TURING 图灵原版计算机科学系列

PEARSON

双色印刷

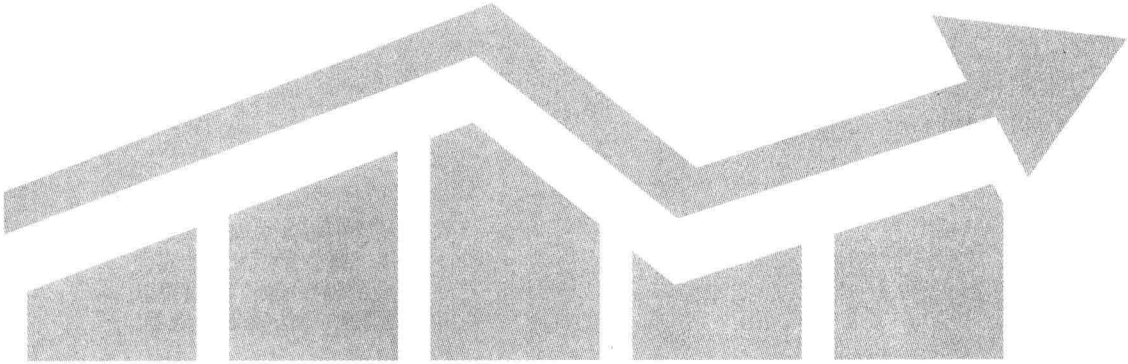
- Sedgewick之巨著，与高德纳TAOCP一脉相承
- 几十年多次修订，经久不衰的畅销书
- 涵盖所有程序员必须掌握的50种算法

# 算法 Algorithms Fourth Edition （英文版·第4版）

[美] Robert Sedgewick
Kevin Wayne 著

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵原版计算机科学系列



# 算法 Algorithms Fourth Edition （英文版·第4版）

[美] Robert Sedgewick
Kevin Wayne 著

人民邮电出版社
北京

**图书在版编目（CIP）数据**

算法 : 第4版 = Algorithms,Fourth Edition : 英文 / (美) 塞奇威克 (Sedgewick,R.), (美) 韦恩 (Wayne,K.) 著. -- 北京 : 人民邮电出版社, 2012.3
(图灵原版计算机科学系列)
ISBN 978-7-115-27146-4

Ⅰ. ①算… Ⅱ. ①塞… ②韦… Ⅲ. ①计算机算法－英文 Ⅳ. ①TP301.6

中国版本图书馆CIP数据核字(2011)第258500号

## 内 容 提 要

本书作为算法领域经典的参考书，全面介绍了关于算法和数据结构的必备知识，并特别针对排序、搜索、图处理和字符串处理进行了论述。第4版具体给出了每位程序员应知应会的50个算法，提供了实际代码，而且这些Java代码实现采用了模块化的编程风格，读者可以方便地加以改造。本书配套网站提供了本书内容的摘要及更多的代码实现、测试数据、练习、教学课件等资源。

本书适合用作大学教材或从业者的参考书。

图灵原版计算机科学系列
**算法（英文版·第4版）**

◆ 著 [美] Robert Sedgewick Kevin Wayne
责任编辑 朱 巍
执行编辑 毛倩倩

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：787×1092 1/24
印张：40.33
字数：966千字 2012年3月第1版
印数：1－2 500册 2012年3月第1次印刷

著作权合同登记号 图字：01-2011-7803号
ISBN 978-7-115-27146-4

定价：99.00元

**读者服务热线：(010)51095186转604 印装质量热线：(010)67129223**
**反盗版热线：(010)67171154**

# 版权声明

谨以此书献给Adam、Andrew、Brett、Robbie，并特别感谢Linda。

——Robert Sedgewick

献给Jackie和Alex。

——Kevin Wayne

# 前言

本书力图研究当今最重要的计算机算法并将一些最基础的技能传授给广大求知者。它适合用作计算机科学进阶教材，面向已经熟悉了计算机系统并掌握了基本编程技能的学生。本书也可用于自学，或是作为开发人员的参考手册，因为书中实现了许多实用算法并详尽分析了它们的性能特点和用途。这本书取材广泛，很适合作为该领域的入门教材。

*算法和数据结构的学习*是所有计算机科学教学计划的基础，但它并不只是对程序员和计算机系的学生有用。任何计算机使用者都希望计算机能运行得更快一些或是能解决更大规模的问题。本书中的算法代表了近50年来的大量优秀研究成果，是人们工作中必备的知识。从物理中的$N$体模拟问题到分子生物学中的基因序列问题，我们描述的基本方法对科学研究而言已经必不可少；从建筑建模系统到模拟飞行器，这些算法已经成为工程领域极其重要的工具；从数据库系统到互联网搜索引擎，算法已成为现代软件系统中不可或缺的一部分。这仅是几个例子而已，随着计算机应用领域的不断扩张，这些基础方法的影响也会不断扩大。

在开始学习这些基础算法之前，我们先要熟悉全书中都将会用到的栈、队列等低级抽象的数据类型。然后我们依次研究排序、搜索、图和字符串方面的基础算法。最后一章将会从宏观角度总结全书的内容。

## 独特之处

本书致力于研究有实用价值的算法。书中讲解了多种算法和数据结构，并提供了大量相关的信息，读者应该能有信心在各种计算环境下实现、调试并应用它们。本书的特点涉及以下几个方面。

*算法* 书中均有算法的完整实现，并讨论了程序在多个样例上的运行状况。书中的代码都是可以运行的程序而非伪代码，因此非常便于投入使用。书中程序是用Java语言编写的，但其编程风格方便读者使用其他现代编程语言重用其中的大部分代码来实现相同算法。

*数据类型* 我们在数据抽象上采用了现代编程风格，将数据结构和算法封装在了一起。

*应用* 每一章都会给出所述算法起到关键作用的应用场景。这些场景多种多样，包括物理模拟与分子生物学、计算机与系统工程学，以及我们熟悉的数据压缩和网络搜索等。

*学术性* 我们非常重视使用数学模型来描述算法的性能。我们用模型预测算法的性能，然后在真实的环境中运行程序来验证预测。

*广度* 本书讨论了基本的抽象数据类型、排序算法、搜索算法、图及字符串处理。我们在算法的讨论中研究数据结构、算法设计范式、归纳法和解题模型。这将涵盖20世纪60年代以来的经典方法以及近年来产生的新方法。

我们的主要目标是将今天最重要的实用算法介绍给尽可能广泛的群体。这些算法一般都十分巧妙奇特，20行左右的代码就足以表达。它们展现出的问题解决能力令人叹为观止。没有它们，创造计算智能、解决科学问题、开发商业软件都是不可能的。

### 本书网站

本书的一个亮点是它的配套网站algs4.cs.princeton.edu。这一网站面向的是老师、学生和专业人士，包含了关于算法和数据结构的丰富资料，均可免费获得。

*一份在线大纲* 包含了本书内容的结构并提供了链接，浏览起来十分方便。

*全部实现代码* 书中所有的代码均可以在这里找到，且其形式适合用于程序开发。此外，还包括算法的其他实现，例如高级的实现、书中提及的改进的实现、部分习题的答案以及多个应用场景的客户端代码。我们的重点是用真实的应用环境来测试算法。

*习题与答案* 网站还提供了一些附加的选择题（只需要一次单击便可获取答案）、很多算法应用的例子、编程练习和答案以及一些有挑战性的难题。

*动态可视化* 书是死的，但网站是活的，在这里我们充分利用图形类演示了算法的应用效果。

*课程资料* 网站包含和本书及网上内容对应的一整套幻灯片，以及一系列编程作业、核对表、测试数据和备课手册。

*相关资料链接* 网站包含大量的链接，提供算法应用的更多背景知识以及学习算法的其他资源。

我们希望这个站点和本书互为补充。一般来说，建议读者在第一次学习某种算法或是希望获得整体概念时看书，并把网站作为编程时的参考或是在线查找更多信息的起点。

### 作为教材

本书为计算机科学专业进阶的教材。它涵盖了这门学科的核心内容，并能让学生充分锻炼编程、定量推理和解决问题等方面的能力。一般来说，此前学过一门计算机方面的先导课程就足矣，只要熟悉一门现代编程语言并熟知现代计算机系统，就都能够阅读本书。

虽然本书使用Java实现算法和数据结构，但其代码风格使得熟悉其他现代编程语言的人也能看懂。我们充分利用了Java的抽象性（包括泛型），但不会依赖这门语言的独门特性。

书中涉及的多数数学知识都有完整的讲解（少数会有延伸阅读），因此阅读本书并不需要准备太多数学知识，不过有一定的数学基础当然更好。应用场景都来自其他学科的基础内容，同样也在书中有完整介绍。

本书涉及的内容是任何准备主修计算机科学、电气工程、运筹学等专业的学生应了解的基础知识，并且对所有对科学、数学或工程学感兴趣的学生也十分有价值。

### 背景介绍

这本书意在接续我们的一本基础教材《Java程序设计：一种跨学科的方法》，那本书对计算机领域做了概括性介绍。这两本书合起来可用作两到三个学期的计算机科学入门课程教材，为所有学生在自然科学、工程学或社会科学中解决计算问题提供必备的基础知识。

本书大部分内容来自Sedgewick的算法系列书籍。本质上，本书和该系列的第1版和第2版最接近，但还包含了作者多年教学和学习的经验。Sedgewick的《C算法（第3版）》、《C++算法（第3版）》、《Java算

法（第3版）》更适合用作参考书或是高级课程的教材，而本书则是专门为大学一、二年级学生设计的一学期教材，也是最新的基础入门书或从业者的参考书。

致谢

本书的编写花了近40年时间，因此想要一一列出所有参与人是不可能的。本书的前几版一共列出了好几十人，其中包括（按字母顺序）Andrew Appel、Trina Avery、Marc Brown、Lyn Dupré、Philippe Flajolet、Tom Freeman、Dave Hanson、Janet Incerpi、Mike Schidlowsky、Steve Summit和Chris Van Wyk。我要感谢他们所有人，尽管其中有些人的贡献要追溯到几十年前。至于第4版，我们要感谢试用了本书样稿的普林斯顿及其他院校的数百名学生，以及通过本书网站发表意见和指出错误的世界各地的读者。

我们还要感谢普林斯顿大学对于高质量教学的坚定支持，这是本书得以面世的基础。

Peter Gordon几乎从本书写作之初就提出了很多有用的建议，这一版奉行的“归本溯源”的指导思想也是他最早提出的。关于第4版，我们要感谢Barbara Wood认真又专业的编辑工作，Julie Nahil对生产过程的管理，以及Pearson出版公司中为本书的付梓和营销辛勤工作的朋友。所有人都在积极地追赶进度，而本书的质量并没有受到丝毫影响。

# 目录

ONE

# Fundamentals

The objective of this book is to study a broad variety of important and useful *algorithms*—methods for solving problems that are suited for computer implementation. Algorithms go hand in hand with *data structures*—schemes for organizing data that leave them amenable to efficient processing by an algorithm. This chapter introduces the basic tools that we need to study algorithms and data structures.

First, we introduce our *basic programming model*. All of our programs are implemented using a small subset of the Java programming language plus a few of our own libraries for input/output and for statistical calculations. SECTION 1.1 is a summary of language constructs, features, and libraries that we use in this book.

Next, we emphasize *data abstraction*, where we define *abstract data types* (ADTs) in the service of modular programming. In SECTION 1.2 we introduce the process of implementing an ADT in Java, by specifying an *applications programming interface* (API) and then using the Java class mechanism to develop an implementation for use in client code.

As important and useful examples, we next consider three fundamental ADTs: the *bag*, the *queue*, and the *stack*. SECTION 1.3 describes APIs and implementations of bags, queues, and stacks using arrays, resizing arrays, and linked lists that serve as models and starting points for algorithm implementations throughout the book.

Performance is a central consideration in the study of algorithms. SECTION 1.4 describes our approach to analyzing algorithm performance. The basis of our approach is the *scientific method*: we develop hypotheses about performance, create mathematical models, and run experiments to test them, repeating the process as necessary.

We conclude with a case study where we consider solutions to a *connectivity* problem that uses algorithms and data structures that implement the classic *union-find* ADT.

**Algorithms** When we write a computer program, we are generally implementing a *method* that has been devised previously to solve some problem. This method is often independent of the particular programming language being used—it is likely to be equally appropriate for many computers and many programming languages. It is the method, rather than the computer program itself, that specifies the steps that we can take to solve the problem. The term *algorithm* is used in computer science to describe a finite, deterministic, and effective problem-solving method suitable for implementation as a computer program. Algorithms are the stuff of computer science: they are central objects of study in the field.

We can define an algorithm by describing a procedure for solving a problem in a natural language, or by writing a computer program that implements the procedure, as shown at right for *Euclid's algorithm* for finding the greatest common divisor of two numbers, a variant of which was devised over 2,300 years ago. If you are not familiar with Euclid's algorithm, you are encouraged to work EXERCISE 1.1.24 and EXERCISE 1.1.25, perhaps after reading SECTION 1.1. In this book, we use computer programs to describe algorithms. One important reason for doing so is that it makes easier the task of checking whether they are finite, deterministic, and effective, as required. But it is also important to recognize that a program in a particular language is just one way to express an algorithm. The fact that many of the algorithms in this book have been expressed in multiple programming languages over the past several decades reinforces the idea that each algorithm is a method suitable for implementation on any computer in any programming language.

English-language description

Compute the greatest common divisor of two nonnegative integers $p$ and $q$ as follows: If $q$ is 0, the answer is $p$. If not, divide $p$ by $q$ and take the remainder $r$. The answer is the greatest common divisor of $q$ and $r$.

Java-language description

```
public static int gcd(int p, int q)
{
   if (q == 0) return p;
   int r = p % q;
   return gcd(q, r);
}
```

**Euclid's algorithm**

Most algorithms of interest involve organizing the data involved in the computation. Such organization leads to *data structures*, which also are central objects of study in computer science. Algorithms and data structures go hand in hand. In this book we take the view that data structures exist as the byproducts or end products of algorithms and that we must therefore study them in order to understand the algorithms. Simple algorithms can give rise to complicated data structures and, conversely, complicated algorithms can use simple data structures. We shall study the properties of many data structures in this book; indeed, we might well have titled the book *Algorithms and Data Structures*.

When we use a computer to help us solve a problem, we typically are faced with a number of possible approaches. For small problems, it hardly matters which approach we use, as long as we have one that correctly solves the problem. For huge problems (or applications where we need to solve huge numbers of small problems), however, we quickly become motivated to devise methods that use time and space efficiently.

The primary reason to learn about algorithms is that this discipline gives us the potential to reap huge savings, even to the point of enabling us to do tasks that would otherwise be impossible. In an application where we are processing millions of objects, it is not unusual to be able to make a program millions of times faster by using a well-designed algorithm. We shall see such examples on numerous occasions throughout the book. By contrast, investing additional money or time to buy and install a new computer holds the potential for speeding up a program by perhaps a factor of only 10 or 100. Careful algorithm design is an extremely effective part of the process of solving a huge problem, whatever the applications area.

When developing a huge or complex computer program, a great deal of effort must go into understanding and defining the problem to be solved, managing its complexity, and decomposing it into smaller subtasks that can be implemented easily. Often, many of the algorithms required after the decomposition are trivial to implement. In most cases, however, there are a few algorithms whose choice is critical because most of the system resources will be spent running those algorithms. These are the types of algorithms on which we concentrate in this book. We study fundamental algorithms that are useful for solving challenging problems in a broad variety of applications areas.

The sharing of programs in computer systems is becoming more widespread, so although we might expect to be *using* a large fraction of the algorithms in this book, we also might expect to have to *implement* only a small fraction of them. For example, the Java libraries contain implementations of a host of fundamental algorithms. However, implementing simple versions of basic algorithms helps us to understand them better and thus to more effectively use and tune advanced versions from a library. More important, the opportunity to reimplement basic algorithms arises frequently. The primary reason to do so is that we are faced, all too often, with completely new computing environments (hardware and software) with new features that old implementations may not use to best advantage. In this book, we concentrate on the simplest reasonable implementations of the best algorithms. We do pay careful attention to coding the critical parts of the algorithms, and take pains to note where low-level optimization effort could be most beneficial.

The choice of the best algorithm for a particular task can be a complicated process, perhaps involving sophisticated mathematical analysis. The branch of computer science that comprises the study of such questions is called *analysis of algorithms*. Many

of the algorithms that we study have been shown through analysis to have excellent theoretical performance; others are simply known to work well through experience. Our primary goal is to learn reasonable algorithms for important tasks, yet we shall also pay careful attention to comparative performance of the methods. We should not use an algorithm without having an idea of what resources it might consume, so we strive to be aware of how our algorithms might be expected to perform.

**Summary of topics** As an overview, we describe the major parts of the book, giving specific topics covered and an indication of our general orientation toward the material. This set of topics is intended to touch on as many fundamental algorithms as possible. Some of the areas covered are core computer-science areas that we study in depth to learn basic algorithms of wide applicability. Other algorithms that we discuss are from advanced fields of study within computer science and related fields. The algorithms that we consider are the products of decades of research and development and continue to play an essential role in the ever-expanding applications of computation.

*Fundamentals* (CHAPTER 1) in the context of this book are the basic principles and methodology that we use to implement, analyze, and compare algorithms. We consider our Java programming model, data abstraction, basic data structures, abstract data types for collections, methods of analyzing algorithm performance, and a case study.

*Sorting* algorithms (CHAPTER 2) for rearranging arrays in order are of fundamental importance. We consider a variety of algorithms in considerable depth, including insertion sort, selection sort, shellsort, quicksort, mergesort, and heapsort. We also encounter algorithms for several related problems, including priority queues, selection, and merging. Many of these algorithms will find application as the basis for other algorithms later in the book.

*Searching* algorithms (CHAPTER 3) for finding specific items among large collections of items are also of fundamental importance. We discuss basic and advanced methods for searching, including binary search trees, balanced search trees, and hashing. We note relationships among these methods and compare performance.

*Graphs* (CHAPTER 4) are sets of objects and connections, possibly with weights and orientation. Graphs are useful models for a vast number of difficult and important problems, and the design of algorithms for processing graphs is a major field of study. We consider depth-first search, breadth-first search, connectivity problems, and several algorithms and applications, including Kruskal's and Prim's algorithms for finding minimum spanning tree and Dijkstra's and the Bellman-Ford algorithms for solving shortest-paths problems.