Graduate Texts in Mathematics

Neal Koblitz

A Course in Number Theory and Cryptography

Second Edition

数论和密码学教程 第2版

Springer

世界图书出版公司
www.wpcbj.com.cn

0156/Y16
2008.

Neal Koblitz

A Course in Number Theory and Cryptography

Second Edition

Springer

Neal Koblitz
Department of Mathematics
University of Washington
Seattle, WA 98195
USA

Editorial Board

S. Axler
Mathematics Department
San Francisco State University
San Francisco, CA 94132
USA
axler@sfu.edu

K.A. Ribet
Department of Mathematics
University of California, Berkeley
Berkeley, CA 94720-3840
USA
ribet@math.berkeley.edu

Mathematics Subject Classification (2000): 11-01, 11T71

With 5 Illustrations.

Library of Congress Cataloging-in-Publication Data
Koblitz, Neal, 1948–
A Course in number theory and cryptography / Neal Koblitz. — 2nd ed.
p. cm. — (Graduate texts in mathematics ; 114)
Includes bibliographical references and index.
ISBN 0-387-94293-9 (New York : acid-free). — ISBN 3-540-94293-9 (Berlin : acid-free
1. Number theory. 2. Cryptography. I. Title II. Series.
QA169.M33 1998
512'.7—dc20 94-11613

© 1994, 1987 Springer Science+Business Media, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, Inc., 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

This reprint has been authorized by Springer-Verlag (Berlin/Heidelberg/New York) for sale in the People's Republic of China only and not for export therefrom.

springeronline.com

Foreword

> ...both Gauss and lesser mathematicians may be justified in rejoicing that there is one science [number theory] at any rate, and that their own, whose very remoteness from ordinary human activities should keep it gentle and clean.
>
> — G. H. Hardy, *A Mathematician's Apology*, 1940

G. H. Hardy would have been surprised and probably displeased with the increasing interest in number theory for application to "ordinary human activities" such as information transmission (error-correcting codes) and cryptography (secret codes). Less than a half-century after Hardy wrote the words quoted above, it is no longer inconceivable (though it hasn't happened yet) that the N.S.A. (the agency for U.S. government work on cryptography) will demand prior review and clearance before publication of theoretical research papers on certain types of number theory.

In part it is the dramatic increase in computer power and sophistication that has influenced some of the questions being studied by number theorists, giving rise to a new branch of the subject, called "computational number theory."

This book presumes almost no background in algebra or number theory. Its purpose is to introduce the reader to arithmetic topics, both ancient and very modern, which have been at the center of interest in applications, especially in cryptography. For this reason we take an algorithmic approach, emphasizing estimates of the efficiency of the techniques that arise from the theory. A special feature of our treatment is the inclusion (Chapter VI) of some very recent applications of the theory of elliptic curves. Elliptic curves have for a long time formed a central topic in several branches of theoretical

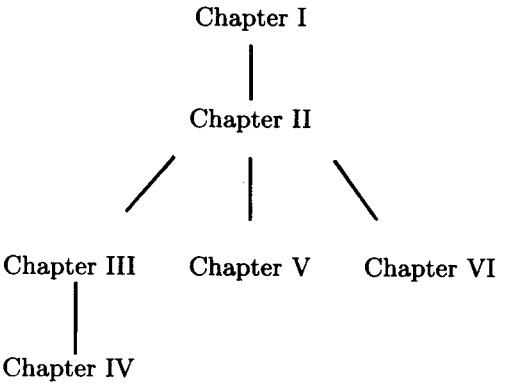
mathematics; now the arithmetic of elliptic curves has turned out to have potential practical applications as well.

Extensive exercises have been included in all of the chapters in order to enable someone who is studying the material outside of a formal course structure to solidify her/his understanding.

The first two chapters provide a general background. A student who has had no previous exposure to algebra (field extensions, finite fields) or elementary number theory (congruences) will find the exposition rather condensed, and should consult more leisurely textbooks for details. On the other hand, someone with more mathematical background would probably want to skim through the first two chapters, perhaps trying some of the less familiar exercises.

Depending on the students' background, it should be possible to cover most of the first five chapters in a semester. Alternately, if the book is used in a sequel to a one-semester course in elementary number theory, then Chapters III–VI would fill out a second-semester course.

The dependence relation of the chapters is as follows (if one overlooks some inessential references to earlier chapters in Chapters V and VI):



This book is based upon courses taught at the University of Washington (Seattle) in 1985–86 and at the Institute of Mathematical Sciences (Madras, India) in 1987. I would like to thank Gary Nelson and Douglas Lind for using the manuscript and making helpful corrections.

The frontispiece was drawn by Professor A. T. Fomenko of Moscow State University to illustrate the theme of the book. Notice that the coded decimal digits along the walls of the building are not random.

This book is dedicated to the memory of the students of Vietnam, Nicaragua and El Salvador who lost their lives in the struggle against U.S. aggression. The author's royalties from sales of the book will be used to buy mathematics and science books for the universities and institutes of those three countries.

Seattle, May 1987

此为试读,需要完整PDF请访问: www.ertongbook.com

Preface to the Second Edition

As the field of cryptography expands to include new concepts and techniques, the cryptographic applications of number theory have also broadened. In addition to elementary and analytic number theory, increasing use has been made of algebraic number theory (primality testing with Gauss and Jacobi sums, cryptosystems based on quadratic fields, the number field sieve) and arithmetic algebraic geometry (elliptic curve factorization, cryptosystems based on elliptic and hyperelliptic curves, primality tests based on elliptic curves and abelian varieties). Some of the recent applications of number theory to cryptography — most notably, the number field sieve method for factoring large integers, which was developed since the appearance of the first edition — are beyond the scope of this book. However, by slightly increasing the size of the book, we were able to include some new topics that help convey more adequately the diversity of applications of number theory to this exciting multidisciplinary subject.

The following list summarizes the main changes in the second edition.

- Several corrections and clarifications have been made, and many references have been added.
- A new section on zero-knowledge proofs and oblivious transfer has been added to Chapter IV.
- A section on the quadratic sieve factoring method has been added to Chapter V.
- Chapter VI now includes a section on the use of elliptic curves for primality testing.
- Brief discussions of the following concepts have been added: k–threshold schemes, probabilistic encryption, hash functions, the Chor–Rivest knapsack cryptosystem, and the U.S. government's new Digital Signature Standard.

Seattle, May 1994

Contents

I

Some Topics in Elementary Number Theory

Most of the topics reviewed in this chapter are probably well known to most readers. The purpose of the chapter is to recall the notation and facts from elementary number theory which we will need to have at our fingertips in our later work. Most proofs are omitted, since they can be found in almost any introductory textbook on number theory. One topic that will play a central role later — estimating the number of bit operations needed to perform various number theoretic tasks by computer — is not yet a standard part of elementary number theory textbooks. So we will go into most detail about the subject of time estimates, especially in §1.

1 Time estimates for doing arithmetic

Numbers in different bases. A nonnegative integer n written to the *base* b is a notation for n of the form $(d_{k-1}d_{k-2}\cdots d_1d_0)_b$, where the d's are *digits*, i.e., symbols for the integers between 0 and $b-1$; this notation means that $n = d_{k-1}b^{k-1} + d_{k-2}b^{k-2} + \cdots + d_1b + d_0$. If the first digit d_{k-1} is not zero, we call n a k-digit base-b number. Any number between b^{k-1} and b^k is a k-digit number to the base b. We shall omit the parentheses and subscript $(\cdots)_b$ in the case of the usual decimal system ($b = 10$) and occasionally in other cases as well, if the choice of base is clear from the context, especially when we're using the binary system ($b = 2$). Since it is sometimes useful to work in bases other than 10, one should get used to doing arithmetic in an arbitrary base and to converting from one base to another. We now review this by doing some examples.

Remarks. (1) Fractions can also be expanded in any base, i.e., they can be represented in the form $(d_{k-1}d_{k-2}\cdots d_1d_0.d_{-1}d_{-2}\cdots)_b$. (2) When $b > 10$ it is customary to use letters for the digits beyond 9. One could also use letters for *all* of the digits.

Example 1. (a) $(11001001)_2 = 201$.

(b) When $b = 26$ let us use the letters A—Z for the digits 0—25, respectively. Then $(\mathrm{BAD})_{26}$=679, whereas $(\mathrm{B.AD})_{26} = 1\frac{3}{676}$.

Example 2. Multiply 160 and 199 in the base 7. **Solution:**

$$\begin{array}{r} 316 \\ \underline{403} \\ 1254 \\ \underline{16030} \\ 161554 \end{array}$$

Example 3. Divide $(11001001)_2$ by $(100111)_2$, and divide $(\mathrm{HAPPY})_{26}$ by $(\mathrm{SAD})_{26}$.

Solution:

$$\begin{array}{r} 101\ \frac{110}{100111} \\ 100111\,\overline{|11001001} \\ \underline{100111} \\ 101101 \\ \underline{100111} \\ 110 \end{array} \qquad \begin{array}{r} \mathrm{KD}\ \frac{\mathrm{MLP}}{\mathrm{SAD}} \\ \mathrm{SAD}\,\overline{|\mathrm{HAPPY}} \\ \underline{\mathrm{GYBE}} \\ \mathrm{COLY} \\ \underline{\mathrm{CCAJ}} \\ \mathrm{MLP} \end{array}$$

Example 4. Convert 10^6 to the bases 2, 7 and 26 (using the letters A—Z as digits in the latter case).

Solution. To convert a number n to the base b, one first gets the last digit (the ones' place) by dividing n by b and taking the remainder. Then replace n by the quotient and repeat the process to get the second-to-last digit d_1, and so on. Here we find that

$$10^6 = (11110100001001000000)_2 = (11333311)_7 = (\mathrm{CEXHO})_{26}.$$

Example 5. Convert $\pi = 3.1415926\cdots$ to the base 2 (carrying out the computation 15 places to the right of the point) and to the base 26 (carrying out 3 places to the right of the point).

Solution. After taking care of the integer part, the fractional part is converted to the base b by multiplying by b, taking the integer part of the result as d_{-1}, then starting over again with the fractional part of what you now have, successively finding d_{-2}, d_{-3}, ... In this way one obtains:

$$3.1415926\cdots = (11.001001000011111\cdots)_2 = (\mathrm{D.DRS}\cdots)_{26}.$$

Number of digits. As mentioned before, an integer n satifying $b^{k-1} \leq n < b^k$ has k digits to the base b. By the definition of logarithms, this gives the following formula for the number of base-b digits (here "[ ]" denotes the greatest integer function):

$$\text{number of digits } = \left[log_b n\right] + 1 = \left[\frac{log\, n}{log\, b}\right] + 1,$$

where here (and from now on) "log" means the natural logarithm log_e.

Bit operations. Let us start with a very simple arithmetic problem, the addition of two binary integers, for example:

$$\begin{array}{r} {\scriptstyle 1\,1\,1\,1\,\cdot} \\ 1111000 \\ +\,\underline{0011110} \\ 10010110 \end{array}$$

Suppose that the numbers are both k bits long (the word "bit" is short for "binary digit"); if one of the two integers has fewer bits than the other, we fill in zeros to the left, as in this example, to make them have the same length. Although this example involves small integers (adding 120 to 30), we should think of k as perhaps being very large, like 500 or 1000.

Let us analyze in complete detail what this addition entails. Basically, we must repeat the following steps k times:

1. Look at the top and bottom bit, and also at whether there's a carry above the top bit.
2. If both bits are 0 and there is no carry, then put down 0 and move on.
3. If either (a) both bits are 0 and there is a carry, or (b) one of the bits is 0, the other is 1, and there is no carry, then put down 1 and move on.
4. If either (a) one of the bits is 0, the other is 1, and there is a carry, or else (b) both bits are 1 and there is no carry, then put down 0, put a carry in the next column, and move on.
5. If both bits are 1 and there is a carry, then put down 1, put a carry in the next column, and move on.

Doing this procedure once is called a *bit operation*. Adding two k-bit numbers requires k bit operations. We shall see that more complicated tasks can also be broken down into bit operations. The amount of time a computer takes to perform a task is essentially proportional to the number of bit operations. Of course, the constant of proportionality — the number of nanoseconds per bit operation — depends on the particular computer system. (This is an over-simplification, since the time can be affected by "administrative matters," such as accessing memory.) When we speak of estimating the "time" it takes to accomplish something, we mean finding an estimate for the number of bit operations required. In these estimates we shall neglect the time required for "bookkeeping" or logical steps other

此为试读,需要完整PDF请访问: www.ertongbook.com

than the bit operations; in general, it is the latter which takes by far the most time.

Next, let's examine the process of *multiplying* a k-bit integer by an ℓ-bit integer in binary. For example,

$$\begin{array}{r} 11101 \\ \underline{\quad 1101} \\ 11101 \\ 111010 \\ \underline{11101\quad\ } \\ 101111001 \end{array}$$

Suppose we use this familiar procedure to multiply a k-bit integer n by an ℓ-bit integer m. We obtain at most ℓ rows (one row fewer for each 0-bit in m), where each row consists of a copy of n shifted to the left a certain distance, i.e., with zeros put on at the end. Suppose there are $\ell' \leq \ell$ rows. Because we want to break down all our computations into bit operations, we cannot simultaneously add together all of the rows. Rather, we move down from the 2nd row to the ℓ'-th row, adding each new row to the partial sum of all of the earlier rows. At each stage, we note how many places to the left the number n has been shifted to form the new row. We copy down the right-most bits of the partial sum, and then add to n the integer formed from the rest of the partial sum — as explained above, this takes k bit operations. In the above example 11101×1101, after adding the first two rows and obtaining 10010001, we copy down the last three bits 001 and add the rest (i.e., 10010) to $n = 11101$. We finally take this sum $10010 + 11101 = 101111$ and append 001 to obtain 101111001, the sum of the $\ell' = 3$ rows.

This description shows that the multiplication task can be broken down into $\ell' - 1$ additions, each taking k bit operations. Since $\ell' - 1 < \ell' \leq \ell$, this gives us the simple bound

$$\text{Time(multiply integer } k \text{ bits long by integer } \ell \text{ bits long)} < k\ell.$$

We should make several observations about this derivation of an estimate for the number of bit operations needed to perform a binary multiplication. In the first place, as mentioned before, we counted only the number of bit operations. We neglected to include the time it takes to shift the bits in n a few places to the left, or the time it takes to copy down the right-most digits of the partial sum corresponding to the places through which n has been shifted to the left in the new row. In practice, the shifting and copying operations are fast in comparison with the large number of bit operations, so we can safely ignore them. In other words, we shall *define* a "time estimate" for an arithmetic task to be an upper bound for the number of bit operations, without including any consideration of shift operations,

changing registers ("copying"), memory access, etc. Note that this means that we would use the very same time estimate if we were multiplying a k-bit binary expansion of a fraction by an ℓ-bit binary expansion; the only additional feature is that we must note the location of the point separating integer from fractional part and insert it correctly in the answer.

In the second place, if we want to get a time estimate that is simple and convenient to work with, we should assume at various points that we're in the "worst possible case." For example, if the binary expansion of m has a lot of zeros, then ℓ' will be considerably less than ℓ. That is, we could use the estimate Time(multiply k-bit integer by ℓ-bit integer)$< k \cdot$(number of 1-bits in m). However, it is usually not worth the improvement (i.e., lowering) in our time estimate to take this into account, because it is more useful to have a simple uniform estimate that depends only on the size of m and n and not on the particular bits that happen to occur.

As a special case, we have: Time(multiply k-bit by k-bit)$< k^2$.

Finally, our estimate $k\ell$ can be written in terms of n and m if we remember the above formula for the number of digits, from which it follows that $k = [log_2 n] + 1 \leq \frac{log\, n}{log\, 2} + 1$ and $\ell = [log_2 m] + 1 \leq \frac{log\, m}{log\, 2} + 1$.

Example 6. Find an upper bound for the number of bit operations required to compute $n!$.

Solution. We use the following procedure. First multiply 2 by 3, then the result by 4, then the result of that by 5,..., until you get to n. At the $(j-1)$-th step $(j = 2, 3, \ldots, n-1)$, you are multiplying $j!$ by $j+1$. Hence you have $n-2$ steps, where each step involves multiplying a partial product (i.e., $j!$) by the next integer. The partial products will start to be very large. As a worst case estimate for the number of bits a partial product has, let's take the number of binary digits in the very last product, namely, in $n!$.

To find the number of bits in a product, we use the fact that the number of digits in the product of two numbers is either the sum of the number of digits in each factor or else 1 fewer than that sum (see the above discussion of multiplication). From this it follows that the product of n k-bit integers will have at most nk bits. Thus, if n is a k-bit integer — which implies that every integer less than n has at most k bits — then $n!$ has at most nk bits.

Hence, in each of the $n-2$ multiplications needed to compute $n!$, we are multiplying an integer with at most k bits (namely $j+1$) by an integer with at most nk bits (namely $j!$). This requires at most nk^2 bit operations. We must do this $n-2$ times. So the total number of bit operations is bounded by $(n-2)nk^2 = n(n-2)([log_2 n] + 1)^2$. Roughly speaking, the bound is approximately $n^2(log_2 n)^2$.

Example 7. Find an upper bound for the number of bit operations required to multiply a polynomial $\sum a_i x^i$ of degree $\leq n_1$ and a polynomial $\sum b_j x^j$ of degree $\leq n_2$ whose coefficients are positive integers $\leq m$. Suppose $n_2 \leq n_1$.

Solution. To compute $\sum_{i+j=\nu} a_i b_j$, which is the coefficient of x^ν in the product polynomial (here $0 \leq \nu \leq n_1 + n_2$) requires at most $n_2 + 1$ multi-

plications and n_2 additions. The numbers being multiplied are bounded by m, and the numbers being added are each at most m^2; but since we have to add the partial sum of up to n_2 such numbers we should take n_2m^2 as our bound on the size of the numbers being added. Thus, in computing the coefficient of x^ν the number of bit operations required is at most

$$(n_2+1)(log_2m+1)^2+n_2(log_2(n_2m^2)+1).$$

Since there are n_1+n_2+1 values of ν, our time estimate for the polynomial multiplication is

$$(n_1+n_2+1)\big((n_2+1)(log_2m+1)^2+n_2(log_2(n_2m^2)+1)\big).$$

A slightly less rigorous bound is obtained by dropping the 1's, thereby obtaining an expression having a more compact appearance:

$$\frac{n_2(n_1+n_2)}{log\,2}\left(\frac{(log\,m)^2}{log\,2}+(log\,n_2+2\log\,m)\right).$$

Remark. If we set $n=n_1\geq n_2$ and make the assumption that $m\geq 16$ and $m\geq\sqrt{n_2}$ (which usually holds in practice), then the latter expression can be replaced by the much simpler $4n^2(log_2m)^2$. This example shows that there is generally no single "right answer" to the question of finding a bound on the time to execute a given task. One wants a function of the bounds on the imput data (in this problem, n_1, n_2 and m) which is fairly simple and at the same time gives an upper bound which for most input data is more-or-less the same order of magnitude as the number of bit operations that turns out to be required in practice. Thus, for example, in Example 7 we would not want to replace our bound by, say, $4n^2m$, because for large m this would give a time estimate many orders of magnitude too large.

So far we have worked only with addition and multiplication of a k-bit and an ℓ-bit integer. The other two arithmetic operations — subtraction and division — have the same time estimates as addition and multiplication, respectively: Time(subtract k-bit from ℓ-bit)$\leq \max(k,\ell)$; Time(divide k-bit by ℓ-bit)$\leq kl$. More precisely, to treat subtraction we must extend our definition of a bit operation to include the operation of subtracting a 0- or 1-bit from another 0- or 1-bit (with possibly a "borrow" of 1 from the previous column). See Exercise 8.

To analyze division in binary, let us orient ourselves by looking at an illustration, such as the one in Example 3. Suppose $k\geq\ell$ (if $k<\ell$, then the division is trivial, i.e., the quotient is zero and the entire dividend is the remainder). Finding the quotient and remainder requires at most $k-\ell+1$ subtractions. Each subtraction requires ℓ or $\ell+1$ bit operations;-but in the latter case we know that the left-most column of the difference will always be a 0-bit, so we can omit that bit operation (thinking of it as "bookkeeping" rather than calculating). We similarly ignore other administrative details, such as the time required to compare binary integers (i.e., take just enough

bits of the dividend so that the resulting integer is greater than the divisor), carry down digits, etc. So our estimate is simply $(k-\ell+1)\ell$, which is $\leq k\ell$.

Example 8. Find an upper bound for the number of bit operations it takes to compute the binomial coefficient $\binom{n}{m}$.

Solution. Since $\binom{n}{m} = \binom{n}{n-m}$, without loss of generality we may assume that $m \leq n/2$. Let us use the following procedure to compute $\binom{n}{m} = n(n-1)(n-2)\cdots(n-m+1)/(2\cdot 3\cdots m)$. We have $m-1$ multiplications followed by $m-1$ divisions. In each case the maximum possible size of the first number in the multiplication or division is $n(n-1)(n-2)\cdots(n-m+1) < n^m$, and a bound for the second number is n. Thus, by the same argument used in the solution to Example 6, we see that a bound for the total number of bit operations is $2(m-1)m([log_2 n]+1)^2$, which for large m and n is essentially $2m^2(log_2 n)^2$.

We now discuss a very convenient notation for summarizing the situation with time estimates.

The big-O notation. Suppose that $f(n)$ and $g(n)$ are functions of the positive integers n which take *positive* (but not necessarily integer) values for all n. We say that $f(n) = O(g(n))$ (or simply that $f = O(g)$) if there exists a constant C such that $f(n)$ is always less than $C \cdot g(n)$. For example, $2n^2 + 3n - 3 = O(n^2)$ (namely, it is not hard to prove that the left side is always less than $3n^2$).

Because we want to use the big-O notation in more general situations, we shall give a more all-encompassing definition. Namely, we shall allow f and g to be functions of several variables, and we shall not be concerned about the relation between f and g for small values of n. Just as in the study of limits as $n \longrightarrow \infty$ in calculus, here also we shall only be concerned with large values of n.

Definition. Let $f(n_1, n_2, \ldots, n_r)$ and $g(n_1, n_2, \ldots, n_r)$ be two functions whose domains are subsets of the set of all r-tuples of positive integers. Suppose that there exist constants B and C such that whenever all of the n_j are greater than B the two functions are defined and positive, and $f(n_1, n_2, \ldots, n_r) < C\, g(n_1, n_2, \ldots, n_r)$. In that case we say that f is *bounded* by g and we write $f = O(g)$.

Note that the "=" in the notation $f = O(g)$ should be thought of as more like a "<" and the big-O should be thought of as meaning "some constant multiple."

Example 9. (a) Let $f(n)$ be *any* polynomial of degree d whose leading coefficient is positive. Then it is easy to prove that $f(n) = O(n^d)$. More generally, one can prove that $f = O(g)$ in any situation when $f(n)/g(n)$ has a finite limit as $n \longrightarrow \infty$.

(b) If ϵ is any positive number, no matter how small, then one can prove that $log\, n = O(n^\epsilon)$ (i.e., for large n, the log function is smaller than any power function, no matter how small the power). In fact, this follows because $lim_{n\to\infty} \frac{log\, n}{n^\epsilon} = 0$, as one can prove using l'Hôpital's rule.

(c) If $f(n)$ denotes the number k of binary digits in n, then it follows from the above formulas for k that $f(n) = O(\log n)$. Also notice that the same relation holds if $f(n)$ denotes the number of base-b digits, where b is any fixed base. On the other hand, suppose that the base b is not kept fixed but is allowed to increase, and we let $f(n, b)$ denote the number of base-b digits. Then we would want to use the relation $f(n, b) = O(\frac{\log n}{\log b})$.

(d) We have: $\text{Time}(n \cdot m) = O(\log n \cdot \log m)$, where the left hand side means the number of bit operations required to multiply n by m.

(e) In Exercise 6, we can write: $\text{Time}(n!) = O\big((n \log n)^2\big)$.

(f) In Exercise 7, we have:

$$\text{Time}\Big(\sum a_i x^i \cdot \sum b_j x^j\Big) = O\Big(n_1 n_2 \big((\log m)^2 + \log(\min(n_1, n_2))\big)\Big).$$

In our use, the functions $f(n)$ or $f(n_1, n_2, \ldots, n_r)$ will often stand for the amount of time it takes to perform an arithmetic task with the integer n or with the set of integers $n_1, n_2, \ldots, n_r$ as input. We will want to obtain fairly simple-looking functions $g(n)$ as our bounds. When we do this, however, we do not want to obtain functions $g(n)$ which are much larger than necessary, since that would give an exaggerated impression of how long the task will take (although, from a strictly mathematical point of view, it is not incorrect to replace $g(n)$ by any larger function in the relation $f = O(g)$).

Roughly speaking, the relation $f(n) = O(n^d)$ tells us that the function f increases approximately like the d-th power of the variable. For example, if $d = 3$, then it tells us that doubling n has the effect of increasing f by about a factor of 8. The relation $f(n) = O(\log^d n)$ (we write $\log^d n$ to mean $(\log n)^d$) tells us that the function increases approximately like the d-th power of the number of binary digits in n. That is because, up to a constant multiple, the number of bits is approximately $\log n$ (namely, it is within 1 of being $\log n / \log 2 = 1.4427 \log n$). Thus, for example, if $f(n) = O(\log^3 n)$, then doubling the number of bits in n (which is, of course, a much more drastic increase in the size of n than merely doubling n) has the effect of increasing f by about a factor of 8.

Note that to write $f(n) = O(1)$ means that the function f is bounded by some constant.

Remark. We have seen that, if we want to multiply two numbers of about the same size, we can use the estimate $\text{Time}(k\text{-bit} \cdot k\text{-bit}) = O(k^2)$. It should be noted that much work has been done on increasing the speed of multiplying two k-bit integers when k is large. Using clever techniques of multiplication that are much more complicated than the grade-school method we have been using, mathematicians have been able to find a procedure for multiplying two k-bit integers that requires only $O(k \log k \log\log k)$ bit operations. This is better than $O(k^2)$, and even better than $O(k^{1+\epsilon})$ for any $\epsilon > 0$, no matter how small. However, in what follows we shall always

be content to use the rougher estimates above for the time needed for a multiplication.

In general, when estimating the number of bit operations required to do something, the first step is to decide upon and write down an outline of a detailed procedure for performing the task. An explicit step-by-step procedure for doing calculations is called an *algorithm*. Of course, there may be many different algorithms for doing the same thing. One may choose to use the one that is easiest to write down, or one may choose to use the fastest one known, or else one may choose to compromise and make a trade-off between simplicity and speed. The algorithm used above for multiplying n by m is far from the fastest one known. But it is certainly a lot faster than repeated addition (adding n to itself m times).

Example 10. Estimate the time required to convert a k-bit integer to its representation in the base 10.

Solution. Let n be a k-bit integer written in binary. The conversion algorithm is as follows. Divide $10 = (1010)_2$ into n. The remainder — which will be one of the integers 0, 1, 10, 11, 100, 101, 110, 111, 1000, or 1001 — will be the ones digit d_0. Now replace n by the quotient and repeat the process, dividing that quotient by $(1010)_2$, using the remainder as d_1 and the quotient as the next number into which to divide $(1010)_2$. This process must be repeated a number of times equal to the number of decimal digits in n, which is $\left\lfloor \frac{\log n}{\log 10} \right\rfloor + 1 = O(k)$. Then we're done. (We might want to take our list of decimal digits, i.e., of remainders from all the divisions, and convert them to the more familiar notation by replacing 0, 1, 10, 11, ..., 1001 by 0, 1, 2, 3, ..., 9, respectively.) How many bit operations does this all take? Well, we have $O(k)$ divisions, each requiring $O(4k)$ operations (dividing a number with at most k bits by the 4-bit number $(1010)_2$). But $O(4k)$ is the same as $O(k)$ (constant factors don't matter in the big-O notation), so we conclude that the total number of bit operations is $O(k) \cdot O(k) = O(k^2)$. If we want to express this in terms of n rather than k, then since $k = O(\log n)$, we can write

$$\text{Time(convert } n \text{ to decimal)} = O(\log^2 n).$$

Example 11. Estimate the time required to convert a k-bit integer n to its representation in the base b, where b might be very large.

Solution. Using the same algorithm as in Example 10, except dividing now by the ℓ-bit integer b, we find that each division now takes longer (if ℓ is large), namely, $O(k\ell)$ bit operations. How many times do we have to divide? Here notice that the number of base-b digits in n is $O(k/\ell)$ (see Example 9(c)). Thus, the total number of bit operations required to do all of the necessary divisions is $O(k/\ell) \cdot O(k\ell) = O(k^2)$. This turns out to be the same answer as in Example 10. That is, our estimate for the conversion time does not depend upon the base to which we're converting (no matter how large it may be). This is because the greater time required to find each digit is offset by the fact that there are fewer digits to be found.

此为试读,需要完整PDF请访问: www.ertongbook.com