HZ BOOKS
华章教育

计 算 机 科 学 丛 书

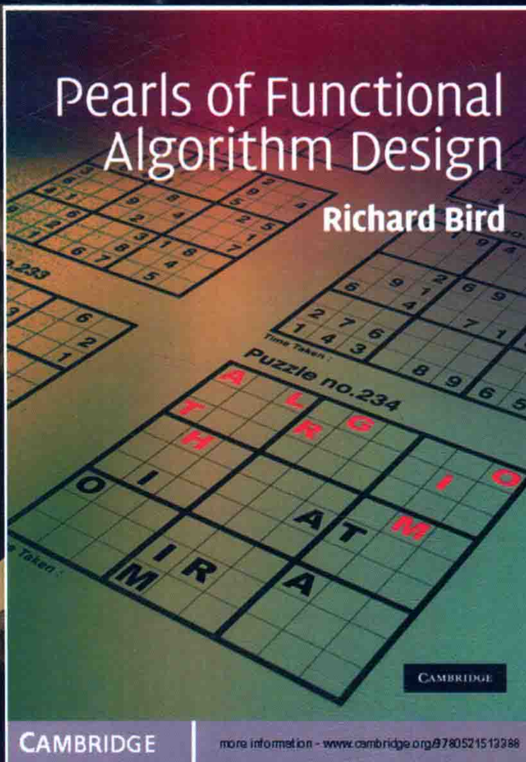
CAMBRIDGE

函数式算法设计珠玑

[英] 理查德·伯德（Richard Bird） 著

苏统华 孙芳媛 郝文超 徐琴 译

Pearls of Functional Algorithm Design



机械工业出版社
China Machine Press

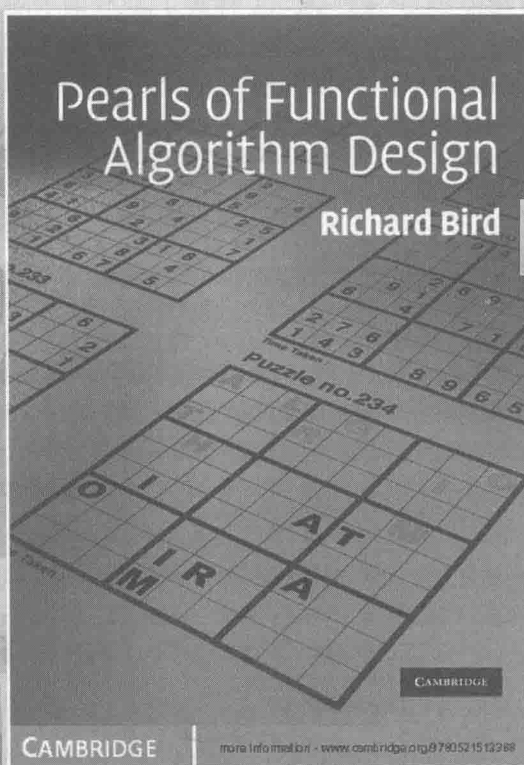
计算机科学丛书

函数式算法设计珠玑

[英] 理查德·伯德（Richard Bird）著

苏统华 孙芳媛 郝文超 徐琴 译

Pearls of Functional Algorithm Design



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

函数式算法设计珠玑 /（英）理查德·伯德（Richard Bird）著；苏统华等译．—北京：机械工业出版社，2017.3

（计算机科学丛书）

书名原文：Pearls of Functional Algorithm Design

ISBN 978-7-111-56251-1

I. 函… II. ①理… ②苏… III. 程序语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2017）第 044949 号

本书版权登记号：图字：01-2016-3483

This is a Chinese simplified edition of the following title published by Cambridge University Press:

Richard Bird: Pearls of Functional Algorithm Design (ISBN 978-0-521-51338-8).

© Cambridge University Press 2010.

This Chinese simplified edition for the People's Republic of China (excluding Hong Kong, Macau and Taiwan) is published by arrangement with the Press Syndicate of the University of Cambridge, Cambridge, United Kingdom.

© Cambridge University Press and China Machine Press in 2017.

This Chinese simplified edition is authorized for sale in the People's Republic of China (excluding Hong Kong, Macau and Taiwan) only. Unauthorized export of this simplified Chinese is a violation of the Copyright Act. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of Cambridge University Press and China Machine Press.

本书原版由剑桥大学出版社出版。

本书简体字中文版由剑桥大学出版社与机械工业出版社合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

此版本仅限在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）销售。

本书采用完全崭新的方式介绍算法设计。全书由 30 个珠玑构成，每个珠玑单独列为一章，用于解决一个特定编程问题。这些问题的出处五花八门，有的来自游戏或拼图，有的是有趣的组合任务，还有的是散落于数据压缩及字符串匹配领域的更为熟悉的算法。每个珠玑以使用函数式编程语言 Haskell 对问题进行描述作为开始，每个解答均是诉诸函数式编程法则从问题表述中计算得到。

本书适用于那些喜欢学习算法设计思想的函数式编程人员、学生和老师，同样适用于那些期望以数学推理方式处理程序的人员。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：迟振春　　责任校对：李秋荣

印　刷：北京市荣盛彩色印刷有限公司　　版　次：2017 年 4 月第 1 版第 1 次印刷

开　本：185mm×260mm 1/16　　印　张：14.5

书　号：ISBN 978-7-111-56251-1　　定　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brian W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www. hzbook. com
电子邮件：hzjsj@hzbook. com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037

华章科技图书出版中心

译者序

当前的算法设计主要涵盖的是命令式编程，对函数式编程言之甚少。从知识的发展角度看，这是极其危险的。我们承接这本书翻译任务的初衷，就是为了把本书关于函数式程序设计的深邃思想介绍给国人，同时也希望读者能够构建出更为完整的算法设计知识体系。

多年来，函数式编程一直没有得到应有的重视和普及。这种不幸可能根源于早期研究者没能使用简单易懂、务实有效的方式解释抽象的数学概念。实际上，函数式程序具有命令式程序的全部计算能力。命令式程序基于图灵机模型（更具体点是冯·诺伊曼模型），函数式程序则基于λ演算，两者在数学意义上具有能力等效性。对于冯·诺伊曼架构的计算机的一些不足，函数式编程思想具有天然的优势。我们不断看到最近的一些分布式计算框架常常引入函数式程序的一些特性，甚至它的很多思想正在不断融入如 Java、C♯或 C++ 等命令式语言中。John Backus 在其 1977 年的图灵奖演讲中就具有先见之明地指出，函数式程序设计思想是解决冯·诺伊曼模型计算瓶颈的替代方案。这一趋势已经发生。同时函数式编程语言也在不断迭代更新中，我们可以在越来越多的工程应用中看到它的威力（请参考 Haskell in industry）。更可喜的是，国内某些公司也有一些项目组一直在实际产品中使用 Haskell。

本书的内容取材自作者近三十年来的研究成果和深刻思考，每一章围绕一个经典问题，如同在讲述一个引人入胜的故事，不断扫清迷雾，找出事物的本质。每个珠玑在处理方式上，都首先诉诸 Haskell，对问题进行正确的表述，然后继续做一些数学推理，计算出更有效的解法。这种由粗到细不断深化的思想非常具有启发性，是难得的珍珠！

本书作者 Bird 是牛津大学计算机科学系教授，并担任过系主任一职，也是牛津林肯学院的兼职研究员。Bird 长期从事函数式程序设计的研究和教学工作，在代数、算法、Haskell 语言等方面均有建树，深受学界敬爱。其中 Bird-Meertens 形式体系（Bird-Meertens Formalism）就是他和合作者提出的。Bird 也撰写了多本专著和教材，颇受读者推崇。

本书是函数式编程书架上必备的参考书，通过本书的学习，相信会提高你的函数式编程功力。本书可以用作大学教材，也适合程序员在实践过程中参考。本书需要读者具有一定的 Haskell 编程经验，本书作者撰写的《Haskell 函数式程序设计》（已由机械工业出版社引进出版，ISBN 978-7-111-52932-3，定价 69.00 元），可以用来补足 Haskell 方面的欠缺。

这本书很薄，但耗费了我们一年零九个月的时间才得以翻译完成。翻译过程也是深入学习和品味函数式编程之美的过程。希望读者在研读本书时，也可以细细品味，静静思

试读结束：需要全本请在线购买： www.ertongbook.com

考，勤于练习，必然能够不断精进。由于本书的很多术语较新，对应的中文资料较少，再加上一些术语含义深邃，有的甚至是新造的英文单词，翻译难度很大。限于译者水平，翻译中难免存在不当之处，欢迎读者批评指正。

感谢机械工业出版社的姚蕾编辑在整个翻译过程中精心的组织和及时的帮助。

苏统华

哈尔滨工业大学软件学院

2017 年 2 月

前　言

Pearls of Functional Algorithm Design

1990年，《函数式编程期刊》（Journal of Functional Programming，JFP）正处于筹划阶段。我受到两位编辑 Simon Peyton Jones 和 Philip Wadler 之邀，定期撰写名为“函数式珠玑”（Functional Pearls）的专栏。他们内心的想法是模仿 Jon Bentley 曾经在 20 世纪 80 年代所撰写的“编程珠玑”（Programming Pearls）连载，这些珠玑为《ACM 通讯》（Communications of the ACM）期刊所写，获得了极大的成功。Bentley 在他的珠玑中写道：

> 正像天然的珍珠产生自刺激了牡蛎的砂砾，编程珠玑产生自折磨了程序员的实际问题。这些程序充满趣味，同时教给我们重要的编程技巧和基本的设计原理。

两位编辑为什么会选择我来承担这项工作呢？我觉得应该是我当时正对与此相关的特定任务感兴趣。这些任务先使用清晰却低效的函数式程序进行问题的表述，然后使用数学推理进一步计算出更高效的程序。20 世纪 90 年代，对函数式编程语言的关注不断增加，原因之一在于这些语言很适合进行数学推理。实际上，函数式编程语言 GOFER（全称为 GOod For Equational Reasoning）由 Mark Jones 发明，正如它的首字母缩略词所表达的那样，擅长数学推理。GOFER 是推动 Haskell 发展的语言之一，后者正是本书使用的语言。数学推理是本书的主导主题。

在最近 20 年里，大约有 80 个珠玑发表在 JFP 上，另外有少量珠玑出现在函数式编程国际会议（International Conference of Functional Programming，ICFP）和程序构造数学会议（Mathematics of Program Construction Conference，MPC）上。我大概撰写了其中的四分之一，更多的是由其他研究者撰写的。这些珠玑的主题包括有趣的程序计算、新颖的数据结构和为特殊应用而基于 Haskell 和 ML 构建的小而妙的特殊领域语言。

我的研究兴趣一直是算法和算法设计，因此本书的书名是函数式算法设计珠玑而不是函数式珠玑。很多珠玑以 Haskell 表述作为开始，继而通过计算得出一个更高效的版本。在写作这些珠玑时，我的目的是看一看算法设计可以在多大程度上沿袭我们熟悉的数学传统：通过已有的数学原理、定理和法则计算出结果。数学中的计算通常是为了对复杂的事物进行简化，而在算法设计中，它表现为另一种形态：把简易却低效的程序转化为完全不透明的高效的版本。珠玑所指的并非是最终的程序，而是指产生这一结果的计算。剩下的珠玑致力于为有趣且巧妙的算法提供简单易懂的解释，其中的一部分珠玑可能涉及不多的计算。从函数式角度解释算法背后的思想要比从过程式角度解释简单得多：函数式程序中作为构建块的函数可以非常容易地分离出来，这些函数很简短，但刻画计算模式的能力很强大。

本书中的一些珠玑虽然已经在 JFP 或者其他地方出版过，但这里对它们进行了精心打磨。实际上，很多珠玑已经跟原始版本大相径庭了。即使这样，它们肯定还有进一步打磨

和优化的空间。对于数学之美的黄金标准是 Aigner 和 Ziegler 撰写的《数学天书中的证明》(Proofs from The Book)(第 3 版，Springer 出版社，2003)，书中包含了一些数学定理的完美证明。我一直把该书当作目标，努力向它的标准看齐。

接近三分之一的珠玑是全新的。虽然本书的章节在一定意义上是按主题组织的，例如分治法、贪心算法、穷举搜索等，但除非明确指出，所有的珠玑可以按任何顺序阅读。珠玑中存在一些重复材料，多数与我们使用的库函数的属性有关，或者与更一般的法则(例如融合法则的多种叠加)有关。

最后，很多人为本书提供了素材。实际上，有几个珠玑最初是跟其他作者合写的。在此感谢我的合作者 Sharon Curtis、Jeremy Gibbons、Ralf Hinze、Geraint Jones 和 Shin-Cheng Mu，谢谢他们慷慨地允许我修订这些材料。Jeremy Gibbons 还阅读了最后阶段的草稿并提出了大量有助于提高行文质量的宝贵意见。有些珠玑也得到牛津大学编程代数研究组会议的仔细讨论。尽管很多瑕疵和错误已经消除，但是毫无疑问，新的瑕疵和错误也会被引入。除了上面提到的人员，还要感谢 Stephen Drape、Tom Harper、Daniel James、Jeffrey Lake、Meng Wang 和 Nicholas Wu，他们给出了很多正面意见，提高了文稿质量。我也要感谢 Lambert Meertens 和 Oege de Moor，与他们多年的合作带来了丰富的成果。最后，感谢剑桥大学出版社的编辑 David Tranah，他给予我鼓励和支持，包括在准备终稿时有用的技术建议。

Richard Bird

目　录

Pearls of Functional Algorithm Design

出版者的话
译者序
前言
第 1 章　最小未出现数 …… 1
第 2 章　优胜问题 …… 6
第 3 章　优化马鞍峰搜索算法 …… 10
第 4 章　一个选择问题 …… 17
第 5 章　排序成对的加和 …… 22
第 6 章　合成 100 …… 27
第 7 章　构建最小高度树 …… 34
第 8 章　拆分的贪心算法 …… 41
第 9 章　找出名人 …… 46
第 10 章　删除重复项 …… 52
第 11 章　最大非段和 …… 59
第 12 章　后缀排序问题 …… 64
第 13 章　Burrows-Wheeler 变换 …… 73
第 14 章　最末尾部 …… 82
第 15 章　所有的公共前缀 …… 90
第 16 章　Boyer-Moore 算法 …… 94
第 17 章　Knuth-Morris-Pratt 算法 …… 102
第 18 章　规划算法解决 Rush Hour 问题 …… 109
第 19 章　一个简单的数独求解机 …… 117
第 20 章　Countdown 问题 …… 124
第 21 章　hylomorphism 和 nexus …… 133
第 22 章　计算行列式的三种方法 …… 142
第 23 章　凸包 …… 148
第 24 章　有理数算术编码 …… 156
第 25 章　整数算术编码 …… 164
第 26 章　Schorr-Waite 算法 …… 175
第 27 章　有序插入 …… 183
第 28 章　无回路函数式算法 …… 192
第 29 章　Johnson-Trotter 算法 …… 199
第 30 章　蜘蛛纺丝问题完全解析 …… 205
索引 …… 218

最小未出现数

1.1 概述

假设有限集合 X 由自然数组成，我们本章考察的问题是计算不在 X 中的最小自然数。这个问题凝练自一个常见编程任务。其中的数字标识对象，而 X 代表正在使用的对象集合。任务的目标是发现某个未使用的对象，不妨说，那个拥有最小标识的对象。

显然，这个问题的解法取决于集合 X 如何表示。如果集合 X 是一个没有重复元素且元素递增排列的列表，那么解法很简单：只需找到第一个未出现的自然数。但如果集合 X 是由一组无序、互异数字组成的列表，比如说

[08，23，09，00，12，11，01，10，13，07，41，04，14，21，05，17，03，19，02，06]

如何才能找到不在这个列表的最小自然数呢？

我们无法立刻判定这个问题存在线性时间的解法(linear-time solution)。毕竟，我们无法在线性时间内完成对由自然数组成的任意列表的排序。但实际上，线性时间复杂度的解法确实存在，本章珠玑将介绍其中的两种解法：一种是基于 Haskell 数组，另一种是基于分治法(divide and conquer)。

1.2 数组解法

这个问题可以描述为函数 *minfree*，其定义如下：

$$\begin{array}{lcl} minfree & :: & [Nat] \rightarrow Nat \\ minfree\ xs & = & head\ ([0\,..] \setminus\setminus xs) \end{array}$$

表达式 $us \setminus\setminus vs$ 表示从 us 中删去出现在 vs 中的元素所幸存的列表：

$$\begin{array}{lcl} (\setminus\setminus) & :: & Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \\ us \setminus\setminus vs & = & filter\ (\notin vs)\ us \end{array}$$

函数 *minfree* 是可以得到正确结果的，但是一个长度为 n 的列表最坏情况下需要执行 $\Theta(n^2)$ 步。例如，求 $minfree[n-1, n-2..0]$，需要对区间 $0 \leqslant i \leqslant n$ 中每个 i 判断 $i \notin [n-1, n-2..0]$，因此测试次数为 $n(n+1)/2$ 步。

不论是数组解法还是分治解法，都是基于如下关键事实：并不是每个在区间 $[0..length\ xs]$ 里面的数字都在 xs 里。因此不在 xs 里面的最小数就是不在 $filter(\leqslant n)\ xs$(这里 $n = length\ xs$)里面的最小数。数组解法的程序利用这个事实对出现在函数 $filter(\leqslant n)\ xs$ 里面的数字建立一个检查列表(checklist)。这个检查列表是编号从 0 到 n、长度为 $n+1$、

全部初始化为 *False* 的布尔型数组。对于每个在 xs 中的 x，如果满足 $x \leqslant n$，我们将数组 x 位置的元素设为 *True*。然后在数组中第一个值为 *False* 的位置就能发现最小未出现数(smallest free number)。因此，$minfree = search \cdot checklist$，其中

```
search  ::  Array Int Bool → Int
search  =   length · takeWhile id · elems
```

函数 *search* 接受一个布尔型数组，将这个数组转化成一个布尔型列表，然后返回从起点开始由 *True* 组成的最长片段的长度，这个数就是第一个为 *False* 的元素的位置。

为了在线性时间内实现 *checklist*，一种方法是调用 Haskell 的 *accumArray* 函数(在 *Data. Array* 库中)。这个函数可以使用任何数据类型：

$$Ix\ i \Rightarrow (e \to v \to e) \to e \to (i,i) \to [(i,v)] \to Array\ i\ e$$

类型约束 $Ix\ i$ 限制用来标识数组索引或位置的 i 必须是 Index 类型，比如 *Int* 型或 *Char* 型。这个函数的第一个参数是可以将数组元素和值转换成新元素的累加函数(accumulating function)，第二个参数是每个索引的初始元素，第三个参数是一对标识索引下界和上界的值，第四个是由索引与值对构成的关联列表。函数 *accumArray* 从左到右地处理关联列表，调用累加函数将元素与值转化成新元素来建立数组。假设累加函数消耗的时间为常数，那么这个处理的耗时与关联列表的长度呈线性关系。

checklist 函数定义为 *accumArray* 的一个实例：

```
checklist     ::  [Int] → Array Int Bool
checklist xs  =   accumArray (∨) False (0, n)
                  (zip (filter (≤ n) xs) (repeat True))
                  where n = length xs
```

2

这种实现方法不要求 xs 的元素是互异的，但是要求元素必须是自然数。

值得一提的是，如果列表内所有数都在给定区间(0，n)内，*accumArray* 函数可以在线性时间内对它们排序。我们用 *countlist* 替代 *checklist*，其中

```
countlist     ::  [Int] → Array Int Int
countlist xs  =   accumArray (+) 0 (0, n) (zip xs (repeat 1))
```

然后进行 $sort\ xs = concat\ [replicate\ k\ x \mid (x,\ k) \leftarrow assocs(countlist\ xs)]$。事实上，如果我们用 *countlist* 而不是 *checklist*，那么就能通过查找第一个值为 0 的位置来实现 *minfree*。

上述实现的方法利用一个巧妙的库函数，一气呵成地建立了数组。一个实现 *checklist* 的更一般方法是通过一个常数时间的更新操作，一步步列举元素。只要所需的数组操作有合适的单子执行(monad)，比如状态单子(state monad)，就有可能用 Haskell 实现。接下来的 *checklist* 程序使用了 *Data. Array. ST* 库：

```
checklist xs  =  runSTArray (do
                   {a ← newArray (0, n) False;
                    sequence [writeArray a x True | x ← xs, x ≤ n];
                    return a})
                 where n = length xs
```

这个解法不会让纯函数式编程者(pure functional programmer)满意，因为它本质上是

试读结束：需要全本请在线购买： www.ertongbook.com

披着函数式程序外衣的过程式程序(procedural program)。

1.3 分治解法

现在我们回到这个问题的分治解法。它的基本思想是用 *minfree xs* 和 *minfree ys* 表示 *minfree*$(xs +\!\!+ ys)$。我们先给出 \\ 的属性:

$$
\begin{aligned}
(as +\!\!+ bs) \backslash\backslash cs &= (as \backslash\backslash cs) +\!\!+ (bs \backslash\backslash cs) \\
as \backslash\backslash (bs +\!\!+ cs) &= (as \backslash\backslash bs) \backslash\backslash cs \\
(as \backslash\backslash bs) \backslash\backslash cs &= (as \backslash\backslash cs) \backslash\backslash bs
\end{aligned}
$$

这些属性反映了与集合运算相似的法则。这里是把集合的并操作 $\cup$ 使用 $+\!\!+$ 代替、集合的差操作 \ 使用 \\ 代替。假设现在 as 和 vs 是不相交的(disjoint),那就意味着 $as \backslash\backslash vs = as$,同样也假设 bs 和 vs 不相交,那么 $bs \backslash\backslash us = bs$。下面的映射可以从属性 $+\!\!+$ 和 \\ 中推出:

$$(as +\!\!+ bs) \backslash\backslash (us +\!\!+ vs) = (as \backslash\backslash us) +\!\!+ (bs \backslash\backslash vs)$$

3

现在,选择任意一个自然数 b,令 $as=[0..b-1]$,以及 $bs=[b..]$。进一步令 $us=$ *filter*$(<b)\,xs$,$vs=$ *filter*$(\geqslant b)\,xs$。于是,as 和 vs 是不相交的,同理 bs 和 us 也是如此,因此有

$$
\begin{aligned}
[0\,..\,] \backslash\backslash xs &= ([0\,..\,b-1] \backslash\backslash us) +\!\!+ ([b\,..\,] \backslash\backslash vs) \\
&\quad \textbf{where}\ (us, vs) = \mathit{partition}\ (<b)\ xs
\end{aligned}
$$

Haskell 提供了一个高效实现的分治函数 *partition p* 的实现方法,该函数可以将一个列表分成满足 p 和不满足 p 的元素集合。由于

$$\mathit{head}\ (xs +\!\!+ ys) = \textbf{if}\ \mathit{null}\ xs\ \textbf{then}\ \mathit{head}\ ys\ \textbf{else}\ \mathit{head}\ xs$$

仍旧对任意选择的自然数 b,我们得到

$$
\begin{aligned}
\mathit{minfree}\ xs = \ &\textbf{if}\ \mathit{null}\ ([0\,..\,b-1] \backslash\backslash us) \\
&\textbf{then}\ \mathit{head}\ ([b\,..\,] \backslash\backslash vs) \\
&\textbf{else}\ \mathit{head}\ ([0\,..\,] \backslash\backslash us) \\
&\textbf{where}\ (us, vs) = \mathit{partition}\ (<b)\ xs
\end{aligned}
$$

如果通过直接计算来测试 *null*$([0..b-1] \backslash\backslash us)$,消耗时间将与 us 长度的平方有关。接下来的问题是:我们是否可以更加高效地实现这一测试呢?对于输入是由互异自然数构成的列表来说,回答是肯定的。这同样适用于 us。并且,us 中的每个元素都小于 b。此时,

$$\mathit{null}\ ([0\,..\,b-1] \backslash\backslash us) \equiv \mathit{length}\ us == b$$

请注意,基于数组的解法未假设给定的列表不包含重复元素,但是要得到高效的分治解法,这一假设是至关重要的。

对于上述 *minfree* 代码的进一步观察,建议我们应该将 *minfree* 推广为一个函数,比如说 *minfrom*,定义成

$$
\begin{array}{lll}
minfrom & :: & Nat \to [Nat] \to Nat \\
minfrom\ a\ xs & = & head\ ([a\ ..] \setminus\setminus xs)
\end{array}
$$

其中 xs 的每一个元素都被假设为大于或等于 a。然后选择 b 使得 $length\ us$ 和 $length\ vs$ 的长度都比 $length\ xs$ 的长度短，那么下面的 $minfree$ 递归定义就是可行的：

$$
\begin{array}{llll}
minfree\ xs & = & minfrom\ 0\ xs & \\
minfrom\ a\ xs & \mid & null\ xs & = a \\
 & \mid & length\ us == b - a & = minfrom\ b\ vs \\
 & \mid & \mathbf{otherwise} & = minfrom\ a\ us \\
 & & \mathbf{where}\ (us, vs) = partition\ (< b)\ xs &
\end{array}
$$

[4]

接下来要确定 b 的值。很显然，我们希望 $b>a$。同时，我们也希望所选择的 b 使得 us 和 vs 的长度的较大值尽可能小。选择 b 以满足这些要求的正确做法是

$$b \;=\; a + 1 + n \operatorname{div} 2$$

这里 $n = length\ xs$。当 $n \neq 0$ 并且 $length\ us < b - a$ 时，有

$$length\ us \leq n \operatorname{div} 2 < n$$

而当 $length\ us = b - a$ 时，有

$$length\ vs = n - n \operatorname{div} 2 - 1 \leq n \operatorname{div} 2$$

这样选择的话，执行 $minfrom\ 0\ xs$ 的步数 $T(n)$ 在 $n = length\ xs$ 时满足 $T(n) = T(n \operatorname{div} 2) + \Theta(n)$，可以得出 $T(n) = \Theta(n)$。

作为最终的一个优化，我们能够通过对数据进行简单的改良，即将 xs 用一对数值 $(length\ xs,\ xs)$ 表示，避免重复地计算 $length$。所得到的最终程序如下：

$$
\begin{array}{llll}
minfree\ xs & = & minfrom\ 0\ (length\ xs, xs) & \\
minfrom\ a\ (n, xs) & \mid & n == 0 & = a \\
 & \mid & m == b - a & = minfrom\ b\ (n - m, vs) \\
 & \mid & \mathbf{otherwise} & = minfrom\ a\ (m, us) \\
 & & \mathbf{where}\ (us, vs) & = partition\ (< b)\ xs \\
 & & \qquad\quad b & = a + 1 + n \operatorname{div} 2 \\
 & & \qquad\quad m & = length\ us
\end{array}
$$

结果表明，上述程序大约比基于递增数组的程序快两倍，也比使用 $accumArray$ 大约快 20%。

1.4 本章小结

本章的问题比较简单，至少有两种简洁解法。第二种解法基于分治思想，这是一种常用的算法设计技术。将一个列表划分成一个小于给定值的元素子表和一个剩余子表，这一思想经常在一些算法中出现，其中最著名的是快排算法(Quicksort)。当要为包含 n 个元素的列表寻找时间复杂度为 $\Theta(n)$ 的算法时，比较倾向于马上寻找一种能在常量时间内或者至少在平摊意义的常量时间内处理列表中每个元素的方法。但是一个为了将原问题减少到

[5]

至少原来的一半规模而执行 $\Theta(n)$ 步处理的递归程序也是很好的解法。

纯函数式算法设计者和过程式设计者的一个不同点在于，前者并不假设存在可以在常量时间内完成更新操作的数组，至少在没有深入探索的情况下是不假设的。对于一个纯函数式的编程者，一个更新操作消耗的时间是与数组大小成对数关系的[⊖]。这就解释了为什么有时看起来一个问题的最好的函数式解法和过程式解法会有对数级别的差距。但是有时，恰如本章，在经过更仔细的检查后，这个差距会消失。

6

⊖ 公平地讲，过程式编程者也认识到，常量时间的索引和更新操作只有在数组很小时才可能实现。

第 2 章

优胜问题

2.1 概述

本章珠玑我们完成来自 Martin Rem(1998a)的一个编程小练习。Rem 的解决方案使用了二分法搜索，我们的解决方案是分治法的另外一种应用。根据定义，一个数组中某个元素的*优胜元素*是位于其右侧的一个更大的元素，所以当满足 $i<j$ 且 $x[i]<x[j]$时，$x[j]$是$x[i]$的一个优胜元素。一个元素的*优胜元素数*是它优胜元素的数量。比如说，字符串 GENERATING 中各个字母的优胜元素数如下所示：

G	E	N	E	R	A	T	I	N	G
5	6	2	5	1	4	0	1	0	0

优胜元素数的最大值是 6。第一个出现的字母 E 有 6 个优胜元素，分别是 N，R，T，I，N 和 G。Rem 问题是使用一个复杂度为 $O(n \log n)$的算法计算一个长度 $n>1$ 的数组中优胜元素数的最大值。

2.2 说明

假设给出的输入值是一个列表而不是一个数组。函数 *msc*(最大优胜元素数的简称)的详细说明如下：

$$
\begin{aligned}
&msc && :: && Ord\ a \Rightarrow [a] \rightarrow Int \\
&msc\ xs && = && maximum\ [scount\ z\ zs \mid z : zs \leftarrow tails\ xs] \\
&scount\ x\ xs && = && length\ (filter\ (x <)\ xs)
\end{aligned}
$$

scount x xs 的值是在列表 *xs* 中 *x* 的优胜数，*tails* 以长度的降序返回一个非空列表的非空尾部[⊖]：

$$
\begin{aligned}
&tails\ [\,] && = && [\,] \\
&tails\ (x : xs) && = && (x : xs) : tails\ xs
\end{aligned}
$$

从定义来看 *msc* 是可执行的，但需要平方时间。

2.3 分治解法

面对复杂度为 $O(n \log n)$的目标，将解决方案趋向于分治算法似乎是合理的。如果我

⊖ 与同名的标准 Haskell 函数不同，返回可能为空列表的可能空尾部。

们可以找到一个函数 $join$ 使得

$$msc\ (xs +\!\!+ ys) \quad = \quad join\ (msc\ xs)\ (msc\ ys)$$

而且函数 $join$ 可以在线性时间内被计算出来，那么在一个长度为 n 的列表中，用于计算 msc 函数的分治算法的时间复杂度 $T(n)$ 满足 $T(n)=2T(n/2)+O(n)$，使用了解决方案 $T(n)=O(n \log n)$。但很明显这样的函数 $join$ 不可能存在：单个数字 $msc\ xs$ 为任何这样的分解提供的信息太少。

最小限度的扩展从所有的优胜元素计数的表开始：

$$table\ xs \quad = \quad [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ xs]$$

接下来有 $msc = maximum \cdot map\ snd \cdot table$。我们可以找到一个在线性时间内完成的函数 $join$ 使之满足以下条件吗?

$$table\ (xs +\!\!+ ys) \quad = \quad join\ (table\ xs)\ (table\ ys)$$

好的。让我们看看。我们将需要具有以下分治特性的 $tails$：

$$tails\ (xs +\!\!+ ys) \quad = \quad map\ (+\!\!+ ys)\ (tails\ xs) +\!\!+ tails\ ys$$

计算过程如下：

$$\begin{aligned}
& table\ (xs +\!\!+ ys) \\
= \quad & \{\text{定义}\} \\
& [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ (xs +\!\!+ ys)] \\
= \quad & \{tails\ \text{的分治特性}\} \\
& [(z, scount\ z\ zs) \mid z : zs \leftarrow map\ (+\!\!+ ys)\ (tails\ xs) +\!\!+ tails\ ys] \\
= \quad & \{\text{分布} \leftarrow \text{在} +\!\!+ \text{上}\} \\
& [(z, scount\ z\ (zs +\!\!+ ys)) \mid z : zs \leftarrow tails\ xs] +\!\!+ \\
& [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ ys]) \\
= \quad & \{\text{由于}\ scount\ z\ (zs +\!\!+ ys) = scount\ z\ zs + scount\ z\ ys\} \\
& [(z, scount\ z\ zs + scount\ z\ ys) \mid z : zs \leftarrow tails\ xs] +\!\!+ \\
& [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ ys]) \\
= \quad & \{table\ \text{的定义且}\ ys = map\ fst\ (table\ ys)\} \\
& [(z, c + scount\ z\ (map\ fst\ (table\ ys))) \mid (z, c) \leftarrow table\ xs] +\!\!+ table\ ys
\end{aligned}$$

因此函数 $join$ 可以通过以下表达式定义：

8

$$\begin{aligned}
join\ txs\ tys \quad &= \quad [(z, c + tcount\ z\ tys) \mid (z, c) \leftarrow txs] +\!\!+ tys \\
tcount\ z\ tys \quad &= \quad scount\ z\ (map\ fst\ tys)
\end{aligned}$$

然而，这个定义的问题在于 $join\ txs\ tys$ 在长度 txs 和 tys 内消耗的并不是线性时间。

如果在第一个部分中 tys 是以升序排列的话，我们可以改善 $tcount$ 的计算。然后我们可以推出：

$$\begin{aligned}
& tcount\ z\ tys \\
= \quad & \{tcount\ \text{和}\ scount\ \text{的定义}\}
\end{aligned}$$

$$
\begin{aligned}
& length\ (filter\ (z <)\ (map\ fst\ tys)) \\
= & \quad \{\text{由于}\ filter\ p \cdot map\ f = map\ f \cdot filter\ (p \cdot f)\} \\
& length\ (map\ fst\ (filter\ ((z <) \cdot fst)\ tys)) \\
= & \quad \{\text{由于}\ length \cdot map\ f = length\} \\
& length\ (filter\ ((z <) \cdot fst)\ tys) \\
= & \quad \{\text{由于}\ tys\ \text{在第一个参数上是已排序的}\} \\
& length\ (dropWhile\ ((z \geq) \cdot fst)\ tys)
\end{aligned}
$$

因此

$$tcount\ z\ tys \ = \ length\ (dropWhile((z \geq) \cdot fst)\ tys) \tag{2.1}$$

这个计算表明在第一个部分中以升序维护 $table$ 是明智的：

$$table\ xs \ = \ sort\ [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ xs]$$

重复上述计算，但对于 $table$ 已排序的版本，我们提出

$$join\ txs\ tys \ = \ [(x, c + tcount\ x\ tys) \mid (x, c) \leftarrow txs] \wedge\!\!\wedge\ tys \tag{2.2}$$

这里$\wedge\!\!\wedge$合并两个已排序的列表。现在我们可以使用这个特性得出一个更高效的函数 $join$ 的递归定义。其中一个基本例子非常直接，即 $join[\,]tys = tys$。另外一个紧随其后的基本的例子是 $join\ txs[\,] = txs$，因为 $tcount\ x[\,] = 0$。对于递归的例子我们通过比较 x 和 y 简化一下(在 Haskell 中，@符号引入了一个同义词，所以 txs' 是(x, c)：txs' 的一个同义词，类似于 tys)：

$$join\ \ txs@((x, c) : txs')\ \ tys@((y, d) : tys') \tag{2.3}$$

使用(2.2)和(2.3)简化

9

$$((x, c + tcount\ x\ tys) : [(x, c + tcount\ x\ tys) \mid (x, c) \leftarrow txs']) \wedge\!\!\wedge\ tys$$

为了探寻哪个元素是通过$\wedge\!\!\wedge$首先产生的，我们需要比较 x 和 y。如果 $x < y$，那么是左边的元素，因为通过(2.1)我们有 $tcount\ x\ tys = length\ tys$，表达式(2.3)简化成

$$(x, c + length\ tys) : join\ txs'\ tys$$

如果 $x = y$，我们需要比较 $c + tcount\ x\ tys$ 和 d。但是通过 $table$ 的定义和(2.1)的 $tcount\ x\ tys = tcount\ x\ tys'$，我们得出 $d = tcount\ x\ tys'$，所以(2.3)简化成(y, d)：$join\ txs\ tys'$。这也就是在最终的情况 $x > y$ 下的结果。

将这些结果放在一起，为了避免重复地对长度进行计算，将 $length\ tys$ 作为一个额外的参数传递给函数 $join$，我们得出 $table$ 的以下分治算法：

$$
\begin{aligned}
table\ [x] \quad & = \ [(x, 0)] \\
table\ xs \quad & = \ join\ (m - n)\ (table\ ys)\ (table\ zs) \\
& \qquad \textbf{where}\ m \ = \ length\ xs \\
& \qquad\qquad\ \ n \ = \ m\ \mathrm{div}\ 2 \\
& \qquad\qquad\ \ (ys, zs) \ = \ splitAt\ n\ xs
\end{aligned}
$$

试读结束：需要全本请在线购买： www.ertongbook.com