

## ALGORITHM DESIGN WITH HASKELL

This book is devoted to five main principles of algorithm design: divide and conquer, greedy algorithms, thinning, dynamic programming, and exhaustive search. These principles are presented using Haskell, a purely functional language, leading to simpler explanations and shorter programs than would be obtained with imperative languages. Carefully selected examples, both new and standard, reveal the commonalities and highlight the differences between algorithms. The algorithm developments use equational reasoning where applicable, clarifying the applicability conditions and correctness arguments. Every chapter concludes with exercises (nearly 300 in total), each with complete answers, allowing the reader to consolidate their understanding and apply the techniques to a range of problems. The book serves students (both undergraduate and postgraduate), researchers, teachers, and professionals who want to know more about what goes into a good algorithm and how such algorithms can be expressed in purely functional terms.

Cambridge University Press  
978-1-108-49161-7 — Algorithm Design with Haskell  
Richard Bird , Jeremy Gibbons  
Frontmatter  
[More Information](#)

---

# ALGORITHM DESIGN WITH HASKELL

RICHARD BIRD

*University of Oxford*

JEREMY GIBBONS

*University of Oxford*



CAMBRIDGE  
UNIVERSITY PRESS

Cambridge University Press  
978-1-108-49161-7 – Algorithm Design with Haskell  
Richard Bird, Jeremy Gibbons  
Frontmatter  
[More Information](#)

CAMBRIDGE  
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom  
One Liberty Plaza, 20th Floor, New York, NY 10006, USA  
477 Williamstown Road, Port Melbourne, VIC 3207, Australia  
314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India  
79 Anson Road, #06–04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

[www.cambridge.org](http://www.cambridge.org)

Information on this title: [www.cambridge.org/9781108491617](http://www.cambridge.org/9781108491617)

DOI: 10.1017/9781108869041

© Richard Bird and Jeremy Gibbons 2020

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2020

Printed in the United Kingdom by TJ International Ltd, Padstow Cornwall

*A catalogue record for this publication is available from the British Library.*

ISBN 978-1-108-49161-7 Hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Cambridge University Press  
978-1-108-49161-7 — Algorithm Design with Haskell  
Richard Bird , Jeremy Gibbons  
Frontmatter  
[More Information](#)

---

For Stephen Gill (RB) and Sue Gibbons (JG).

Cambridge University Press  
978-1-108-49161-7 — Algorithm Design with Haskell  
Richard Bird , Jeremy Gibbons  
Frontmatter  
[More Information](#)

---

---

## Contents

	<i>Preface</i>	<i>page xiii</i>
	PART ONE BASICS	1
1	Functional programming	5
	1.1 Basic types and functions	5
	1.2 Processing lists	7
	1.3 Inductive and recursive definitions	9
	1.4 Fusion	11
	1.5 Accumulating and tupling	14
	1.6 Chapter notes	16
	References	16
	Exercises	16
	Answers	19
2	Timing	25
	2.1 Asymptotic notation	25
	2.2 Estimating running times	27
	2.3 Running times in context	32
	2.4 Amortised running times	34
	2.5 Chapter notes	38
	References	38
	Exercises	38
	Answers	40
3	Useful data structures	43
	3.1 Symmetric lists	43

viii	Contents	
	3.2 Random-access lists	47
	3.3 Arrays	51
	3.4 Chapter notes	53
	References	54
	Exercises	54
	Answers	56
	 PART TWO DIVIDE AND CONQUER	 59
4	Binary search	63
	4.1 A one-dimensional search problem	63
	4.2 A two-dimensional search problem	67
	4.3 Binary search trees	73
	4.4 Dynamic sets	81
	4.5 Chapter notes	84
	References	84
	Exercises	85
	Answers	87
5	Sorting	93
	5.1 Quicksort	94
	5.2 Mergesort	96
	5.3 Heapsort	101
	5.4 Bucketsort and Radixsort	102
	5.5 Sorting sums	106
	5.6 Chapter notes	110
	References	110
	Exercises	111
	Answers	114
6	Selection	121
	6.1 Minimum and maximum	121
	6.2 Selection from one set	124
	6.3 Selection from two sets	128
	6.4 Selection from the complement of a set	132
	6.5 Chapter notes	135
	References	135
	Exercises	135
	Answers	137



## Contents

ix

	PART THREE GREEDY ALGORITHMS	141
7	Greedy algorithms on lists	145
	7.1 A generic greedy algorithm	145
	7.2 Greedy sorting algorithms	147
	7.3 Coin-changing	151
	7.4 Decimal fractions in $\text{\TeX}$	156
	7.5 Nondeterministic functions and refinement	161
	7.6 Summary	165
	7.7 Chapter notes	165
	References	166
	Exercises	166
	Answers	170
8	Greedy algorithms on trees	177
	8.1 Minimum-height trees	177
	8.2 Huffman coding trees	187
	8.3 Priority queues	196
	8.4 Chapter notes	199
	References	199
	Exercises	199
	Answers	201
9	Greedy algorithms on graphs	205
	9.1 Graphs and spanning trees	205
	9.2 Kruskal's algorithm	208
	9.3 Disjoint sets and the union–find algorithm	211
	9.4 Prim's algorithm	215
	9.5 Single-source shortest paths	219
	9.6 Dijkstra's algorithm	220
	9.7 The jogger's problem	224
	9.8 Chapter notes	228
	References	228
	Exercises	229
	Answers	231
	PART FOUR THINNING ALGORITHMS	237
10	Introduction to thinning	241
	10.1 Theory	241
	10.2 Paths in a layered network	244
	10.3 Coin-changing revisited	248

x	Contents	
	10.4 The knapsack problem	252
	10.5 A general thinning algorithm	255
	10.6 Chapter notes	257
	References	257
	Exercises	257
	Answers	261
11	Segments and subsequences	267
	11.1 The longest upsequence	267
	11.2 The longest common subsequence	270
	11.3 A short segment with maximum sum	274
	11.4 Chapter notes	280
	References	281
	Exercises	281
	Answers	283
12	Partitions	289
	12.1 Ways of generating partitions	289
	12.2 Managing two bank accounts	291
	12.3 The paragraph problem	294
	12.4 Chapter notes	299
	References	300
	Exercises	300
	Answers	303
	<b>PART FIVE DYNAMIC PROGRAMMING</b>	<b>309</b>
13	Efficient recursions	313
	13.1 Two numeric examples	313
	13.2 Knapsack revisited	316
	13.3 Minimum-cost edit sequences	319
	13.4 Longest common subsequence revisited	322
	13.5 The shuttle-bus problem	323
	13.6 Chapter notes	326
	References	326
	Exercises	327
	Answers	330
14	Optimum bracketing	335
	14.1 A cubic-time algorithm	336
	14.2 A quadratic-time algorithm	339
	14.3 Examples	341

	Contents	xi
14.4	Proof of monotonicity	345
14.5	Optimum binary search trees	347
14.6	The Garsia–Wachs algorithm	349
14.7	Chapter notes	358
	References	358
	Exercises	359
	Answers	362
PART SIX EXHAUSTIVE SEARCH		365
15	Ways of searching	369
15.1	Implicit search and the $n$ -queens problem	369
15.2	Expressions with a given sum	376
15.3	Depth-first and breadth-first search	378
15.4	Lunar Landing	383
15.5	Forward planning	386
15.6	Rush Hour	389
15.7	Chapter notes	393
	References	394
	Exercises	395
	Answers	398
16	Heuristic search	405
16.1	Searching with an optimistic heuristic	406
16.2	Searching with a monotonic heuristic	411
16.3	Navigating a warehouse	415
16.4	The 8-puzzle	419
16.5	Chapter notes	425
	References	425
	Exercises	426
	Answers	428
	<i>Index</i>	432

Cambridge University Press  
978-1-108-49161-7 — Algorithm Design with Haskell  
Richard Bird , Jeremy Gibbons  
Frontmatter  
[More Information](#)

---

---

## Preface

Our aim in this book is to provide an introduction to the principles of algorithm design using a purely functional approach. Our language of choice is Haskell and all the algorithms we design will be expressed as Haskell functions. Haskell has many features for structuring function definitions, but we will use only a small subset of them.

Using functions, rather than loops and assignment statements, to express algorithms changes everything. First of all, an algorithm expressed as a function is composed of other, more basic functions that can be studied separately and reused in other algorithms. For instance, a sorting algorithm may be specified in terms of building a tree of some kind and then flattening it in some way. Functions that build trees can be studied separately from functions that consume trees. Furthermore, the properties of each of these basic functions and their relationship to others can be captured with simple equational properties. As a result, one can talk and reason about the ‘deep’ structure of an algorithm in a way that is not easily possible with imperative code. To be sure, one can reason formally about imperative programs by formulating their specifications in the predicate calculus, and using loop invariants to prove they are correct. But, and this is the nub, one cannot easily reason about the properties of an imperative program directly in terms of the language of its code. Consequently, books on formal program design have a quite different tone from those on algorithm design: they demand fluency in both the predicate calculus and the necessary imperative dictions. In contrast, many texts on algorithm design traditionally present algorithms with a step-by-step commentary, and use informally stated loop invariants to help one understand why the algorithm is correct.

With a functional approach there are no longer two separate languages to think about, and one can happily calculate better versions of algorithms, or parts of algorithms, by the straightforward process of equational reasoning. That, perhaps, is the main contribution of this book. Although it contains a fair amount of equational reasoning, we have tried to maintain a light touch. The plain fact of the matter is

that calculation is fun to do but boring to read – well, too much of it is. Although it does not matter very much whether imperative algorithms are expressed in C or Java or pseudo-code, the situation changes completely when algorithms are expressed functionally.

Many of the problems considered in this book, especially in the later parts, begin with a specification of the task in hand, expressed as a composition of standard functions such as maps, filters, and folds, as well as other functions such as *perms* for computing all the permutations of a list, *parts* for computing all the partitions, and *mtrees* for building all the trees of a particular kind. These component functions are then combined, or *fused*, in various ways to construct a final algorithm with the required time complexity. A final sorting algorithm may not refer to the underlying tree, but the tree is still there in the structure of the algorithm. The notion of fusion dominates the technical and mathematical aspects of the design process and is really the driving force of the book.

The disadvantage for any author of taking a functional approach is that, because functional languages such as Haskell are not so well known as mainstream procedural languages, one has to spend some time explaining them. That would add substantially to the length of the book. The simple solution to this problem is just to assume the necessary knowledge. There is a growing range of textbooks on languages like Haskell, including our own *Thinking Functionally with Haskell* (Cambridge University Press, 2014), and we will just assume the reader is familiar with the necessary material. Indeed, the present book was designed as a companion volume to the earlier book. A brief summary of what we do assume, and an even briefer reprise of some essential ideas, is given in the first chapter, but you will probably not be able to learn enough about Haskell there to understand the rest of the book. Even if you do know something about functional programming, but not about how equational reasoning enters the picture (some books on functional programming simply don't mention equational reasoning), you will probably still have to refer to our earlier book. In any case, the mathematics involved in equational reasoning is neither new nor difficult.

Books on algorithm design traditionally cover three broad areas: a collection of design principles, a study of useful data structures, and a number of interesting and intriguing algorithms that have been discovered over the centuries. Sometimes the books are arranged by principles, sometimes by topic (such as graph algorithms, or text algorithms), and sometimes by a mixture of both. This book mostly takes the first approach. It is devoted to five main design strategies underlying many effective algorithms: divide and conquer, greedy algorithms, thinning algorithms, dynamic programming, and exhaustive search. These are the design strategies that every serious programmer should know. The middle strategy, on thinning algorithms, is new, and serves in many problems as an alternative to dynamic programming.

Each design strategy is allocated a part to itself, and the chapters on each strategy cover a variety of algorithms from the well-known to the new. There is only a little material on data structures – only as much as we need. In the first part of the book we do discuss some basic data structures, but we will also rely on some of Haskell’s libraries of other useful ways of structuring data. One reason for doing so is that we wanted the book not to be too voluminous; another reason is that there does exist one text, Chris Okasaki’s *Purely Functional Data Structures* (Cambridge University Press, 1998), that covers a lot of the material. Other books on functional data structures have been published since we began writing this book, and more are beginning to appear.

Another feature of this book is that, as well as some firm favourites, it describes a number of algorithms that do not usually appear in books on algorithm design. Some of these algorithms have been adapted, elaborated, and simplified from yet another book published by Cambridge University Press: *Pearls of Functional Algorithm Design* (2010). The reason for this novelty is simply to make the book entertaining as well as instructive. Books on algorithm design are read, broadly speaking, by three kinds of people: academics who need reference material, undergraduate or graduate students on a course, and professional programmers simply for interest and enjoyment. Most professional programmers do not design algorithms but just take them from a library. Yet they too are a target audience for this book, because sometimes professional programmers want to know more about what goes into a good algorithm and how to think about them.

Algorithms in real life are a good deal more intricate than the ones presented in this book. The shortest-path algorithm in a satellite navigation system is a good deal more complicated than a shortest-path algorithm as presented in a textbook on algorithm design. Real-life algorithms have to cope with the problems of scale, with the effective use of a computer’s hardware, with user interfaces, and with many other things that go into a well-designed and useful product. None of these aspects is covered in the present book, nor indeed in most books devoted solely to the principles of algorithm design.

There is another feature of this book that deserves mention: all exercises are answered, if sometimes somewhat briefly. The exercises form an integral part of the text, and the questions and answers should be read even if the exercises are not attempted. Rather than have a complete bibliography at the end of the book, each chapter ends with references to (some of) the books and articles pertinent to the chapter.

Most of the major programs in this book are available on the web site

[www.cs.ox.ac.uk/publications/books/adwh](http://www.cs.ox.ac.uk/publications/books/adwh)

You can also use this site to see a list of all known errors, as well as report new ones. We also welcome suggestions for improvement, including ideas for new exercises.

### Acknowledgements

Preparation of this book has benefited enormously from careful reading by Sue Gibbons, Hsiang-Shang Ko, and Nicolas Wu. The manuscript was prepared using the `lhs2TeX` system of Ralf Hinze and Andres Löh, which pretty-prints the Haskell code and also allows it to be extracted and type-checked. The extracted code was then tested using the wonderful QuickCheck tool developed by Koen Claessen and John Hughes. Type-checking and QuickChecking the code has saved us from many infelicities; any errors that remain are, of course, our own responsibility.

We also thank David Tranah and the team at Cambridge University Press for their advice and hard work in the generation of the final version of the text.

Richard Bird  
Jeremy Gibbons