UNIVERSITY OF LONDON | INTERNATIONAL PROGRAMMES

Software engineering, algorithm design and analysis Volume 2

I. Pu

CO2226

2006

Undergraduate study in
Computing and related programmes

This is an extract from a subject guide for an undergraduate course offered as part of the University of London International Programmes in Computing. Materials for these programmes are developed by academics at Goldsmiths.
For more information, see: www.londoninternational.ac.uk

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London International Programmes by:

I. Pu

University of London International Programmes
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

www.londoninternational.ac.uk

Published by: University of London

Contents

Preface **v**

1 Algorithm analysis **1**
1.1 Essential reading 1
1.2 Learning outcomes 1
1.3 Problems and algorithms 1
1.3.1 Implementation 2
1.4 Pseudo-code for algorithm description 2
1.5 Efficiency 3
1.6 Measures of performance 3
1.7 Algorithm analysis 4
1.8 Model of computation 5
1.8.1 Counting steps 6
1.8.2 Implementation 6
1.8.3 Characteristic operations 7
1.9 Asymptotic behaviour 8
1.9.1 Big O notations 8
1.9.2 Comparing orders of two functions 9
1.10 The worst and average cases 9
1.10.1 Implementation 10
1.10.2 Typical growth rates 12
1.11 Verification of an analysis 12

2 Abstract data types I: lists and hashing tables **15**
2.1 Essential reading 15
2.2 Learning outcomes 15
2.3 From Abstraction to Implementation 16
2.3.1 The string abstract data type 17
2.3.2 The matrix abstract data type 18
2.3.3 The keyed list abstract data type 18
2.3.4 Common operations 19
2.4 Data structures and software performance 19
2.5 Motivation of abstract data types 19
2.6 Arrays 20
2.6.1 Applications 21
2.6.2 Array of objects 23
2.6.3 Two and multi-dimensional arrays 24
2.7 Lists 24
2.7.1 References, links or pointers 25
2.7.2 Implementation of the links 26
2.7.3 Linked lists 27
2.7.4 Operations on lists 28
2.7.5 Add one node to a linked list 29
2.7.6 Delete one node from a linked list 30
2.7.7 Implementation 31
2.7.8 Construct a list 31
2.7.9 Implementation 32
2.7.10 Other operations 34
2.7.11 Comparison with arrays 34
2.8 Stacks 36
2.8.1 Operations on stacks 37

2.8.2 Implementation . . . 37
2.8.3 Applications . . . 39
2.9 Queues . . . 40
2.9.1 Operations on queues . . . 40
2.9.2 Implementation of queues . . . 41
2.9.3 Variation of queues . . . 43
2.10 Hashing . . . 44
2.10.1 Collision . . . 46
2.10.2 Collision resolving . . . 46
2.10.3 Extra work for retrieval process . . . 50
2.10.4 Observation . . . 50

3 Algorithm design techniques 53
3.1 Essential reading . . . 53
3.2 Learning outcomes . . . 53
3.3 Recursion . . . 53
3.3.1 Implementation . . . 56
3.3.2 What happens . . . 57
3.3.3 Why recursion? . . . 58
3.3.4 Tail recursion . . . 63
3.3.5 Principles of recursive problem solving . . . 63
3.3.6 Common errors . . . 63
3.4 Divide and conquer . . . 65
3.4.1 Steps in the divide and conquer approach . . . 66
3.4.2 When Divide and Conquer inefficient . . . 69
3.5 Dynamic programming . . . 70
3.5.1 Overlapped subproblems . . . 70
3.5.2 Dynamic programming approach . . . 71
3.5.3 Efficiency of dynamic programming . . . 72
3.5.4 Similarity to the Divide and Conquer approach 72
3.5.5 Observation . . . 73

4 Abstract data types II: trees, graphs and heaps 75
4.1 Essential reading . . . 75
4.2 Learning outcomes . . . 75
4.3 Trees . . . 75
4.3.1 Terms and concepts . . . 76
4.3.2 Implementation of a binary tree . . . 77
4.3.3 Recursive definition of Trees . . . 79
4.3.4 Basic operations on binary trees . . . 80
4.3.5 Traversal of a binary tree . . . 80
4.3.6 Construction of an expression tree . . . 81
4.4 Priority queues and heaps . . . 84
4.4.1 Binary heaps . . . 84
4.4.2 Basic heap operations . . . 86
4.5 Graphs . . . 92

5 Traversal and searching 101
5.1 Essential reading . . . 101
5.2 Learning outcomes . . . 101
5.3 Traversal . . . 101
5.3.1 Traversal on a linear data structure . . . 102
5.3.2 Binary tree traversal . . . 102
5.3.3 Graph traversal . . . 102
5.3.4 Depth-first traversal . . . 102
5.3.5 Breadth-first traversal . . . 102
5.4 Searching . . . 104
5.5 Sequential search . . . 105
5.6 Binary search . . . 106

5.7 Binary search trees . . . 109

6 Sorting 113
6.1 Essential reading . . . 113
6.2 Learning outcomes . . . 113
6.3 Introduction . . . 113
6.4 Motivation . . . 113
6.5 Insertion Sort . . . 114
6.5.1 Algorithm analysis . . . 115
6.6 Selection sort . . . 115
6.7 Shellsort . . . 117
6.8 Mergesort . . . 118
6.9 Quicksort . . . 121
6.10 General lower bounds for sorting . . . 124
6.11 Bucket sort . . . 126
6.12 Sorting large records . . . 127
6.13 Heapsort . . . 128

7 Optimisation problems 137
7.1 Essential reading . . . 137
7.2 Learning outcomes . . . 137
7.3 Optimisation problems . . . 137
7.3.1 Multiplication of matrices . . . 138
7.3.2 Knapsack problem . . . 139
7.3.3 Coin changes . . . 141
7.4 Greedy approach . . . 145
7.4.1 Huffman coding . . . 145
7.4.2 Huffman decompression algorithm . . . 148

8 Limits of computing 151
8.1 Essential reading . . . 151
8.2 Learning outcomes . . . 151
8.3 Computability and computational complexity theory . . . 151
8.4 Computational model . . . 152
8.5 Decision problems . . . 153
8.6 Measure of problem complexity . . . 154
8.7 Problem classes . . . 155
8.8 Class $\mathcal{P}$ . . . 156
8.9 Class $\mathcal{NP}$ . . . 156
8.10 $\mathcal{P}$ and $\mathcal{NP}$ . . . 157
8.11 NP-complete problems . . . 158
8.11.1 What do we mean by *hard*? . . . 158
8.11.2 How to prove a new problem is NP-complete . . . 159
8.11.3 The first NP-complete problem . . . 159
8.11.4 More NP-complete problems . . . 160

9 Text string matching 161
9.1 Essential reading . . . 161
9.2 Learning outcomes . . . 161
9.3 String matching . . . 161
9.4 The string ADT . . . 162
9.5 String matching . . . 164
9.5.1 Naive string matching . . . 164
9.5.2 Observation . . . 166
9.5.3 Boyer-Moore Algorithm . . . 166
9.5.4 Observation . . . 168
9.6 KMP Algorithm . . . 169
9.7 Tries . . . 174
9.7.1 Standard tries . . . 174

9.7.2 Compressed tries 175

10 Graphics and geometry **177**
10.1 Essential reading 177
10.2 Learning outcomes 177
10.3 Quadtrees 177
10.4 Octtrees 178
10.5 Grid files 178
10.6 Operations 178
10.7 Simple geometric objects 179
10.8 Parameter spaces 179

11 Revision for CIS226b examination **181**
11.1 Examination 181
11.2 Revision materials 181
11.3 Questions in the examination 181
11.4 Read questions carefully 182
11.5 Recommendation 182
11.6 Revision topics I 183
11.7 Revision topics II 183
11.8 Good luck! 183

A Sample examination paper **185**

B Sample solutions **189**

C Pseudocode notation **195**
C.1 Values 195
C.2 Types 195
C.3 Operations 195
C.4 Priority 195
C.5 Data structures 196
C.6 Other reserved words 196
C.7 Control keywords 196
C.8 Examples of sequential structures 196

Preface

Introduction

Abstract data types or data structures provide powerful methods of organising large amounts of data. Algorithm analysis enables you to make a decision about the most suitable algorithm before programming. Techniques using abstract data types and design patterns are essential in conventional software development.

Once a good solution method is determined, a program must still be written and implementation has to be completed. In this module we therefore conduct a one hour laboratory session for every three hours of lectures at Goldsmiths College, University of London. This approach is to give you the opportunity to enhance your programming skills in Java.[1] In addition, you will do two assignments to gain problem-solving experience.

[1] Or in other similar programming languages.

Textbooks

Although there are many books on algorithms and data structures available on the market, it has been difficult to find a single text that suits all the needs of this module. On the other hand, you may already have some books on algorithms and data structures from previous programming modules such as CIS212 or CIS109 that you may not have finished studying.

Instead of any single text for essential and desirable reading, we therefore recommend chapters from a number of text books. *All topics covered in these chapters and in this subjectguide are examinable* [2]. There is also a list of books for further reading, and for supporting and historical background reading in the subject guide. Details on availability of the books can be found at `www.amazon.co.uk` or in your institution libraries.

[2] See Chapter 11 for a list of important examinable topics.

The materials covered in the books may overlap and not every chapter of a book is required for the examination. Hence you are not expected to read all the books, nor all the chapters of a single book on the list. You do, however, need to get hold of at least ONE book on algorithms and data structures for frequent reference and for studying individual topics in depth. Your book does *not* have to be in Java but it should cover at least 80 percent of the examinable topics below[3]:

[3] Of course, you still need to have an access to the other 20 percent materials in various sources such as in the library.

1. Algorithms and efficiency analysis
2. Abstract data types and data structures
3. Lists, stacks, queues and sets
4. Recursions, divide and conquer
5. Trees, graphs and maps

6. Sorting, searching and hashing
7. Dynamic programming
8. Graph algorithms, greedy heuristics and approximation
9. Complexity theory
10. Text processing.

Note it would be inconvenient, if not impossible, for you to have to share a library's textbook with other students to study more than 20 percent of the examinable materials, because the book may be unavailable just when you need it urgently for a coursework or examination.

To best study module CIS226b, you need to follow closely the reading instruction for each chapter in the subject guide, paying a particular attention to whether the word 'and' is used between reading items. For example, if 'A and B' is found on the reading list, you are strongly recommended to consult the materials in both A and B.

Below is a collection of textbooks that are recommended at various points in the subject guide which are undated periodically. Note the books are updated frequently these days so search on the internet for the newest edition before any purchase.

Essential reading

The chapters of various books are specified as essential reading. These can be found at the beginning of each chapter of the subject guide. You should read carefully the specified chapter(s) in *at least* one of the books on the list for each chapter of this subject guide. If there is any doubt, you should consult other books on the list. A full list of these books can be found in the next section.

Desirable reading

The following texts are recommended because they provide the detailed background for understanding and appreciation of some topics in this module.

However, you are *not* required to study all the topics in these texts, for no single text can entirely meet the requirements of this course unit. The essential reading chapters are listed at the beginning of each chapter of the subject guide.

List A

Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in Java*. (John Wiley & Sons, Inc., 2005, fifth edition) [ISBN10 0-471-73884-0], [ISBN13 978-471-73884-8].

Duane A. Bailey *Java Structures: Data Structures in Java for the principled programmer*. (McGraw-Hill Companies, Inc. 1999, McGraw-Hill International editions) [ISBN 0-13-489428-6].

Jurg Nievergelt and Klaus H Hinrichs *Algorithms & Data Structures*. (Prentice Hall, Inc., 1993) [ISBN 0-13-489428-6].

Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957].

Supporting and historical reading

These books are of interest and are recommended, but you will be fully prepared for the examination should you have studied only those essential texts above. These are provided for completeness and to allow the interested reader to pursue some topics in more depth. You will not be examined on the content of those books listed in the references other than where the material appears in the texts listed above.

List B

Additional reading

Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7].

Jeffrey Kingston *Algorithms & Data Structures: Design, Correctness, Analysis*. (Addison-Wesley, 1997, or 1998, second edition) [ISBN 0-201-40374-9].

Richard Johnsonbaugh and Marcus Schaefer *Algorithms*. (Pearson Education International, 2004) [ISBN 0-13-122853-6].

Michael Main, *Data Structures and Other Projects Using Java*. (Addison Wesley Longman Inc., 1999) [ISBN 0-201-35744-5].

Sara Baase and Allen Van Gelder *Computer Algorithms: Introduction to Design and Analysis*. (Addison-Wesley Publishing Company, 2000) third edition [ISBN 0-201-61244-5].

Thomas H. Cormen, Stein Clifford Charles E. Leiserson and Ronald L. Rivest *Introduction to Algorithms*. (The MIT Press and McGraw-Hill Book Company, 2001) second edition [ISBN 0-262-03141-8] (MIT Press); [ISBN 0-07-013143-0] (McGraw-Hill).

Mark Allen Weiss *Data Structures and Problem Solving Using Java*. (Addison Wesley Longman Inc., 1998) [ISBN 0-201-54991-3].

Derick Wood *Data Structures, Algorithms, and Performance*. (Addison-Wesley Publishing Company, Inc., 1993) [ISBN 0-201-52148-2].

Gregory J.E. Rawlins, *Compared to What? - An Introduction to the Analysis of Algorithms*. (Computer Science Press, 1992) [ISBN 0-7167-8243-X].

Christos H. Papadimitriou *Computational Complexity*. (Addison-Wesley Publishing Company, 1994) [ISBN 0-201-53082-1].

Sara Baase *Computer Algorithms: Introduction to Design and*

Analysis. (Addison-Wesley Publishing Company, 1988) second edition [ISBN 0-201-06035-3].

Niklaus Wirth *Algorithms + Data Structures = Programs*. (London: Englewood Cliffs; Prentice-Hall, 1976) [ISBN 0130224189].

About this subject guide

This subject guide outlines the main topics in the syllabus. It can be used as a reference which summarises, highlights and draws attention to some important points of the subject. It cannot, however, replace a textbook although it is fairly self-contained. The guide sets out a sequence which helps you to study the topics in the module within limited hours. The guide provides some additional background material including examples, lab exercises and sample examination questions. It also provides guidance for further reading. recommended textbooks.

One thing you should always bear in mind is the fact that the algorithm subject, like subjects in any other area of computer science, has kept evolving and has been updated frequently. You should therefore not be surprised if you find different approaches, explanations or results in the books you read including this guide.

About CIS226b, volume 2 of CIS226

This module provides an introduction to algorithm design and analysis techniques. Our *aim* is to give students an insight into various standard abstract data types and the common techniques for designing efficient algorithms.

The *objectives* include to:

- introduce fundamental issues in algorithm design such as efficiency, recursion, abstract data types (data structures) and complexity
- demonstrate the standard techniques in algorithm design
- study some well-known problems and algorithms
- further develop your skills in Java programming.

The *learning outcomes* are:

On completion of the second half of the module, students should be able to:

- demonstrate the knowledge of techniques for identifying and solving a computing problem
- choose appropriate data structures for different computation problems
- conduct a basic analysis on time-efficiency of an algorithm in the worst case
- design efficient algorithms and implement them in Java programs

- explain the limit of computations and the complexity classes for decision problems.

Prerequisites

You should be familiar with some basic discrete mathematics, such as functions, big-O notations, sets and logarithms. These topics can be found in CIS102 or equivalent.

You should have already had some prior experience with programming in Java, which is covered in CIS109 or equivalent. This module, CIS226b, focuses on algorithm design and analysis and relies upon most of your programming skills to implement algorithms. It would be an advantage if you are familiar with built-in data structures in Java such as arrays and vectors, concepts in Java such as types versus classes, inheritance, constructor methods, method overloading, method overriding.

Most importantly, you must also have easy access to a Java platform or have a Java platform installed on a computer at home.

Installing Java

There are lots of public domain versions of Java among which the most popular one is called JDK (free). It is at `http://www.javasoft.com/` or `http://java.sun.com/`. A great amount of information is provided on these sites and you can download the software.

If you are using Linux, then the free software package normally already includes a free Java platform.

Testing the installation

An easy way to test your installation is to type the following at a command prompt:

```
java -version
```

A message in response should be seen on the screen with other system information on your platform.

For example:

```
java version "1.5.0_06"
Java(TM) 2 Runtime Environment, Standard Edition (build
1.5.0_06-b05) Java HotSpot(TM) Client VM (build 1.5.0_06-b05,
mixed mode, sharing)
```

Or, for the earlier version:

```
Kaffe Virtual Machine

Copyright (c) 1996-2004 Kaffe.org project contributors
```

```
(please see the source code for a full list of contributors).
All rights reserved. Portions Copyright (c) 1996-2002
Transvirtual Technologies, Inc.

The Kaffe virtual machine is free software, licensed under
the terms of the GNU General Public License.  Kaffe.org is
an independent, free software community project, not directly
affiliated with Transvirtual Technologies, Inc.  Kaffe is
a Trademark of Transvirtual Technologies, Inc.  Kaffe comes
with ABSOLUTELY NO WARRANTY.

Engine: Just-in-time v3   Version: 1.1.4   Java Version: 1.1
```

If you see a flawed response such as a `bad command or file name` or a `command not found` in response to your command '`java -version`', you know that your installation may have not been completely successful. This sometimes may be simply because your system cannot find the correct version of the file which runs Java programs.

As Java has grown to so many versions and variations, we recommend that you focus on the basic functions which can be run in all environments.

CmapTool

Algorithm design, like other types of design, requires a process of development from vague ideas to production details. CmapTools is free software that may help designers to ease the journey of converting their concepts to the design objects. You can find more information about the CmapTool and download the software from `http://cmap.ihmc.us`.

Study time

The materials covered in this module are taught internally at Goldsmiths College, University of London in one academic term, that is a three hour lecture and a one hour supervised lab session per week for ten weeks. For each one hour lecture, students are expected to spend at least two further hours on homework including revision, attempting exercises and lab implementation.

You would, however, normally have to double the study hours if you could not attend lectures. For example, if you self study at home, you would expect to spend six hours on intensive study of the materials and two hours for the lab exercises, plus a similar amount of additional homework time, every week for ten weeks or the equivalent.

It is, of course, impossible to tell you precisely the number of hours required for you to study the materials in this module. It may depend on many elements such as your academic background, the condition of the environment, your health status, the complexity of the subjects and your study methods. I recommend that you add an extra couple of free hours to your plan at least in the first two weeks

and record the time it took you to meet the requirements, and adjust your plan accordingly.

Study methods

One effective way to study algorithms is to commit yourself to various DIY (do it yourself) activities. You should not believe an algorithm until you have implemented and tested it. In theory, the performance of an algorithm can be explored by analysis or implementation. Implementation is not the main concern in this module. However, implementation is an important way forward. It is the only way, sometimes, to prove correctness (or more likely, incorrectness) in practice. Investigation of certain behaviours of an algorithm can also be an important motivation of algorithm design and analysis.

You should try to implement as many algorithms as possible in a conventional language such as Java. Attempting exercises and doing courseworks often offer good opportunities to help your understanding.

It can be useful to remember the Confucian[4] saying about learning as you start your studies:

[4] A famous ancient Chinese philosopher.

> Tell me and I forget;
>
> Show me and I remember;
>
> Let me do it and I understand.

As experts have predicted that more and more people in future will write programs without being programmers, you are recommended to learn the important principles and apply them in your programming practice as much as possible. The experience could be very useful for your future career whatever you do.

More specifically:

1. For every hour of study on new material in a lecture, two hours of lab work and two hours of revision or exercises are highly recommended.
2. Use examples to help gain understanding of each problem.
3. Always ask the question: 'Is there a better solution for the problem?'
4. Practise as much as you can.

Laboratory exercises

There is a one-hour supervised lab session every week for each student at Goldsmiths, University of London.

Lab exercise sheets are set for students to practise their programming skills using the theoretical knowledge gained from the course and are available soon after lectures each week. If you are studying at an institution, your lecturer may provide a similar resource.

Examination

> **Important** *The information and advice given in the following section are based on the examination structure used at the time this guide was written. However, the university can alter the format, style or requirements of an examination paper without notice. Because of this we strongly advise you to check the rubric/instructions on the paper you actually sit.*

The content covered in CIS226b will be examined in the *second* half of a three-hour examination for CIS226.

Students will normally be required to answer a number of questions;[5] each includes a few subquestions (or 'parts') in each half of the paper. These subquestions may be *Bookwork, Similar* and *Unseen*.

[5] You should check the details before the examination.

Details about the examination and revision can be found in Chapter 11.

Every year we advise the candidates to read the questions on the examination paper **carefully**. You should make sure that you fully understand what is required and what subquestions are involved in an examination question. You are encouraged to make notes (crossed through later as not to be marked), if necessary, while attempting the questions. Above all, you should be completely familiar with the course material. To achieve a good grade, you need to have prepared well for the examination and to be able to solve problems by applying the knowledge gained from your studies of the module.

Content and plan

The main topics that we normally cover internally at Goldsmiths College, University of London for CIS226b are as below, but you may adjust them according to your level and your own time available.

```
Week 1
Lecture 1-3
        The aim, objectives and plan of the course
        Problems and algorithms
        Big-O notation
        Pseudocode
        Cmaptools (http://cmap.ihmc.us)
Ex  1   Time efficiency

Week 2
Lecture 4-6
        Abstract Data Types
          array, lists, stacks, queues, sets, (trees,
          graphs, hash tables, heaps)
        Specialised data structures
        Algorithms Design and Implementation
Ex  2   Abstract Data Types
Lab 1   Estimating time efficiency
```

```
Week 3
Lecture 7-9
        Algorithm Design Techniques (1)
          Sorting, selection, searching and traversal.
Ex  3   Sorting, searching, traversal and selection
Lab 2   Implementation of ADT list, or binary tree

Week 4
Lecture 10-12
        Algorithm Design Techniques (2)
           Divide and conquer, Recursion
Ex  4   Divide and conquer, Recursion
Lab 3   Implementation of searching a sorted list, or
           traversal of a connected graph.

Week 5
Lecture 13-15
        Algorithm Design Techniques (3)
           Dynamic programming
        trees, graphs
Ex  5   Dynamic programming
Lab 4   Implementation of a Recursion programme

Study week
        no lectures/labs

Week 7
Lecture 16-18
        Algorithm Design Techniques (4)
           Greedy approach and heuristics
        hash tables, heaps
Ex  7   Greedy approach and heuristics
Lab 5   Implementation of dynamic programming

Week 8
Lecture 19-21
        Limits of Computing
           Intractable problems and approximation
           Introduction to NP-completeness
Ex  8   Intractability and approximation
Lab 7   Greedy approach and heuristics

Week 9
Lecture 22-24
        Some well known problems and algorithms (1)
           String matching problems
Ex  9   String matching problems
Lab 8   Intractability and approximation

Week 10
Lecture 25-27
        Some well known problems and algorithms (2)
           Computational geometry problems
Ex  10  Computational geometry problems
Lab 9   String matching problems

Week 11
Lecture 28-30
```

```
        Revision
Ex  11  Sample examination questions
Lab 11  Computational geometry problems
```

Activity 0.0

WEB SITES[6] AND SOFTWARE

[6] These addresses were accessible when the Guide was written. In case they have changed, you may use a search engine such as Google to search the new web address.

1. Free Java Books
   (a) Thinking in Java
   `http://www.mindview.net/Books/TIJ/`
   (b) Java Gently
   `http://javagently.cs.up.ac.za/jg3e/`
2. What are covered in the first year Java courses in other places?
   (a) David J.Eck's Java course:
   `http://math.hws.edu/eck/cs124/`
   (b) Java Tutorial
   `http://java.sun.com/docs/books/tutorial/`
3. Installing Java system

   ```
   http://burks.bton.ac.uk
   http://java.sun.com/
   http://textpad.com
   ```

4. Download and install CmapTool

   ```
   http://cmap.ihmc.us
   ```

Chapter 1

Algorithm analysis

1.1 Essential reading

Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957]. Chapter 2

Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in Java*. (John Wiley & Sons, Inc., 2005, fifth edition) [ISBN10 0-471-73884-0], [ISBN13 978-471-73884-8]. Chapter 4

Mark Allen Weiss *Data Structures and Problem Solving Using Java*. (Addison Wesley Longman Inc., 1998) [ISBN 0-201-54991-3]. Chapter 5

Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 9

Jurg Nievergelt and Klaus H Hinrichs *Algorithms & Data Structures*. (Prentice Hall, Inc., 1993) [ISBN 0-13-489428-6]. Chapter 16

1.2 Learning outcomes

This chapter is concerned with *algorithms and algorithm analysis*. Having read this chapter and consulted the relevant material you should be able to:

- explain the term *algorithm* and the concept of the efficiency of an algorithm in terms of big-O notation
- develop a perspective on the study of computer science beyond the learning of a particular programming language such as Java
- describe commonly-used big-O categories.

1.3 Problems and algorithms

In this section, we introduce the concept of algorithms and discuss the issues of fundamental analysis of algorithms.

A *problem* in this course is a general question to be answered, usually possessing one or more *parameters*. It can be specified by describing the form of parameters taken and the questions about the parameters. An *instance* of a problem is an assignment of values to the parameters. An *algorithm* is a clearly specified set of simple instructions to be followed to solve a problem. In other words, it is a step-by-step procedure for taking *any* instance of a problem and producing a correct answer for that instance.

Example 1.1 Finding the minimum*:*

Problem *Given a non-empty set of numbers, what is the minimum element of the set?*

Instance:
What is the minimum element of (2,5,8,3) entered on a single line from the keyboard?

Algorithm 1.1 is a solution for solving the problem of finding the minimum of a set of data that are input from the keyboard. The algorithm should give a correct answer for any set of data.

Algorithm 1.1 Minimum key

```
INPUT:   nothing
OUTPUT:  the minimum
1: read min;
2: while not eoln do
3:    read x
4:    if x < min then
5:       min ← x
6:    end if
7: end while
8: print min;
```

1.3.1 Implementation

Algorithm 1.1 can be *implemented* to a Java method and run on a computer:

```
import java.util.Scanner;

int min() {

   Scanner input = new Scanner( System.in );
   System.out.println("x=?  (999 to end) ");
   int x = input.nextInt();
   int min = x;

   while (x!=999) {
      System.out.println("x=?  (999 to end) ");
      x = input.nextInt();
      if (x < min) {
         min = x;
      }
   }
   return min;
}
```

1.4 Pseudo-code for algorithm description

Using a natural language such as plain English to describe an algorithm is not impossible. However, one quickly realises that any human language is too rich to be concise or precise enough for the task. A common practice is to use so-called *pseudo-code* to describe algorithms (see Appendix C for examples). The syntax of any pseudo-code is similar to that of a high-level computer language such as Java. It is therefore much more convenient for an algorithm

in pseudo-code to be translated into a computer program in some higher language than from a human language.

Thus once an algorithm for a problem is properly developed, it is a relatively easy matter to implement or translate it into a program in some computer language.

In this course, instead of giving a formal definition of the pseudo-code, we just borrow conventional syntax in Java, or the like. We encourage you to separate the algorithm design and implementation in the first and second stages respectively. During the first stage, i.e. the algorithm design stage we ignore the implementation details and focus on problem-solving techniques and algorithmic issues.

1.5 Efficiency

In this module, our goal is not only to develop a working algorithm, but also an efficient algorithm for a given problem.

The speed of hardware computation of basic operations has been improved dramatically, but efficiency matters more than ever today. This is because our ambition for computer applications has grown with computer power. Many areas demand a great increase in speed of computation. Examples include the simulation of continuous systems, high resolution graphics, and the interpretation of physical data, medical applications, and information systems.

On the other hand (and more importantly), an algorithm may be so inefficient that, even with computation speed vastly increased, it would not be possible to obtain a result within a useful period of time. The time that many algorithms take to execute is a non-linear function of the input size. This can reduce their ability to benefit from the increase in speed when the input size is large.

Example 1.2 *A particular sorting algorithm takes n^2 comparisons to sort n numbers. Suppose that computing speed increases by a factor of 100. In the time that it was required to take to execute the n^2 comparisons, it is now possible to do $100n^2 = (10n)^2$ comparisons. Unfortunately, with 100 times speed-up, only 10 times as many numbers can be sorted as before.*

1.6 Measures of performance

Naturally, the efficiency of an algorithm is estimated by its performance. The performance of an algorithm can be measured by the *time* and *the space* required in order to fulfil a task. The time and space requirement of an algorithm is called the *computational complexity* of the algorithm. The greater the amount of the time and space required, the more complex is the algorithm.

The *time complexity* of an algorithm is, loosely speaking, an imaginary *execution time* of the algorithm. The execution time can be measured by the number of some *characteristic operations* performed by the algorithm in order to transform the input data to

the results.

Note that we do *not* measure the time complexity by the running time of a program. This means that we do not use the common time units such as *second, minute* and *hour*. The unit of the time complexity, strictly speaking, should be the *number of execution steps*, although we often do not use any unit.

An algorithm consists of a set of ordered instructions and the time complexity, that is, the number of execution steps in an algorithm, is irrelevant to the real time.

Note our main interest here is in an algorithm instead of a program. A program is an implementation of an algorithm. The execution time of a program depends on the implementation including not only the operating system but also the speed of the computer itself. The same program may run faster on a computer with a faster CPU, but the same algorithm should perform the same number of algorithmic steps to accomplishment a task.

Normally we are concerned with the *time complexity* rather than *space complexity* of an algorithm. The reasons are that firstly it becomes easier and cheaper to obtain space. Secondly techniques to achieve space efficiency by spending more time are available.

In what follows, we use 'complexity' to mean the *time complexity* if not otherwise indicated.

Observation

- The complexity of an algorithm normally depends on the size of input.
- The number of operations may depend on a particular input.

Solution

- For different sets of input data, we analyse the performance of an algorithm in the *worst* case or in the *average* case.
- For different algorithms, we focus on the *growth rate* of the time taken by the algorithms as the input size increases.

The time complexity of an algorithm can be expressed by a function of input size: $T(n)$. We are normally interested in the behaviour of $T(n)$ as n grows large.

1.7 Algorithm analysis

With the measures of performance introduced earlier, it is possible to conduct an analysis and estimate the cost of an algorithm.

Many reasons can be given to explain why we need algorithm analysis. One main reason is that there are usually several algorithmic ideas for a problem, and we would like to eliminate the inefficient algorithms *early*. By early, we mean that we would like to compute or estimate the computational complexity of any two algorithms *before* actually implementing (coding) them into programs.

Secondly, algorithms may behave differently for different input sizes, and we would like to estimate their computational complexity for *large* inputs. For some problems we simply have not found an efficient algorithm yet, but we may find those algorithms feasible and useful for input within some limited range.

Furthermore, the ability to do an analysis usually provides insight into designing efficient algorithms. The analysis also could pinpoint the bottlenecks which should be taken care of during coding.

In what follows, we shall introduce some fundamental methods for algorithm analysis. Before moving on, we have to first agree a model of 'normal' computer.

1.8 Model of computation

We adopt a so-called 'random access machine' (RAM) model. In this model, certain hardware constraints for a real computer are ignored in order to focus on algorithmic issues. Routine operations at machine level, such as fetching instructions or data from the memory are also ignored for the same reason.

We assume that our computer has the following convenient properties:

1. It has a single processor (CPU) and runs our pseudo-code algorithms in a sequential manner. A pseudo-code algorithm consists of an ordered sequence of instructions in pseudo-code (Appendix C). The instructions in each algorithm are executed one after another in the given order.
2. It takes exactly one *time unit* to execute a standard instruction (in pseudo code) for operations such as addition, subtraction, multiplication, division, comparison, assignment and conditional control. No complex operations, such as sorting and matrix inversion, can be done in one time unit.
3. The storage for integers is of a fixed size, for example, 32 bits.
4. It has an infinitely large memory.[1]

[1] So there is no need to consider any overflow issues. This assumption is based on the fact that memory techniques has developed to allow the logical memory to be of a larger size than its physical size.

The assumptions are necessary because our analysis result depends on the model. For example, assigning 100 data into an array would require 100 unit times in our model. The same task can be done in one time unit, however, in a parallel computation model such as a 'parallel random-access machine' (PRAM), in which the 100 memory cells can be accessed simultaneously.

The assumptions help keep analysis feasible and focused, for certain hardware details are ignorable from an algorithmic point of view. For example, standard operations such as *addition, subtraction, multiplication, division, comparison, assignment* and *conditional control* would require different amount of time to run on a real computer. The storage of real computers is limited and the memory for integers and reals may be of a different size. However, taking the difference made by these hardware details into consideration gives little impact on the result in comparison of different algorithms, because these standard operations are required for almost every algorithm. Taking too many details into consideration can, if anything, make an analysis too complicated to be carried out.

1.8.1 Counting steps

Given two algorithms A_1 and A_2, which one is more efficient? In other words, which one has lower computational complexity? To answer this question, one way is to simply count the *execution* steps of each algorithm and compare the numbers of the steps of the two.

Example 1.3
Problem: *Compute* $\sum_{k=1}^{n} k$, *where* n *is an integer.*

Algorithm 1.2 Sum1(n)

INPUT: n
OUTPUT: sum
1: $sum \leftarrow 0$
2: **for** $k \leftarrow 1, k \leq n, k \leftarrow k+1$ **do**
3: $sum \leftarrow sum + k$
4: **end for**
5: print sum

From the fact $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$, *we have*

Algorithm 1.3 Sum2(n)

INPUT: n
OUTPUT: sum
1: print $n(n+1)/2$

1.8.2 Implementation

These algorithms can be implemented to the following Java methods:

```
int sum1( int n ) {
   int sum = 0;
   for (int k=1; k<=n; k++) {
   sum = sum + k;
   }
   return sum;
}

int sum2( int n ) {
   int sum;
   return (n*(n+1)/2);
}
```

We look at the *time* complexity by counting the steps taken in execution. Let any assignment, arithmetic computation `+,-,*,/`, `read`, `print` be all counted as *one* step. So for *Algorithm 1.2*, it takes $1 + n \times 1 + 1 = n + 2$ steps; and for *Algorithm 1.3*, it takes $1 + 1 + 1 = 3$ steps to execute. Obviously, *Algorithm 1.3* is more efficient in terms of *execution time*.

How about the space efficiency? Let a simple variable require one unit of storage. *Algorithm 1.2* needs three units since it involves three variables `sum`, `k` and `n`, and *Algorithm 1.3* only needs one unit since it involves only one variable `n`. We can therefore conclude that *Algorithm 1.3* is more efficient in terms of *storage*, too.

In fact, *Algorithm 1.3* has an important advantage, that is, it takes *constant* time to execute no matter how large n is. This means that it takes the same amount of time to run no matter how many such numbers need to be added up.

We have done an analysis. Had we, however, to undertake all the counting every time to analyse an algorithm, the task would be tedious and quickly become *infeasible*. We need some easier approach.

1.8.3 Characteristic operations

A complexity analysis gives an *estimate* of the resources consumed by an algorithm. The relationship between the amount of time and space allows us to focus on the time efficiency only. The time complexity is, after all, $T(n)$, a *function* of the input size n. The more data, the longer it takes to run the algorithm. The task of counting steps can be made easier if a reasonable measure of the *input* size of some major operations can be established and then only those operations relevant to the input size need to be considered.

Example 1.4 *For each algorithm below, we choose a relevant characteristic operation:*

1. *An algorithm searching an element x in a list of names:*
   Choose the comparison of x with an entry in the list;
2. *An algorithm multiplying two matrices with real entries:*
   Choose the multiplication of two real numbers;
3. *An algorithm sorting a list of numbers:*
   Choose the comparison of two list entries;
4. *An algorithm to traverse a binary tree:*
   Choose the visit of a tree node.

The characteristic operations could be some very expensive operations compared to others, or they might be of some theoretical interest. It allows a good sense of flexibility to choose these fundamental operations as a measure of time complexity.

To ease the analysis of algorithms, it would be useful to summarise the time complexity for common algorithmic structures below as a reference. You are encouraged to spend some time to derive the formulae by yourself and then compare yours with the ones in a textbook.

- For loops:
- Consecutive statements:
- If-then-else:

Think of an algorithm which takes a number of steps of the following order:

- Logarithmic
- Exponential.

1.9 Asymptotic behaviour

We are often interested in the rate of growth of the time required for an algorithm when the input size gets larger. So the lower order terms of the time complexity $T(n)$ could be ignored, where n is a positive integer. In other words, we only need to master the asymptotic behaviour of $T(n)$. Here the term *asymptotic* means approximate in a specific way.

Example 1.5 *Suppose that the time complexity of an algorithm is*

$$T(n) = \frac{1+n^2}{n}$$

where n is the input size to the algorithm.

It is easy to see some asymptotic behaviours of $T(n)$ such as, $T(n) \sim n$ when $n \to \infty$ [2]

because

$$T(n) = \frac{1+n^2}{n} = \frac{\frac{1}{n}+n}{1} \sim \frac{0+n}{1} = n$$

Similarly, $T(n) \sim \frac{1}{n}$ when $n \to 0$.

Here n and $\frac{1}{n}$ are both simpler than $T(n)$ and it is easier to handle their behaviour in an analysis.

We say that n and $\frac{1}{n}$ are *asymptotic behaviour* of $T(n)$ when $n \to \infty$ and $n \to 0$ respectively.

[2] Here symbol '$\sim$' means 'will be approaching'; and '$\to$' means 'goes to'. So '$T(n) \sim n$ when $n \to \infty$' reads 'The values $T(n)$ will be approaching n when n grows to infinitely large.'.

1.9.1 Big O notations

In general, the asymptotic behaviour of functions can be described by so-called "big-O" notations, often consisting of four members $O()$, $\Omega()$, $\Theta()$ and $o()$. They are called "big-oh", "omega", "theta" and "small-oh" respectively.

Let g be a function of n. Each of $O(g)$, $\Omega(g)$, $\Theta(g)$ and $o(g)$ defines a set of functions related to g.

$O(g(n))$ is a set of functions that grow *at most* as fast as g when $n \to \infty$.

$\Omega(g(n))$ is a set of functions that grow *at least* as fast as g when $n \to \infty$.

$\Theta(g(n))$ is a set of functions that have *the same* growth rate as g when $n \to \infty$.

$o(g(n))$ is a set of functions that grow *slower* than g when $n \to \infty$.

Conventionally, we use $T(n) = O(g(n))$ to mean $T(n) \in O(g(n))$. We define $T(n) = O(g(n))$ if there are positive constants c and n_0 such that $T(n) \leq cg(n)$ when $n \geq n_0$.

Similarly, $T(n) = \Omega(g(n))$ if there are positive constants c and n_0 such that $T(n) \geq cg(n)$ when $n \geq n_0$.

$T(n) = \Theta(g(n))$ if and only if $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$.

$T(n) = o(g(n))$ if $T(n) = O(g(n))$ and $T(n) \neq \Theta(g(n))$.

Example 1.6 *Let $T_A(n)$ be the time complexity of an algorithm A, and n is the input size of the algorithm. Suppose $T_{A_1} = \frac{n^2}{2}$ and $T_{A_2} = 7n$.*

Illustrating definitions, we see that $7n$ is $O(n^2)$ but that $7n \neq \Theta(\frac{n^2}{2})$ because $\frac{n^2}{2} \neq \Theta(7n)$.

1.9.2 Comparing orders of two functions

When comparing two functions in terms of *order*, it is often convenient to take the alternative definitions: Let $\lim_{n\to\infty} \frac{T(n)}{g(n)} = L$. The limit can have four possible values:

1. If $L = 0$ then $T(n) = o(g(n))$
2. If $L = a \neq 0$ then $T(n) = \Theta(g(n))$
3. If $L = \infty$ then $g(n) = o(T(n))$
4. If L oscillates then there is no relation (but this will not happen in our context).

Example 1.7 *Given $T_{A_1}(n) = 1000n$ and $T_{A_2}(n) = n^3$, which function grows faster when $n \to \infty$? What is the relationship between the two functions?*

Solution $\lim_{n\to\infty} \frac{1000n}{n^3} = 0$. *Therefore, $T_{A_1}(n)$ is $o(n^3)$, which means $T_{A_1}(n)$ grows strictly slower than $T_{A_2}(n)$.*

1.10 The worst and average cases

The behaviour of an algorithm usually depends not only on the size of the input, but also on the input itself. Look at Algorithm 1.4 which determines whether integer x is an element of array $Y[0..n-1]$, where n is a non-negative integer.

Example 1.8

Algorithm 1.4 boolean foundFirstX(int x, int Y[])

```
1: i ← 0
2: boolean found ← false
3: while (notfound) && (i < length(Y)) do
4:     found ← (x = Y[i])
5:     i ← i + 1
6: end while
7: returnfound
```

The time complexity of `foundFirstX(x,Y)` depends on the value of x and on the contents of the array Y. For example, if $Y[0] = x$, that is, the first element in array Y equals to x, then the while loop (step 3–6 in the algorithm) will only be executed once, and the number of execution steps is only $1 + 1 + 1 \times (1 + 1 + 1 + 1) + 1 = 7$,

including step 1, step 2, $1 \times$ (step $3, 4, 5, 6$), and step 7. If x equals the kth element of Y, the number of execution steps becomes $1 + 1 + k \times (1 + 1 + 1 + 1) + 1 = 3 + 4k$, where $k \leq length(Y)$, including step 1, step 2, $k \times$ (step $3, 4, 5, 6$), and step 7. Each of these different situations is called a *case*. In each case, the algorithm gives a different performance.

We therefore need to consider the behaviour of the algorithm for two special cases, namely the *worst* case and the *average* case.[3]

[3] Although it is desirable, the best case is not very interesting, for it does not help much with a budget. In contrast, the *worst* case prepares us for the most costly situation, and the *average* case tells us what to normally expect.

In terms of time complexity, the worst case is the situation where the algorithm would take the longest time. The average case is the case where the average behaviour is estimated after every instance of the problem has been taken into consideration. We define two functions of n, the input size, for the two cases respectively.

In general, an algorithm may accept k different instances of size n. Let $T_i(n)$ be the time complexity of the algorithm when given the ith instance, for $1 \leq i \leq k$. Let p_i be the probability that this instance occurs.

Then the time complexity for

- The worst case:
$$W(n) = \max_{1 \leq i \leq k} T_i(n)$$
- The average case:[4]
$$A(n) = \sum_{i=1}^{k} p_i T_i(n)$$

[4] This can be extremely complex sometimes.

In words, $W(n)$ is the maximum number of characteristic operations performed by the algorithm on any input of size n. $A(n)$ gives the behaviour of the algorithm on average for different instances. Clearly, $A(n) \leq W(n)$.

The worst case analysis could help to provide an estimate for a time limit for a particular implementation of an algorithm. It is particularly useful in real time applications. The average case analysis is more meaningful in providing an overall picture because it computes the number of steps performed for each possible input instance of size n and then takes the (probability-weighted) average.

In this course, we shall consider only the worst case analysis if not specified otherwise.

The result of an algorithm analysis can sometimes turn out to be unsatisfactory or extremely difficult to achieve. In these cases, an empirical [5] approach should be considered.

[5] An empirical approach is a means of investigating the efficiency of an algorithm by experiments.

1.10.1 Implementation

Algorithm 1.4 can be realised in the following Java method:

```
boolean foundFirstX( int x, int Y[]) {
   boolean found = false;
   int i=0;
   while ( ( !found ) && ( i< Y.length) ) {
```

```
        found = ( x==Y[i] );
        i++;
      }
      return found;
   }
```

It is easy to modify the program slightly in order to compute the *actual* number of Java statements executed. In the example below, we define a variable `count` before (or after) each statement, and display the value of `count` at a few places of the program.

```
   boolean foundFirstX( int x, int Y[]) {
      int count = 1;
      boolean found = false;
      count ++;
      int i=0;
      System.out.println("Step 1--2: "+count);
      count ++;
      while ( ( !found ) && ( i< Y.length) ) {
         count ++;
         found = ( x==Y[i] );
         count ++;
         i++;
         count ++;  // for while
      }
      System.out.println("Step 1--6: "+count);
      count ++;
      System.out.println("All steps: "+count);
      return found;
   }
```

You can conduct an experiment in which different integer arrays Y[] are input and the different number of execution steps are displayed on your screen. For example, if call the methods with "`int A[] = 9, 4, 5, 7, 1, 2;`", you would see

```
...
Step 1 and 2: 2
Step 1--6: 21
All steps: 22
...
```

If with "`int A[] = 2, 4, 5, 7, 1, 2;`", you would get

```
...
Step 1--2: 2
Step 1--6: 6
All steps: 7
...
```

This can be implemented in two classes as in Example 1.9:

1. Place the method `foundFirstX` in Section 1.10.1 in a class `experimentCount`;
2. Write the main class `test` which inputs a `x` and an array, say, `A`, and print out the `foundFirstX(x, A)`.

Example 1.9 `class experimentCount {`
`  boolean foundFirstX( int x, int Y[]) {`

```
      ...     // copy the method here
  }
}

class test {

   public static void main(String args[]) {
      experimentCount acount = new experimentCount();

      int A[] = {9, 4, 5, 7, 1, 2};
      int B[] = {2, 4, 5, 7, 1, 2};

      acount.printArray(A);
      System.out.println(acount.foundFirstX(2, A));
      System.out.println();
      acount.printArray(B);
      System.out.println(acount.foundFirstX(2, B));
   }
}
```

1.10.2 Typical growth rates

Some functions are commonly seen with typical growth rates in the algorithm analysis. We list some common ones here.

Note All logarithms in this subject guide are of base 2 if not stated otherwise.

Functions	Name
c	constant
$\log n$	logarithmic
$\log^2 n$	log-squared
n	linear
$n \log n$	n-log-n
n^2	quadratic
n^3	cubic
2^n	exponential

1.11 Verification of an analysis

It is important to conduct an algorithm analysis, before any implementation, to avoid any unnecessary expensive labour. It is even more important to make sure that the analysis result is correct. Hence verification of an analysis is necessary and highly recommended, although it sometimes turns out to be difficult.

For example, we can always:

- check if the empirical running time matches the running time predicted by the analysis
- for a range of n, compute the value $T(n)/f(n)$ where $f(n)$ is the analysis result and $T(n)$ is the empirically observed running time. This should be ideally a constant as n varies.

Activity 1.11

TIME COMPLEXITY

1. Discuss briefly the time complexity in the *worst* case for the algorithm below. Indicate the input, output of the algorithm and the main comparison you have counted.

Algorithm 1.5 insertionsort(int array[0..n-1])

1: **for** $i \leftarrow 1, i \leq n-1, i++$ **do**
2: $current \leftarrow array[i]$
3: $position \leftarrow i-1$
4: **while** $position \geq 1$ and $current < array[position]$ **do**
5: $array[position+1] \leftarrow array[position]$
6: $position \leftarrow position - 1$
7: **end while**
8: $array[position+1] \leftarrow current$
9: **end for**

2. Consider the big O behaviour of the code below in terms of N. Discuss briefly its time complexity.

1: $k \leftarrow 1$
2: **repeat**
3: $k \leftarrow 2 \times k$
4: **until** $k \geq N$

3. A certain algorithm always requires 1000 operations, regardless of the amount of data input. Provide a big-O classification of the algorithm that reflects the efficiency of the algorithm as accurately as possible.
4. Modify the main class `test` in Example 1.9 (Section 1.10.1) so that the program can take a `x` and an array `A` of integers from the keyboard, and print out the number of execution steps of program `foundFirstX(x, A)`.

   Hint You only need to re-write some lines of the main method below (see the comments):

```
class test {

   public static void main(String args[]) {
      experimentCount acount = new experimentCount();

      // ... to be modified from here to the end.
      int A[] = {9, 4, 5, 7, 1, 2};
      int B[] = {2, 4, 5, 7, 1, 2};
      acount.printArray(A);
      System.out.println(acount.foundFirstX(2, A));
      System.out.println();
      acount.printArray(B);
      System.out.println(acount.foundFirstX(2, B));
   }
}
```