an anti-clockwise direction as shown. As you pass underneath a node (where the blue dot is marked) output the data in that node, i.e. ABCDEFG.

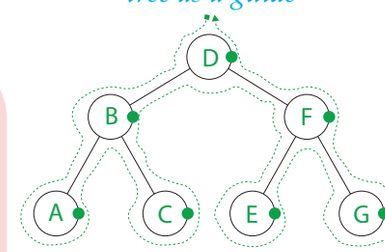


Figure 3.2.1.9 Post-order traversal of a binary tree using an outline around the tree as a guide

Post-order traversal

Place a dot on each node as shown in *Figure 3.2.1.9* then starting at the root, draw an outline around the binary tree in an anti-clockwise direction as shown. As you pass to the right of a node (where the green dot is marked) output the data in that node, i.e. ACBEGFD.

Questions

1 State the output from each of the following tree traversal algorithms applied to the binary expression tree shown in *Figure 3.2.1.10* (↑ means exponentiation, e.g. $3 \uparrow 2 = 3^2 = 9$).

(a) pre-order

(b) in-order

(c) post-order.

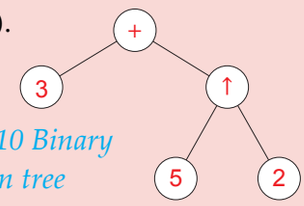


Figure 3.2.1.10 Binary expression tree

2 State the output from each of the following tree traversal algorithms applied to the binary expression tree shown in *Figure 3.2.1.11*.

(a) pre-order

(b) in-order

(c) post-order.

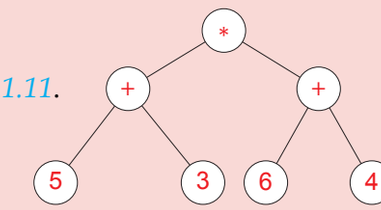


Figure 3.2.1.11 Binary expression tree

3 State the output from each of the following tree traversal algorithms applied to the binary expression tree shown in *Figure 3.2.1.12*.

(a) pre-order

(b) in-order

(c) post-order.

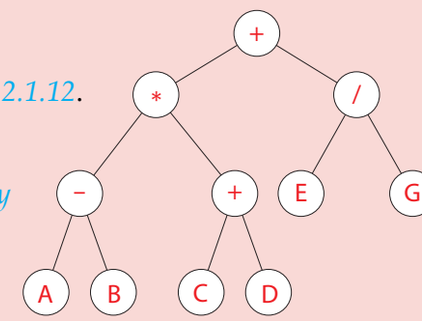


Figure 3.2.1.12 Binary expression tree

Single licence - Abingdon School

Uses of tree-traversal algorithms

Pre-order: copying a tree

Pre-order traversal may be used to copy a binary tree. We need to copy three things in the following order if the tree is non-empty:

1. the root
2. the left sub-tree
3. the right sub-tree

This is the order in which pre-order traversal works. First the parent is visited (the root) and then its children (the left and right sub-trees).

If the tree is empty we have nothing to copy.

Information

Copying a binary tree recursively given a pointer Ptr to its root:

CopyTree is called with a pointer to the root of the tree to be copied. It returns a pointer to the root of the copy or a nil pointer if the original tree was empty.

```
Function CopyTree(Ptr : PointerType): PointerType
  TreeNodePtr ← Nil
  If Ptr <> nil
    Then
      TreeNodePtr ← new Node
      TreeNodePtr.Data ← Ptr.data
      TreeNodePtr.LeftPtr ← CopyTree(Ptr.Left)
      TreeNodePtr.RightPtr ← CopyTree(Ptr.Right)
  Return TreeNodePtr
EndFunction
```

Pre-order: prefix expression from expression tree

Pre-order traversal may be used to convert an infix expression, e.g. 5 + 3, to its equivalent prefix form, +5 3. In infix form, the (dyadic) operator + is placed between the two operands, 5 and 3. In prefix form the (dyadic) operator + is placed before the two operands.

The infix expression to be converted is represented by its equivalent expression tree first as shown in the example in *Figure 3.2.1.13*.

To get the prefix expression the expression tree is traversed in pre-order.

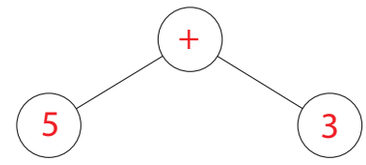


Figure 3.2.1.13 Expression tree for infix expression 5 + 3

In-order: binary search tree

A binary search tree is an ordered binary tree - see *Figure 3.2.1.14* and *Chapter 2.5.1*, page 276.

An in-order traversal of an ordered binary tree supplies the tree's node data in ascending order.

For example, traversing the binary search tree in *Figure 3.2.1.14* using in-order produces output

17 20 25 30 37 41 56.

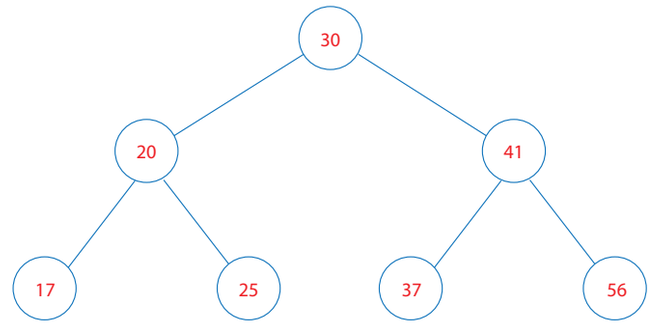


Figure 3.2.1.14 Binary search tree

Post-order: infix to RPN (Reverse Polish Notation)

Post-order traversal may be used to convert an infix expression, e.g. 5 + 3, to its equivalent expression in postfix or Reverse Polish Notation (RPN) form, 5 3 +. In infix form, the (dyadic) operator is placed between the two operands. In postfix form the (dyadic) operator is placed after the two operands.

Evaluating an RPN expression by computer is a relatively straightforward process. *Figure 3.2.1.15* shows how a stack can be used to evaluate an RPN expression.

Working from left to right, push each operand onto stack. On encountering an operator, pop top two items and apply operator. Push result onto stack.

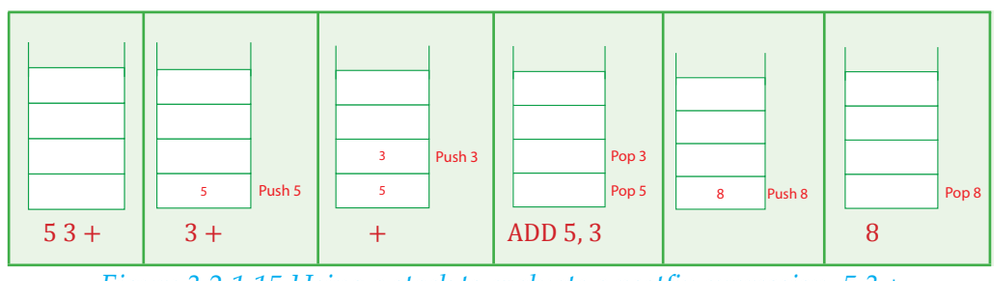


Figure 3.2.1.15 Using a stack to evaluate a postfix expression, 5 3 +

Single licence - Abingdon School

But how can an infix expression be converted into its postfix equivalent in the first place?

The answer: convert the infix expression to its equivalent expression tree first then traverse the binary tree in post-order. For example, the infix expression (16 – 6) / (3 + 2) is represented by the expression tree shown in *Figure 3.2.1.16*.

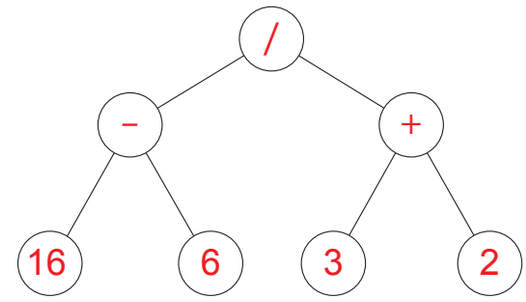


Figure 3.2.1.16 Expression tree for (16 - 6) / (3 + 2)

Its postfix form, 16 6 – 3 2 + / is generated by performing a post-order traversal of this tree.

The postfix form has no need for brackets because the order of evaluation is implicit. Brackets are required however to express the order of evaluation in the infix form.

If the brackets in (16 – 6) / (3 + 2) are removed, the order of evaluation is changed. The infix expression is now 16 – 6 / 3 + 2 with brackets removed. Operator precedence dictates that 6 / 3 is evaluated first. The new expression tree is shown in *Figure 3.2.1.17*. It post-order traversal generates the postfix expression 16 6 3 / – 2 +.

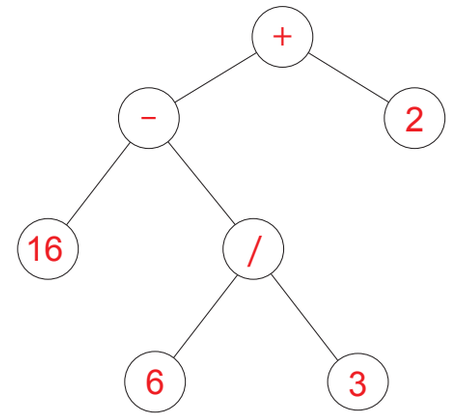


Figure 3.2.1.17 Expression tree for 16 – 6 / 3 + 2

Post-order: emptying a tree

Post-order traversal may be used to delete a binary tree.

We need to delete three things in the following

order if the tree is non-empty:

1. the left sub-tree
2. the right sub-tree
3. the root

This is the order in which post-order traversal works. First the children (the left and right sub-trees) are deleted, then the parent (the root).

If the tree is empty we have nothing to delete.

Information

Deleting a binary tree recursively given a pointer Ptr to its root:

DeleteTree is called with a pointer to the root of the tree to be deleted.

DeleteNode deletes a node given a pointer to it.

```
Procedure DeleteTree(Ptr : PointerType)
  If Ptr <> nil
    Then
      DeleteTree(Ptr.Left)
      DeleteTree(Ptr.Right)
      DeleteNode(Ptr)
EndProcedure
```

Questions

4 Describe **one** use for each of the following tree traversal algorithms

(a) pre-order

(b) in-order

(c) post-order.

Single licence - Abingdon School

In this chapter you have covered:

- Tracing the tree-traversal algorithms:
  - pre-order
    1. If binary tree empty do nothing
       Otherwise
       2. Visit the root (e.g. output the value of the data item at the root)
       3. Traverse the left sub-tree in pre-order
       4. Traverse the right sub-tree in pre-order
  - in-order
    1. If binary tree empty do nothing
       Otherwise
       2. Traverse the left sub-tree in in-order
       3. Visit the root (e.g. output the value of the data item at the root)
       4. Traverse the right sub-tree in in-order
  - post-order
    1. If tree empty do nothing
       Otherwise
       2. Traverse the left sub-tree in post-order
       3. Traverse the right sub-tree in post-order
       4. Visit the root (e.g. output the value of the data item at the root)
- Describing uses of tree-traversal algorithms
  - Pre-order: copying a tree given a pointer to its root
  - Pre-order: generating a prefix expression of an infix expression by traversing the infix expression tree in pre-order
  - In-order: if tree is a binary search tree then its node data can be output in ascending order by an in-order traversal of the tree.
  - Post-order: converting an infix expression to postfix or RPN (Reverse Polish Notation) form by traversing the infix expression tree in post-order
  - Post-order: emptying a tree given a pointer to its root

Single licence - Abingdon School

3 Fundamentals of algorithms

3.3 Reverse Polish

Learning objectives:

- *Be able to convert simple expressions in infix form to Reverse Polish notation (RPN) form and vice versa. Be aware of why and where it is used.*

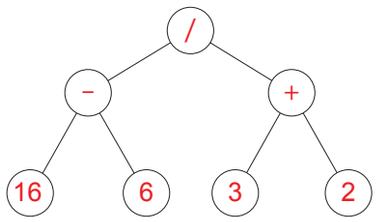


Figure 3.3.1.1 Expression tree

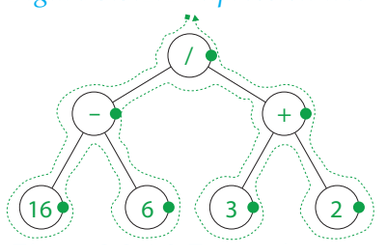


Figure 3.3.1.2 Expression tree and post-order traversal

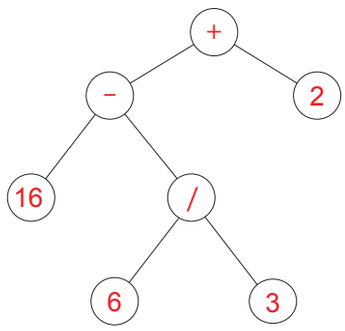


Figure 3.3.1.3 Expression tree

Key concept

Polish notation:
Polish notation is a parenthesis-free notation. The order of evaluation of an expression is made unambiguous by the notation so there is no requirement for parentheses (brackets).

Reverse Polish notation:
In the Reverse Polish notation (RPN) form of an expression, a binary (dyadic) operator follows its two operands, e.g. 5 3 +.

3.3.1 Reverse Polish - infix transformations

Infix to Reverse Polish

Post-order tree traversal may be used to convert an infix expression, e.g. 5 + 3, to its equivalent expression in postfix or Reverse Polish notation (RPN) form, 5 3 +.

In infix form, the (dyadic) operator is placed between the two operands. In postfix form the (dyadic) operator is placed after the two operands.

Step 1

First convert the infix expression to its equivalent expression tree (see *Chapter 2.5.1*).

Step 2

Traverse the binary tree in post-order.

For example, the infix expression (16 – 6) / (3 + 2) is represented by the expression tree shown in *Figure 3.3.1.1*.

Its postfix form, 16 6 – 3 2 + / is generated by performing a post-order traversal of this tree as indicated in *Figure 3.3.1.2*.

The postfix form has no need for brackets because the order of evaluation is implicit. Brackets are required however to express the order of evaluation in the infix form.

If the brackets in (16 – 6) / (3 + 2) are removed, the order of evaluation is changed. The infix expression is now 16 – 6 / 3 + 2 with brackets removed. Operator precedence dictates that 6 / 3 is evaluated first. The new expression tree is shown in *Figure 3.3.1.3*. It post-order traversal generates the postfix expression 16 6 3 / – 2 +.

Questions

1 Convert the following infix expressions to Reverse Polish form

(a) (5 + 3) * (6 + 4) (b) 5 ↑ 2 (c) 3 + 5 ↑ 2

(d) (5 + 3) / (6 + 4)

↑ is the exponentiation operator, i.e. $5 \uparrow 2 = 5^2$.

Single licence - Abingdon School

RPN to infix

An expression in Reverse Polish form can be evaluated with the assistance of a stack as demonstrated in *Chapter 3.2.1*. Converting from RPN to infix can be done in a similar manner. We work from left to right until an operator is encountered. The previous two operands and the operator are translated to infix form as shown in *Figure 3.3.1.4* which also shows the tree equivalent.

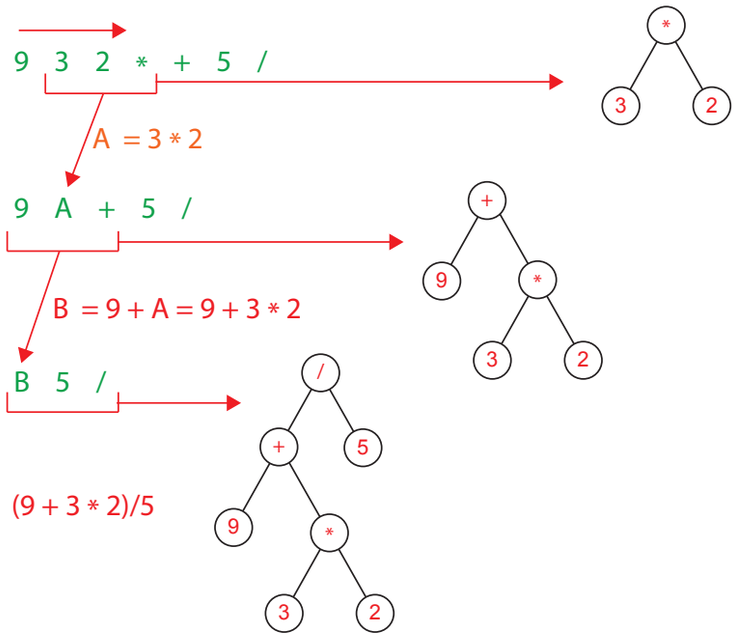


Figure 3.3.1.4 Converting RPN expression to an infix equivalent expression

The sub-expression 3 * 2 is replaced by A.
The next operator encounter is +. The previous two operands and the plus operator are translated to infix form 9 + A.

The sub-expression 9 + A which is 9 + 3 * 2 is replaced by B.

The next operator encounter is /.
The previous two operands and the / operator are translated to infix form B / 5.

Finally A and B are replaced to obtain the infix expression

(9 + 3 * 2) / 5.

An in-order traversal of the tree in *Figure 3.3.1.4* confirms this expression.

Information

Jan Łukasiewicz, a Polish mathematician was responsible for the introduction of parenthesis-free notation, now called Polish notation. It is said that it became known as Polish notation because people experienced difficulty pronouncing Jan's name. Reverse Polish notation is the post-fix form of Polish notation.

Questions

2 Convert the following Reverse Polish expressions to infix form

(a) 4 7 2 + * 3 − (b) 6 4 2 ↑ + 8 *

(c) 3 5 2 − * 6 ↑ 5 /

Why use Reverse Polish?

Reverse Polish notation is a parenthesis-free notation. The order of evaluation of an expression is made unambiguous by the notation so there is no requirement for parentheses (brackets). Expressions in Reverse Polish form can be evaluated easily using a stack.

Where is it used?

Virtual stack machine

Desktop and laptop computers are register-based machines in which machine instructions operate on numbers in registers. However, there is another type of machine called a stack machine whose instructions don't operate on numbers in registers but instead operate on numbers placed on a stack.
Even when a computer's instruction set is register-based it is possible to emulate a stack machine in software running on the hardware. This software is called a virtual stack machine.

Single licence - Abingdon School

Interpreters and compilers

Most of a stack machine's instruction set assume that operands will be from the stack, and results will be placed in the stack.

A stack is just what is needed to evaluate expressions in Reverse Polish form. Hence, Reverse Polish is used extensively in stack machines and their instruction sets are therefore designed to work with Reverse Polish notation.

There are two very important benefits of virtual stack machines

- Interpreters for virtual stack machines are easier to build than interpreters for register or memory-to-memory machines.
- Compilers for stack machines are simpler and quicker to build than compilers for other machines. Code generation is trivial. The simplicity of the compilers enable them to fit onto very small machines.

Applications of virtual stack machines

Java Virtual Machine (JVM)

These two benefits have been exploited to enable Java programs to be portable and run on any machine with the Java Virtual Machine, a stack machine, installed.

Java programs compile to bytecode.

Bytecode contains instructions from an instruction set designed for a stack machine and Reverse Polish Notation.

The Java virtual machine interprets bytecode to execute a bytecode program and therefore a Java program.

If speed of execution is important then the bytecode program can be compiled to native code.

The compiler that does this is relatively simple and quick to build.

Common Intermediate Language(CIL)

The Common Intermediate Language (CIL), used by the .NET Framework and Mono is a bytecode-style intermediate language designed to be interpreted by a virtual stack machine.

.NET high level languages such as C#, VB.NET, and F# compile to CIL so that they can execute in the .NET Framework or Mono environments.

Adobe Postscript

Adobe's Postscript is designed to be interpreted by a virtual stack machine. PostScript (PS) is a computer language for creating vector graphics (see Unit 2, page 153).

The language syntax uses Reverse Polish notation which makes the order of operations unambiguous.

Postscript processors (stack machines) are still used in high-end colour laser printers but Postscript as the basis for electronic document distribution has been replaced by PDF which, itself, is based on Postscript.

Compact programs

Stack machines have much smaller instructions than the other styles of machines and this fact is exploited in applications downloaded to browsers over slow Internet connections, and in ROMs for embedded applications. Also caches and instruction pre-fetchs are more effective as a result because there is less to fetch and cache.

Questions

3. Explain why the Reverse Polish form of arithmetic expressions is used.
4. Describe **two** examples where the benefits of Reverse Polish notation have been put to good use in the technology of the application and explain what benefits are exploited in each example.

Single licence - Abingdon School

In this chapter you have covered:

- Converting simple expressions in
  - infix form to Reverse Polish notation (RPN) form
  - Reverse Polish notation (RPN) to infix form
- Why Reverse Polish notation is used
- Where it is used.

Single licence - Abingdon School

3 Fundamentals of algorithms

3.4 Searching algorithms

Learning objectives:

- *Understand and explain how the linear search algorithm works*
- *Understand and explain how the binary search algorithm works*
- *Compare and contrast linear and binary search algorithms*

3.4.1 Linear search

Imagine a pile of animal name playing cards placed face down on a table in no particular order. The playing cards are labelled *ant*, *bee*, *cat*, *dog*, and *fox* and the pile is arranged as shown in *Figure 3.4.1*.

Searching for a particular card, say "cat", by turning over the cards in turn, starting from the card on top, is called a **linear search**. The red arrow in *Figure 3.4.1* indicates the cards that have to be examined before the card labelled "cat" is found.

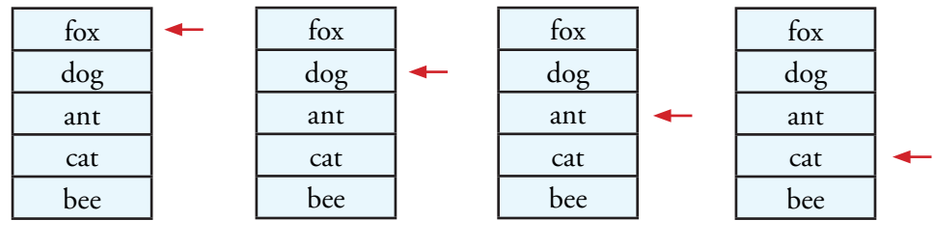


Figure 3.4.1 Linear search for the card labelled "cat"

Key point

Linear search:
Linear search scans each item or element in a collection of items, e.g. playing cards, in turn, starting from the beginning, until a match is found or the end of the collection is reached.

Linear search doesn't care whether the list is ordered or not.

Information

Linear search is often performed on lists of things, e.g. names.

Questions

1. A pack of cards is shuffled to ensure that the cards are in no particular order and then placed face down on a table. Starting from the top of the pack, one playing card is turned over at a time until the Ace of Spades is found.

   (a) If the task was repeated many times, shuffling the pack of 52 cards before each new search, on average how many cards would need to be turned over to find the Ace of Spades?

   (b) What is the maximum number of cards that need turning over to find a match?

2. Half of the pack is removed. Starting from the top of the pack, one playing card is turned over at a time until either the Ace of Spades is found or all the cards have been examined.
   What is the maximum number of cards that need turning over to find the Ace of Spades or to discover that the half-pack doesn't contain the Ace of Spades?

3. What is the maximum number of cards that have to be turned over in the pile of cards in *Figure 3.4.1* to discover that "rat" is not amongst them?

Single licence - Abingdon School

Key concept

Search length:
Search length = no of elements of the vector which are examined before a match is found

Key fact

Average search length:
Average search length $\approx \frac{\text{NoOfElementsInVector}}{2}$

Algorithm for linear search

Labelling the pile of animal name playing cards with the name, `Vector`, enables us to refer to the card on top as the card in location `Vector[1]`, the card below this card as the card in location in `Vector[2]`, and the jth card as the card in location in `Vector[j]`.

1	fox
2	dog
3	ant
4	cat
5	bee

Vector

Labelling the card that we are searching for, `ElementSought`, means that we can change this card to a different one and continue to refer to the card to search for by the label `ElementSought`. The number of elements, `NoOfElementsInVector`, is 5 in our example. The algorithm in *Figure 3.4.2* performs a linear search on `Vector` assigning to `Result` the position in `Vector` of the element if found otherwise assigning it the value 0.

Linear Search Algorithm

```
j ← 0
Found ← False
Repeat
  j ← j + 1
  If Vector[j] = ElementSought
    Then Found ← True
  EndIf
Until Found Or j = NoOfElementsInVector
If Found
  Then Result ← j
  Else Result ← 0
EndIf
```

Figure 3.4.2 Linear search algorithm

Task

1. Code the linear search algorithm in a programming language with which you are familiar. `Vector` can be implemented as a one-dimensional array of animal name strings or its equivalent. The animal name to search for should be entered at the keyboard and assigned to `ElementSought`. Your program should display the value assigned to `Result`.

Questions

4. What is meant by linear search?

Single licence - Abingdon School

3.4.2 Binary search

If the elements have been ordered then a much shorter average search length can be achieved as follows:

Assuming elements in a list are stored in ascending order as shown in *Figure 3.4.3*, a search for an element with a particular value, e.g. "dog", resembles the way a telephone directory might be searched.

The approximate middle of the list is located (location labelled 5 in *Figure 3.4.3*) and its value examined.

If this value is too high (e.g. alphabetically) then the approximate position of the middle element of the first half is calculated and its value examined.

If the value is too low then the approximate position of the middle element of the second half is calculated and its value examined.

This process continues until the desired element is found or the search interval becomes empty.

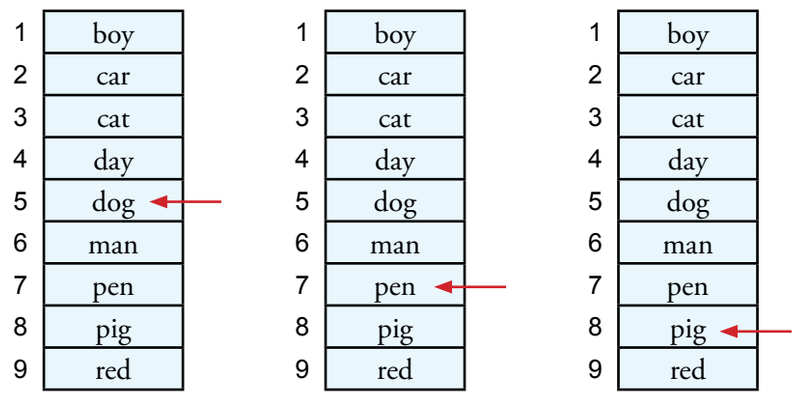


Figure 3.4.3 Performing a binary search for the word "pig" on an ordered list of words

Figures 3.4.3 and 3.4.4 show an example of binary search on an ordered list of three-letter words. The elements in the list have been numbered, 1, 2, 3, ... 7, 8, 9, for convenience. The list is searched for the word "pig" which is located at position 8 in the list.

The middle element, "dog" is selected first and compared with "pig". It doesn't match.

As "pig" is alphabetically greater than "dog", the second half of the list "boy" to "red" is chosen to search next. This second half runs from "man" to "red".

Its middle lies between the word "pen" and the word "pig". We have to choose one or the other so the word that comes first, "pen", is chosen. It doesn't match the word "pig". As

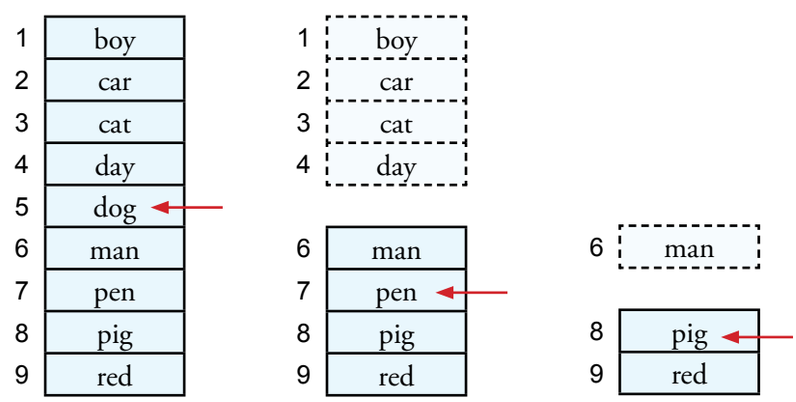


Figure 3.4.4 Performing a binary search for the word "pig" on an ordered list of words

Key point

Search interval:
The range over which the search is conducted, e.g. from list elements 1 to 9 inclusive.

Key principle

Binary search:
Searching for "pig" in the list in *Figure 3.4.3* with elements labelled 1 to 9, the position of the middle element is calculated as follows

middle position = (1 + 9) div 2
= 10 div 2 = 5

middle position = (6 + 9) div 2
= 15 div 2 = 7

middle position = (8 + 9) div 2
= 17 div 2 = 8

Generalising,

middle position =
(low + high) div 2

where *low* is position no of lowest item and *high*, position no of highest item in list, e.g.

low = 8, item = pig
high = 9, item = red

Single licence - Abingdon School

"pig" is alphabetically greater than "pen", the second half of the list "man" to "red" is chosen for the next search. This second half runs from "pig" to "red".

Its middle lies between the word "pig" and the word "red". We have to choose one or the other so the word that comes first, "pig", is chosen . It matches. So "pig" is present in the list and is located at position 8 in this list.

Key principle

Binary search:
Binary search uses a "divide and conquer" approach to search a list, chopping the list into smaller and smaller lists to search until item found or list cannot be divided anymore.

Question

5 What is meant by binary search?

Algorithm for binary search

Labelling the list to be binary searched as `Vector`, enables us to refer to the first element by its location `Vector[1]`, the next element by its location `Vector[2]`, and the jth element by its location `Vector[j]`. The range of the vector to be searched is stored in `Low` and `High`.
For example, `Low = 1`, `High = 9` means that the beginning of the range is location `Vector[1]` and the end of the range is `Vector[9]`.

Labelling the element that we are searching for, `ElementSought`, means that we can change the value to a different one and continue to refer to the element to search for by the label `ElementSought`. The algorithm in *Figure 3.4.5* performs a binary search on `Vector` assigning to `Result` the position in `Vector` of the element if found otherwise assigning it the value -1.

Binary Search Algorithm

```
Result ← -1
While (Low <= High) And (Result = -1)
  Middle ← (Low + High) Div 2  {Find middle of list}
  If ElementSought = Vector[Middle]
    Then Result ← Middle    {Found}
    Else
      If ElementSought < Vector[Middle]
        Then High  ← Middle - 1  {search first half}
        Else
          If ElementSought > Vector[Middle]
            Then Low  ← Middle + 1 {search second half}
          EndIf
      EndIf
  EndIf
EndWhile
```

Figure 3.4.5 Binary search algorithm

Single licence - Abingdon School

Task

2 Using the list shown in *Figure 3.4.6*, hand trace the binary search algorithm given above for the value "red". Complete a copy of the table shown below

Low	High	Middle
1	16	

1	ale	1	ale	1	ale	1	ale	1	ale	1	ale
2	ant	2	ant	2	ant	2	ant	2	ant	2	ant
3	ark	3	ark	3	ark	3	ark	3	ark	3	ark
4	bat	4	bat	4	bat	4	bat	4	bat	4	bat
5	boy	5	boy	5	boy	5	boy	5	boy	5	boy
6	car	6	car	6	car	6	car	6	car	6	car
7	cat	7	cat	7	cat	7	cat	7	cat	7	cat
8	day	8	day	8	day	8	day	8	day	8	day
9	dog	9	dog	9	dog	9	dog	9	dog	9	dog
10	fox	10	fox	10	fox	10	fox	10	fox	10	fox
11	jar	11	jar	11	jar	11	jar	11	jar	11	jar
12	jug	12	jug	12	jug	12	jug	12	jug	12	jug
13	man	13	man	13	man	13	man	13	man	13	man
14	pen	14	pen	14	pen	14	pen	14	pen	14	pen
15	pig	15	pig	15	pig	15	pig	15	pig	15	pig
16	red	16	red	16	red	16	red	16	red	16	red

Figure 3.4.6 Performing a binary search for the word "red" on an ordered list of words

Tasks

3 How many elements of the list in *Figure 3.4.6* have to be examined when binary searching for the element "red"?

4 How many elements have to be examined when binary searching for

(a) the element "day" in a list constructed from elements 1 to 8 of *Figure 3.4.6*?

(b) the element "bat" in a list constructed from elements 1 to 4 of *Figure 3.4.6*?

(c) the element "ant" in a list constructed from elements 1 to 2 of *Figure 3.4.6*?

Single licence - Abingdon School

Comparing linear and binary search algorithms

Table 3.4.1 summarises the outcomes of completing tasks 3 and 4. From *Table 3.4.1* we conclude that for binary search the maximum search length **increases linearly when the number of elements or items in a list doubles**. For example, if the number of items in the list is 8 (2^3), the maximum search length is 3 + 1, i.e. 4 items have to be examined at most to find a match or conclude that the sought item is not in the list.

No of items in list	No of items in list as a power of 2	Maximum search length
1	2^0	1
2	2^1	2
4	2^2	3
8	2^3	4
16	2^4	5

Table 3.4.1 Relationship between maximum search length and no of items in list for binary search

If we have, say, 16777216 (2^{24}) in a list, the maximum search length is 24 + 1, i.e. 25 items have to be examined at most to find a match or conclude that the sought item is not in the list.

If we contrast this with linear search, then searching a list of 8 items requires 8 items to be examined if the sought item is the last item, i.e. maximum search length for this linear search = 8.

Similarly, searching a list of 16777216 items requires 16777216 items to be examined if the sought item is the last item, i.e. maximum search length for this linear search = 16777216.

No of items in list	Maximum search length binary search	Maximum search length linear search
1	1	1
2	2	2
4	3	4
8	4	8
16	5	16
32768	16	32768
65536	17	65536
16777216	25	16777216

Table 3.4.2 Comparing maximum search length for binary and linear searches

Table 3.4.2 compares binary search with linear search for different lengths of list. This table shows clearly that **binary search is more efficient than linear search, time-wise**. Each element or item of a list that has to be examined costs time. If, for argument's sake, it takes one microsecond to examine an item, then for a list of 16777216 items, binary search will take a maximum of 25 microseconds whilst linear search will take 16777216 microseconds or approximately 17 seconds.

We may draw a similar conclusion for the average search length.

Binary search can only be performed on ordered lists whereas **linear search can be performed on both ordered and unordered lists**. Sorting a list into order will take time but once ordered binary search will perform searches on the list faster than linear search will on the unordered list with the speed advantage increasing with the size of the list.

Key fact

Binary search versus linear search:
Binary search is more efficient time-wise than linear search.

Key fact

Binary search versus linear search:
Binary search can only be performed on ordered lists, linear search can be performed on both ordered and unordered lists.

Single licence - Abingdon School

Questions

6 State **two** requirements that a list must satisfy for an item to be found using binary search.

7 Explain why binary search is more efficient than linear search, time -wise.

8 State whether it is possible to search an unordered list using
(a) binary search
(b) linear search

9 A stall is planned for a school fête which is expected to make some money for the school by challenging visitors to guess a secret number between 1 to 1,000,000 within 20 guesses.
After each guess, one of the following answers to the guess is supplied by the stall holder:
Too Low, Too High, or You Win.
Each visitor is charged £1 to have a go and the prize for guessing correctly is £10.
Do you think that this stall will make a profit? Explain the reasoning you used to reach your conclusion.

3.4.3 Binary search tree

Figure 3.4.7 shows a binary search tree. The nodes of the tree are ordered so that all the integers in the nodes in the left sub-tree are smaller than the integer in the root and all the integers in the nodes in the right sub-tree are greater than the root. *Figure 3.4.8* and *Figure 3.4.9* show how this binary search tree may be stored in three one-dimensional arrays, Left, Key and Right. The null pointer is -1. The array index starts at 0.

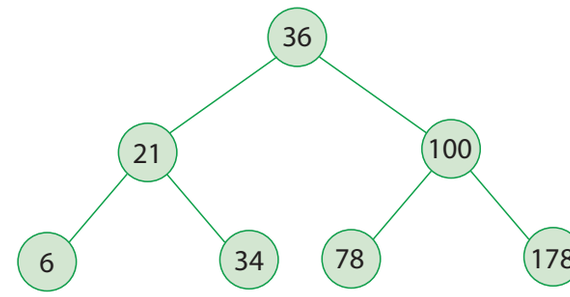


Figure 3.4.7 Binary search tree

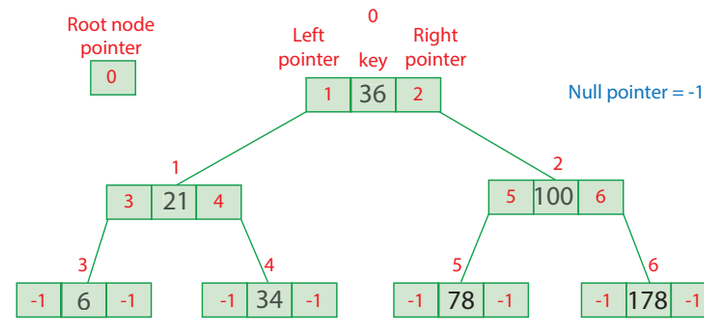


Figure 3.4.8 Binary search tree with node structure consisting of a left pointer field, right pointer field and key field

Root node pointer: 0

Array index	Left	Key	Right
0	1	36	2
1	3	21	4
2	5	100	6
3	-1	6	-1
4	-1	34	-1
5	-1	78	-1
6	-1	178	-1

Figure 3.4.9 Binary search tree represented by three one-dimensional arrays, Left, Key and Right

The algorithm in *Figure 3.4.10* uses the symbolic name Nil for the null pointer but it could easily have used -1.

The algorithm searches for a match with k, e.g. 21 in the array Key. NodePtr is initially set to 0, the root of the whole tree.

If a match is not found with the root then the search is switched to the left or right sub-tree.
The algorithm is now applied to the selected sub-tree.

Single licence - Abingdon School

Binary Search Tree Algorithm

```
{Searches for k in Key array given pointer to root node, NodePtr}

If (NodePtr = Nil) Or (k = Key[NodePtr])
  Then Return NodePtr {returns Nil or position of match}
EndIf
While (NodePtr <> Nil) And (k <> Key[NodePtr])
  If k < key[NodePtr]
    Then NodePtr ← Left[NodePtr] {search left sub-tree}
    Else NodePtr ← Right[NodePtr] {search right sub-tree}
  EndIf
EndWhile
Return NodePtr {returns Nil or position of match}
```

Figure 3.4.10 Binary search tree algorithm

Tasks

5 Hand trace the binary search tree algorithm given above by searching for the value 34 in the binary search tree shown in *Figure 3.4.7* and represented by three one-dimensional arrays shown in *Figure 3.4.9*. Complete a copy of the table shown below

k	NodePtr	Key[NodePtr]
34	0	36

6 Hand trace given the binary search tree algorithm given above by searching for the value 85 in the binary search tree represented by three one-dimensional arrays shown in *Figure 3.4.9*. Complete a copy of the table shown below

k	NodePtr	Key[NodePtr]
34	0	36

Single licence - Abingdon School

Analysing the complexity of algorithms

Upper and lower bounds

It is useful to know the average time an algorithm may take to execute (measured in terms of number of steps) but it is often more difficult to calculate than the alternatives which seek to bound the execution time by either an

- Upper bound: the algorithm will take no more than 4 time units

Or

- Lower bound: the algorithm will take at least 3 units of time.

These bounds correspond to the **worst-case** and the **best-case**. The worst-case provides an upper bound on possible execution time, and the best case, a lower bound. The expected execution time must fall between these two bounds.

This is useful because it tells us that a program implementing the given algorithm will run for at least a minimum time, but will definitely complete by some definite, but potentially long, time.

Information

Average-case analysis can be difficult as well as misleading. Average-case analysis is based on the performance of the algorithm averaged over randomly chosen instances. However, it is very hard to express the full range of possible input instances that arise in practice and often these are not being produced from a random distribution. The same algorithm can perform well on one class of random inputs and very poorly on another.

Order of growth

Execution time of an algorithm can also vary with the size of the input.

It makes sense therefore when comparing algorithms to consider how execution time $f(n)$ varies with the size of the input n. This is done by focusing on the rate of growth of $f(n)$ as a function of the input size n.

We say that an algorithm runs in, say, $O(n)$ where n is the size of the input, when the time taken as measured by $f(n)$ grows at a rate proportional to n as shown in *Figure 3.4.11*. $O(n)$ is called Big-O notation (see *Chapter 4.4.3*).

With Big-O, we don't care whether it takes for example, n, $2n$, $6n$, or $n/6$ time units, we say that the algorithm still runs in $O(n)$ because we are only interested in how the execution time grows with input.

Also, $O(n)$ applies when n is large ($n \geq n_0$ where is a constant). The same is true of any other Big-O classification, e.g. $O(n^3)$.

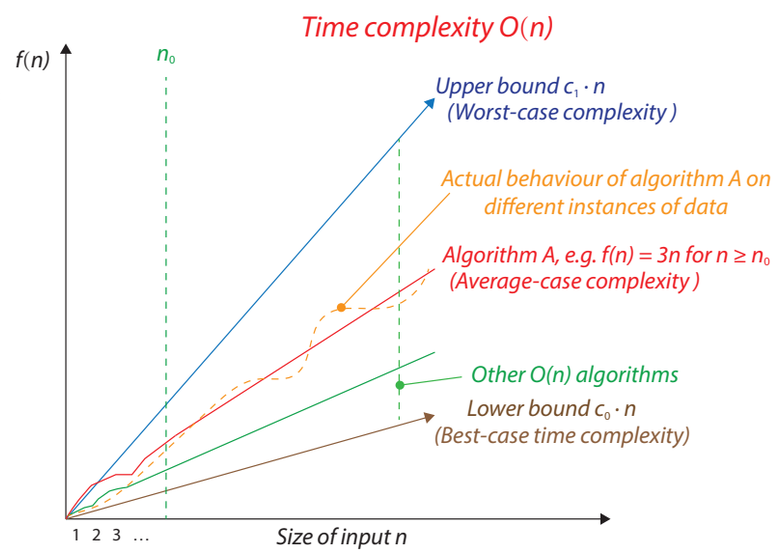


Figure 3.4.11 Algorithm A that grows at O(n) between upper and lower bounds for $n \geq n_0$

Figure 3.4.11 shows upper and lower bounds for $O(n)$ and an algorithm A that falls within these bounds because its execution time $f(n) = 3 \cdot n$ for $n \geq n_0$ when its behaviour is averaged over differently instances of input data.

The upper bound growth rate is $c_1 \cdot n$ where c_1 is a constant and $c_1 > 3$.

The lower bound growth rate is $c_0 \cdot n$ where c_0 is a constant and $c_0 < 3$.

Single licence - Abingdon School

Figure 3.4.12 shows upper and lower bounds for $O(\log n)$ and an algorithm B that falls within these bounds because its execution time $f(n)$ is given by

$$f(n) = 3 \cdot \log n \text{ for } n \geq n_0.$$

The upper bound growth rate is $c_1 \cdot \log n$ where c_1 is a constant and $c_1 > 3$.
The lower bound growth rate is $c_0 \cdot \log n$ where c_0 is a constant and $c_0 < 3$.

Note that $O(\log n)$ grows much more slowly than $O(n)$.

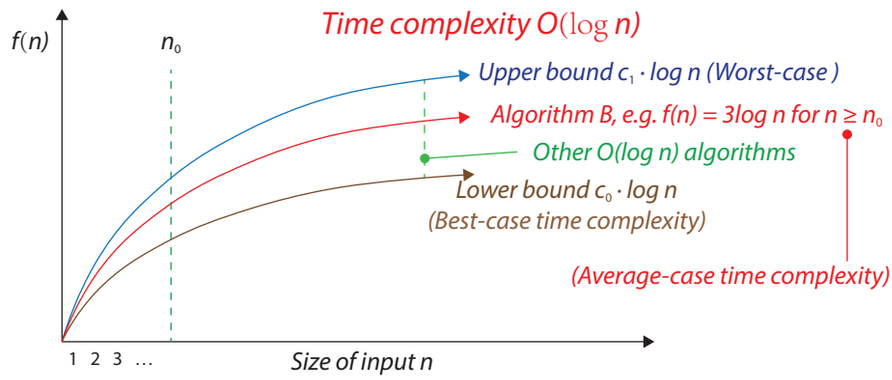


Figure 3.4.12 Algorithm B that grows at O(log n) between upper and lower bounds for $n \geq n_0$

Information

What is $\log_{10} 100$ in words is "log or logarithms to the base 10 of 100" What answer does $\log_{10} 100$ give? The number of 10s we multiply together to get 100 i.e. 2 because 10 x 10 = 100.
Therefore $\log_{10} 100 = 2$.
Logs are the opposite of raising something to a power, e.g. 100^2.

$100 = 10^2$	$\leftrightarrow$	$\log_{10} 100 = 2$
$1000 = 10^3$	$\leftrightarrow$	$\log_{10} 1000 = 3$
$4 = 2^2$	$\leftrightarrow$	$\log_2 4 = 2$
$8 = 2^3$	$\leftrightarrow$	$\log_2 8 = 3$
$16 = 2^4$	$\leftrightarrow$	$\log_2 16 = 4$

$\log_2 8$ is log to the base 2 of 8. It is the number of 2s we multiply together to get 8.
See Khan Academy videos on logarithms - www.khanacademy.org.

The worst-case complexity proves to be the most useful of the three measures in practice.
For example, consider a safety-critical system such as a program that controls part of an aircraft. The designer of the control needs to know, for example, "how long it will take from the pilot pressing the button that activates the control before the aircraft responds"? If this time is too long another algorithm/mechanism will be needed that takes less time. Note that it doesn't matter if the system works fast enough on average or even nearly every time. Its performance has to work fast enough on every occasion.

Time complexity of linear search

The worst-case time complexity occurs when the whole list has to be searched. Therefore, the search length cannot be any bigger than n, where n is the number of items in list. The maximum search length thus increases as n increases giving a worst-case time complexity of $O(n)$.

For example, consider the list of integers

36, 12, 34, 178, 2, 21, 6, 78, 100

To discover if the integer value 200 is in the list requires the whole list of nine items to be accessed. Thus the maximum search length is nine in this case.

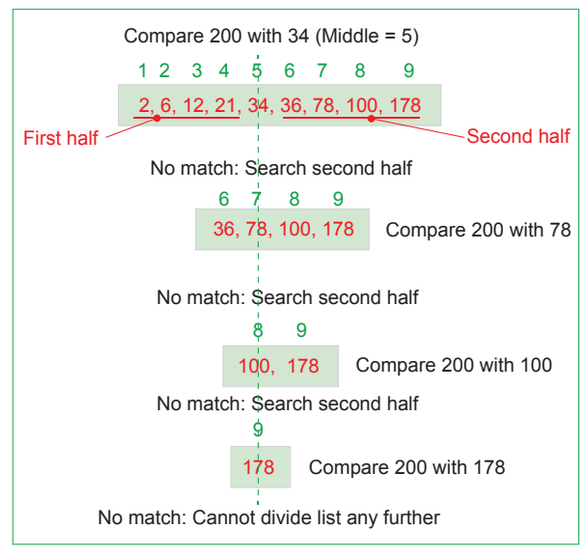


Figure 3.4.13 Searching for 200 using binary search

Single licence - Abingdon School

Time complexity of binary search

The binary search algorithm requires the search list of n items to be ordered in ascending order.

Figure 3.4.13 shows the same set of integers used in the linear search example but this time they form an ordered search list.

A binary search is carried out on this ordered list for integer 200. Binary search is a divide and conquer algorithm. It works by progressively shortening the list by removing one half until a match is found or until the list cannot be shortened any further and just one list element remains. How many times can a list of length n be halved until one list element remains?

We need to solve

$1 = {}^{n}/_{2^x}$ where x is the number of times the list is divided by 2

$2^x = n$

$\log_2 2^x = \log_2 n$

$x \log_2 2 = \log_2 n$

$x = \log_2 n$ because $\log_2 2 = 1$

Length of list, n	Maximum no of divisions	$\lfloor \log_2 n \rfloor$
9	3	3
16	4	4
32	5	5
64	6	6
32	7	7
65536	16	16
4194304	22	22

Table 3.4.3 Shows how maximum number of comparisons varies with length of list

Information

The base doesn't matter when comparing growth rates so the base subscript is dropped, e.g. $\log_{10}$ and $\log_2$ become log in O(log n).

In the example in *Figure 3.4.13* the total number of divisions is three and the number of comparisons is four. Therefore, worst-case time complexity occurs when four comparisons are made on this list. The best-case time complexity is when a match is made on the middle item of the whole list. This takes one comparison.

We say that the search length for the worst-case time complexity is four in this example.
Generalising,

Maximum number of comparisons = Maximum search length = Maximum number of times list divided in half + 1

Table 3.4.3 shows some list lengths n, the corresponding maximum number of divisions of list, and $\lfloor \log_2 n \rfloor$. The combination of symbols $\lfloor\ \rfloor$ surrounding $\log_2 n$ means round down after calculating $\log_2 n$.
For example, $\log_2 9 = 3.1699250014$, and $\lfloor \log_2 9 \rfloor = 3$.

(A log to the base 2 calculator can be found at http://www.rapidtables.com/calc/math/Log_Calculator.htm)

Therefore,

Maximum search length = $\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 n \rceil$

The combination of symbols $\lceil\ \rceil$ surrounding $\log_2 n$ means round up after calculating $\log_2 n$.

n	n as power of 2	$\log_2 n$
2	2^1	1
4	2^2	2
8	2^3	3
16	2^4	4
32	2^5	5
64	2^6	6
128	2^7	7

Table 3.4.4 Shows how $\log_2 n$ for various n which are multiples of 2

We may conclude from this analysis that binary search can be classified as $O(\log n)$.

Big-O expresses the growth rate for large n which means that we can ignore constants and any rounding down.

Table 3.4.4 shows how $\log_2 n$ grows with n for values of n which are powers of 2.

Figure 3.4.12 shows an algorithm B that grows at $O(\log n)$ between upper and lower bounds for $n \geq n_0$. However, the lower bound for binary search is not $O(\log n)$ but a constant, one comparison is sufficient whatever the size of the list. The best-case time complexity occurs when the match occurs on the middle item of the whole list.

Single licence - Abingdon School

Binary search tree time complexity

Binary searching on an array or list in memory is efficient because the worst-case time complexity is $O(\log n)$. However, using ordered lists/arrays becomes drastically less effective when the data set changes frequently because frequent re-ordering/insertion in the correct order is required.

In order to maintain acceptable search performance, an alternative approach is necessary which relies upon a search tree to store data sets which change frequently.

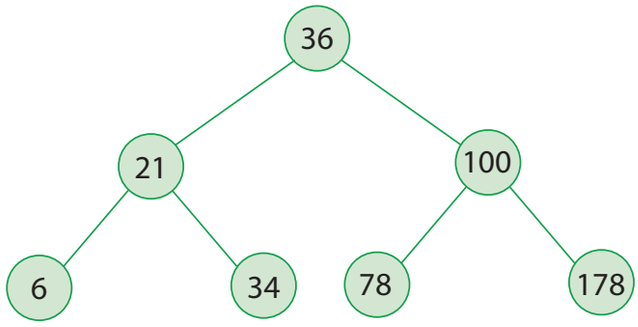


Figure 3.4.14 Balanced binary search tree

The most common type of search tree is the binary search tree (BST), which is composed of nodes as shown in *Figure 3.4.14*.

Each node contains a single value from the data set and stores references to potentially two child nodes, left and right.

The BST in *Figure 3.4.14* is balanced and complete. Balanced means that the path from root to leaf node is the same across the tree. Complete means that every leaf node is present.

No of nodes n	Maximum depth of balanced tree	$\lceil \log_2 n \rceil$
3	2	2
7	3	3
9	4	4
15	4	4
31	5	5
63	6	6
127	7	7

Table 3.4.5 Shows how maximum depth of balanced tree varies with number of nodes

Binary search tree is a divide and conquer algorithm. It works by progressively reducing the size of the tree by removing one half, until a match is found or until the tree cannot be reduced any further.

If we are able to maintain a balanced or nearly balanced search tree then

Maximum number of comparisons = Maximum search length = Maximum depth of balanced tree + 1 = $\lceil \log_2 n \rceil$

The combination of symbols ⌈ ⌉ surrounding $\log_2 n$ means round up after calculating $\log_2 n$.

Table 3.4.5 shows how the maximum depth of a balanced tree (and one unbalanced tree of 9 nodes) varies with number of nodes and how this is mirrored by $\lceil \log_2 n \rceil$. The maximum depth of a binary tree is defined as the number of edges along the path from the root node to the deepest leaf node. *Figure 3.4.15* shows a search for an integer from the range 0..20 for a balanced binary tree of maximum depth 3.

Figure 3.4.16 shows a search for an integer from the range 0..5 for an unbalanced binary tree of maximum depth 3.

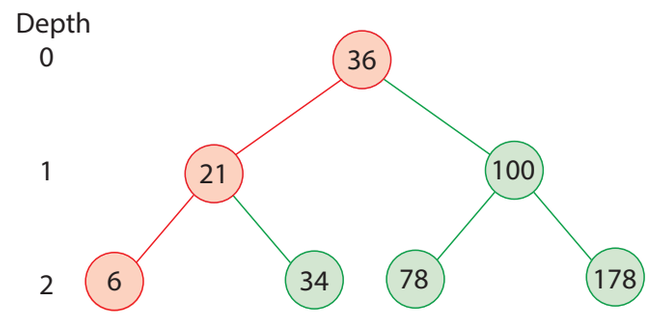


Figure 3.4.15 Binary search tree with seven nodes and maximum depth 3

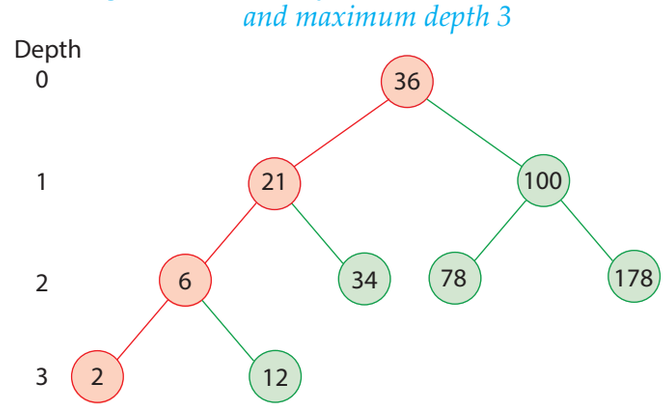


Figure 3.4.16 Binary search tree with nine nodes and maximum depth 3

Single licence - Abingdon School

From this simple analysis, we may conclude that the binary search tree algorithm worst-case time complexity can be classified as O(log n).

Big-O expresses the growth rate for large *n* which means that we can ignore constants and any rounding up.

Figure 3.4.16 shows a case when the binary search tree cannot be balanced because the number of nodes makes it impossible. However, even when the number of nodes allows the tree to be balanced, there is always the possibility of arranging the nodes in the tree in an unbalanced way.

Figure 3.4.17 shows an example of the possible binary search trees, ordered alphabetically on the letters A, B, C in three nodes, for which only one is balanced.

In all the cases where the depth is 3, the maximum search path is the same as the number of nodes, i.e. 3.
If the number of nodes is *n*, the binary search tree may have depth *n*. The maximum search length is *n* and the worst-case time complexity is $O(n)$.

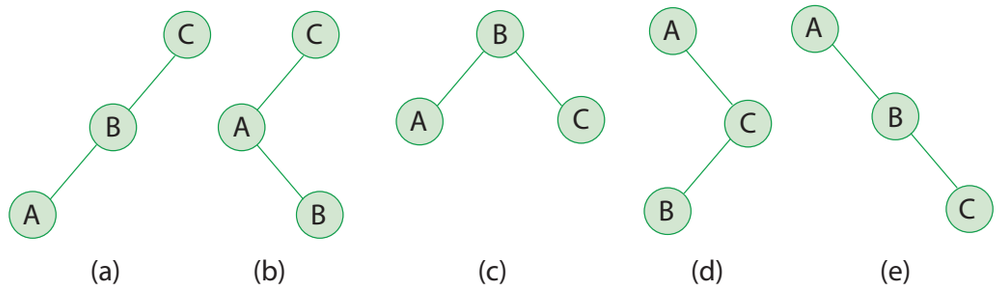


Figure 3.4.17 Possible three-node BSTs

If the data set is known in advance, it would be prudent to create a balanced binary search tree or one that is close to being balanced so that the worst-case complexity is $O(\log n)$.

If the data set is dynamic, i.e. subject to change by insertion and deletion then it becomes difficult, if not impossible, to predict the pattern of data and frequencies. For such data, a technique named after its inventors Adelson, Velski & Landis, is employed which keeps the tree balanced or at least close to balance. These height/depth balancing binary search trees are called AVL tree. The AVL tree technique checks the height of the left and the right sub-trees and ensures that the difference is not more than one level, e.g. as in *Figure 3.4.16*.
The worst-case time complexity is then $O(\log n)$ which is an improvement over $O(n)$ for a severely unbalanced tree such as shown in *Figure 3.4.17(a)* (which is more like a linear linked list).

Table 3.4.6 shows a comparison of best-case, average and worst-case time complexities for two kinds of binary search tree, binary search and linear search.

Algorithm	Best	Average	Worst	Comment
Binary Search Tree	O(1)	O(log n)	O(n)	Can be unbalanced
AVL Binary Search Tree	O(1)	O(log n)	O(log n)	Balanced
Linear search	O(1)	O(n)	O(n)	Brute force
Binary Search	O(1)	O(log n)	O(log n)	Divide & conquer

Table 3.4.6 Comparison of best-case, average and worst-case time complexities

Single licence - Abingdon School

Questions

10 Worst-case time complexity is one measure that can be used to analyse the performance of search algorithms.
(a) State **two** other measures of time complexity that can be used.
(b) Explain why worst-case complexity proves to be the most useful of the three measures in practice.

11 Explain why the worst-case time complexity for linear search is O(n).

12 Explain why the worst-case time complexity for binary search is O(log n)

13 Explain why the worst-case time complexity for a binary search tree is O(log n) if the tree is balanced.

In this chapter you have covered:

- linear search algorithm scans a list from the beginning until a match is found or the end of the list is reached.
- binary search algorithm uses a "divide and conquer" approach to searching a list by chopping the list into smaller and smaller lists to search until item found or list cannot be divided anymore.
- binary search is more efficient than linear search, timewise, because it examines less elements of a list
- binary search can only be performed on ordered lists
- linear search can be performed on both ordered and unordered lists

Single licence - Abingdon School

3 Fundamentals of algorithms

3.5 Sorting algorithms

Learning objectives:

- *Know and be able to trace and analyse the time complexity of the bubble sort algorithm*

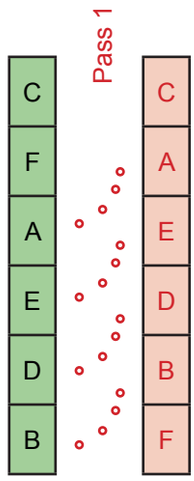


Figure 3.5.1.1 First pass

C	C	C	C	C	C
F	F	A	A	A	A
A	A	F	E	E	E
E	E	E	F	D	D
D	D	D	D	F	B
B	B	B	B	B	F

Figure 3.5.1.2 First pass showing comparisons and exchanges

C	A	A	A	A
A	C	C	C	C
E	E	E	D	D
D	D	D	E	B
B	B	B	B	E
F	F	F	F	F

Figure 3.5.1.3 Second pass

3.5.1 Bubble sort

In computer science sorting means arranging items into ascending or descending order, e.g. the following unsorted list of letters of the alphabet is sorted into ascending order when letter A is first, letter B second, and so on.

Unsorted list: C F A E D B

Sorted list (ascending): A B C D E F

The following unsorted list of letters of the alphabet are sorted into descending order when the letter F is first, the letter E second, and so on.

Unsorted list: C F A E D B

Sorted list (ascending): F E D C B A

Bubble sort is a simple sorting algorithm but not a very efficient one. It belongs to a family of sorting algorithms called "exchange" or "transposition" sorts. In an exchange sort, pairs of items that are out of order are interchanged until no more out of order pairs exist. *Figure 3.5.1.1* shows the "lighter" items bubbling up the list whilst the "heaviest" sinks to the bottom.

The pairs that are compared are as follows on the first pass through the list:

C & F - no exchange; F & A - exchange; F & E - exchange;

F & D - exchange; B & F = exchange.

The first pass is shown in *Figure 3.5.1.2*. The comparisons are pink squares ( ), the exchanges are red letters and the no-exchanges are green letters.

As the list is not sorted by the end of the first pass, another pass is needed but we don't need to compare the last two items because the "heaviest", F, has sunk to the bottom.

The list is now C A E D B F.

The pairs that are compared are as follows on the second pass through the list:

C & A - exchange; C & E - no exchange; E & D - exchange;

E & B - exchange.

The second pass is shown in *Figure 3.5.1.3*. The comparisons are pink squares, the exchanges are red letters and the no-exchanges are green letters.

As the list is not sorted by the end of the second pass, another pass is needed but we don't need to compare the last three items because the "heaviest", F, and the next "heaviest", E, have sunk to the bottom.

Single licence - Abingdon School

A	A	A	A
C	C	C	C
D	D	D	B
B	B	B	D
E	E	E	E
F	F	F	F

Figure 3.5.1.4 Third pass

The list is now A C D B E F.

The pairs that are compared are as follows on the third pass through the list:

A & C - no exchange; C & D - no exchange; D & B - exchange.

The third pass is shown in *Figure 3.5.1.4*. The comparisons are pink squares, the exchanges are red letters and the no-exchanges are green letters.

As the list is not sorted by the end of the third pass, another pass is needed but we don't need to compare the last four items because the "heaviest", F, the next "heaviest", E, and the next D have sunk to the bottom.

The list is now A C B D E F.

The pairs that are compared are as follows on the fourth pass through the list:

A & C - no exchange; C & B - exchange.

The fourth pass is shown in *Figure 3.5.1.5*. The comparisons are pink squares, the exchanges are red letters and the no-exchanges are green letters.

A	A	A
C	C	B
B	B	C
D	D	D
E	E	E
F	F	F

Figure 3.5.1.5 Fourth pass

The list is now A B C D E F. It is sorted in ascending alphabetical order.

Unfortunately, the bubble sort algorithm makes one more pass.

The pairs that are compared are as follows on the fifth pass through the list:

A & B - no exchange.

The fifth pass is shown in *Figure 3.5.1.6*. The comparisons are pink squares.

A	A
B	B
C	C
D	D
E	E
F	F

Figure 3.5.1.6 Fifth pass

The number of passes is one less than the number of items to be sorted.

Number of passes = NoOfItems - 1

On the first pass, the number of comparisons is one less than the number of items to be sorted, i.e. 5. On the next pass, 4, then on the next 3, the next 2 and the last 1 as shown in *Figure 3.5.1.8*.

PassNo	NoOfItems-(PassNo+1)
1	6 - (1 + 1) = 4
2	6 - (2 + 1) = 3
3	6 - (3 + 1) = 2
4	6 - (4 + 1) = 1
5	6 - (5 + 1) = 0

Figure 3.5.1.8

To compare a pair of adjacent items at position `j` and `j+1` in an array of items `ArrayOfItems` (*Figure 3.5.1.7*)

0	C
1	F
2	A
3	E
4	D
5	B

Figure 3.5.1.7 Array, ArrayOfItems

we can use

```
If ArrayOfItems[j] > ArrayOfItems[j+1]
```

The array's index runs from `0` to `NoOfItems - 1`

To swap or exchange a pair of items at position `j` and `j+1` we can use the following

```
Temp ← ArrayOfItems[j]
ArrayOfItems[j] ← ArrayOfItems[j+1]
ArrayOfItems[j + 1] ← Temp
```


Single licence - Abingdon School

To compare 5, then 4, 3, 2 and 1 pairs we can use a For loop as follows

```
For j ← 0 To (NoOfItems - (PassNo + 1))
```

As `PassNo` is 1, then 2, 3, 4 and 5, `j` is `0 To 4`, then `0 To 3`, `0 To 2`, `0 To 1` and finally `0 To 0` as shown in *Figure 3.5.1.9*.

Figure 3.5.1.10 shows this sequence for each pass.

Figure 3.5.1.11 shows how j varies on each pass.

PassNo	For j ← 0 To NoOfItems-(PassNo+1)
1	For j ← 0 To 4
2	For j ← 0 To 3
3	For j ← 0 To 2
4	For j ← 0 To 1
5	For j ← 0 To 0

Figure 3.5.1.9

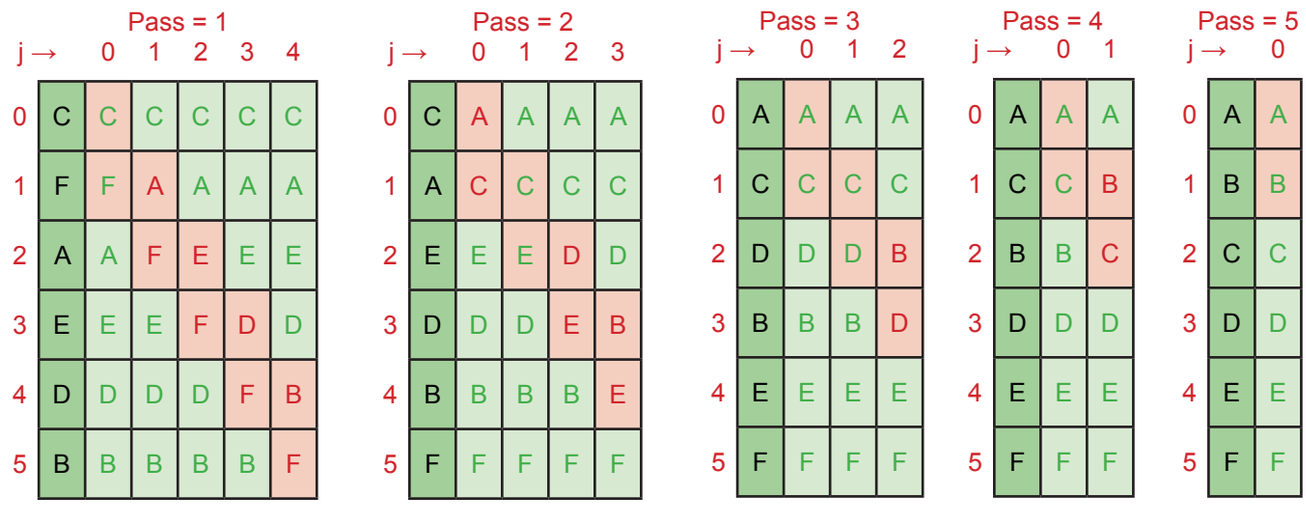


Figure 3.5.1.10 The range of values of loop counter j on each pass

The bubble sort algorithm is given in *Figure 3.5.1.12*.

It consists of an outer loop

```
For PassNo ← 1 To (NoOfItems - 1) ...  EndFor
```

which controls an inner loop which is executed once for each value of `PassNo`

```
For j ← 0 To (NoOfItems - (PassNo + 1))
```

```
For PassNo ← 1 To (NoOfItems - 1)
  For j ← 0 To (NoOfItems - (PassNo + 1))
    If ArrayOfItems[j] > ArrayOfItems[j + 1]
      Then
        Temp ← ArrayOfItems[j]
        ArrayOfItems[j] ← ArrayOfItems[j + 1]
        ArrayOfItems[j + 1] ← Temp
    EndIf
  EndFor
EndFor
```

Figure 3.5.1.12 Bubble sort algorithm

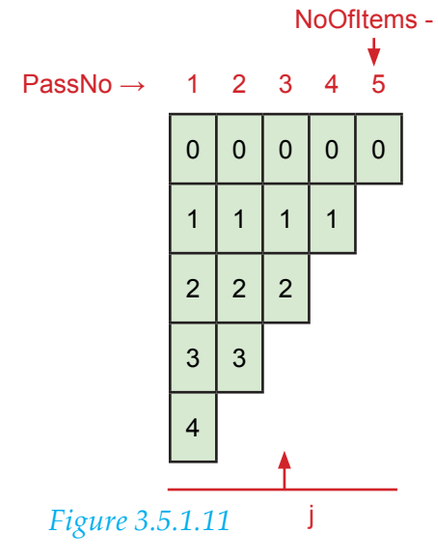


Figure 3.5.1.11

Information

Bubble sort is very inefficient. For example, if bubble sort is required to sort an ordered list of items into ascending order and the list is already in ascending order, it fails to notice this. Instead it ploughs on merrily through the outer and inner loops making unnecessary comparisons. Adding a flag `NoExchanges` can be used to stop the iterations.

NoExchanges is set to True at the beginning of each pass and to False inside the inner loop if an exchange occurs. If NoExchanges is still True at the end of a pass, the list is ordered and the loop can be exited prematurely.

Single licence - Abingdon School

Questions

1 The following list of letters of the alphabet is to be sorted by bubble sort into ascending order with A first followed by B, and so on,

F E D C B A

Using the algorithm specified in *Figure 3.5.1.12*, hand trace sorting this list into ascending order by completing a copy of the trace tables shown in *Figure 3.5.1.13*. The first column in each is the unsorted or partially sorted list at the beginning of each pass. The first column of the first pass table is already completed. The list of numbers to the left of each table is the index of the array ArrayOfItems.

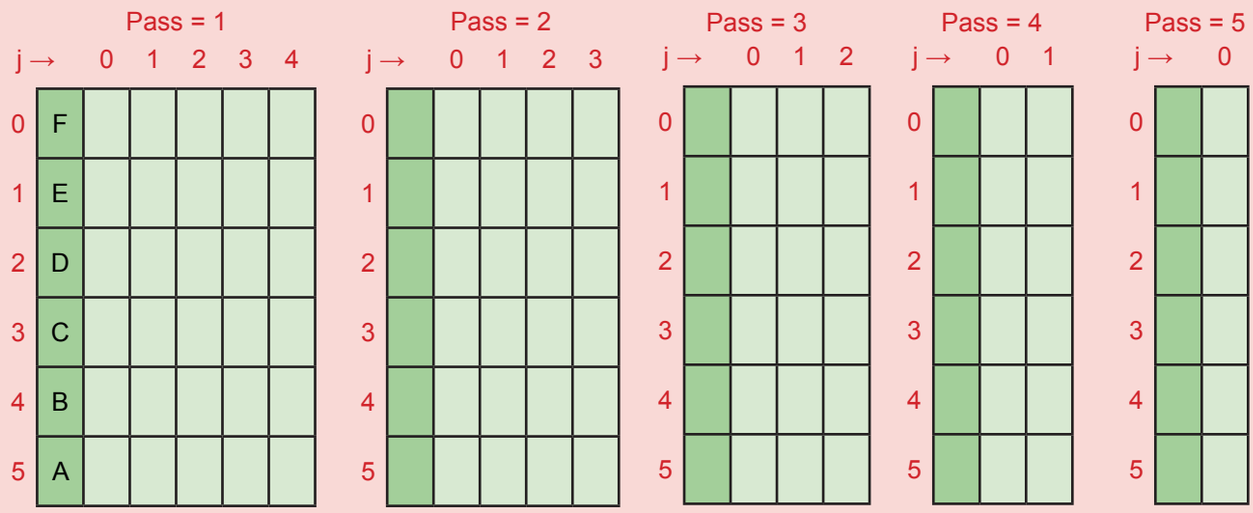


Figure 3.5.1.13 State of array, ArrayOfItems, during trace of bubble sort algorithm

2 What change(s) should be made to the bubble sort algorithm specified in *Figure 3.5.1.12* to enable it to sort an unsorted list of items into descending order?

Programming task

1 In a programming language with which you are familiar, code the bubble sort algorithm to sort a given list of integers into ascending order.
Apply it to sorting the following list of integers 23 17 45 6 78 21 into ascending order.

2 Modify your program so that it sorts a given list of integers into descending order.

3 Modify your program so that it counts and totals each of the following:

- *the number of comparisons*
- *the number of exchanges*
- *the number of passes (outer loop).*

The result of each should then be written to the screen once sorting is complete.
Include in your program a subroutine which generates and populates array, `ArrayOfItems` with integers in ascending order. This subroutine should take a parameter, *n* which is the size of the list of integers to be generated. Your program is then to sort the integers in array `ArrayOfItems` into descending order.
Run your program so that it calls this subroutine for $n = 6$, $n = 100$, $n = 1000$ and outputs each of the totals

Single licence - Abingdon School

Time complexity of bubble sort algorithm

How long would one have to wait for the bubble sort algorithm given in *Figure 3.5.1.12* to terminate?

A precise mathematical derivation of the running time is

$7.5n^2 + 0.5n + 1$ where n is the number of items.

The asymptotic running time is then $O(n^2)$ i.e. for values of n larger enough that the term $7.5n^2$ dominates.

An imprecise derivation by example:

Consider sorting the following unsorted list of 6 letters of the alphabet F E D C B A into ascending order.

The number of comparisons + the number of exchanges on the first pass is 10, on the second 8, then 6, 4, and 2 on the last.

In total, the number of comparisons + exchanges is 30.

Thus 30 is the running cost for 6 items.

The average (number of comparisons + exchanges) is $6 = \frac{(10 + 8 + 6 + 4 + 2)}{5}$

Therefore, on average, each pass costs 6 running time units.

The number of passes is 5.

Therefore,

Running time = Number of passes x Average time per pass = 5 x 6 = 30.

For 7 items, there are 6 passes and 42 comparisons + exchanges.

Thus 42 is the running cost for 7 items.

The average (number of comparisons + exchanges) is $7 = \frac{(12 + 10 + 8 + 6 + 4 + 2)}{6}$

Therefore, on average, each pass takes 7 running time units.

The number of passes is 6.

Therefore,

Running time = Number of passes x Average time per pass = 6 x 7 = 42.

…

Now,

Number of passes = NoOfItems - 1

n is the same as NoOfItems

Therefore, Number of passes = n - 1

Also, Average time per pass is the same as NoOfItems, i.e n

Therefore,

Running time = Number of passes x Average time per pass = $(n - 1) \times n = n^2 - n$

The asymptotic running time is then $O(n^2)$ i.e. for values of n larger enough that the term n^2 dominates.

In this chapter you have covered:

- Tracing and analysing the time complexity of the bubble sort algorithm which is $O(n^2)$.

Single licence - Abingdon School

3 Fundamentals of algorithms

3.5 Sorting algorithms

Learning objectives:

- *Be able to trace and analyse the time complexity of the merge sort algorithm*

3.5.2 Merge sort

Merge sort is an example of a divide and conquer algorithm, meaning that the problem of sorting an unsorted list of items is solved by dividing the problem into subproblems, solving the subproblems, and then merging the results to produce the sorted list.

The divide and conquer approach divides the work into roughly equal parts.

These equal parts are subdivided into roughly equal sub parts, and then each of these are divided equally, and so on until the work can be divided no further or until the problem is trivial to solve.

This approach is recursive as are almost all divide and conquer algorithms.

For example, suppose the task is to sort the following unsorted list of letters of the alphabet into ascending alphabetical order (A first, B second, and so on):

Unsorted list: G C H F A E D B

We divide the list into two separate sub lists with each containing half the letters:

1. Unsorted sub list: G C H F
2. Unsorted sub list: A E D B

Did you know?

Alternatively, the base case can be

- the sub list with two elements which, if necessary, are rearranged into the required order

or

- it can be one letter.

We then sort each sub list recursively using as base case the sub list with one or zero letters, which by default is already sorted.

The final merge occurs with two sorted sub lists C F G H and A B D E as shown in *Figure 3.5.2.1*.

This figure shows successive divide (split) and merge stages.

Task

Watch video of the merge sort algorithm at www.youtube.com/watch?v=EeQ8pwjQxTM

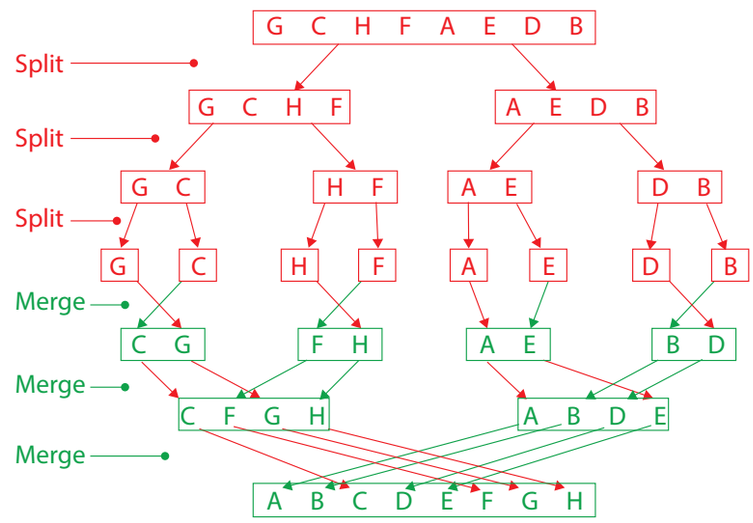


Figure 3.5.2.1 Dividing and merging an unsorted list into a sorted list

Single licence - Abingdon School

Figure 3.5.2.2 shows the merge sort algorithm in outline. *Figure 3.5.2.3* shows a recursive trace of this algorithm.

```
Subroutine MergeSort (Array A)
  If A has less than 2 elements
    Then Return
    Else
      If A has 2 elements
        Then
          If out of order
            Then exchange elements
          EndIf
          Return
      EndIf
      MergeSort(left half of A)
      MergeSort(right half of A)
      Merge left and right halves
   EndIf
EndSubroutine
```

Figure 3.5.2.2 Merge sort

Figure 3.5.2.3 Merge sort recursive trace

The algorithm for merging the left and right halves of a sub list is shown in *Figure 3.5.2.4*.

We need two arrays, one to copy from, array A, and one to copy to, array B.

In the video that you were tasked to view, this was simulated by moving the plastic cups between two levels, one upper and one lower.

This implies that the call `1.1.1 Merge left and right halves` and the call `1.1 Merge left and right halves` in *Figure 3.5.2.3* must use two arrays, corresponding to the two levels in the video.

When subdividing the lists we must ensure that merging using two arrays is correctly set up.

Figure 3.5.2.5 shows how this can be achieved.

The original unsorted list is in array U and a copy CU when `MergeSort (U, CU, 0, 8)` is called.

On return the sorted list is in array U.

Figure 3.5.2.6 shows a more detailed algorithm which uses two arrays.

Figures 3.5.2.7 and *3.5.2.8* show a trace of the final merge stage for call `1.1 Merge left and right halves`.

```
Subroutine Merge(A, B, Start, Middle, End)
  i ← Start
  j ← Middle
  Index ← Start
  While Index < End
    If (j >= End) Or (i < Middle And A[i] < A[j])
      Then
        B[Index] ← A[i]
        i ← i + 1
      Else
        B[Index] ← A[j]
        j ← j + 1
    EndIf
    Index ← Index + 1
  EndWhile
EndSubroutine
```

Figure 3.5.2.4 Merge subroutine

Single licence - Abingdon School

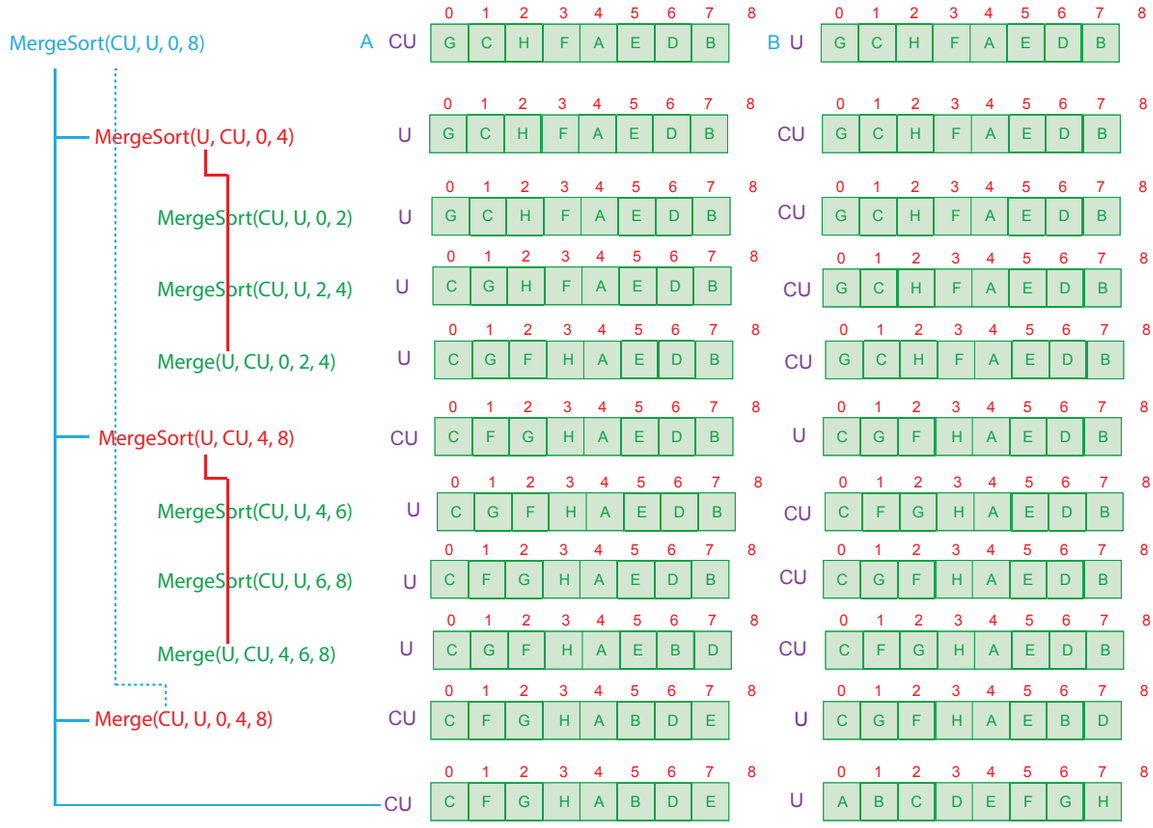
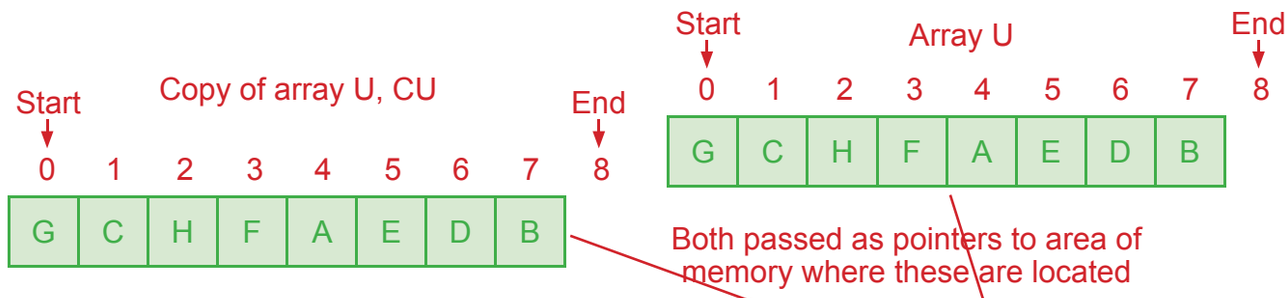


Figure 3.5.2.5 Recursive call trace for MergeSort(CU, U, 0, 8)




```
Subroutine MergeSort (Array A)
  If A has less than 2 elements
    Then Return
    Else
      If A has 2 elements
        Then
          If out of order
            Then exchange elements
          EndIf
          Return
      EndIf
      MergeSort(left half of A)
      MergeSort(right half of A)
      Merge left and right halves
   EndIf
EndSubroutine
```

```
Subroutine MergeSort(A, B, Start, End)
  If End - Start < 2
    Then Return {Exit subroutine}
  EndIf
  If End - Start = 2
    Then
      If B[Start] > B[Start + 1]
        Then exchange elements
      EndIf
      Return {Exit subroutine}
  EndIf
  Middle ← (Start + End) Div 2
  {Rearrange letters in left half of array B}
  MergeSort (B, A, Start, Middle)
  {Rearrange letters in right half of array B}
  MergeSort (B, A, Middle, End)
  Merge(A, B, Start, Middle, End)
EndSubroutine
```

Figure 3.5.2.6 Merge sort algorithm that uses two arrays and correspondence with previous algorithm

Single licence - Abingdon School

```
Subroutine Merge(A, B, Start, Middle, End)
  i ← Start
  j ← Middle
  Index ← Start
  While Index < End
    If (j >= End) Or (i < Middle And A[i] < A[j])
      Then
        B[Index] ← A[i]
        i ← i + 1
      Else
        B[Index] ← A[j]
        j ← j + 1
    EndIf
    Index ← Index + 1
  EndWhile
EndSubroutine
```

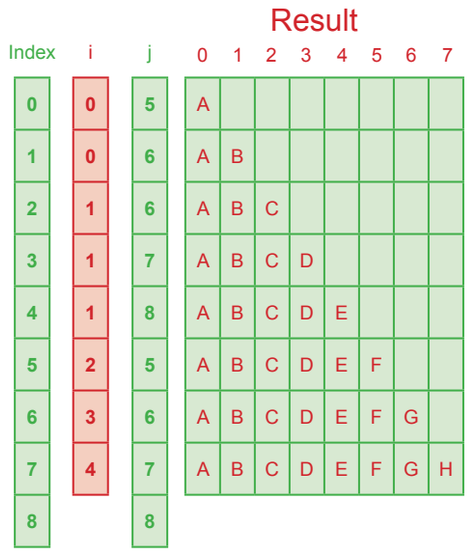


Figure 3.5.2.7 Trace table for final merge in which the two sorted halves are copied in the correct order into the result array by the given merge algorithm

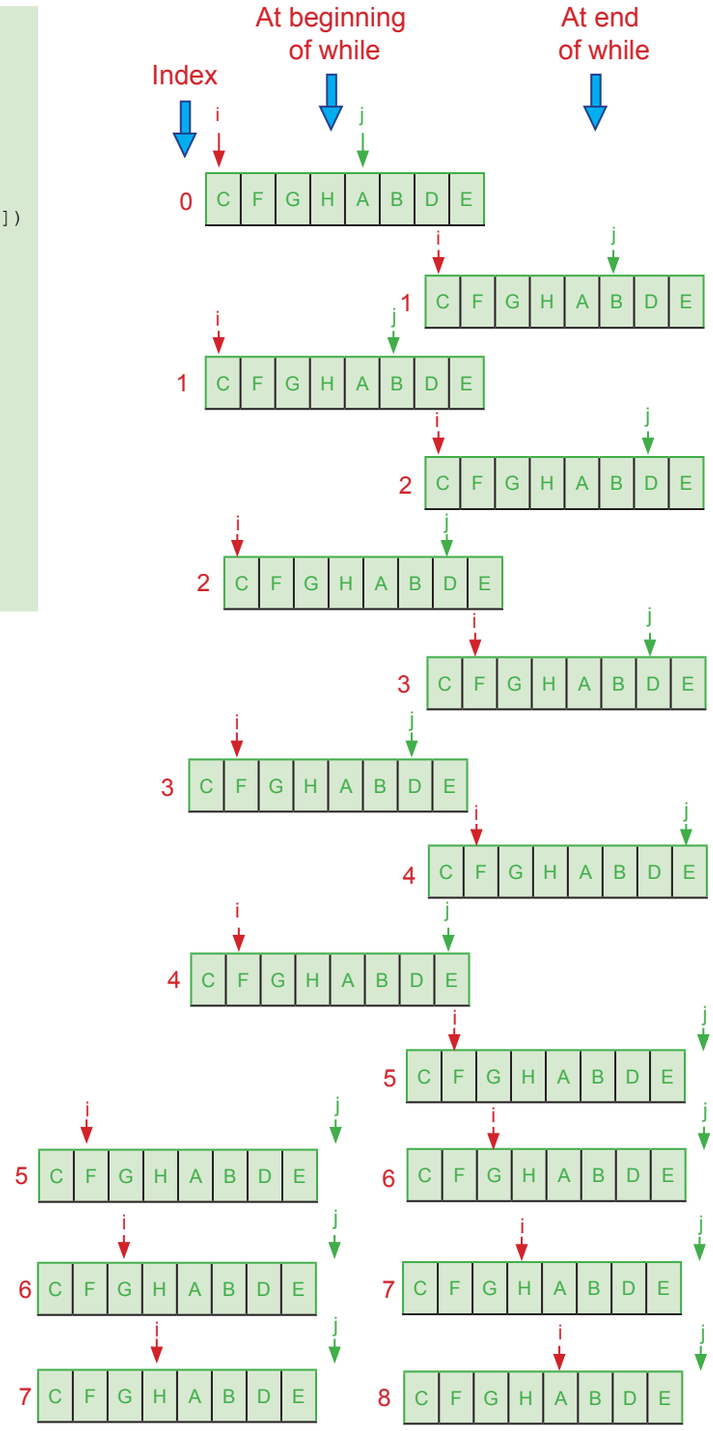


Figure 3.5.2.8 State of result array at beginning and end of While loop

Single licence - Abingdon School

Questions

1 Explain how merge sort would sort the following unsorted list of integers into a list of integers in ascending order:

54 45 33 28 19 12 7 3

2 Complete a copy of *Table 3.5.2.1*.

Start	End	Middle ← (Start + End) Div 2
0	8	
0	4	
0	2	
2	4	
4	8	
4	6	
6	8	

Table 3.5.2.1

3 On a copy of *Figure 3.5.2.10* trace the call to MergeSort(CU, U, 0, 4) using the algorithm shown in *Figure 3.5.2.9*. Array CU is a copy of array U.

```
Subroutine MergeSort(A, B, Start, End)
  {Parameter A : copy of array U,
   Parameter B : array U }
  If End - Start < 2
    Then Return {Exit subroutine}
  EndIf
  If End - Start = 2
    Then
      If B[Start] > B[Start + 1]
        Then exchange elements
      EndIf
      Return {Exit subroutine}
  EndIf
  Middle ← (Start + End) Div 2
  MergeSort (B, A, Start, Middle)
  MergeSort (B, A, Middle, End)
  Merge(A, B, Start, Middle, End)
EndSubroutine
```

```
Subroutine Merge(A, B, Start, Middle, End)
  i ← Start
  j ← Middle
  Index ← Start
  While Index < End
    If (j >= End) Or (i < Middle And A[i] < A[j])
      Then
        B[Index] ← A[i]
        i ← i + 1
      Else
        B[Index] ← A[j]
        j ← j + 1
    EndIf
    Index ← Index + 1
  EndWhile
EndSubroutine
```

Figure 3.5.2.9 Merge sort

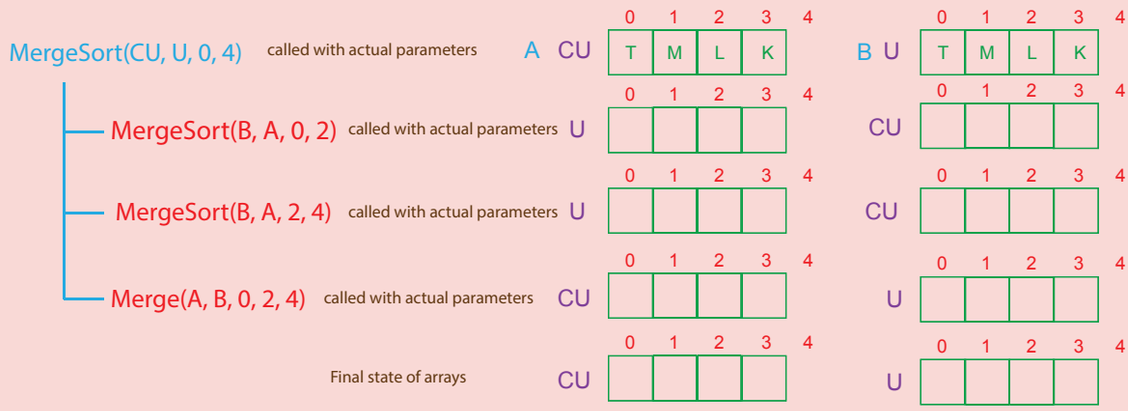


Figure 3.5.2.10 Trace table

4 The arrays CU and U are doubled in size so that they can accommodate 8 letters of the alphabet. The call to MergeSort(CU, U, 0, 8) now involves more recursive calls. Write down the sequence of recursive calls using actual parameters, CU and U for the algorithm shown in *Figure 3.5.2.9*.

Single licence - Abingdon School

Time complexity of merge sort algorithm

How long would one have to wait for merge sort to sort an unsorted list of items into ascending order?
It depends on the degree to which the list is already sorted. As with searching, the most useful time complexity to consider is the worst case time complexity. The worst case occurs when a list of items is as unordered as it can be, i.e.

- when the list is in descending order and the requirement is for merge sort to place the list in ascending order
- when the list is in ascending order and the requirement is for merge sort to place the list in descending order

Merge Sort completes the "merge" phase in $O(n)$ time after recursively sorting the left- and right-half of the range [start, end) of the array to be sorted, placing the properly ordered elements in the array referenced as the result array (see *Chapter 4.4.3 on* Big-O notation).

Merging involves `Index` ranging from `0` to `End - 1`. This range has length equal to the number of items to be sorted, i.e. the length of the list, n. The number of comparisons performed in the While loop shown in *Figure 3.5.2.11* is therefore linear in n.

```
...
While Index < End
    If (j >= End) Or (i < Middle And A[i] < A[j])
      Then
        B[Index] ← A[i]
        i ← i + 1
      Else
        B[Index] ← A[j]
        j ← j + 1
    EndIf
    Index ← Index + 1
EndWhile
...
```

Figure 3.5.2.11 Merge sort

For the recursive part of dividing the list in half, we will assume that n is even in order to keep the analysis as simple as possible.
The algorithm spends $O(n)$ time dividing the input into two pieces of size $n/_2$ each.
It then spends time $T(n/_2)$ to solve each one (T time is the worst-case time).
Finally it spends merge time combining the solutions.
The number of times the input must be halved in order to reduce its size from n to 2 is $\log_2 n$ (see *Chapter 3.4.2* on binary search).

Each time the size of the problem is halved so the number of new but smaller problems doubles.
For example, if sorting the full list takes cn time, then it is reasonable to assume that a sub list of length $n/_2$ will take time $cn/_2$ to sort.
For two sub lists each of length $n/_2$, the total time taken is $cn/_2 + cn/_2 = cn$.
At the next level created by dividing again there will be four problems each of size $n/_4$ and each taking $cn/_4$ time to sort.
The total time at this level is $cn/_4 + cn/_4 + cn/_4 + cn/_4 = cn$ where c is a constant. So the time taken at each level is the same.

Let $T(n)$ be the worst case running time of the procedure `MergeSort`.
We can write a recurrence or difference equation that upper bounds $T(n)$ as follows $T(n) <= 2T(n/_2) + c_1n$ if $n > 1$ where the second term covers the breaking of the list into two equal parts and the merge operation.

The splitting of the list in two and the merging operation is proportional to n.

Single licence - Abingdon School

We know that the first term has an upper bound of cn, e.g. if we consider the next level down of four sub lists we have the following $T(n/_2) <= 4T(n/_4) + c_1 n/_2$ if $n > 1$

Substituting for $T(n/_2)$ in $T(n) <= 2T(n/_2) + c_1 n$ we obtain $T(n) <= cn + 2c_1 n$
If we consider eight equal sub lists each of length $n/_8$ and substitute as before we obtain $T(n) <= cn + 3c_1 n$.
The number of times the input must be halved to reduce its size from n to 2 is $\log_2 n$, (e.g. if $1/_8$th of its size, $\log_2 n = 3$, $1/_4$ of its size $\log_2 n = 2$, etc).

Therefore, we may deduce that $T(n) <= cn + c_1 n \log_2 n$.

The total running time's upper bound for large enough n will be dominated by the second term, therefore we may say that the worst case time complexity for merge sort is $O(n \log_2 n)$.

Key point

Worst case time complexity of merge sort is $(n\log_2 n)$

In this chapter you have covered:

- Tracing and analysing the time complexity of the merge sort algorithm which is $O(n \log_2 n)$.

Single licence - Abingdon School

3 Fundamentals of algorithms

3.6 Optimisation algorithms

Learning objectives:

- *Understand and be able to trace Dijkstra's shortest path algorithm.*
- *Be aware of applications of shortest path algorithm*

3.6.1 Dijkstra's shortest path algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between vertices in a graph, which may represent, for example, a road network.

In particular, Dijkstra's algorithm solves the single-source shortest-path problem on a weighted, acyclic directed graph G = (V, E) for the case in which all edge weights are nonnegative.

In the single-source shortest path algorithm, one is given a specific vertex, s, and asked to compute the shortest path (by sum of edge weights) to all other vertices in the graph.

Graph G = (V, E) is defined by a set of vertices, V, and a set of edges, E, over pairs of these vertices.

Figure 3.6.1.1 shows an example of a weighted, acyclic directed graph.

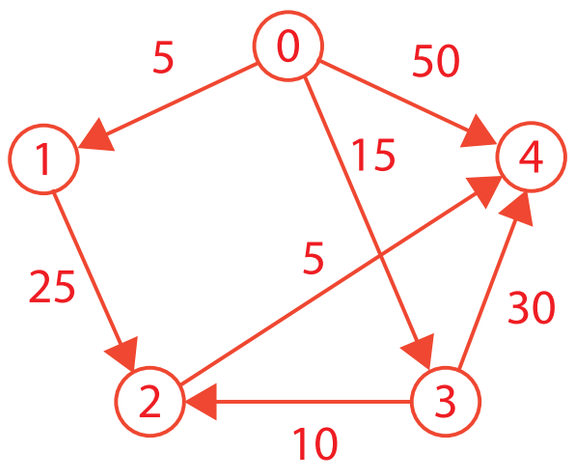


Figure 3.6.1.1 Weighted, directed acyclic graph G

The graph G in *Figure 3.6.1.1* is composed of five vertices and seven edges.

The numeric values on the edges in this graph might, for example, represent distances between towns, or the estimated travelling time.

Graph G could be stored as an adjacency matrix indexed in both dimensions by the vertices as shown in *Figure 3.6.1.2*. The entry A[*i*, *j*] stores the weight of the edge from vertex *i* to vertex *j*. When no edge exists from vertex *i* to vertex *j*, A[*i*, *j*] is set to a special value which in this case is ∞.

i \ *j*	0	1	2	3	4
0	0	5	∞	15	50
1	∞	0	25	∞	∞
2	∞	∞	0	∞	5
3	∞	∞	10	0	30
4	∞	∞	∞	∞	0

Figure 3.6.1.2 Graph G stored in an adjacency matrix A

Single licence - Abingdon School

Figure 3.6.1.3 Map of Australia showing some flight routes in red

Figure 3.6.1.3 shows a map of Australia overlaid with some flight routes between cities.

Suppose the task is to find the path that takes the least amount of time flying by commercial aeroplane from Adelaide to Darwin. Dijkstra's Shortest Path Algorithm may be used to solve this problem if the flight times are known between airports for all pairs of cities that are reachable from each other in one nonstop flight.

Dijkstra's algorithm can find the shortest time from Adelaide to all other connected airports, although the search may be halted once the shortest path to Darwin is discovered.

Although the shortest path is easily found by inspection in the simple examples in *Figures 3.6.1.1* and *3.6.1.3*, they are helpful in understanding how Dijkstra's algorithm achieves its goal.

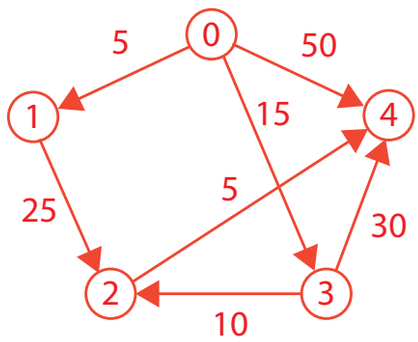


Figure 3.6.1.1 Repeated for convenience

Dijkstra's algorithm in outline

Figure 3.6.1.1 is repeated for convenience.

Its vertices can be expressed as a set V where V = {0, 1, 2, 3, 4}.

The algorithm works progressively through the vertices in V, adding them to set S, the set of visited vertices until every vertex has been visited.

Suppose we want to know the shortest distance from vertex 0 to every other vertex. We start by adding 0 to S, i.e. S = {0} and recording the distances from 0 to every vertex in D, a vector (one-dimensional array) as shown in *Table 3.6.1.1*. Vertex 2 is not directly reachable so ∞ is entered for this vertex in vector D. Vertices 1, 3 and 4 are directly reachable from 0 along paths 0-1, 0-3, and 0-4 respectively. The lengths of these (weights) are 5, 15, and 50 respectively.

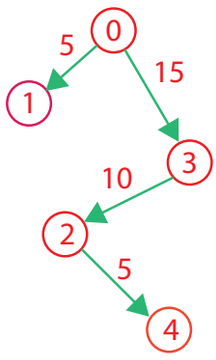


Figure 3.6.1.4 Tree T

D[0]	D[1]	D[2]	D[3]	D[4]
0	5	∞	15	50

Table 3.6.1.1 Vector D records distance from vertex 0

As the algorithm works through the graph, D is updated to reflect the shortest path discovered so far. When the algorithm is finished D will contain the shortest distances from vertex 0. Another vector P will contain these shortest paths.

Essence of algorithm:

Start with vertex 0 and greedily grow tree T until all the shortest paths have been found as shown in Figure 3.6.1.4. At each step, take edge to vertex that is closest to 0.

The phrase "greedily grow the tree T" means choose the next vertex to add to the tree that looks best at that moment, i.e. the one with the shortest known distance from vertex 0.

Task

Watch video of Dijkstra's shortest path algorithm at https://www.youtube.com/watch?v=pVfj6mxhdMw

Single licence - Abingdon School

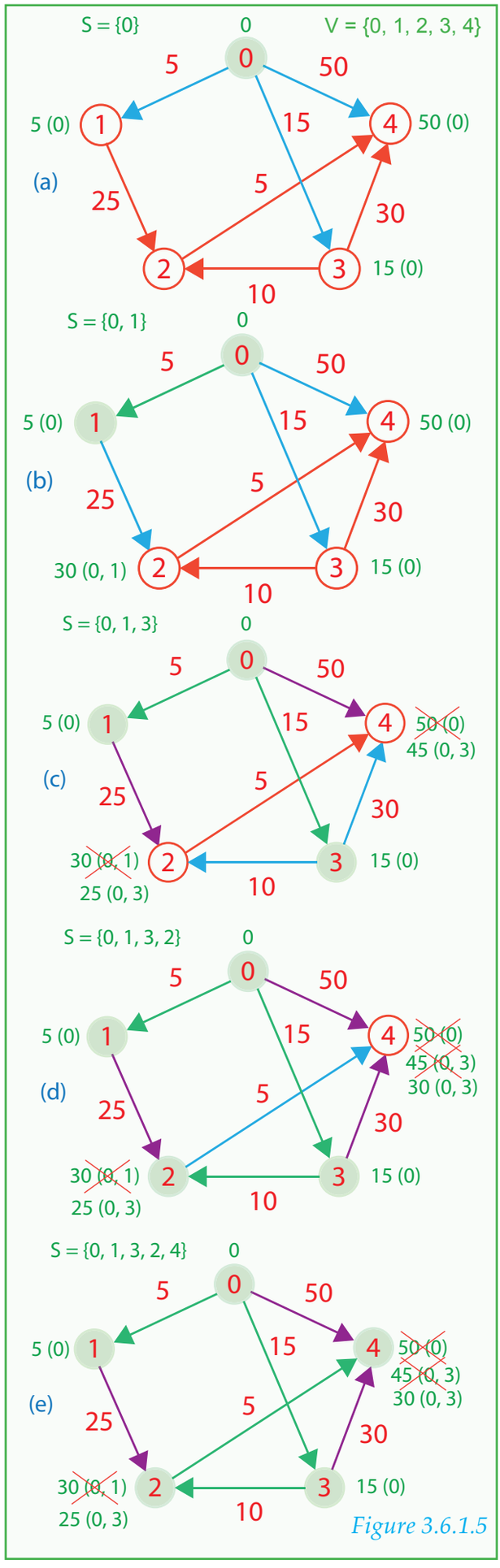


Figure 3.6.1.5

The first stage that is considered is shown in *Figure 3.6.1.5 (a)* with S = {0}. The edges to its unvisited neighbours are shown in blue. The initial stage with S = {} has been omitted so that the diagram will fit the page. This allows four iterations after the first stage to be shown which find the shortest paths from vertex 0 to the other four vertices. In each of these iterations the vertex in V - S which is the shortest known distance from vertex 0 is added to S. The edges to its unvisited neighbours are shown in blue. The vertices of the set S are filled in green. The shortest path from vertex 0 to each vertex is shown in green as it is discovered.

Vectors D and P are updated if necessary. The corresponding path to a vertex from vertex 0 is shown in brackets in the *Figure 3.1.6.5* together with the shortest known distance from vertex 0. The tree T as it grows is shown by green edges. An edge excluded from tree T is shown in **purple**. *Figure 3.6.1.5 (e)* shows the final result.

Tables 3.6.1.2, *3.6.1.3*, *3.6.1.4* show in vector D how the distance from vertex 0 changes during each iteration of the algorithm. Another vector P is used to record the predecessor of a vertex in the shortest path.

D[0]	D[1]	D[2]	D[3]	D[4]
0	5	∞	15	50

P[0]	P[1]	P[2]	P[3]	P[4]
0	0	-1	0	0

Table 3.6.1.2 Updated vector D and vector P which records vertices and their direct predecessor, stage (a) in Figure 3.6.1.5

D[0]	D[1]	D[2]	D[3]	D[4]
0	5	30	15	50

P[0]	P[1]	P[2]	P[3]	P[4]
0	0	1	0	0

Table 3.6.1.3 Updated vector D and vector P, stage (b) in Figure 3.6.1.5

D[0]	D[1]	D[2]	D[3]	D[4]
0	5	25	15	45

P[0]	P[1]	P[2]	P[3]	P[4]
0	0	3	0	3

Table 3.6.1.4 Updated vector D and vector P, stage (c) in Figure 3.6.1.5

D[0]	D[1]	D[2]	D[3]	D[4]
0	5	25	15	30

P[0]	P[1]	P[2]	P[3]	P[4]
0	0	3	0	2

Table 3.6.1.5 Updated vector D and vector P, stage (d)

Single licence - Abingdon School

Table 3.6.1.5 shows the shortest distances between vertex 0 and each of the other vertices. It also shows the shortest paths. For example, the shortest path from vertex 0 to vertex 4 is obtained from vector P by working backwards from 4 as follows

- P[4] says its predecessor is 2,
- P[2] says its predecessor is 3,
- P[3] says its predecessor is 0.

This gives the path 0-3-2-4.

Figure 3.6.1.6 shows the steps in Dijkstra's algorithm. *Figure 3.6.1.5* omitted the first step in order to fit the diagram on the page.

```
Predecessor of all vertices ← -1
Distance of start vertex from start vertex ← 0
Predecessor of start vertex ← 0
Distance of all other vertices from start vertex ← ∞
Visited vertices S ← {}
Unvisited vertices, U ← {...}
Repeat
  Visit the unvisited vertex, w, with the smallest known distance from the start vertex
  Add this vertex to the set of visited vertices, S
  Calculate distance of each unvisited neighbour from start vertex
  If calculated distance of an unvisited neighbour vertex
          from start vertex < its known distance from start vertex, then update the distance
  Update the previous vertex for each updated distance
Until all vertices visited
```

Figure 3.6.1.6 Dijkstra's shortest path algorithm for a single source, vertex 0

Questions

1 Trace the algorithm shown in *Figure 3.6.1.6* for the directed graph shown in *Figure 3.6.1.7*, completing a copy of the table shown In *Table 3.6.1.6*.

The array D records the shortest distance of each vertex from the vertex 0.

The array P records the shortest path from vertex 0 to every vertex.

Each cell of this array stores the vertex number of its predecessor in the shortest path to it. The set U is a set of all unvisited vertices in the graph.

The set S is a set of all visited vertices in the graph.

The first row of the trace table has been completed for you.

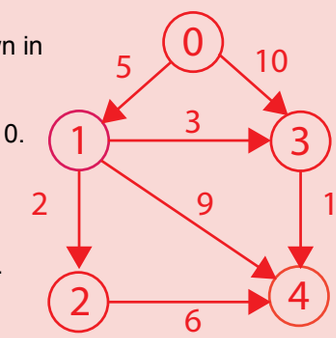


Figure 3.6.1.7

Iteration	S	U	w	D[0]	D[1]	D[2]	D[3]	D[4]	P[0]	P[1]	P[2]	P[3]	P[4]
Initial	{}	{0,1, 2, 3, 4}	-	0	∞	∞	∞	∞	0				
1													
2													
3													
4													
5													

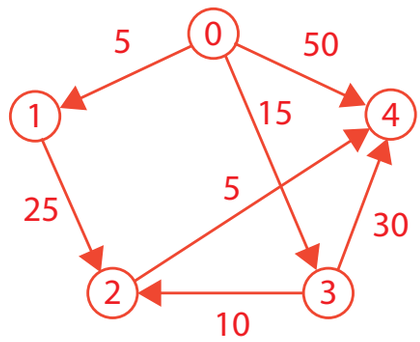
Table 3.6.1.6 Trace table

Single licence - Abingdon School

Dijkstra's algorithm in detail

Figure 3.6.1.8 shows Dijkstra's shortest path algorithm for a single starting vertex or single source, vertex 0. Its input is a set S containing every vertex in the graph, and the adjacency matrix A which stores the graph.

For the graph in *Figure 3.6.1.1*, S = {0, 1, 2, 3,4}, and A =



i \ j	0	1	2	3	4
0	0	5	∞	15	50
1	∞	0	25	∞	∞
2	∞	∞	0	∞	5
3	∞	∞	10	0	30
4	∞	∞	∞	∞	0

Figure 3.6.1.1 Repeated for convenience

```
Subroutine SingleSourceShortestPath
  {Inputs:
     V, set containing every vertex in graph
     A, adjacency matrix representing graph}
  ----------------------------------------------
  S ← {}, D[0] ← 0, P[0] ← 0
  For i ← 1 To NoOfVertices - 1
    D[i] ← ∞
    P[i] ← -1 {P references vertex immediately before vertex i in shortest path}
              {initialised to null pointer, - 1}
  EndFor
  For NoOfIterations ← 0 To NoOfVertices - 1
    Choose a vertex w in V - S such that D[w] is a minimum
    Add w to S
    For each vertex v in V - S
      D[v] ← Minimum(D[v], D[w] + A[w, v])
      If D[w] + A[w, v] < D[v]
        Then P[v] ← w
    EndFor
  EndFor
EndSubroutine
```

distance of vertex w from vertex 0

vertex 0 chosen on 1st iteration

distance of vertex v from vertex 0

distance of vertex v from vertex w

distance of vertex w from vertex 0

Figure 3.6.1.8 Dijkstra's shortest path algorithm for a single source, vertex 0

Hand-tracing Dijkstra's algorithm in detail

To trace the algorithm in *Figure 3.6.1.8* we start from vertex 0.

We initialise distances from vertex 0 to every vertex in the graph in a vector (one-dimensional array) D as shown in *Table 3.6.1.7*. We initialise a predecessor array P as shown in *Table 3.6.1.7*.

D[0]	D[1]	D[2]	D[3]	D[4]
0	∞	∞	∞	∞

P[0]	P[1]	P[2]	P[3]	P[4]
0	-1	-1	-1	-1

Table 3.6.1.7 Vector D records distance from vertex 0, vector P records predecessor in shortest path

We now start the first of five iterations. *Table 3.6.1.8* shows the stages of these five iterations and their effect on the vectors D and P. *Table 3.6.1.9* shows more detail of the iterations.

Single licence - Abingdon School

Tables 3.6.1.8 and *3.6.1.9* show that in the first iteration of the algorithm,

vertex 0 is added to S which becomes S = {0}.

1.1 Now we consider what remains, i.e. V - S = {1, 2, 3, 4}.

1.2 The distances from 0 to 1, 2, 3, and 4 are 5, ∞ ,15, and 50 respectively.

1.3 We want to know if going from 0 to any of these, if possible, is shorter than going the currently recorded distance (distance ∞ in *Table 3.6.1.7*)?

1.4 If going from 0 to any of these is shorter then we update D's entry for the vertex and mark its predecessor as 0 in vector P.

Table 3.6.1.9 shows the outcome in the first four rows. This is summarised in the first row in *Table 3.6.1.8*.

Itera-tion	S	V - S	w	D[0]	D[1]	D[2]	D[3]	D[4]	P[0]	P[1]	P[2]	P[3]	P[4]
1	{0}	{1, 2, 3, 4}	0	0	5	∞	15	50	0	0	-1	0	0
2	{0, 1}	{2, 3, 4}	1	0	5	30	15	50	0	0	1	0	0
3	{0, 1, 3}	{2,4}	3	0	5	25	15	45	0	0	3	0	3
4	{0, 1, 3, 2}	{4}	2	0	5	25	15	30	0	0	3	0	2
5	{0, 1, 3, 2, 4}	{}	4	0	5	25	15	30	0	0	3	0	2

Table 3.6.1.8 Trace table showing finding the shortest paths between 0 and the vertices of set S

V - S	w	v	D[v]	D[w]	A[w, v]	D[w] + A[w, v]	Minimum (D[v], D[w] + A[w, v])	D[w] + A[w, v] < D[v]?	P[v]
{1, 2, 3, 4}	0	1	∞	0	5	5	5	Yes	P[1]←0
{1, 2, 3, 4}	0	2	∞	0	∞	∞	∞	No	
{1, 2, 3, 4}	0	3	∞	0	15	15	15	Yes	P[3]←0
{1, 2, 3, 4}	0	4	∞	0	50	50	50	Yes	P[4]←0
{2, 3, 4}	1	2	∞	5	25	5 + 25 = 30	30	Yes	P[2]←1
{2, 3, 4}	1	3	15	5	∞	∞	15	No	
{2, 3, 4}	1	4	50	5	∞	∞	50	No	
{2,4}	3	2	30	15	10	15 + 10 = 25	25	Yes	P[2] ←3
{2,4}	3	4	50	15	30	15 + 30 = 45	45	Yes	P[4] ←3
{4}	2	4	45	25	5	25 + 5 = 30	30	Yes	P[4] ←2
{}	4	-	-	25	-	-	-	-	-

Table 3.6.1.9 Trace table showing the iterations in greater detail for finding the shortest paths between 0 and the vertices of set S

Tables 3.6.1.8 and *3.6.1.9* show that in the next iteration of the algorithm,

vertex 1 is added to S which becomes S = {0, 1}.

2.1 Now we consider what remains, i.e. V - S = {2, 3, 4}.

2.2 The distances from 0 to 2, 3, and 4 are ∞ ,15, and 50 respectively.

2.3 We want to know if going from 0 through 1 to any of these, if possible, is shorter than going directly from 0, e.g. is going along the path 0-1-2 shorter than going directly along path 0-2 (distance ∞)?

Single licence - Abingdon School

2.4 If going from 0 through 1 to any of these is shorter then we update D's entry for the vertex and mark its predecessor as 1 in another vector P.

2.5 D[2] has been updated to reflect the fact that here is a shorter path of distance 30 along path 0-1-2.

2.6 It also shows the shortest paths determined so far, e.g. the shortest path from 0 to 2, vertex 2's predecessor is 1 and 1's predecessor is 0, and the shortest path between 0 to 4 is the path where 4's predecessor is 0, and so on.

3.1 The process is now repeated, by choosing a vertex from V - S = {2, 3, 4} that has the shortest distance from vertex 0. This is vertex 3. We add vertex 3 to S which becomes {0, 1, 3}.

3.2 Now we consider what remains, i.e. V - S = {2, 4}.

3.3 We first check whether or not the distance from 0 to 3 plus the distance from 3 to 2 is shorter than the distance from 0 to 2 recorded in D.

3.4 It is, so we have found a shorter path 0-3-2 of distance 15 + 10 = 25 than path 0-1-2 of path length 30. Vectors D and P are updated accordingly.

A new shortest path is shown in *Tables 3.6.1.8* and *3.6.1.9* between 0 and 2.

4.1 We next check whether or not the distance from 0 to 3 plus the distance from 3 to 4 is shorter than the distance from 0 to 4 recorded in D.

4.2 It is, so we have found a shorter path 0-3-4 of distance 15 + 30 = 45 than path 0-4 of path length 50. Vectors D and P are updated accordingly to show the new shortest path between 0 and 4.

5.1 We next repeat the process, by choosing a vertex from V - S = {2, 4} that has the shortest distance from vertex 0.

5.2 This is vertex 2. We add vertex 2 to S which becomes {0, 1, 3, 2}.

5.3 Now we consider what remains, i.e. V - S = {4}.

5.4 We first check whether or not the distance from 0 to 2 plus the distance from 2 to 4 is shorter than the distance from 0 to 4 recorded in D.

5.5 It is, so we have found a shorter path 0-3-2-4 of distance 15 + 10 + 5 = 30 than path 0-3-4 of path length 45. *Table 3.6.1.8* shows the updated vectors D and P.

6.1 We next repeat the process, by choosing a vertex from V - S = {4} that has the shortest distance from vertex 0.

6.2 We have just one vertex to choose, vertex 4.

6.3 We add vertex 4 to S which becomes {0, 1, 3, 2, 4}.

7.1 Now we consider what remains, i.e. V - S = {}. We are done.

Single licence - Abingdon School

Questions

2 Complete a copy of the table shown in *Table 3.6.1.10* by tracing the algorithm shown in *Figure 3.6.1.9* for the directed graph of flight routes, shown in *Figure 3.6.1.10*, between some Australian cities and the corresponding flight cost for each route. The array D records the cost of the cheapest flight to each vertex from the vertex 0. The array P records the cheapest path from vertex 0 to every vertex. Each cell of array P stores the vertex number of its predecessor in the cheapest path to it. The set V - S is the set of all unvisited vertices in the graph. The set S is a set of visited vertices.

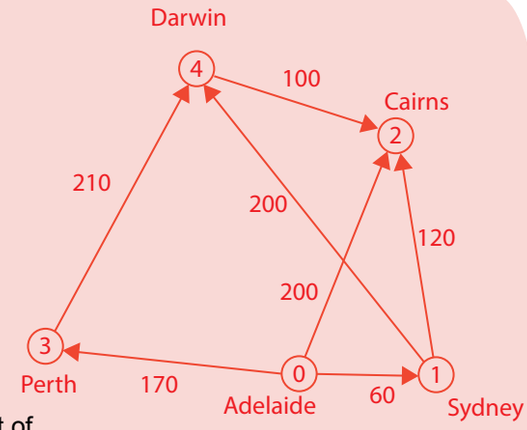


Figure 3.6.1.10

Figure 3.6.1.11 is the equivalent adjacency matrix for the directed graph. The first row of the trace table has been completed for you.

i \ j	0	1	2	3	4
0	0	60	200	170	∞
1	∞	0	120	∞	200
2	∞	∞	0	∞	∞
3	∞	∞	∞	0	210
4	∞	∞	100	∞	0

Figure 3.6.1.11

```
Subroutine SingleSourceShortestPath
  S ← {}, D[0] ← 0, P[0] ← 0
  For i ← 1 To NoOfVertices - 1
    D[i] ← ∞
    P[i] ← -1  {P references vertex immediately before vertex i in
                shortest path, initialised to null pointer, - 1}
  EndFor
  For NoOfIterations ← 0 To NoOfVertices - 1
    Choose a vertex w in V - S such that D[w] is a minimum
    Add w to S
    For each vertex v in V - S
      D[v] ← Minimum(D[v], D[w] + A[w, v])
      If D[w] + A[w, v] < D[v]
        Then P[v] ← w
    EndFor
  EndFor
EndSubroutine
```

Figure 3.6.1.9

Iteration	S	V - S	w	D[0]	D[1]	D[2]	D[3]	D[4]	P[0]	P[1]	P[2]	P[3]	P[4]
1	{0}	{1, 2, 3, 4}	-	0	60	200	170	∞	0	0	0	0	-1
2													
3													
4													
5													

Table 3.6.1.10 Trace table

Single licence - Abingdon School

Applications of Dikjstra's shortest path algorithm

Dijkstra's algorithm finds the shortest path or the path with the least cost in a graph.
It can therefore be used in any field involving

- finding the shortest route, e.g. if the vertices are locations, the edges are roads, and the weights are the road distances between the locations then Dijkstra's algorithm may be used to find the shortest route by road between two given locations
- finding the route with the shortest cost where cost could be money or time or some other cost quantity, e.g. if the vertices are routers in a computer network, the edges are network links between routers, and the weights the times for a packet to traverse links then Dijkstra's algorithm may be used to find the network path between two given routers for which packet travel time is a minimum. The source router will maintain a record of this time in its forwarding table.

Questions

3 State **two** applications of Dijkstra's shortest path algorithm.

4 A travel agent requests software for making an agenda of flights for clients. The software will have access to a database with all airports, direct flights between these airports and their flight times. The agent wants the software to determine the shortest time in the air for the destination given an origin airport. Explain in principle how this could be done by applying an appropriate algorithm to a graph representation for the given scenario.

Extension question

5 The travel agent now requests software for making an agenda of flights for clients. The software will have access to a database with all airports and flights. Besides the flight number, origin airport and destination, the flights have departure and arrival time. Specifically the agent wants to determine the earliest arrival time for the destination given an origin airport and start time. Explain in principle how this could be done by applying an appropriate algorithm to a graph representation for the given scenario.

In this chapter you have covered:

- Understand and be able to trace Dijkstra's shortest path algorithm.
- Be aware of applications of shortest path algorithm

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Be able to develop solutions to simple logic problems.*
- *Be able to check solutions to simple logic problems.*

Key term

Proposition:
A proposition can be defined as a statement which can be either true or false, e.g. "Today is Friday"

Information

An argument:
An argument in propositional logic is a sequence of propositions. All but the final proposition in the argument are called premises and the final proposition is called the conclusion.
An argument is valid if the truth of all its premises implies that the conclusion is true.

Argument form:
An argument form in propositional logic is a sequence of propositions involving propositional variables, e.g. P, Q. An argument form is valid if no matter which particular propositions are substituted for the propositional variables in its premises, the conclusion is true if the premises are all true.
For example, a given set of propositions $P_1, P_2, P_3, \ldots P_n$, called premises, yields another proposition Q called the conclusion. Such an argument is denoted by the argument form

$$P_1, P_2, P_3, \ldots P_n \vdash Q$$

$P_1, P_2, P_3, \ldots P_n$ and Q are propositional variables. The symbol $\vdash$ denotes an assertion.

4.1.1 Problem solving

Developing and checking solutions to simple logic problems

It is well-known that it snows only if it is cold. "It snows only if it is cold" is called a **proposition**.

> **A proposition can be defined as a statement which can be either *true* or *false*.**

For example, "Today is Friday" is a proposition which is *true* or *false*. If today is Friday then it is *true*, if it isn't Friday today it is *false*.

The following are alternative phrasings of the proposition "It snows only if it is cold":

- "If *it snows*, *it is cold*"
- "If *it is snowing*, *it is cold*"
- "If *it snows* then *it is cold*"
- "If *it is snowing* then *it is cold*"
- "A necessary condition for *it to snow* is *it must be cold*"
- "*Being cold* is necessary for *it to be snowing*".
- "*It is cold* when *it snows*"
- "*It is cold* whenever *it snows*"
- "*It is cold* if *it snows*"
- "*It is snowing* implies *it is cold*"
- "*It is cold* follows from *it is snowing*"
- "a sufficient condition for it *to be cold* is *it is snowing*".

Question

1 Write each of these statements in the form "*If P, then Q*".
(a) It snows whenever the wind blows from the north.
(b) Getting unfit follows from not exercising enough.
(c) Leaves on the trees turning brown implies it is autumn.
(d) Being at a temperature of 100 °C is necessary for water to boil.
(e) A sufficient condition for a refund is that you bought the goods in the last two weeks.

Compound proposition

Actually, "It snows only if it is cold" is a compound proposition because it is made up of

- two simpler propositions "It snows" and "It is cold" and
- the conditional connective, "only if".

If we know that "It snows only if it is cold" is *true* then "It snows" or "It is snowing" is *true* only if "it is cold" is *true*.

Single licence - Abingdon School

We use the major proposition and the minor proposition as shown below **to infer** or argue **by deduction** that if "*If it is snowing then it is cold*" is *true* and if "*It is snowing*" is *true* we may conclude that "*It is cold*" is *true*.

Major proposition: *If it is snowing then it is cold*

Minor proposition: *It is snowing*

Conclusion: *Therefore, it is cold*

Now consider

Major proposition: *If it is snowing then it is cold*

Minor proposition: *It is cold*

Conclusion: *Therefore, it is snowing*

Is this a valid conclusion?

The answer is no. It can be cold and not snowing at the same time. We are not entitled to deduce that it is snowing just because it is cold.

Question

2 Given that the two propositions are true
- If it is cold, he wears a hat
- It is cold

What conclusion can you draw?
Layout your argument in the form of a major and a minor proposition followed by a conclusion as shown above.

4 A politician says in his manifesto:
"*If I am elected, then I will lower taxes.*"
The politician is elected.
Assuming that the two statements are true, what conclusion are you entitled to draw?

6 A politician says in his manifesto:
"*If I am elected, then I will lower taxes.*"
The following are true statements:
The politician is elected.
The politician doesn't lower taxes.
Is the politician's manifesto statement a true statement?

3 Given that the two propositions are true
- If it is cold, he wears a hat
- He wears a hat

Are you entitled to deduce that "it is cold"?
Explain your answer.

5 A politician says in his manifesto:
"*If I am elected, then I will lower taxes.*"
The politician manages to get taxes lowered.
Assuming that the two statements are true, are you entitled to deduce that the politician got elected? Explain your answer.

7 A student, Alice, says to another student, Ben:
"*If you have a valid network password, then you can log onto the network.*"
Ben replies
"*I have a valid network password.*"
What are the logical conditions that must apply to these two statements for the conclusion "*Ben can log onto the network*" to be true?

Suppose without looking, you take a card from a pack of 52 playing cards and place it face down on the table and say "If the card is the King of Diamonds, then it is red." Anyone familiar with the colour of playing card suits (Diamonds and Hearts are red, Spades and Clubs are black) will know that this statement is true.

If you now turn over this card to reveal that it is the Queen of Spades, a black card, is it still true that "If the card is the King of Diamonds, then it is red"? The answer is, of course, yes.

Single licence - Abingdon School

Table 4.1.1 shows the truth table for the proposition "If the card is the King of Diamonds, then it is red." The only case when this proposition is *false* is when "The card is the King of Diamonds" is true and "It is red" is false.

If the card is not the King of Diamonds then it could be red or black in colour but the column headed "If the card is the King of Diamonds, then it is red." will still be *true* for these rows.

The card is the King of Diamonds	It is red	If the card is the King of Diamonds, then it is red
True	True	True
True	False	False
False	True	True
False	False	True

Table 4.1.1 Truth table for the truth of the proposition "If the card is the King of Diamonds, then it is red"

Major proposition: The card is the King of Diamonds only if it is red

Minor proposition: The card is the King of Diamonds

Conclusion: *Therefore, it is a red card*

Figure 4.1.1 Valid argument and a conclusion which is true

Major proposition: The card is the King of Diamonds only if it is red

Minor proposition: The card is not the King of Diamonds

Conclusion: Therefore, *?*

Figure 4.1.2 Valid argument but the conclusion is inconclusive

In *Figure 4.1.1*, the conclusion "*It is a red card*" is true because both major and minor propositions are true.

In *Figure 4.1.2*, both major and minor propositions are true. However, we cannot draw any conclusion other than the card could be red or black because we know that cards are either red or black. But restricting ourselves to the evidence supplied, we are not entitled to use this additional knowledge.

Question

8 Draw the truth table for the proposition "*If I am elected, then I will lower taxes.*"
State the truth values for the row of your table which corresponds to the politician breaking his promise?

Conditional connective ⇒

If we know that "*It is snowing*" is true only if "*It is cold*" is true then using propositional variables, P and Q, where P = "It is snowing", Q = "It is cold", we write P ⇒ Q which reads P is true only if Q is true or P implies Q.
In the conditional statement P ⇒ Q, P is called the hypothesis (or antecedent or premise) and Q is called the conclusion (or consequence). So if the conditional statement is true and hypothesis is true then the conclusion is true.

The argument

Major proposition: *If it is snowing then it is cold*

Minor proposition: *It is snowing*

Conclusion: *Therefore, it is cold*

is a valid argument with a conclusion which is true.

Information

Conditional connective ⇒:
If we know that "*It is snowing*" is true only if "*It is cold*" is true then using propositional variables, P and Q, where P = "*It is snowing*", Q = "*It is cold*" we write

P ⇒ Q

which reads P is true only if Q is true.
In the conditional statement P ⇒ Q, P is called the hypothesis (or antecedent or premise) and Q is called the conclusion (or consequence). So if the conditional statement is true and the hypothesis is true then the conclusion is true.

Single licence - Abingdon School

Using propositional variables, the equivalent **argument form** of the above **argument** is

where ∴ is the symbol that denotes "therefore".

Whenever, both P and P ⇒ Q are true then Q is also true.

$$\begin{array}{l} P \Rightarrow Q \\ P \\ \hline \therefore Q \end{array}$$

The form of the argument that uses propositional logic variables, e.g. P and Q, is called the **argument form** to differentiate it from the **argument** which uses actual statements such as "It is snowing".

Is the conclusion always true if the argument is valid?

Consider the following valid argument

Major proposition: *If computer program A compiles then it will terminate when run*

Minor proposition: *Computer program A compiles*

Conclusion: *Therefore, it will terminate when run*

Even though this is a valid argument because the argument form is

the conclusion is not true if the major proposition is not true.

It is a fact that some programs can get stuck in a loop and therefore will not terminate.

$$\begin{array}{l} P \Rightarrow Q \\ P \\ \hline \therefore Q \end{array}$$

Question

9 Consider the following argument involving propositions:
"*If the card is the King of Diamonds, then it is red.*"
"*The card is the King of Diamonds.*"
Therefore, "The card is red."
Write the argument form using the propositional variables, P and Q, where P = "*The card is the King of Diamonds*" and Q = "*It is red.*" Is this a valid argument? Is the conclusion true?

10 The following argument consisting of two propositions and another proposition, the conclusion, is a valid argument but the conclusion is not true even though it is snowing. Why is the conclusion not true?

If it is snowing, it is warm.
It is snowing.
Therefore, it is warm.

11 The following valid argument consists of three propositions, one of which is the conclusion. The first two propositions are true. Is the conclusion true?

If it is snowing then it is cold.
It is not cold.
Therefore, it is not snowing.

12 The following valid argument consists of three propositions, one of which is the conclusion. The first two propositions are true. Is the conclusion true?

It is snowing or it is cold or it is both.
It is not snowing.
Therefore, it is cold.

Single licence - Abingdon School

Logical reasoning

A fundamental principle of logical reasoning states:

If P then Q, If Q then R, therefore, If P then R

that is, the following argument is valid

$$P \Rightarrow Q,\ \ Q \Rightarrow R \vdash P \Rightarrow R$$

Consider the following statements

- If it is freezing, the streets are icy.
- If the streets are icy, accidents will happen

Applying the fundamental principle of logical reasoning we get

It is freezing implies the streets are icy	*It is freezing* $\Rightarrow$ *the streets are icy*	$P \Rightarrow Q$
the streets are icy implies accidents will happen	*the streets are icy* $\Rightarrow$ *accidents will happen*	$Q \Rightarrow R$
It is freezing implies accidents will happen	*It is freezing* $\Rightarrow$ *accidents will happen*	$\therefore P \Rightarrow R$

Question

13 *If it snows today, then I will not go to school today. If I do not go to school today, I can catch up on my homework.*
Use logical reasoning to draw a conclusion.

14 *If the train arrives early then Jamin will be early for his meeting. If Jamin is early for his meeting, he can call in on his friend John.* What conclusion can you draw?

15 *If I don't watch the late night film, I will go to bed early. If I go to bed early, I will wake up feeling refreshed.* What conclusion can you draw?

16 Consider the following scenario:

If Alex is allowed a TV in his bedroom, he will neglect his homework. If he neglects his homework, he will fall behind at school. If he falls behind at school, he will not progress at school. If he does not progress at school, he will need extra tution.

Use logical reasoning to draw a conclusion.

17 *If the plane arrives late and there are no taxis at the airport then Jack is late for his meeting. Jack is not late for his meeting. The plane did arrive late.* What conclusion can you draw?

18 *If it is raining and Isla does not have her raincoat with her, then she will get wet. Isla is not wet. It is raining.* What conclusion can you draw.

19 Consider this rule about a set of cards:

"*If a card has a vowel on one side, then it has an even number on the other side.*"

Look at the cards below and answer the question which cards do I need to turn over to tell if the rule is actually true? Explain your reasoning.

U	L	8	3

Single licence - Abingdon School

Did you know?

Mathematical induction: Unfortunately, the term "mathematical induction" clashes with the terminology used in reasoning by logic. Deductive reasoning uses rules of inference to draw conclusions from premises, whereas inductive reasoning makes conclusions only supported, but not guaranteed, by evidence. Proofs that use mathematical induction are deductive, not inductive.

Did you know?

Rationalism and empiricism: Rationalism is the philosophy that believes that the senses deceive and that logical reasoning is the only sure path to knowledge.
Empiricism is the philosophy that believes that all reasoning is fallible and that knowledge must come from observation and experimentation.
Empiricism gave birth to the scientific method during the Enlightenment.
Rationalism was responsible for the invention of some fundamental mathematical techniques such as calculus but on theories of the universe did not do so well. The empiricists' approach of generalising from what has been observed and measured has been more successful. This approach has also proved useful in machine learning.

From deduction to induction

Newton is well known for his three laws of motion. He is less well known for his four rules of induction. His third rule of induction is said to be at the heart of the Newtonian revolution and modern science. This rule, known as Newton's Principle, can be expressed in modern parlance as follows

> *Newton's Principle: Whatever is true of everything before our eyes is true of everything in the universe.*

Newton's principle allows scientists to draw conclusions of a general nature from the results of experiments involving a limited set of observations and measurements.

We have learned so far about deductive reasoning, e.g.

John Smith is a footballer

All footballers kick a football when playing soccer

Therefore, ...?.....

The first statement is a fact about John Smith and the second is a general rule about footballers. We can infer from these two statements that *John Smith kicks a football when playing soccer* by applying the general rule to John.

Inductive reasoning is the inverse of **deductive reasoning**. In inductive reasoning we start instead with the initial and the derived facts, and look for a rule that allows the derived fact(s) to be inferred from the initial fact(s). For example,

John Smith is a footballer

.....?.....

Therefore, John Smith kicks a football when playing soccer

One such rule is: *If John Smith is a footballer, then he wears football boots when playing soccer*

This is a rule but not a very useful one because it is specific to John Smith. We use Newton's principle to generalise this rule to make a more useful general rule:

If x is a footballer, then x wears football boots when playing soccer

where x can be anyone including people who are not footballers.

Another way of expressing this is

All footballers wear football boots when playing soccer

We would need more evidence to have faith in this general rule. This could be obtained by observing more than just one footballer playing soccer and noting over and over again that they wear football boots when playing soccer.

Single licence - Abingdon School

Note that the statement

If x is a footballer, then x wears football boots when playing soccer

is neither true nor false whilst the value of x is unknown, but if x = *John Smith* then it becomes

If John Smith is a footballer, then John Smith wears football boots when playing soccer

This is either true or false depending on whether John is a footballer. It is therefore a proposition.

In this logic we have statements that are not propositions until a value has been assigned to the variable(s) in the propositional function.

If we denote

If x is a footballer, then x wears football boots when playing soccer by P(x)

then P("John Smith") is a proposition. P(x) is said to be the value of the propositional function P at x.

Question

20 Let P(x) denote the statement "x > 5". What are the truth values of P(3) and P(7)?

21 The following statements are true:

Gerry is a computer science student.
All computer science students drink coffee.

What conclusion can you draw?

22 The following statements are true:

Deemei is in class CS1.
Every student in class CS1 has learned at least one programming language.

What conclusion can you draw?

Extension Question

23 A boy and a girl, whose names we do not know, are sitting next to each other. One of them has fair hair, and the other dark hair. The dark-haired one says "*I am a girl.*" The fair-haired one says, "*I am a boy*". We are told at least one of them is lying. What is your conclusion?

In this chapter you have covered:

- Developing solutions to simple logic problems.
- Checking solutions to simple logic problems.

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Understand the term algorithm*
- *Be able to express the solution to a simple problem as an algorithm using pseudo-code, with the standard constructs:*
  - *sequence*
  - *assignment*
  - *selection*
  - *iteration*
- *Be able to hand-trace algorithms*
- *Be able to convert an algorithm from pseudo-code into high level language program code*
- *Be able to articulate how a program works, arguing for its correctness and its efficiency using logical reasoning, test data and user feedback.*

4.1.2 Following and writing algorithms

What is an algorithm?

An algorithm is a precise description of steps necessary to accomplish a certain task or solve a particular problem.

The notion of an algorithm is not peculiar to computer science.

Algorithms have been around for thousands of years.

Mathematics is full of algorithms, some of which we rely on today, e.g. the algorithm we use for adding two numbers column-by-column.

Table 4.1.2.1 shows some fields of human activity in which algorithms are used. In each case the process is carried out by a human being.

Process	Algorithm	Example steps
Knitting a cardigan	Knitting pattern	1st row: Pearl9, Knit2, ...
Putting together flat pack furniture	Assembly instructions	Screw side panel to front panel
Bisecting an angle with compass and ruler	Drawing instructions	Place the point of the compass on A, and swing an arc ED
Playing music	Musical score	

Table 4.1.2.1 Some fields of human activity in which algorithms are used

Figure 4.1.2.1 shows that the process of knitting a cardigan consists of

- an input - balls of wool of a particular colour and yarn
- an output - the finished cardigan
- a processor - a human
- a precise finite sequence of steps expressed in a code which the processor (in this case a human) can interpret

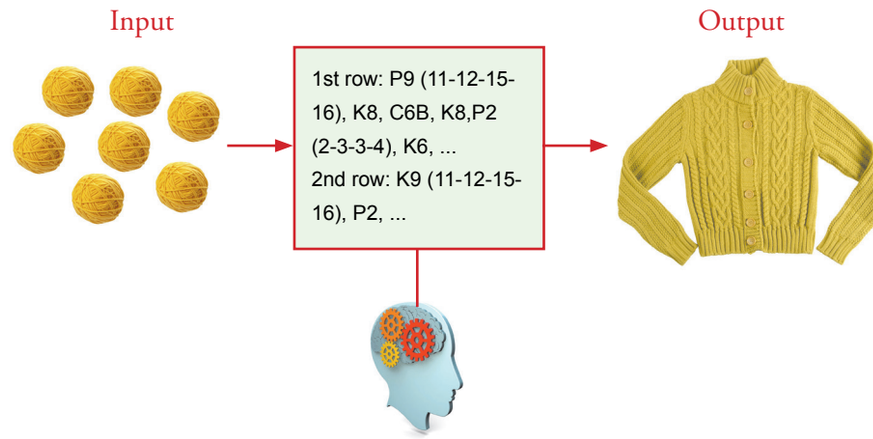


Figure 4.1.2.1 The process of knitting a cardigan

Single licence - Abingdon School

Information

Derivation of term algorithm:
The term algorithm is a corruption of the name al-Khwarizmi, a mathematician of the ninth century (see below), whose book on Hindu numerals is the basis of modern decimal notation. He coined the term *algorism* to describe the rules for performing arithmetic using decimal notation. By the eighteenth century, *algorism* had evolved into the word algorithm. With the advent of machines that could compute, the concept of an algorithm was given a more general meaning to include all definite procedures for solving problems, not just procedures for performing arithmetic.

Abu Ja'Far Mohammed Ibn Musa Al-Khwarizmi (C. 780 -C. 850) an astronomer and mathematician, was a member of the House of Wisdom, an academy of scientists in Baghdad.
He wrote many books. Western Europeans first learned about algebra from his works.
The word algebra comes from al-jabr, part of the title of the book Kitab al-jabr w'al muquabala which he wrote.

Key concept

Algorithm:
A description, independent of any programming language, of a procedure that solves a problem or task.
It consists of a precisely described sequence of steps for solving a problem or completing a task.
The algorithm must terminate and its action must be capable of completing in a finite amount of time.

An algorithm is a method for solving a problem (a task).
It describes an effective path that leads to the problem solution.
Interesting algorithms are ones which involve repeating instructions many times, something which a digital computer is capable of doing.

All but the simplest of algorithms are challenging to design because:

- The description of the algorithm has to be absolutely unambiguous in the sense that different interpretations are excluded
- Each application of the algorithm has to reach the same result
- An algorithm must be designed in a such a way that it works correctly for each of the possible inputs/problem instances and finishes in a finite time
- An algorithm may have many problem instances as possible inputs, too many for it to be possible to test all of them.

Most cooking recipes are not algorithms. They are expressed too imprecisely and rely too much on the knowledge, experience and skill of the cook to be correctly interpreted. A less experienced cook may easily misinterpret the recipe's instructions or worse still not understand some. The range of input is also limited.

Task

1 This task illustrates the difficulty of writing an algorithm which is not open to misinterpretation.
Using pen/pencil and paper execute the following instructions by hand:

1. Draw a diagonal line
2. Draw another diagonal line connected to the top of the first one
3. Draw a straight line from the point where the diagonal lines meet
4. Draw a horizontal line over the straight line
5. At the bottom of the straight line, draw a curvy line
6. Draw a diagonal line from the bottom of the first diagonal to the straight line
7. Draw a diagonal line from the bottom of the second diagonal to the straight line

What object have you drawn?
Compare your result with a friend's.
(CSInside, Algorithm Development, reproduced with kind permission of Professor Quintin Cutts, Glasgow university)

The following constructs are sufficient for constructing algorithms:

- sequence
- assignment
- selection
- iteration

Single licence - Abingdon School

Writing algorithms

Problem

Describe an algorithm for finding the maximum (largest) value in a finite sequence of integers.

Solution

Even though the solution to this problem is very straightforward it provides a good illustration of the concept of an algorithm.

It is possible to express a method for solving this problem in several ways. One method is simply to use the English language to describe the sequence of steps used.

English description of algorithm

The following steps must be performed:

1. *Set the temporary maximum equal to the first integer in the sequence (the temporary maximum will be the largest integer examined at any stage of the procedure)*
2. *Compare the next integer in the sequence to the temporary maximum, if it is larger than the temporary maximum, set the temporary maximum equal to this integer*
3. *Repeat the previous step if there are more integers in the sequence*
4. *Stop when there are no integers left in the sequence (the temporary maximum at this point is the largest integer in the sequence)*

English is notoriously ambiguous, relying on context to resolve ambiguity.
For example, the grammatically correct sentence

Fruit flies like a banana

could refer to fruit such as bananas or alternatively "fruit flies" which are members of the genus Drosophila of small flies.

It is also verbose and too complex to be easily understood by computers.

If you are French then the steps above would be written in French.

One solution is to use a programming language.

Programming language description

An algorithm can be written in a computer programming language.
However, when this is done, only those instructions allowed in the language may be used. The result can be more complicated and difficult to understand than necessary.
Also, there are many computer languages so it would be inappropriate to favour one particular language over another.

Pseudo-code description

Therefore, instead of using a particular programming language to write algorithms, pseudo-code is often used.

Pseudo-code provides an intermediate step between an English language description of an algorithm and an expression of this algorithm in a particular programming language.
In pseudo-code, the steps of the algorithm are expressed using instructions resembling those used in programming languages. However, the writer has the freedom to choose how closely or how loosely the pseudo-code resembles a programming language.

Single licence - Abingdon School

A computer program can then be created, using any computer language, with the pseudo-code description as a starting point.

The principle aim of writing in pseudo-code is to communicate with the reader, a human being, so the syntax should be sufficiently informal to satisfy this aim but it must not lose the ability to be mapped accurately and relatively effortlessly into a program in one of the currently used programming languages.

Table 4.1.2.2 shows a pseudo-code description of the algorithm for finding the maximum element in a finite sequence of *n* integers.

The first step in this algorithm assigns the first integer in the sequence, a_1, to the variable Maximum.

```
Input: a1, a2, a3, a4, a5, ..., an : Integer
Algorithm:
  Maximum ← a1
  For i ← 2 To n
    If Maximum < ai
      Then Maximum ← ai
    EndIf
  EndFor
Output: Maximum is the largest element
```

Table 4.1.2.2 Algorithm expressed in pseudo-code to find the maximum element in a finite sequence of n integers

The *For loop* is used to examine the integers of the sequence, in turn. If the next integer examined is larger than the current value of Maximum, it is assigned to Maximum replacing the current value.

Programming task

1. Write and test a program that implements the algorithm in *Table 4.1.2.2*.

Questions

1. Complete the algorithm shown in *Table 4.1.2.3* which relies on addition to calculate the product of two integers, *x* and *y*. Replace each instance of *?* with an appropriate addition step expressed in pseudo-code.

```
Input: x                [Integer > 0]
       y                [Any integer]
Algorithm:
  Product ← 0
  NoOfTimes  ← 0
  Repeat
     ?
     ?
  Until NoOfTimes = x
Output: product of x and y
```

Table 4.1.2.3 Incomplete algorithm expressed in pseudo-code to calculate the product of x and y

2. Complete the algorithm shown in *Table 4.1.2.4* which calculates integer *x* raised to the power of integer *m* by replacing each instance of *?* with an appropriate step expressed in pseudo-code.

```
Input: m                [Integer > 0]
       x                [Any integer]
Algorithm:
  Power ← x
  Count  ← 1
  While Count < m
     ?
     ?
  EndWhile
Output: x^m
```

Table 4.1.2.4 Incomplete algorithm expressed in pseudo-code to calculate x^m

3. Write an algorithm to find the positive integer m such that $2^m \leq N$ but also such that $2^{m+1} > N$.
   Input is N. Output is m.

4. Write an algorithm to find the sum of the first *n* natural numbers,
   i.e. 1 + 2 + 3 + ... + n.

5. Write an algorithm to find the product of the first *n* natural numbers,
   i.e. 1 x 2 x 3 x ... x n.

Single licence - Abingdon School

Hand-tracing algorithms

Integer division was studied in *Chapter 1.1.3*.

In integer division the quotient q and the remainder r are obtained when dividing an integer x by an integer y.

In integer division, the relationship between x, y, q and r is as follows

$$x = y * q + r$$

For example, when the dividend x = 5, the divisor y = 2, the quotient q = 2, and the remainder r = 1.

To verify this computation, we substitute the given values of x, y, q and r in

$$x = y * q + r$$

and check that the left hand side is equal to the right hand side

$$5 = 2 * 2 + 1$$

Task

2 Complete *Table 4.1.2.5* for x = y ∗ q + r.

Dividend x	Divisor y	Quotient q	Remainder r
5	2	2	1
6	3	2	0
25	4	6	1
36	6		
121	7		
23	3		
1	3		
5	10		

Table 4.1.2.5 Values of x, y, q, r which satisfy x = y ∗ q + r

```
r ← x
q ← 0
While r > y
  r ← r - y
  q ← q + 1
EndWhile
```

Table 4.1.2.6 Algorithm for calculating quotient q and remainder r

A first attempt at writing an algorithm in pseudo-code for calculating q and r is shown in *Table 4.1.2.6*. *Table 4.1.2.7* is a completed trace table which shows the result of executing the algorithm in *Table 4.1.2.6* by hand.

Iteration No	x	y	r	q	r > y
0	5	2	5	0	True
1	5	2	3	1	True
2	5	2	1	2	False

Table 4.1.2.7 Shows a completed trace table for the algorithm in Table 4.1.2.6

Extracting the value for q and r from the last row of *Table 4.1.2.7* and plugging these into x = y ∗ q + r along with dividend x and divisor y, we get confirmation that q and r have been calculated correctly

$$5 = 2 * 2 + 1$$

Task

3 Complete *Table 4.1.2.8* by hand-tracing the algorithm shown in *Table 4.1.2.6*. (Hand-trace = execution by hand).

4 (a) Does your completed table in task 3 demonstrate that

$$x = y * q + r\ ?$$

(b) Is your answer correct, i.e. are q and r correct?

Iteration No	x	y	r	q	r > y
0	6	3	6	0	True

Table 4.1.2.8 Incomplete trace table of a hand trace of the algorithm shown in Table 4.1.2.6

Single licence - Abingdon School

Task

5 Complete *Table 4.1.2.9* by hand-tracing the algorithm shown in *Table 4.1.2.6*.

6 Does the algorithm in *Table 4.1.2.6* terminate when x = 6 and y = 0?

Iteration No	x	y	r	q	r > y
0	6	0	6	0	True

Table 4.1.2.9 Incomplete trace table of a hand trace of the algorithm shown in Table 4.1.2.6

Articulating how a program works

Correctness

It can be concluded from the outcome for *Task 5* and *Task 6*, that before the algorithm is executed, input y to the algorithm must be constrained such that y > 0 is true for the algorithm to terminate and for x = y ∗ q + r to be true after execution of the algorithm - *Table 4.1.2.10*.

```
{y > 0}
r ← x
q ← 0
While r > y
  r ← r - y
  q ← q + 1
EndWhile
{x = y * q + r}
```

Table 4.1.2.10 Require y > 0 to be true at beginning, and x = y ∗ q + r true at end

In *Task 4*, x = 6, y = 3, q = 1 and r = 3 makes x = y ∗ q + r true but something is wrong in the algorithm because q should be 2.

Table 4.1.2.11 shows a trace of the values of x, y, q, r.

```
{y > 0}              { x = 6, y = 3 }
r ← x                { r = 6 }
q ← 0                { q = 0 }
While r > y
  r ← r - y          { r = 3 }
  q ← q + 1          { q = 1 }
EndWhile
{x = y * q + r}      { x = 6, y = 3, q = 1, r = 3 }
```

Table 4.1.2.11 Trace of the values of x, y, q, r

Close inspection of the algorithm reveals that the *While condition* is incorrect - *Table 4.1.2.12*.

It should be r >= y, not r > y.

```
{y > 0}
r ← x                    r >= y
q ← 0
While   r > y
Do
  r ← r - y
  q ← q + 1
EndWhile
{x = y * q + r and r < y}
```

Table 4.1.2.12 Error in the algorithm revealed

Table 4.1.2.13 shows the corrected algorithm.

The *While loop* is now exited when r < y.

We can now assert that if *y > 0* is true before the algorithm is entered then it will terminate with *x = y * q + r and r < y* true.

The quotient q is now calculated correctly, i.e. q = 2 when x = 6, y = 3, q = 1 and r = 3.

Also x = y ∗ q + r is true and r < y is true.

```
{y > 0}
r ← x
q ← 0
While r >= y
  r ← r - y
  q ← q + 1
EndWhile
{x = y * q + r and r < y}
```

Table 4.1.2.13 Corrected algorithm

Single licence - Abingdon School

Task

7 Complete *Table 4.1.2.14* by hand tracing the algorithm shown in *Table 4.1.2.13*.

8 Is x = y * q + r for the values recorded in *Table 4.1.2.14*?

9 Is negative remainder r allowed?

Iteration No	x	y	r	q	r >= y
0	-2	1			

Table 4.1.2.14 Incomplete trace table of a hand trace of the algorithm shown in Table 4.1.2.13

Task 7 calculates a negative remainder. A negative remainder is impossible! - see *Figure 4.1.2.2*.

We must therefore constrain input x to be greater than or equal to zero to avoid a negative remainder.

Table 4.1.2.15 shows the correct input constraints to make condition $\{x = y * q + r \text{ and } 0 \le r < y\}$ true after execution of algorithm. The algorithm will now terminate with the value of q and r calculated correctly.

```
{x ≥ 0 and y > 0}
r ← x
q ← 0
While r >= y Do
  r ← r - y
  q ← q + 1
EndWhile
{x = y * q + r and 0 ≤ r < y}
```

Table 4.1.2.15 Correct input constraints to make condition $\{x = y * q + r \text{ and } 0 \le r < y\}$ at end of algorithm true

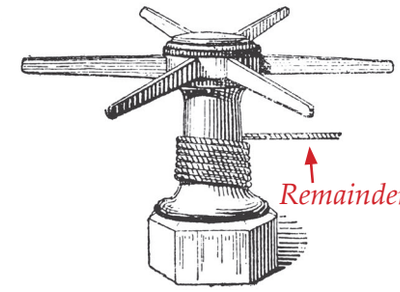


Figure 4.1.2.2 Rope remainder after winding rope on capstan

Programming task

2 Write and test a program that implements the algorithm expressed in *Table 4.1.2.15*.

Key concept

Specifying the range of inputs of an algorithm:
The range of inputs are specified as a pre-condition of executing the algorithm.
Pre-condition:
A pre-condition specifies the state of the computation that must be true before the algorithm is executed.
For example,
$\{x \ge 0 \text{ and } y > 0\}$

Key concept

Specifying the desired effect of an algorithm:
The desired effect of executing an algorithm can be specified as a post-condition that must be true after executing the algorithm.
Post-condition:
A post-condition specifies the state of the computation after the algorithm is executed.
For example,
$\{x = y * q + r \text{ and } 0 \le r < y\}$
must be true.

Efficiency

Given an integer *n*, compute its prime factorisation[1].
For example, if *n* is 60, its prime factorisation is 2 x 2 x 3 x 5.

Table 4.2.1.16 shows an algorithm expressed in pseudo-code for finding the prime factors of *n*.
The algorithm checks if *i* is a factor. If it is, it outputs this factor before dividing it out of *n*.

Table 4.1.2.17 shows another algorithm expressed in pseudo-code for finding the prime factors of *n* in fewer steps (the for loop will repeat fewer times) than the algorithm shown in *Table 4.1.2.16*.

1 Prime factorization of a number is the determination of the set of prime numbers which multiply together to give the original integer.

Single licence - Abingdon School

Input: n [Integer ≥ 1]

Algorithm:

```
For i ← 2 To n Do
   While n MOD i = 0
     Output i
     n ← n DIV i
   EndWhile
EndFor
```

Output: prime factors of n

Table 4.1.2.16 Algorithm expressed in pseudo-code to find prime factors of n

Programming task

3 Write and test a program that implements the algorithm expressed in *Table 4.1.2.16*.

If *n* has a factor then this must be one less than or equal to the square root of *n*.

For example, suppose n = 9. Its square root is 3. The first factor that is found is 3, the second is 3. The *for loop* in the algorithm in *Table 4.1.2.17* finds and outputs the prime factor 3 when i = 3. Next *n* is divided by *i* and the result 3 is assigned to *n*. The *for loop* now exits with n = 3.
The next step *If n > 1* is true so 3 is output. The two prime factors of 9 have now been output.

Now suppose n = 10. Its square root is 3.1623 to 4 decimal places. This becomes 3 after applying the operation *Floor*.

Input: n [Integer > 1]

Algorithm:

```
SqrtOfn ← Floor(Sqrt(n))
For i ← 2 To SqrtOfn
  While (n MOD i) = 0
    Output i
    n ← n DIV i
  EndWhile
EndFor
If n > 1
  Then Output n
EndIf
```

Output: prime factors of n

Table 4.1.2.17 Algorithm expressed in pseudo-code to find prime factors of n using fewer iterations of for loop than algorithm in Table 4.1.2.16

The first factor that is found is 2, the second is 5. The *for loop* in the algorithm in *Table 4.1.2.17* finds and outputs the prime factor 2.
It then divides *n* by *i* and assigns the result 5 to *n*.
The next step *If n > 1* is true so 5 is output.

When using this method on a number *n*, only divisors up to $\lfloor \text{sqrt}(n) \rfloor$ (where $\lfloor x \rfloor$ is the *floor* function) need be tested.

This algorithm when programmed in a programming language should therefore consume less execution time than the program equivalent of the algorithm in *Table 4.1.2.16*.

Information

Floor function:
The floor function returns the largest (closest to positive infinity) value that is less than or equal to the argument and is equal to a mathematical integer.

We say that this algorithm is more **time-efficient** than the algorithm in *Table 4.1.2.16*.

Programming task

4 Write and test a program that implements the algorithm expressed in *Table 4.1.2.17*.

Single licence - Abingdon School

In this chapter you have covered:

- The term algorithm
- Expressing the solution to a simple problem as an algorithm using pseudo-code, with the standard constructs:
  - sequence
  - assignment
  - selection
  - iteration
- Hand-tracing algorithms
- Converting an algorithm from pseudo-code into high level language program code
- Articulating how a program works, arguing for its correctness and its efficiency using logical reasoning, test data and user feedback

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

4.1.3 Abstraction

Learning objectives:

- *Be familiar with the concept of abstraction as used in computations and know that:*
  - *representational abstraction is an abstraction arrived at by removing unnecessary details*
  - *abstraction by generalisation or categorisation is a grouping by common characteristics to arrive at a hierarchical relationship of the 'is a kind of' type.*

Key term

Representational abstraction: Means omitting unnecessary details.

What is abstraction?

The human brain is an exceptional piece of biological machinery capable of recognising objects even when most of their detail has been removed.

This is illustrated in *Figure 4.1.3.1* which is an abstraction of an image of a child. Despite most of the details being removed from the image, it is still very easy for us to recognise the image of a child.

Figure 4.1.3.1 Silhouette image of a child

Representational abstraction

Humans deal with abstractions all the time, because they are useful in everyday problem solving.

For example, travelling from Marylebone to Russell Square by London Underground involves taking the Bakerloo line from Marylebone as far as Piccadilly Circus then changing to the Piccadilly line and travelling as far as Russell Square (*Figure 4.1.3.2*). It is not necessary to include more details than these to succeed in reaching the desired destination. In fact, this example of Harry Beck's London Underground map is itself an abstraction. Unnecessary details have been removed (like the names of nearby roads) and the layout of the stations adjusted to make the map easy to use. Ease of use is a very good reason to work with an abstraction rather than the real thing. Abstraction means **omitting unnecessary details**. What we end up with when unnecessary details are removed is an example of **representational abstraction**.

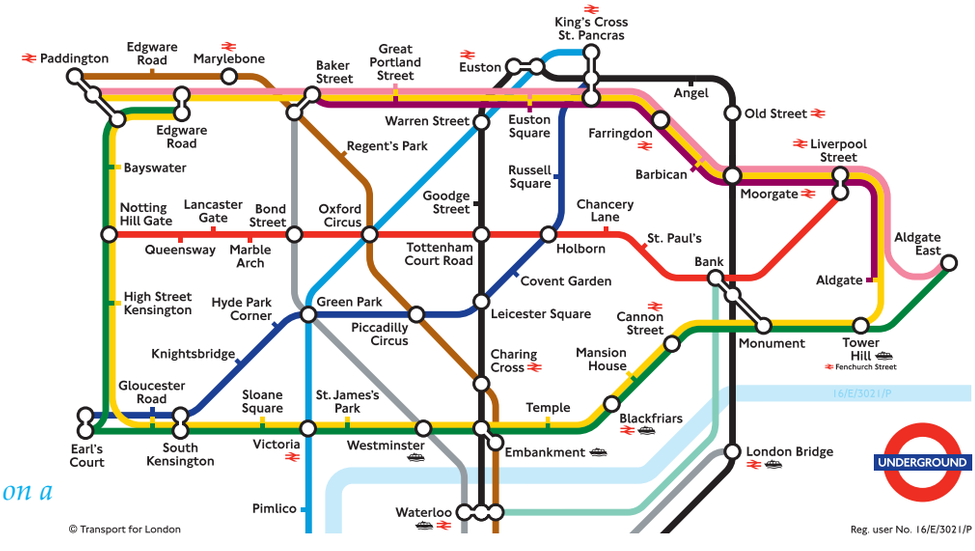


Figure 4.1.3.2 London Underground map based on a design by Harry Beck

Single licence - Abingdon School

Abstraction as used in computations

Modelling

Ease of use is a very good reason to work with an abstraction rather than the real thing. The London Tube map is a **model** of the real underground system. Models are used to understand the world by removing complexity and focussing on details relevant to the task in hand. The **process of abstraction** allows problems to be solved by separating the logical and physical views. To navigate the London Underground, we just need information consisting of stations, their links and the classification of links, e.g. Marylebone is connected directly to Oxford Circus by the Bakerloo line. This is the **logical view**. To take in the **physical view** requires going to these stations and travelling on the underground.

Data modelling

Figure 4.1.3.3 shows parking bays in a car park. These could be modelled by an abstraction known as an **array** as shown in *Figure 4.1.3.4*. An array is a logical element. It has a meaning which crops up in everyday life, e.g. an array of solar panels as shown in *Figure 4.1.3.5*.

Figure 4.1.3.3 Car park with bays arranged in a rectangular grid

Figure 4.1.3.5 Solar panel arrays

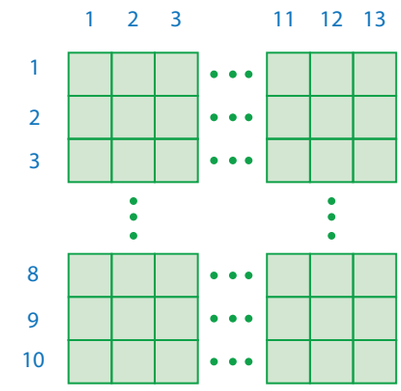


Figure 4.1.3.4 Modelling car park as a two-dimensional array

A programmer would find it more convenient to work with the two-dimensional array model of the car park than to work directly with the electronic units from which the memory of a computer is built. This separation of the physical view of the computer's memory from the programmer's logical view of an array is an example of (representational) abstraction.

Layers of abstraction

The complexities of the hardware of a computer make it difficult for users to interact directly with the hardware. The solution is to progressively hide these complexities by layers of abstraction as shown in *Figure 4.1.3.6*. The BIOS (Basic Input Output System) which resides in firmware interfaces directly with the layer below, the hardware, and the operating system software layer above the BIOS. The operating system is multilayered. The user interacts with the user interface layer of the operating system which in turn interacts with the operating system layer immediately below. A computation submitted to the computer through the user interface is carried out by the user interface layer calling on services from the layer immediately below which in turn does something similar with the layer below it and so on. Eventually, the computation reaches the

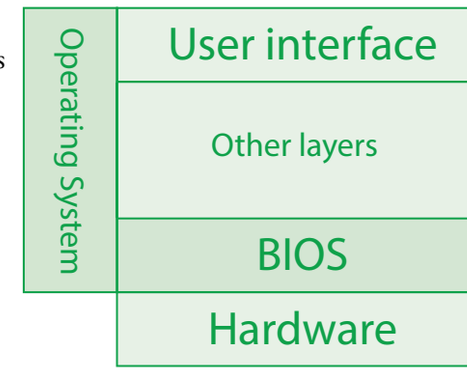


Figure 4.1.3.6 Layers of abstraction

Single licence - Abingdon School

hardware where it is executed. The result of the computation is returned up through the layers until it reaches the user interface where it is presented to the user. This distinguishes abstraction in Computer Science from abstraction in mathematics. Mathematics only ever deals with one layer of abstraction, e.g. 5 x 6 + 4 is abstracted to a x b + c. Computer Science deals with two layers at a time: the layer above which calls on services from the layer below.

Abstraction and subprograms

Figure 4.1.3.7(a) shows pseudo-code for a method that swaps the contents of variables `x` and `y` using a temporary variable `temp` in the process. We are able to abstract away the detail of this method by creating a subprogram `swap(a, b)` shown in *Figure 4.1.3.7(b)* whose body contains this pseudo-code. To swap the contents of two variables `x` and `y` we would substitute `x` and `y` for the formal parameters `a` and `b` and call subprogram `swap(x, y)`.

```
temp ← x
x ← y
y ← temp
```

Figure 4.1.3.7(a) Pseudo-code to swap the contents of two variables x, y

```
swap(a, b)
```

Figure 4.1.3.7(b) Subprogram to swap the contents of two variables, a, b

Generalisation

Figure 4.1.3.8 illustrates what is meant by abstraction by **generalisation** or **classification** (by category).

Key term

Abstraction by generalisation: Means grouping by common characteristics to arrive at a hierarchical relationship of the 'is a kind of' type.

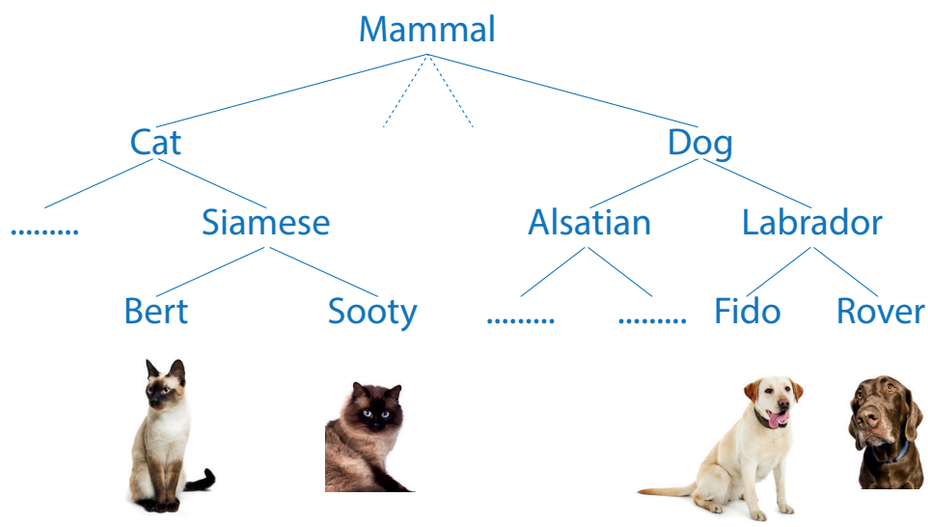


Figure 4.1.3.8 An example of abstraction by generalisation

Rising up the hierarchy, we go from specific examples of mammals to categories they belong to.

The relationship is classified as '*is a kind of*'. For example, Rover is a kind of Labrador, which is a kind of dog, etc.

All Labrador dogs have characteristics in common. Similarly, all types of dog have dog characteristics in common.

This grouping together by identifying common characteristics is a feature of generalisation.

Figure 4.1.3.9 shows how the dog part of this hierarchy could be modelled with arrays.

Table 4.1.3.1 shows the data structures that could be used to model the dog part of the hierarchy.

Single licence - Abingdon School

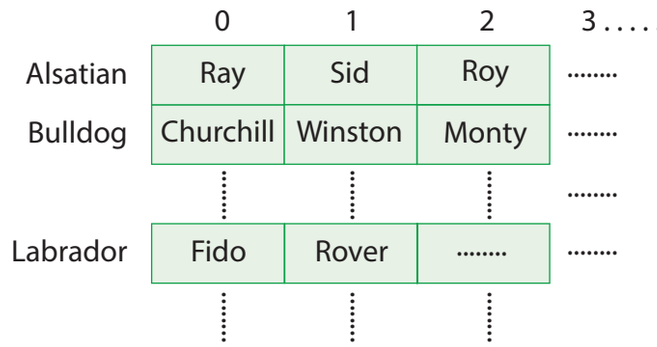


Figure 4.1.3.9 An array of one-dimensional arrays

Data structure	Variable
`TDogArray = Array[0..100] Of TString`	
`TDogBreedArray = Array[TDogBreed] Of TDogArray`	`DogBreedArray : TDogBreedArray`

Table 4.1.3.1 Data structures for modelling the Dog part of the hierarchy in Figure 4.1.3.7

The Array `DogBreedArray` could itself be an array element of a `MammalArray`.

These data structures use a user-defined data type

```
TDogBreed = (Alsation, Bulldog, ...., Labrador, ....)
```

This is an enumerated data type which enables meaningful names Alsation, Bulldog, ... to be used in place of 0, 1, 2, 3, ...

Figure 4.1.3.10 shows assignment of the names of some dogs to array `DogBreedArray`.

```
DogBreedArray[Alsation] [0]  ← 'Ray'
DogBreedArray[Alsation] [1]  ← 'Sid'
DogBreedArray[Alsation] [2]  ← 'Roy'
DogBreedArray[Labrador] [0]  ← 'Fido'
DogBreedArray[Labrador] [1]  ← 'Rover'
```

Figure 4.1.3.10 Assigning names of dogs to array DogBreedArray

Question

1. What is meant by
   (a) representational abstraction
   (b) abstraction by generalisation?

2. Why is it useful to work with an abstraction of reality when problem solving?

In this chapter you have covered:

- The concept of abstraction as used in computations:
  - representational abstraction - an abstraction arrived at by removing unnecessary details
  - abstraction by generalisation or categorisation - a grouping by common characteristics that produces a hierarchical relationship of the 'is a kind of' type.

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Information hiding:*

  Be familiar with the process of hiding all details of an object that do not contribute to its essential characteristics.

Key concept

Information hiding:
Information hiding means hiding design details behind a standard interface.

4.1.4 Information hiding

Modules

Large programs, or systems, benefit from being decomposed into modules. Modules can be separately-compiled units consisting of several subroutines (procedures and functions) or they can be a single subroutine (which itself can be composed of other subroutines nested inside it).

Decomposing a software system into separate modules is considered the key to good design. However, there are many ways that a software system can be decomposed into modules. It is the job of design to decide how exactly the system should be decomposed.

Several guiding design principles inform the process of achieving good design:

- Each task that a software system or program is required to carry out should form a separate, distinct program module
- Each module and its inputs and outputs should be well-defined so it is clear to other modules how to use the module
- Each module should know as little as possible about how the other modules are designed internally.

When the design for the software system follows these principles, it should be possible to

- work independently on each module
- change the internal design of a module without affecting other modules which rely on the changed module, i.e. the other modules will not require changing as well
- understand how a particular module works without needing to know how other modules work internally
- test modules independently of other modules
- trace errors of the whole system or limitations of the whole system to individual modules thus limiting the scope of searching for errors.

Information hiding

Following the guiding design principles mentioned above means applying the principle of information hiding.

This means that each module should know as little as possible about how other modules with which it interacts are designed internally. The temptation to design a module using knowledge of some aspect of the internal design of

Single licence - Abingdon School

Information

Information hiding:
David Parnas first introduced the concept of information hiding around 1972 in a seminal paper 'On the Criteria to Be Used in Decomposing Systems into Modules'.
There are many ways to decompose a system into modules. A guiding principle is to minimise the number of "connections" between modules. This means that each module should know as little as possible about how other modules it interacts with are designed internally. The temptation then to design this module using knowledge of some aspect of the internal design of another module is minimised.
Adopting this design principle means that modules can only interact through well-defined interfaces. Hiding information (design information) in this manner:

- isolates other modules from the effects of changing the internal design decisions of a module.
- allows modules to be worked on independently of other modules
- means an intimate knowledge of the design of a module is not needed in order to use it.

another module is minimised if this is the case. Modules then can interact only through well-defined interfaces. Also, modules can be worked on independently of other modules.

Module interface(s)

Interfaces hide the complexities of a module and force interaction to take place only through an interface as illustrated in *Figure 4.1.4.1*. **Information hiding** means hiding design details behind a standard interface.

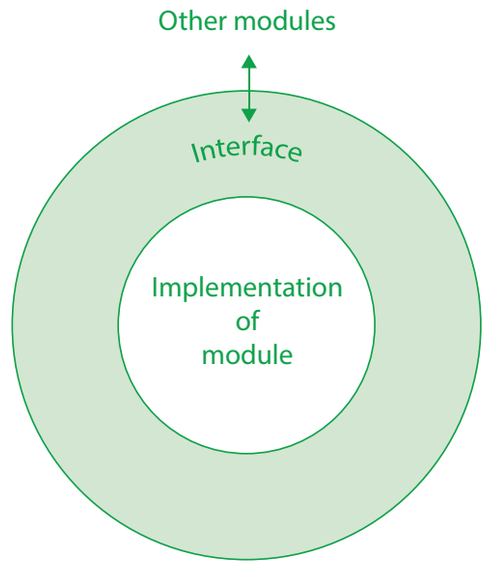


Figure 4.1.4.1 Hiding the complexity of a module behind an interface

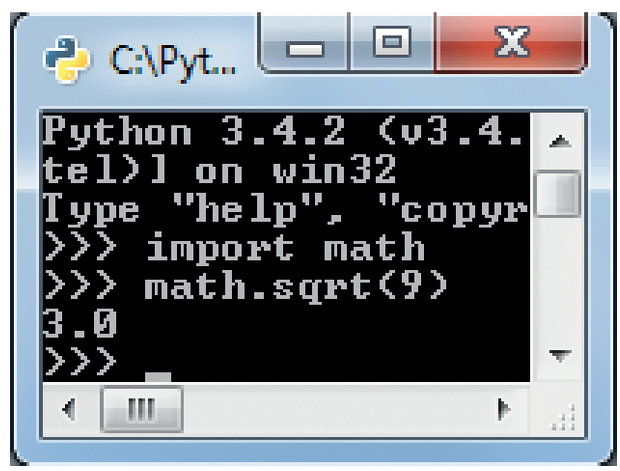


Figure 4.1.4.2 Importing the Python Math module and applying the sqrt function to argument 9

Figure 4.1.4.2 shows a Python example which imports a Python module *math*. Module *math* implements several mathematical routines each of which has its own interface.

The example uses one of these mathematical routines called *sqrt* and applies it to the argument 9. This routine calculates the positive square root of 9 without revealing how it does it. This is because Python hides the implementation of the mathematical routines in module *math* behind interfaces. If we only need

Single licence - Abingdon School

to calculate the positive square root of a number, we don't need to know how the square root is being calculated. This is a form of abstraction known as functional abstraction.

The routines then are known as functions. To use mathematical routines in module *math* we need to know

- the name of each routine, e.g. *sqrt*
- the argument(s) that needs to be supplied
- what will be returned.

In the case of a module such as math, we call these three things in each case, the **interface** to the module. *Figure 4.1.4.3* shows the interface to Python's math module which is made up of maths functions, a sample of which are shown in the figure.

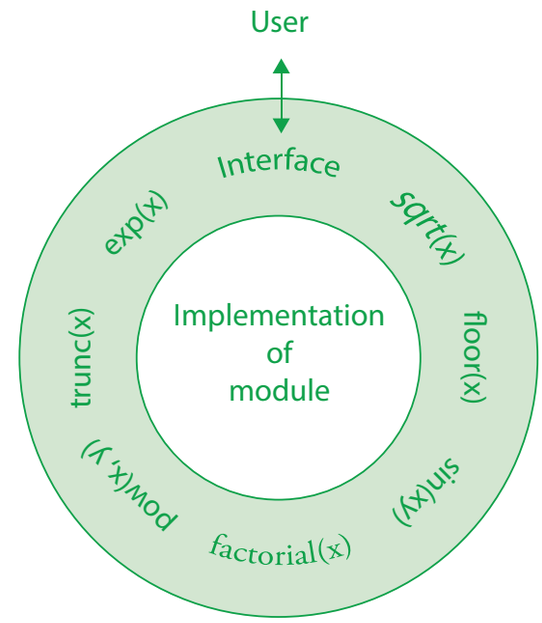


Figure 4.1.4.3 Some functions available in the Python Math module

Question

1 The modular design of a modern motor car consists among other things of the following major units:

An internal combustion engine

- a gear box
- a drive shaft
- wheels that are connected to the drive shaft

The engine is connected to the gear box which is connected to the drive shaft which in turn is connected to the wheels and which drives the wheels.

A driver controls the driving of the car through the following:

- a steering wheel
- a gear stick to select a gear, low gears for climbing uphill or pulling away and higher gears for cruising
- an accelerator pedal with which to accelerate the car
- a brake pedal with which to slow and stop the car
- various instruments including a speedometer which indicates the speed of the car.

Explain the design principle of information hiding using the design of a motor car. In your answer, you should consider whether the following benefits of information hiding are achieved in the design of a motor car and if they are, exactly how:

- isolates other modules from the effects of changing the internal design decisions of a module.
- an intimate knowledge of the design of a module is not needed in order to use it
- allows modules to be worked on independently of other modules.

Single licence - Abingdon School

Subroutine interface

The design principles embedded in the principle of information hiding were arrived at by studying how large scale software systems were developed. Nevertheless, these design principles can be usefully applied to even small scale projects. *Figure 4.1.4.4* shows a very simple problem broken down into three modules which are actually the subroutines *Getn*, *CalculateSum* and *DisplayResult*. The problem posed is how to "*sum the natural numbers from 1 to n*", for example, if n = 6, Sum = 1 + 2 + 3 + 4 + 5 + 6 = 21.

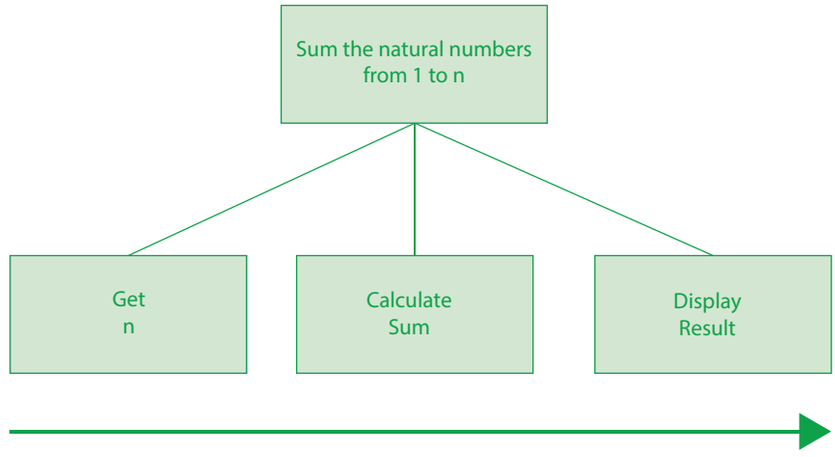


Figure 4.1.4.4 Hierarchy chart for Sum natural numbers from 1 to n with an arrow indicating the order of execution

We view the subroutine *CalculateSum* as a black box with one input and one output as shown in *Figure 4.1.4.5*. In the case, we call the input and the output the **interface** because the module is a single subroutine.

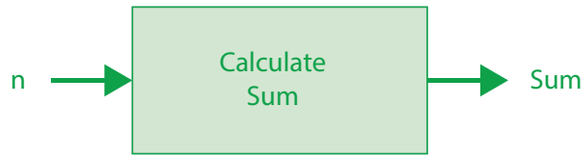


Figure 4.1.4.5 Subroutine CalculateSum as a black box

Figure 4.1.4.6 Subroutine DisplayResult as a black box

Figure 4.1.4.7 Subroutine Getn as a black box

Figure 4.1.4.6 shows subroutine *DisplayResult* as a black box. It accepts as input the value of *Sum* which it then displays on the console. In this case, we call the input the **interface** because the module is a single subroutine

Figure 4.1.4.7 shows subroutine *Getn* as a black box. It outputs the value of *n* which it has read from the keyboard. In this case, we call the output the **interface** because the module is a single subroutine.

The implementation details of each subroutine is hidden when we view each as a black box. This means that we should be able to substitute subroutine *CalculateSum* by one which calculates the sum in a different way, but without affecting how subroutine *CalculateSum* is used in the program as long as its interface is the same.

Task

1. Using a programming language of your choice, implement a program that uses the three subroutines *Getn*, *CalculateSum*, *DisplayResult* to sum the natural numbers from 1 to *n*. For example, n = 6, Sum = 1 + 2 + 3 + 4 + 5 + 6 = 21. You should use Gauss' method of summing natural numbers in *CalculateSum* (see *Chapter 5.1.1*).
2. Now without altering *CalculateSum*'s interface replace Gauss's method by the brute force method of summing natural numbers in *CalculateSum*, i.e. Sum = 1 + 2 + 3 + 4 + 5 + 6 + ...

Single licence - Abingdon School

Data encapsulation

We may hide the implementation of a queue behind an interface (*Figure 4.1.4.8*) and restrict access to the queue to the following subroutines:

- AddItem(x)
- RemoveItem
- IsEmpty
- IsFull
- Flush

This is called **data encapsulation**, another form of information hiding.

The queue could be implemented as an array capable of storing up to 100 items but because the implementation is hidden, modules which use the queue module would not be given this information.

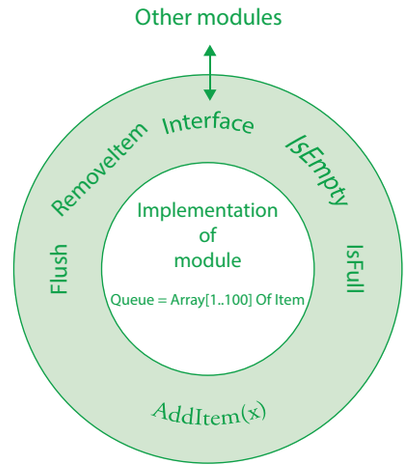


Figure 4.1.4.8 Interface for the queue

They would interact with the queue through the module's interface routines. The implementation of these would also be hidden.

Hiding all details of an object that do not contribute to its essential characteristics

The Python *math* module, the modules *Getn*, *CalculateSum*, *DisplayResult*, and the *queue* module are all objects that hide details that are not essential for other modules which use this module to know. These are examples of information hiding by which design details are hidden behind a standard interface.

Information

Queue:
A queue is a first in first out data structure for storing data items. Data items are removed in turn from the front of the queue. Data items join the queue at the rear in the order of their arrival.

Question

2 Explain with examples drawn from software systems what is meant by information hiding.

In this chapter you have covered:

- Information hiding:
  - the process of hiding all details of an object that do not contribute to its essential characteristics.

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Know that procedural abstraction represents a computational method.*

4.1.5 Procedural abstraction

Computations

The arithmetic expression 3 × 4 + 2 represents a **computation** which can be described with reductions as follows

$$3 \times 4 + 2 \rightarrow 12 + 2 \rightarrow 14$$

The given **expression** describes a **single computation**.

Formula

A **formula** is an expression containing variables. It represents a **whole class of computations**, since the variable(s) can assume different values.

For example, the formula corresponding to the computation above

`a x b + c`

uses the variables `a, b, c`.

Substituting integers for the variables, `a, b, c`, produces an expression whose value can be computed. For example, a = 3, b = 4, c = 2

$$3 \times 4 + 2 \rightarrow 12 + 2 \rightarrow 14$$

The formula `a x b + c` describes one computation for each possible combination of values for `a, b, c`.

Since there are an infinite number of integers, the formula represents an infinite number of computations.

Procedural abstraction

The formula `a x b + c` is an abstraction because it omits the actual numbers to be used in the computation. The formula `a x b + c` represents a computational method, a **procedure**. Such an abstraction is called a **procedural abstraction**, since the result of the abstraction is a procedure, a method.

In general, there are many methods for obtaining a desired result.

The result of a procedural abstraction is a procedure.

For example, suppose we wished to calculate the `sum` of the first `n` natural numbers. We have two choices of method or formula for calculating this sum:

$$\frac{n \times (n + 1)}{2}$$

and

$$1 + 2 + 3 + 4 + \ldots + (n - 1) + n$$

The above two methods are examples of procedural abstraction.

Key term

Computation:
An arithmetic expression such as

3 × 4 + 2

describes a single computation.

Key term

Formula:
A formula is an expression containing variables. It represents a whole class of computations, since the variable(s) can assume different values, e.g.

`a x b + c`

Key term

Procedural abstraction:
The formula a x b + c represents a computational method, a procedure. Such an abstraction is called a procedural abstraction, since the result of the abstraction is a procedure, or method.

Single licence - Abingdon School

Question

1 Using words and phrases such as computation, formula, and computational method explain procedural abstraction.

In this chapter you have covered:

- Procedural abstraction which represents a computational method.

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Know that functional abstraction the particular computation method is hidden.*

4.1.6 Functional abstraction

Functional abstraction

The result of a procedural abstraction is a procedure, not a function. To get a **function** another abstraction, which disregards the particular computation method, must be performed.

The result of this abstraction is a function and the abstraction is called **functional abstraction**. The focus is then on the input(s) and the output.

For example, suppose we wished to calculate the `sum` of the first `n` natural numbers. We have two choices of method or formula for calculating this sum:

$$\frac{n \times (n + 1)}{2}$$

and

$$1 + 2 + 3 + 4 + \ldots + (n - 1) + n$$

The above two methods are examples of procedural abstraction.

The black box in *Figure 4.1.6.1* hides the particular method used to calculate the `sum` for a particular `n` and so is an example of **functional abstraction**.

Figure 4.1.6.1 Calculation of sum of first n natural numbers

All that a user needs to know are the number and order of the inputs and the name of the function in order to be able to **apply the function to these inputs**.

Key term

Functional abstraction: By disregarding the particular computation method we get an abstraction known as a function. By knowing the number and order of the inputs and the name of the function we are able to apply the function to these inputs.

Parameters

The formula used to define a function is sometimes called the **body of the function**. The name used for the quantity that can vary is called the **parameter of the function**. Therefore in the example above `n` is the parameter.

The function associated with the formula $\frac{n \times (n + 1)}{2}$ is given a name, e.g. `sum`.

To use the sum function, we apply it to an argument.

For instance, to find the sum of the first 6 natural numbers, the function `sum` is applied to the argument 6.

Thus the parameter, `n`, is the name used in the function body to refer to the argument, 6.

Single licence - Abingdon School

To compute the value of the function for some argument, replace the parameter in the body of the function by the argument and compute the expression.

For example,

$$n = 6 \longrightarrow \frac{n \times (n + 1)}{2}$$

$$= \frac{6 \times (6 + 1)}{2}$$

$$= 21$$

Question

1 What is functional abstraction?

In this chapter you have covered:

- Functional abstraction which hides the particular computation method.

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Know that details of how data are actually represented are hidden, allowing new kinds of data objects to be constructed from previously defined data types*

4.1.7 Data abstraction

Isolating the use of a compound data object from its construction details

Creating a compound object

Learning how to program is a process of developing the ability to model problems in such a way that a computer can solve them.

Suppose that we want to model a queue at an ATM such as shown in *Figure 4.1.7.1*. People join the queue at the rear and leave from the front to use the ATM to withdraw cash.

Figure 4.1.7.1 Queue of people at an ATM

To model the queue, we first ignore all but the essential details of each person which are necessary to solve the problem. The problem might be to know how much money to load the ATM with at the beginning of each day.

We reduce each person to a single uniform object illustrated in *Figure 4.1.7.2*. We have abstracted away unnecessary details of a person.

Figure 4.1.7.2 Queue of people with unnecessary details removed

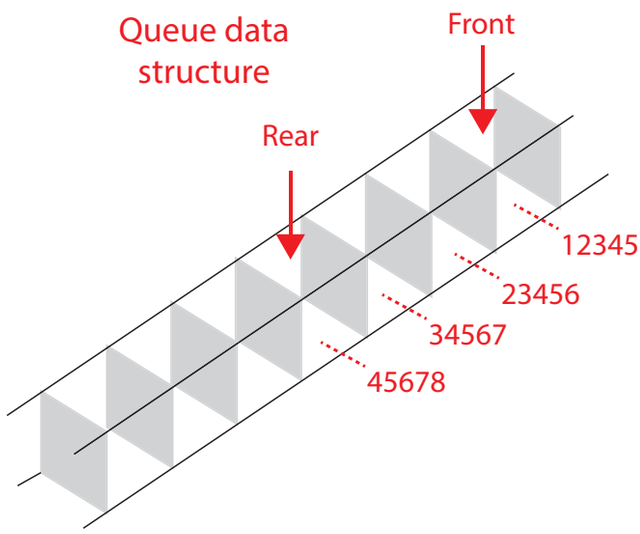


Figure 4.1.7.3 Queue data structure

The next stage models a person joining the queue by adding their values for these essential details (e.g. card number, card expiry date, name on card) to the queue. We use a data structure to store these details as illustrated in *Figure 4.1.7.3*.

The data structure must have storage space for the essential details of each person currently in the queue and for persons who join the queue subsequently.

It is a **compound object** because it is constructed to store the details of more than one person, e.g. `Person1` with card number '12345', `Person2` with card number '23456', etc.

Single licence - Abingdon School

Using the queue compound object

We could use a subroutine, `AddItem(Person)` to add a person's details to the rear of the queue, passing the person's essential details in the variable `Person`.

To model a person leaving the queue to use the ATM, we could use a subroutine `RemoveItem` which returns the person's essential details.

This would remove the person's essential details from the front of the queue.

Of course, we need to apply constraints to this model:

1. A person may only join the queue at the rear
2. A person may only leave the queue from its front
3. A person cannot join a full queue[1]
4. A person cannot be removed from an empty queue

Therefore, we need subroutines to test whether the queue is full or not and to test whether the queue is empty or not:

- `IsFull`
- `IsEmpty`

Finally we need to be able to clear the queue so that we can start with an empty queue. For this we use a subroutine `Flush`.

Figure 4.1.7.4 shows a Pascal program which uses a queue. However, the details of the queue's construction are hidden in a module `HiddenQueueDetails`.

```
Program QueueExample;
{$APPTYPE CONSOLE}
Uses HiddenQueueDetails;
Var
  Person1 : String = '12345';
  Person2 : String = '23456';
  Person3 : String = '34567';
  Person4 : String = '45678';
Begin
  Flush;
  AddItem(Person1);
  AddItem(Person2);
  AddItem(Person3);
  If Not IsEmpty
    Then Writeln(RemoveItem)
    Else Writeln('Queue Empty');
  Readln;
End.
```

Figure 4.1.7.4 Pascal program which uses a queue

The program can only access the queue by calling subroutines defined in the module `HiddenQueueDetails` and made available through the module's interface - see *Figure 4.1.7.5* and *Figure 4.1.7.6*.

The module is also written in Pascal but it is only available to the program in object code form so that how it has been constructed is hidden.

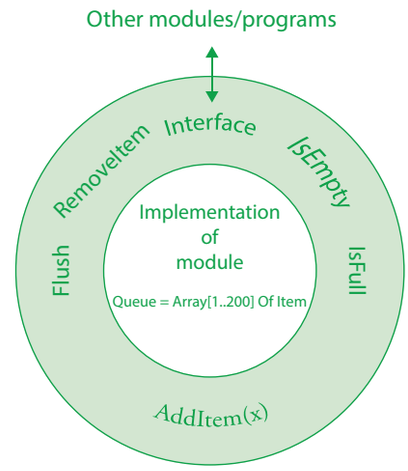


Figure 4.1.7.5 Interface to the implementation of a queue of 200 items

Constructing and hiding the queue compound object

In module `HiddenQueueDetails`, the queue is constructed from an array of 200 cells. The cell data type is string in this example. A more realistically application of queues might use a record type.

The module/unit implements a circular queue (*Figure 4.1.7.7*).

In *Figure 4.1.7.6* the only part of the module/unit which is exposed to other modules is placed in the *Interface* section.

The hidden part of the module/unit is placed in the *Implementation* section.

This includes the `type` definition `TQueue = Array[1..200] Of String` and the variable declaration

```
Var Queue : TQueue; Front, Rear : Integer;
```

`Front` and `Rear` are queue pointers.

1 There is a waiting area and people are not allowed to wait for the ATM unless they are standing in the waiting area and the waiting area can only hold X people.

Single licence - Abingdon School

```
Unit HiddenQueueDetails;
Interface
  Procedure Flush;
  Procedure AddItem (Item : String);
  Function RemoveItem : String;
  Function IsEmpty : Boolean;
  Function IsFull : Boolean;
Implementation
  Uses System.Contnrs;
  Type
    TQueue = Array[1..200] Of String;
  Var
    Queue : TQueue;
    Front, Rear : Integer;
  Procedure Flush;
    Begin
      Front := 0;
      Rear := 0;
    End;
  Procedure AddItem (Item : String);
    Begin
      If (Front=0) And (Rear=0)
        Then
          Begin
            Front := 1;
            Rear := 1;
            Queue[Rear] := Item;
         End
        Else
          Begin
            If (Rear Mod 200) + 1 = Front
              Then Writeln('Queue Full!')
              Else
                Begin
                  Rear := (Rear Mod 200) + 1;
                  Queue[Rear] := Item;
                End;
          End;
    End;
  Function RemoveItem : String;
    Begin
      If (Front=0) And (Rear=0)
        Then Writeln('Queue Empty!')
        Else
          Begin
            RemoveItem := Queue[Front];
            If Front=Rear
              Then
                Begin
                  Front := 0;
                  Rear := 0;
                End
              Else Front := (Front Mod 200) + 1;
          End;
    End;
  Function IsEmpty : Boolean;
    Begin
      If Front = 0
        Then IsEmpty := True
        Else IsEmpty := False;
    End;
  Function IsFull : Boolean;
    Begin
      If (Rear Mod 200) + 1 = Front
        Then IsFull := True
        Else IsFull := False;
    End;
End.
```

The only part of this unit which is exposed to other modules/programs is placed in the Interface section.

The hidden part of this unit is placed in the Implementation section.

Figure 4.1.7.6 Pascal module which creates a queue of 200 items by declaring an array of 200 cells, with each designed to store a string.

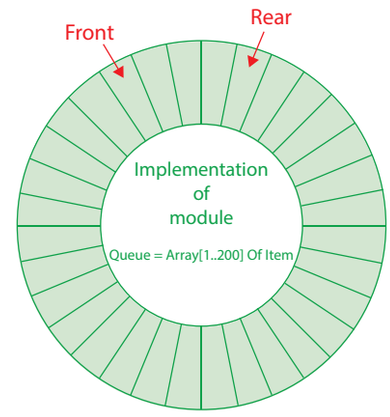


Figure 4.1.7.7 The unit implements a circular queue

Key point

Data abstraction:
Data abstraction is a methodology that enables us to isolate how a data object is used from the details of how it is constructed.

Information

Keyboard buffers are usually implemented as circular buffers. The buffering allows typing ahead. and lazy I/O. Lazy I/O means that a line typed at the keyboard is not processed until the return key is pressed. Until this happens the line may be changed.

Single licence - Abingdon School

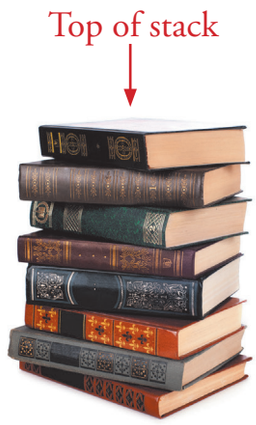


Figure 4.1.7.8 A stack of books

Questions

1. What is meant by *data abstraction*?

2. *Figure 4.1.7.8* shows a stack of books. Books are added and removed from a stack at one end only which is called the top of stack.
   (a) Suggest one way that the stack of books could be modelled in a computer.
   (b) What operations might be needed for this stack?
   (c) Explain how you would isolate the compound data object which models the stack from its construction.

Programming tasks

1. Model the stack exercise described in Question 2 in a programming language with which you are familiar. The stack should just store a string representing the ISBN of the book recorded on the stack (use a made-up truncated ISBN that is easy to type).
   The details of how the stack is constructed and the details of the operations which access this stack should be hidden from the program which uses the stack in a separate module/unit - see queue code example in *Figure 4.1.7.6*.
   The program should use the operations exported from the stack module/unit to add and remove books from the stack.

In this chapter you have covered:

- That details of how data are actually represented are hidden, allowing new kinds of data objects to be constructed from previously defined data types

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

4.1.8 Problem abstraction/reduction

Learning objectives:

- *Know that the details are removed until the problem is represented in a way that is possible to solve because the problem reduces to one that has already been solved.*

Pigeonhole principle

Generalisation can be applied to problem solving by identifying a principle that is shared among solutions to different problems.

This is an example of **problem abstraction** or **reduction**.

One example of this is the **pigeonhole principle**, which states that, given two natural numbers n and m with n > m, if n items are put into m pigeonholes, then at least one pigeonhole must contain more than one item (*Figure 4.1.8.1*).

A simple application of this principle is a situation where there are six people who want to play football (n = 6 objects) but there are only five teams available that they can play for (m = 5 holes).

This would not be a problem except that each of the six refuses to play on a team with any of the other five.

To prove that there is no way for all six people to play football together, the pigeonhole principle says that it is impossible to allocate six people among five teams without putting two of the people on the same team. Since they refuse to play on the same team, at most five of the people will be able to play.

Why is this principle applicable to more than just the football team problem? The reason is that when **unnecessary details of specific problems are ignored**, **the problems reduce to the common problem** of fitting n items into m holes. For example, there must be at least two people in London with the same number of hairs on their head. Is this true or false?

We will use the pigeonhole principle to answer this as follows.

A typical head of hair has around 150 000 hairs; it is reasonable to assume that no one has more than 1 000 000 hairs on his or her head (m = 1 000 000 holes) but there are more than 1 000 000 people in London (n is bigger than 1 million objects). If we assign a pigeonhole for each number of hairs on a head and assign people to the pigeonhole with their number of hairs on it, there must be at least two people with the same number of hairs on their head.

Figure 4.1.8.1

Question

1. If there are n persons (where n > 1) who can shake hands with one another, explain using the pigeonhole principle why there is always a pair of persons who shake hands with the same number of people. Here the holes correspond to the number of hands shaken.

2. Assume that in a box there are 10 black socks and 12 blue socks and you need to get one pair of socks of the same colour. Supposing you can take socks out of the box only once and only without looking. How many socks do you have to pull out of the box before you are guaranteed to have two socks of the same colour? Use the pigeonhole principle.

Single licence - Abingdon School

Data representation

Representation abstraction is applied to problem solving. It is also known as **problem abstraction** or **reduction**. Details are removed until it becomes possible to represent the problem in a way that is possible to solve.

For example, mobile phone networks consist of a series of base stations, each of which provides coverage over a limited range for mobile phones as shown in *Figure 4.1.8.2*.

A mobile phone signal is not strong enough to reach across the country. Instead, a mobile phone communicates with its nearest base station, usually mounted on a mast. The nearest base station then relays the signal over a cabled network to a base station in a neighbouring cell, which in turn passes the signal onto another base station.

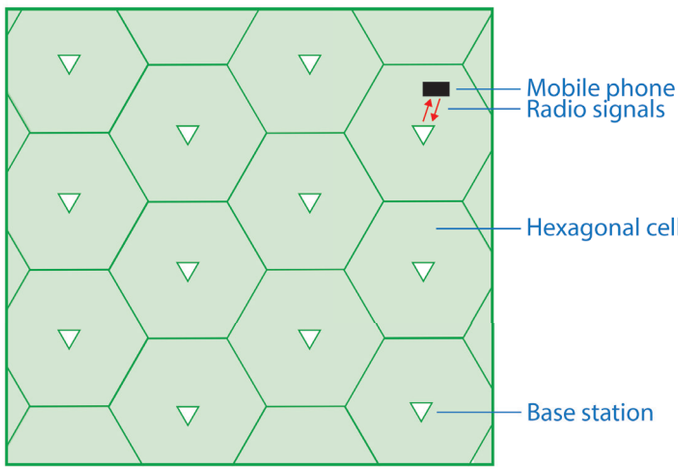


Figure 4.1.8.2 Mobile phone cellular network

The mobile phone company needs to pay for the radio frequencies used by its customers and base stations. These frequencies are very expensive, so the company needs to pay for as few as possible and reuse them.

The problem is that if mobile phones and base stations in neighbouring cells use the same broadcast frequency, calls will interfere with each other, resulting in poor communications.

Suppose that you are required to work out the minimum number of frequencies that a mobile phone company must buy in order to provide interference-free coverage of the area covered by phone masts located as indicated in *Table 4.1.8.1* and *Figure 4.1.8.3*.

How could you represent the information from *Figure 4.1.8.3* and *Table 4.1.8.1* to make it easier to understand?

The solution is to remove unnecessary detail and represent it as a collection of circles (vertices) and connecting lines (edges)

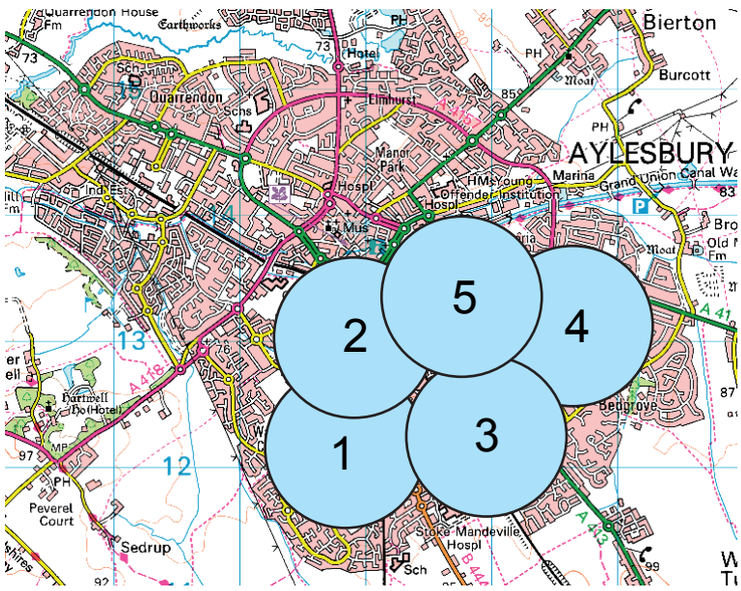


Figure 4.1.8.3 Locations and mobile coverage of the mobile phone masts

Anquet Technology Ltd © Crown copyright 2016 OS 100057707

Mobile phone masks	Location	Potential for interference with
1	Stoke Mandeville Road	2, 3
2	Town centre	1, 5
3	Wendover Road	1, 4, 5
4	Broughton	3, 5
5	Canal Walk	2, 3, 4

Table 4.1.8.1 Mobile phone mast potential signal interference

Single licence - Abingdon School

(*Figure 4.1.8.4*). *Figure 4.1.8.4* is known as a graph. This is a model of the mobile phone network shown in *Figure 4.1.8.3*.

Next represent *Figure 4.1.8.4* in table form (*Table 4.1.8.2*). *Table 4.1.8.2* shows that mobile phone masts labelled 2 and 3 are adjacent to mobile phone mast labelled 1. By adjacent is meant that the transmissions from mobile phone masts 2 and 3 overlap with those from mobile phone mast 1.

Figure 4.1.8.5 shows a solution to this problem. Different frequencies are represented by different colours in this figure. The solution shows that only three frequencies are required. Every vertex (base station) uses a frequency that will not interfere with a neighbouring vertex (base station) with which it is connected by an edge, i.e. mobile transmissions overlap.

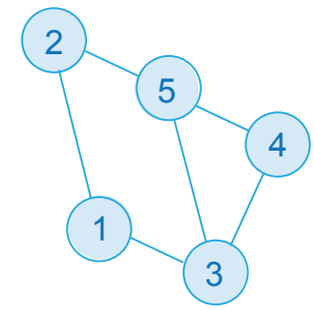


Figure 4.1.8.4 Abstract representation of mobile phone network

Vertex	Adjacent
1	2, 3
2	1, 5
3	1, 4, 5
4	3, 5
5	2, 3, 4

Table 4.1.8.2 Shows the adjacency of mobile phone masts to each other

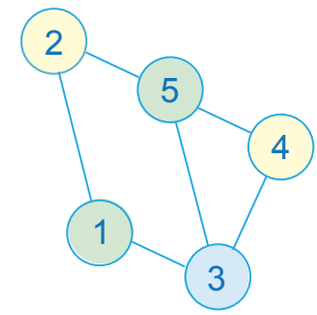


Figure 4.1.8.5 Solution to the mobile phone frequency allocation problem

Map colouring problem

Figure 4.1.8.6 shows a map divided into numbered sections.

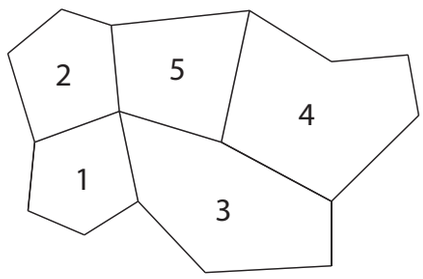


Figure 4.1.8.6 Outline map

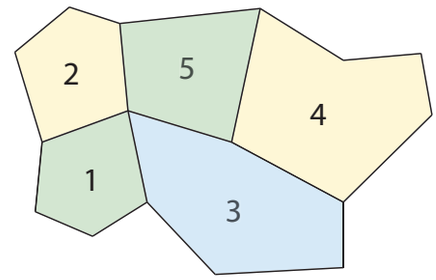


Figure 4.1.8.7 Outline map with countries coloured

Figure 4.1.8.7 shows the solution to the problem of colouring the countries in this map according to the two rules:

- No countries that share a border should be the same colour.
- As few colours as possible should be used.

If we abstract away the shape of each section of the outline map of *Figure 4.1.8.6* and represent each section by a circle (vertex) and each boundary between two countries by a connecting line (edge) we obtain a similar graph to *Figure 4.1.8.4*. If we do the same for the solution to the map colouring problem shown in *Figure 4.1.8.7*, we obtain a similar graph to that shown in *Figure 4.1.8.5*.

Both problems - allocation of mobile phone frequencies and map colouring - share a common representation when unnecessary details are removed. Both **reduce** to a similar **abstraction**, a graph - *Figure 4.1.8.4*. If a problem can be transformed into a representation of another for which a solution exists then this solution can be used to solve the problem. Simply apply the algorithm of the existing solution to the new problem.

Single licence - Abingdon School

Algorithm to solve mobile phone mast frequency selection and map colouring problem

Table 4.1.8.3 shows the algorithm to allocate mobile phone mast frequencies and to colour map sections.

```
Repeat
  Remove vertex with lowest connectivity and its links from graph
  Add this vertex and its list of connected vertices to a stack
Until graph empty
Repeat
  Add vertices back into graph in reverse order of removal, noting a legal colour each time
Until Stack empty
```

Table 4.1.8.3 Frequency selection and map colouring algorithm

Figure 4.1.8.8 shows the first stage of this algorithm being traced. *Figure 4.1.8.9* shows the second stage of this algorithm being traced.

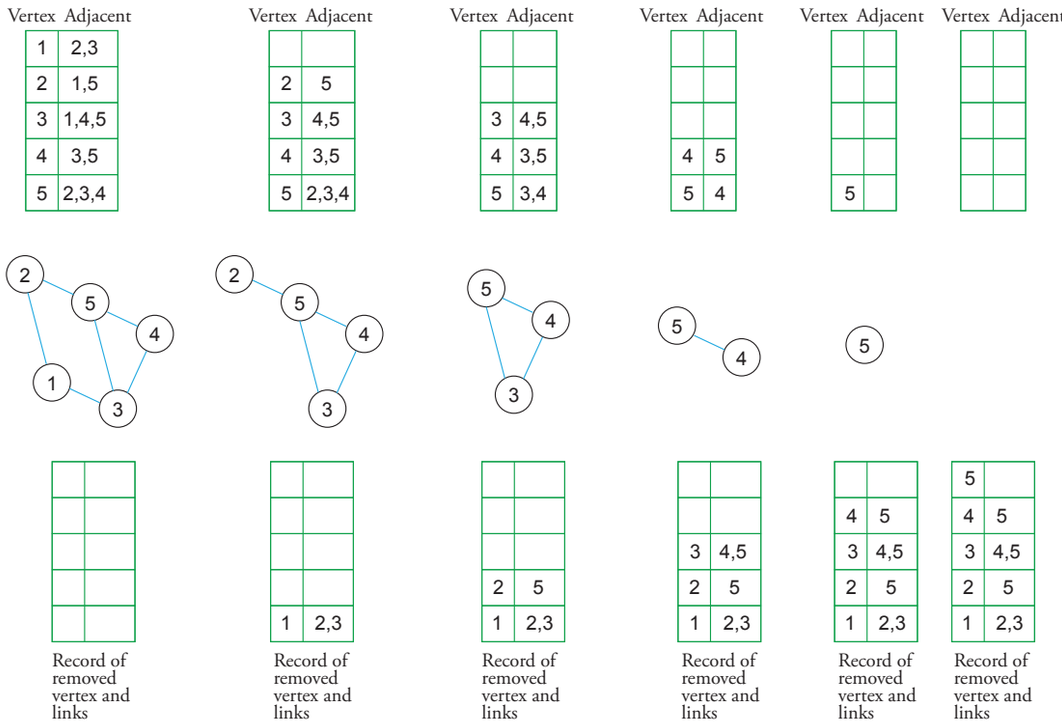


Figure 4.1.8.8 Tracing the first stage of the algorithm for selecting frequencies

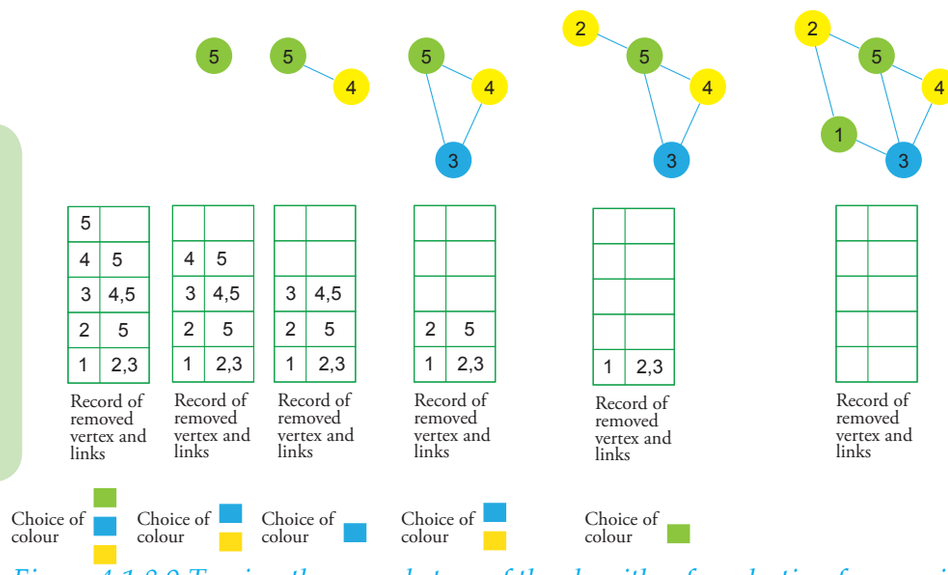


Figure 4.1.8.9 Tracing the second stage of the algorithm for selecting frequencies

Key point

Common representation: Find a common representation of a problem. Find an algorithm to solve problems so represented. Learn to transform other problems into this representation.

Single licence - Abingdon School

Question

3 Using a map of South America, draw a representation that would help in the task of colouring this map with the minimum number of colours so that no two countries sharing a border would have the same colour.

Task

1 Watch Professor Paul Curzon's video on Graphs https://www.youtube.com/watch?v=xRKCDcRVpQg. This video illustrates problem abstraction/reduction by considering a puzzle game based the Knight's tour problem which can be reduced to the same problem of visiting tourist attractions in which each is visited only once.

In this chapter you have covered:

- Problem abstraction/reduction whereby *details of the problem are removed until the problem is represented in a way that is possible to solve because the problem reduces to one that has already been solved.*

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Know that procedural decomposition means breaking a problem into a number of sub-problems, so that each sub-problem accomplishes an identifiable task, which might itself be further subdivided*

4.1.9 Decomposition

Procedural decomposition

Breaking big problems into smaller problems, or sub-problems, is called a top-down strategy. It may be far easier to deal with a number of smaller problems each of which accomplishes a single identifiable task than one large problem.

When the top-down strategy is used to plan a solution it is called **top-down design**.

The process of breaking a problem down through successive steps into smaller problems is also known as **stepwise refinement**.

Stepwise refinement has been covered in detail in *Chapter 1.2.2* on structured programming.

In essence, a problem is sub-divided into a number of sub-problems. If these can be solved directly then the decomposition is complete. If not then sub-problems are further divided and so on until small enough problems result which identify a single task and which can be directly coded in a functionally cohesive subroutine (see *Chapter 1.2.2*).

The solution to the problem now consists of a sequence of subroutine calls each of which contributes to a part of the solution.

Key concept

Procedural decomposition: Procedural decomposition is a top-down strategy in which big problems are broken into smaller problems by a process of stepwise refinement which continues until small enough problems result identifying a single task which can be directly coded as a functionally cohesive subroutine.

Questions

1. What is meant by decomposition in the context of problem solving?

In this chapter you have covered:

- *Procedural decomposition which means breaking a problem into a number of sub-problems, so that each sub-problem accomplishes an identifiable task, which might itself be further subdivided*

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Know how to build a composition abstraction by combining procedures to form compound procedures*
- *Know how to build data abstractions by combining data objects to form compound data.*

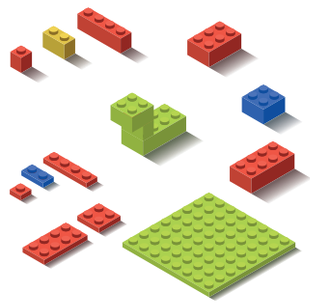


Figure 4.1.10.1 Various types of Lego bricks which have proved useful and which are used by millions to construct Lego® buildings, etc

Did you know?

David Wheeler, University of Cambridge, Computer Laboratory, was the inventor of closed subroutines and the subroutine call in 1949.

Maurice Wilkes was director of this laboratory at the time and led a team of people tasked with building of a library of subroutines for the EDSAC 1 and subsequent digital stored program computers.

4.1.10 Composition

Composition abstraction

Composition

In the process known as bottom-up development, a solution is developed by "plugging" together existing components whose reliability and usefulness have been demonstrated already, e.g. assorted Lego® bricks as shown in *Figure 4.1.10.1* can be used to build structures such as model toy houses. A model toy house built of Lego bricks is an example of composition.

There is little abstraction in Lego bricks, "what you see is what you get". However, in something as complex as a motor car, the building blocks that most people can identify are the engine, the gearbox, the brakes, etc which a car is composed of. Each of these components is an abstraction hiding considerable complexity. Workers on car production lines can put cars together without needing to focus on the internal workings of the components because the engine comes as a unit as does the gearbox and so on. These components just need to be "plugged" together correctly to make a motor car.

Combining procedures to form compound procedures

In a similar manner, software can be assembled from tried and tested components. Writing correct software is a demanding, time-consuming activity. It makes sense therefore to reuse software wherever possible. If the software is modular, then modules can be combined in different ways to solve new problems. The procedure is one form of module.

> *"By June 1949, people had begun to realize that it was not so easy to get a program right as had at one time appeared. It was on one of my journeys between the EDSAC[1] room and the punching equipment that the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs."*
>
> Maurice Wilkes, computing pioneer

Suppose a programmer relies upon a routine task again and again in different programs that he or she writes. This task might be to sort a list of integers or to extract all English words of length *n* from a given text file.

After using and debugging these procedures in many programs, the programmer might be confident in the reliability of these procedures and decide that a better approach would be to combine together the procedures needed, say, for the sorting operation, into a **compound procedure** called *Sort*.

1 EDSAC 1 was the world's first complete and fully operational regular electronic digital stored program computer.

Single licence - Abingdon School

```
Procedure Sort (List)
  Procedure A
    Begin
      ...
      ...
    End;
  Procedure B
    Begin
      ...
      ...
    End
Begin
  ...
  A
  ...
  ...
  B
  ...
  ...
End
```

Table 4.1.10.1 Compound procedure called sort which defines other procedures within itself

This procedure could itself contain subordinate procedures called nested procedures which it could rely upon for the task of sorting a list. *Table 4.1.10.1* shows an example outline structure for such a compound procedure.

The detail of the sorting operation has been abstracted away by creating this named compound procedure, *Sort*. This is a form of **encapsulation**.
The example shown in *Figure 4.1.10.1* is one way that encapsulation can be achieved.
If this compound procedure is provided in object form, i.e. compiled form, then whenever the programmer needs to sort a list of integers at a particular place in a program, it is sufficient to write the compound procedure name *Sort* at this point in the source code form of the program.
When this program is compiled, its object form will need to be linked to the object form of the compound procedure *Sort* for the executable program produced from the linking process to call *Sort* at runtime.

Libraries of (compound) generally useful procedures

It may transpire that the compound procedure *Sort* is so useful that other programmers would like to use it. To do this, the programmer could add it to a library of useful compound procedures which is made available for others to incorporate into their programs.
The standard Java graphics library, Java Swing, is quite challenging to use.
By wrapping Java Swing operations in higher-level procedures, access to graphics in Java can be made much easier.

Table 4.1.10.2 shows an example in which simpler "compound procedures" e.g. *StdDraw.line*, available in a Java class file (object file) *StdDraw* are used by the class *Plot*.
These "compound procedures" have abstracted away the messier detail of the Java Swing drawing procedures. *Figure 4.1.10.2* shows the output from the Java program in *Table 4.1.10.2* when it is executed.

```
public class Plot {
    public static void main(String[] args) {
        StdDraw.setPenRadius(0.05);
        StdDraw.setPenColor(StdDraw.BLUE);
        StdDraw.point(0.5, 0.5);
        StdDraw.setPenColor(StdDraw.MAGENTA);
        StdDraw.line(0.2, 0.2, 0.8, 0.2);
    }
}
```

Table 4.1.10.2 Java program which uses compound procedures in an object class file StdDraw

Figure 4.1.10.2 Drawing created by Java program in Table 4.1.10.2

Table 4.1.10.3 shows a Delphi program that uses a compound procedure *Map* which can take a function, e.g. *Sqr* and apply (map) this to each element of an array, *A*, whose length is not specified until runtime.
In *Table 4.1.10.3* *Map* is called as follows

```
Map(Sqr, MyArray);
```

The output of this program is shown in *Figure 4.1.10.3*.

Procedure *Map* is passed array *MyArray* by reference and function *Sqr*.
Map applies function *Sqr* to each element of *MyArray*. On return from procedure *Map*, the *For loop* iterates through *MyArray* element by element, displaying each element's value on its own line.

Single licence - Abingdon School

Table 4.1.10.4 shows the Delphi Unit in which compound procedure *Map* is defined.

Although in this example, *Map* has been designed to work with any function that takes a single integer input and returns an integer result, *Map* could have been written to use a procedure instead of a function.

The details of how procedure *Map* is implemented are hidden by being placed in the implementation section of the unit. Only items placed in the interface section are visible outside the unit.

The program *ArrayTypeMap* in *Table 4.1.10.3* only needs access to the object form of *Unit1* when it is compiled and linked into an executable.

```
Program ArrayTypeMap;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils,
  Unit1 in 'Unit1.pas';
Var
  MyArray : Array[1..5] Of Integer = (1,2,3,4,5);
  i : Integer;
Function Sqr(x : Integer) : Integer;
  Begin
    Sqr := x * x;
  End;
Procedure Map(F : FunctionPtr; Var A : Array Of Integer);
  Var
    i : Integer;
  Begin
    For i := 1 To Length(A)
      Do A[i] := F(A[i]);
  End;
Begin
  Map(Sqr, MyArray);
  For i :=1 To Length(MyArray)
    Do Writeln(MyArray[i]);
  Readln;
End.
```

Figure 4.1.10.3

Table 4.1.10.3 Delphi program that uses a compound procedure Map which can take a function, e.g. Sqr and applies this to each element of an array, A

The compiler uses the interface information of *Unit1* to compile program *ArrayTypeMap* and the linker, which "stitches" object files together, requires the object forms of *ArrayTypeMap* and *Unit1* to produce an executable file, *ArrayTypeMap.Exe*.

```
Unit Unit1;
Interface
  Type
    FunctionPtr = Function(x : Integer) : Integer;
    Procedure Map(F : FunctionPtr; Var A : Array Of Integer);
Implementation
  Procedure Map(F : FunctionPtr; Var A : Array Of Integer);
    Var
      i : Integer;
    Begin
      For i := 1 To Length(A)
        Do A[i] := F(A[i]);
    End;
End.
```

Table 4.1.10.4 Delphi Unit in which compound procedure Map is defined.

Single licence - Abingdon School

Programming tasks

1 Combine procedures together to form a compound procedure which opens a given text file, A, for reading, opens a new text file for writing, B, reads the contents of A separating words from the surrounding white space (e.g. space character(s)), writes each word on a separate line to B.
Write and test a program which uses your compound procedure on text files. The program should prompt the user for the name of the text file to be read and the name of the text file to be created and written to. Your compound procedure should be written to accept as parameters the filenames of the reading and writing text files and nothing else.

Data abstraction

Combining data objects to form compound data

Chapter 2.1.2 introduced arrays. An array is a compound or composite data type which allows a collection of data objects of the same type to be grouped into a single object. The objects placed in the cells of an array must be of the same data type but this can be any data type including another composite data type.

By aggregating data objects in this way we can compose named data structures which can be referenced collectively by their name. For example, it is possible to create queue data structures and stack data structures in this way.

This is data abstraction which has been covered in depth in *Chapter 4.1.7*.

In this chapter you have covered:

- *How to build a composition abstraction by combining procedures to form compound procedures*
- *How to build data abstractions by combining data objects to form compound data*

Single licence - Abingdon School

4 Theory of computation

4.1 Abstraction and automation

Learning objectives:

- *Understand that automation requires putting models (abstraction of real world objects/phenomena) into action to solve problems. This is achieved by*
  - *creating algorithms*
  - *implementing the algorithms in program code (instructions)*
  - *implementing the models in data structures*
  - *executing the code*

4.1.11 Automation

Models into action

The key process in Computer Science is **abstraction** which means building models which represent aspects of behaviour in the real-world which are of interest. For example, if we wanted to build an automated recommendation system for an online book store, we might choose to record the types of book and number of each type purchased as well as details that identify the respective customer.

Computer Science is not alone in building abstractions, mathematics and the natural sciences also build abstractions but their models only serve to describe and explain whereas Computer Science must, in addition, perform actions on and with the data that has been modelled if it is to solve problems. These actions are described by **algorithms** or step-by-step instructions which form what is called the **automation** stage of problem solving.

Whilst it is true that automation of tasks existed before Computer Science, their nature involved concrete, real-world objects, e.g. the Jacquard loom, not informational abstractions such as an online book recommendation system.

Computer scientists must turn their models into data structures which are supported in the programming language that will be used to implement the algorithms.

When the program-coded algorithms execute, the information stored in the data structures is transformed into new forms of information. This is the automation stage.

The scale and speed of processing of informational abstractions made possible by modern digital computers surpasses by many orders of magnitude anything that could be achieved by manual means.

Questions

1. Describe **two** informational abstractions in which automation plays a large role.

In this chapter you have covered:

- Automation requires putting models (abstraction of real world objects/phenomena) into action to solve problems. This is achieved by
  - creating algorithms
  - implementing the algorithms in program code (instructions)
  - implementing the models in data structures
  - executing the code

Single licence - Abingdon School

4 Theory of computation

4.2 Regular languages

Learning objectives:

- *Be able to draw and interpret simple state transition diagrams and state transition tables for FSMs with no output and with output (Mealy machines only).*

Figure 4.2.1.2 Combination lock

4.2.1 Finite state machines (FSM)

What is a finite state machine?

A **finite state machine** is a machine that consists of a fixed set of possible states, a set of allowable inputs some of which can change the machine's state, and, if it is a finite state machine with outputs, a set of possible outputs. *Figure 4.2.1.1* is the **state transition diagram** for a combination lock with combination code 537. A state transition diagram is a way of describing a FSM graphically.

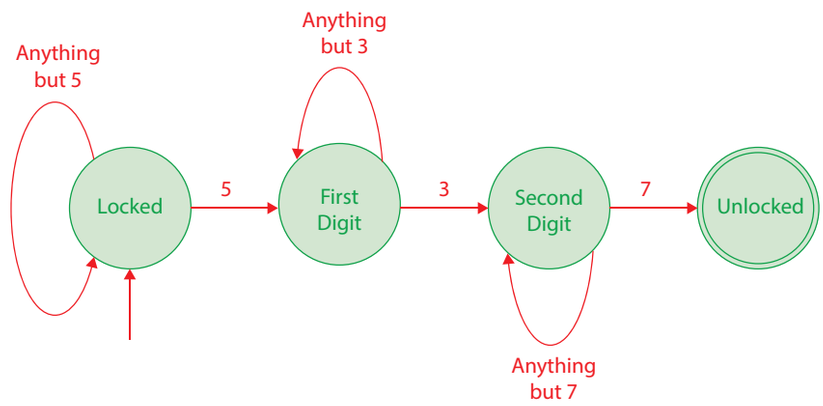


Figure 4.2.1.1 State transition diagram for a combination lock with combination code 537

The combination lock is an example of a finite state machine (FSM) with the following states

- Locked
- First Digit
- Second Digit
- Unlocked

and the set of allowable inputs {0-9} of which 5, 3 and 7 can change the state of the machine when entered in the right sequence. This FSM is without outputs.

The initial state of the combination lock is indicated by an arrow pointing to the state 'locked'. When the lock is in state 'locked' and it receives the input 5, the lock moves to the state 'First Digit'. When it receives the input 3, the lock moves to the state 'Second Digit'. When it receives the input 7, the lock moves to the state 'Unlocked'. At any stage, if a digit other than the required digit is entered, the lock remains in its current state. The final or halting state is reached when the combination lock is unlocked. The state 'Unlocked' is indicated by a double circle because it is the goal state. The goal state is more commonly known as the **accepting state**.

Key term

Finite state machine (FSM): A finite state machine is a machine that consists of a fixed set of possible states, a set of allowable inputs (input symbol alphabet) some of which can change the machine's state, and, if it is a finite state machine with output, a set of possible outputs.

Key term

State transition diagram: A way of describing an FSM graphically. Each state is represented by a circle and each transition by an arrow labelled with the input that causes the transition.

Single licence - Abingdon School

Key term

Finite state machine with no output:
A finite state machine with no output is an FSM that produces no output while processing the input but which responds YES/ACCEPT or NO/REJECT when it has finished processing the input.

FSMs with no output have an initial state and one or more accepting or goal states.
State transition diagrams use a special arrow to indicate the initial or starting state and a double circle to indicate the accepting states or goal states.

Finite state machine with no output

A **finite state machine with no output** is an FSM that produces no output while processing the input.
Finite state machines with no output solve decision problems such as "does the input string end in the symbol a?".
Decision problems are ones where the outcome is either YES or NO.
A finite state machine with no output simply runs through the input sequence of symbols, one by one, changing state or not as a result of the current state and the current symbol from the input it sees.
On reaching the end of the input, it stops and, depending on which state it stopped in, accepting state or a non-accepting state, it is able to decide whether the input sequence has met the criteria or not, e.g. "does the input sequence of a's and b's end in the symbol a?".

The states which cause the FSM to indicate YES, the criteria have been met, are called **accepting states**. All other states are, by default, states that cause the FSM to indicate NO, the criteria have not been met. Accepting states are indicated by putting two rings round them, like S_1 in *Figure 4.2.1.3*. An FSM with no output always has one or more accepting states.

Figure 4.2.1.3 is a finite state machine with no output which accepts a sequence of a's and b's which end in the symbol a. The finite state machine uses an alphabet {a, b}.

Its start state is state S_0 indicated by →. It has one accepting state, S_1. The arrow-headed arcs represent transitions between states or a transition which starts and ends on the same state or the start state.

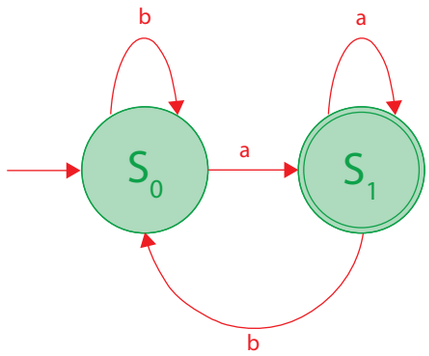


Figure 4.2.1.3 A finite state machine with no output that accepts a sequence of a's and b's which end in the symbol a

Figure 4.2.1.4 shows this finite state machine in a simulator accepting bbbbabbaaa because this sequence ends in the symbol a.

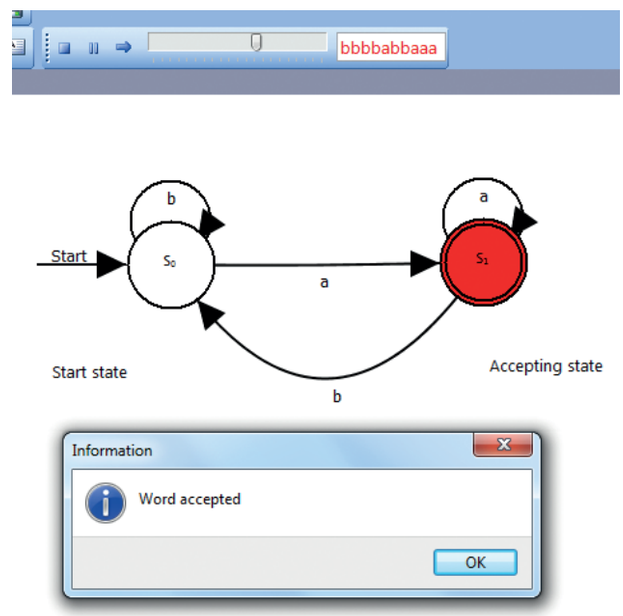


Figure 4.2.1.4 A finite state machine simulator FSMSim showing that the sequence bbbbabbabaaaa is accepted

Key point

FSMs with no output solve decision problems, i.e. problems where the outcome is either YES or NO.

Key point

The states which cause the FSM to produce a YES or ACCEPT response are called accepting states. All other states cause the FSM to produce a NO or reject response.

Single licence - Abingdon School

Figure 4.2.1.5 shows this finite state machine in a simulator rejecting bbbbabbaab because the sequence of symbols does not end in the symbol a.

One reason that FSMs are so useful is that they can recognise sequences, e.g. the set of valid strings for some application.

Key point

To avoid an FSM having to report an error, ensure that for each state there is an outgoing transition for every symbol in the machine's input alphabet.

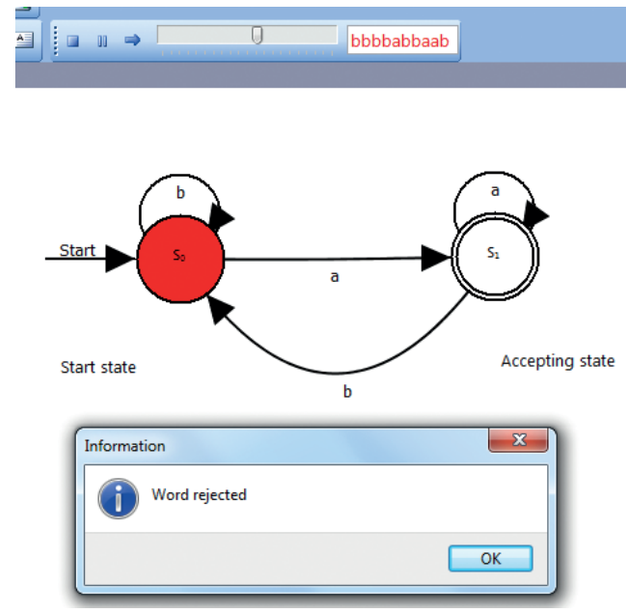


Figure 4.2.1.5 A finite state machine simulator FSMSim showing that the sequence bbbbabbaab is rejected

Question

1 *Figure 4.2.1.6* is a finite state machine with alphabet {a, b}. State whether the following sequences of symbols from this alphabet are accepted or rejected:
(a) aa (b) ab (c) ba (d) bbaa (e) aaabbbbaa
(f) aabbbabbba

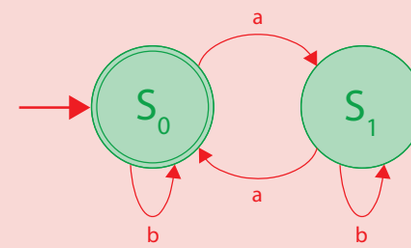


Figure 4.2.1.6 A finite state machine with no output for alphabet {a, b}

2 *Figure 4.2.1.7* is a finite state machine with alphabet {a, b}. State whether the following sequences of symbols from this alphabet are accepted or rejected:
(a) abaa (b) ababa
(c) abababaa
(d) ababababa
(e) ababaaa (f) ababaab
(g) ababaabaa

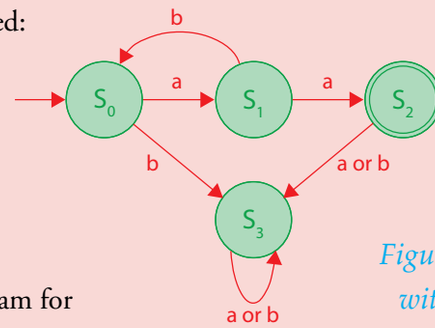


Figure 4.2.1.7 A finite state machine with no output for alphabet {a, b}

3 Draw the state transition diagram for a finite state machine which accepts a sequence of symbols containing an odd number of 1's. The alphabet for this machine is {0, 1}.

4 Draw the state transition diagram for a finite state machine which accepts a sequence of symbols beginning with an a followed by zero or more a's or b's. The alphabet for this machine is {a, b}. For example, abbbaababb is a valid sequence of symbols from the alphabet.

5 *Figure 4.2.1.8* is a finite state machine with alphabet {0, 1}. State whether the following sequences of symbols from this alphabet are accepted or rejected:
(a) 0 (b) 1 (c) 0111 (d) 0111110
(e) 11111 (f) 1011 (g) 111000

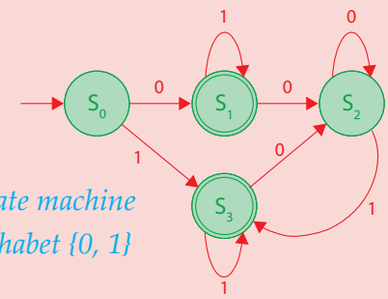


Figure 4.2.1.8 Finite state machine with no output and alphabet {0, 1}

Single licence - Abingdon School

Key term

State transition table:
A state transition table shows the effect of particular inputs on the current state of an FSM with no output.

State transition table

We use a table as shown in *Table 4.2.1.1* to select the next state given the current state and the current input symbol. For example, if the current state is S_0 and the input symbol is a then the FSM moves to state S_1 because the table shows that this is as the next state. If the current state is S_0 and the input symbol is b then the FSM moves to state S_3. This table is called a **state transition table** or just a **state table**. The state transition table in *Table 4.2.1* corresponds to the state transition diagram shown in *Figure 4.2.7*.

Current state	S_0	S_0	S_1	S_1	S_2	S_2	S_3	S_3
Input symbol	a	b	a	b	a	b	a	b
Next state	S_1	S_3	S_2	S_0	S_3	S_3	S_3	S_3

Table 4.2.1.1 A state transition table for finite state machine with no output

Question

6 Draw the state transition table for the finite state machine in *Figure 4.2.1.9*.

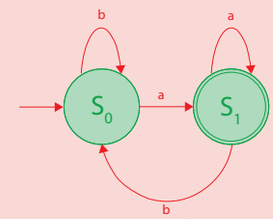


Figure 4.2.1.9 Finite state machine with no output and alphabet {a, b}

7 Draw the state transition table for the finite state machine in *Figure 4.2.1.10*.

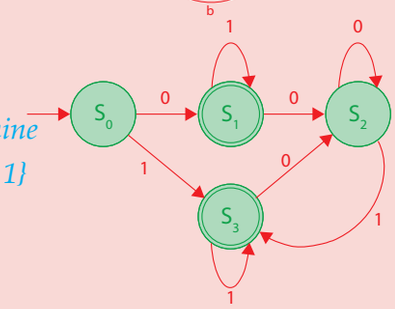


Figure 4.2.1.10 Finite state machine with no output and alphabet {0, 1}

Did you know?

Why are finite state machines useful?
Finite state machines (FSMs) are used extensively in applications in computer science. For example, finite state machines are the basis for

- Any kind of controller, e.g. traffic lights
- Specifying a language, e.g. given any string, an FSM determines if that string is in the language or not
- Programs for
  - spell checking
  - grammar checking
  - indexing or searching large bodies of text
  - recognising speech
  - processing text containing mark-up languages such as XML and HTML
- networking protocols that specify how computers communicate.

Single licence - Abingdon School

Finite state machine with output (Mealy machine)

A state transition diagram for a finite state machine with outputs is shown in *Figure 4.2.1.11*. The edges, such as the curved arrow from state ***s*** to state ***t*** are called transitions and are labelled with a symbolic code, e.g. a|b, where ***a*** and ***b*** are symbols. The ***a*** part of the label is called the transition's trigger and denotes the input symbol. The ***b*** part, which is optional, denotes the output symbol.

A finite state machine (FSM) with outputs has a set of input symbols (input symbol alphabet) and a set of output symbols (output symbol alphabet).
We will assume that the FSM in *Figure 4.2.1.11* has an input alphabet consisting of the symbols ***a*** and ***b*** and an output alphabet consisting of the symbols ***a*** and ***b***. We can express this using set notation as follows:

input alphabet = {***a***, ***b***}

output alphabet = {***a***, ***b***}

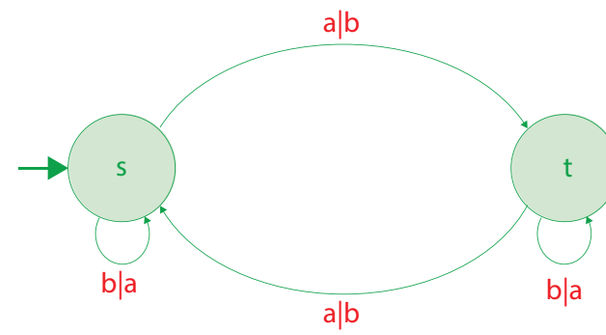


Figure 4.2.1.11 A finite state machine with outputs

{} enclose the symbols, which are members of the set.
Using this input alphabet, input strings to this FSM can be any combination of ***a*** and ***b***, such as ***aabbaab***, ***bbaa***, ***ba***, ***bbb***.

The FSM has a finite set of states, {***s***, ***t***}.
The FSM has a **transition function** that maps

a state–symbol pair *(current state and input symbol)*
to a state *(next state and output symbol)*.

Table 4.2.1.2 shows the transition table which represents this transition function.

Current state	s	s	t	t
Input symbol	a	b	a	b
Next state	t	s	s	t
Output symbol	b	a	b	a

Table 4.2.1.2 Transition function for a FSM with outputs

The FSM has one state that is designated the start or initial state. This is indicated by the symbol → on the state transition diagram.

Starting in state ***s***, the FSM takes input ***aabba*** and generates output ***bbaab***.
The purpose of this FSM is to convert ***a***'s into ***b***'s and ***b***'s into ***a***'s.
To prevent ambiguity about the machine's next move, i.e. in order that its behaviour is deterministic, we require that each trigger has just one outgoing transition from a particular state. *Figure 4.2.1.12* shows a state transition diagram for which there is more than one outgoing transition for the same trigger.
State ***s*** has two transitions labelled with the trigger ***a***.
This diagram describes an FSM which is non-deterministic because it contains an ambiguity. The FSM has two choices to exit state ***s*** when the input symbol is ***a***.

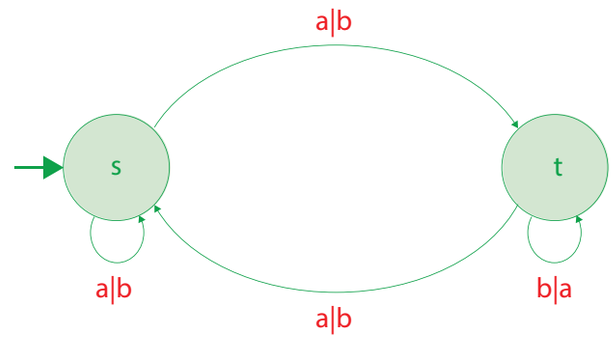


Figure 4.2.1.12 A non-deterministic finite state machine with outputs

States that have no outgoing transitions are called **halting states**.

Single licence - Abingdon School

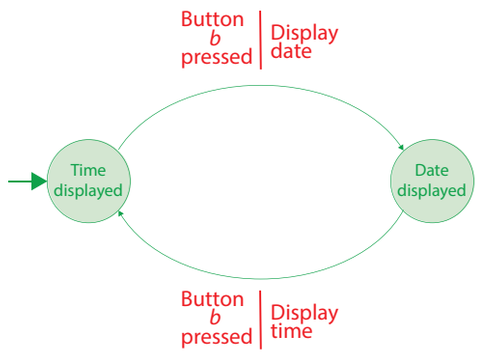


Figure 4.2.1.13 A finite state machine with outputs that are actions

There are two error conditions that may arise

- If the FSM has not exhausted its input when it finds itself in a halting state, it will usually report an error.
- It will also usually report an error if it is in a state with outgoing transitions, but none of them has a trigger which matches the current input symbol.

The FSM may be constructed to perform an action at the output stage instead of outputting a symbol:

(input symbol, current state) → (action, next state)

If a match is found for the pair *(input symbol, current state)* then an associated action function will be called and the current state will be set to the next state.

Figure 4.2.1.13 illustrates this for a digital watch with a button *b* which is pressed to switch state. The input symbol is replaced by a signal that signals the event pressing button *b*.

In *Figure 4.2.1.13* the input stage is a signal that signals an event – pressing button *b* – rather than a symbol.

An FSM can be designed so that if it cannot find a match for *(input symbol, current state)*, it will search the table of transitions for transitions of the form *(current state) → (action, next state)*

Current state	s	s	t	t	t
Input symbol	a	b	a	b	
Next state	t	s	s	t	s
Action	Action 1	Action 2	Action 3	Action 4	Action 5

Table 4.2.1.3 Transition function for a FSM with outputs

In this way, transitions can be defined for a current state and any input symbol (*Table 4.2.1.3*). Any input symbol transitions must be searched after first searching for a specific *(input symbol, current state)* match.

It is possible to define a default transition as a catch-all case, to be used when

(input symbol, current state) → (action, next state)

and

(current state) → (action, next state)

produce no matches in the transition table. The action will call an exception handler.

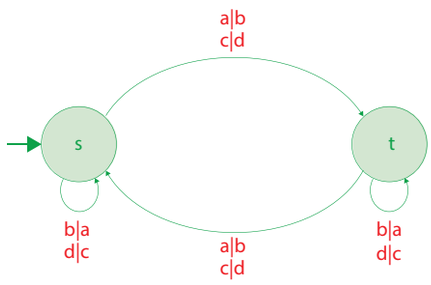


Figure 4.2.1.14 A finite state machine with outputs and more than one trigger

It is possible to have more than one trigger from one state to another state, as shown in *Figure 4.2.1.14*.

If the FSM is in state *s* and the input symbol is *c*, the FSM outputs symbol *d* and moves to state *t*. Similarly, if while in state *s* the input symbol is *a*, the FSM outputs symbol *b* and moves to state *t*.

Single licence - Abingdon School

Questions

8 What is the output Y from the FSM shown in *Figure 4.2.1.15* when the input X is 001011? Assume that the bits are submitted to the FSM in the order of most significant bit through to least significant bit.

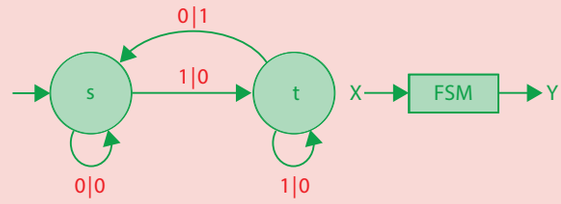


Figure 4.2.1.15

9 (a) What is the output Y from the FSM shown in *Figure 4.2.1.16* when the input X is 00101100? Assume that the bits are submitted to the FSM in the order of most significant bit through to least significant bit.

(b) What is the final state of the machine when all the input has been processed?

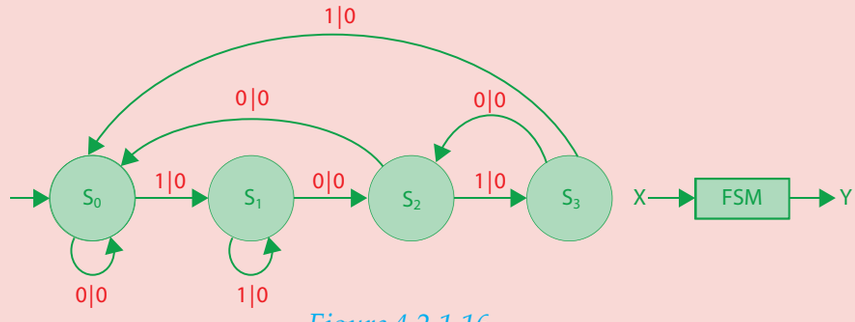


Figure 4.2.1.16

10 *Table 4.2.1.4* shows the transition table for a finite state machine with outputs. The alphabet is {0, 1}. Draw the equivalent state transition diagram.

Current state	S_0	S_0	S_1	S_1
Input symbol	0	1	0	1
Next state	S_1	S_0	S_0	S_1
Output symbol	1	0	1	0

Table 4.2.1.4

11 *Figure 4.2.1.17* is an FSM with an input alphabet of {0, 1} and an output alphabet of {0, 1, =, c}.
It has four halting states: S_2, S_4, S_5, S_6.
What is the machine's output for the inputs (a) 00, (b) 01, (c) 10, (d) 11?

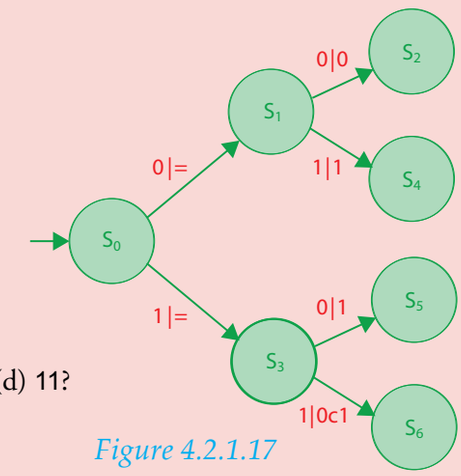


Figure 4.2.1.17

In this chapter you have covered:

- Drawing and interpreting simple state transition diagrams and state transition tables for FSMs with no output and with output (Mealy machines only).

Single licence - Abingdon School

4 Theory of computation

4.2 Regular languages

Learning objectives:

- *Be familiar with the concept of a set and notations for specifying a set*
- *Be familiar with compact representation of a set*
- *Be familiar with concept of:*
  - *finite sets*
  - *infinite sets*
  - *countably infinite sets*
  - *cardinality of a finite set*
  - *Cartesian product of sets*
- *Be familiar with the meaning of the term:*
  - *subset*
  - *proper subset*
  - *countable set*
- *Be familiar with the set operations:*
  - *membership*
  - *union*
  - *intersection*
  - *difference*

Key concept

Set:
A set is an unordered collection of values in which each value occurs at most once.

Key point

We can use set comprehension to define sets, e.g. the set of integers for which each element satisfies 2x + 1 where x is an integer, and x > 1, as follows

$A = \{ 2x + 1 \mid x \in \mathbb{Z} \wedge x > 1 \}$

4.2.2 Maths for regular expressions

Sets

A set is an unordered collection of different things or objects, which we call the members or elements of the set.
A set has the property that no two members of the set are the same.
A set can contain anything—numbers, people, things, words, and so on, but we will mostly deal with sets that contain numeric values such as 1, 2 ,3 or symbolic values e.g. letters A, B, C.
A set is defined by its members.
To indicate that a collection is a set, we use curly braces, so that the set of the numbers 1, 4, and 5 is written as

$$\{1, 4, 5\}$$

Two sets are considered equal if they have exactly the same members. For example, the following two sets are equal because the order of the elements within a set does not matter

$$\{7, 9, 32, 8\} \text{ and } \{32, 8, 7, 9\}$$

We can also define a set A by stating a property that its members have.
An example is "the set of integers greater than 1".
We express this as

$$A = \{ x \mid x \text{ is an integer and } x > 1 \}$$

Here the vertical bar "|" is read as "such that".
The set A consists of those objects x such that x is an integer and x > 1 is true.
This may also be expressed in a form called set comprehension as follows

$$A = \{ x \mid x \in \mathbb{Z} \wedge x > 1 \}$$

where $x \in \mathbb{Z}$ means x is a member of the set of integers, $\mathbb{Z}$.
The symbol $\wedge$ means AND.
The term $\wedge\ x > 1$ means AND x is greater than 1.
The set with no elements, the empty set, is denoted by {}.
An alternative symbol used for an empty set is Ø.

Questions

1. Write in set notation the following collections of numbers
   (a) 4, 6, 9, 2 (b) 1, 2, 2, 7, 8, 6, 6
2. Write the set comprehension for the following
   'x is a natural number and x <= 10'
3. List the members of the set $\{x^2 \mid x \in \mathbb{Z} \wedge 2 \le x \le 6\}$

Single licence - Abingdon School

Key concept

Finite set:
If a set S contains exactly n distinct elements where n is a nonnegative integer, we say that S is a finite set and that n is the cardinality of S.

Key concept

Infinite set:
A set is said to be infinite if it is not finite.

Key concept

Countably infinite set:
A countably infinite set is one that can be be placed in one-to-one correspondence with the natural numbers.

Key point

The set of natural numbers $\mathbb{N}$, the set of integers $\mathbb{Z}$ and the set of rational numbers $\mathbb{Q}$ are countable infinite sets as are any sets formed of infinite subsets of them, e.g. the infinite set of even natural numbers.

Compact representation of a set

The following notation can be used to represent a set which contains all strings with an equal number of 0s and 1s

$$\{0^n1^n \mid n \geq 1\}$$

This is the set {01, 0011, 000111, 00001111, …}.

Finite and infinite sets

Countable Collections

If the items in a collection can be listed, then they can be counted by assigning to them the natural numbers 1, 2, 3 and so on.
Conclusion: Such a collection is said to be countable.

Finite sets

A finite set is one whose elements can be counted off by the natural numbers up to a particular number, e.g. 6 in the following example

1	2	3	4	5	6
{ 15,	34,	12,	65,	23,	19 }

Countably infinite sets

The collection of counting numbers, i.e. natural numbers, can be listed and is therefore countable. Imagine placing the counting numbers in a list, and assigning counting numbers to them as follows:

1	2	3	4	5	6	…
1,	2,	3,	4,	5,	6,	…

The list never reaches an end. Why? Suppose it is thought that the natural number 66 is at the end of the list, but it can't be because it is followed by the number 67 and so on. The original list with 66 supposedly at the end is therefore not complete.
Doing the same thing with a really enormous number is defeated by the same argument: it is always possible to find a new number which is not in the new list, by simply adding one to the enormous number.

Conclusion: there is no counting number that represents the total number of all counting numbers.
Instead this total is given its own symbol $\aleph_0$ and special name, **aleph-zero**.

Now bizarrely, we can apply the same thought process to $\aleph_0$ i.e.

$$\aleph_0 + 1 = \aleph_0$$

$$\aleph_0 + \aleph_0 = \aleph_0$$

A rational p/q is defined by a pair (p, q) of integers, so the number of rationals is $\aleph_0 \times \aleph_0$. But this is just $\aleph_0$.

This means that other infinite sets can be counted off by the natural numbers i.e. **put in one-to-one correspondence with the set of natural numbers**.

Single licence - Abingdon School

For example, the set of positive even numbers

$$\begin{array}{cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \{ & 2, & 4, & 6, & 8, & 10, & 12, & \dots \} \end{array}$$

The total number of items in the set of positive even numbers is also $\aleph_0$.

Figure 4.2.2.1 shows by analogy how this is possible.

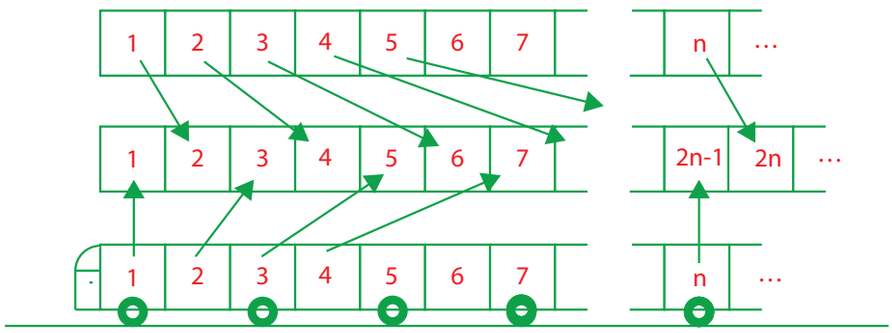


Figure 4.2.2.1 How Infinity Hotel accommodates a bus with infinitely many passengers even though it is already full

The top row shows the room numbers of Infinity Hotel which got its name because it has an infinite number of rooms, numbered by the natural numbers as shown. One day a bus arrives at the hotel with infinitely many passengers sitting in seats 1, 2, 3, 4, and so on. The bus driver asks the hotelier to accommodate all his passengers, one per room but the hotelier knows that the hotel is full. However, to avoid losing the reputation of the hotel as the Infinity Hotel, the hotelier decides to ask each guest to move as shown, the guest in room 1 is asked to move to room 2, the guest in room 2 to room 4, the guest in room 3 to room 6, and so on. The guest in room n is moved to room $2n$. This frees up all the odd-numbered rooms for the coach passengers. The passenger in seat 1 can move into room 1, seat 2 to room 3, and so on. The coach passenger in seat n is accommodated in room $2n - 1$.

We have succeeded in matching one-to-one the set of natural numbers with the set of even natural numbers.

Conclusion: Any set which can be put into a one-to-one correspondence with a countable set is also countable.

Questions

4 What is meant by a countable finite set?

5 What is meant by a countable infinite set?

6 Give **two** examples of infinite sets which are countable.

Key point

Pigeonhole principle:
It is not possible to put n + 1 objects into n boxes with at most one in each.
Clearly this principle applies only to finite sets because as we have seen with Infinity Hotel it is possible to overcome the pigeonhole principle if dealing with countably infinite sets, i.e. when $n = \aleph_0$.

Information

The idea of a hotel infinity was conceived by German mathematician David Hilbert (1862-1943)

Key point

Knowing a set is countable is useful because instead of writing

1, 2, 3, …, n

we can write

1, 2, 3, …

because eventually it is possible to get to anything you want, e.g. the infinite series

$1 + {}^1/_4 + {}^1/_9 \cdots = \pi^2/6$

Single licence - Abingdon School

Cardinality of a set

The total number of elements in a set is known as the **cardinality of the set** and this total number, a **cardinal number**.

The cardinal number of a set A is commonly denoted by |A|.

For example, the cardinality of the following finite set is 6.

$$\begin{array}{c} 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \\ \{15,\ 34,\ 12,\ 65,\ 23,\ 19\} \end{array}$$

Strange as it may seem, the cardinality of the set of positive even numbers is the same as the set of natural numbers, ℕ, the counting numbers, i.e. $\aleph_0$ as we discovered in the last section.

To indicate the order in which elements are placed, e.g the first, second and so on we use ordinal numbers. For example,the **ordinal number** 6 refers to an item in a row of items - the sixth item.

We can think of an ordinal number as being defined by the set that comes before it, e.g. ordinal number 6 is defined by the set {1, 2, 3, 4, 5}.

$$\begin{array}{c} 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \\ \{15,\ 34,\ 12,\ 65,\ 23,\ 19\} \end{array}$$

The ordinal number of the last item in a finite set is the same as the set's cardinal number.

Information

When we count objects in a collection, we notionally put them in a line and give the objects labels, 1, 2, 3, 4 until the end of the line is reached.

The last label is the count of the number of objects.

Key concept

Cardinality of a set:

The cardinality of a set is the number of elements in the set.

Questions

7 What is the cardinality of the following sets

(a) {1, 2, 3, 4} (b) {5, 7, 9} (c) {}

Infinite sets which aren't countable

Consider the table of sequences s_1, s_2, s_3, … in *Table 4.2.2.1* representing every number (rationals and irrationals) between 0 and 1 in base 2.

The subscripts in s_1, s_2, s_3, … indicate that we have put the numbers in one-to-one correspondence with the natural numbers. So, at first sight it would appear that the set of numbers defined by the numbers between 0 and 1 are countable. But the row labelled s is a new number not accounted for. It is definitely different from all the numbers in the table above by design. It has been created as follows

1. The most significant bit is chosen to be the inverse of the most significant bit of s_1
2. The next most significant bit is chosen to be the inverse of the most significant bit of s_2
3. Every other significant bit is chosen in a similar manner

$$\begin{array}{l}
s_1 = 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\ldots \\
s_2 = 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\ldots \\
s_3 = 0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\ldots \\
s_4 = 1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\ldots \\
s_5 = 1\,1\,0\,1\,0\,1\,1\,0\,1\,0\,1\ldots \\
s_6 = 0\,0\,1\,1\,0\,1\,1\,0\,1\,1\,0\ldots \\
s_7 = 1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\ldots \\
s_8 = 0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\ldots \\
s_9 = 1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\ldots \\
s_{10} = 1\,1\,0\,1\,1\,1\,0\,0\,1\,0\,1\ldots \\
s_{11} = 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,0\ldots \\
\vdots \qquad \vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\ddots
\end{array}$$

$$s = 1\,0\,1\,1\,1\,0\,1\,0\,0\,1\,1\ldots$$

Table 4.2.2.1 A failed attempt to represent every number (rationals and irrationals) between 0 and 1 in base 2

Single licence - Abingdon School

Choosing the bits of s in this way ensures that it is different from every number in the table.

After working all the way through the table of numbers there are still numbers unmatched with the counting numbers.

Conclusion 1: It is not possible to achieve a one-to-one matching of every single number, rational and irrational, between 0 and 1 with the counting numbers.

Conclusion 2: There are more numbers between 0 and 1 than there are integers and more than $\aleph_0$. As we count with the natural numbers and there are just as many natural numbers as there are integers (same $\aleph_0$), we cannot count the set of numbers between 0 and 1.

With a similar argument the same conclusion can be made of the set of rational and irrational numbers between 0 and 2 and between 0 and any number.

The conclusion thus applies to the set of real numbers $\mathbb{R}$.
The set $\mathbb{R}$ consists of the rational and the irrational numbers.
The set of real numbers is not countable.

Key point

The set of real numbers $\mathbb{R}$ is not countable.

Why is the significance of this to computation?

Chapter 4.5.1 considers Turing machines and their importance as a model of computation, what can and cannot be computed.
Each Turing machine can be identified by its description number, a finite and positive integer or natural number which codes its instructions e.g.

313325311731133531117311133225311117311113353 17

A Turing machine with a given description number is capable of generating as many digits of a number as required.
For example, π which consists of an infinite sequence of digits can be computed to any desired number of digits by a Turing machine.
Normally, π is indicated as follows where the ellipsis … means there are infinitely more digits

$$\pi = 3.14159265358979323846264338327 95\ldots$$

But it is also possible to represent π by a finite integer, the description number of the Turing machine that can calculate the digits of π.
The description number is in a sense more fundamental because it describes the algorithm for calculating π.
By reducing each Turing machine to a number, Turing made it possible, in effect, to generate machines just by working through the set of natural numbers and assigning to each machine a natural number.
This process is called enumeration.
Not every natural number is a valid description number of a Turing machine or one that is capable of calculating an infinite sequence of digits but there are some and each of these corresponds to a computable number.

Key point

Computer programs:
Computer programs are just a sequence of numbers.

Single licence - Abingdon School

Information

When we enumerate we designate a first element, second element etc. The process is called enumeration, and we say that what we have enumerated is enumerable.

Conclusion: **computable numbers are enumerable.**

But the set of real numbers is not countable, i.e. they cannot be enumerated by assigning each a natural number 1, 2, 3, ….

Conclusion: virtually all **real numbers are not computable.**

Cartesian product of sets

Sometimes the order of elements in a collection is important, e.g. the x, y coordinates of points on an X-Y graph.

Sets are unordered so a different structure is needed to represent ordered collections. This is provided by ordered tuples. An ordered pair is a tuple (a, b) in which a is the first element and b the second. The order is important as the ordered pair (a, b) is not the same as the ordered pair (b, a).

The Cartesian product of two sets A and B is the set of all ordered pairs in which the first element of the pair comes from A and the second element of the pair comes from B.

The Cartesian product of two sets A and B written as $A \times B$ is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$.

In set comprehension form

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

For example, if $A = \{1, 2, 3\}$ and $B = \{2, 4, 6\}$

$$A \times B = \{(1, 2), (1, 4), (1, 6), (2, 2), (2, 4), (2, 6), (3, 2), (3, 4), (3, 6)\}$$

Key concept

Cartesian product of sets:
The Cartesian product of two sets A and B is the set of all ordered pairs in which the first element of the pair comes from A and the second element of the pair comes from B.
It is written as $A \times B$ where
$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$

Questions

8 Set A is the set of all students in school and set B is a set of all lockers that can be assigned to these students.
What does the Cartesian product $A \times B$ represent?

9 What is the Cartesian product of $A = \{a, b\}$ and $B = \{1, 2, 3\}$ when evaluated?

10 Set A is the set of all airlines in the world and B is the set of all airports. What is the Cartesian product $A \times B \times B$?

11 Set A is the set of all horizontal pixel coordinates and set B the set of all vertical pixel coordinates. What is the Cartesian product $A \times B$?

Single licence - Abingdon School

Subsets

A **subset is defined as a set that is wholly contained within another set.**
For example if A = {0, 1, 2, 3} and B = {1, 2} then B is a subset of A, and A is a superset of B.

The symbol ⊆ is used to mean 'is a subset of', so B ⊆ A reads 'B is a subset of A'.
Technically, a subset can have exactly the same elements as the superset, i.e. if C = {0, 1, 2, 3} and D = {0, 1, 2, 3} then C = D and C ⊆ D are both true.

If we remove the line from ⊆ we get ⊂ which is the symbol for 'is a proper subset of'. A proper subset is defined as one where at least one element in the superset is not in the subset. For example, {0, 1, 2} ⊂ {0, 1, 2, 3} is true because 3 is not in {0, 1, 2} and {0, 1, 2, 3} is a proper subset of ℕ because there is at least one element in ℕ that is not in {0, 1, 2, 3}.

Information

The power set of set *S* is the set of all subsets of the set *S*. The power set of *S* is denoted by *P*(*S*). For example, the power set of the set {0, 1, 2} is

P({0, 1, 2}) = {∅, {0}, {1}, {2}, {0, 1}, {0, 2}, {1, 2}, {0, 1, 2}}

Set operations

If an object x is a **member** (element) of a set A, then we write x ∈ A. If it is not a member then we write x ∉ A.
For example, 3 ∈ {1, 3, 6} is true but 4 ∈ {1, 3, 6} is not.
4 ∉ {1, 3, 6} is true but 3 ∉ {1, 3, 6} is not.

The **union** of the sets A and B, denoted by A ∪ B, is the set that contains those elements that are either in A or in B, or in both.
For example, {1, 3, 6} ∪ {2, 6} = {1, 2, 3, 6}.

The **intersection** of two sets A and B, denoted by A ∩ B, is the set containing those elements in both A and B.
For example, {1, 3, 6} ∩ {2, 6} = {6}.

The **difference** of sets A and B, denoted by A – B, is the set containing those elements that are in A but not in B.

For example, {1, 3, 6} – {2, 6} = {1, 3}.

Information

The symbol \ or the symbol ~ can also be used to denote set difference, e.g. A\B, A ~ B.

Questions

12 Determine whether these statements are true or false.
(a) 8 ∈ {2, 4, 6, 8, 10} (b) {2, 4} ⊂ {2, 4, 6, 8, 10} (c) {2, 4} ⊂ {2, 4} (d) {2, 4} ⊆ {2, 4}
(e) {4} ⊆ {2, 4}

13 Evaluate the following
(a) {2, 4} ∪ {1, 3} (b) {1, 2, 3} ∪ {2, 4} (c) {1, 2, 3} ∩ {2, 4} (d) {1, 2, 3} ∩ { 4, 6}
(e) {2, 4, 6, 8} – {2, 6} (f) {2, 4} – {2, 4}

Single licence - Abingdon School

In this chapter you have covered:

- The concept of a set and notations for specifying a set
- The compact representation of a set
- The concept of:
  - finite sets
  - infinite sets
  - countably infinite sets
  - cardinality of a finite set
  - Cartesian product of sets
- The meaning of the term:
  - subset
  - proper subset
  - countable set
- The set operations:
  - membership
  - union
  - intersection
  - difference

Single licence - Abingdon School

4 Theory of computation

4.2 Regular languages

Learning objectives:

- *Know that a regular expression is simply a way of describing a set and that regular expressions allow particular types of languages to be described in a convenient shorthand notation*
- *Be able to form and use simple regular expressions for string manipulation and matching*
- *Be able to describe the relationship between regular expressions and FSMs*
- *Be able to write a regular expression to recognise the same language as a given FSM and vice versa.*

4.2.3 Regular expressions

Regular expressions - a way of describing a set

Figure 4.2.3.1 shows a pencil with a lead hardness of HB. HB is a string of the language L_p used to describe the hardness of the lead used in pencils (the "lead" in pencils is actually graphite).

Figure 4.2.3.1 Pencil with lead hardness HB

The **alphabet** of a subset of the language L_p is the set of symbols {H, B, 2, 3, 4}. Traditionally the symbol Σ is used to stand for an alphabet.

The **valid strings in the language L_p** form the set

{4B, 3B, 2B, B, HB, H, 4H, 3H, 2H}

which is constructed from symbols of the alphabet {H, B, 2, 3, 4}.

A string over an alphabet is a sequence of symbols of the alphabet.

It is possible to make other languages from the alphabet {H, B, 2, 3, 4}. For example, the language L_x is defined by the set

{4BHH, 3BHH, 2BHH, BH, HBB, HB, 4HB, 3HB, 2HB}

In fact, there is one set for a given alphabet which contains all strings that may be created from the symbols of the given alphabet including the empty string. We use Σ* to stand for this set.

The symbol * means "zero or more things drawn from...".

Since symbols may be repeated and since strings may be arbitrary long, Σ* is necessarily an **infinite set**, unless Σ is itself an empty alphabet.

If the alphabet Σ = {H, B, 2, 3, 4}, then Σ* is the set of all possible strings over Σ.

Let's use the symbol L for the language Σ* then L_p and L_x above are language subsets of L, i.e. $L_p \subset L$ and $L_x \subset L$.

In general, a language over Σ is any subset of Σ*.

Table 4.2.3.1 shows three alphabets, {a, b, c}, {α, β, γ}, {☺😐☹} and some possible strings over these alphabets. The special symbol ε denotes the **empty string**.

alphabet	{a, b, c}	{α, β, γ}	{☺😐☹}
some possible strings over the given alphabets	aaabaabbcaa	α α β α α α γ	😐😐😐
	bcbbbc	ε	😐☺☺☹☺☺☺
	ccc	β β β γ γ α	ε
	ε	γ γ γ γ	☺😐😐☺

Table 4.2.3.1 Some strings for given alphabets

Key concept

Set:
A set is an unordered collection of **different** things, e.g. set of all the students in a classroom.

Sequence:
A sequence is simply an ordered collection of things, e.g. the sequence of digits in your mobile phone number.
Unlike a set, a sequence does not have the property that all the things in it are necessarily different. For example, many mobile phone numbers contain some digit more than once.

Single licence - Abingdon School

Questions

1. Given an alphabet Σ = {0, 1}, if L_1 is the language of strings of size 3 over Σ, complete *Table 4.2.3.2* to show all strings of this language?

2. Given an alphabet Σ = {0, 1}, what are the strings of L_2 if L_2 is the language of strings of size 3 which are palindromes (strings which are the same backwards and forwards)?

Valid string
000

Table 4.2.3.2 All valid strings of size 3 over the alphabet {0, 1}

Key concept

Regular expression:
A regular expression is an expression that specifies in rules the valid strings of a language. A language is a set of strings over some alphabet therefore a regular expression is a description of a set of strings over some alphabet.

So far, we have considered only simple languages, e.g. pencil hardness, binary strings of size 3, but other strings such as "sales@educational-computing.co.uk" also belong to a language.

How can we define the language consisting of strings which are email addresses?

It is not feasible to write down every valid email addresses, it is an enormous set so we turn to an alternative rule-based approach called a **regular expression**. A regular expression is an expression that specifies in rules the valid strings of a language as shown in *Table 4.2.3.3*.

A language is a set of strings over some alphabet therefore **a regular expression is a description of a set of strings over some alphabet**.

Regular expression	Description
∅	Denotes the set/language of no strings, i.e. {} the empty set.
ε	Denotes the set {ε}, i.e. the set/language containing only the empty string.
a, where a ∈ Σ	Denotes the set {a}, i.e. the set/language over alphabet Σ and containing only the string with only the symbol a.
r|*s* where *r* and *s* are regular expressions	Denotes the set/language *r* ∪ *s* (*r* and *s* each denote a set, ∪ denotes set union), e.g. a|b|c represents the language {a, b, c}.
rs where *r* and *s* are regular expressions	Denotes the set/language of strings each composed of a string from *r* concatenated with a string from *s*. Formally, $\{RS \mid R \in r \text{ and } S \in s\}$. For example, (a|b)c(d| ε) represents the language {acd, ac, bcd, bc}.
*r** where *r* is a regular expression	Denotes the set/language of strings composed of the concatenation of zero or more strings, each from *r*. For example, a(b)*c represents the language {ac, abc, abbc, abbbc, abbbc, ...}.

Table 4.2.3.3 Regular expressions

Single licence - Abingdon School

Regular expressions use parentheses (brackets) for grouping things, e.g. (a|b)c(d|ε). This regular expression means either an a or b followed by c followed by d or nothing (ε).

000(0|1)*111

Figure 4.2.3.2 Regular expression describing a set of strings over the alphabet {0, 1}

Figure 4.2.3.2 shows a regular expression which describes a set of strings over the alphabet {0, 1}.

This set describes a language of strings beginning with three 0s, ending with three 1s, and having any combination of 0s and 1s in the middle or no combination at all.

For example, 000111 is a valid string in this language because (0|1)* means zero or more 0*s* and 1*s*. *Table 4.2.3.4* shows some more valid strings in this language.

Valid string
0000111
00010111
00000111
0001010111
000111100111
00011110011111

*Table 4.2.3.4 Some valid strings over the alphabet {0, 1} for the language described by regular expression 000(0|1)*111*

Questions

3 The regular expression 001(0|1)*0 describes a language of strings over the alphabet {0, 1}.
State whether the following strings are valid strings in this language
(a) 0010 (b) 00110 (c) 00011010 (d) 0010101

Extensions to regular expressions

There are several extensions to regular expressions that are often used because they can make a regular expression more concise than it would be otherwise.

For example, r+ which means "one or more occurrences of r" can be substituted for rr*.

Table 4.2.3.5 shows more abbreviations and their equivalent regular expressions which do not use abbreviations. The table shows that these abbreviations do not extend regular expressions in terms of what is possible to define since any language defined using these abbreviations can also be defined without using them. The abbreviations just add notational convenience. The square bracket notation defines a symbol class.

Abbreviation	Meaning	Equivalence	Example
[abc]	One occurrence of any of these symbols	(a|b|c)	
[a-c]	One occurrence of any symbol from this range	(a|b|c)	
r?	An optional occurrence of a string defined by *r*	*r*|ε	a?
r {*n*}	Exactly *n* occurrences of a string defined by *r*	*rrr* (if *n* = 3)	a {3} matches aaa
r {*n*,}	*n* or more occurrences of a string defined by *r*	*rrrr** (if *n* = 3)	a {3,} = aaaa*
r {*n*, *m*}	Between *n* and *m* occurrences of a string defined by *r*	*r* {*n*}| *r* {*n* + 1}|...| *r* {*m*}	a {3,5}= aaa|aaaa|aaaaa
r+	One or more occurrences of a string defined by *r*	*rr**	a+

Table 4.2.3.5 Extensions to regular expressions created to offer greater conciseness

Single licence - Abingdon School

Metasymbols

The symbol - that appears in the abbreviation [a-c] is known as a **metasymbol** or **metacharacter**. Other metacharacters include +, ?, *, [,] and |.

The - symbol is used in [a-c] to mean a range of symbols, i.e. a or b or c. The equivalent regular expression is (a|b|c). The symbol - is excluded from the matching process when it acts as a metasymbol.

A metasymbol or metacharacter is a symbol that is not used for matching purposes but instead is used as notation with special meaning, e.g. to indicate a range. Symbols that normally function as metasymbols are preceded by a backslash (\) when they need to be interpreted literally for matching purposes. For example, the regular expression a*b will match the symbol string a*b but the regular expression a*b will not. The regular expression a*b matches, for example, aaab.

Predefined shorthand classes

A predefined symbol/character class defines a set of symbols/characters, any one of which can occur in an input string for a match to succeed. They are a convenient shorthand without which a regular expression would be much longer. For example, the predefined class \d matches any single numeric character or symbol. The predefined class \s matches a single space. With predefined classes, the metasymbol \ functions to escape the normal meaning of the following symbol, e.g. \d means treat d as a metasymbol not as the literal symbol d.

Interpreting a regular expression

To interpret a regular expression requires knowledge of the order in which to apply the metasymbols * + ? |

In arithmetic, we need to know whether 5 + 2 x 6 should be interpreted as 5 + (2 x 6) or (5 + 2) x 6.

In arithmetic, we use operator precedence rules and brackets to override these rules.

In regular expressions, precedence rules apply and brackets are used to override these rules.

Regular expression precedence rules:

- *, + and ? are always done first
- Concatenation is done next (e.g. abc)
- | comes last.

Key concept

Metasymbol:

A metasymbol or metacharacter is a symbol that is not used for matching purposes but instead is used as notation with special meaning, e.g. to indicate a range.

Symbols that normally function as metasymbols are preceded by a backslash (\) when they need to be interpreted literally for matching purposes. This is called escaping the symbol, e.g. *

Information

Online regular expression tester:

An online regular expression tester is available to use at https://regex101.com/.

Questions

4 The regular expression (ab|cd)(e|f)*(g|h|ε) describes a language of strings over the alphabet {a, b, c, d, e, f, g, h}.
State whether the following strings are valid strings in this language
(a) ab (b) abeeh (c) cdffef (d) bcffef (e) cdeeefefgh

5 A regular expression for defining the language of DNA sequences can be expressed as `[ACGT]+`
Rewrite this regular expression in a form which doesn't use square brackets.

6 The case insensitive form of the above regular expression for defining the language of DNA sequences is expressed as follows `[ACGTacgt]+`
Rewrite this regular expression in a form which doesn't use square brackets.

Single licence - Abingdon School

Questions

7 The rules for forming identifiers in a particular programming language can be expressed in a regular expression as `[A-Za-z][A-Za-z0-9_]*`

(a) State this rule in words.

(b) State whether the following strings are valid identifiers in this language

(i) Form1 (ii) outer_loop (iii) _unknown (iv) anon_

8 A regular expression for dates is as follows

`([1-9]|(1|2)[0-9]|(30|31))/(1[0-2]|[1-9])/[1-9][0-9]*`

(a) Check that the following dates are valid strings according to this regular expression

(i) 20/11/2 (ii) 30/8/2016 (iii) 0/0/0 (iv) 13/12\2017

(b) The set of dates that this regular expression describes contains semantically invalid dates, i.e. dates that do not appear in the Gregorian calendar, the one in use today. Give one example of a semantically invalid date that appears in the set of dates.

9 Explain why

(a) `a\+b` will match `a+b` but not `a\+b`

(b) `a\\b` will match `a\b` but not `a\\b`

(c) [a-d] will not match - but [a\-d] will

Worked example

The regular expression for valid strings in a language is `ab*c+de?fg|d`

This will match, for example, abbcccdfg and d because `ab*c+de?fg|d` means either "a single a followed by zero or more bs followed by one or more cs followed by a single d and zero or one es and then fg" or "a single d".

Table 4.2.3.6 shows a summary of useful metasymbols.

Metasymbol	Meaning
-	Used to indicate a range, e.g. [0-9] means 0 or 1 or ... or 8 or 9
*	Used to indicate zero or more occurrences, e.g. a* means the empty string or a or aa or aaa or ...
?	Used to indicate zero or one occurrences, e.g. a? means the empty string or a
+	Used to indicate one or more occurrences, e.g. a+ means a or aa or aaa or ...
.	The dot matches any single symbol except a symbol which represents newline, e.g. .an matches aan, ban, can, dan, ...
\	Used to escape the metasymbol interpretation so symbol can be interpreted literally, e.g. \+
|	Used to indicate an alternative, e.g. a|b matches a or b
(and)	Parentheses (brackets) are used to group things, e.g. (a|b)c(d|ε)
[and]	Square brackets are used to abbreviate regular expressions, e.g. [a-d] means a|b|c|d
^	Typing a caret after an opening square bracket will negate the symbol class, e.g. [^A-Z] means will match a single occurrence of a symbol which isn't from this range.

Table 4.2.3.6 Metasymbols or metacharacters and their meaning

Information

Regular expression tester:
If you want to install a regular expression tester then one which is suitable, The Regex Coach, may be downloaded from http://www.weitz.de/regex-coach/.

Single licence - Abingdon School

Worked example

The regular expression for a real number is `[+\-]?\d+(\.\d+)?`

This is because `[+\-]` matches a single character which is `+` or `-`.

The `\` is used because `-` is a metacharacter. `[+\-]?` matches zero or one character which is `+` or `-`.

`\d` matches any digit. `\d+` matches one or more digits.

`\.` matches the decimal point (`\` is used to escape the metacharacter meaning of '.', which is match any character).

`(\.\d+)` means match a decimal point followed by one or more digits.

`(\.\d+)?` means no instance or a single instance of a decimal point followed by one or more digits.

Questions

10 The regular expression `[+\-]?\d+` describes a set of numbers. Are the following numbers members of this set and why?

(a) `431` (b) `-6273` (c) `+19274` (d) `+`

11 The regular expression `[+\-]?\d+(\.\d+)?` describes a set of numbers. Are the following numbers members of this set and why?

(a) `431` (b) `-6273.4` (c) `+19274.385` (d) `+45.4d5`

12 Explain why mary@quitecontrary.co.uk is a member of the set described by the regular expression

`[_a-zA-Z\d\-\.]+@([_a-zA-Z\d\-]+(\.[_a-zA-Z\d\-]+)+)`

but er!@;#£M is not.

Forming and using simple regular expressions for string manipulation and matching

Pattern matching

Searching text is an important computing task, e.g. searching for one occurrence or all occurrences of some specific string in a large document. Searching relies on **pattern matching**.

For example, we might want to search for the word "needle" in a large word-processing document, or we might want to search for all occurrences of the string "GATCGGAATAG" in the human genome.

There is a need to check whether or not an e-mail address is formed correctly.

Pattern-matching is used to do this. For example, mary@educational-computing.co.uk looks like an e-mail address, but er!@;#£M does not.

Regular expressions (regex) offer a way to specify such patterns and also to solve the corresponding pattern-matching problem.

The regular expression [A-Za-z_][A-Za-z_0-9]* could be used to search for an identifier in a programming language.

Regular expressions are used extensively in operating systems for pattern matching in commands and when performing a search for files or folders.

String manipulation

The manipulation of text is an important computing task is, e.g. word-processing documents, DNA sequences, computer programs. Often we want to find a particular string and replace it with another string.

This means searching for a word in a block of text.

You can use a regular expression to find a word, even if it is misspelled. For example, if searching for the word separate (correct spelling), the regular expression sep[ae]r[ae]te would find separate, seperate, seperete and separete.

Single licence - Abingdon School

Other applications are scanning for virus signatures, search and replace in word processors, searching for information using search engines like Google, filtering text (spam, malware, firewall traffic), validating data entry fields (e-mail, dates, URLs, debit and credit card numbers).

Questions

13 State which of the following strings match the regular expression `se[ea]n`
(a) `seen` (b) `sean` (c) `sen` (d) `sea`

14 State which of the following strings match the regular expression `le?ad`
(a) `led` (b) `lad` (c) `lead` (d) `le?ad`

15 State which of the following strings match the regular expression `10(1)*01`
(a) `1001` (b) `100101` (c) `10101` (d) `1011101`

16 State which of the following strings match the regular expression `10(1)+01`
(a) `1001` (b) `100101` (c) `10101` (d) `1011101`

17 State which of the following strings match the regular expression `\d+(\-|\s)\d+`
(a) `01296-433006` (b) `101296--433006` (c) `01793  2345678`
(d) `01793 2345678` (e) `017932345678`

18 The word licence has been spelled in the following ways in a piece of text: licence, license, lisence, lisense. Write a regular expression that will find all instances of the word licence spelt in these different ways.

19 The human FMR-1 gene has a region that repeats the sequence CGG a number of times. The number of repeats varies from person to person; a higher number of repeats is associated with fragile X syndrome, a genetic disease that causes mental retardation and other symptoms in 1 in 2000 children. The pattern seems to be bracketed by GCG and CTG. Searching for this region can be done with a regular expression. Write this regular expression.

Relationship between regular expressions and FSMs

A finite state machine (FSM) without outputs can be used to accept or reject finite strings of symbols. If the FSM without outputs produces a unique computation for each input string it is known as a deterministic finite automaton (DFA).

Deterministic refers to the uniqueness of the computation.

Figure 4.2.3.3 shows the state transition diagram for a deterministic finite automaton (DFA) which takes a finite sequence of 0s and 1s as input. For each state, there is a transition arrow (→) leading out to a next state for both 0 and 1. In the automaton, there are six states: S_0, S_1, S_2, S_3, S_4, and S_5 (denoted graphically by circles).

Upon reading a symbol, a DFA jumps deterministically from one state to another (which could be the same state) by following the transition arrow. For example, if the automaton is currently in state S_0 and the current input symbol is 0, then it deterministically jumps to state S_1. If the automaton is currently in state S_0 and the current input symbol is 1, then it deterministically jumps to state S_5.

Key concept

Deterministic finite-state automaton (DFA):
A finite state automaton is said to be deterministic if and only if its present state and the next symbol to be accepted uniquely determine the next state.

Single licence - Abingdon School

A DFA has a start state (denoted graphically by an arrow coming in from nowhere) where computations begin (state S_0 in *Figure 4.2.3.3*), and a set of accept states (each denoted graphically by a double circle) which help define when a computation is successful. In the state transition diagram in *Figure 4.2.3.3* there is only one accept state, S_4.

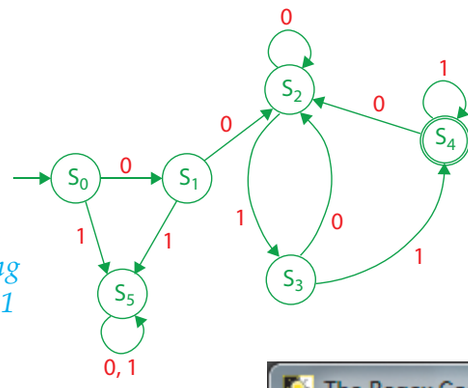


*Figure 4.2.3.3 State transition diagram for a DFA corresponding to regular expression 00(0|1)*11*

DFAs and regular expressions can each be used to define sets of strings, i.e. languages.

Any language that can be defined by a regular expression can be accepted by a DFA, and vice versa.

Figure 4.2.3.4 shows a partial trace of the regular expression 00(0|1)*11 applied to input string 0010011.

Figure 4.2.3.5 shows a finite state machine simulator (FSMSim) part way through accepting the input string 0010011. *Figure 4.2.3.6* shows the same finite state machine simulator (FSMSim) accepting the input string 0010011.

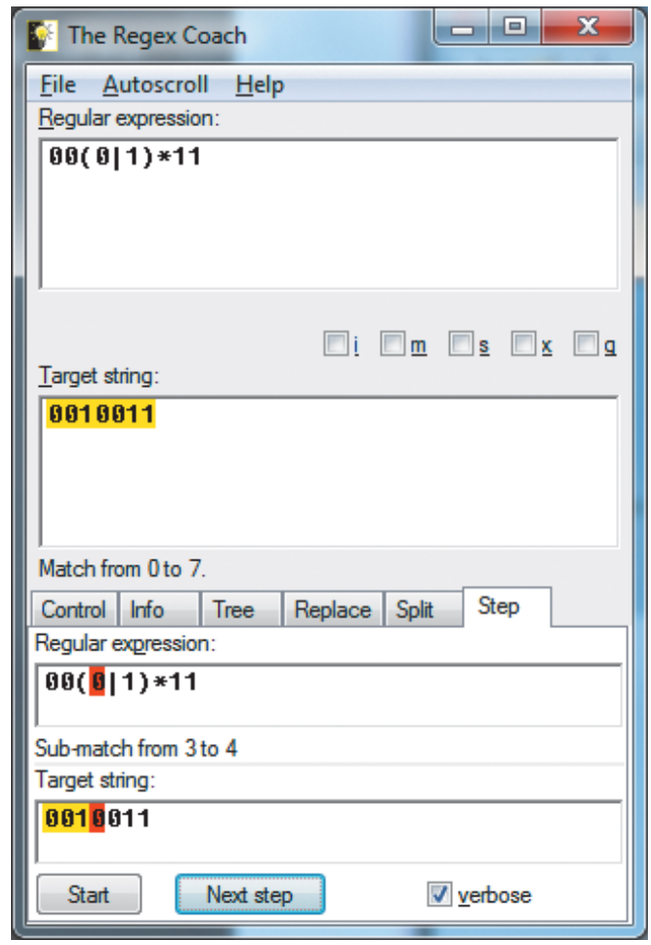


*Figure 4.2.3.4 Partial trace of regex 00(0|1)*11 for input string 0010011*

What range of languages can a DFA or its equivalent regular expression define?

It is not very likely that a DFA could be created which is able to determine whether a sentence expressed in English is a valid string or an expression in the programming language Haskell is a valid string. Languages such as English and Haskell are too complicated for DFA string checkers or regular expression string checkers. However, we can define DFAs and their equivalent regular expressions to check that an input string is a valid language identifier as defined in, say, the programming language Haskell.

A language that can be specified by a regular expression is called a **regular language**.

A regular language is also any language that a finite state machine (FSM)/finite state automaton (FSA) will accept.
Essentially if the language is definable by a regular expression then it will have an equivalent FSA/FSM.

Key concept

Regular language:
A language that can be specified by a regular expression is called a regular language.
A regular language is also a language that a finite state machine/finite state automaton will accept.

Single licence - Abingdon School

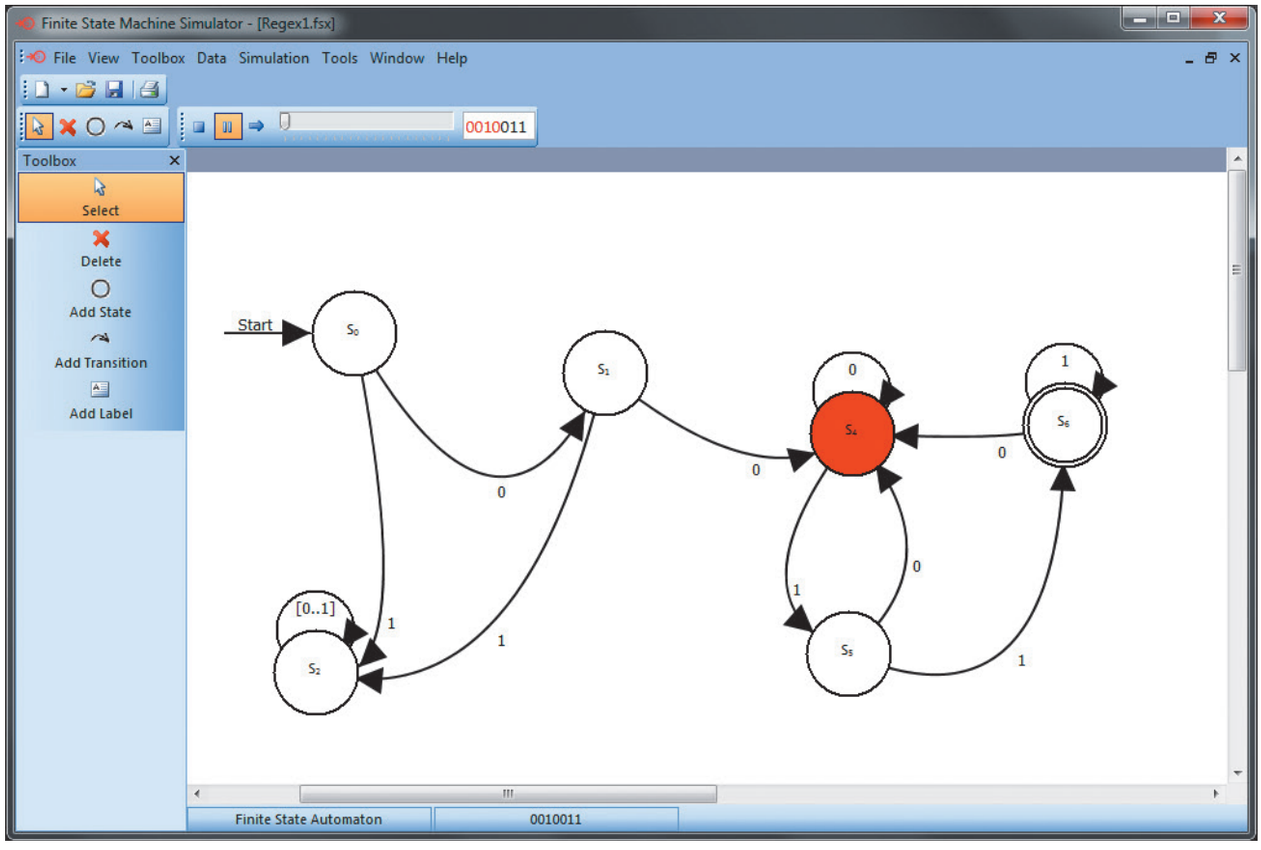


Figure 4.2.3.5 Finite state machine simulator part way through accepting 0010011

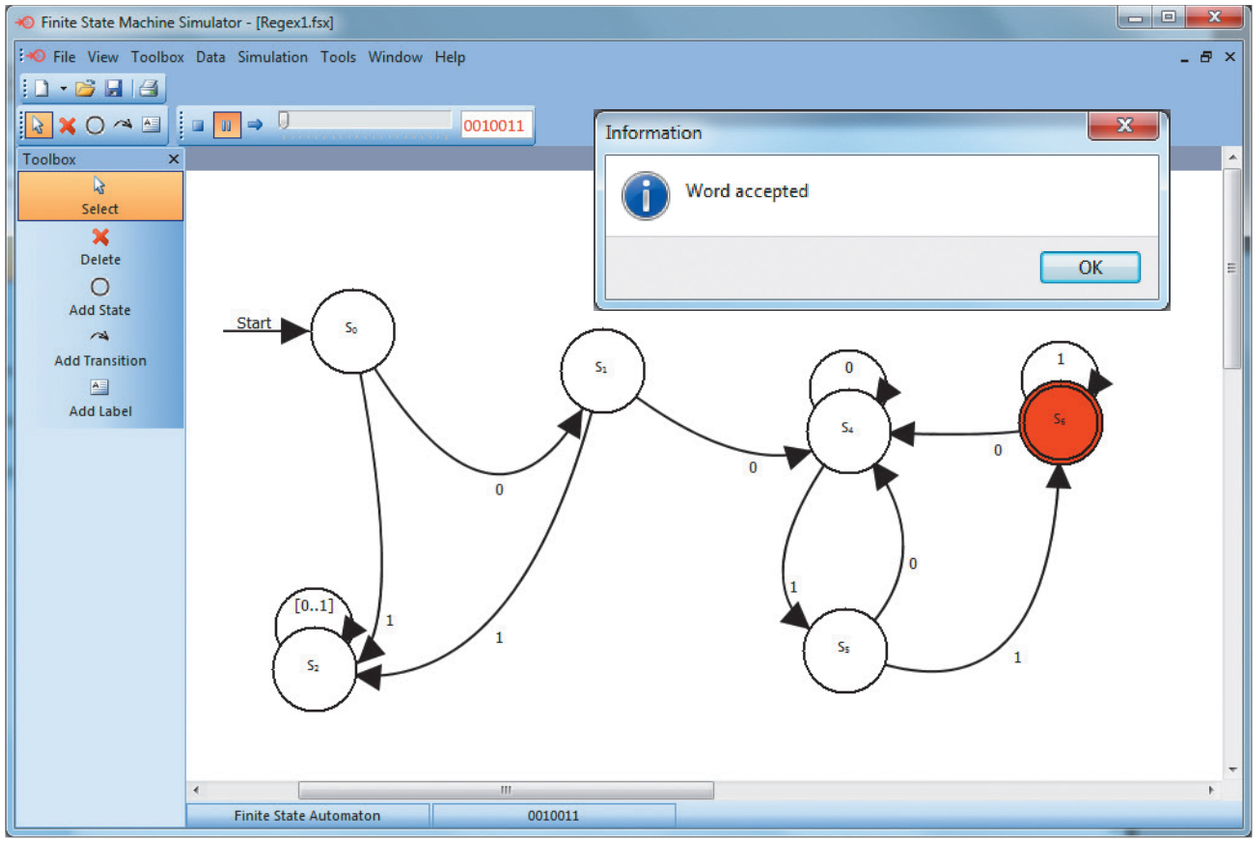


Figure 4.2.3.6 Finite state machine simulator accepting 0010011

Single licence - Abingdon School

Writing a regular expression to recognise the same language as a given FSM and vice versa

Regular expression to FSM (DFA)

It is possible to convert simple regular expressions to equivalent finite state machines fairly easily using intuition and a knowledge of four basic FSMs. These are the first four shown in *Figure 4.2.3.7*. These FSMs are deterministic finite state automata which means that they have a start state and one or more accepting states. If after processing the input string the DFA is in one of its accepting states then the input string is a valid string of the language.

The first regular expression describes a language with a single string A. The corresponding DFA accepts a single A as input. It transitions from the start state to the accepting state on receiving a single A.

The second regular expression describes a language with a string composed of an A followed by B. The corresponding DFA has three states, S_0, S_1 and S_2, and two transitions. It accepts an A followed by B.

The third regular expression describes a language with two strings one composed of an A the other a B. The corresponding DFA has two states and two transitions. The DFA accepts either an A or a B.

The fourth regular expression describes a language which consists of strings that belong to the set {ε, A, AA, AAA, AAAA, ...}, i.e. zero or more As. The corresponding DFA has one state which is both the start state and the accepting state.

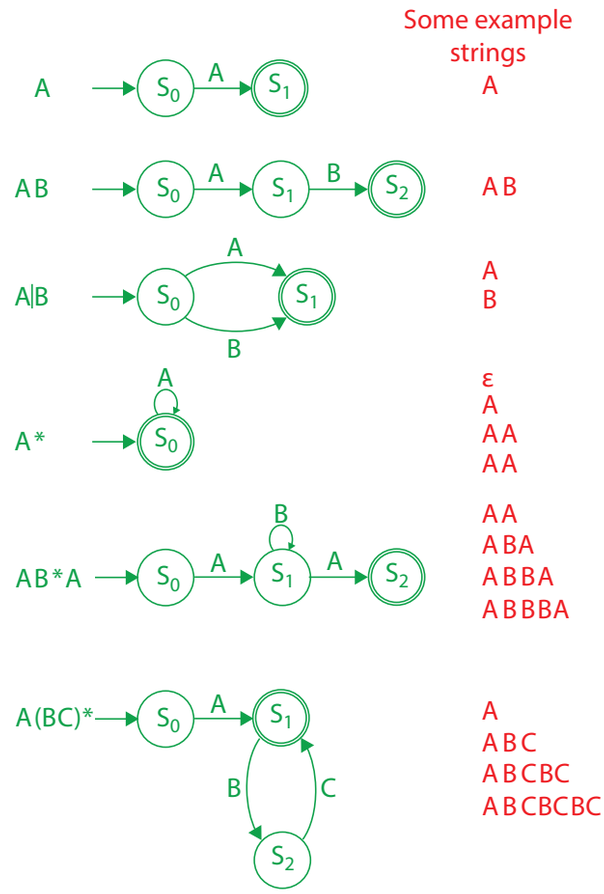


Figure 4.2.3.7 Regular expressions to FSM(DFA)

With these four basic regular expressions and their corresponding DFAs we can convert other regular expressions to their corresponding DFAs. Two examples are given in *Figure 4.2.3.7*, AB*A and A(BC)*.

The second example shows how to handle (BC)*.

More complicated regular expressions require a methodical approach, one that can be cast as an algorithm so that the conversion can be done by computer.

Task

1 Watch the YouTube video "Regular expression conversion to finite state automaton" https://www.youtube.com/watch?v=GwsU2LPs85U.

Questions

20 Convert the following regular expressions to DFAs

(a) `a|bc*` (b) `(ab|ac)*`

(You can check your own answer by using the regex to DFA tool at http://hackingoff.com/compilers/regular-expression-to-nfa-dfa)

Single licence - Abingdon School

FSM (DFA) to regular expression

The reverse process for simple DFAs can also be carried out intuitively.

Given any of the DFAs in *Figure 4.2.3.7* it is a question of working backwards reducing the number of states until left with the accepting state. For example, as shown in *Figure 4.2.3.8*.

Alternatively, it can be useful to first write down a list of strings that would be accepted by the DFA to see if any pattern emerges. It can also be useful to draw paths through the DFA in different coloured highlighter pens to see if any patterns emerge.

For example, as shown in *Figure 4.2.3.9*.

The patterns 00 and 11 are alternatives which can be preceded by (0|1)* and followed by (0|1)*.

This is the regular expression (0|1)*(00)|(11)(0|1)*.

We can reduce the DFA in *Figure 4.2.3.9* to that shown in *Figure 4.2.3.10*.

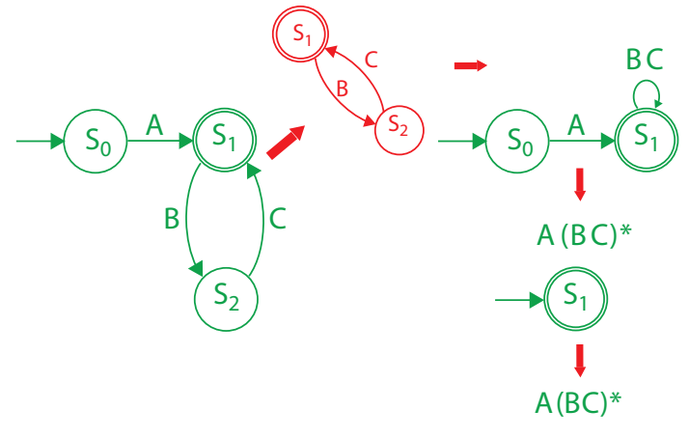


Figure 4.2.3.8 FSM (DFA) to regular expression

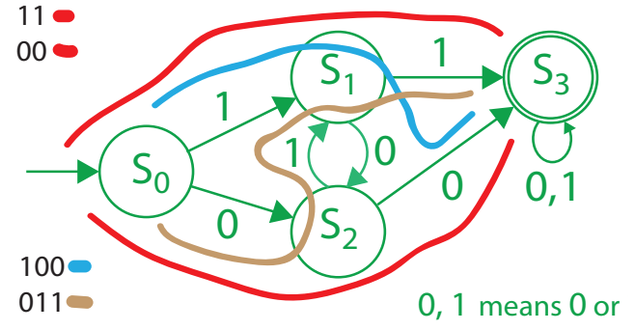


Figure 4.2.3.9 FSM (DFA) to regular expression

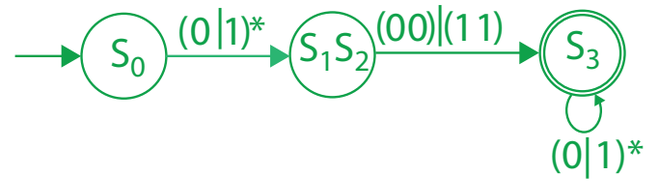
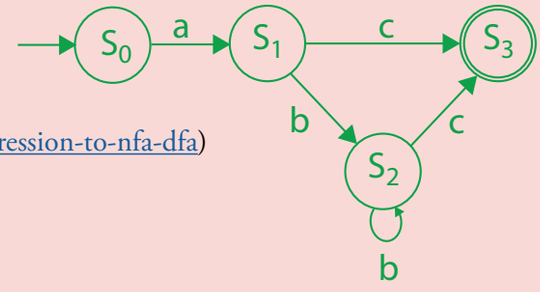


Figure 4.2.3.10 FSM (DFA) to regular expression

Questions

21 Convert the following DFA to regular expression



(You can check your own answer by using the regex to DFA tool at http://hackingoff.com/compilers/regular-expression-to-nfa-dfa)

In this chapter you have covered:

- Regular expressions which are simply a way of describing a set and which allow particular types of languages to be described in a convenient shorthand notation
- Forming and using simple regular expressions for string manipulation and matching
- The relationship between regular expressions and FSMs
- Writing a regular expression to recognise the same language as a given FSM and vice versa.

Single licence - Abingdon School

4 Theory of computation

4.3 Context-free languages

Learning objectives:

- *Be able to check language syntax by referring to BNF or syntax diagrams and formulate simple production rules*
- *Be able to explain why BNF can represent some languages that cannot be represented using regular expressions*

4.3.1 Backus-Naur Form (BNF)/syntax diagrams

Checking language syntax

Why regular expressions and DFAs cannot specify some languages

There are some languages which cannot be specified by a regular expression or Deterministic Finite-state Automaton (DFA). For example, the language of strings that begin with a certain **number** of 1s followed by the same **number** of 0s (including the empty string ε), i.e.

{ε, 10, 1100, 111000, 11110000, ...}

Regular expression

The regular expression 1*0* will accept every string in this language, but it will also accepts strings such as 100.

The reason is that regular expressions have no way of counting and no way of doing recursion.

Counting:

If regular expressions could count then they could first count the number of 1s, then the number of 0s, and compare.

Recursion:

If regular expressions could do recursion then they could test the string recursively, checking if the string had a 1 at the front, a 0 at the rear, and in the middle another shorter string belonging to the language.

DFA

Figure 4.3.1.1 shows a DFA which accepts the following strings:

ε. 10, 1100, 111000.

To handle longer strings in this language requires extending this DFA.

To handle arbitrarily long strings in this language, the DFA would need to be arbitrarily large - in which case it would no longer be finite - *Figure 4.3.1.2*.

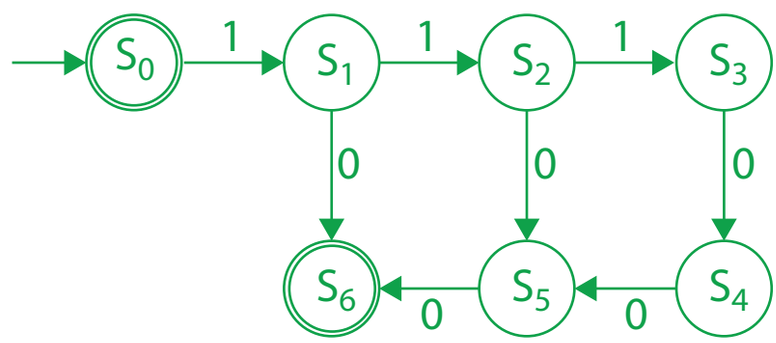


Figure 4.3.1.1 Deterministic finite state automata for ε, 10, 1100, 111000

Key fact

Why not all languages can be specified by regular expressions:

Regular expressions have no way of counting and no way of doing recursion and so cannot specify languages of strings such as

{ε, 10, 1100, 111000, 11110000, ...}.

Key term

DFA:

A DFA is a deterministic finite state automaton.

A DFA is a finite state machine (FSM) which either accepts an input string or rejects it.

To accept means that the string is a valid string in some language of strings defined by the DFA.

To reject means that the string is not a valid string in some language of strings defined by the DFA.

It is deterministic because there is only one path through the FSA that can be followed for a given input string (i.e. no options).

Single licence - Abingdon School

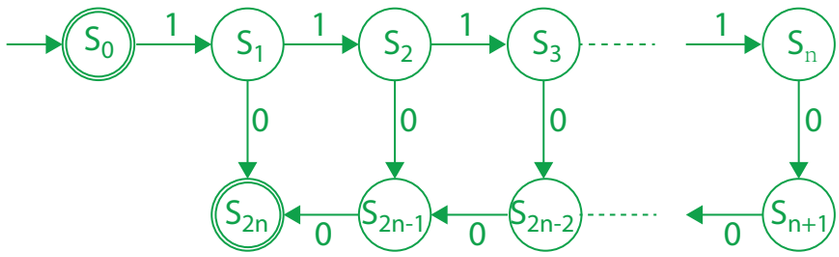


Figure 4.3.1.2 Deterministic finite state automata for an arbitrary number n 1s and n 0s

Languages which are not regular

The following notation and its interpretation is able to define the language of strings made up of a series of 1s followed by the same number of 0s

```
<string> ::= ε | 1 <string> 0
```

The term `<string>` ranges over strings of the language.
The notation means that `<string>` can be either the empty string, ε , or a 1 followed by another string in the language followed by a 0.

The notation actually expresses a rule for specifying valid strings in the language. This rule is called a **production** or **production rule**.
This form of specifying a language is called a **context-free grammar (CFG)**. The grammar is composed of productions in the form S ::= E or S → E.
A production specifies that a nonterminal symbol S can be replaced by the syntax expression E regardless of the context in which S occurs.

The notation

- The symbol ::= means 'consists of' or 'is defined as' or 'is composed of'.
  The symbol ::= may be replaced by a right-directed arrow →.
- The phrases enclosed in angle brackets are called **nonterminal symbols**. Nonterminal symbols are not a literal part of the language but something which can be expanded into a string or substring of the language.
- ε, 1 and 0 on the right-hand side are examples of what are called **terminal symbols**. Terminal symbols are symbols from the alphabet of the language.
- The vertical bar | separates alternatives.

We can see how this rule can generate say, the string 11110000, by using a tree with each `<string>` having children which are either ε or `1<string>0` as shown in *Figure 4.3.1.3*. We then form the string by joining the leaves together, i.e. 1111 is joined to 0000.

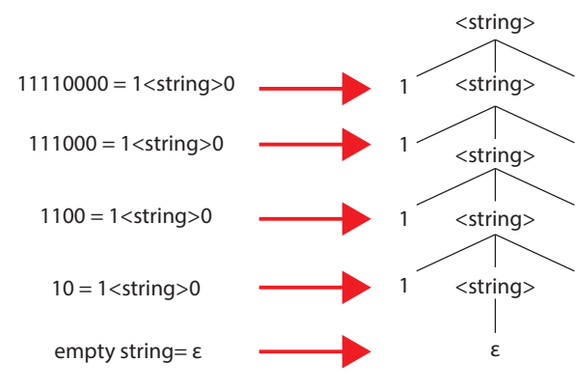


Figure 4.3.1.3 Generating 11110000 from rule by using a parse tree

A tree showing how nonterminals are expanded into a string is called a **parse tree**.

Key fact

Why not all languages can be specified by DFAs:

{ε, 10, 1100, 111000, 11110000, ...}.

cannot be specified by a DFA because it would have to handle arbitrarily long strings in this language. Such a DFA would need to be arbitrarily large to do this - in which case it would no longer be finite.

Information

Context-free grammar (CFG):
A grammar composed of productions in the form S ::= E or S → E is a context-free grammar
The concept of a CFG is very old. An ancient grammarian named Panini used essentially the same idea to describe the grammar of Sanskrit in the fourth century BC.

Single licence - Abingdon School

Backus-Naur Form (BNF)

The notation which was used in the previous section for defining productions (production rules), i.e. a context-free grammar, is known as **Backus-Naur Form** or **BNF**.

In *Chapter 4.2.3* we saw two ways to describe patterns - regular expressions and finite state automata.

We learned that regular expressions and finite state automata are equivalent.

Let's see how BNF can be used to write rules (productions) for the following regular expressions:

Regular expressions:

number = digit+

digit = [0-9]

A number consists of one or more digits and a digit is either 0 or 1 or ... 8 or 9.

The equivalent productions:

<number> → <digit> | <digit><number >

<digit> → 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Key fact

Parse tree:

A parse tree is a tree which shows how a particular sequence of tokens can be derived under the rules of a context-free grammar.

The start symbol is the root of the parse tree.

The leaves of the tree are its yield or terminals.

Each internal node, together with its children, represents the use of a production.

Suppose we have what we think is a number, e.g. 35, and we want to check if this number can be generated by the above productions, here is how to proceed:

- One of the nonterminals must be designated the **start symbol**. It is usually the one on the left-hand side of the first production. In this example it is <number>. The start symbol is also the root of the **parse tree**.
- We now replace <number> by the right-hand side of the production. We choose between <digit> and <number><digit>. Clearly 35 does not match any of the patterns defined by <digit> so we choose <digit><number>.
- Now we have two nonterminals to match, <digit> and <number>. The second, <number>, matches the left-hand side of the first production, <number>. We replace <number> by the right-hand side of this production choosing <digit>. So <digit><number> becomes <digit><digit>.
- We now match the first <digit> with the left-hand side of the production <digit> and replace <digit> by 3.
- We next match the second <digit> with the left-hand side of the production <digit> and replace <digit> by 5.
- The nonterminals <digit><digit> now become the number 35 and we have achieved our goal.

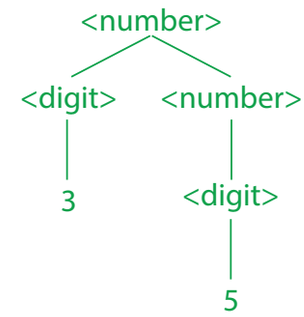


Figure 4.3.1.4 Parse tree for number 35

Figure 4.3.1.4 shows the parse tree for this example.

Questions

1 *Table 4.3.1.1* shows production rules used to define lead hardness of pencils.

(a) Name the notation method used in *Table 4.3.1.1* for production rules.

Production rules
<lead hardness> ::= HB | <scale of hardness> | <simple hardness>
<scale of hardness> ::= <numeric value><simple hardness>
<numeric value> ::= 2 | 3 | 4
<simple hardness> ::= H | B

Table 4.3.1.1 Production rules for pencil hardness

(b) Complete *Table 4.3.1.2* by writing **Yes** or **No** in the empty column to indicate whether or not the symbol strings are valid pencil lead hardnesses according to the production rules from *Table 4.3.1.1*.

Symbol string	Valid (Yes/No)
3B	
3HB	

Table 4.3.1.2 Symbol strings

Single licence - Abingdon School

Questions

2 *Table 4.3.1.3* shows production rules used to define arithmetic expressions, such as (6 + 7) x 4.

Production rules
<expression> ::= <number>| (<expression>) | <expression> + <expression> | <expression> - <expression> | <expression> x <expression> | <expression> / <expression>
<number ::= <digit> | <digit><number>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

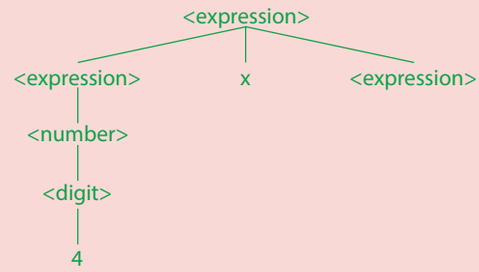
Table 4.3.1.3 Production rules for arithmetic expressions

(a) Complete *Table 4.3.1.4* by writing **Yes** or **No** in the empty column to indicate whether or not the symbol strings are valid arithmetic expressions according to the production rules from *Table 4.3.1.3*.

Symbol string	Valid (Yes/No)
12	
35.45	
(6 + 7) x 4	

Table 4.3.1.4 Symbol strings

(b) A parse tree can be used to demonstrate that an arithmetic expression is valid. Complete the parse tree below to show that 4 x (7 - 15) is a valid expression.



3 The production rules in *Table 4.3.1.5* define the syntax of a number of programming language constructs.

Production rules
<assignment_statement> ::= <identifier> ←<expression>
<expression> ::= <identifier>|<integer>|<expression>∗<expression>|<expression>+<expression>|<expression>−<expression>
<identifier> ::= <letter> | <letter> <string>
<string> ::= <character> | <character> <string>
<integer> :::= <digit> | <digit> <integer>
<character> ::= <digit> | <letter>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
A <letter> is any alphabetic character from a to z or A to Z.

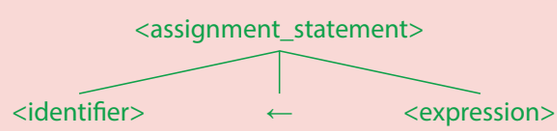
Table 4.3.1.5 Production rules for some programming language constructs

(a) *Table 4.3.1.6* contains identifier names. Complete *Table 4.3.1.6* by writing **Yes** or **No** in the empty column to indicate if the stated identifier name is a valid <identifier> for the production rules in *Table 4.3.1.5*.

Symbol string	Valid (Yes/No)
3Loop	
Test6	
Loop1Test	

Table 4.3.1.6 Symbol strings

(b) Complete the parse tree below to show that T1 ← R * 6 is a valid assignment statement.



Single licence - Abingdon School

Context-free grammar (CFG)

A **syntax** or **grammar** defines how strings in the language are combined to form **language structures** such as identifiers, expressions, program statements, etc.

The syntax, or grammar, of a language can be expressed in the form of structuring rules.

A context-free grammar (CFG) is one that defines a language using BNF productions of the form $S \rightarrow x$ which indicate that the concept symbolised by S can be expanded by the rule x.

How complicated the syntax, and therefore how expressive it can be, depends on the kinds of rules that are used. We have seen that strings that represent numbers may be defined by a regular expression and by BNF. We have also seen that strings that belong to the set {10, 1100, 111000, ...} cannot be defined by a regular expression but can be defined by BNF.

Why BNF can represent some languages which regular expressions cannot

A regular language is one that can be specified by a regular expression.

A language is called a context-free language if a context-free grammar (CFG) exists that can generate the language.

Any regular language is context-free since it is always possible to define a context-free grammar for it, i.e. define it using BNF (see the previous sections).

However, **not all context-free languages are regular**.

We have seen in the first two sections of this chapter a CFG that generates a language that no regular expression can specify and no DFA can accept, i.e.

`<string> ::= ε | 1<string>0.`

The basic building blocks of programs (e.g. language keywords such as For, While, identifiers, literal constants such as 45.6, etc) can be constructed from individual characters using three kinds of formal rules:

1. Concatenation
2. Alternation (choice among a finite set of alternatives)
3. Repetition an arbitrary number of times.

We have seen earlier in the chapter that the regular expressions

number = digit+

digit = [0-9]

can also be defined using BNF as follows

<number> → <digit> | <digit><number >

<digit> → 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

However, the following rule in BNF could not be rewritten as a regular expression because the right-hand side contains matched brackets:

<expression> ::= <number>| (<expression>) | <expression> + <expression>

Therefore, to specify the syntax of such a language requires one additional kind of rule:

4. Recursion (creation of a construct from simpler instances of the same construct).

Information

Chomsky's classification:
Regular languages are Chomsky Type 3 languages. They form a restricted subset of the Type 2 context-free languages.

Information

Lexical analyser or scanner:
Flex is a modern version of Lex (that was coded by Eric Schmidt, ex-CEO of Google) for generating C lexical analysers (scanners). Lex is a computer program that generates lexical analyzers ("scanners" or "lexers").

Information

Parser:
Yacc is a computer program for the Unix operating system. It is a parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, It is based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF).
Bison, which is more recent, is a derivative of Yacc.

Information

The lexer produces an array of tokens; the parser produces a tree of nodes. This tree of nodes will be how the language represents the program in memory.

Single licence - Abingdon School

Programming languages use nested brackets and other nested constructs, e.g.

```
for (int a = 10; a < 20; a = a + 1)
  {
    for (int b = 20; b < 30; b = b + 1)
      {
        .......
      }
  }
```

All four kinds of rules (1..4 above) are needed to define the syntax of a programming language.

We use BNF to define a context-free grammar for a programming language.

However, we can use a DFA to recognise the tokens of a programming language. The tokens are the building blocks referred to earlier, e.g. identifiers, keywords, literal constants, etc.
Tokens are strings that can be defined in terms of the first three rules. They form a regular set, or regular language. Regular sets are generated by regular expressions and can be recognised by scanners. A scanner is a concrete realisation of a finite state automaton used in the lexical analysis phase of compilers to identify tokens.

We use a parser to recognise and check the structure of a program. A parser generates a parse tree.

Questions

4 What is meant by a *syntax* or *grammar*?

5 Write a context-free grammar in BNF for each of the following regular expressions
(a) (a | b)* b (b) a* | b* |(ab)*

6 Give **one** language example that can be described by a grammar, but not by any regular expression. Justify your choice.

Syntax diagrams

Syntax diagrams are an alternative way of defining the syntax of a language. Suppose that we need to define a subset of the English language that includes the following sentence.

The cat caught the mouse.

We consider a sentence to be composed of a *subject*, "The cat", followed by a *verb*, "caught", followed by an *object*, "the mouse", followed by a *full stop* as shown in *Figure 4.3.1.5*.

Each of the labelled rectangular containers, *subject*, *verb*, and *object* in *Figure 4.3.1.5* are nonterminal symbols in BNF as is the label *sentence*. The nonterminals *subject*, *verb* and *object* are replaced as shown in *Figure 4.3.1.6*.
The oval shaped container in *Figure 4.3.1.5* requires no further expansion as it represents a terminal symbol, the full stop, in BNF and so does not need replacing.

Figure 4.3.1.5 A sentence is composed of a subject followed by a verb, followed by an object, followed by a full stop

Single licence - Abingdon School

The nonterminals referenced in the syntax diagrams are

sentence, *subject*, *object*, *adjective*, *noun*, *verb*

The one syntax diagram in *Figure 4.3.1.5* and the five syntax diagrams in *Figure 4.3.1.6* are labelled on the left with the nonterminals.

Each syntax diagram has one entry point and one exit point, and describes possible paths between these two. A string is valid in a language if there is a route for it from the entry point to the exit point.

Terminals are represented by oval containers and nonterminals by rectangular containers.

The terminals in *Figure 4.3.1.6* are

caught, ate, saw, killed, a, the, A, The, cat, mouse, bird, fly, fat, thin, small, black

Tasks

Check whether these sentences are valid sentences according to the syntax defined by *Figure 4.3.1.6*:

1. A mouse ate the cat.
2. A small bird caught the thin mouse.
3. The black cat saw the fly.
4. The fat bird ate fly.
5. A bird saw a fly and a mouse.

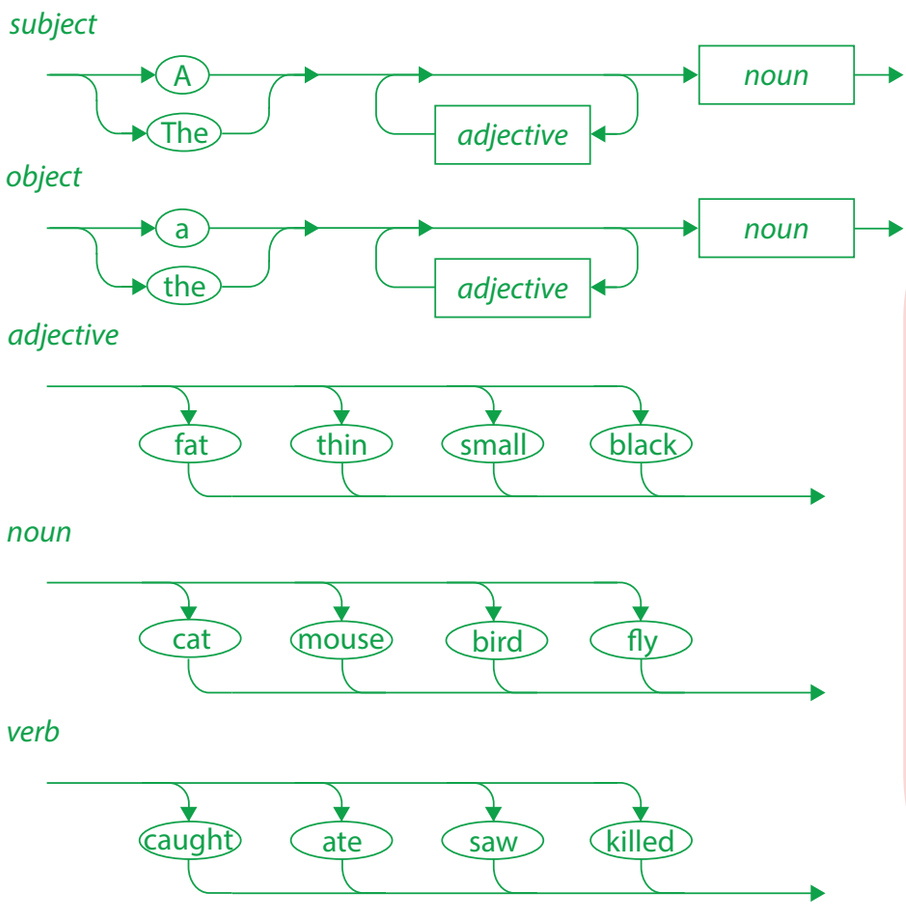


Figure 4.3.1.6 Syntax diagram for a subset of the English language

Task

6. Locate a programming language book which contains a syntax diagram presentation of the language.
   Focus on a part of the syntax diagram(s) and see if you can understand the rules for constructing, say, identifiers and statements in this language.

Single licence - Abingdon School

Questions

7 In a particular programming language, the correct syntax for three different constructs is defined by the diagram, *Figure 4.3.1.7*.

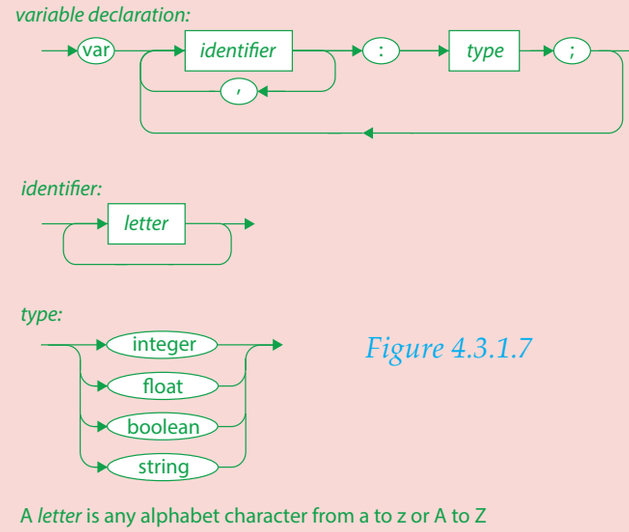


Figure 4.3.1.7

(a) Name the type of diagram shown in this figure.

In this diagram an example of a valid *identifier* is `nameString` and an example of a valid type is `string`.

(b) For each row in the table below, write **Yes** or **No** in the empty column to identify whether or not the **Example** is a valid example of the listed **Construct**.

Construct	Example	Valid (Yes/No)
identifier	MyCount3	
variable declaration	x : integer;	
variable declaration	var x, y : integer; flag : boolean;	

(c) Write the context-free grammar in BNF for the variable declaration shown in *Figure 4.3.1.7*.

8 An identifier in a programming language consists of a letter followed by zero or more letters or digits. Draw a syntax diagram for valid identifiers in this language.

9 A **UserID** for logging into a computer system consists of **two** digits followed by one of the abbreviations **H**, **L**, **D**, **R**, **Pa**, **Ph**, followed by a surname consisting of one or more upper case letters **A...Z**, followed by a single initial chosen from the set of upper case letters **A...Z**. Draw a syntax diagram for **UserID**s to this system.

In this chapter you have covered:

- Checking language syntax by referring to BNF or syntax diagrams and formulating simple production rules
- Explaining why BNF can represent some languages that cannot be represented using regular expressions

Single licence - Abingdon School

4 Theory of computation

4.4 Classification of algorithms

Learning objectives:

- *Understand that algorithms can be compared by expressing their complexity as a function relative to the size of the problem.*
- *Understand that the size of the problem is the key issue*
- *Understand that some algorithms are more efficient*
  - *time-wise than other algorithms*
  - *space-wise than other algorithms.*

4.4.1 Comparing algorithms

Types of problem

There are problems such as calculating the sum of the first N natural numbers for which it is easy to develop and implement computer solutions.

There are also types of problem for which:

- computer solutions can be implemented, but for which the waiting time before results are produced is inordinately long even when N, the size of the data input for the problem, is modestly sized, e.g. N connected cities for the travelling salesman problem where $N = 10$
- computer solutions can be implemented provided enough computer resources are available, e.g. memory, parallel processors
- it is possible to prove that there are no solutions, e.g. the Halting problem.

We use algorithms to solve problems.

An algorithm is of little practical use if it takes years to compute the solution or requires more memory than is practicable. To be usable it must use as little time and as little memory as possible when computing results.
Usually there is more than one algorithm which solves a particular problem if the problem is solvable by algorithm.
The runtime (execution time) of different algorithms can grow at different rates as the size of input increases as can the amount of memory consumed by each.
The runtime of an algorithm is important when choosing which algorithm to use to solve a particular problem because often we need results to be calculated quickly. Where memory is limited, e.g. in an embedded device, it is important to minimise the amount of memory that an algorithm needs to solve a problem.

Time and memory are resources which are consumed by algorithms.

Unit 2 (page 3) covers two different algorithms for solving the problem of calculating the sum of the first N natural numbers. One uses brute-force approach and the other a formula.
For large N the brute-force approach is considerably slower than the formula approach.
We say that the former consumes more time than the latter as N increases.

Single licence - Abingdon School

Resources consumed

It is of interest to know the quantity of resources consumed by each algorithm when solving a particular problem.

Only those algorithms that use a feasible amount of resources are useful in practice.

The major computer resources of interest are

- time
- memory
- hardware, e.g. number of parallel processors.

Time

Often the choice between competing algorithms comes down to which one takes the least amount of computing time.

How could the amount of time that an algorithm takes to solve a problem, of a given size of input N, be estimated?

The first possibility is to simply code the algorithm and then measure the execution time for its program when run.

For two different algorithms implemented in a particular programming language on a particular computer and which solve the same problem, the conclusion can be drawn that the one whose program has the shorter execution time (run-time) is clearly the better algorithm time-wise or the more time-efficient one.

However, we would have difficulty comparing the two different algorithms if one was implemented on a "slow" computer and the other on a "fast" computer.
The particular programming language used will also have a bearing on the amount of time taken:

- some languages require that programs written in these languages are executed by an interpreter whilst for other languages, programs can be compiled to native code which will execute more quickly on the same machine than their interpreted equivalent.
- some languages generate more machine code instructions than others which will also affect execution times.

We need a better method that is independent of any particular computer, any particular programming language and whether a program is interpreted or compiled.
One such way is to isolate and focus on a particular operation or group of operations that are fundamental to the algorithm and which will dominate in the estimation of time taken.
We call this particular operation(s) the **basic operation**.
For the problem of calculating the sum of the first N natural numbers, i.e. $1 + 2 + 3 + 4 + \ldots + (N-1) + N$, we could use a brute-force algorithm that adds the next natural number to a running total started from 0,
e.g. for $N = 3$

$0 + 1 = 1$
$1 + 2 = 3$
$3 + 3 = 6$

The particular fundamental operation to this algorithm is the addition operation (an integer addition operation).

Therefore, a measure of time taken could use a count of the number of integer addition operations, e.g. if $N = 100$ there are 100 addition operations.

Key term

Basic operation:
The operation contributing the most to the total running time of the algorithm.

But to estimate the execution time of this algorithm do we actually need to count the number of integer addition operations?

Single licence - Abingdon School

No, it is sufficient to know that the **time taken is some function of the number of elements to be summed, N**. Let's call this function $f(N)$.

We can express the number of addition operations in terms of N:

for a list of N natural number elements, there are N addition operations.

Therefore, the function, $f(N)$ = some constant x N

If we call the constant, c,

$$f(N) = cN$$

The constant c is the estimated time taken for one addition operation.

Whilst intuitively we might conclude that this algorithm will take longer to sum the first 20000 natural numbers than it will to sum the first 10000, we now have a function that we can use to conclude roughly how much longer, i.e. twice as long summing the first 20000 natural numbers compared with summing the first 10000.

This approach has given us a method of comparing different algorithms
that solve the same problem: *find the function $f(N)$ that gives a useful measure of the time taken for the given input N.*

Size of problem

Algorithms usually accept input data and perform some processing on the input data.
The amount of resource used by an algorithm may vary with the size of the input data (and for a given input size by different inputs but we will not consider this type of variation).

For the brute-force algorithm which solves the problem of calculating the sum of the first N natural numbers, the resource considered is **time**.

The time to calculate this sum varies linearly with N, the size of the input, i.e. $f(N) = cN$.

The size of the input is also referred to as **the size of the problem**.

For other problems, the function f, which gives an estimate of execution time of an algorithm, may involve more terms.
For example,

$$f(N) = N^5 + 100N^2 + 10N + 50$$

We can express an approximation of this function using a mathematical notation called **order of magnitude**.

Order of magnitude is a bit like expressing the size of things by reference to familiar objects that people know to be of different magnitudes, e.g. a gnat, a cat, a horse, an elephant, etc... "My pet gerbil is as big as a cat".

The **order of magnitude of a function** or the **order of a function** is identified by the term in the function that **increases fastest relative to the size of the problem**.

In the above example, this term in $f(N)$ is N^5.

This means that for large values of N, N^5 dominates.
This doesn't mean that $100N^2 + 10N + 50$ is not important, it is just that as N gets larger, all other terms in $f(N)$ become irrelevant because N^5 grows bigger more rapidly and swamps these other terms.

The dominating term is called the **asymptotic behaviour of the algorithm**.

To get the asymptotic behaviour of an algorithm we drop all the irrelevant terms.

We are then left with the (complexity) order of the function.

Key term

Order of magnitude of a function or order of function: Identifies with the term in the function that increases fastest relative to the size of the problem.

Single licence - Abingdon School

Key point

Asymptotic behaviour:
By considering the asymptotic behaviour of an algorithm, rather than its actual output, it is possible to ignore/hide factors and constants that change with the speed of the processor, the arrangement of data in memory, and other specifics such as whether the program is interpreted or compiled, whether one programming language generates more instructions for the same task than another.

Key point

Growth rate:
The growth rate of an algorithm is a measure of how the required resources to execute an algorithm scale with size of input, i.e. size of problem.
It can be measured by the order of its asymptotic function, $f(N)$.

Although we might wish to achieve a full understanding of the behaviour of an algorithm, writing down the exact function of N which represents the amount of resource used, e.g. time, can be difficult. We therefore often resort to using the asymptotic behaviour.

To help understand why the asymptotic behaviour is often sufficient, imagine that you had a budget to buy one lorry for £40000 and one bicycle for £100. The total cost of lorry and bicycle is £40100. Ignoring the cost of the bicycle will only introduce an error of 1 in 401 or about 0.25% because the cost of the bicycle is trivial in comparison with the cost of the lorry.

Questions

1. What is meant by the asymptotic behaviour of a function?
2. List the following functions from lowest to highest order. If any are of the same order, place them next to each other in the list and circle them.

$N + 5N^2 + 2N^3$ $\quad N^3 \quad N^2 \quad N^4 - 4 \quad N^2 + 6N \quad N$

Growth rate

The asymptotic behaviour of an algorithm is a useful means of comparing algorithms because it expresses the rate of growth of the consumption of some resource e.g. time, with size of input.

Size of input data N	$f(N) = cN$ ($c = 1$)	$f(N) = cN^2$ ($c = 1$)	$f(N) = c2^N$ ($c = 1$)
1	1	1	2
10	10	100	1024
100	100	10000	$\approx 1.27 \times 10^{30}$
1000	1000	1000000	$\approx 1.07 \times 10^{301}$
10000	10000	100000000	$\approx 2 \times 10^{3010}$

Table 4.4.1.1 Rate of growth of some asymptotic functions, f(N), with size of input

Table 4.4.1.1 shows how some functions, $f(N)$, that give a useful measure of the execution time of an algorithm for the given input N, grow with the size of N.

Quite clearly, the function that varies as $c2^N$ grows considerably more rapidly with input size than either of the other two functions.

Function $f(N) = cN^2$ grows more rapidly with input size than function $f(N) = cN$.

Table 4.4.1.2 shows the same asymptotic functions and the descriptive names used to describe their growth rates.

It is convenient to use a descriptive name for the growth rate of the execution time of an algorithm estimated by asymptotic function.
For example, if the execution time of an algorithm grows as the square of the size of the input, we say that the execution time grows **quadratically**.

This means if the size of the input is doubled, its estimated execution time is quadrupled.

Single licence - Abingdon School

Asymptotic function	Growth rate	
f(N) does not change with N		The amount of work is a constant and does not change with *N*
$f(N) = cN$	Linear	The amount of work is some *constant* times *the size of the problem*.
$f(N) = cN^2$	Quadratic	Algorithms of this type typically involve applying a linear algorithm *N* times.
$f(N) = c2^N$	Exponential	These algorithms are costly as the work increases dramatically in relation to the size of the problem.

Table 4.4.1.2 Asymptotic functions, f(N), and the descriptive names used to describe their growth rates

Questions

3 Three different algorithms, A, B and C have the following rates of growth with time:

Algorithm A: *quadratic*

Algorithm B: *exponential*

Algorithm C: *linear*

List the algorithms A, B and C in order with the slowest rate of growth first.

Complexity class of an algorithm

The more complex an algorithm, the more work (processing) it will have to do to generate an output from its input. The more work an algorithm does the more resource(s) it will consume. If the resource is time then more work means a longer time before the algorithm produces its output.

Algorithms may be classified by their growth rates into complexity classes. *Figure 4.4.1.1* shows some of these complexity classes, their names and an example of an algorithm which belongs to the class.

When comparing an algorithm for a particular task with a different algorithm for the same task, we compare the worst-case times in the knowledge that each represents the slowest case or upper limit.

Worst-case complexity is the complexity computed when we just happen to choose an input that requires the longest time or greatest workload. For example, if we search for the value 90 in the following unsorted list of integers, every number in the list must be examined before it is possible to conclude that 90 is not present in the list.

203, 67, 34 ,123, 89, 45, 7

The worst-case number of comparisons that need to be performed varies with the length of the list, *N*, i.e. *linearly*.

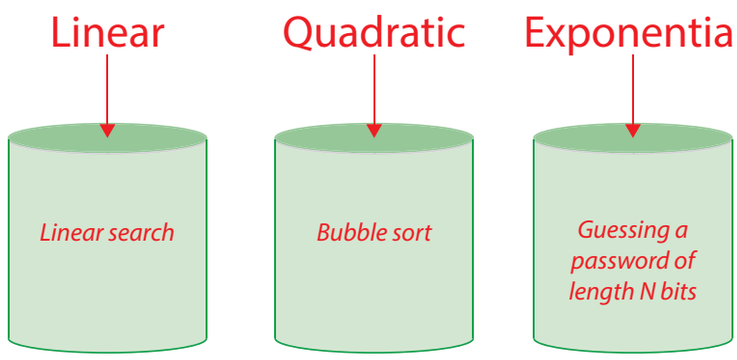


Figure 4.4.1.1 Some complexity classes and examples of algorithms that belong to these classes

Key term

Complexity class of an algorithm:
Algorithms are classified by their growth rates into complexity classes.
The growth rate of an algorithm is given by the order of the asymptotic function *f(N)* for the worst-case.

Single licence - Abingdon School

Key term

Worst-case time complexity of an algorithm:
Worst-case time complexity is the complexity computed when we just happen to choose an input that requires the longest time or greatest workload.

Figure 4.4.1.2 shows a bubble sort applied to this list to produce a list sorted in ascending order. This bubble sort algorithm compares each adjacent pair of integers in turn, e.g. 203 with 67, starting from the beginning of the list. If necessary, the adjacent pair integers are swapped, e.g. 203, 67 → 67, 203. The algorithm does this six times in the example. The growth rate of a single pass is proportional to $N - 1$, the number of integers in the list. The growth rate of the number of passes is proportional to $N - 1$. Therefore the growth rate for this bubble sort algorithm is proportional to $(N - 1) \times (N - 1)$, i.e. $N^2 - 2N + 1$. Asymptotically this is N^2 or **quadratic**.

INPUT Unsorted list	6 comparisons 6 swaps	6 comparisons 3 swaps	6 comparisons 2 swaps	6 comparisons 2 swaps	6 comparisons 1 swap	6 comparisons 1 swap	OUTPUT Sorted list
203	67	34	34	34	34	7	7
67	34	67	67	45	7	34	34
34	123	89	45	7	45	45	45
123	89	45	7	67	67	67	67
89	45	7	89	89	89	89	89
45	7	123	123	123	123	123	123
7	203	203	203	203	203	203	203

Figure 4.4.1.2 Bubble sort applied to an unsorted list of integers

Figure 4.4.1.3 shows decision trees for passwords of length N bits where N is 1, 2 and 3, respectively.
For example, if $N = 1$ then there are two choices of password, 0 or 1.
If we apply a brute-force algorithm to guess the password then for $N = 1$ in the worst-case we take 2 guesses.
If $N = 2$ we take 4 guesses, $N = 3$, 8 guesses.

In general, we will take 2^N guesses which means we can classify this brute-force guessing algorithm as **exponential**.

An algorithm that belongs to a complexity class that grows at a slower rate time-wise than another is said to be more time efficient.

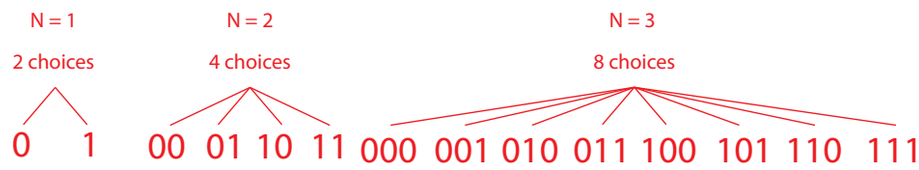


Figure 4.4.1.3 Guessing a password of length N bits

Questions

4 The order of time complexity is given as follows for each of three different algorithms, A, B and C:

Algorithm A: $f(N) = N^2$
Algorithm B: $f(N) = 2^N$
Algorithm C: $f(N) = N$

List the algorithms A, B and C in order with the most time efficient first.

Single licence - Abingdon School

Complexity of a problem

The complexity of a problem is taken to be the worst-case complexity of the most efficient algorithm which solves the problem.

The emphasis on problems is very important because problems can be classified according to their complexity.

The discovery of a more efficient algorithm for a particular problem can then be applied to all other problems belonging to the same class of complexity.

Key term

Complexity of a problem: The complexity of a problem is taken to be the worst-case complexity of the most efficient algorithm which solves the problem.

Space complexity

Space complexity is a measure of the amount of working storage an algorithm needs. This means how much memory, in the worst case, is needed at any point in the algorithm. As with time complexity, the focus is on how the space required grows (asymptotically), as the size *N* of the input problem grows.

Devices such as embedded devices have limited memory for their data and their software. Therefore, it is very important to develop programs with a small memory footprint.

Suppose that an algorithm is designed to process a linear list of *N* integers. If each integer requires four bytes of memory then *N* integers will require *4N* bytes. The rate of growth of consumption of memory is therefore proportional to the size of the list, *N*. We say that the space complexity is linear in *N*, the size of the input list to the algorithm.

Suppose that an algorithm is designed to process a two-dimensional array, $N \times M$ of integers and $N = M$.
If each integer requires four bytes of memory then $N \times N$ integers will require $4 \times N \times N$ bytes
Now if each dimension of the array is doubled, the memory consumed is now $4 \times (2N \times 2N)$ bytes, four times as much. In this case, the rate of growth of consumption of memory is therefore proportional to, $N \times N$ or N^2.
Therefore, we say that the space complexity of the algorithm is quadratic in the dimension of the input array.

Trade-off between time-complexity and space-complexity

To some extent, there is a trade-off between time and space. Suppose the factors of every integer from 1 to 1000 are to be computed. *Table 4.4.1.3* shows a simple algorithm for finding the factors of an integer *N*.

It uses the algorithm shown in *Table 4.4.1.4*.

The algorithm `FindFactors(N)` will output the factors of *N* in increasing order.

```
Algorithm FindFactors(N)
//Finds every factor of a given natural number, N
//Input: An integer N ≥ 2
//Output: Factors
While N > 1
  Factor ← LeastFactor(N)
  Output Factor
  N ← N Div Factor //Integer division
EndWhile
```

Table 4.4.1.3 Algorithm for finding factors of an integer N

```
Algorithm LeastFactor(N)
//Finds smallest factor of a given natural number, N
//Input: An integer N ≥ 2
//Output: Smallest factor
i ← 1
Repeat
  i ← i + 1
Until (N Mod i) = 0 //Integer remainder division
Return i
```

Table 4.4.1.4 Algorithm for finding the least factor of N

Single licence - Abingdon School

Now suppose this algorithm is used in a `For loop` to calculate the factors of every natural number between 2 and 1000 as follows:

```
For i ← 2 To 1000 Do FindFactors(N)
```

`FindFactors(N)` is very efficient in terms of space since it uses only three variables `i`, `N` and `Factor`. However, it is very wasteful in terms of time. For example, to calculate the factors of the natural number 200, the algorithm calls `LeastFactor(200)`. Its first output is 2. The algorithm continues by halving its argument `N` using `N ← N Div Factor`, which results in a new intermediate value of `N` of 100. `LeastFactor(100)` is then called.

However, well before the `For loop` reached `FindFactors(200)`, it would have calculated `FindFactors(100)`, when `N` was 100. Thus, in calculating `FindFactors(200)`, the algorithm repeats the calculation `FindFactors(100)` because it does not remember that it has found these factors already.

In fact, for the number 10, this algorithm works out its factors 110 times in calculating the factors of each natural number in the range 10 to 1000.

It obviously makes sense to store the factors of the number 10 the first time and then refer to the stored values whenever the factors of 10 are needed again. In fact, it makes sense to store the factors for each value of *N* for future use when first calculated.

A 999 x 9 array is sufficient for this (range of *N* is 2 to 1000, i.e. 999 numbers). *Table 4.4.1.5* shows a sample from this array.

N									
55	5	11	1	1	1	1	1	1	1
56	2	2	2	7	1	1	1	1	1
57	3	19	1	1	1	1	1	1	1
58	2	29	1	1	1	1	1	1	1
59	59	1	1	1	1	1	1	1	1
60	2	2	3	5	1	1	1	1	1
61	61	1	1	1	1	1	1	1	1
62	2	31	1	1	1	1	1	1	1
63	3	3	7	1	1	1	1	1	1

Table 4.4.1.5 Sample of the 999 x 9 factor array for values of N from 2 to 1000

An algorithm that uses this table should now take less time but will consume more space. This example illustrates a general principle that one can often improve time complexity of an algorithm at the expense of space complexity.

In this algorithm, the input *N* does not define the length of a list of elements as was the case with the algorithm for calculating the sum of the first *N* natural numbers. Instead the input to be processed is the integer *N*.

Key term

Complexity of an algorithm: Measures how economical the algorithm is with time and space.

Time complexity of an algorithm: A measure of time complexity of an algorithm is the number of steps in a computation. An algorithm is of time complexity *f*(*N*) if for all *N* and all inputs of length *N*, the execution of the algorithm takes at most *f*(*N*) steps.

Space complexity of an algorithm: Indicates how much memory an algorithm needs.

Single licence - Abingdon School

Questions

5 The following algorithm calculates the sum of the first N natural numbers. It declares an array A which it populates with the sum. The first element contains 1, the second 3, the third 6, and so on.

```
Algorithm CalculateSum(N)
  //Calculates sum of first N natural numbers
  //Input: N, where N ≥ 1
  //Output: Returns the sum
Declare Array A[1..N]
A[1] ← 1
For i ← 2 To N
  A[i] = A[i-1] + i
EndFor
Return A[N]
```

By considering the asymptotic growth rate of this algorithm, select the correct order of space complexity for this algorithm from the following

$f(N) = N^2$

$f(N) = 2^N$

$f(N) = N$

$f(N)$ does not change with N

6 The following algorithm calculates the sum of the first N natural numbers.

```
Algorithm CalculateSum(N)
  //Calculates sum of first N natural numbers
  //Input: N, where N ≥ 1
  //Output: Returns the sum
Sum ← 0
For i ← 1 To N
   Sum = Sum + 1
EndFor
Return Sum
```

By considering the asymptotic growth rate of this algorithm, select the correct order of space complexity for this algorithm from the following

$f(N) = N^2$

$f(N) = 2^N$

$f(N) = N$

$f(N)$ does not change with N

Single licence - Abingdon School

In this chapter you have covered:

- That algorithms can be compared by expressing their complexity as a function relative to the size of the problem
- That the size of the problem is the key issue
- That some algorithms are more efficient
  - time-wise than other algorithms
  - space-wise than other algorithms.

Single licence - Abingdon School

4 Theory of computation

4.4 Classification of algorithms

Learning objectives:

- *Be familiar with the mathematical concept of a function as a mapping from one set of values, the domain, to another set of values, drawn from the codomain, for example* $\mathbb{N} \rightarrow \mathbb{N}$
- *Be familiar with the concept of:*
  - *a linear function, for example* $y = 2x$
  - *a polynomial finction, for example* $y = 2x^2$
  - *an exponential function, for example* $y = 2^x$
  - *a logarithmic function, for example* $y = \log_{10} x$
- *Be familiar with the notion of permutation of a set of objects or values, for example, the letters of a word and that the number of permutations of n distinct objects is n factorial (n!).*

Key point

Function $f(x) = x^2$ defined on domain A = {0,1,2,3,4,5} maps A into codomain B as follows

$0 \mapsto 0$
$1 \mapsto 1$
$2 \mapsto 4$
$3 \mapsto 9$
$4 \mapsto 16$
$5 \mapsto 25$

where $\mapsto$ means function f maps element x from domain A to element y from codomain B.

4.4.2 Maths for understanding Big-O notation

Mathematical concept of a function

A function is a mapping that maps the set of inputs onto the set of outputs. For example,

Input set		Output set
{0, 1, 2, 3, 4, 5}	*maps onto*	{0, 1, 4, 9, 16, 25}

For functions, the following special condition applies:

An element of the input set is related to exactly one element of the output set.

Figure 4.4.2.1 shows two sets such that an element of the output set is the square of an element of that belongs to the input set.

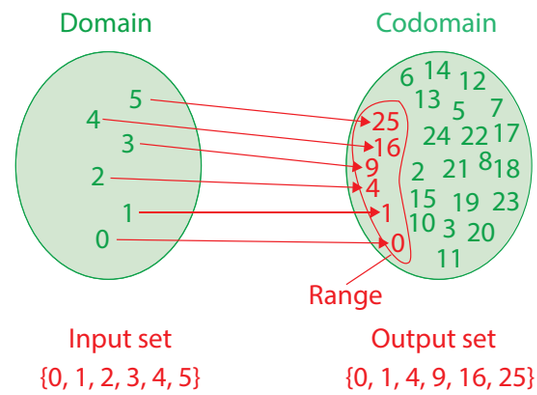


Figure 4.4.2.1 Mapping of a set of inputs into a set of outputs

The set of all possible inputs of a function f is called the **domain** of f.

The set of all the outputs of f is called the **range** of f.

The range of a function f is conveniently viewed as a subset of a larger set called the **codomain**.

We say that a function *maps* the domain *onto* its range and *into* a larger set, the codomain that contains the range.

If the domain shown in *Figure 4.4.2.1* is labelled A and the codomain B then function f maps domain A into codomain B.

We use the following notation to represent this

$$f : A \rightarrow B$$

where $\rightarrow$ mean maps into.

The function f maps an input x from domain A onto exactly one element, the output $f(x)$ belonging to the codomain B as follows:

$$f(x) = x^2$$

Single licence - Abingdon School

x	$f(x)$
0	0
1	1
2	4
3	9
4	16

Table 4.4.2.1 f(x) for some values of x

Table 4.4.2.1 shows the values of $f(x)$ for the given values of x.

The variable x is referred to as the independent variable quantity because we are free to assume a value for it as long as that value is within its domain.

The values of $f(x)$ for the given values of x *are shown plotted* on a two-dimensional graph in *Figure 4.4.2.2*. This plot was produced by the Python program shown in *Figure 4.4.2.3*.

```
import matplotlib.pyplot as plt
from pylab import savefig
fig = plt.figure()
fig.suptitle('f(x) plot', fontsize = 20)
plt.xlabel('x', fontsize = 20)
plt.ylabel('f (x)', fontsize = 20)
plt.axis([0, 6, -5, 30])
plt.plot([0,1,2,3,4,5], [0,1,4,9,16,25], 'ro')
plt.show()
savefig('Quadratic.png')
```

Figure 4.4.2.3 Python program Quadratic.py

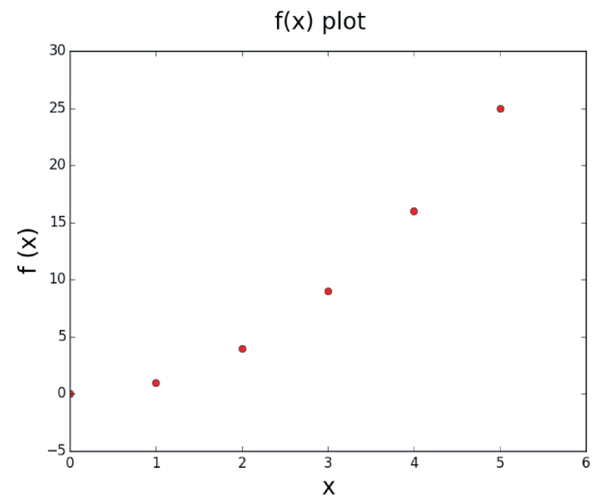


Figure 4.4.2.2 f(x) plot for values of x drawn from the domain

If we change the domain to a subset of the real numbers $\mathbb{R}$ by restricting x to the interval 0.0 <= x <= 5.0, we obtain the plot for $f(x) = x^2$ shown in *Figure 4.4.2.4*.

Figure 4.4.2.5 shows the Python program which generated this plot.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import spline
from pylab import savefig
T = np.array([0.0, 0.5, 1.0, 1.5, 2.0, 2.5,
3.0, 3.5, 4.0, 4.5, 5.0])
power = np.array([0, 0.25, 1.0, 2.25, 4.0,
6.25, 9.0, 12.25, 16, 20.25,  25])
xnew = np.linspace(T.min(),T.max(),300)
power_smooth = spline(T,power,xnew)
fig = plt.figure()
fig.suptitle('f (x) plot', fontsize = 20)
plt.xlabel('x', fontsize = 20)
plt.ylabel('f (x)', fontsize = 20)
plt.plot(xnew,power_smooth)
plt.show()
savefig('quadraticsmooth.png')
```

Figure 4.4.2.5 Python program QuadraticSmooth.py

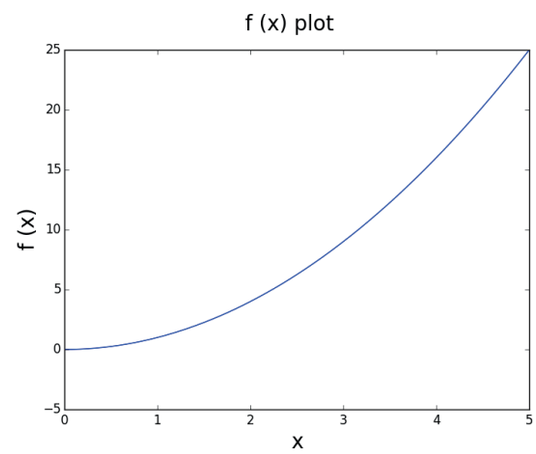


Figure 4.4.2.4 f(x) plot for values of x drawn from a subset of the domain ℝ

Single licence - Abingdon School

Questions

1. A function f is defined on domain set A. This set has four elements, 0, 1, 2, 3.
   The function f maps an input x from domain A to output $f(x)$ in codomain B as follows
   $$f(x) = x^3$$
   Write the set B assuming that B is also the range of f.

2. Give an example of a function f that maps all integers onto the set of all even integers.

Linear function

A linear function is a mapping of the form

$$f(x) = ax + b$$

where a and b are called coefficients.

For example, if $a = 2$ and $b = 0$

$$f(x) = 2x$$

Table 4.4.2.2 shows $f(x) = 2x$ defined on domain {0, 1, 2, 3, 4}.
Figure 4.4.2.6 shows a plot of $f(x)$.

x	$f(x)$
0	0
1	2
2	4
3	6
4	8

Table 4.4.2.2 $f(x) = 2x$ for some values of x

If we change the domain to a subset of the real numbers ℝ by restricting x to the interval 0.0 <= x <= 4.0, we obtain the plot for $f(x) = 2x$ shown in *Figure 4.4.2.7*.

It is often convenient to write y for $f(x)$ i.e.

$$y = f(x)$$

Therefore, for the linear function $f(x) = 2x$

$$y = 2x$$

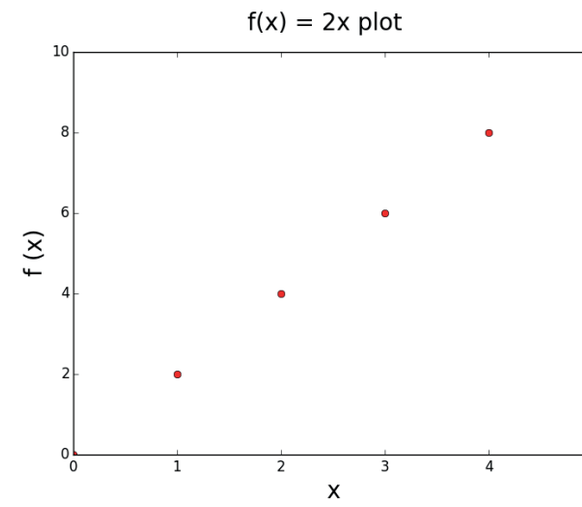


Figure 4.4.2.6 $f(x) = 2x$ plot defined on domain {0,1,2,3,4}

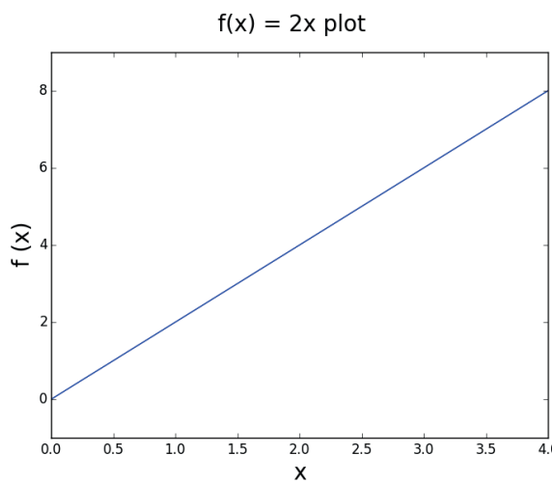


Figure 4.4.2.7 $f(x) = 2x$ plot defined on subset of domain ℝ

Single licence - Abingdon School

Polynomial function

An expression of the form

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

is called a polynomial. The a_n, a_{n-1}, … are numbers called coefficients of the polynomial. For example,

$$4x^2 + 12x + 8$$

where $n = 2$, $a_n = 4$, $a_{n-1} = 12$ and $a_0 = 8$.

a_0 is called the constant term. n is called the degree of the polynomial.

$4x^2 + 12x + 8$ is a polynomial of degree 2. It is also called a **quadratic polynomial** and defines a **quadratic function**.

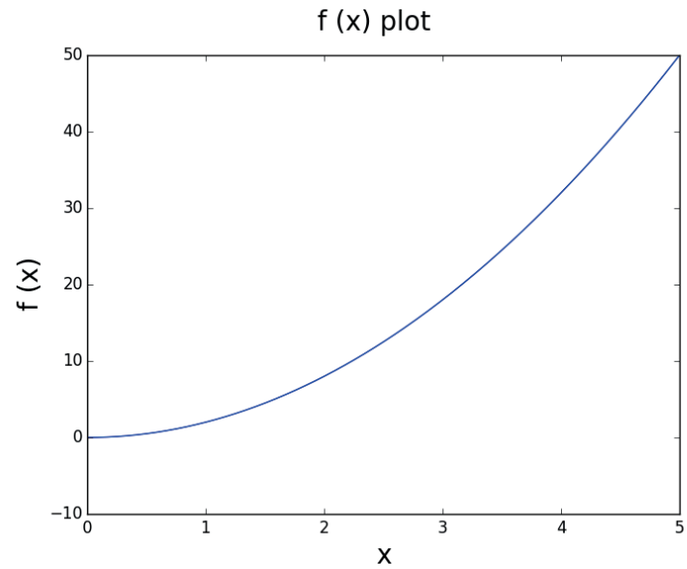


Figure 4.4.2.8 $f(x) = 2x^2$ plot defined on subset of domain ℝ

Figure 4.4.2.8 shows a plot of $f(x) = 2x^2$.

For convenience we can write y for $f(x)$, i.e. $y = 2x^2$

Key term

Polynomial function:

A function of the form

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

is called a polynomial function.

Exponential function

An exponential function is a function of the form

$$f(x) = a^x$$

in which the input variable x occurs as an exponent of a constant a.

For example,

$$f(x) = 2^x$$

Table 4.4.2.3 shows values of $f(x)$ for inputs 0, 1, 2, 3, 4. Note that the value of $f(x)$ doubles as x increments by 1.

Figure 4.4.2.9 shows a plot of $f(x) = 2^x$ defined on a subset of domain ℝ.

Thus, the exponential function f, defined as $f(x) = 2^x$, grows very rapidly with x.

x	$f(x)$
0	1
1	2
2	4
3	8
4	16

Table 4.4.2.3 $f(x) = 2^x$ for some values of x

Key term

Exponential function:

$$f(x) = a^x$$

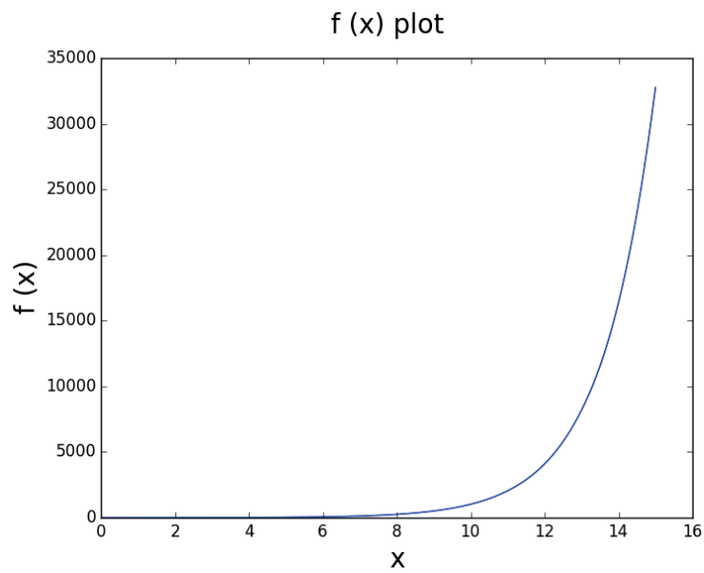


Figure 4.4.2.9 $f(x) = 2^x$ plot defined on subset of domain ℝ

Single licence - Abingdon School

Logarithmic function

The logarithm of a number x is the number n such that $b^n = x$.

For example, if $b = 10$ and $x = 1000$, $n = 3$ because $10^3 = 1000$.

b is called the **base of the logarithm**.

We would say, for the given example, log to the base 10 of 1000 is 3.

Log to the base 10 of 100000 is 5. *Table 4.4.2.4* shows $\log_{10} x$ for some values of x. Notice that $\log_{10} x$ corresponds to the number of noughts in x.

x	$\log_{10} x$
1	0
10	1
100	2
1000	3
10000	4
100000	5

Table 4.4.2.4 $\log_{10} x$ for some values of x

The most striking thing to note is that as x increases by a factor of 10, $\log_{10} x$ increments by 1.

Thus, the logarithmic function f, defined as

$f(x) = \log_{10} x$, grows very slowly with x.

Figure 4.4.2.10 shows a plot of $f(x) = \log_{10} x$ defined on a subset of domain $\mathbb{R}$.

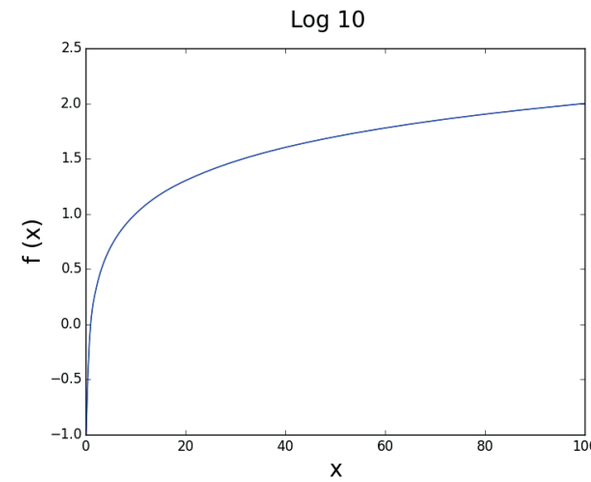


Figure 4.4.2.10 $f(x) = \log_{10} x$ plot defined on subset of domain $\mathbb{R}$

We may use other bases, e.g. $b = 2$.

This is called log to the base 2. It is written as $\log_2 x$.

Table 4.4.2.5 shows $\log_2 x$ for some values of x

x	$\log_2 x$
1	0
2	1
4	2
8	3
16	4
32	5

Table 4.4.2.5 $\log_2 x$ for some values of x

Figure 4.4.2.11 shows a plot of a quadratic function, $f(x) = x^2$, a cubic function, $f(x) = x^3$, and an exponential function, $f(x) = 2^x$, for comparison of their growth rate with x.

As can be seen from the figure, the exponential function dominates when x exceeds 12.

The scale is too small to show both a logarithmic function and a linear function.

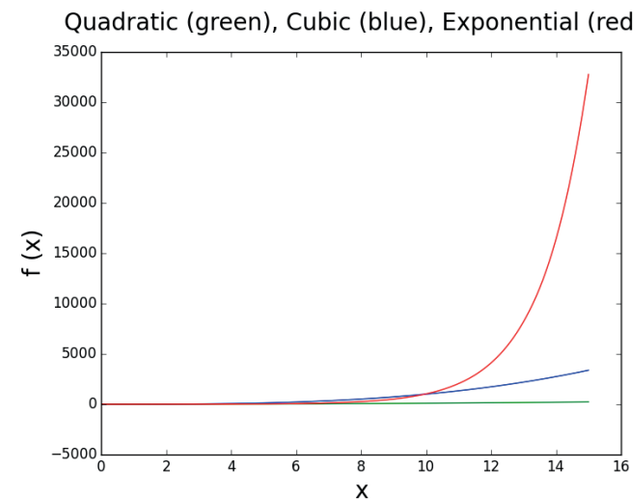


Figure 4.4.2.11 Plotting $f(x) = 2^x$, $f(x) = x^3$ and $f(x) = x^2$ defined on subset of domain $\mathbb{R}$

The value of $f(x) = x^2$ has increased with increasing x but that this is not obvious due to the scale used for the y axis.

Key term

Logarithmic function:

$f(x) = \log_b x$

where $b^n = x$ and $n = f(x)$

Single licence - Abingdon School

Questions

3 What is $\log_2 2$?

4 How does y vary when x is doubled if $y = 2x^2$?

5 How does y vary when x is doubled if $y = \log_2 x$?

6 For large x ($x > 12$), place the following functions in order of growth rate with the fastest first and the slowest last

$$2x,\ \log_2 x,\ 2x^3,\ 2x^2$$

7 A courtier presented the Persian king with a beautiful, hand-made chessboard. When the king asked what he would like in return, the courtier surprised the king by asking for two grains of rice on the first square, four grains on the second, eight grains on the third, and so on.
The king readily agreed and asked for the rice to be brought. How many grains of rice were placed on
(a)
(i) the 4th square, (ii) the 10th square and (iii) the 64th square?
(b) The function $f(x)$ defined on domain A = {1, 2, 3, 4, ..., 63, 64} calculates the number of grains of rice. Define $f(x)$ in terms of x.

Permutation of n objects

An ordering of n distinct objects or symbols is called a **permutation**.

We can choose the object for the first position in n ways, for the second position in n - 1 ways, and so on. Only one object remains when choosing the last position.

Therefore the total number of ways is

$$n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1$$

For example, suppose we have a set, A, consisting of the lowercase letters of the alphabet, {a, b, c, ..., x, y, z}.

Table 4.4.2.6 shows the permutations of various subsets of A. Note how the number of permutations grows as the size of the set grows.

We use $n!$ as shorthand for $n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1$.
$n!$ is pronounced **n factorial**.

Subset	Permutations	No of permutations	Product
{a, b}	ab, ba	2	2 x 1
{a, b, c}	abc, acb, bac, bca, cab, aba	6	3 x 2 x 1
{a, b, c, d}	abcd, abdc, acbd, acdb, adbc, adcb, badc, bacd, bcda, bcad, bdca, bdac, cabd, cadb, cbad, cbda, cdab, cdba, dacb, dabc, dbca, dbac, dcba, dcab	24	4 x 3 x 2 x 1
{a, b, c, d, e}	...	120	5 x 4 x 3 x 2 x 1
{a, b, c, d, e, f}	...	720	6 x 5 x 4 x 3 x 2 x 1
{a, b, c, d, e, f, g}	...	5040	7 x 6 x 5 x 4 x 3 x 2 x 1

Table 4.4.2.6 Permutations of sets of distinct objects

Single licence - Abingdon School

The factorial function is a function of the form

$$f(x) = x!$$

defined on the domain of natural numbers, ℕ, i.e. {1, 2, 3, 4, ...}.

Figure 4.4.2.12 is a plot of $f(x) = x!$ shown in red. The figure also shows a plot of $f(x) = 2^x$ in green.

The growth rate of the factorial function is greater than exponential when the size of the set exceeds quite a small value.

To show this more clearly, the functions are re-plotted in *Figure 4.4.2.13* as smooth plots although care must be exercised as the factorial function is not defined for non-integral numbers.

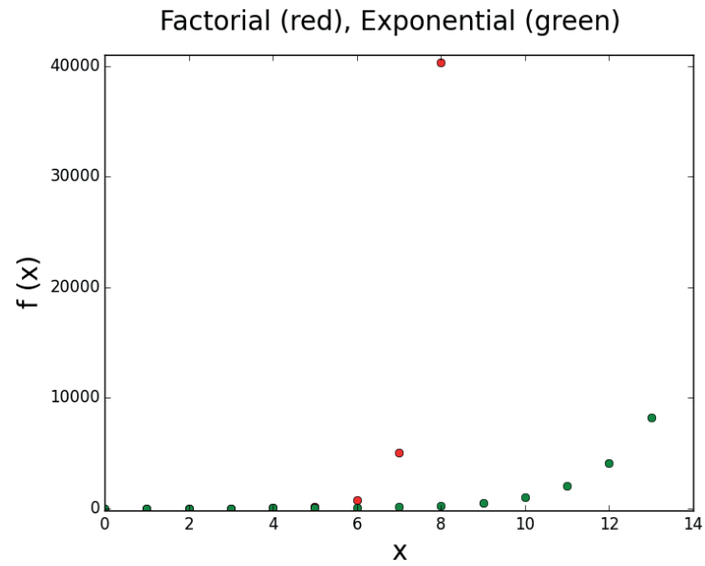


Figure 4.4.2.12 Plot of $f(x) = x!$, and $f(x) = 2^x$

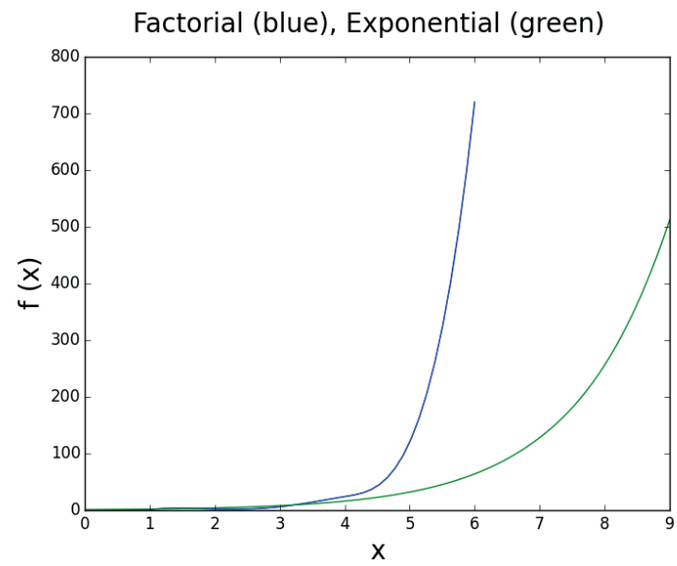


Figure 4.4.2.13 Smooth plot of $f(x) = x!$, and $f(x) = 2^x$

Key term

Permutation of n distinct objects:
$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1$

Questions

8 How many permutations are there of the letters of the following words?
(a) flow (b) flowchart (c) flowchartings

Single licence - Abingdon School

In this chapter you have covered:

- The mathematical concept of a function as a mapping from one set of values, the domain, to another set of values, drawn from the codomain, for example $\mathbb{N} \rightarrow \mathbb{N}$
- The rule that an element of the input set is related to exactly one element of the output set
- The set of all possible inputs of a function f is called the **domain** of f.
- The set of all the outputs of f is called the **range** of f.
- The range of a function f is conveniently viewed as a subset of a larger set called the **codomain**.
- The concept of:
  - a linear function, for example $y = 2x$
  - a polynomial function, for example $y = 2x^2$
  - an exponential function, for example $y = 2^x$
  - a logarithmic function, for example $y = \log_{10} x$
- The notion of permutation of a set of objects or values, for example, the letters of a word and that the number of permutations of n distinct objects is n factorial or in shorthand $n!$
- $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1$.

Single licence - Abingdon School

4 Theory of computation

4.4 Classification of algorithms

Learning objectives:

- *Be familiar with Big-O notation to express time complexity and be able to apply it to cases where the running time requirements of the algorithm grow in:*
  - *constant time*
  - *logarithmic time*
  - *linear time*
  - *polynomial time*
  - *exponential time*
- *Be able to derive the time complexity of an algorithm.*

4.4.3 Order of complexity

Big-O notation

Big-O notation[1] is special notation for classifying algorithms by how their computation time grows as their input size grows.

How computation time grows with the size of the input is important to know.

For example, suppose the computation involves searching a list of 128 elements. Linear search needs to potentially check each element of an unsorted list, which takes 128 operations, e.g. searching for Zut in the unordered list shown in *Figure 4.4.3.1* if Zut is not present in the list.

Abe	Minns	Vim	Aggit	Smith	.....	Zac	Bond	Codd	Raj	Ames

Figure 4.4.3.1 Unsorted list of names

Task

Watch the following videos on time complexity of algorithms in the given order

1. https://www.youtube.com/watch?v=V42FBiohc6c
2. https://www.youtube.com/watch?v=8syQKTdgdzc
3. https://www.youtube.com/watch?v=OpebHLAf99Y

In video 3, the presenter uses functions *g* and *f* in the opposite manner to their use in this chapter. The concept is the same. You need only watch video 3 up to the end of Big-O coverage. The content of the rest of this video is not in the specification.

Binary search operates on a sorted list. Binary search needs to check 8 elements at most in a sorted list of 128 elements, which takes 8 operations. Binary search repeatedly divides the list in two giving at most $1 + \log_2 128$ operations.

However, in linear search we might get lucky and find a match on the first element in the list, i.e. after one operation. In *Figure 4.4.3.1* this occurs when searching for Abe in the unsorted list. In this instance, binary search on the sorted equivalent of this list would actually perform much worse because it is designed to always examine the middle element of a shrinking list.

A much more useful measure is growth rate (of the worst case).

For example suppose our list consists of 1048576 (2^{20}) elements.

Linear search potentially needs to check each element, which takes 1048576 operations. This is twice as many operations as searching a list of length 2^{19}, and four times as many as a list of 2^{18} ..., and so on - see *Figure 4.4.3.2*.

Size of list	List	Maximum no of operations
2^{20}	.....	2^{20}
⋮		
8		8
4		4
2		2
1		1

Figure 4.4.3.2 How maximum no of comparisons varies with size of list for linear search

1 Big-O also can be applied to the space complexity of an algorithm.

Single licence - Abingdon School

Binary search on the equivalent sorted list needs to check 21 elements at most, which is 21 operations. To search a list half as long, i.e. 2^{20} elements, needs at most one less operation, i.e. 20, ..., and so on - see *Figure 4.4.3.3*.

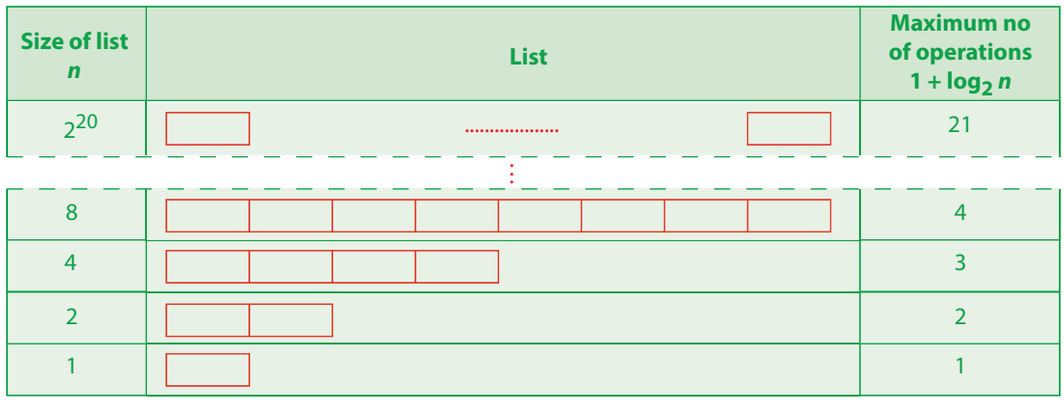


Figure 4.4.3.3 How maximum no of comparisons varies with size of list for binary search

Classifying how fast functions grow

Suppose we have three functions

1. $g_1(n) = n$
2. $g_2(n) = 2n$
3. $g_3(n) = 3n$

Function	Output for inputs 1	2	3	4
$g_1(n) = n$	1	2	3	4
$g_2(n) = 2n$	2	4	6	8
$g_3(n) = 3n$	3	6	9	12

Table 4.4.3.1 Growth rate of three functions

Table 4.4.3.1 shows for input 1, 2, 3, 4 in turn, how the output of each function grows.

In each case, if the size of the input is doubled, e.g. from 1 to 2, the output is also doubled. If input size is tripled so is the output size for all three functions. We say that output grows as the size of the input, n.

We collect all functions whose output grows linearly with input n in the same set and call this set $O(n)$.

In general,

$O(f)$ is the set of functions that grow no faster than function f grows.

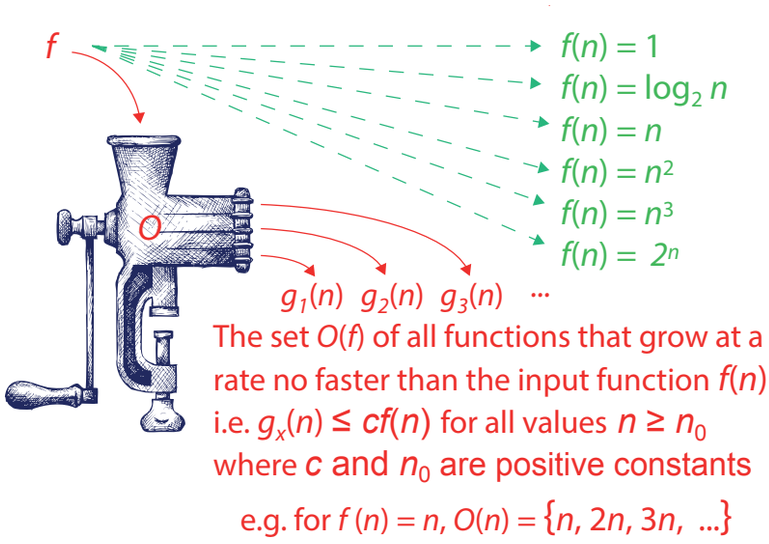


Figure 4.4.3.4 The O function takes as input a function f(n), and produces as output the set of all functions that grow no faster than the input function.

For example, g_1, g_2, and g_3 belong to $O(n)$ where $f(n) = n$.

$O(f)$ captures the asymptotic behaviour of functions, that is, how they behave as the inputs get arbitrarily large.

The function g is a member of the set $O(f)$ if and only if there exist positive constants c and n_0 such that, for all values $n \geq n_0$,

$$g(n) \leq cf(n)$$

i.e. $g(n)$ grows no faster than $cf(n)$.

For example, $f(n) = n$, $g(n) = 3n$ and $c = 3$.

Table 4.4.3.1 shows that functions $g_1(n)$, $g_2(n)$, $g_3(n)$ belong to $O(n)$ where function $f(n) = n$.

Single licence - Abingdon School

Figure 4.4.3.5 shows a visualisation of the set $O(f)$ for the functions $f(n) = n$, $f(n) = n^2$, $f(n) = 2^n$. They are overlapping sets.

The time complexity classes expressed in Big-O notation are as follows, with $n!$ growing the fastest in time as a function of input size:

$O(n^0) << O(\log n) << O(n^1) << O(n\log n) << O(n^2) << \ldots << O(n^k) << \ldots << O(2^n) << O(3^n) << \ldots << O(n!)$

where $O(n^0) = O(1)$.

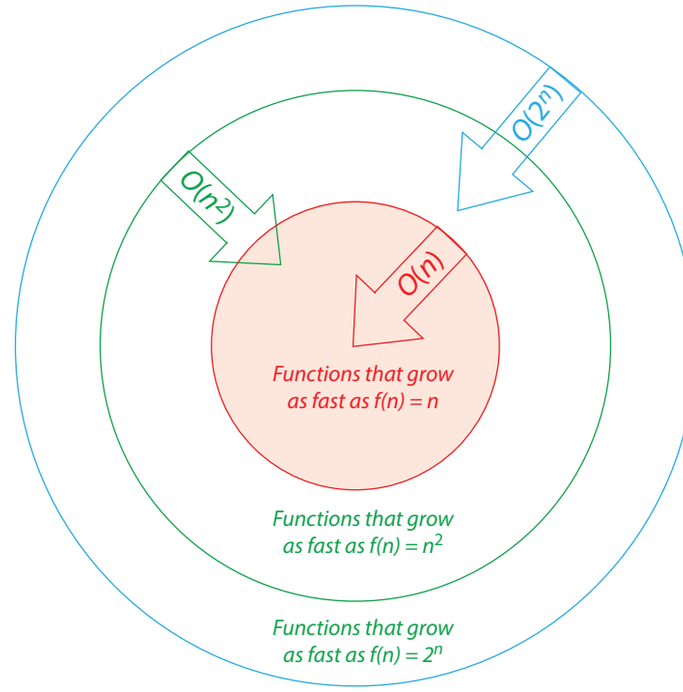


Figure 4.4.3.5 Visualisation of the set O(f) for the functions f(n) = n, f(n) = n², f(n) = 2ⁿ

Big-O analysis

Big-O analysis allows the time-complexity of algorithms to be compared without reference to the details of any computer or an machine instruction set and other factors which bear on the real time execution of the algorithm.

Constant time O(1)

$O(1)$ or $O(n^0)$ is called **bounded time**.

The order of magnitude amount of time taken is bounded by a constant and is independent of the size of the input n.

Logarithmic time O(log n)

The order of magnitude amount of time taken depends on the logarithm of the size of the input n. Algorithms that successively cut the amount of data to be processed in half at each step typically fall into this category, e.g. binary search is $O(\log_2 n)$.

Linear time O(n)

The order of magnitude amount of time taken is some constant times the size of the input n. For example, searching for a particular element in a list of unsorted elements is $O(n)$ because potentially every element must be examined in the list.

Polynomial time $O(n^k)$

Algorithms whose order of magnitude time complexity can be expressed as a polynomial in the size of the input n, i.e. input size raised to a nonnegative integral power. The Big-O time complexity is the highest power in the polynomial. Examples are $O(n^2)$ and $O(n^3)$. Quadratic time $O(n^2)$ algorithms typically involve applying a linear time algorithm n times. For example, bubble sort.

Exponential time $O(2^n)$ or $O(a^n)$

For algorithms in this category, the order of magnitude time taken grows dramatically with the size of the input n. In general, exponential growth is of the form a^n where a is a nonnegative integral constant.

Typically, $a = 2$ because for many algorithmic problems, the time taken doubles when the input size increases by 1. For example, guessing a password of length n bits is $O(2^n)$. *Table 4.4.3.2* illustrates that when the length of the password is increased from 2 bits and 3 bits the potential number of comparisons doubles from 4 to 8.

2-bit password
00
01
10
11

3-bit password
000
001
010
011
100
101
110
111

Table 4.4.3.2 2-bit and 3-bit password bit patterns

Single licence - Abingdon School

Questions

1. What is meant by Big-O notation?
2. Place the following in ascending order of time complexity from the least complex to the most complex: $O(2^n)$, $O(n^3)$, $O(n)$, $O(n^2)$, $O(\log_2 n)$.
3. Explain the following terms:
   (a) linear time algorithm (b) exponential time algorithm (c) polynomial time algorithm.

Deriving the time complexity of an algorithm

Order of growth

We learned in *Chapter 4.4.1* that the growth rate of an algorithm is a measure of how the required resources (time and space) to execute an algorithm scale with size of input, i.e. size of problem. It can be measured by the order of its asymptotic function, $f(n)$.The O in Big-O refers to this order of growth.

If t_b is the execution time of an algorithm's basic operation on a particular computer and $C(n)$ is the number of times this operation needs to be executed for input n for this algorithm, then the estimated running time $T(n)$ of a program implementing this algorithm on this computer is given by the following formula:

$$T(n) = t_b \times C(n)$$

This formula will only give an estimate of the running time. But it allows us to answer the question, How much longer will the algorithm take to execute if its input size is doubled?

Suppose $C(n) = \frac{1}{2} n (n + 1) = \frac{1}{2} n^2 + \frac{1}{2}n$

For all but very small values of n, this formula becomes

$$C(n) \approx \tfrac{1}{2}n^2$$

To understand why this is true, consider $n = 1000$, so $n^2 = 1000000$.

Ignoring the n term introduces an error of only 1000 in 1000000, which is 0.1%. Therefore, doubling the size of the input quadruples the execution time as follows:

$$\frac{T(2n)}{T(n)} = \frac{t_b \times C(2n)}{t_b \times C(n)} = \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = \frac{4n^2}{n^2} = 4$$

n	n^2
1000	1000000
2000	4000000
3000	9000000
4000	16000000
5000	25000000
6000	36000000
7000	49000000
8000	64000000
9000	81000000
10000	100000000

Table 4.4.3.3 Some values of n and n^2

Note that the question has been answered without needing to know the value of t_b. It cancelled out in the ratio, as did the multiplicative constant $\frac{1}{2}$. It is for these reasons that time efficiency analysis ignores multiplicative constants and concentrates on the count's order of growth for large inputs. Order of growth assesses by what factor execution time increases when the size of the input is increased.

In the example, if the input size doubles, the algorithm takes four times longer to execute. The order of growth is therefore quadratic, i.e. varies as the square power. We say that the algorithm is $O(n^2)$. This is a consequence of the fact that the number of times that the basic operation needs to be executed, $C(n)$, is proportional to the square of the input size, n, for all but very small values of n. *Table 4.4.3.3* shows values of n^2 for corresponding values of n. Doubling n always quadruples n^2. Notice how n^2 changes when n increases from 1000 to 2000 to 4000 to 8000.

Single licence - Abingdon School

Example

It is required to rearrange a line of discs consisting of an equal number of green and white discs so that the green discs end up on the right in one group and the white discs end up on the left in another group. For example, two green discs followed by two white discs become two white discs followed by two green discs, as shown in *Figure 4.4.3.6*.

Figure 4.4.3.6 Rearranging the order of four discs, two green, two white

A general algorithm for rearranging a line of discs containing an equal number of black and white discs for which the total number of discs ≥ 2 is shown in *Figure 4.4.3.7*.

```
Algorithm RearrangeDiscs(LineOfDiscs, n)
// Rearranges a line of an equal number of green and white discs
// Inputs: Line of green and white discs
// n, the total no of discs in line n ≥ 2
//Output: Rearranged line of discs
n ← n - 1
For NumberOfPairs ← n DownTo 1
  For j ← 1 To NumberOfPairs
    If LineOfDiscs[j] = green And LineOfDiscs[j + 1] = white
      Then Swap(LineOfDiscs[j], LineOfDiscs[j + 1])
    EndIf
  EndFor
EndFor
```

Figure 4.4.3.7 Algorithm for rearranging a line of n discs containing an equal number of green and white discs

Table 4.4.3.4 shows a trace of the execution of this algorithm for $n = 4$.

n	NumberOfPairs	j	LineOfDiscs
4			●●○○
3	3	1,2,3	●○○●
2	2	1,2	○○●●
1	1	1	○○●●

Table 4.4.3.4 Trace table for the algorithm RearrangeDiscs showing the values of the variables n, NumberOfPairs and j

The basic operation B_{op} for this algorithm is the selection statement containing two comparison operations and one logical operation

```
If LineOfDiscs[j] = green
  And LineOfDiscs[j + 1] = white
```

The number of basic operations B_{op} is 3 when NumberOfPairs = 3, 2 when NumberOfPairs = 2 and 1 when NumberOfPairs = 1.

Therefore the total number of basic operations B_{op} is 1 + 2 + 3 = the sum of the first n - 1 natural numbers.

The sum of the first k natural numbers is $\frac{1}{2} k (k + 1)$.

To obtain the sum of the first n - 1 natural numbers we write $k = n - 1$, which gives $\frac{1}{2} n(n - 1)$.

This result also applies if n is 2, 3, 4, 5, 6 or any natural number greater than *1*.

We can expand this formula to obtain $\frac{1}{2} n (n - 1) = \frac{1}{2} (n^2 - n)$

And for large n, we get $\frac{1}{2} (n^2 - n) \approx \frac{1}{2} n^2$.

This means that the order of growth of this algorithm is $O(n^2)$.

Single licence - Abingdon School

Questions

4 A hotel chain has 10 hotels each with 1000 rooms. Each hotel uses its own laundering facilities to launder n sets of bedsheets every day. Each room supplies one set of bedsheets for laundering.

The average time it takes each hotel to launder a set of bedsheets varies from hotel to hotel by up to ten minutes. If it takes on average c_1 minutes to launder and press one set of bedsheets at the first hotel, c_2 at the second and so on, then the total daily time to launder all sets of bedsheets at a hotel is given by the function f_k where $f_k(n) = c_k n$ and k is a subscript in range 1..10.

For example, the function for the third hotel is $f_3(n) = c_3 n$.

(a) To which category of Big-O do the functions f_k belong if the value of n is always very large?

The hotel chain scraps its laundering system and instead each hotel sends all its bedsheet laundering to a professional laundry once a day. A driver is employed at each hotel to deliver and collect the laundry from the professional laundry daily. The time taken daily is given by the function $g_k(n) = t_k$ where k is a subscript in range 1..10 and t_k is a constant for the kth hotel but whose value may be different for different hotels.

(b) To which category of Big-O do the functions g_k belong?

5 Imagine having a pond with water lily leaves floating on the surface. The lily doubles in size every day and if left unchecked will smother the pond in 30 days, killing all the other living things in the water.

(a) How would you classify the lily's growth rate?

It is decided to leave water lily to grow until it half-covers the pond, before cutting it back.

(b) On what day will this occur?

6 The algorithm shown in *Figure 4.4.3.8* sorts an unsorted array of n integers. What is the Big-O time complexity of this algorithm?

```
Algorithm Sort(a, n)
// Rearranges a set of n integers in a given array, a, in ascending order
// Inputs: Array a[1..n], and n, the number of integers
// Output: Rearranged set of n integers in ascending order
For i ← 2 To n
  x ← a[i]
  j ← i
  While x < a[j - 1]
    a[j] ← a[j - 1]
    j ← j - 1
  EndWhile
  a[j] ← x
 EndFor
```

Figure 4.4.3.8 Algorithm for sorting an array of n integers in ascending order

7 *Figure 4.4.3.9* shows 16 equally-sized squares created on a square piece of paper. One way to do this is to draw one at a time, taking sixteen steps, one per box, in total.

Another way is to fold the paper in half repeatedly until the fold lines form 16 equally-sized boxes then unfold the paper.

(a) If folding the paper is one operation, how many operations are required in total?

(b) What is the Big-O running time of each method?

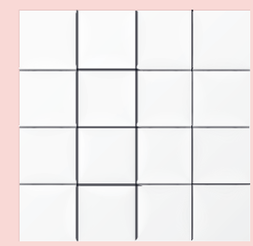


Figure 4.4.3.9 16 equally-sized squares created on a square piece of paper

Single licence - Abingdon School

In this chapter you have covered:

- Big-O notation to express time complexity applied it to cases where the running time requirements of the algorithm grow in:
  - constant time
  - logarithmic time
  - linear time
  - polynomial time
  - exponential time
- Deriving the time complexity of an algorithm.

Single licence - Abingdon School

4 Theory of computation

4.4 Classification of algorithms

Learning objectives:

- *Be aware that algorithmic complexity and hardware impose limits on what can be computed.*

4.4.4 Limits of computation

Just as a roadblock stops traffic, hardware, software and the nature of the problem can impose a limit on what can be computed in a reasonable time or computed at all.

Integer numbers can get very big very fast - see question 7, *Chapter 4.4.2*.

The hardware of a machine determines the limits of the numbers, both real and integer, that can be represented by its word length, e.g. 32 bits, 64 bits.

Software can overcome the limitation imposed by a computer's word length by representing a very large number as a list of smaller numbers, e.g. one digit per memory word.

However, each computer has only a limited amount of memory.

Suppose we have a 1000-bit sequence consisting of all 0's and we want to explore all direct routes between this sequence and the 1000-bit sequence with all 1's which is reached when when one bit is flipped at a time. *Figure 4.4.4.1* shows an example for 3-bit sequences, one such direct route is marked in green.

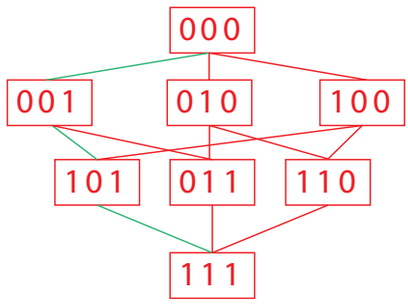


Figure 4.4.4.1 Routes from 000 to 111 for a 3-bit sequence

3 ways

There are 1000! direct routes for a sequence length of 1000 bits.

$1000! = 4.023872600770937735437024339 23 \times 10^{+2567}$

2 ways

If we omit the starting sequence of all 0's, and the final sequence of all 1's, each direct route will consist of a list with 999 elements each a 1000-bit sequence,

1 way

e.g. 000…001, 000…011, …, 011…111.

If we wish to store every 999-element list we will require

$$\frac{999000 \times 1000!}{64} = 6 \times 10^{2571} \text{ 64-bit memory words.}$$

The number of atoms in the observable universe is estimated to be between 4×10^{79} and 4×10^{81} which is considerably less than the memory required for storing these direct routes.

Replace the 0's and 1's by DNA base-pairs and consider a real genome such as that of E. coli with four million base-pairs and the hardware would limit investigations by computer of mutation sequences of E. coli's genome.

Key term

Limited amount of memory: Some problems require more computer memory than is feasible.

Single licence - Abingdon School

Key term

Complexity exponential: Some problems require an exponential amount of work to be done which means that a solution cannot be arrived at in a reasonable amount of time even when the size of the input is relatively small.

Using faster processors or more processors makes little difference when the amount of work per processor is still exponential.

In *Chapter 4.4.5* the total number of subsets of a set with 100 whole numbers is calculated to be

$$2^{100} = 1267650600228229401496703205376$$

If a computer was used to check if a subset's elements summed to half the sum of the entire set of 100 elements and if the computer could check one hundred million subsets per second, the time taken would be

$$\frac{1267650600228229401496703205376}{100000000 \times 60 \times 60 \times 24 \times 365.25} = 401694235375386 \text{ years}$$

which is approximately 4×10^{14} years.

This problem requires an exponential amount of work and takes an unreasonable amount of time. The algorithmic complexity of the problem is exponential.

This limits what can be done.

Even if future computers are 10000 times faster than they are now, the problem would still require an inordinate amount of time to solve, ≈ 40169423538 years.

Even using 10000 processors working in parallel will not improve matters much, the time required would again be of the order of 40169423537 years.
Perhaps a quantum computer could reduce the time taken to one that is reasonable time.
If a quantum computer could put every atom in the universe to work on a different part of the problem, the time taken in years would be of the order of

$$\frac{4 \times 10^{14}}{10^{80}} = 4 \times 10^{-66}$$

assuming each atom could perform a check in 10^{-8} seconds and that there are approximately 10^{80} atoms in the observable universe.

However, if the set now consist of 1000 numbers, the time taken in years would be of the order of

$$\frac{1 \times 10^{146}}{10^{80}} = 4 \times 10^{66}$$

$$\frac{2^{1000}}{100000000 \times 60 \times 60 \times 24 \times 365.25}$$

$$\approx 1 \times 10^{146} \text{ years}$$

Increasing the size of the set by a factor of 10 now makes solving this problem impossible in a reasonable amount of time even if all the atoms in the observable universe are employed.

Questions

1. Explain why algorithmic complexity and hardware impose limits on what can be computed.

In this chapter you have covered:

- Algorithmic complexity and hardware impose limits on what can be computed.

Single licence - Abingdon School

4 Theory of computation

4.4 Theory of computation

Learning objectives:

- *Know that algorithms may be classified as being either:*
  - *tractable - problems that have a polynomial (or less) time solution are called tractable problems*
  - *intractable - problems that have no polynomial (or less) time solution are called intractable problems.*

4.4.5 Classification of algorithmic problems

Tractable and intractable problems

Even if the problem is an algorithmic problem and it is computable, whether a solution can be produced may still depend on the size of the input and the amount of work that the computer may need to do.

Example

Suppose that the task is to split the following set of whole numbers

{18, 23, 27, 19, 22, 21, 66, 26}

into two subsets with equal sums.

After some trial and error, the following two subsets are found

{18, 27, 66} and {22, 23, 26, 21, 19}, each of sum 111.

Alternatively, a computer could be tasked to generate all subsets of the set, sum each of these and then check if the sum is equal to half the sum of all the elements in the sum, i.e. 222/2 = 111.

For this task it is important to know the answer to the question

"how many subsets are there?"

because this will tell us how much work the computer will have to do.

Calculating the number of subsets

For each number in a given set, we can either place the number in one set or in the other set. So there are two possibilities for the first number, two possibilities for the second number, … and two possibilities for the nth number.

Totalling up this is $2 \times 2 \times 2 \times 2 \times \cdots \times 2 = 2^n$ subsets.

For $n = 8$, the total number of subsets is 256 subsets.

But now suppose that the set contains 100 numbers.

The total number of subsets is

$$2^{100} = 1267650600228229401496703205376$$

If a computer was used that could check a hundred million subsets per second, the time taken would be

$$\frac{1267650600228229401496703205376}{100000000 \times 60 \times 60 \times 24 \times 365.25} = 401694235375386 \text{ years}$$

As the size of the problem grows (the number of values in the set) the time taken to solve the problem grows exponentially.

Are there faster methods for solving this problem in all instances?

Information

Wirth's law:
Software is getting slower more rapidly than hardware is getting faster.
Former Intel researcher Randall Kennedy wrote that, "Microsoft Office 7, when deployed on Windows Vista, consumes more than 12 times as much memory and three times as much processing power as the version that graced PCs just seven short years ago, Office 2000."
Kennedy, R. C. (2008) Fat, Fatter, Fattest: Microsoft's Kings of Bloat. InfoWorld Applications, 14 April.

(Used with permission of InfoWorld Copyright 2017. All rights reserved).

Single licence - Abingdon School

Key term

Tractable problem:
A tractable problem is one for which there is a polynomial time algorithm which solves the problem.

The answer is no, the only known method that will always find the exact solution in all cases is a brute-force search through all 2^n subsets.

Tractable problems

Problems for which there is a polynomial time (polynomial time or better) algorithm are called **tractable**.

The symbol **P** is used to denote a class of problems that can be solved by an algorithm causing a computer to run for a "polynomial time", i.e. a time proportional to n^k where k is an integer such as 0, 1, 2 or larger and n is the size of the input or problem. Tractable problems are labelled "easy problems" because they are relatively easy to solve and have time efficient solutions.

Key term

Intractable problem:
An intractable problem is one for which a polynomial time algorithm does not exist or has not yet been found.

Intractable problems

Problems that (may) have no reasonable (polynomial) time solutions are called **intractable** and are considered unsolvable by a standard that requires an algorithm to terminate with the right answer within polynomial time.

We have seen one such problem already, partitioning a set of 100 numbers. Intractable problems are not time efficient. Their time complexity is a^n or $n!$ or worse.

Questions

1 Consider a salesperson who has to drive from city A to three other cities, B, C, D, visiting each exactly once before returning back to the starting city, A.
The distances by road are shown in *Table 4.4.5.1*. The same information is represented on the weighted undirected graph shown in *Figure 4.4.5.1*.

	A	B	C	D
A		80	130	60
B	80		140	150
C	130	140		120
D	60	150	120	

Table 4.4.5.1 Distance in miles between four cities, A, B, C, D

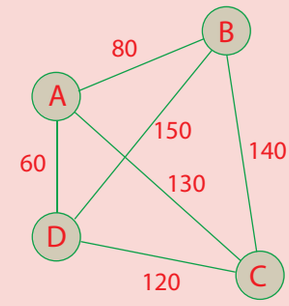


Figure 4.4.5.1 Weighted undirected graph showing road connections and distances between four cities, A, B, C, D

The salesperson would like to travel by the shortest route.
The salesperson needs to calculate the road distance for the round trip from A back to A for each possible route that visits B, C, D once.
There are three choice for the next city after A, then two choices for the next and one choice thereafter, i.e. six possible routes.

(a) Which route is the shortest?

The next job involves travelling from city A to 100 other cities and back to A after visiting each other city exactly once.

(b) How many possible routes are there?

(c) Is this a tractable problem or an intractable problem? Explain.

Single licence - Abingdon School

Questions

2 A single gene encoding one protein molecule is a DNA sequence compromising about one thousand code symbols.
For simplicity, assume a code symbol is 0 or 1 and that we have a one thousand bit sequence consisting of all 0's.
This sequence is to be adapted to some task by changing its one thousand symbols to all 1's.

(a) If one bit is changed at a time, how many possible direct routes are there from the all 0's sequence to the all 1's sequence? (An example of a direct route is indicated in green in *Figure 4.4.5.2* for a 3-bit sequence).

(b) Estimate the minimum amount of time it would take to go from a sequence of all 0's to all 1's if each bit change takes 10^{-8} second.

(c) Why is adapting the sequence of all 0's by flipping one bit at a time considered an intractable problem?

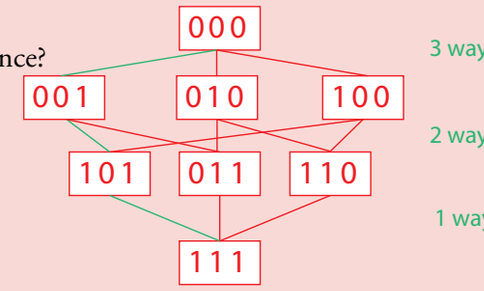


Figure 4.4.5.2 Routes from 000 to 111 for a 3-bit sequence

3 Passwords are generated randomly and stored as n-bit strings, e.g. 100110101010 is a 12-bit string.
The length of each password is chosen to be 64 bits in order to make it difficult to guess. Generating a password can be done in polynomial time.
Why is generating a 64-bit password considered a tractable problem but guessing this password is considered an intractable problem?

Heuristic methods

An approximate solution to an intractable problem may be found in a reasonable time by trading optimality, completeness, accuracy, or precision for speed. The methods employed that achieve approximate solutions use what are called heuristics. A heuristic is a rule of thumb or shortcut that enables a goal to be achieved. Sometimes the heuristic takes the form of a relaxation of some constraint or constraints that apply to the problem, e.g. generating a school timetable might allow as a heuristic some science lessons to be taken in classrooms which are not science laboratories if they are single periods.

In this chapter you have covered:

- That algorithms may be classified as being either:
  - tractable - problems that have a polynomial (or less) time solution are called tractable problems
  - intractable - problems that have no polynomial (or less) time solution are called intractable problems.

Single licence - Abingdon School

4 Theory of computation

4.4 Classification of algorithms

Learning objectives:

- *Be aware that some problems cannot be solved algorithmically.*

Key term

Computable problem:
A problem is computable if there exists an algorithm that solves the problem.

Information

Number of computable solutions for 8-puzzle game:
Whether a solution is possible or not depends on the 8! possible configurations of the 8-puzzle board.
Half of these configurations are called "even permutations" while the other half are called "odd permutations". The usual movements of tiles have a certain symmetry that takes even permutations to even permutations and odd permutations to odd permutations.
However, it is not possible to take an odd permutation to an even permutation and vice versa.
To determine the "oddness" or "evenness" we calculate the number of transpositions required to turn the initial configuration written in a linear way into the goal configuration written similarly.

23861475
23614758 5 transpositions
23614578 1 transposition
23145678 3 transpositions
21345678 1 transposition
12345678 1 transposition

The number of transpositions required to turn 23861475 into 12345678 is 11, an odd number.
The number of transpositions required to turn 12345678 into 12345678 is 0, an even number.

4.4.6 Computable and non-computable problems

Computers can do many useful tasks. However, there are types of human processes that are beyond computation, e.g. falling in love, and determining the beauty of something. Their nature is subjective and "computers don't compute emotions very well". Instead, computers are applied to tasks of a mechanical nature, e.g. computing the sum of the first N natural numbers. Mechanical tasks are ones whose steps can be specified by an algorithm. However, not all mechanical tasks are computable, i.e. have an algorithmic solution.

Computable

A problem is **computable** if there exists an algorithm that solves the problem.

In order for an algorithm to be a solution for a problem, it must always terminate and must always produce the correct output for all possible inputs to the problem. If no such algorithm exists, the problem is **non-computable**.

Non-computable problems

Figure 4.4.6.1 shows an example of the eight-puzzle problem. The puzzle has an initial state I. The objective is to reach to the goal state G from the given initial state I by sliding the numbered tiles where this is possible. Unfortunately, there is no sequence of steps that will achieve this. Therefore, any algorithm that attempts to solve this problem will not terminate for the given initial and goal states. The problem is not solvable, and therefore the problem is **non-computable**.

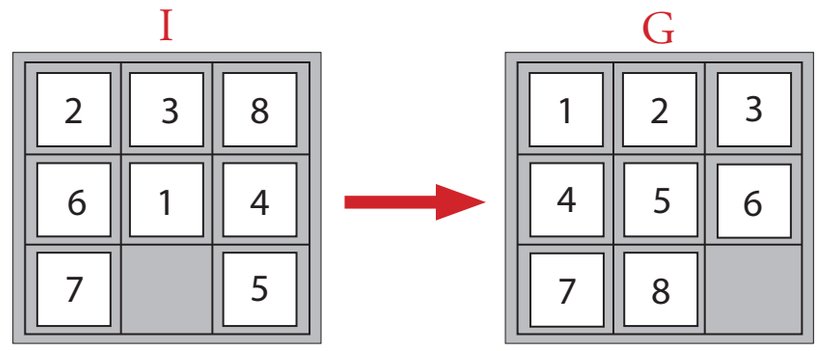


Figure 4.4.6.1 8-puzzle problem initial state, I and goal state, G

Even though the problem has no algorithmic solution for achieving the goal state by sliding the tiles, a solution can be achieved using a screwdriver to lift the tiles from the board before snapping them back into the board in the goal state configuration.

Key term

Non-computable problem:
A problem is non-computable if no algorithm exists which solves the problem.

Single licence - Abingdon School

Key term

Decision problem:
A decision problem is a problem that requires a yes/no answer.

Key concept

Solving algorithmically:
Solving a problem by applying mechanically the rules of the system.

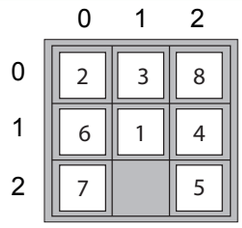


Figure 4.4.6.2

Non-computability and decision problems

Decision problems are problems that require a yes/no answer.

Is it possible to determine in advance if there is a finite number of steps that lead from the initial state to the goal state of the 8-puzzle problem, i.e. an algorithmic solution?

The answer is yes.

We write the board configuration shown in *Figure 4.4.6.2* in a linear way as 23861475, ignoring the blank tile.
A tile is selected in turn and the number of tiles to its right with a smaller digit value is counted. These tiles are coloured yellow in *Figure 4.4.6.3*.
Only if the total count is even can the given goal state G be reached.
Figure 4.4.6.3 shows that the total count is odd for the given initial state I.
This means that after any number of exchanges there will always be at least one tile out of sequence. Hence there is no algorithmic solution which reaches the goal state G from this initial state I.

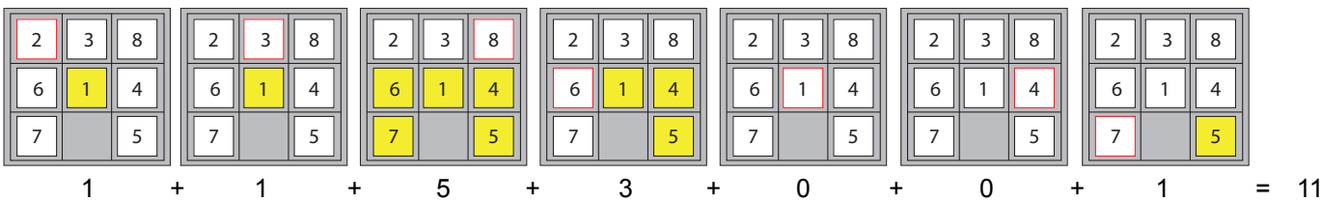


Figure 4.4.6.3 Counting the number of transpositions required to go from I's permutation of the set {1,2,3,4,5,6,7,8} to the permutation which is G

Information

The number of transpositions required to turn the permutation 83465127 into 12345678 is 16 an even number:

83465127
34651278 7 transpositions
34512678 3 transpositions
34125678 2 transpositions
31245678 2 transpositions
12345678 2 transpositions

The number of transpositions required to turn 12345678 into 12345678 is 0, an even number. So the solution is computable because it is an even to even transformation.

Figure 4.4.6.4 shows an initial state for which the solution is computable.

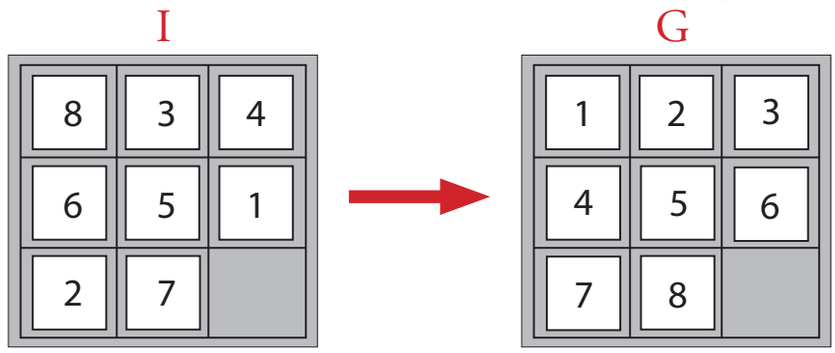


Figure 4.4.6.4 8-puzzle problem for which solution is computable

Again, we write the board configuration shown in *Figure 4.4.6.5* in a linear way as 83465127, ignoring the blank tile.
A tile is selected in turn and the number of tiles to its right with a smaller digit value is counted. These tiles are coloured yellow in *Figure 4.4.6.6*.
Only if the total count is even can the given goal state G be reached.
Figure 4.4.6.6 shows that the total count is even for the given initial state I.
Hence there is an algorithmic solution which reaches the goal state G from this initial state I.

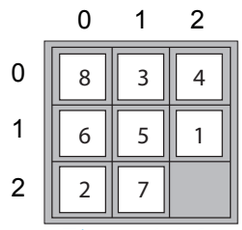
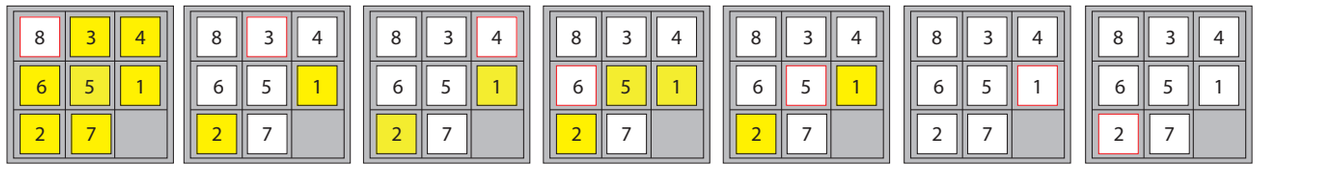


Figure 4.4.6.5

Single licence - Abingdon School



7 + 2 + 2 + 3 + 2 + 0 + 0 = 16

Figure 4.4.6.6 Counting the number of transpositions required to go from I's permutation of the set {1,2,3,4,5,6,7,8} to the permutation which is G

Labelling rows and columns as shown in *Figure 4.4.6.5*, the steps to the solution for the given initial state shown in *Figure 4.4.6.4* are as follows where the coordinates are given in column, row order, and *Figures 4.4.6.7* and *4.4.6.8* show a Right and a Down move, respectively

Right(2,1), Down(1,1), Down(0,1), Left(0,2), Up(1,2), Right(1,1), Right(1,0), Down(0,0), Left(0,1), Left(0,2), Up(1,2), Right(1,1), Up(2,1), Left(2,2), Down(1,2), Right(1,1), Right(1,0), Up(2,0), Left(2,1), Down(1,1), Right(1,0), Down(0,0), Left(0,1), Up(1,1), Left(1,2), Up(2,2)

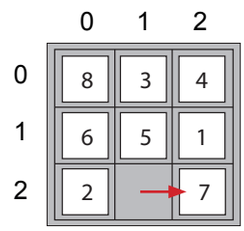


Figure 4.4.6.7

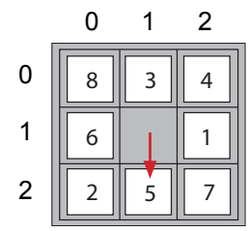


Figure 4.4.6.8

In the case of the 8-puzzle game, no algorithm exists that turns an odd permutation board configuration into an even one and vice versa by sliding the tiles as allowed.
However, an algorithm exists that can tell if a solution is possible.

This isn't always the case. There are many decision problems that are undecidable, i.e. no algorithm exists for these which can give either a yes or a no answer.
For example, given a program and an input for it, does an algorithm exist that is able in general (i.e. not just in specific cases) to decide whether the program with that input will eventually halt or go into an infinite loop? This is called the **Halting Problem**. It is covered more extensively in *Chapter 4.4.7*.

Questions

1 A person who shaves another person is called a barber. A certain village has just one barber who is male. This barber shaves everyone who does not shave himself and no one else.
(a) Who shaves the barber? (b) How would you classify this problem?

2 The following algorithm may or may not go into an infinite loop:

```
Input x
While x is not equal to 1
  If x is even
    Then Divide by 2
    Else Set x to 3x + 1
  EndIf
EndWhile
```

(a) Does this algorithm terminate for
(i) x = 15? (ii) x = 105?
(b) Does it terminate for any positive x?

Single licence - Abingdon School

Non-computability and undecidable decision problems

If the problem is a decision problem and a solution to the problem is non-computable, then the problem is said to be undecidable. Demonstrating that a decision problem is undecidable can in many instances be done by contradiction.

Worked example

Look at the following statement and say whether it is true or false:

This statement is false.

Solution

If the statement is true, then the statement is telling us that it is false. On the other hand, if the statement is false, then the statement is telling us that it is true. Both interpretations are contradictory. The conclusion is therefore that the answer is undecidable.

Information

A software-based 8-puzzle to explore the two configurations covered in this chapter, one non-computable and one computable may be downloaded from www.educational-computing.co.uk/aqacs/alevelcs.html

You are not required to understand why there are solutions for some configurations and not for others. The key point to carry away is that a problem is computable if there exists an algorithm that solves the problem and non-computable if not.

In this chapter you have covered:

- The nature of problems that cannot be solved algorithmically
  - These are not problems that cannot be solved because of resource limitations. They are unsolvable because logic prohibits inventing an algorithm that computes a solution.

Single licence - Abingdon School

4 Theory of computation

4.4 Classification of algorithms

Learning objectives:

- *Describe the Halting Problem (but not prove it), that is the unsolvable problem of determining whether any program will eventually stop if given particular input.*
- *Understand the significance of the Halting Problem for computation.*

Key term

Algorithm:
An algorithm is a procedure for accomplishing a certain task in a finite number of steps.

Information

The Halting Problem is undecidable and therefore cannot be solved in any amount of time. Alan Turing proved that it is impossible to write a program to solve the Halting Problem. For this he used an abstract machine called a Turing machine.
One can conclude from Turing's proof that no physical computing device obeying reason would be able to solve the Halting Problem.

Task

Does the algorithm in *Figure 4.4.7.2* terminate for $x = 15$? Does it terminate for $x = 105$? Does it terminate for any positive x?

Key term

Pseudo-code:
Pseudo-code refers to any informal syntax that resembles a programming language.

4.4.7 Halting problem

Halting problem

The Halting Problem asks

> "*Is it possible in general to write a program that can tell, given a description of another program and its inputs and without executing this program, whether the given program with its given inputs will halt?*"

It sheds light on why it is not possible in general to predict in advance when a computer will get stuck in some loop, therefore enabling such a situation to be avoided. Predicting in advance would require a method that could decide, just by inspecting another program and its input, whether or not the program would go into an infinite loop on the given input; an infinite loop can signify a bug or that the problem is undecidable on the given input. A general method, *Halt*(*P*, *x*), to do this for any program *P*, and its input *x*, does not exist and can be proven to not exist. The Halting Problem decider shown in *Figure 4.4.7.1* is impossible.

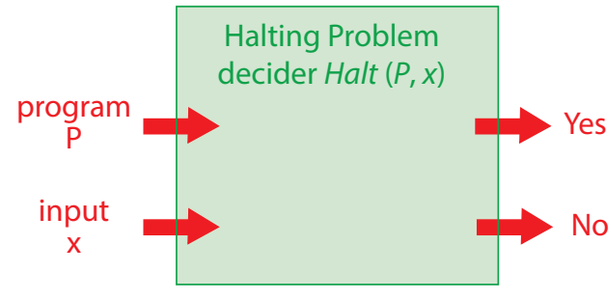


Figure 4.4.7.1 Halting Problem decider Halt(P, x) applied to a description of program P and its input x

Figure 4.4.7.2 shows an algorithm expressed in pseudo-code that may or may not go into an infinite loop.

```
Input x
While x is not equal to 1
  If x is even, divide x by 2
  Else set x to 3x + 1
EndWhile
```

Figure 4.4.7.2 Algorithm that may or may not go into an infinite loop

The Halting Problem is not the only undecidable problem. There are many other problems of an objective nature that cannot be solved or decided by computer.
For example, writing an algorithm or program that will compare any two programs and determine whether or not they are equivalent, i.e. produce the same output for a given output.

Single licence - Abingdon School

Programming Task

Code the algorithm in *Figure 4.4.7.2* in a programming language that you are most familiar with.

x	k	$1 + kx$
1	5	6
5	1	6
1	27	28
27	1	28
3	9	28
9	3	28
3	165	496
5	99	496
9	55	496
11	45	496
15	33	496

Table 4.4.7.1 Values of x and k which generate some perfect numbers when plugged into formula 1 + kx

Programming Task

Write a program based on the algorithm in *Figure 4.4.7.3* to find a perfect number and try with $x = 1$ and $s = 7, 29, 497$. Next try $x = 2$ and $s = 2$.

How long should one wait?

Often it is difficult to tell whether a program has entered an infinite loop, because it may be that the program just needs a little longer to do its calculations.

For example, in mathematics, a perfect number is defined as a positive integer that is the sum of its positive divisors excluding the number itself.

The number 6 is a perfect number because its divisors are 1, 2 and 3 and $1 + 2 + 3 = 6$.

The next two perfect numbers are 28 and 496.

Table 4.4.7.1 shows that we can express these perfect numbers as $1 + kx$, where k and x are integers such that $k, x > 0$.

Given x, is there a perfect number n of the form $1 + kx$ for some $k > 0$ and $k < s$?

The following algorithm finds perfect numbers 6, 28, 496, 8128 when $x = 1$ and $s = 5, 27, 29, 497$, respectively.
However, the algorithm takes many steps to find the perfect number 8128.

When $x = 2$ the algorithm does not terminate; it searches for odd perfect numbers. Finding an odd perfect number has defeated the best mathematicians to date.

```
Algorithm FindPerfectNumber(x, s : Integer)
//Given x, is there a perfect number n greater than s
// Inputs: x, s
//Output: Perfect number n
Found ← False
n ← 1
While Not Found
  Sum ← 0
  i ← 1
  While i < n
    If (n Mod i) = 0 {Find next factor}
      Then Sum ← Sum + i {Add factor to Sum}
    EndIf
    i ← i + 1
  EndWhile
  {If true then found first perfect number after s}
  If (Sum = n) And (n >= s)
     Then Return n {Exit algorithm and return perfect number}
  EndIf
  If Not Found
    Then n ← n + x // Try next n
  EndIf
EndWhile
```

Figure 4.4.7.3 Algorithm to find perfect numbers

Single licence - Abingdon School

Figure 4.4.7.4 shows pseudo-code for a program `TestProgram` that calls a procedure `Test`. Procedure `Test` takes as input a function, `F`, and a single parameter, `x`, and executes `F` on input `x`.

```
Program TestProgram
  Type
    TFunction = Function (y : Integer) : Integer
    {Declare a type which is a function with a single integer
     parameter y and which returns an integer result}
  Function A (Value : Integer) : Integer
    Return 2 * Value {Function returns 2 times Value}
  EndFunction
  Procedure Test (F : TFunction; x : Integer)
  {Takes as input a function F and executes function F on input
   parameter x}
    Output F(x)
  EndProcedure
Begin
  Test(A, 6) {Executes function F on input 6}
EndProgram
```

Figure 4.4.7.4 Pseudo-code program to execute a function on its input

How useful is a procedure that is designed to accept as input another subprogram, e.g. a function, plus its input?

It is very useful, because subprograms such as the function `FindPerfectNumber` which halt for values of $x = 1$ and $s = 7$, 29 and 497 but not for values of $x = 2$ could be submitted to a differently constructed procedure `Test` that would be used to determine without executing `FindPerfectNumber` if `FindPerfectNumber` halts for the given input values of x and s. This would be extremely useful, because it would solve the problem of having to wait an indeterminate amount of time for the subprogram to terminate.

The significance of the Halting Problem for computation

Is it possible to construct such a procedure `Test` whose purpose is to accept as input a description of a subprogram or program and its input, and to determine without executing the subprogram or program whether it will halt on this input for any arbitrary program? Unfortunately, the answer is no. This has serious implications for algorithms in general. No decider program exists that we could use on the algorithm. Instead, to discover if an algorithm will halt, running it on its range of specified inputs may be the only option. But when running the algorithm, if we find ourselves waiting for a long time, then do we conclude that the algorithm is stuck in a loop or do we conclude that we have not allowed enough time for the algorithm to calculate its output?

Secondly, the Halting Problem dispels the myth that the only limits on computation are those of the speed of the processor and other components. Faith in the advance of technology, of better programming tools, of better-skilled programmers to enable a program to solve any problem is misplaced.

Information

Turing's proof that the Halting Problem is undecidable relies on proof by contradiction. This proof by contradiction relies on logical assertions such as the following which contradict each other:
"The following statement is true."
"The preceding statement is false."
Watch the YouTube video at https://www.youtube.com/watch?v=wGLQiHXHWNk to get a feel for how Turing could prove that the Halting Problem is undecidable.

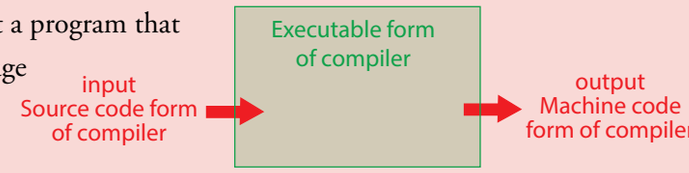
Key term

Halting Problem:
It is not possible in general to write a program that can tell, given any arbitrary program and its inputs and without executing this program, whether the given program with its given inputs will halt.
There is no purely mechanical method (i.e. no algorithm) which is guaranteed to solve the Halting Problem in every case.

Single licence - Abingdon School

Questions

1 A compiler is a program that takes as its input a program that needs to be translated from a high-level language (e.g. Pascal) to a low-level language (e.g. machine language).



In general, a program can process any data, so it can have as its input a program to process (i.e. the program is treated as data). A Compiler *C* is available in both executable and source code forms.
Can a compiler *C* compile the source code form of itself?

2 The question "Can a compiler *C* compile the source code form of itself?" could be put to the Halting Problem decider shown in *Figure 4.4.7.1.* The machine code form of the compiler is fed to the P input and the source code form is fed to the x input. Can the Halting Problem decider answer this question?

3 An algorithmic problem which outputs either yes or no can be solved by an algorithm containing a table which maps each of its inputs to the appropriate output.
A virus detection program cannot detect if a program is a virus for all possible programs.
(a) Give one reason why this is the case.
(b) Most computers have virus detection software installed which works quite well most of the time.
Explain how this could be achieved.

4 By inspecting the following Basic program, it is easy to conclude that it does not halt.

```
10 Print "Hello world"
20 GOTO 10
```

To remove the need to inspect a program every time, a machine is designed to automate the process of inspecting a given program to determine if it never halts on certain inputs. The manufacturer of such a machine claims that it will never fail no matter what program it is asked to check.
Why is the claim of the manufacturer not believable?

In this chapter you have covered:

- The Halting Problem (but not proven it), that is the unsolvable problem of determining whether any program will eventually stop if given particular input.
- The significance of the Halting Problem for computation.

Single licence - Abingdon School

4 Theory of computation

4.5 A model of computation

Learning objectives:

- *Be familiar with the structure and use of Turing machines that perform simple computations*
- *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*

  One of the states is called a start state and states that have no outgoing transition are called halting states
- *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*
- *Be able to explain the importance of Turing machines and the Universal Turing machine to the subject of computation*

4.5.1 Turing machine

Structure

The novel technical apparatus called a Turing machine that Alan Turing developed as a theoretical abstract machine in the 1930s embodies the principles underpinning the dominant technology of the modern world, the digital general purpose computer.

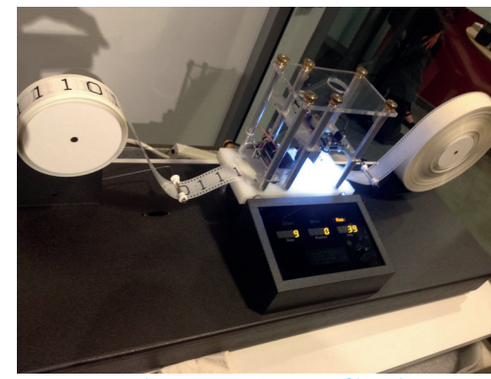


Figure 4.5.1.1 Realisation of a Turing machine

Figure 4.5.1.1 shows a physical model of a Turing machine with tape, sensing read/write head and control unit (see https://www.youtube.com/watch?v=E3keLeMwfHY for a demonstration).

Figure 4.5.1.2 shows an abstract Turing machine for determining the parity of the binary pattern written on the tape.

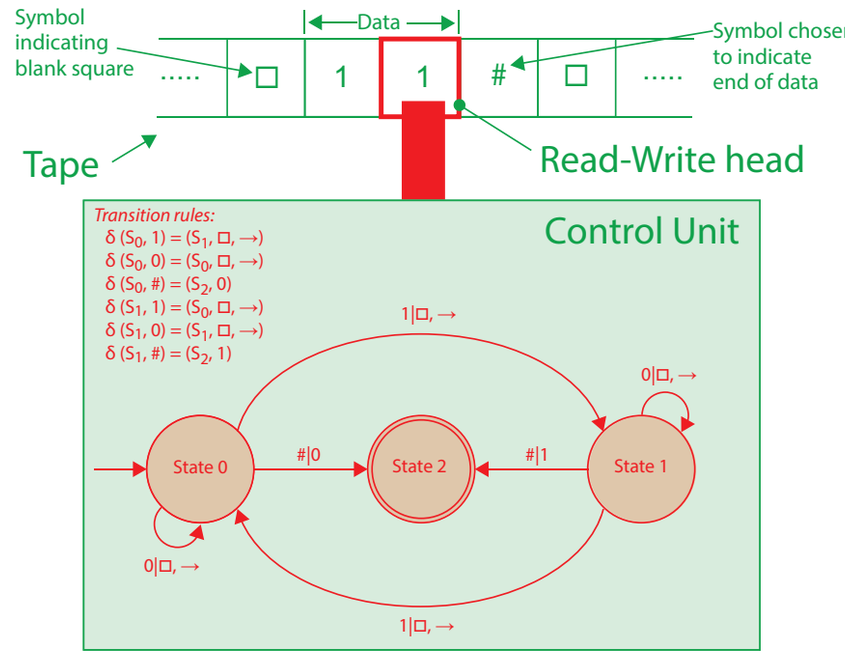


Figure 4.5.1.2 Abstract Turing machine

A Turing machine consists of a read-write head, an infinitely long tape and a control unit. The particular Turing machine shown in *Figure 4.5.1.2* can be in one of three states at any one time, State 0, State 1 or State 2, abbreviated to S_0, S_1, S_2, respectively. State S_2 is a halting or stop state. The machine starts in state S_0.

Single licence - Abingdon School

A Turing machine also has a set of instructions which tell the machine what to do when it finds itself in a particular state. These instructions are called **transition rules**. Transition rules can be expressed in the form of a **transition function** δ as follows

δ (current state, input symbol) = (next state, output symbol, movement)

Table 4.5.1.1 shows the transition rules for the Turing machine in *Figure 4.5.1.2* expressed as a set of transition functions δ. This Turing machine calculates the parity of the input bit string. This string is shown on the tape in *Table 4.5.1.1* as 11 in the first row. The output 0 is shown in the last row. The output indicates even parity. If the tape input bit string was 111 then the output would be 1, indicating odd parity.

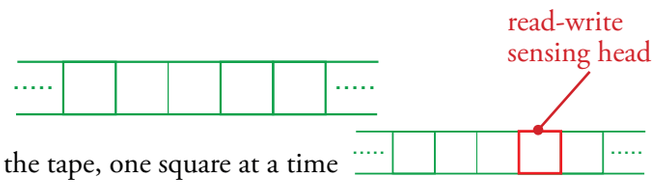
..... □ 1 1 # □	Before reading any symbols on the tape the machine starts in state S_0. The read-write head starts on the first non-blank symbol on the far left of the tape which happens to be 1. A blank square is indicated with □
$\delta(S_0, 1) = (S_1, \square, \rightarrow)$	If the state is S_0 and the symbol under the read-write head is 1, then erase this 1, replace with a □ (i.e. erase square), move read-write head one square to the right (achievable by moving tape left on square), and change to state S_1
$\delta(S_0, 0) = (S_0, \square, \rightarrow)$	If the state is S_0 and the symbol under the read-write head is 0, then erase this 0 and replace with a □ (i.e. erase square), move read-write head one square to the right, remain in state S_0
$\delta(S_0, \#) = (S_2, 0)$	If the state is S_0 and the symbol under the read-write head is #, erase this #, replace with 0 then change to state S_2 and halt because S_2 is a Halt state
$\delta(S_1, 1) = (S_0, \square, \rightarrow)$	If the state is S_1 and the symbol under the read-write head is 1, erase this 1, replace with □, move read-write head one square to the right then change to state S_0
$\delta(S_1, 0) = (S_1, \square, \rightarrow)$	If the state is S_1 and the symbol under the read-write head is 0, erase this 0, replace with □, move read-write head one square to the right (achievable also by moving tape left one square), remain in state S_1
$\delta(S_1, \#) = (S_2, 1)$	If the state is S_1 and the symbol under the read-write head is #, erase this #, replace with 1 then change to state S_2 and halt because S_2 is a Halt state
..... □ □ □ 0 □	In the Halt state S_2 the read-write is over the square containing the output. The tape shows 0 which indicates that the parity of the input bit string is even.

Table 4.5.1.1 Transition rules for Turing machine shown in Figure 4.5.1.2

A Turing machine can be viewed as a computer with a single fixed program

A Turing machine M consists of a

- finite set of states, e.g. S_0, S_1, S_3
- a finite alphabet of symbols, e.g. □, 0, 1, #
- an infinite tape with marked-off squares, e.g.
- a sensing read-write head that can travel along the tape, one square at a time



In addition, at the heart of the machine is a state transition diagram or its equivalent table of transition rules for the instructions that the machine steps through. One of the states is called a start state and states that have no outgoing transition are called halting or stop states.

Single licence - Abingdon School

Figure 4.5.1.3 shows the state transition diagram for the Turing machine that determines the parity of the bit string placed on the tape.

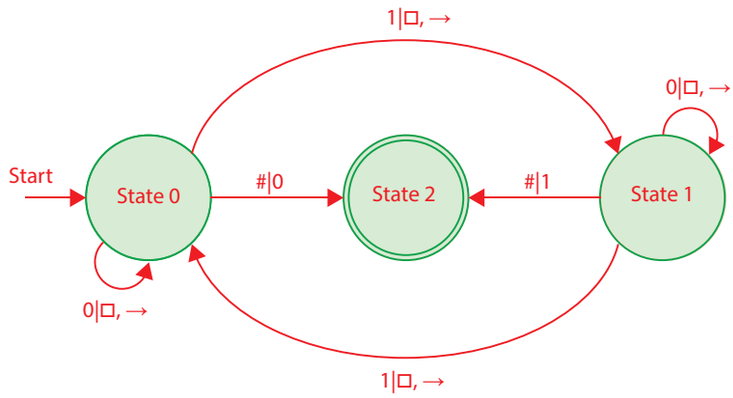


Figure 4.5.1.3 State transition diagram

The state transition diagram is a directed graph with nodes representing the states that the Turing machine may be in during its operation.
State 0 is the start state. State 2 is a halting or stop state. A halting state has no outgoing transitions.

An edge leading from one state to another is called a **transition**, e.g. State 0 to State 1.
An edge is labelled with a code in the form

symbol | symbol, movement, e.g. 1 | □, → where 1 and □ are symbols and → indicates direction of movement, one square to right.

The *symbol* before | is called the **transition's trigger**. It denotes the symbol that is read from the tape.
The *symbol* after | is called the **transition's action**. It denotes the symbol written onto the tape. No symbol means no action.
When the action symbol is the blank symbol □, the square on the tape under the read-write head becomes a blank square.
The movement part indicates the direction to move, either one square to the right (→) or one square to the left(←). No movement is indicated by omitting the arrow. Both movement and symbol may be omitted in which case the transition just involves a state transition including no change of state.

The precise meaning of the transition from State S to State T labelled a | b, → is as follows

During its operation, whenever the Turing machine is in State S, and a is the symbol sensed at that moment by the read-write head, the machine will erase the symbol a, writing b in its place, then will move one square to the right, and enter State T.

There must be no ambiguity as to the machine's next move, therefore no two transitions with the same trigger must emerge from the same state - *Figure 4.5.1.4* is an example of a machine with ambiguity.

Key term

Transition rules:
The following are two examples of transition rules.
A transition rule T1 takes the following form for Turing machine M
T1: If M reads symbol a when in state S_1, then the reading head moves one square to the right and M enters state S_2;
A transition rule T2 takes the following form for Turing machine M
T2: If M reads symbol b when in state S_0, it erases b, writes an a then M enters state S_1.
Symbols a and b could be 0 or 1 or # for example.

Key term

Transition function:
A transition function is just a compact way of expressing a transition rule.

$$\delta(S_0, 1) = (S_1, 0, \rightarrow)$$

Information

If no movement is required the movement action is omitted, from the transition action
e.g. 1 | □

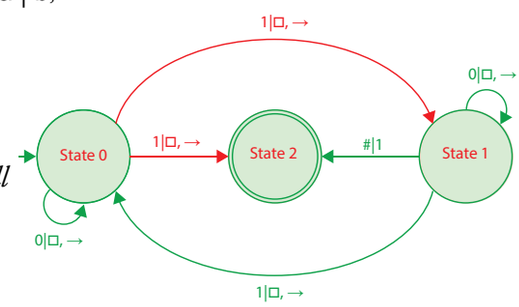


Figure 4.5.1.4 State transition diagram with ambiguity

Single licence - Abingdon School

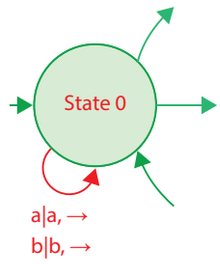


Figure 4.5.1.5 Multiple labels on a transition

One of the states is marked as the start state, indicated by a small arrow entering the state from the left as shown in *Figure 4.5.1.3*.

Halting states are indicated with a double circle as shown in *Figure 4.5.1.3*.

Several labels may be attached to a single transition as shown in *Figure 4.5.1.5*.

It is also possible to have multiple triggers on a transition as shown in *Figure 4.5.1.6*. For example, the state Go Left has an edge labelled 1, 0 | ← meaning if the symbol sensed by the read-write head is a 1 or a 0 move one square to the left.

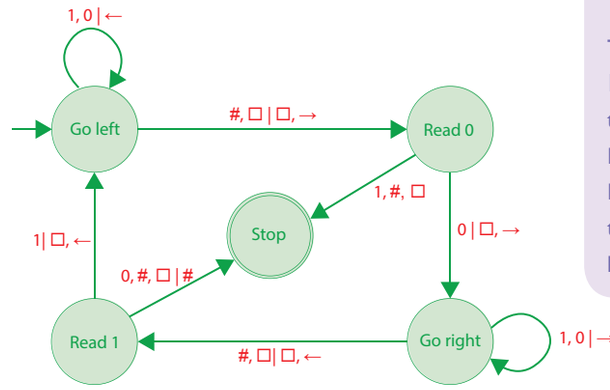


Figure 4.5.1.6 Multiple triggers on a transition

Information

The edge labelled 1, #, □ just involves a state transition from state Read 0 to state Stop. Nothing is written to the tape and the tape head is not moved.

Information

Extract from Alan Turing's 1936 paper "On Computable Numbers, with an Application to the ENTSCHEIDUNGSPROBLEM":

"At any moment there is just one square, say the r-th, bearing the symbol B(r) which is "in the machine". We may call this square the "scanned square". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its state the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the state S_n and the scanned symbol B(r).
This pair S_n, B(r) will be called the "configuration": thus the configuration determines the possible behaviour of the machine."
It is this paper that amongst other things introduces and describes the Turing machine, an abstract machine for reasoning about what can and cannot be computed.

A Turing machine can be viewed as a computer with a single fixed program if we consider

- the state transition diagram or transition rules table to be its software
- the tape, read-write head, the mechanism that moves through the state transition diagram changing states and controlling the head's reading, writing, erasing, and motion to be its hardware (computer)

The computer or hardware is thus the same for all Turing machines but the state transition diagram or transition rules table is changed to create a different Turing machine with a different action.

Therefore, each Turing machine is programmed via its state transition diagram or transition rules table to carry out a specific computation.

The Turing machine programmed with the state transition diagram shown in *Figure 4.5.1.3* performs a computation which determines the parity of the input bit string. It is a single purpose computer.
If we require a different computation the state transition diagram or transition rules table must be replaced with a different one, i.e. the Turing machine must be reprogrammed.

Single licence - Abingdon School

Equivalence between a transition function and a state transition diagram

Figure 4.5.1.7 shows a state transition diagram and its equivalent set of transition functions for a palindrome checker for bit strings, e.g. 010. A palindrome is a string that is the same when reversed, e.g. 010.

$\delta(S_s, 0) = (S_0, \square, \rightarrow)$
$\delta(S_s, 1) = (S_1, \square, \rightarrow)$
$\delta(S_s, \square) = (S_y, \square, \rightarrow)$
$\delta(S_0, 0) = (S_0, 0, \rightarrow)$
$\delta(S_0, 1) = (S_0, 1, \rightarrow)$
$\delta(S_0, \square) = (S_{t0}, \square, \leftarrow)$
$\delta(S_1, 0) = (S_1, 0, \rightarrow)$
$\delta(S_1, 1) = (S_1, 1, \rightarrow)$
$\delta(S_1, \square) = (S_{t1}, \square, \leftarrow)$
$\delta(S_{t0}, 0) = (S_r, \square, \leftarrow)$
$\delta(S_{t0}, 1) = (S_n, 1, \leftarrow)$
$\delta(S_{t0}, \square) = (S_y, \square, \leftarrow)$
$\delta(S_{t1}, 0) = (S_n, 0, \leftarrow)$
$\delta(S_{t1}, 1) = (S_r, \square, \leftarrow)$
$\delta(S_{t1}, \square) = (S_y, \square, \leftarrow)$
$\delta(S_r, 0) = (S_r, 0, \leftarrow)$
$\delta(S_r, 1) = (S_r, 1, \leftarrow)$
$\delta(S_r, \square) = (S_s, \square, \rightarrow)$

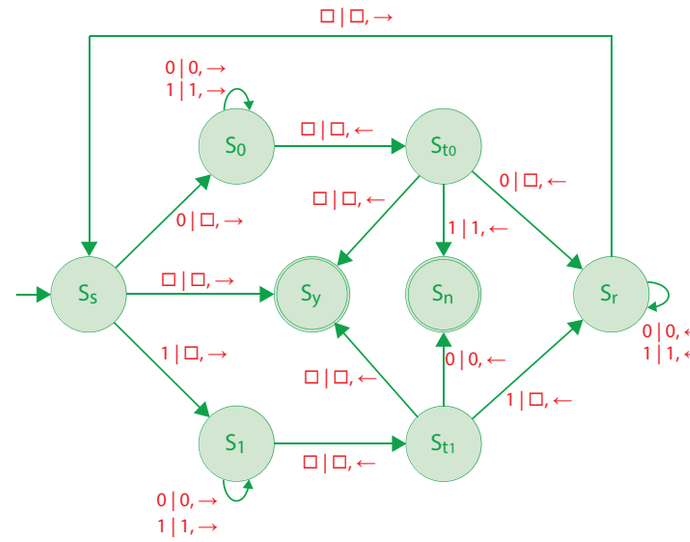


Figure 4.5.1.7 State transition diagram and its equivalent set of transition functions

Hand-tracing simple Turing machines

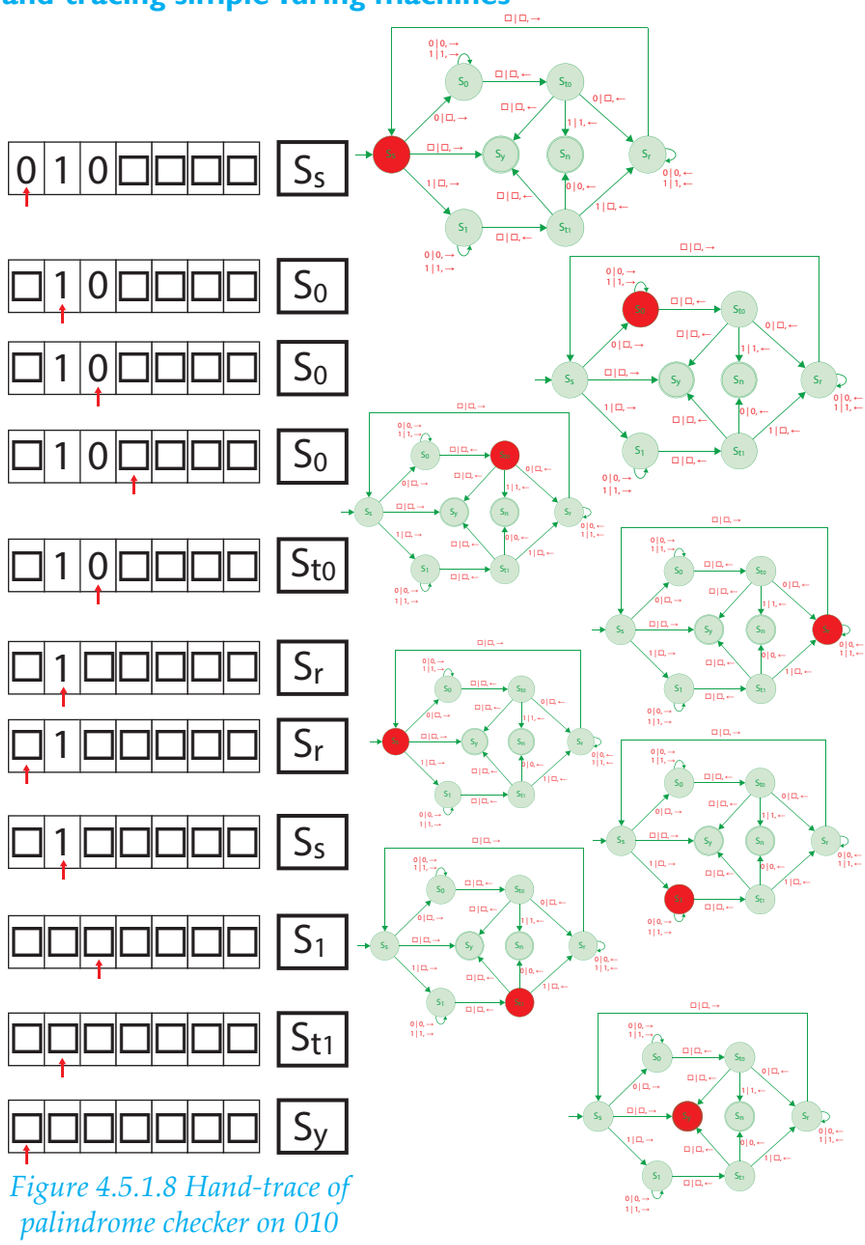


Figure 4.5.1.8 Hand-trace of palindrome checker on 010

Figure 4.5.1.8 shows a hand-trace of the palindrome checker on input bit string 010.

The Turing machine starts in state S_s with the string 101 on the tape. All other squares contain the blank symbol, □.

The read-write head starts on the left hand symbol of the string and is indicated with a red upward arrow.

The read-write head senses a 0 on the tape. Using the transition functions, a match with $\delta(S_s, 0)$ is made. The rule expressed by this transition function tells the machine to erase the 0, write □, then move to the square to the right, and change state to S_0.

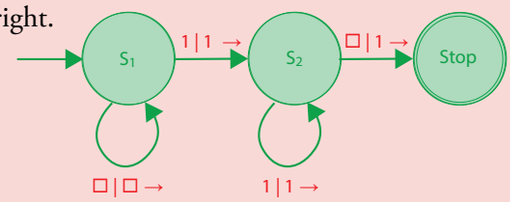
The machine continues to match current state and symbol sensed with a transition function, applying the rule specified by the function until a halting state is reached. In this case, state S_y.

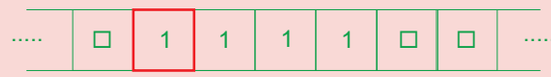
State S_y is reached if the bit string is a palindrome which is the case for 010. If it is not a palindrome then the machine halts in state S_n.

Single licence - Abingdon School

Questions

1 Show the state of the tape after the Turing machine is executed on a tape containing four 1s.
The read-write head starts on the leftmost 1. The symbol → means move the read/write head one square to the right.





2 The following Turing machines are expressed as sets of transition rules.
Draw the state transition diagram for each. Assume that the starting state for each is state 1.

(a) $\delta(S_1, b) = (S_1, a, \rightarrow)$
$\delta(S_1, a) = (S_1, b, \rightarrow)$
$\delta(S_1, \square) = (\text{Halt}, \leftarrow)$

(b) $\delta(S_1, 1) = (S_1, x, \rightarrow)$
$\delta(S_1, \square) = (S_2, x, \leftarrow)$
$\delta(S_2, x) = (S_3, 1, \rightarrow)$
$\delta(S_2, 1) = (S_2, 1, \leftarrow)$
$\delta(S_2, \square) = (\text{Halt}, \square, \leftarrow)$
$\delta(S_3, \square) = (S_2, 1, \leftarrow)$
$\delta(S_3, 1) = (S_3, 1, \rightarrow)$

3 Complete a copy of *Table 4.5.1.2* by hand-tracing the Turing machine whose transition functions are given in *Figure 4.5.1.7* and reproduced below in *Table 4.5.1.4*. You may need more table rows than shown.
The machine starts in state S_s with the string 011 on the tape. The initial configuration of the machine has been completed for you in step 1.

Step	Tape	State
1.	0 1 1 □ □ □ □	S_s
2.		
3.		
4.		
5.		
6.		

Table 4.5.1.2

4 Complete a copy of *Table 4.5.1.3* by hand-tracing the Turing machine whose transition functions are given in *Table 4.5.1.4*.
You may need more table rows than shown.
The machine starts in state S_s with the string 101 on the tape.
The initial configuration of the machine has been completed for you in step 1.

Step	Tape	State
1.	1 0 1 □ □ □ □	S_s
2.		
3.		
4.		
5.		
6.		

Table 4.5.1.3

$\delta(S_s, 0) = (S_0, \square, \rightarrow)$	$\delta(S_{t0}, 0) = (S_r, \square, \leftarrow)$
$\delta(S_s, 1) = (S_1, \square, \rightarrow)$	$\delta(S_{t0}, 1) = (S_n, 1, \leftarrow)$
$\delta(S_s, \square) = (S_y, \square, \rightarrow)$	$\delta(S_{t0}, \square) = (S_y, \square, \leftarrow)$
$\delta(S_0, 0) = (S_0, 0, \rightarrow)$	$\delta(S_{t1}, 0) = (S_n, 0, \leftarrow)$
$\delta(S_0, 1) = (S_0, 1, \rightarrow)$	$\delta(S_{t1}, 1) = (S_r, \square, \leftarrow)$
$\delta(S_0, \square) = (S_{t0}, \square, \leftarrow)$	$\delta(S_{t1}, \square) = (S_y, \square, \leftarrow)$
$\delta(S_1, 0) = (S_1, 0, \rightarrow)$	$\delta(S_r, 0) = (S_r, 0, \leftarrow)$
$\delta(S_1, 1) = (S_1, 1, \rightarrow)$	$\delta(S_r, 1) = (S_r, 1, \leftarrow)$
$\delta(S_1, \square) = (S_{t1}, \square, \leftarrow)$	$\delta(S_r, \square) = (S_s, \square, \rightarrow)$

Table 4.5.1.4

Single licence - Abingdon School

Be able to explain the importance of Turing machines and the Universal Turing machine to the subject of computation

Now that we have an understanding of the operation of a Turing machine, we should explore why Turing machines are so important.

Rigorous definition of an algorithm

In *Chapter 4.1.2* an informal definition of an algorithm was given as

" An algorithm is a precise description of steps necessary to accomplish a certain task or solve a particular problem."

Another informal definition is

"A sequence of steps that can be followed to complete a task and that always terminates"

It is sufficient to have just an intuitive understanding of what is meant by an algorithm and computability if we only want to show that some specific result can be computed by following a specific algorithm.

However, this is not enough if we wish to show that a desired result is not computable, i.e. no algorithm exists which solves the given problem.

This requires us to assess whether something is computable by answering the question, "Does a Turing machine exist for this computation which halts on every possible input allowed for this function (computation) when executed?"

If such a Turing machine exists, then this Turing machine represents an algorithm that solves the problem.

It represents this algorithm in its transition rules or instructions, i.e. its program.

If a Turing machine exists, then the solution to the problem is computable.

Putting it another way, a task is computable if and only if it can be computed by a Turing machine.

Every algorithm can be represented as a Turing machine program.

If a function is computable, then a Turing machine exists that computes the function.

The algorithm that computes the function is coded in the transition rules or instructions for this Turing machine. This has become known as the Church–Turing thesis.

The Church–Turing thesis states that "if an algorithm exists, then there is an equivalent Turing machine for that algorithm". All models of computation are the same as Turing machines is an alternative way of expressing this thesis.

Key point

Turing machines are usually interpreted either as computing some function, say the square of an integer, or as accepting a language, i.e. set of strings. A Turing machine to compute a function receives its (coded) input as the initial content of its tape or tapes, and produces its output as the final content of its tape or tapes, if it halts. If it doesn't halt on all inputs specified for the function, then no effective procedure exists that will calculate the output of the function for all inputs, i.e. no algorithm exists. The function is not computable.

Information

Turing's 1936 paper concluded that "a number is computable if its decimal can be written down by a machine."

Here number refers to elements of $\mathbb{R}$ the real numbers, generally thought of as corresponding to the measurement of continuous quantities like distance and time, and expressible as infinite decimal fractions.

The insight that Turing had was to enumerate each Turing machine, by assigning it a natural number which he called its description number.

Given that the set of integers is countably infinite whilst the set of real numbers $\mathbb{R}$ is not, there must be some real numbers that are not computable. Those that are can be computed by the corresponding Turing machine.

The first conclusion is that Turing showed that there are some things that cannot be computed.

The second is that Turing showed how to encode operations on integers, by integers (the decimal expansion of a real number after the point is a string of digits, which can be treated as an integer).

Thus Turing had perceived that operations and numerical data can be codified in a similar manner with integers. In current parlance, there is no difference between data and programs as far as how they are codified.

This insight led Turing to invent the universal machine, the concept that underpins today's general purpose computer.

Single licence - Abingdon School

Information

A well-defined algorithm is supposed to terminate yet the subject of Turing's 1936 paper is calculating the digits of an infinite decimal, e.g, π. a process involving a Turing machine which never terminates. (Page 15 Unit 2 defines a real number as a decimal with an infinite decimal expansion).
Turing also shows in his paper how a Turing machine formalizes the concept of an algorithm.
So what is wrong?
Actually, there is no contradiction because Turing's model for computing a real number demands that the calculation of any one digit is indeed guaranteed to be a terminating task. Computation of the computable number corresponds to the calculation of each digit successively.
What is important is that the same finite package of information, i.e. how to calculate each digit, can be coded as a single 'description number' or specific Turing machine which suffices for infinitely many different digits.

Universal Turing machines

Suppose that we have a set of transition rules expressed as transition functions in table form as follows

$$\delta\ (S_0, 0) = (S_1, 1, \rightarrow)$$

$$\delta\ (S_1, 0) = (S_2, 1, \leftarrow)$$

et cetera

and suppose we write these more compactly as follows

$$S_0 0\ S_1 1 \rightarrow$$

$$S_1 0 S_2 1 \leftarrow$$

et cetera

Furthermore, suppose that S_n, the state n, is replaced by the letter 'D' followed by the letter 'A' repeated n times, and each symbol, a_j (e.g. 0, 1, #, □) by the letter 'D' followed by 'C' repeated j times. If no tape movement is specified we use 'N'. If tape movement is specified we use 'L' for ←, and 'R' for →.

We obtain

DDDADCR; DADDAADCL; et cetera

If finally letter 'A' is replaced by 1, 'C' by 2, 'D' by 3, 'L' by 4, 'R' by 5, 'N' by 6, and ';' by 7 a description of the machine is obtained in numeric form, called a **description number**

33313257313311324 et cetera

Suppose that we have only two rules, the two expressed above for which the description number is 33313257313311324.
In binary, 33313257313311324 is

1110110010110100011101100100100011110001001101001011100

Now if the input to this Turing machine is 101, both program (transition rules table) and data expressed numerically are indistinguishable in form, i.e. both are binary.

Both program and data may be placed on the same tape as shown in *Figure 4.5.1.9*.

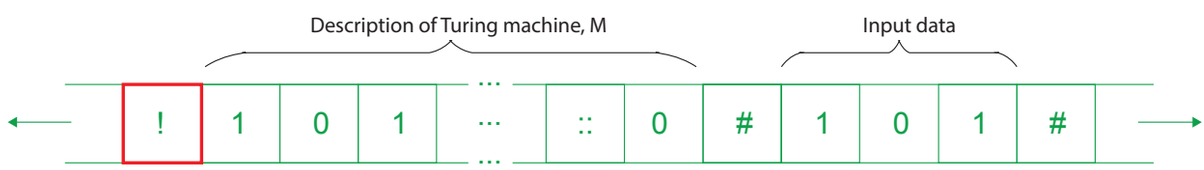


Figure 4.5.1.9 Tape containing both program and input data

Information

Description number:
A numerical encoding of the definition of a Turing machine.

Single licence - Abingdon School

Figure 4.5.1.10 shows a control unit which contains a program to read and execute the instructions of a Turing machine M placed on the same tape as the input data.

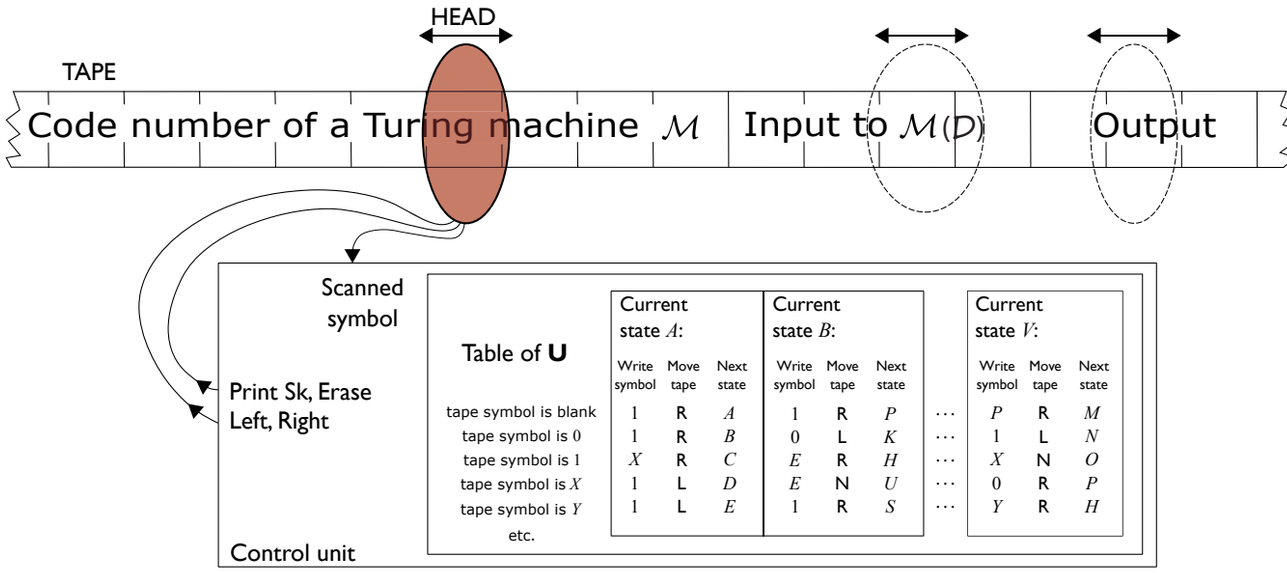


Figure 4.5.1.10 Universal Turing Machine (UTM) showing its control unit which has to process the instructions for Turing machine M placed on the same tape as the input data

The control unit in *Figure 4.5.1.10* defines a Turing machine U that can execute other Turing machines by simulating the behaviour of any Turing machine. This Turing machine U is called a **Universal Turing Machine (UTM)**.

In a universal Turing machine that uses a single one-dimensional tape, the instructions of the Turing machine M being simulated are placed on the tape and followed by the data D to be processed by M.

The read-write head must start on M and then move between M and D as M is executed. Naturally, U may be a lot slower than the Turing machine M that it simulates.

A consequence of discovering that UTMs can exist is that **programs and data are really the same thing**.

A program is just a sequence of symbols that looks like any other piece of input, but when fed to a UTM, this input wakes up and begins to compute.

The Turing machines that we have considered earlier in this chapter are single purpose machines.

The Universal Turing Machine is very different.

It is a **general purpose machine** or **general purpose computer**.

A change of purpose is effected by simply changing the Turing machine description M placed on the tape. This was the insight that Alan Turing documented in his 1936 paper "*On Computable Numbers, with an Application to the ENTSCHEIDUNGSPROBLEM*". This was the outstanding achievement that anticipated the advent of the modern digital general purpose computer.

Information

A single desktop computer can run a word processor, a spreadsheet, as well as new applications that have yet to be devised. This may seem completely natural, but most technology does not work this way at all. In most cases the device is the application and they are the same thing. If you own a radio and want to watch TV, you must buy a new device. You cannot download a new set of instructions to turn your radio into a TV. Desktop PCs are based on the computing principle of universality.

Key point

Universal Turing machine (UTM):

A UTM, U, is an interpreter that reads the description M of any arbitrary Turing machine M and faithfully executes operations on data D precisely as M does. For single-tape Turing machines, imagine that M is written at the beginning of the tape, followed by D.

Single licence - Abingdon School

Key term

Interpreter:
An interpreter works its way through a set of instructions identifying the next instruction then executing it.

Key point

Programs and data can be treated as the same thing.

Key term

Principle of universality:
A universal machine is a machine capable of simulating any other machine.

Key point

Anything a real computer can compute, a Turing machine can also compute. Therefore a statement about the limitations of Turing machines will also apply to real computers. A Turing machine can simulate any type of subroutine found in programming languages.
A subroutine is a subprogram that can be used as a component in a larger program.

Key point

Don't confuse 'not computable' with 'we don't know how to compute it' or 'we will never know how to compute it'. It is possible that a method for computing a solution may be found at some time in the future for a problem that has a computable solution. It will never be found for a problem whose solution is theoretically proven to be noncomputable.

Programs as data

Examples of program as data abound.
For example, a computer downloads Java applets and viruses as data but can then proceed to execute these as programs.
Nature also treats programs as data. DNA contains programs and data. Its programs carry out the task of replicating DNA by treating the DNA as data.

The type of translator software known as an interpreter which takes a description of a program, e.g. a Python script and executes it, behaves like a Universal Turing Machine.
In fact, it is possible to reduce all other types of computing machine to an equivalent Turing machine.

Power of a Turing machine

Another even more important consequence is that the world of computation is powered by UTMs.

Turing devised a machine whose basic operations cannot be divided any further.
A consequence of this is that all other types of computing machines can be reduced to an equivalent Turing machine.
A conclusion that can be drawn from this is that no computing device that can be realised by controlling some physical process of the universe can be more powerful than a Turing machine.
Effectively, this means that a Turing machine can do anything that a computer can do.
Very importantly, a Turing machine provides a formal definition of the digital computer.

Any program written to run on an Intel® Core™ processor can be broken down into a sequence of the processor's most primitive machine instructions.
In a similar manner, a sequence of Turing machines may be connected together to perform more complex operations matching those of any program that could run on the Intel Core processors.
Very importantly, processors come and go, instruction sets and clock speeds change. If we are to reason about computability, then basing such reasoning on Intel Core processors would require the reasoning to be constantly re-evaluated.
As Turing machines are independent of any real processor, it is possible to reason about computations with Turing machines and be confident that such reasoning will not need to be reevaluated.

Single licence - Abingdon School

Questions

5. What is a Turing machine?
6. What is the principle of universality?
7. Explain what is meant by a Universal Turing machine (UTM).
8. Explain how a UTM can be considered to be an interpreter.
9. Explain why programs can be treated as data.

In this chapter you have covered:

- The structure and use of Turing machines that perform simple computations
- 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
  - one of the states is called a start state and states that have no outgoing transition are called halting states
- The equivalence between a transition function and a state transition diagram
- Representing
  - transition rules using a transition function
  - transition rules using a state transition diagram
- Hand-tracing simple Turing machines
- The importance of Turing machines and the Universal Turing machine to the subject of computation

Single licence - Abingdon School

Index

Symbols

^ 54
::= 428
! 54
!= 37, 52
{} 408
\ 414
& 54
&& 54
# 186
→ 428
↦ 101
⇒ 359
∅ 417
∈ 414
< 52
<= 52
<> 52
= 52
== 52
> 52
>= 52
| 54, 408
|| 54
~ 414
– 414
⊂ 414, 416
⊆ 414
∧ 408
∩ 414
⌈ ⌉ 333
⌊ ⌋ 332
[0-9] 429
2-ary 280
3-ary 280
32-bit IEEE 754 floating point 17
64-bit IEEE 754 floating point 17
(a|b) regular expression 418
(action, next state) 406
<digit> 429
<expression> 431
(input symbol, current state) 406
\n 217
<number> 429
<string> 428
ε 416, 428
π 412
Σ 416
Σ* 416

A

abstract class 175, 176, 177
abstract data type 227, 245, 250
abstraction 140, 166, 223, 373, 382, 386, 390, 395, 396, 400
abstraction and subprograms 375
abstraction as used in computations 374
abstract method 174, 175, 176, 177
accepting state 401, 402
access modifiers or specifiers 180, 186
access/visibility level specifier 186
actual parameter 25, 28, 102, 131
adding an item to a circular queue 256
adding an item to an empty linear queue 253
adding an item to a non-empty linear queue 253
addition 44
addition operation 436
addition operator 234
adding two vectors 246
adjacency list 267
adjacency matrix 265, 348
Adobe Postscript 320
ADT 227, 245
Advantages of the structured approach 137
Advantages of using subroutines 100
A finite state machine with outputs
and more than one trigger 406
aggregation 155, 157, 182, 185, 187
aggregation class diagram 187
algebraic dot product 301

Single licence - Abingdon School

algorithm 364, 400, 435, 469, 479
algorithm for finding factors of an integer N 441
algorithm for finding the least factor of 441
algorithmic problem 462
alphabet 416
alternation 431
ancestor class 146
ancestors 273
AND 54
ANSI 59
AnsiChar 8
ANSI standard 59
appending to a text file 220
applications of Dikjstra's shortest path algorithm 356
arcs 235, 262
argument 29
argument form 360
arithmetic expression 44, 234, 382
Arithmetic operations 44
arithmetic operators 44
arithmetic overflow 93
array 13, 214, 231, 289, 374, 399
array cell 13
array data structure 13, 191
array data type 13
array index 14
array of integer 13
articulating how a program works 369
ASCII 6, 59, 217
assignment 18, 365
assignment operator 18
assignment statement 18
association 185, 187, 188
association class diagram 188
associations 159
associativity 234
asymptotic behaviour 438, 454
asymptotic behaviour of the algorithm 437
attribute fields 180
attributes 141, 145, 186
attributes of an object 165
attribute values 140
automation 386, 390, 395, 396, 400
automaton 422
average-case 330
average search length 323
AVL Binary Search Tree 334

B

back button on a browser 232
Backus-Naur Form 427, 429
balanced tree 258, 333
base 2 log calculator 332
base case 116, 341, 117, 123
base class 146, 147, 165
base of a logarithm 449
base of current stack frame 233
base of stack pointer 259
basic operation 436
behaviour of an object 145
best-case 330
Big-O 330
Big-O notation 453
binary files 223
binary search 453
binary search 324, 454
binary search algorithm 325
binary search tree 276, 328
binary search tree algorithm 329
binary search tree time complexity 333
binary tree 274, 312
binding 112, 149
binding time 112
BIOS 374
bitmap 224
black box 380
BNF 427
body of the subroutine 25
bool keyword 8
Boolean 8
Boolean data type 6
Boolean expressions 52, 54
Boolean operands 54
Boolean operations 54
Boolean operators 54
bottom-up development 396

Single licence - Abingdon School

bounded time 455
branch 273
breadth-first search 304, 310
brute-force approach 435
bubble sort 336, 440
buffering 388
built-in data types 7, 14

C

C# 30, 72, 74, 76, 77, 78
call by address 101
call by reference 101
call by Reference/Call by Address 102
call by value 101, 102
calling a procedure 26
call a subroutine 25
cardinality of a set 411
cardinal number 411
Cartesian product of sets 413
case selector 23
Case/Switch statement 23
ceil keyword 50
cell 227, 253
cell graphs 268
CFG 428
char keyword 8
character 6
character 59
character → character code 70
character code → character 71
character data type 6
characteristics 145
child 273
Chomsky's classification 431
Chr 98
Church–Turing thesis 479
circular buffer 388
circular queue 255
class 139, 140, 142
class abstraction 140
class definitions 160
class diagram 185
classification of algorithms 435, 453, 460, 465, 469
classifying how fast functions grow 454
class of computations 382
class type 177
closed addressing 295
closed hashing 290, 296
closed subroutines 396
code page identifiers 59
code pages 59
Code reusability 160
codomain 445
cohesion 125
cohesive 124
collision 290
combining data objects to form compound data 399
combining principles 23
combining procedures 396
comments 42
Common Era 9
Common Intermediate Language(CIL) 320
common pattern 139
compact representation of a set 409
comparing algorithms 435
compiler 112
complete tree 280
complexity classes 439
complexity class of an algorithm 439
complexity exponential 461
complexity of a problem 441
component of the system 124
components 124, 160, 396
composite data type 106, 191, 399
composition 106, 156, 163, 185, 396
composition abstraction 396
composition class diagram 187
compound data 399
compound data object 386
compound procedure 396
compound proposition 357
computability, 482
computable 462, 479
computable number 412
computable numbers are enumerable 413
computable problem 465

Single licence - Abingdon School

computation 382, 412, 471, 479
computation method 382
computation time 453
computer program 412
concatenation 85, 431
conclusion 359
condition 34
conditional connective 359
conditional statement 359
connectedness 271
connected undirected graph with no cycles 239
connectivity 269
consistency 161
console application 67
Console.WriteLine 67
Const keyword 18
constant 57
constant 17
constant declaration 17
constant time 455
constructing and hiding the queue compound object 387
constructor 141, 151, 170, 171, 172, 173, 174, 177
contents of registers 120
context-free grammar 428, 431
context-free languages 427
control characters 217
control hierarchy chart 130
control unit 473
convex combination of two vectors, u and v 301
correctness 369
countable collections 409
countably infinite sets 409
coupling 126, 164
Create keyword 171
cubic function 449
current stack frame pointer 233
current state 477

D

data abstraction 386, 388, 399
data encapsulation 144, 381
data hiding 144
data modelling 374
data representation 391
data structure 191, 216, 227, 250, 375, 400
data type 1, 2, 3
date 9
datetime 9
Date/time 9
DateTime 9, 79, 80
datetime.date 9
datetime.time 9
Date/time to string 78
datum 2
debugging 111
descendent classes 147
decision problem 402, 466
decision tree 279
declaring local variables 109
decomposition 377, 395
deduction 358
deductive reasoning 362
def keyword 28
definite iteration 33
delete(key) 242
deleting a binary tree recursively 316
deleting a record in a table 286
Delphi 72, 76
depth-first search 306, 310
depth of balanced tree 333
dereferenced 251
derived classes 146, 147
descendants 273
descendent class type 177
description number 412, 480
design 124
design time 250
Destroy keyword 171
destructors 177
deterministic 90
deterministic Finite-state Automaton 427
deterministic finite-state automaton (DFA) 422
DFA 427
Dictionaries 299
dictionary 242, 248
dictionary abstract data type 242

Single licence - Abingdon School

difference - set difference 414
Difference between a class and an interface 163
digit+ regular expression 429
digital stored program computer 396
digraph 263
Dijkstra's shortest path algorithm 348
directed edge 263
directed graph 263, 267, 348, 475
directory structure 239
disciplined approach 132
divide and conquer 124
divide and conquer algorithm 333, 341
division 44
division by zero 93
domain 445
dot or scalar product of two vectors 246, 301
double keyword 8, 18
double circle notation 423
double precision 5
do while loop 39
dyadic operator 318
dynamic array 227, 250, 258
dynamic data structure 250

E

edge 235, 262, 271, 348, 475
EDSAC 396
efficiency 370
element 245
elementAtRank(k) 245
element data type 214
empiricism 362
empty branch 275
empty list 258
empty set 408
empty string 416
encapsulate 162
encapsulation 140, 143, 397
end-of-file on input 93
English description of algorithm 366
English-French dictionary example 288
entry point 132
enumerate 413
enumeration 412
equal precedence 234
Equal To 52
evaluating an expression 234
exception 93
exception handling 93
execution time 436
exit point 132
exponential 440
exponential function 448
exponential time 455
exponentiation 48
expressions 52
expression tree 282, 318
extend keyword 145
Extended-ASCII 6, 10

F

Facebook graph 237
factorial 450
factorial function 451
False 54
field 11, 216
FIFO 228
file 216
file data structure 192
file handle 219
file mode 220
filename 216
files 216
file type declaration 223
FILO 230
finding angle between two vectors 302
find(key) 242
finite set 409
finite state automaton 432
finite state machine 401
finite state machine with no output 402, 405
finite state machine with output 405
First In First Out (FIFO) abstract data type 228
First In Last Out 230
float 8, 17
floating point data type 4

Single licence - Abingdon School

floating point quotient 44
FloatToStr 77
float to string 77
floor operation 50, 371
flow of control 132
following algorithms 364
For loop 34, 367
formal parameter 28, 101, 131
formal parameter of the subroutine 25
formula 382
front 228
front pointer 254
front pointer variable 228
FSharp 118
FSM 401
FSM (DFA) to regular expression 426
function 26, 28, 29
functional abstraction 384
functional cohesion 125
function argument 384
function body 384
function mapping 445
function parameter 384
fundamental operation 436

G

game tree 283
garbage collection 172
garbage collector 172
general case in recursion 117, 123
generalisation 375, 390
general purpose computer 481
general purpose machine 481
geometric dot product 302
geometric point 246
getter method 165
global scope 115
global scope 113
global variables 112
glyphs 60
goal state 401
good design 124
Go To 132
grammar 431
grandchildren 239
graph 235, 236, 262, 271, 348
graph abstract data type 235
Graphical User Interface 160
graph-traversal 304
graph visualisation 235
Greater Than 52
Greater Than Or Equal To 52
Gregorian Calendar 9
growth rate 438, 451
GUID 165

H

half precision 5
Halting Problem 467, 469, 471
Halting Problem decider 469
Halting state 405, 474, 475, 476
hand-trace 19, 211
hand-tracing 477
hand-tracing algorithms 368
“has-a” test 157
hash 287, 288
hash function 241, 287
hashing 241, 287
hash key 287

hash table 241, 285, 286
head 228
heap 258
height of a rooted tree 280
heuristic methods 464
hierarchy chart 129, 380
high watermark pointer 259
Hollywood graph 268
hotel infinity 410
hypothesis 359

I

identifier 41, 432
If statement 22
immutable 67
implementation 387
implementation encapsulation 144

Single licence - Abingdon School

implementation hiding 143
implementation-independent view 227
implementation section 143, 387
impure functions 29
IN keyword 104, 131
indefinite iteration 33
indefinite iteration 36
index 66, 84, 214
index 84
index calculator 241
induction 362
inductive reasoning 362
infinite decimal 480
infinite loop 36, 469
infinitely long tape 473
infinite sequence of digits 412
infinite set 416, 409
infinite sets which aren't countable 411
infix 234
infix expression 318
infix to Reverse Polish 318
information 7
informational abstractions 400
information hiding 143, 144, 377, 378
information retrieval 299
inherit 160
inheritance 145
initialiser 34
initialising variables 17
in-order: binary search tree 315
in-order traversal 312
IN-OUT keyword 131
INOUT keyword 104, 105
input 364
input conversion 93
input set 445
insertAtRank(k, element) 245
inserting a new record into a table 285
insert(key, value) 242
instantiation 141
int 7, 8, 30
integer 8, 32
integer data type 4
integer division 45, 287
integer division and integer remainder operators 46
integer division operator 45
integer quotient 44
integer remainder operators 46
Integer to string 76
interface 104, 143, 165, 166, 378, 379, 380, 387
interface information 398
interface operations 168
interface section 143, 180, 387
interface type 162
interface unit 162
internal design of the components 134
internal node 273
internal state 160
Interpreter 482
interpreters and compilers 320
intersection - set intersection 414
intractable problems 462, 463
invoking a method 140
Iterating through the elements
of a two-dimensional array 211
iteration 19, 20, 365
iterator 34

J

Java 31, 74, 76, 77, 78
Java Swing 397
Java Virtual Machine (JVM) 151, 320

K

keyboard buffer 229, 388
key, value pair 242
keyword 166, 177, 180, 432

L

label 132
Language keyword 17, 28
languages which are not regular 428
Last In First Out 230
late binding 149, 177
layers of abstraction 374
lazy I/O 229, 388
leaf 273

Single licence - Abingdon School

length function 67
length of a string 67
length of a vector 302
Less Than 52
Less Than Or Equal To 52
lexical analyser 431
lexical analysis phase 432
libraries 397
libraries of (compound) generally useful procedures 397
lifetime 109
LIFO 230
limits of computation 460, 471
linear and binary search algorithms compared 327
linear congruential method 89
linear function 447
linear list 227
linearly 439
linear plot 447
linear queue 227, 228, 253
linear rehash 290
linear search 322, 453
linear search algorithm 323
linear time complexity 455
line feed character 217
linked list 227
linker 398
list 248, 250, 258
literal constants 432
LocalDate 9
LocalDateTime 9
LocalTime 9
local variable 109, 233
local variables in subroutines 109
logarithmic function 449
logarithmic plot 449
Logarithmic time 455
logarithms 331
logical reasoning 361
logical view 374
logic problem 357
looking up a record in a hash table 288
loop 20
loop body 20
loop control variable 34
loop terminating condition 33
loose coupling 164
loosely coupled 124, 126
lower bound 330

M

magic methods 169
Main 27
map operation 397
map colouring problem 392
mathematical concept of a function 445
maximum positive value 5
maze 237, 310
Mealy machine 405
meaningful identifier names 41
meaning of value 2
member - set membership 414
memory 435
memory map 101
merge sort 341
merge sort algorithm 342
message 144, 160
metacharacter 419
metasymbols 419
method declaration 177
methods 160, 180, 186
methods of the object 140
minimum positive value 5
Mod keyword 255, 287
model 374
modelling 374
modelling data by variables 17
models 400
models of computation 412, 479
modular approach 160
modularisation 110
modularity 143
modular software 396
module 27, 124, 125, 126, 160, 377
module interface 378
modulo 288
modulo arithmetic operator 287

Single licence - Abingdon School

multi-dimensional array 214
multiplication 44
mutual friendship graph 262

N

named constants 57
N-dimensional arrays 213
nested brackets 432
nested constructs 432
nested iteration statements 40
nested procedures 397
nested selection statements 39
new keyword 172, 173, 174
newline 217
node 235, 239, 260, 273
non-accepting state 402
non-computable problems 465
non-deterministic finite state machine with outputs 405
non-recursive solution 121
nonterminal symbol 428
non-text files 217
non-volatile 216
NOT 54
Not Equal To 52
null pointer 253
Numpy 14

O

Ø 408
O(1) 455
O(2^n) 455
O(a^n) 455
object 139, 160, 432
object instantiation 141
object messaging 144
object-oriented design principles 162
object-oriented paradigm 160
object-oriented programming 139, 160

object property 165
object type 151
OEM 61
offset 65, 84
O(log n) 332, 455
O(n) 346, 454, 455
O(n^2) 455, 456
O(n^3) 455
On Computable Number 481
one-based string indexing 66
one-dimensional array 245, 248
O(n^k) 455
O(n $\log_2$ n) 347
OOP 160
open addressing 290
open hashing 296
operand 54
operator precedence 55, 234
optimisation algorithms 348
OR 54
Ord 98, 288
ordered binary tree 276
ordered pair 263, 413
order of a function 437
order of complexity 453
order of growth 330
order of magnitude 437
order of magnitude of a function 437
ordinal data type 34
ordinal number 66, 411
OUT keyword 105, 131
out of line block of code 99
out of scope 115
output 364
output set 445
overridable 151
override 148, 149, 177

P

palindrome checker 477
parameters 233
parameters of subroutines 101
parent 273
parent-child relationship 239
parent class type 177
parity 475
parser 431, 432
parse tree 428, 429, 432

Single licence - Abingdon School

Pascal 72, 75
Pascal/Delphi 76, 77, 80
path 269
pattern matching 421
patterns 429
Peek or top operation 260
perfect hash function 241, 292
perfect number 470
permutation 450
permutation of n distinct objects 450, 451
physical states 7
physical view 227, 374
Pigeonhole principle 390, 410
pixel 224
pixel intensity 214
point 246
pointer 146, 295, 250
Polish notation 318
polymorphically 175
polymorphism 148, 149, 175
polynomial 455
polynomial coefficients 448
polynomial function 448
polynomial time 455
polynomial time solution 462
Pop operation 260
popping 259
Position 87
postfix 234
post-order 318
post-order: emptying a tree 316
post-order: infix to RPN (Reverse Polish Notation) 315
post-order traversal 312, 313
power of a Turing machine 482
power set 414
pre-order: copying a tree 315
pre-order: prefix expression from expression tree 315
pre-order traversal 312
prime number 293
primitive 229
primitive data type 191
primitive machine instructions 482
Principle of universality 482
principles of structured programming 132
priority queue 257
private 17, 186
Private keyword 144, 180
PRNG 88
problem 366
problem abstraction 390
problem reduction 390
problem solving 357, 400
procedural abstraction 382, 384
procedural decomposition 395
procedure 26, 27, 396
procedure interface 28
procedure name 26, 28
procedures and functions 26
process of abstraction 374
production 428, 429
production rule 428
program 27
program-coded algorithms 400
programming language description 366
programs and data 481
programs as data 482
proof by contradiction 471
properties 160
proposition 357
propositional variables 359
protected keyword 181, 186
pseudo-code 19, 118, 253, 259, 366, 367, 471, 469
pseudo-code description 366
pseudorandom number generator 88
pseudorandom numbers 88
public keyword 17, 113, 144, 180 , 186
public, private and protected access level specifiers 190
public, private and protected specifiers 180
pushing 259
Push operation 259
Python 72, 75, 76, 77, 81

Q

quadratic 440
quadratically 438
quadratic function 448

Single licence - Abingdon School

quadratic order of growth 456
quadratic plot 446
quadratic polynomial 448
queue 228, 253, 381
queue data structure 386

R

r? - regular expression 418
r* - regular expression 4 17
r+ - regular expression 418
RAM 216
randomising algorithm 241
randomising function 287
randomizing the seed 90
random number generation 88
random numbers 88
random.seed 90
random sequence 88
range 445
rank 245
rank of an element 245
rationalism 362
reading from a text file 217
read-write head 473, 477
real 8
real/float 5
real/float and integer division 45
real/float data type 4
real number/float division 44
rear 228
rear pointer 253
rear pointer variable 228
record 10, 11, 13, 216, 289
record data structure 191
record data type 11
recurrence relation 121
recursion 116, 431
recursive algorithms 121
recursive definition 117 , 312
recursively-defined function/procedure 117
recursive object 117
recursive techniques 116
reference variable 149, 169
reflect a single vector 246
regular expression 416, 17, 427
regular expressions and FSMs 422
regular expression to FSM 425
regular language 401, 416, 423, 431, 432
regular set 432
relational operators 52
reliability 161
remainder 46, 287
removeAtRank(k) 245
removing an item from a non-empty circular queue 256
removing an item from a non-empty linear queue 254
repeat until 33
repetition 20, 431
replaceAtRank(k, element) 245
representation abstraction 391
representational abstraction 373
resources consumed 436
return address 120, 233
return expression 106
returning a result 105
returning a value(s) from a subroutine 105
return type 30, 32
reusability 160
Reverse Polish 318
Reverse Polish notation 318
role of stack in recursion 120
root 272
rooted multi-way tree 273
rooted tree 239, 272, 273
root node 273
root node pointer 240
rotate a single vector 246
rounding 49
rounding down 50
rounding off 49
rounding up 50
route finding 235
RPN to infix 319
r|s - regular expression 417
running total 21
runtime 250, 435

Single licence - Abingdon School

S

scale a single vector 246
scanner 432
scope 112
scope of global and local variables 112
Searching a table for a record 285
Searching for a specific record in a hash table accommodating collisions 292
search interval 324
search length 323
secondary storage 216
seed 89
Select Case statement 24
selection 22, 365
self keyword 169, 171
self-contained 110
self-contained block of instructions 25
self-describing identifiers 42
self reference parameter/variable 170
semantic dependency 159
sentence 432
sequence 2, 22, 365, 416
sequence statements 22
sequential collection of elements 13
set 3, 408, 416
set comprehension 408
set operations 414
setter method 165
setting up a hash table that uses closed hashing 292
siblings 273
side-effect 102, 110
significant digits 5
simple calculator 129
simple hashing functions 288
single inheritance 185
single inheritance class diagram 186
single precision 5
size of problem 437
social network 262
software architecture 129
software artefact 124
software module 396
software reuse 396
solving problem algorithmically 466
space complexity 441 , 453
space complexity of an algorithm 442
stack 120, 140, 230, 232, 259
stack base 230, 259
stack data type 231
stack frame 120, 233
stack overflow 120
stack pointer register 233
start state 423, 474, 476
start symbol 429
state-changing operations 168
state of an object 145
states 401
state table 404
state transition diagram 401, 422, 474, 475, 477
state transition table 404
static 17, 30
static array 227
static class 178
static data structure 250
static memory allocation 112
static method 178
static scoping 112
static variables 112
Steganography 224
stepwise refinement 135, 395
stepwise refinement 134
stop state 474
str 8
string 8, 61, 67
string conversion operations 72
string data type 7
string-handling operations 59
string indexing 63
string manipulation 421
string operations 63
String to date/time 82
String to float 74
String to integer 72
StrToFloat 75, 76
struct 13
structure 11, 13, 216

Single licence - Abingdon School

structured data type 106
structured design 124
structured programming 124, 132, 160, 164, 395
structured programming computations 168
structured result 29
structure of a program 432
subclass 147,160, 176
subject 432
subrange error 93
subroutine 25, 26, 98, 105, 124
subroutine call 396
subroutine calls and stack frames 233
subroutine interface 104, 380
subscript 93
subset 84, 414, 416, 432
substring 84
subtraction 44
sub-tree 274, 275, 328
super 151
superset 414
Switch Case 24
symbol 475
symbol class 418
symbolic name 17
symbolic name for constant 17
syntax 431
syntax diagram 427, 432, 433
syntax expression 428
syntax of a programming language 432
syntax tree 277

T

table in computer science 285
tail 228
TDateTime 9, 80
terminal symbols 428
terminate 480
terminating condition 20
test for empty stack 260
test for full stack 260
testing for a full circular queue 256
testing for a full linear queue 253
testing for an empty circular queue 256
testing for an empty linear queue 253
text 216
text file 216, 218
this keyword 172
time 9, 435, 436
time complexity class 455
time complexity of an algorithm 442, 456
time complexity of binary search 332
time complexity of bubble sort algorithm 340
time complexity of linear search 331
time complexity of merge sort algorithm 346
time-efficient 371
tokens of a programming language 432
top 230
top-down design 395
top of stack 232, 259
top of stack pointer 259
tournament tree 283
trace table 211, 344
tractable 463
tractable problems 462, 463
trade-off between time-complexity
and space-complexity 441
transition 475
transition function 405, 474, 475, 477
transition rules 474, 475
transition's action 475
transition's trigger 405, 475
tree 239, 271, 272
tree-traversal 312
True 54
truncation 50
tuple 214, 413
Turing machine 412, 473, 474
Turing machine program 479
two-dimensional array 134, 213

U

unbalanced binary search tree 334
undecidable decision problem 468
undecidable problem 469
undesirable side-effects 110
undirected graph 271

Single licence - Abingdon School

undirected graph 263, 267
Unicode 8, 60
Unicode code point 60
union - set union 414
unique path 271
unit of modularity 164
units 124
Universal Turing machine 479, 480
upper and lower bounds 330
upper bound 330
user-defined data types 14
uses of dictionaries 244
uses of queues 229
uses of stacks 232
using local variables 109
using parameters to pass data within programs 101
using subroutines with interfaces 104
UTF-8 6, 60, 217
UTF-16 6, 60
UTF-16 and .NET 62
UTF-32 6, 60
UTM 481

V

valid argument 359
valid string 416
variable 16, 57, 382
variable declaration 16
VB.NET 32, 72, 74, 76, 77, 80
vector 245, 247, 246, 300
vector - 3-vector over R 301
vector abstract data type 248
vector - algebraic sense 300
vector as a function 301
vector as a list 301
vector as a one-dimensional array 301
vector - dictionary of key:value pairs 301
vector - geometric sense 300
vectors and geometry 246
verb 432
vertex 235
vertices 262, 271, 348
virtual 149, 177
virtual directive 177
virtual method 177
virtual stack machine 319
visibility 30, 32, 143, 109
visualising a vector as an arrow 301
visual programming language 160
Visual Studio 2015 30
void 30
volatile 216
volatile environment 233

W

Web crawling 310
weighted graph 263
weight of an edge 348
well-defined algorithm 480
while 33
whitespace 217
whole/part association 185
Wilkes 396
Wirth's law 462
Wolfram programming language 235, 244
worst-case 330
worst-case complexity 331, 439
worst-case time complexity 332, 440
Worst case time complexity of merge sort 347
wraparound 255
write mode 219, 223
writing algorithms 364, 366
writing to a text file 219

X

XOR 54

Z

ZEROBASEDSTRINGS 66

Single licence - Abingdon School

Single licence - Abingdon School

A Level Computer Science for AQA Unit 1

This digital textbook covers Unit 1 of AQA A Level Computer Science in an accessible and student-friendly way.

Additional resources to accompany the text will be available from our web site **www.educational-computing.co.uk.** Please note that these resources have not been entered into the AQA approval process. Only the student textbook has been approved.

This book has been approved by AQA

Cover picture © Dr K R Bond
Pulau Redang, Kuala Terrangganu, West Malaysia

About the author

Kevin Bond has many years of teaching and examining A Level Computing/Computer Science experience.
He has worked at the interface between Science, Computer Science and Engineering, first as a Research Scientist, then as a Senior Development Engineer for a major Defence Contractor and then as a Senior Systems Analysis for a major Telecommunications company.

Educational Computing Services Ltd
42 Mellstock Road
Aylesbury
Bucks
HP21 7NU
Tel: 01296 433004
www.educational-computing.co.uk

ISBN 978-0-9927536-5-8

Single licence - Abingdon School