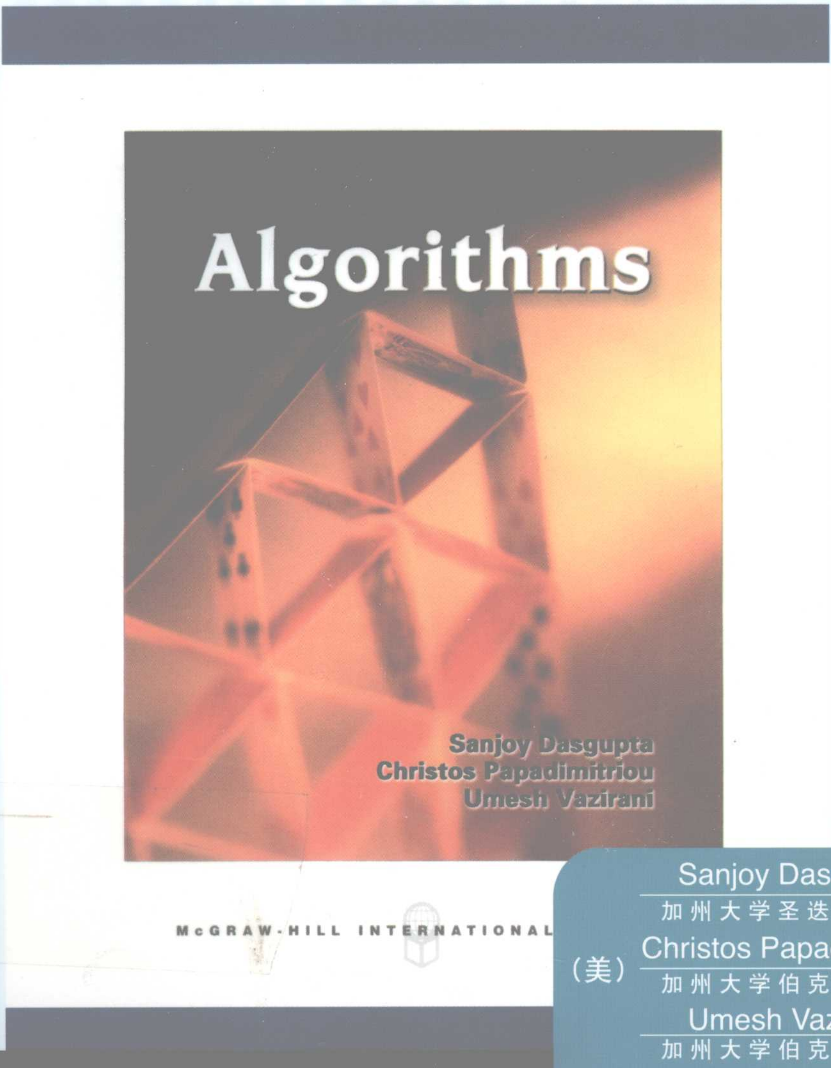
HZ BOOKS
华章教育

McGraw Hill Education

经 典 原 版 书 库

算法概论

（注释版）



（美） Sanjoy Dasgupta
加州大学圣迭戈分校
Christos Papadimitriou
加州大学伯克利分校
Umesh Vazirani
加州大学伯克利分校
著

钱枫 邹恒明 注释

机械工业出版社
China Machine Press

TP301.6
86-参

经 典 原 版 书 库

算法概论

（注释版）

Algorithms

Sanjoy Dasgupta
加州大学圣迭戈分校
（美） Christos Papadimitriou 著
加州大学伯克利分校
Umesh Vazirani
加州大学伯克利分校

[redacted]

钱枫 邹恒明 注释

[redacted]

机械工业出版社
China Machine Press

本书版权登记号：图字：01-2007-3083

图书在版编目（CIP）数据

算法概论（注释版）/（美）达斯格普塔（Dasgupta，S.）等著；钱枫，邹恒明注释.
—北京：机械工业出版社，2009.1
（经典原版书库）
书名原文：Algorithms

ISBN 978-7-111-25361-7

Ⅰ.算… Ⅱ.① 达… ② 钱… ③ 邹… Ⅲ.算法理论-英文 Ⅳ.O241

中国版本图书馆CIP数据核字（2008）第161444号

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
北京京北印刷有限公司印刷 · 新华书店北京发行所发行
2009年1月第1版第1次印刷
186mm × 240mm · 24.25印张
标准书号：ISBN 978-7-111-25361-7
定价：55.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

出版者的话

文艺复兴以降，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；也正是这样的传统，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分社较早意识到“出版要为教育服务”。自1998年开始，华章分社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，Brain 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襄助，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，有的还专程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。华章分社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

HZ BOOKS
华章教育

序 言

《算法概论》的前身是加州大学伯克利分校和加州大学圣迭戈分校本科生的算法课讲义。经过十年课堂教学的检验，这本书以其生动有趣的风格、精心挑选的内容和精确严谨的叙述受到了学术界和读者的一致好评。到目前为止，它是Amazon上获得五星的两本算法教材中的一本（另一本是《算法导论》，中文版已由机械工业出版社出版）。

算法是计算机科学的灵魂，其复杂与抽象让许多学生望而却步。这本书最显著的特点是生动的写作风格：作者贯穿一条主线，以讲故事的形式将概念娓娓道来，非常易于理解和消化。例如，在全书的开头，作者以计数法的更迭为背景（从复杂的罗马数字到简洁的阿拉伯数字），给算法下了这样生动的定义（这样的例子在书中比比皆是）：

"……十进制计数法是由印度人在公元600年左右发明的，它对数学推理简直是一场革命：仅仅需要10个符号，就能很容易地把非常大的数写下来，而且数的运算也能用非常简单的步骤完成。这个绝妙的主意在漫长的时间中克服了语言、距离和人们的无知这些传统因素的重重阻碍而传播到世界各地。推动此传播的是一本9世纪的阿拉伯文的教科书，该书由一个生活在巴格达的叫Al Khwarizmi的人写成。书中不仅列举了加、减、乘、除这些基本运算规则，还包括了求平方根和计算圆周率数值的方法。这些步骤的特点是：简单、没有歧义、机械、有效和正确——这就是算法。几百年后，当十进制计数法在欧洲被广泛使用时，算法（algorithm）这个单词被人们创造出来以纪念这位聪明的Al Khwarizmi先生……"

当然，这本书没有走另一个极端：过分强调语言的生动而忽视了严谨性。恰恰相反，这本书完美地兼顾了两者。在书中我们看不到很多数学式子，取而代之的是精确的文字叙述。作者认为，这种用严谨的语言代替数学形式化的方法更容易被学生接受，因为读者需要知道的往往是蕴涵在数学公式或者程序代码背后的思想，而正是这些思想促成了精巧的算法。

这本书不是一本字典式的百科全书，而是一本教科书。因此，作者合理地挑选了讲授的内容，用300多页的篇幅使学生对这门博大精深的科学有了深刻的认识。本书共分为四个部分。其中，第一部分是引论和算术运算（这是算法的起源），包括复杂度分析、算术运算、最大公约数、素性测试、散列函数、快速乘法、递归、合并排序、矩阵乘法，还有在一般算法书中不多见的RSA公钥体制和快速傅里叶变换等内容。第二部分是"传统"的算法和数据结构（树和图）：图的搜索、连通性、最短路径、最小生成树、堆、赫夫曼编码等。在第三部分里，作者用新颖的方式介绍了两种强大的运筹学算法——动态规划和线性规划，以及它们的应用。利用这两种运筹学算法，能够优美地解决一大批实际问题。最后一部分是关于如何解决困难的问题，包括NP完全、优化搜索（回溯、分支限界）、近似算法等。值得一提的是本书的最后一章——量子算法。作者首次将理论研究中最前沿的内容以通俗易懂的形式写入算法教科书中，给人耳

目一新的感觉。作者以人类最古老的算法（算术运算）为起点，将各种算法中优美而有代表性的内容囊括书中，并以最前沿的理论结束本书，构成了完整的知识体系。

此外，作者以穿插注解框的形式，对正文的叙述作进一步的解释。这些注解框的内容包括：人文历史背景、对复杂概念的进一步阐述、算法的扩展与重要应用等。书中还包括了近300道习题、补充阅读文献列表和术语索引。

在算法类书籍中，《算法导论》的地位是不可动摇的。但是对大多数读者而言，上千页的篇幅和形式化的叙述决定了《算法导论》只能是工具书而非教科书，最多只能作为研究生算法课程的教材，而不太适合本科生。在剩下的为数不多的算法类书籍中，《算法概论》一书在出版两年的时间内，就获得了读者的青睐。它新颖生动、简洁清新，向每个读者展示着算法的魅力，可以作为大学本科生一学期或两学期的教科书，又可以作为算法学习的参考书和补充读物。

我国大学计算机系的算法课一般开设在低年级，引进这本书是十分有必要的，相信它将对国内的算法教学起到推动作用。

本书的第0、3、4、8、9、10章由钱枫注释，第1、2、5、6、7章由邹恒明注释。

另外，在本书注释过程中，得到了以下人士的帮助：康奈尔大学（Cornell University）的范鸿敏，密歇根大学（University of Michigan—Ann Arbor）的方禄俊、李翼、钱志云、许赟靖，以及上海交通大学的张漳、梁伟明、刘渊杰、孙涵、张华君，在此表示感谢。

钱枫　邹恒明

2008年9月于美国密歇根安娜堡和上海莘庄

Preface

This book evolved over the past ten years from a set of lecture notes developed while teaching the undergraduate Algorithms course at Berkeley and U.C. San Diego. Our way of teaching this course evolved tremendously over these years in a number of directions, partly to address our students' background (undeveloped formal skills outside of programming), and partly to reflect the maturing of the field in general, as we have come to see it. The notes increasingly crystallized into a narrative, and we progressively structured the course to emphasize the "story line" implicit in the progression of the material. As a result, the topics were carefully selected and clustered. No attempt was made to be encyclopedic, and this freed us to include topics traditionally de-emphasized or omitted from most Algorithms books.

Playing on the strengths of our students (shared by most of today's undergraduates in Computer Science), instead of dwelling on formal proofs we distilled in each case the crisp mathematical idea that makes the algorithm work. In other words, we emphasized rigor over formalism. We found that our students were much more receptive to mathematical rigor of this form. It is this progression of crisp ideas that helps weave the story.

Once you think about Algorithms in this way, it makes sense to start at the historical beginning of it all, where, in addition, the characters are familiar and the contrasts dramatic: numbers, primality, and factoring. This is the subject of Part I of the book, which also includes the RSA cryptosystem, and divide-and-conquer algorithms for integer multiplication, sorting and median finding, as well as the fast Fourier transform. There are three other parts: Part II, the most traditional section of the book, concentrates on data structures and graphs; the contrast here is between the intricate structure of the underlying problems and the short and crisp pieces of pseudocode that solve them. Instructors wishing to teach a more traditional course can simply start with Part II, which is self-contained (following the prologue), and then cover Part I as required. In Parts I and II we introduced certain techniques (such as greedy and divide-and-conquer) which work for special kinds of problems; Part III deals with the "sledgehammers" of the trade, techniques that are powerful and general: dynamic programming (a novel approach helps clarify this traditional stumbling block for students) and linear programming (a clean and intuitive treatment of the simplex algorithm, duality, and reductions to the basic problem). The final Part IV is about ways of dealing with hard problems: NP-completeness, various heuristics, as well as quantum algorithms, perhaps the most advanced and modern topic. As it happens, we end the story exactly where we started it, with Shor's quantum algorithm for factoring.

The book includes three additional undercurrents, in the form of three series of separate "boxes," strengthening the narrative (and addressing variations in the needs and interests of the students) while keeping the flow intact, pieces that provide historical context; descriptions of how the explained algorithms are used in practice (with emphasis on internet applications); and excursions for the mathematically sophisticated.

Many of our colleagues have made crucial contributions to this book. We are grateful for feedback from Dimitris Achlioptas, Dorit Aharanov, Mike Clancy, Jim Demmel, Monika Henzinger, Mike Jordan, Milena Mihail, Gene Myers, Dana Randall, Satish Rao, Tim Roughgarden, Jonathan Shewchuk, Martha Sideri, Alistair Sinclair, and David Wagner, all of whom beta tested early drafts. Satish Rao, Leonard Schulman, and Vijay Vazirani shaped the exposition of several key sections. Gene Myers, Satish Rao, Luca Trevisan, Vijay Vazirani, and Lofti Zadeh provided exercises. And finally, there are the students of UC Berkeley and, later, UC San Diego, who inspired this project, and who have seen it through its many incarnations.

方框目录

（中英文对照）

目 录

（中英文对照）

Chapter 0
Prologue

Look around you. Computers and networks are everywhere, enabling an intricate web of complex human activities: education, commerce, entertainment, research, manufacturing, health management, human communication, even war. Of the two main technological underpinnings of this amazing proliferation, one is obvious: the breathtaking pace with which advances in microelectronics and chip design have been bringing us faster and faster hardware.

This book tells the story of the other intellectual enterprise that is crucially fueling the computer revolution: *efficient algorithms*. It is a fascinating story.

Gather 'round and listen close.

0.1 Books and algorithms

Two ideas changed the world. In 1448 in the German city of Mainz a goldsmith named Johann Gutenberg discovered a way to print books by putting together movable metallic pieces. Literacy spread, the Dark Ages ended, the human intellect was liberated, science and technology triumphed, the Industrial Revolution happened. Many historians say we owe all this to typography. Imagine a world in which only an elite could read these lines! But others insist that the key development was not typography, but *algorithms*.

"typography"指凸版印刷术。其实早在11世纪的宋朝万历年间，中国的毕昇就发明了胶泥活板。约翰内斯·古腾堡（Johann Gutenberg）在15世纪发明的铅字活板，据说是受到中国印刷术的影响（参见西班牙历史学家冈萨雷斯·德·门多萨所著的《中华大帝国史》）。

Johann Gutenberg
1398–1468

© Corbis

Today we are so used to writing numbers in decimal, that it is easy to forget that Gutenberg would write the number 1448 as MCDXLVIII. How do you add two Roman numerals? What is MCDXLVIII + DCCCXII? (And just try to think about multiplying them.) Even a clever man like Gutenberg probably only knew how to add and subtract small numbers using his fingers; for anything more complicated he had to consult an abacus specialist.

The decimal system, invented in India around AD 600, was a revolution in quantitative reasoning: using only 10 symbols, even very large numbers could be written down compactly, and arithmetic could be done efficiently on them by following elementary steps. Nonetheless these ideas took a long time to spread, hindered by traditional barriers of language, distance, and ignorance. The most influential medium of transmission turned out to be a textbook, written in Arabic in the ninth century by a man who lived in Baghdad. Al Khwarizmi laid out the basic methods for adding, multiplying, and dividing numbers—even extracting square roots and calculating digits of π. These procedures were precise, unambiguous, mechanical, efficient, correct—in short, they were *algorithms*, a term coined to honor the wise man after the decimal system was finally adopted in Europe, many centuries later.

在Al Khwarizmi写的书中，十分强调求解问题的有条理的步骤。这是算法亘古不变的核心。算法就是求解问题的步骤，它必须是确定的（无歧义）、可行的（算法中的运算都是基本的，理论上能够由人用纸和笔完成）、有限的（算法必须能在有限步骤内实现）。此外，算法必须有输出（计算的结果），通常还至少有一个输入量。注意，不要把算法和计算机程序等同起来，后者只是描述前者的手段之一。我们还可以用流程图、形式语言甚至自然语言描述一个算法。

Since then, this decimal positional system and its numerical algorithms have played an enormous role in Western civilization. They enabled science and technology; they accelerated industry and commerce. And when, much later, the computer was finally designed, it explicitly embodied the positional system in its bits and words and arithmetic unit. Scientists everywhere then got busy developing more and more complex algorithms for all kinds of problems and inventing novel applications—ultimately changing the world.

0.2 Enter Fibonacci

Al Khwarizmi's work could not have gained a foothold in the West were it not for the efforts of one man: the 13th century Italian mathematician Leonardo Fibonacci, who saw the potential of the positional system and worked hard to develop it further and propagandize it.

But today Fibonacci is most widely known for his famous sequence of numbers

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

each the sum of its two immediate predecessors. More formally, the Fibonacci numbers F_n are generated by the simple rule

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0. \end{cases}$$

No other sequence of numbers has been studied as extensively, or applied to more fields: biology, demography, art, architecture, music, to name just a few. And, together with the powers of 2, it is computer science's favorite sequence.

In fact, the Fibonacci numbers grow *almost* as fast as the powers of 2: for example, F_{30} is over a million, and F_{100} is already 21 digits long! In general, $F_n \approx 2^{0.694n}$ (see Exercise 0.3).

But what is the precise value of F_{100}, or of F_{200}? Fibonacci himself would surely have wanted to know such things. To answer, we need an algorithm for computing the nth Fibonacci number.

Leonardo of Pisa (Fibonacci)
1170–1250

© Corbis

An exponential algorithm

One idea is to slavishly implement the recursive definition of F_n. Here is the resulting algorithm, in the "pseudocode" notation used throughout this book:

```
function fib1($n$)
if $n = 0$: return 0
if $n = 1$: return 1
return fib1($n-1$) + fib1($n-2$)
```

Whenever we have an algorithm, there are three questions we always ask about it:

1. Is it correct?
2. How much time does it take, as a function of n?
3. And can we do better?

The first question is moot here, as this algorithm is precisely Fibonacci's definition of F_n. But the second demands an answer. Let $T(n)$ be the number of *computer steps* needed to compute fib1(n); what can we say about this function? For starters, if n is less than 2, the procedure halts almost immediately, after just a couple of steps. Therefore,

$$T(n) \le 2 \quad \text{for } n \le 1.$$

For larger values of n, there are two recursive invocations of `fib1`, taking time $T(n-1)$ and $T(n-2)$, respectively, plus three computer steps (checks on the value of n and a final addition). Therefore,

$$T(n) = T(n-1) + T(n-2) + 3 \text{ for } n > 1.$$

Compare this to the recurrence relation for F_n: we immediately see that $T(n) \geq F_n$.

This is very bad news: the running time of the algorithm grows as fast as the Fibonacci numbers! $T(n)$ *is exponential in* n, which implies that the algorithm is impractically slow except for very small values of n.

Let's be a little more concrete about just how bad exponential time is. To compute F_{200}, the `fib1` algorithm executes $T(200) \geq F_{200} \geq 2^{138}$ elementary computer steps. How long this actually takes depends, of course, on the computer used. At this time, the fastest computer in the world is the NEC Earth Simulator, which clocks 40 trillion steps per second. Even on this machine, `fib1`(200) would take at least 2^{92} seconds. This means that, if we start the computation today, it would still be going long after the sun turns into a red giant star.

But technology is rapidly improving—computer speeds have been doubling roughly every 18 months, a phenomenon sometimes called *Moore's law*. With this extraordinary growth, perhaps `fib1` will run a lot faster on next year's machines. Let's see—the running time of `fib1`(n) is proportional to $2^{0.694n} \approx (1.6)^n$, so it takes 1.6 times longer to compute F_{n+1} than F_n. And under Moore's law, computers get roughly 1.6 times faster each year. So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}. And the year after, F_{102}. And so on: just one more Fibonacci number every year! Such is the curse of exponential time.

截止到2008年7月，世界上最快的超级计算机是位于美国Los Alamos国家实验室的IBM Roadrunner，它比NEC Earth Simulator（2002年）快近30倍。也就是说，用同样的时间执行fib1算法，Roadrunner只能比NEC Earth Simulator多算7个数（$1.6^7 \approx 30$）。

In short, our naive recursive algorithm is correct but hopelessly inefficient. *Can we do better?*

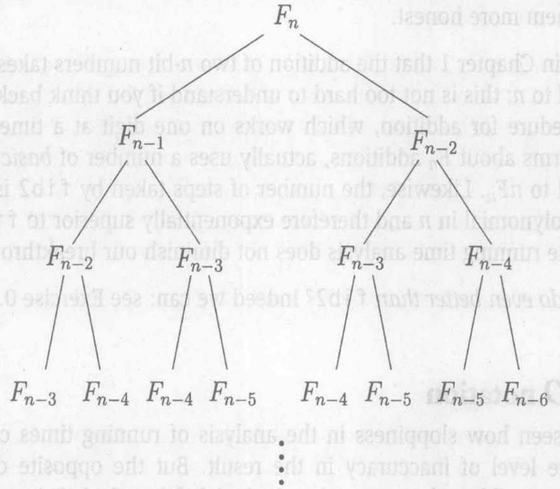
A polynomial algorithm

Let's try to understand why `fib1` is so slow. Figure 0.1 shows the cascade of recursive invocations triggered by a single call to `fib1`(n). Notice that many computations are repeated!

A more sensible scheme would store the intermediate results—the values $F_0, F_1, \ldots, F_{n-1}$—as soon as they become known.

```
function fib2($n$)
if $n = 0$: return 0
create an array f[0...$n$]
f[0] = 0, f[1] = 1
for $i = 2 \ldots n$:
   f[$i$] = f[$i-1$] + f[$i-2$]
return f[$n$]
```

Figure 0.1 **The proliferation of recursive calls in `fib1`.**



As with `fib1`, the correctness of this algorithm is self-evident because it directly uses the definition of F_n. How long does it take? The inner loop consists of a single computer step and is executed $n - 1$ times. Therefore the number of computer steps used by `fib2` is *linear in n*. From exponential we are down to *polynomial*, a huge breakthrough in running time. It is now perfectly reasonable to compute F_{200} or even $F_{200,000}$.[1]

算法fib2采取的策略从本质上说是"空间换时间"：用线性的空间存储以前计算过的结果，从而将复杂度由指数级下降到多项式级（线性）。空间和时间的权衡，是算法设计中的一个重要思想。

As we will see repeatedly throughout this book, the right algorithm makes all the difference.

More careful analysis

In our discussion so far, we have been counting the number of *basic computer steps* executed by each algorithm and thinking of these basic steps as taking a constant amount of time. This is a very useful simplification. After all, a processor's instruction set has a variety of basic primitives—branching, storing to memory, comparing numbers, simple arithmetic, and so on—and rather than distinguishing between these elementary operations, it is far more convenient to lump them together into one category.

But looking back at our treatment of Fibonacci algorithms, we have been too liberal with what we consider a basic step. It is reasonable to treat addition as a single computer step if small numbers are being added, 32-bit numbers say. But the nth Fibonacci number is about $0.694n$ bits long, and this can far exceed 32 as n grows.

[1]To better appreciate the importance of this dichotomy between exponential and polynomial algorithms, the reader may want to peek ahead to *the story of Sissa and Moore* in Chapter 8.

Arithmetic operations on arbitrarily large numbers cannot possibly be performed in a single, constant-time step. We need to audit our earlier running time estimates and make them more honest.

We will see in Chapter 1 that the addition of two n-bit numbers takes time roughly proportional to n; this is not too hard to understand if you think back to the grade-school procedure for addition, which works on one digit at a time. Thus `fib1`, which performs about F_n additions, actually uses a number of *basic steps* roughly proportional to nF_n. Likewise, the number of steps taken by `fib2` is proportional to n^2, still polynomial in n and therefore exponentially superior to `fib1`. This correction to the running time analysis does not diminish our breakthrough.

当n很大时，加法就不再是“原子”运算了。但对于大部分非数值算法（如图论、动态规划等），大家可以认为像加减乘除这样的基本运算只需要常数时间。这一点可以从算法需要解决的问题的描述中清楚地看出。

But can we do even better than `fib2`*?* Indeed we can: see Exercise 0.4.

0.3 Big-*O* notation

We've just seen how sloppiness in the analysis of running times can lead to an unacceptable level of inaccuracy in the result. But the opposite danger is also present: it is possible to be *too* precise. An insightful analysis is based on the right simplifications.

Expressing running time in terms of *basic computer steps* is already a simplification. After all, the time taken by one such step depends crucially on the particular processor and even on details such as caching strategy (as a result of which the running time can differ subtly from one execution to the next). Accounting for these architecture-specific minutiae is a nightmarishly complex task and yields a result that does not generalize from one computer to the next. It therefore makes more sense to seek an uncluttered, machine-independent characterization of an algorithm's efficiency. To this end, we will always express running time by counting the number of basic computer steps, as a function of the size of the input.

用基本指令条数来精确度量一个算法的效率是一件很困难的事情：从理论上分析极为繁琐，从实际中估计也颇为困难：不同的计算机采用的指令集、缓存策略等技术细节是不同的。我们需要寻找一种简便易行的、与具体机器无关的度量方法。它是以输入规模为自变量的函数。

And this simplification leads to another. Instead of reporting that an algorithm takes, say, $5n^3 + 4n + 3$ steps on an input of size n, it is much simpler to leave out lower-order terms such as $4n$ and 3 (which become insignificant as n grows), and even the detail of the coefficient 5 in the leading term (computers will be five times faster in a few years anyway), and just say that the algorithm takes time $O(n^3)$ (pronounced "big oh of n^3").

It is time to define this notation precisely. In what follows, think of $f(n)$ and $g(n)$ as the running times of two algorithms on inputs of size n.

> *Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ (which means that "f grows no faster than g") if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.*

Saying $f = O(g)$ is a very loose analog of "$f \leq g$." It differs from the usual notion of $\leq$ because of the constant c, so that for instance $10n = O(n)$. This constant also allows us to disregard what happens for small values of n. For example, suppose we