Neal Koblitz

A Course in Number Theory and Cryptography

Springer-Verlag

世界图书出版公司

Neal Koblitz

A Course in Number Theory and Cryptography

Springer-Verlag
世界图书出版公司

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

AMS Subject Classification: 10-01, 10H99

Library of Congress Cataloging-in-Pubication Data
Koblitz, Neal, 1948–
A course in number theory and cryptography.
(Graduate Texts in mathematics ; 114)
Bibliography: p.
Includes index.
1. Numbers, Theory of. 2. Cryptography.
I. Title. II. Series.
QA241.K672 1987 512'.7 87-16645

With 5 Illustrations

Reprinted by World Publishing Corporation, Beijing, 1994
for distribution and sale in The People's Republic of China only

ISBN 7—5062—0737—0

ISBN 0-387-96576-9 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-96576-9 Springer-Verlag Berlin Heidelberg New York

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):

$$\begin{tikzcd}
& \text{Chapter I} \arrow[d, no head] & \\
& \text{Chapter II} \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\
\text{Chapter III} \arrow[d, no head] & \text{Chapter V} & \text{Chapter VI} \\
\text{Chapter IV} & &
\end{tikzcd}$$

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 for national self-determination. 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

Contents

Chapter 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. An 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, especially when we're using the binary system ($b = 2$), if the choice of base is clear from the context. Since it is sometimes useful to work in other bases 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:

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

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}, $d_{-3}, \ldots$. In this way one obtains:

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

Number of digits. As mentioned before, a number 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:

$$\begin{array}{r} {\scriptstyle 1111} \\ 1111000 \\ +\ \underline{0011110} \\ 10010110 \end{array}$$

Suppose that the numbers are both k digits long; if one of the two integers has fewer digits 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 (the word "bit" is short for "binary digit") 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-digit 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.

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

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

In general, suppose we use this familiar procedure to multiply a k-bit integer n by an ℓ-bit integer m, where we suppose that $k \geq \ell$, i.e., we write the bigger number on top. 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. Thus, each row is an integer of at most $k + \ell$ bits. We may obtain our answer by first adding the second row to the first, then adding the third row to the result from the first addition, then adding the fourth row to the result of the second addition, and so on. In other words, we need at most ℓ (actually, at most $\ell - 1$) additions of at worst $k+\ell-$bit integers. (Notice that, even though carrying can cause the partial sum of the first j rows to be one bit longer than either the previous partial sum or the $j - th$ row that is being added to it, because of the way the rows are staggered it is easy to see that this can never bring the integers we're adding to a length greater than $k+\ell$ until the very last addition; our final answer will have either $k + \ell$ or $k + \ell + 1$ bits.) Since each addition takes at most $k + \ell$ bit operations, the total number of bit operations to get our answer is at most $\ell \cdot (k + \ell)$. Since $\ell \leq k$, we can give the simpler upper bound for the number of bit operations: $2k\ell$.

We should make several observations about this derivation of an estimate for the number of bit operations needed for performing a binary multiplication. In the first place, we neglected to include any estimate of the time it takes to shift the bits in n a few places to the left. However, in practice the shifting operation is fast in comparision with the large number of bit operations, so we can safely ignore it. In other words, we shall *define* the time it takes to perform an arithmetic task to be an upper bound for the number of bit operations, without including any consideration of shift operations, memory access, etc. Note that this means that we would use the very same time estimate if we were multiplying a k-digit binary expansion of a fraction by an ℓ-digit binary expansion; the only additional element is to 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, most of the additions involved in our multiplication problem will involve fewer than $k + \ell$ bits. But it is probably not worth the improvement (i.e., lowering) in our time estimate to take this into account.

Thus, time estimates do not have a single "right answer." It is correct to say that the time needed to multiply a k-bit number by an l-bit number is at most $(k + \ell)\ell$ bit operations. And it is also correct to say that it is at most $2k\ell$ bit operations. The first answer gives a lower value for the estimate of time, especially if ℓ is much less than k; but the second answer is simpler and a little easier to remember. We shall use the second estimate $2k\ell$. (One could also derive the estimate $k\ell$ by taking into account that, because of the increasing number of zeros to the right as you move from one row to the next, each addition involves only k nontrivial bit operations.)

Finally, our answer 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 $k \leq \frac{\log n}{\log 2} + 1$ and $\ell \leq \frac{\log m}{\log 2} + 1$.

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 in 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 6. (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 $f(n, b) = O(\frac{log\, n}{log\, b})$.

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 bunch of integers $n_1, n_2, \ldots, n_r$. 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 has the effect of increasing f by about a factor of 8. This is, of course, a much more drastic increase in the size of n than merely doubling n.

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

Let us now return to our time estimate for multiplying a k-bit integer by an ℓ-bit integer. We shall abbreviate the result of that discussion by writing:

$$\text{Time}(k - \text{bit} \times \ell - \text{bit}) = O(k\ell).$$

(We actually showed that the constant in the definition of big-O can be taken to be 2 in this case.) If we want to express our estimate in terms of the numbers n and m being multiplied rather than in terms of the number of bits in them, then we can write:

$$\text{Time}(n \times m) = O((\log n)(\log m)).$$

As a special case, if we want to multiply two numbers of about the same size, we can use the estimate

$$\text{Time}(k - \text{bit} \times k - \text{bit}) = O(k^2).$$

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. We did this earlier in the case of our multiplication problem. 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 easiest one 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).

So far we have discussed addition and multiplication in binary. Subtraction works very much like addition: we have the same estimate $O(k)$ for the amount of time required to subtract two k-bit integers. Division can be analyzed in much the same way as multiplication, with the result that it takes $O(k\ell)$ bit operations to obtain the quotient and remainder when a k-bit integer is divided by an ℓ-bit integer, where $k \geq \ell$ (of course, if $k < \ell$, then the quotient is zero and the remainder is all of the k-digit number).

Example 7. 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 $\left[\frac{\log n}{\log 10}\right] + 1 = O(k)$. 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 8. 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 7, 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 6(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 7. 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.

Example 9. Estimate the time required to compute $n!$.

Solution. We use the following algorithm. 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-th step you're multiplying $j!$ by $j + 1$. Here you have n multiplications (actually, $n - 2$), where each multiplication involves multiplying a partial product (i.e., $j!$) by the next integer. The partial product will start to be very large. As a worst case estimate for the number of bits it has, let's take the number of binary digits in the last product, namely, in $n!$.

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

Thus, in each of the $n-2$ multiplications in computing $n!$, we are multiplying an integer with at most k bits (namely $j+1$) by an integer with $O(nk)$ bits (namely $j!$). This requires $O(nk^2)$ bit operations. We must do this $n-2=O(n)$ times. So the total number of bit operations is $O(nk^2)\cdot O(n)=O(n^2k^2)$. Since $k=O(\log n)$, we end up with the estimate: Time(computing $n!$) $=O(n^2\log^2 n)$.

In concluding this section, we make a definition that is fundamental in the theory of algorithms and computer science.

Definition. An algorithm to perform a computation involving integers n_1, n_2, $\ldots, n_r$ of $k_1, k_2, \ldots, k_r$ bits, respectively, is said to be a *polynomial time* algorithm if there exist integers $d_1, d_2, \ldots, d_r$ such that the number of bit operations required to perform the algorithm is $O(k_1^{d_1}k_2^{d_2}\cdots k_r^{d_r})$.

Thus, the usual arithmetic operations $+$, $-$, $\times$, $\div$ are examples of polynomial time algorithms; so is conversion from one base to another. On the other hand, computation of $n!$ is not. (However, if one is satisfied with knowing $n!$ to only a certain number of significant figures, e.g., its first 1000 binary digits, then one can obtain that by a polynomial time algorithm using Stirling's approximation formula for $n!$.)

Exercises

1. Multiply $(212)_3$ by $(122)_3$.

2. Divide $(40122)_7$ by $(126)_7$.

3. Multiply the binary numbers 101101 and 11001, and divide 10011001 by 1011.

4. In the base 26, with digits A—Z representing 0—25, (a) multiply YES by NO, and (b) divide JQVXHJ by WE.

5. Write $e=2.7182818\cdots$ (a) in binary 15 places out to the right of the point, and (b) to the base 26 out 3 places beyond the point.

6. By a "pure repeating" fraction of "period" f in the base b, we mean a number between 0 and 1 whose base-b digits to the right of the point repeat in blocks of f. For example, 1/3 is pure repeating of period 1 and 1/7 is pure repeating of period 6 in the decimal system. Prove that a fraction c/d (in lowest terms) between 0 and 1 is pure repeating of period f in the base b if and only if b^f-1 is a multiple of d.

7. (a) The "hexadecimal" system means $b=16$ with the letters A–F representing the tenth through fifteenth digits, respectively. Divide $(131B6C3)_{16}$ by $(1A2F)_{16}$.

(b) Explain how to convert back and forth between binary and hexadecimal representations of an integer, and why the time required is far less than the general estimate given in Example 8 for converting from binary to base-b.

8. (a) Using the big-O notation, estimate in terms of a simple function of n the number of bit operations required to compute 3^n in binary.

(b) Do the same for n^n.

9. Estimate in terms of a simple function of n and N the number of bit operations required to compute N^n.

10. The following formula holds for the sum of the first n perfect squares:

$$\sum_{j=1}^{n} j^2 = n(n+1)(2n+1)/6.$$

(a) Using the big-O notation, estimate (in terms of n) the number of bit operations required to perform the computations in the left side of this equality.

(b) Estimate the number of bit operations required to perform the computations on the right in this equality.

11. The object of this exercise is to estimate as a function of n the number of bit operations required to compute the product of all prime numbers less than n. Here we suppose that we have already compiled an extremely long list containing all primes up to n.

(a) According to the Prime Number Theorem, the number of primes less than n (this is denoted $\pi(n)$) is asymptotic to $n/\log n$. This means that the following limit approaches 1 as $n \longrightarrow \infty$: $\lim \frac{\pi(n)}{n/\log n}$. Using the Prime Number Theorem, estimate the number of binary digits in the product of all primes less than n.

(b) Find a bound for the number of bit operations in one of the multiplications that's required in the computation of this product.

(c) Estimate the number of bit operations required to compute the product of all prime numbers less than n.

12. Let n be a very large integer written in binary. Find a simple algorithm that computes $[\sqrt{n}]$ in $O(\log^3 n)$ bit operations (here $[\ \ ]$ denotes the greatest integer function).

§2. Divisibility and the Euclidean algorithm

Divisors and divisibility. Given integers a and b, we say that a *divides* b (or "b *is divisible* by a") and we write $a|b$ if there exists an integer d such that $b = ad$. In that case we call a a *divisor* of b. Every integer $b > 1$ has at least two divisors: 1 and b. By a *proper divisor* of b we mean a divisor not equal to b itself, and by a *nontrivial divisor* of b we mean a divisor not equal to 1 or b. A *prime* number, by definition, is an integer greater than one which has no divisors other than 1 and itself; a number is called *composite* if it has at least one nontrivial divisor. The following properties of divisibility are easy to verify directly from the definition:

1. If $a|b$ and c is any integer, then $a|bc$.

2. If $a|b$ and $b|c$, then $a|c$.

3. If $a|b$ and $a|c$, then $a|b \pm c$.

If p is a prime number and α is a nonnegative integer, then we use the notation $p^\alpha || b$ to mean that p^α is the highest power of p dividing b, i.e., that $p^\alpha | b$ and $p^{\alpha+1} \nmid b$. In that case we say that p^α *exactly divides* b.

The *Fundamental Theorem of Arithmetic* states that any natural number n can be written uniquely (except for the order of factors) as a product of prime numbers. It is customary to write this factorization as a product of distinct primes to the appropriate powers, listing the primes in increasing order. For example, $4200 = 2^3 \cdot 3 \cdot 5^2 \cdot 7$.

Two consequences of the Fundamental Theorem (actually, equivalent assertions) are the following properties of divisibility:

4. If a prime number p divides ab, then either $p|a$ or $p|b$.

5. If $m|a$ and $n|a$, and if m and n have no divisors greater than 1 in common, then $mn|a$.

Another consequence of unique factorization is that it gives a systematic method for finding all divisors of n once n is written as a product of prime powers. Namely, any divisor d of n must be a product of the same primes raised to powers not exceeding the power that exactly divides n. That is, if $p^\alpha || n$, then $p^\beta || d$ for some β satisfying $0 \leq \beta \leq \alpha$. To find the divisors of 4200, for example, one takes 2 to the 0-, 1-, 2- or 3-power, multiplied by 3 to the 0- or 1-power, times 5 to the 0-, 1- or 2-power, times 7 to the 0- or 1- power. The number of possible divisors is thus the product of the number of possibilities for each prime power, which, in turn, is $\alpha + 1$. That is, a number $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_r^{\alpha_r}$ has $(\alpha_1 + 1)(\alpha_2 + 1) \cdots (\alpha_r + 1)$ different divisors. For example, there are 48 divisors of 4200.

Given two integers a and b, the *greatest common divisor* of a and b, denoted $g.c.d.(a, b)$ (or sometimes simply (a, b)) is the largest integer d dividing both a and b. It is not hard to show that another equivalent definition of $g.c.d.(a, b)$ is the following: it is the only positive integer d which divides a and b and is divisible by any other number which divides both a and b.

If you happen to have the prime factorization of a and b in front of you, then it's very easy to write down $g.c.d.(a, b)$. Simply take all primes which occur in both factorizations raised to the minimum of the two exponents. For example, comparing the factorization $10780 = 2^2 \cdot 5 \cdot 7^2 \cdot 11$ with the above factorization of 4200, we see that $g.c.d.(4200, 10780) = 2^2 \cdot 5 \cdot 7 = 140$.

One also occasionally uses the *least common multiple* of a and b, denoted $l.c.m.(a, b)$. It is the smallest positive integer that both a and b divide. If you have the factorization of a and b, then you can get $l.c.m.(a, b)$ by taking all of the primes