# 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.

# ALGORITHM DESIGN WITH HASKELL

RICHARD BIRD
*University of Oxford*

JEREMY GIBBONS
*University of Oxford*

CAMBRIDGE
UNIVERSITY PRESS

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
Information on this title: www.cambridge.org/9781108491617
DOI: 10.1017/9781108869041



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

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

# Contents

# 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 *mktrees* 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
```

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