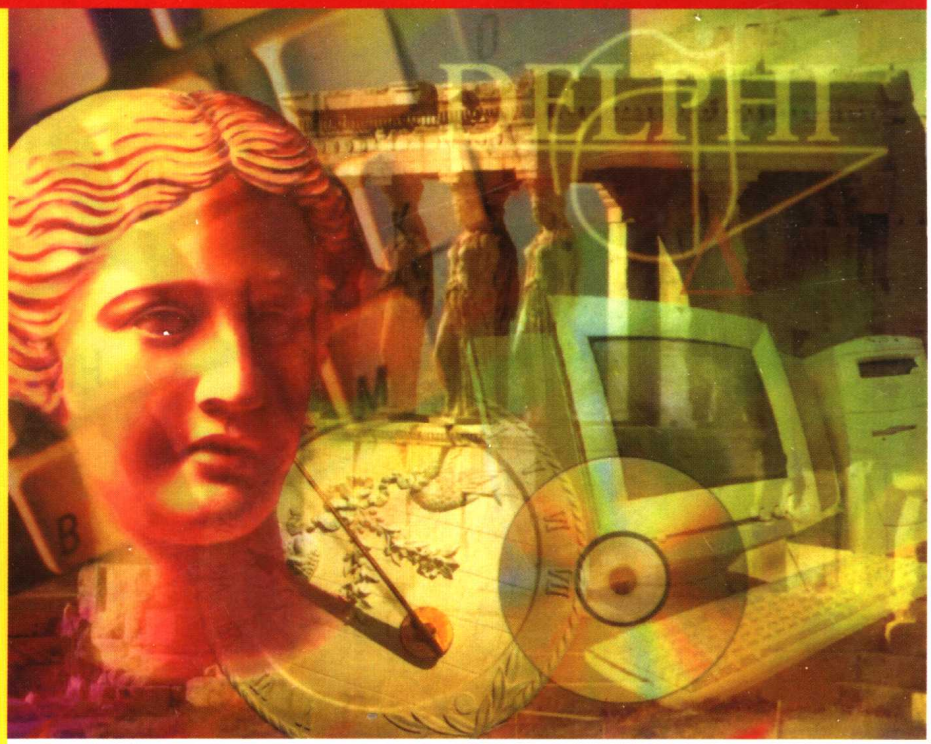
Wordware Delphi Developer's Library

使用特定于 Delphi 的类编写标准算法从而优化你的应用

本书以及附带光盘中的示例代码可运行于所有版本的 Delphi 以及 Kylix



The Tomes of Delphi Algorithms and Data Structures

Delphi 算法与数据结构

[美] Julian Bucknall 著
林琪 朱涛江 译

WORDWARE Publishing, Inc.

中国电力出版社
www.infopower.com.cn

Wordware Delphi Developer's Library

The Tomes of Delphi Algorithms and Data Structures

Delphi算法与数据结构

[美]Julian Bucknall 著
林琪 朱涛江 译

中国电力出版社

内 容 提 要

本书从实用性的角度出发，以 Delphi 程序设计语言为载体，详细介绍了算法和数据结构。其中涵盖了诸如数组、链表和二叉树等内容，并着重介绍了查找算法、排序算法以及相关的优化技术。另外本书还对散列、散列表、优先队列、状态机和正则表达式以及诸如哈夫曼和 LZ77 等数据压缩技术进行了描述。

本书适用于 Delphi 程序员以及数据库开发人员。

The Tomes of Delphi Algorithms and Data Structures(ISBN 1-55622-736-1)
Julian Bucknall
Published by arrangement with Wordware Publishing, Inc.

图书在版编目（CIP）数据

Delphi 算法与数据结构 /（美）巴克纳尔（Bucknall, J.）著；林琪，朱涛江译. —北京：中国电力出版社，2003
（Delphi 技术系列）
ISBN 7-5083-1483-2
Ⅰ.D... Ⅱ.①巴...②林...③朱... Ⅲ.①软件工具－程序设计②电子计算机－计算方法③数据结构 Ⅳ.①TP311.56②TP311.12
中国版本图书馆 CIP 数据核字（2003）第 017929 号

责任编辑：姚贵胜
书　　名： Delphi算法与数据结构
编　　著： (美) Julian Bucknall
翻　　译： 林琪 朱涛江
出版发行： 中国电力出版社
地址：北京市三里河路6号 邮政编码：100044
电话：（010）88515918 传真：（010）88518169
印　　刷： 北京市铁成印刷厂印刷
开　　本： 787×1092 1/16　　**印　　张：** 27　　**字　　数：** 667千字
书　　号： 7-5083-1483-2
版　　次： 2003年8月北京第一版
印　　次： 2003年8月第一次印刷
定　　价： 55.00 元

前　言

你可能刚刚在书店里拿起这本书，也可能已经买回家正在翻阅，现在你所需要了解的大概不外乎以下几个问题……

为什么要写一本关于 Delphi 算法的书呢

尽管书店里有关算法的书可谓林林总总，但是通常仅涉猎标准计算机科学课程范围之内，而很少能够从实用的角度来研究算法。这些书中的代码只是描述了所讨论的算法，并没有对相关技术在实际生活中的具体应用给予更多考虑。从专业程序员的角度来看，其中许多书都只是大学院校相应课程所用的课本，一些很有意思的内容却往往留给读者自行练习，很少有答案，甚至根本没有。

当然，大部分此类书并不使用 Delphi、Kylix 或 Pascal。有一些采用伪代码描述，有些采用 C，有些则采用 C++，还有一些采用特定（du jour）语言；不过在最著名也是最常参考的算法书中则使用了一种根本不存在的汇编语言（如《The Art of Computer Programming》中所用的 MIX 汇编语言[11，12，13]——请参见“参考文献”部分）。这些书在其标题中也确实声称可“应用”于 C、C++乃至 Java。这有什么问题吗？毕竟，算法终归是算法；对于算法采用何种方式描述应该不成问题，这样理解难道不对吗？为什么还要费劲去购买和阅读一本基于 Delphi 的算法书呢？

对于 Delphi，我很自得地认为它在目前应用开发所用的诸多语言和环境中可谓独树一帜。首先，类似于 Visual Basic，Delphi 也是一种可以快速开发 16 位或 32 位 Windows 应用的环境，而使用 Kylix 则可以实现 Linux 应用的快速开发。仅需轻点鼠标，组件即可落于窗体之上。有些组件随后需要双击，再键入些许代码，这样组件之间就可以建立错综复杂而又紧密的关系。如果再加上事件处理程序，就有可能得到一个看上去很不错的半成品了。

其次，像 C++一样，Delphi 也比较接近于底层，可以很容易地访问不同的操作系统 API。有些情况下，Borland 公司会开发出访问 API 的单元，并连同 Delphi 本身一起销售；另外一些情况下，程序员会仔细分析 C 的头文件，从而尝试将其转换为 Delphi 的形式（可参见 *http://www.delphi-jedi.org* 的 Jedi 项目）。无论如何，Delphi 都可以充分利用其优势妥善地完成任务并实现 OS 子系统的管理。

Delphi 程序员将其本身划归为两大阵营：应用程序员和系统程序员。不过有时你也会发现有的程序员二者兼备。这两个阵营之间的联系就在于无论哪一类程序员都必须同算法世界打交道，同时对于算法也必须做到有一定了解。如果你有一定的编程经验，可能会遇到需要编写二分查找（折半查找）代码的情况。当然，在此之前，你可能需要实现某种排序使数据按照一定的顺序排列，从而正常地完成二分查找。最后，你还可能开始使用某种性能评测工

具（Profiler），也许会发现 TStringList 中存在的瓶颈，并希望了解哪一种数据结构能够更有效地完成这一任务。

作为程序员，算法即是我们工作的全部。初学者总是很害怕规范的算法，我的意思是说，在习惯于此之前，甚至这个词（algorithm）本身好像都很难拼写。不过可以这样来考虑：程序可定义为一种从用户获取信息的算法，并为其产生某种输出。

历经计算机科学家们的努力，标准算法得到了充分的发展和完善，这才使诸如你我之辈在编程时可以“享用”到这些算法。掌握基本算法不仅可以使你的编程技艺得到充分发挥，并且还可以使你不为选用的语言所左右。例如，如果你了解散列表，这包括其优缺点、用途以及如此使用的原因等等，另外还得到了可以立即投入使用的一种具体实现，那么对于你目前所开发的子系统或应用而言，你对它的设计将有一个全新的认识，而且会发现某些地方利用散列表应该更为有利。如果对于排序你不感觉发怵，而且知道它是如何工作的，此外对于何时使用选择排序而不是快速排序也了如指掌，那么你很可能会在应用中自行编写相关的排序代码，而不会借助于某个标准的 Delphi 控件来满足要求（例如，我就记得曾听说过一个“耸人听闻”的故事：有人曾使用一个隐藏的 TListBox 控件，并在其中加入了一大堆的串，然后将控件的 Sorted 属性置为 true，力图用这种方法来使这些串做到有序）。

也许你会说：“好吧，写算法固然不错，但为什么非要用 Delphi 或 Kylix 呢？”

> 顺便说一句，在此先来做一个约定；否则我将不得不写上大量的“Delphi 或 Kylix”。后面我在提到“Delphi”时，实际上指的就是“Delphi 或 Kylix”。毕竟，Kylix 的早期版本即被认为是面向 Linux 的“Delphi”。因此在这本书中，“Delphi”就是指面向 Windows 的 Delphi 以及面向 Linux 的 Kylix。

下面来看为什么要用 Delphi？其原因有二：Object Pascal 语言和操作系统。Delphi 的语言中有许多构造在其他语言中均没有，利用构造将使高效的算法和数据结构可以更容易也更自然地得以封装。例如属性即属此类，再如若出现不可预知的错误时，相应的异常也属构造。尽管在 Delphi 中不用这些 Delphi 专用的语言构造也完全可以编写出标准算法，但我认为，如此一来我们将无法感受到这种语言的效率和魅力所在。在本书中，我们将特意大量使用 Delphi 中的 Object Pascal 语言，在此我没有考虑拿到此书的 Java 程序员在转换代码时可能存在的困难。既然封面上标明 Delphi，那么我们就一心一意地使用 Delphi 吧。

其次要考虑的是，传统意义上对算法的认识均体现在通用性上，至少从 CPU 和操作系统的角度来看需要如此。这些算法当然可以针对 Windows 环境得到优化，也可以面向 Linux 进行改进。对于我们所用的各种类型的 Pentium 处理器、各种不同的内存缓存器、OS 中不同的虚存子系统等等，算法还可以做到更为高效。这本书将特别关注于在效率上所获得的收益。不过，我们不至于什么代码都拿汇编语言来编写，尽管它对于当前处理器的管道式体系结构来说应该是最优化的，但有些地方我还是必须明确其使用要有一定限制！

因此，无论怎样，广大 Delphi 群体确实需要一本算法方面的书，而且迫切需要它完全针对于特定的语言、操作系统和处理器，而本书正是应此需而生。它不是由面向其他语言的其他书翻译得来的。不仅这本书本身是从头编写的，而且此书的作者可谓每日均与 Delphi “并肩作战”，他以编写软件库为生，对于开发商业运行例程、类和工具的复杂性可算是轻车熟路。

我需要了解什么呢

这本书并不是要教你学习 Delphi 编程。你需要首先了解 Delphi 程序设计的基础知识，例如创建新的工程、如何编写代码、完成编译和调试等等。在此提醒一句：这本书里不会谈到控件。你必须对于类、过程和方法引用、无类型指针、强大的 TList 以及封装为 Delphi 的 TStream 系列的流相当熟悉。此外，还需要对诸如封装、继承、多态和委托等面向对象的概念有足够的理解。最后，应该不会对 Delphi 中的对象模型感到陌生或害怕！

前面已经提到，这本书中所描述的许多概念都相当简单。学习编程的新手会从本书中学到有关标准算法和数据结构的一些基本内容。实际上，分析源代码可以帮助这些初级程序员掌握到高级程序员的许多技巧和方法。而更高级的结构则留待你在特别需要的时候再来学习。

因此本书基本上要求你有一定的 Delphi 编写经验。在编程时，你有时可能会发现 TList 及相关的一组类型不足以满足需要，而希望有其他类型的数据结构，但是又不太清楚哪些数据结构可用，或者是即使找到了某种结构又不知如何使用。有的情况下，你可能需要一个简单排序例程，但所找到的参考书采用的编码语言却为 C++，而说实话你宁可从头编起也不想由此 C++代码进行转换。还有些情况，你可能还希望看到一本算法书，其中将把性能和效率与算法本身的描述提到同一个高度。那么，这本书正是你所需要的。

需要何种版本的 Delphi 呢

准备好了吗？请不要感到奇怪，这个问题的答案是所有版本。除了在第 2 章讨论动态数组时需要使用 Delphi 4 及以上版本和 Kylix，另外第 12 章中部分内容和偶而零星处对版本有要求以外，这里的代码可以在任何版本的 Delphi 中进行编译和运行。除了前面我提到的少量特定于版本的代码之外，本书中的所有其他代码都曾在各种版本的 Delphi 和 Kylix 中测试通过。

因此你可以认为这本书里所印的所有代码均适用于各种版本的 Delphi。尽管有些代码清单要基于特定版本，不过均已明确指出。

我将看到哪些内容，它们将如何安排

这本书分为 12 章，并包括一个参考文献部分。

第 1 章中列出了一些基本的规则。首先从性能的讨论开始，我们将了解到算法效率的度量，在此将先介绍大 O 符号的含义，再对算法的实际运行时间进行分析，最后将说明如何利用 Profiler 进行度量。我们将讨论针对当前处理器和操作系统的（特别是内存缓存、页面和虚存等等）相应的数据表示效率问题。在此之后，这一章还将谈到测试与调试，这些内容在许多书中都被遗漏了，但是实际上它们对于所有程序员而言都是相当重要的。

第 2 章所涵盖的内容为数组。我们将学习对于数组的标准语言支持，这也包括对动态数组的支持；还会讨论到 TList 类；另外将创建一个封装了一个记录数组的类。另一个需要强

调的数组为串，所以我们在此也将对它加以介绍。

第 3 章会介绍链表，这包括单链表和双向链表。通过利用单链表和数组来实现栈和队列，我们还将学习到如何创建栈和队列。

第 4 章讨论了查找算法，特别是顺序查找和二分查找算法。我们还将看到如何利用二分查找在一个有序数组或链表中插入项。

第 5 章要介绍排序算法。我们将学习诸多类型的排序方法，包括：冒泡排序、摇动排序、选择排序、希尔排序、快速排序和归并排序等。最后还将对数组和链表进行排序。

第 6 章将讨论有关随机数的算法。我们将看到一个 PRNG（pseudorandom number generator，伪随机数生成器），还将介绍一种称为跳转表的卓越的有序数据结构，在此使用了一个 PRNG 从而实现结构的平衡。

第 7 章所考虑的是散列及散列表，在此涉及到为什么使用散列表，以及其优缺点等问题。这里将介绍一些标准散列算法。在散列表中存在着一个问题即冲突，因此我们还将了解如何利用各种类型的试探法和链式方法加以解决。

第 8 章将介绍二叉树，这是应用相当广泛的一种重要的数据结构。我们将了解到如何建立和维护一棵二叉树，以及如何遍历树中的节点。在此还将引入不平衡树，这是通过有序地插入数据而创建的。后面还将提供两个平衡算法：伸展树和红黑树。

第 9 章所解决的是优先队列，并由此介绍了堆结构。我们将考虑一些重要的堆操作，包括上冒和下滴，还会了解到利用堆结构可以为我们提供一种方便的排序算法：堆排序。

第 10 章提供了有关状态机的信息，并说明了如何利用状态机解决一类特定的问题。在举出有限确定状态机的几个示例之后，这一章接下来研究了正则表达式，并介绍了如何对其进行解析并编译为有限非确定状态机，最后还将把状态机应用于接受和拒绝串的情况。

第 11 章将提到一些数据压缩技术。在此会介绍 Shannon-Fano、哈夫曼编码、伸展树压缩和 LZ77 等算法。

第 12 章将涉及到一些高级技术，如果你对搜索算法和结构感兴趣，那么这将很合你的胃口。当然，它们对于满足程序设计需求而言也大有裨益。

最后本书还包括了一个参考文献部分，在此列出了有关的参考文献，由此可以帮助你找到本书所描述算法的更多信息；在这些参考文献中不仅包括其他的算法书籍，还包括一些学术论文和文章。

排版约定

标准正文即采用目前这种字体和字号。标准正文用于讨论、描述等内容。

```
Code Listings are written in this font, at this size.
```

文本中出现的 WWW 网址或 E-mail 地址，均以斜体表示，并带有下划线，如 *http://www.boyet.com/dads*。

在书中将不时出现这种提示。其目的是为了表述一些重要的观点，也可能是一个提醒或警告。

代码中这些奇怪的$IFDEF 是什么

本书中的代码除了某些特别指出的以外均可在 Delphi 1、2、3、4、5 和 6 以及 Kylix 1 中编译通过（以后若出现新的编译器同样将得到支持。有关最新情况请参见 *http://www.boyet.com/dads*）。尽管我已做最大努力，但不同版本的 Delphi 和 Kylix 中的代码有时仍存在着差异。

对此的答案自然是在代码中设置$IFDEF，从而使某些模块只在特定编译器中编译而排除其他编译器的情况。Borland 为我们提供了面向平台定义的 Windows、Win32 和 Linux 编译器，以及面向编译器版本定义的 VERnnn 编译器。

为了解决这一问题，本书中的每个源代码在顶部均包括以下内容：

```
{$I TDDefine.inc}
```

其中包括的文件定义了面向不同编译器的可读编译器定义。以下将分别列出：

DelphiN	针对某一特定 Delphi 版本而定义，N=1，2，3，4，5，6
DelphiNPlus	针对某一特定 Delphi 及以后版本而定义，N=1，2，3，4，5，6
KylixN	针对某一特定 Kylix 版本而定义，N=1
KylixNPlus	针对某一特定 Kylix 及以后版本而定义，N=1
HasAssert	定义编译器是否支持 Assert

在此我还假设除 Delphi 1 以外各种编译器均支持长串。

有什么 Bug 吗

这本书完全出自常人之手，编写、检查和编辑均由人完成。亚历山大教皇在“An Essay on Criticism”一文中所言“出错乃人之常情，原谅将得以永生”。尽管我做了很大努力，甚至用上了《Fowler's Modern English Usage》、放大镜以及齿很密的梳子，仍难以避免书中出现表述不正确、语法错误、拼写错误以及存在 Bug 的情况。对于这样一本技术书籍，旨在将准确的事实持久地反映在纸张之上，因此存在这些错误实在不可饶恕。

正因如此，我将在我的网站上维护一个勘误表，并将对代码中存在的问题进行纠正。另外，在此网站中你还将看到其他的一些文章，相对于本书中所谈到的某些问题，它们将有更为深入的论述。最新的勘误表和代码补丁可参见 *http://www.boyet.com/dads*。如果你确实发现了错误，请通过 E-mail 将详细情况通过 *julianb@boyet.com* 发给我，在此致以诚挚的感谢，我会对有关问题进行修改并更新网站。

致　谢

如果没有某些人的帮助，这本书将永远无法完成。在此我将按时间顺序以及对我的影响程度将曾给予过我帮助的人列出以表示感谢。

首当其冲的两位人士尽管我从未与之谋面也从未与他们交谈过，但确实是他们点燃了我对算法领域的热情。如果没有他们，我现在会在哪里、会做什么真是不得而知。我所说的就是 Donald Knuth（*http://www-cs-staff.stanford.edu/~knuth/*）和 Robert Sedgewick （*http://www.cs.princeton.edu/~rs/*）。实际上，正是后者的《Algorithms》一书[20]使我得以起步，这也是我所购买的第一本算法书，这还要追溯到我学习 Turbo Pascal 的年代。对 Donald Knuth 无须做过多介绍。他的鸿篇巨著《The Art of Computer Programming》[11，12，13]在算法这棵大树上永踞于顶层；我最早是在伦敦大学 Kings 学院用到这本书的，当时我正在攻读数学硕士学位。

时间飞逝，很快许多年过去了，Kim Kokkonen 则是我要致以感谢的另一个人。是他使我在 TurboPower Software 公司（*http://www.turbopower.com*）有了一份工作，并使我有机会学习更多计算机科学领域的技术，这正是我以往可望而不可及的。当然，还要向这些年来我认识的 TurboPower 的所有员工以及客户表示最诚挚的谢意。特别要感谢我们的总裁 Robert DelRossi，是他的鼓励促使我完成了这本书。

下一个要感谢的是一个名为 Natural Systems 的小公司，不过目前它已经不复存在了。1993 年，他们曾推出过一个称为“面向 Trubo Pascal 的数据结构”的产品。我购买了这一产品，而且就我看来，它并不是很完善。嗯，尽管它工作得很好，但是我对其设计和实现存有异议，而且它的运行速度不够快。这就驱使我编写了面向 Borland Pascal 7 的自由软件 EZSTRUCS 库，并由此开发了 EZDSL，这是我为 Delphi 开发的一个知名的数据结构库自由软件。正是这一尝试使我真正理解了数据结构，确实，有些情况下只有真正做了才能有所掌握。

其次还要向 Chris Frizelle，《The Delphi Magazine》（*http://www.thedelphimagazine.com*）的主编表示感谢。由于他的高瞻远瞩，使我得以在这本无与伦比的杂志上发表各种算法，而且最后还每月为我单辟了一个专栏：“Algorithms Alfresco”。虽然没有他的帮助，这本书也可以写出来，但是自然无法达到现在的水平。在此我衷心地推荐你订阅《The Delphi Magazine》，依我的观点，这是 Delphi 程序员可参考的最有深度、最能体现智慧的杂志。另外也需要向我的所有读者致以谢意，感谢他们对此专栏的建议和评论。

另外，我要向 Wordware（*http://www.wordware.com*）的所有人员表示感谢，其中包括我的编辑、出版人 Jim Hill，以及策划编辑 Wes Beckwith。在我最初提议出版一本关于算法的书时，Jim 还有点怀疑，但他很快就充分理解了我的思路，并在这本书的酝酿期间给我提供了大力支持。我还要向我的技术编辑 Steve Teixeira 表示最衷心的感谢，他是《Delphi n Developer's Guide》（在写这本书时，n 尚为 5）一书的合著者，这是一本有关 Delphi 精华的

指南。另外还要向我的朋友 Anton Parris 致以诚挚的谢意。

最后，要向我的妻子 Donna 献上我的感激和爱，是她最早鼓励我写这本书的。没有她的爱、热情和鼓励，我在数年前可能就放弃了。谢谢你，我的爱人。

Julian M. Bucknall

Colorado Springs，1999 年 4 月～2001 年 2 月

目　录

第 1 章 什么是算法

对于一本关于算法的书，我们首先应该明确自己确实要讨论什么。可以看到，之所以要理解和搜索算法，其主要原因是力图使我们的应用更为快捷。当然，必须承认有的时候算法对空间有效性的要求要胜于对时间效率的要求，但一般来讲，时间效率仍是我们所追求的性能目标。

尽管这是一本关于算法、数据结构以及如何用代码加以实现的书，但我们还是要讨论一些过程性的算法，这包括：如何编写代码从而有助于我们在其出错时进行调试，如何测试代码，以及如何确保某一处的调整不至于影响到其他部分。

1.1 什么是算法

事实上，在我们的编程生涯中，一直都在使用算法，但我们往往却并不认为它们是算法；而是认为“它们不是算法，只不过是解决问题的方法而已”。

算法（algorithm）是完成某些计算或处理的步骤序列。这个定义有些过于笼统，不过一旦你了解到算法从本质上讲并没有什么可怕的，那么在认识和使用算法时就不会再顾虑重重了。

回想上小学的日子，当你学习加法的时候，老师可能会在黑板上写出如下的求和式：

```
  45
 +17
```

然后要求你把它们加起来。你已经学习了如何做到这一点：首先处理个位，即让 5 加 7 得到 12，然后将 2 留在个位上，而将 1 进至 4 之上。

```
  4 5
 +1,7
  2
```

然后再把所进的 1 和 4 以及另一个 1 相加得到 6，这就是要写在十位上的值。由此即可得到最终的答案：62。

注意，你所学会的就是完成这个加法（以及所有类似加法）的算法。老师教给你的并非如何让 45 和 17 相加这一特定问题，而是提供了一种完成两个数相加的通用方法。实际上，一旦掌握，你很快就可以实现多个数的相加，而且每个数也可以有多位，在此仍可应

用同样的算法。当然，在那个时候，没有人告诉你这就是算法，它只被认为是数相加的方法。

在程序设计领域中，我们倾向于将算法理解为完成某些计算的复杂方法。例如，如果我们有一个客户记录的数组，并希望从中找到某个特定的客户（如，John Smith），就可能逐元素地遍查整个数组，直至找到 John Smith，或者到达数组尾部。这看上去再明显不过了，我们通常不认为这是算法，但实际上它不折不扣的确实是算法，一般称为“顺序查找（sequential search）”。

在这个假想的数组中查找“John Smith”可能还有其他方法。例如，如果数组已根据客户的姓氏做了排序，我们就可以使用二分查找（binary search）算法来找到 John Smith。首先找到数组中最中间的元素，他是 John Smith 吗？如果是，则大功告成。如果小于 John Smith（这里的“小于”所指的是依字母顺序位于 John Smith 的前面），那么我们就可以想到 John Smith 必然落在数组的前半部分。如果大于，则应在数组的后半部分。此后可以继续做同样的工作，即再挑出中间元素，并选择可能包括 John Smith 的数组部分，从而逐步地将数组切为越来越小的部分，直至找到相应元素，或者所剩余的数组为空。

当然，这个算法比起前面的顺序查找看上去要复杂一些。顺序查找只需利用一个相当简单的 For 循环，并在合适的位置上设置一个 Break 调用即可实现；而二分查找的代码则需要更多的计算以及局部变量。因此顺序查找好像要快一些，原因就在于其代码更易编写。

下面我们就要步入算法分析的世界了，在此会做一些试验，并对不同算法具体如何工作进行总结归纳以得到相关规律。

1.1.1 算法分析

先来看查找数组中“John Smith”的两种可能的算法：顺序查找和二分查找。这两个算法我们均将实现，并加以运行以确定其性能属性。代码清单 1.1 即为简单的顺序查找。

代码清单 1.1　查找数组中某个名字的顺序查找

```
function SeqSearch(aStrs : PStringArray; aCount : integer;
          const aName : string5) : integer;
var
  i : integer;
begin
  for i := 0 to pred(aCount) do
    if CompareText(aStrs^[i], aName) = 0 then begin
      Result := i;
      Exit;
    end;
  Result := -1;
end;
```

代码清单 1.2 所示的是更为复杂的二分查找（目前我们不打算对此例程的具体工作加以分析——第 4 章中将详细讨论二分查找算法）。

代码清单 1.2　查找数组中某个名字的二分查找

```
function BinarySearch(aStrs : PStringArray; aCount : integer;
            const aName : string5) : integer;
var
  L, R, M : integer;
  CompareResult : integer;
begin
  L := 0;
  R := pred(aCount)
 while (L <= R) do begin
  M := (L + R) div 2;
  CompareResult := CompareText(aStrs^[M], aName);
  if (CompareResult = 0) then begin
   Result := M;
   Exit;
  end
  else if (CompareResult < 0) then
    L := M + 1
  else
     R := M - 1;
 end;
 Result := -1;
end;
```

如果光看这两个例程，将很难判定其性能如何。实际上，这也是我们必须了然于胸的一个原则：只靠看，是难于区别代码的速度效率的。要想真正得出代码到底有多快，惟一的办法就是运行它，除此之外别无他法。假若我们需要在诸算法中做出选择，例如目前就是这种情况，那么就需要在不同环境中基于不同的输入进行测试并计算时间，从而确定哪一种算法更适于我们的需求。

用于计算时间的传统方法是借助于一个 Profiler（即评测工具）。Profiler 程序将“装载”测试应用，并将准确地记录我们所感兴趣的不同例程的时间。我建议在所有编程项目中都要使用 Profiler，并把它作为必经之路。只有利用 Profiler 你才能真正确定你的应用将主要时间都花费在了哪里，相应地可以明确哪一个例程值得你花时间进行优化。

我所任职的公司，即 TurboPower Software 公司在其推出的 Sleuth QA Suite 产品中有一个专业的 Profiler。我已经对本书中所有代码利用 StopWatch（Sleuth QA Suite 中的 Profiler 程序名）和 CodeWatch（此套件中的资源和内存泄露调试工具）做了测试。不过，即使你并没有任何 Profiler，仍可对例程进行测试并测算其时间；只不过要略显笨拙一些，因为你必须将计时例程的调用内嵌到代码中。值得购买的 Profiler 是不会对你的代码有所调整的，它会在运行时通过修改内存中的可执行代码来施展其魔力。

对于此查找算法的测试，我编写了一个检测程序以自行完成时间度量。基本做法是，在代码开始处取其系统时间，并在代码结束处再次取系统时间。根据这两个值即可计算出完成任务所需的时间。实际上，由于当前的计算机越来越快，而 PC 时钟的分辨率却很低，因此通常检查数百次例程调用所需的时间更为合适，由此我们可以得到一个平均值。（顺便提一句，这一程序是面向 32 位 Delphi 而编写的，不能在 Delphi 1 中编译，因为它在堆中分配的

数组超出了 Delphi 1 的 64KB 的限制。）

我以多种不同的形式完成了此性能测试。首先，我测量了在分别包括 100、1000、10 000 和 100 000 个元素的数组中查找“Smith”所需的时间，在此对两种算法均做了测试，而且确保数组中必有“Smith”。在下一阶段的测试中，我仍基于相同规模的数组分别采用这两种算法对查找“Smith”所需的时间进行了测定，不过这一次则要确保“Smith”不存在。表 1.1 显示了我的测试结果。

表 1.1　顺序和二分查找的时间度量

	失败	成功
顺序查找		
100	0.14	0.10
1000	1.44	1.05
10 000	15.28	10.84
100 000	149.42	106.35
二分查找		
100	0.01	0.01
1000	0.01	0.01
10 000	0.02	0.02
100 000	0.03	0.02

能够看到，由以上时间度量可得出一些很有意思的结论。完成顺序查找所需的时间与数组中的元素个数呈正比增长。我们把顺序查找的执行特性称为是线性的。

不过，由二分查找统计总结出相应特性则要困难一些。实际上，由于算法过快，看上去就像是时间分辨率问题。所需时间与数组中元素个数之间的关系不再简单地为线性关系。应该比这要小一些，但由这些测试无法具体得出。

我重新做了测试，并将二分查找的时间乘上一个因数 100 以作扩展。

表 1.2　重测二分查找的时间

	失败	成功
100	0.89	0.57
1000	1.47	1.46
10 000	2.06	2.06
100 000	2.50	2.41

在此我们得到了一组更有意义的数据。可以看到随着元素个数呈 10 倍增长，相应的运行时间则按一个常量递增（大致为半个单位的样子）。这是一个对数关系：二分查找所需的时间随数组中元素个数的对数呈比例增长（对于一个不是搞数学的人来说，要看出这一点稍显困难。回忆一下你上学的日子，应该记得让两个数相乘的一种方法是先计算其对数，再予

相加，然后再计算其反对数即可得出答案。由于我们在这些性能测试中逐次均乘了一个因数10，因此以对数的形式看则相当于增加了一个常数。在此例中恰好可以由结果看出这一点：每次均增加半个单位）。

这样，我们能够由这个测试结果了解到什么呢？首先的收获是，我们知道了要理解一个算法的性能特性，惟一的办法就是进行时间度量。

> 通常情况下，要看一段代码的效率如何，惟一的做法就是测试其执行时间。这适用于你所编写的任何代码，无论是在使用一个知名的算法，亦或是自行设计以满足当前状况。不要有疑虑，放手测试吧。

另一个收获是，我们已经看到顺序查找本质上是线性的，而二分查找则是对数性的。如果对数学有所侧重，可以取这些统计结果来证明相应定理。不过在这本书中，我不打算在文字中充斥大量的数学内容，在这一方面许多大学的教材要比我更为在行。

1.1.2 大O符号

我们需要一种简洁的符号来表示所测试的性能特性，而无须采用诸如“算法 X 的性能与元素个数的三次方成正比”的说法，或者是其他同样冗长的提法。在计算机科学中已经有相应的机制，即所谓的大 O 符号（big-Oh notation）。

对于这个符号，我们要得出关于元素个数 n 的数学函数，而算法的性能若与此函数成正比，则称算法是一个 O($f(n)$)算法，在此 $f(n)$即为 n 的某个函数。我们把它读作“大 O $f(n)$”，或者不太严格的话，也称为“与 $f(n)$成正比”。

例如，前面的测试表明顺序查找是一个 O(n)算法。而二分查找则有所不同，是一个O(log(n))算法。由于 log(n)<n，因此对于所有正数 n，我们可以认为二分查找比顺序查找要快。不过，我很快还将提出一些警告以避免你由大 O 符号做出过于武断的结论。

大 O 符号相当简洁明了。假设通过测试我们得出算法 X 是 O(n^2+n)的，换句话说，即其性能与 n^2+n 成正比。在此“成正比”的含义是指可以找到一个常量 k，从而使以下等式为真：

性能=k * (n^2+n)

由此等式，以及其他得自于大 O 符号的等式，我们首先可以看出在大 O 的括号里将数学函数乘上一个常量值不会有任何作用。例如，O(3*$f(n)$)就等于 O($f(n)$)，可以从符号中将 3 取出，并与外部的比值常量相乘，不过为方便起见，我们通常会将此忽略。

如果在测试算法 X 时 n 的值足够大，则可以确定地说“+n”一项的作用将被“n^2”项所掩盖。换句话说，假设 n 足够大，O(n^2+n)就等于 O(n^2)。另外这一原则也适用于任何 n 的附加项：只要有一个足够大的 n，就可以将此附加项予以忽略，这一点是能够保证的，因为它的作用会被另外的 n 项所掩盖。因此，一个 n^2 项会被一个 n^3 项所吸收，而一个 log(n)项则会被一个 n 项所吸收，以此类推。

这说明大 O 符号的运算是相当简单的。为了进行讨论，假设我们有一个完成多种不同任务的算法。第一个任务，就其本身而言是 O(n)的，而第二个任务则对应为 O(n^2)，第三个是 O(log(n))的。那么这个算法性能的总体大 O 值又是什么呢？答案是 O(n^2)，因为这是到此为止算法中最主要的部分。