

Algorithms and Complexity

Herbert S. Wilf
University of Pennsylvania
Philadelphia, PA 19104-6395

Copyright Notice

Copyright 1994 by Herbert S. Wilf. This material may be reproduced for any educational purpose, multiple copies may be made for classes, etc. Charges, if any, for reproduced copies must be just enough to recover reasonable costs of reproduction. Reproduction for commercial purposes is prohibited. This cover page must be included in all distributed copies.

Internet Edition, Summer, 1994

This edition of Algorithms and Complexity is the file "`pub/wilf/AlgComp.ps.Z`" at the anonymous ftp site "`ftp.cis.upenn.edu`". It may be taken at no charge by all interested persons. Comments and corrections are welcome, and should be sent to `wilf@central.cis.upenn.edu`

Chapter 4: Algorithms in the Theory of Numbers

Number theory is the study of the properties of the positive integers. It is one of the oldest branches of mathematics, and one of the purest, so to speak. It has immense vitality, however, and we will see in this chapter and the next that parts of number theory are extremely relevant to current research in algorithms.

Part of the reason for this is that number theory enters into the analysis of algorithms, but that isn't the whole story.

Part of the reason is that many famous problems of number theory, when viewed from an algorithmic viewpoint (like, how do you decide whether or not a positive integer n is prime?) present extremely deep and attractive unsolved algorithmic problems. At least, they are unsolved if we regard the question as not just how to do these problems computationally, but how to do them as rapidly as possible.

But that's not the whole story either.

There are close connections between algorithmic problems in the theory of numbers, and problems in other fields, seemingly far removed from number theory. There is a unity between these seemingly diverse problems that enhances the already considerable beauty of any one of them. At least some of these connections will be apparent by the end of study of Chapter 5.

4.1 Preliminaries

We collect in this section a number of facts about the theory of numbers, for later reference.

If n and m are positive integers then to *divide* n by m is to find an integer $q \geq 0$ (the *quotient*) and an integer r (the *remainder*) such that $0 \leq r < m$ and $n = qm + r$.

If $r = 0$, we say that ' m divides n ,' or ' m is a divisor of n ,' and we write $m|n$. In any case the remainder r is also called ' n modulo m ,' and we write $r = n \bmod m$. Thus $4 = 11 \bmod 7$, for instance.

If n has no divisors other than $m = n$ and $m = 1$, then n is *prime*, else n is *composite*. Every positive integer n can be factored into primes, uniquely apart from the order of the factors. Thus $120 = 2^3 \cdot 3 \cdot 5$, and in general we will write

$$n = p_1^{a_1} p_2^{a_2} \cdots p_l^{a_l} = \prod_{i=1}^l p_i^{a_i}. \quad (4.1.1)$$

We will refer to (4.1.1) as the *canonical factorization* of n .

Many interesting and important properties of an integer n can be calculated from its canonical factorization. For instance, let $d(n)$ be the number of divisors of the integer n . The divisors of 6 are 1, 2, 3, 6, so $d(6) = 4$.

Can we find a formula for $d(n)$? A small example may help to clarify the method. Since $120 = 2^3 \cdot 3 \cdot 5$, a divisor of 120 must be of the form $m = 2^a 3^b 5^c$, in which a can have the values 0,1,2,3, b can be 0 or 1, and c can be 0 or 1. Thus there are 4 choices for a , 2 for b and 2 for c , so there are 16 divisors of 120.

In general, the integer n in (4.1.1) has exactly

$$d(n) = (1 + a_1)(1 + a_2) \cdots (1 + a_l) \quad (4.1.2)$$

divisors.

If m and n are nonnegative integers then their *greatest common divisor*, written $\gcd(n, m)$, is the integer g that

- (a) divides both m and n and
- (b) is divisible by every other common divisor of m and n .

Thus $\gcd(12, 8) = 4$, $\gcd(42, 33) = 3$, etc. If $\gcd(n, m) = 1$ then we say that n and m are *relatively prime*. Thus 27 and 125 are relatively prime (even though neither of them is prime).

If $n > 0$ is given, then $\phi(n)$ will denote the number of positive integers m such that $m \leq n$ and $\gcd(n, m) = 1$. Thus $\phi(6) = 2$, because there are only two positive integers ≤ 6 that are relatively prime to 6 (namely 1 and 5). $\phi(n)$ is called the Euler ϕ -function, or the Euler *totient* function.

Let's find a formula that expresses $\phi(n)$ in terms of the canonical factorization (4.1.1) of n .

We want to count the positive integers m for which $m \leq n$, and m is not divisible by any of the primes p_i that appear in (4.1.1). There are n possibilities for such an integer m . Of these we throw away n/p_1 of them because they are divisible by p_1 . Then we discard n/p_2 multiples of p_2 , etc. This leaves us with

$$n - n/p_1 - n/p_2 - \cdots - n/p_l \tag{4.1.3}$$

possible m 's.

But we have thrown away too much. An integer m that is a multiple of both p_1 and p_2 has been discarded at least twice. So let's correct these errors by adding

$$n/(p_1p_2) + n/(p_1p_3) + \cdots + n/(p_1p_l) + \cdots + n/(p_{l-1}p_l)$$

to (4.1.3).

The reader will have noticed that we added back too much, because an integer that is divisible by $p_1p_2p_3$, for instance, would have been re-entered at least twice. The 'bottom line' of counting too much, then too little, then too much, etc. is the messy formula

$$\begin{aligned} \phi(n) = & n - n/p_1 - n/p_2 - \cdots - n/p_l + n/(p_1p_2) + \cdots + n/(p_{l-1}p_l) \\ & - n/(p_1p_2p_3) - \cdots - n/(p_{l-2}p_{l-1}p_l) \\ & + \cdots + (-1)^l n/(p_1p_2 \cdots p_l). \end{aligned} \tag{4.1.4}$$

Fortunately (4.1.4) is identical with the much simpler expression

$$\phi(n) = n(1 - 1/p_1)(1 - 1/p_2) \cdots (1 - 1/p_l) \tag{4.1.5}$$

which the reader can check by beginning with (4.1.5) and expanding the product.

To calculate $\phi(120)$, for example, we first find the canonical factorization $120 = 2^3 \cdot 3 \cdot 5$. Then we apply (4.1.5) to get

$$\begin{aligned} \phi(120) &= 120(1 - 1/2)(1 - 1/3)(1 - 1/5) \\ &= 32. \end{aligned}$$

Thus, among the integers $1, 2, \dots, 120$, there are exactly 32 that are relatively prime to 120.

Exercises for section 4.1

1. Find a formula for the sum of the divisors of an integer n , expressed in terms of its prime divisors and their multiplicities.
2. How many positive integers are $\leq 10^{10}$ and have an odd number of divisors? Find a simple formula for the number of such integers that are $\leq n$.
3. If $\phi(n) = 2$ then what do you know about n ?
4. For which n is $\phi(n)$ odd?

4.2 The greatest common divisor

Let m and n be two positive integers. Suppose we divide n by m , to obtain a quotient q and a remainder r , with, of course, $0 \leq r < m$. Then we have

$$n = qm + r. \tag{4.2.1}$$

If g is some integer that divides both n and m then obviously g divides r also. Thus every common divisor of n and m is a common divisor of m and r . Conversely, if g is a common divisor of m and r then (4.2.1) shows that g divides n too.

It follows that $\gcd(n, m) = \gcd(m, r)$. If $r = 0$ then $n = qm$, and clearly, $\gcd(n, m) = m$.

If we use the customary abbreviation ‘ $n \bmod m$ ’ for r , the remainder in the division of n by m , then what we have shown is that

$$\gcd(n, m) = \gcd(m, n \bmod m).$$

This leads to the following recursive procedure for computing the g.c.d.

```
function gcd(n, m);
{finds gcd of given nonnegative integers n and m}
  if m = 0 then gcd := n else gcd := gcd(m, n mod m)
end.
```

The above is the famous ‘Euclidean algorithm’ for the g.c.d. It is one of the oldest algorithms known.

The reader is invited to write the Euclidean algorithm as a recursive program, and get it working on some computer. Use a recursive language, write the program more or less as above, and try it out with some large, healthy integers n and m .

The *gcd* program exhibits all of the symptoms of recursion. It calls itself with smaller values of its variable list. It begins with ‘if trivialcase then do trivialthing’ ($m = 0$), and this case is all-important because it’s the only way the procedure can stop itself.

If, for example, we want the g.c.d. of 13 and 21, we call the program with $n = 13$ and $m = 21$, and it then recursively calls itself with the following arguments:

$$(21, 13), (13, 8), (8, 5), (5, 3), (3, 2), (2, 1), (1, 0) \tag{4.2.2}$$

When it arrives at a call in which the ‘ m ’ is 0, then the ‘ n ,’ namely 1 in this case, is the desired g.c.d.

What is the input to the problem? The two integers n, m whose g.c.d. we want are the input, and the number of bits that are needed to input those two integers is $\Theta(\log n) + \Theta(\log m)$, namely $\Theta(\log mn)$. Hence $c \log mn$ is the length of the input bit string. Now let’s see how long the algorithm might run with an input string of that length.*

To measure the running time of the algorithm we need first to choose a unit of cost or work. Let’s agree that one unit of labor is the execution of a single ‘ $a \bmod b$ ’ operation. In this problem, an equivalent measure of cost would be the number of times the algorithm calls itself recursively. In the example (4.2.2) the cost was 7 units.

Lemma 4.2.1. *If $1 \leq b \leq a$ then $a \bmod b \leq (a - 1)/2$.*

Proof: Clearly $a \bmod b \leq b - 1$. Further,

$$\begin{aligned} a \bmod b &= a - \left\lfloor \frac{a}{b} \right\rfloor b \\ &\leq a - b. \end{aligned}$$

Thus $a \bmod b \leq \min(a - b, b - 1)$. Now we distinguish two cases.

First suppose $b \leq (a + 1)/2$. Then $b - 1 \leq a - b$ and so

$$\begin{aligned} a \bmod b &\leq b - 1 \\ &\leq \frac{a + 1}{2} - 1 \\ &= \frac{a - 1}{2} \end{aligned}$$

in this case.

Next, suppose $b > (a + 1)/2$. Then $a - b \leq b - 1$ and

$$a \bmod b \leq a - b < a - \frac{a + 1}{2} = \frac{a - 1}{2}$$

so the result holds in either case. ■

* In *Historia Mathematica* **21** (1994), 401-419, Jeffrey Shallit traces this analysis back to Pierre-Joseph-Étienne Finck, in 1841.

Theorem 4.2.1. (A worst-case complexity bound for the Euclidean algorithm) Given two positive integers a, b . The Euclidean algorithm will find their greatest common divisor after a cost of at most $\lfloor 2 \log_2 M \rfloor + 1$ integer divisions, where $M = \max(a, b)$.

Before we prove the theorem, let's return to the example $(a, b) = (13, 21)$ of the display (4.2.2). In that case $M = 21$ and $2 \log_2 M + 1 = 9.78 \dots$. The theorem asserts that the g.c.d. will be found after at most 9 operations. In fact it was found after 7 operations in that case.

Proof of theorem: Suppose first that $a \geq b$. The algorithm generates a sequence a_0, a_1, \dots where $a_0 = a, a_1 = b$, and

$$a_{j+1} = a_{j-1} \bmod a_j \quad (j \geq 1).$$

By lemma 4.2.1,

$$\begin{aligned} a_{j+1} &\leq \frac{a_{j-1} - 1}{2} \\ &\leq \frac{a_{j-1}}{2}. \end{aligned}$$

Then, by induction on j it follows that

$$a_{2j} \leq \frac{a_0}{2^j} \quad (j \geq 0)$$

$$a_{2j+1} \leq \frac{a_1}{2^j} \quad (j \geq 0)$$

and so,

$$a_r \leq 2^{-\lfloor r/2 \rfloor} M \quad (r = 0, 1, 2, \dots).$$

Obviously the algorithm has terminated if $a_r < 1$, and this will have happened when r is large enough so that $2^{-\lfloor r/2 \rfloor} M < 1$, *i.e.*, if $r > 2 \log_2 M$. If $a < b$ then after 1 operation we will be in the case ' $a \geq b$ ' that we have just discussed, and the proof is complete. ■

The upper bound in the statement of theorem 4.2.1 can be visualized as follows. The number $\log_2 M$ is almost exactly the number of bits in the binary representation of M (what is 'exactly' that number of bits?). Theorem 4.2.1 therefore asserts that we can find the g.c.d. of two integers in a number of operations that is at most a linear function of the number of bits that it takes to represent the two numbers. In brief, we might say that 'Time = $O(\text{bits})$,' in the case of Euclid's algorithm.

Exercises for section 4.2

- Write a nonrecursive program, in Basic or Fortran, for the g.c.d. Write a recursive program, in Pascal or a recursive language of your choice, for the g.c.d.
- Choose 1000 pairs of integers (n, m) , at random between 1 and 1000. For each pair, compute the g.c.d. using a recursive program and a nonrecursive program.
 - Compare the execution times of the two programs.
 - There is a theorem to the effect that the probability that two random integers have g.c.d. = 1 is $6/\pi^2$. What, precisely, do you think that this theorem means by 'the probability that ...'? What percentage of the 1000 pairs that you chose had g.c.d. = 1? Compare your observed percentage with $100 \cdot (6/\pi^2)$.
- Find out when Euclid lived, and with exactly what words he described his algorithm.
- Write a program that will light up a pixel in row m and column n of your CRT display if and only if $\text{gcd}(m, n) = 1$. Run the program with enough values of m and n to fill your screen. If you see any interesting visual patterns, try to explain them mathematically.
- Show that if m and n have a total of B bits, then Euclid's algorithm will not need more than $2B + 3$ operations before reaching termination.

6. Suppose we have two positive integers m, n , and we have factored them completely into primes, in the form

$$m = \prod p_i^{a_i}; \quad n = \prod q_i^{b_i}.$$

How would you calculate $\gcd(m, n)$ from the above information? How would you calculate the least common multiple (lcm) of m and n from the above information? Prove that $\gcd(m, n) = mn/lcm(m, n)$.

7. Calculate $\gcd(102131, 56129)$ in two ways: use the method of exercise 6 above, then use the Euclidean algorithm. In each case count the total number of arithmetic operations that you had to do to get the answer.

8. Let F_n be the n^{th} Fibonacci number. How many operations will be needed to compute $\gcd(F_n, F_{n-1})$ by the Euclidean algorithm? What is $\gcd(F_n, F_{n-1})$?

4.3 The extended Euclidean algorithm

Again suppose n, m are two positive integers whose g.c.d. is g . Then we can always write g in the form

$$g = tn + um \tag{4.3.1}$$

where t and u are integers. For instance, $\gcd(14, 11) = 1$, so we can write $1 = 14t + 11u$ for integers t, u . Can you spot integers t, u that will work? One pair that does the job is $(4, -5)$, and there are others (can you find all of them?).

The extended Euclidean algorithm finds not only the g.c.d. of n and m , it also finds a pair of integers t, u that satisfy (4.3.1). One ‘application’ of the extended algorithm is that we will obtain an inductive proof of the *existence* of t, u , that is not immediately obvious from (4.3.1) (see exercise 1 below). While this hardly rates as a ‘practical’ application, it represents a very important feature of recursive algorithms. We might say, rather generally, that the following items go hand-in-hand:

Recursive algorithms Inductive proofs Complexity analyses by recurrence formulas

If we have a recursive algorithm, then it is natural to prove the validity of the algorithm by mathematical induction. Conversely, inductive proofs of theorems often (not always, alas!) yield recursive algorithms for the construction of the objects that are being studied. The complexity analysis of a recursive algorithm will use recurrence formulas, in a natural way. We saw that already in the analysis that proved theorem 4.2.1.

Now let’s discuss the extended algorithm. Input to it will be two integers n and m . Output from it will be $g = \gcd(n, m)$ and two integers t and u for which (4.3.1) is true.

A single step of the original Euclidean algorithm took us from the problem of finding $\gcd(n, m)$ to $\gcd(m, n \bmod m)$. Suppose, inductively, that we not only know $g = \gcd(m, n \bmod m)$ but we also know the coefficients t', u' for the equation

$$g = t'm + u'(n \bmod m). \tag{4.3.2}$$

Can we get out, at the next step, the corresponding coefficients t, u for (4.3.1)? Indeed we can, by substituting in (4.3.2) the fact that

$$n \bmod m = n - \left\lfloor \frac{n}{m} \right\rfloor m \tag{4.3.3}$$

we find that

$$\begin{aligned} g &= t'm + u'(n - \left\lfloor \frac{n}{m} \right\rfloor m) \\ &= u'n + (t' - u' \left\lfloor \frac{n}{m} \right\rfloor)m. \end{aligned} \tag{4.3.4}$$

Hence the rule by which t', u' for equation (4.3.2) transform into t, u for equation (4.3.1) is that

$$\begin{aligned} t &= u' \\ u &= t' - \left\lfloor \frac{n}{m} \right\rfloor u'. \end{aligned} \tag{4.3.5}$$

We can now formulate recursively the extended Euclidean algorithm.

```

procedure gcdext(n, m, g, t, u);
{computes g.c.d. of n and m, and finds
  integers t, u that satisfy (4.3.1)}
  if m = 0 then
    g := n; t := 1; u := 0
  else
    gcdext(m, n mod m, g, t, u);
    s := u;
    u := t - [ n/m ] u;
    t := s
end.{gcdext}

```

It is quite easy to use the algorithm above to make a proof of the main mathematical result of this section (see exercise 1), which is

Theorem 4.3.1. *Let m and n be given integers, and let g be their greatest common divisor. Then there exist integers t, u such that $g = tm + un$. ■*

An immediate consequence of the algorithm and the theorem is the fact that finding inverses modulo a given integer is an easy computational problem. We will need to refer to that fact in the sequel, so we state it as

Corollary 4.3.1. *Let m and n be given positive integers, and let g be their g.c.d. Then m has a multiplicative inverse modulo n if and only if $g = 1$. In that case, the inverse can be computed in polynomial time.*

Proof: By the extended Euclidean algorithm we can find, in linear time, integers t and u such that $g = tm + un$. But this last equation says that $tm \equiv g \pmod{n}$. If $g = 1$ then it is obvious that t is the inverse mod n of m . If $g > 1$ then there exists no t such that $tm \equiv 1 \pmod{n}$ since $tm = 1 + rn$ implies that the g.c.d. of m and n is 1. ■

We will now trace the execution of *gcdext* if it is called with $(n, m) = (14, 11)$. The routine first replaces $(14, 11)$ by $(11, 3)$ and calls itself. Then it calls itself successively with $(3, 2)$, $(2, 1)$ and $(1, 0)$. When it executes with $(n, m) = (1, 0)$ it encounters the ‘if $m = 0$ ’ statement, so it sets $g := 1, t := 1, u := 0$.

Now it can complete the execution of the call with $(n, m) = (2, 1)$, which has so far been pending. To do this it sets

$$\begin{aligned} u &:= t - [n/m] u = 1 \\ t &:= 0. \end{aligned}$$

The call with $(n, m) = (2, 1)$ is now complete. The call to the routine with $(n, m) = (3, 2)$ has been in limbo until just this moment. Now that the $(2, 1)$ call is finished, the $(3, 2)$ call executes and finds

$$\begin{aligned} u &:= 0 - [3/2] 1 = 1 \\ t &:= 1. \end{aligned}$$

The call to the routine with $(n, m) = (11, 3)$ has so far been languishing, but its turn has come. It computes

$$\begin{aligned} u &:= 1 - [11/3] (-1) = 4 \\ t &:= -1. \end{aligned}$$

Finally, the original call to *gcdext* from the user, with $(n, m) = (14, 11)$, can be processed. We find

$$\begin{aligned} u &:= (-1) - [14/11] 4 = -5 \\ t &:= 4. \end{aligned}$$

Therefore, to the user, *gcdext* returns the values $g = 1, u = -5, t = 4$, and we see that the procedure has found the representation (4.3.1) in this case. The importance of the ‘trivial case’ where $m = 0$ is apparent.

Exercises for section 4.3

1. Give a complete formal proof of theorem 4.3.1. Your proof should be by induction (on what?) and should use the extended Euclidean algorithm.
2. Find integers t, u such that
 - (a) $1 = 4t + 7u$
 - (b) $1 = 24t + 35u$
 - (c) $5 = 65t + 100u$
3. Let a_1, \dots, a_n be positive integers.
 - (a) How would you compute $\gcd(a_1, \dots, a_n)$?
 - (b) Prove that there exist integers t_1, \dots, t_n such that

$$\gcd(a_1, \dots, a_n) = t_1 a_1 + t_2 a_2 + \dots + t_n a_n.$$

- (c) Give a recursive algorithm for the computation of t_1, \dots, t_n in part (b) above.
4. If $r = ta + ub$, where r, a, b, u, v are all integers, must $r = \gcd(a, b)$? What, if anything, can be said about the relationship of r to $\gcd(a, b)$?
 5. Let (t_0, u_0) be one pair of integers t, u for which $\gcd(a, b) = ta + ub$. Find *all* such pairs of integers, a and b being given.
 6. Find *all* solutions to exercises 2(a)-(c) above.
 7. Find the multiplicative inverse of 49 modulo 73, using the extended Euclidean algorithm.
 8. If *gcdext* is called with $(n, m) = (98, 30)$, draw a picture of the complete tree of calls that will occur during the recursive execution of the program. In your picture show, for each recursive call in the tree, the values of the input parameters to that call and the values of the output variables that were returned by that call.

4.4 Primality testing

In Chapter 1 we discussed the important distinction between algorithms that run in polynomial time *vs.* those that may require exponential time. Since then we have seen some fast algorithms and some slow ones. In the network flow problem the complexity of the MPM algorithm was $O(V^3)$, a low power of the size of the input data string, and the same holds true for the various matching and connectivity problems that are special cases of the network flow algorithm.

Likewise, the Fast Fourier Transform is really Fast. It needs only $O(n \log n)$ time to find the transform of a sequence of length n if n is a power of two, and only $O(n^2)$ time in the worst case, where n is prime.

In both of those problems we were dealing with computational situations near the low end of the complexity scale. It is feasible to do a Fast Fourier Transform on, say, 1000 data points. It is feasible to calculate maximum flows in networks with 1000 vertices or so.

On the other hand, the recursive computation of the chromatic polynomial in section 2.3 of Chapter 2 was an example of an algorithm that might use exponential amounts of time.

In this chapter we will meet another computational question for which, to date, no one has ever been able to provide a polynomial-time algorithm, nor has anyone been able to prove that such an algorithm does not exist.

The problem is just this: *Given a positive integer n . Is n prime?*

The reader should now review the discussion in Example 3 of section 0.2. In that example we showed that the obvious methods of testing for primality are slow in the sense of complexity theory. That is, we do an amount of work that is an exponentially growing function of the length of the input bit string if we use one of those methods. So this problem, which seems like a ‘pushover’ at first glance, turns out to be extremely difficult.

Although it is not known if a polynomial-time primality testing algorithm exists, remarkable progress on the problem has been made in recent years.

One of the most important of these advances was made independently and almost simultaneously by Solovay and Strassen, and by Rabin, in 1976-7. These authors took the imaginative step of replacing ‘certainly’ by ‘probably,’ and they devised what should be called a probabilistic compositeness (an integer is *composite* if it is *not* prime) test for integers, that runs in polynomial time.

Here is how the test works. First choose a number b uniformly at random, $1 \leq b \leq n - 1$. Next, subject the pair (b, n) to a certain test, called a *pseudoprimalty test*, to be described below. The test has two possible outcomes: either the number n is correctly declared to be composite or the test is inconclusive.

If that were the whole story it would be scarcely have been worth the telling. Indeed the test ‘Does b divide n ?’ already would perform the function stated above. However, it has a low probability of success even if n is composite, and if the answer is ‘No,’ we would have learned virtually nothing.

The additional property that the test described below has, not shared by the more naive test ‘Does b divide n ?,’ is that *if n is composite, the chance that the test will declare that result is at least $1/2$.*

In practice, for a given n we would apply the test 100 times using 100 numbers b_i that are independently chosen at random in $[1, n - 1]$. If n is composite, the probability that it will be declared composite at least once is at least $1 - 2^{-100}$, and these are rather good odds. Each test would be done in quick polynomial time. If n is not found to be composite after 100 trials, and if certainty is important, then it would be worthwhile to subject n to one of the nonprobabilistic primality tests in order to dispel all doubt.

It remains to describe the test to which the pair (b, n) is subjected, and to prove that it detects compositeness with probability $\geq 1/2$.

Before doing this we mention another important development. A more recent primality test, due to Adleman, Pomerance and Rumely in 1983, is completely deterministic. That is, given n it will *surely* decide whether or not n is prime. The test is more elaborate than the one that we are about to describe, and it runs in tantalizingly close to polynomial time. In fact it was shown to run in time

$$O((\log n)^{c \log \log \log n})$$

for a certain constant c . Since the number of bits of n is a constant multiple of $\log n$, this latter estimate is of the form

$$O((Bits)^{c \log \log Bits}).$$

The exponent of ‘*Bits*,’ which would be constant in a polynomial time algorithm, in fact grows extremely slowly as n grows. This is what was referred to as ‘tantalizingly close’ to polynomial time, earlier.

It is important to notice that in order to prove that a number is not prime, it is certainly *sufficient* to find a nontrivial divisor of that number. It is not *necessary* to do that, however. All we are asking for is a ‘yes’ or ‘no’ answer to the question ‘is n prime?.’ If you should find it discouraging to get only the answer ‘no’ to the question ‘Is 7122643698294074179 prime?,’ without getting any of the factors of that number, then what you want is a fast algorithm for the factorization problem.

In the test that follows, the decision about the compositeness of n will be reached without a knowledge of any of the factors of n . This is true of the Adleman, Pomerance, Rumely test also. The question of finding a factor of n , or all of them, is another interesting computational problem that is under active investigation. Of course the factorization problem is at least as hard as finding out if an integer is prime, and so no polynomial-time algorithm is known for it either. Again, there are probabilistic algorithms for the factorization problem just as there are for primality testing, but in the case of the factorization problem, even they don’t run in polynomial-time.

In section 4.9 we will discuss a probabilistic algorithm for factoring large integers, after some motivation in section 4.8, where we remark on the connection between computationally intractable problems and cryptography. Specifically, we will describe one of the ‘Public Key’ data encryption systems whose usefulness stems directly from the difficulty of factoring large integers.

Isn't it amazing that in this technologically enlightened age we still don't know how to find a divisor of a whole number quickly?

4.5 Interlude: the ring of integers modulo n

In this section we will look at the arithmetic structure of the integers modulo some fixed integer n . These results will be needed in the sequel, but they are also of interest in themselves and have numerous applications.

Consider the ring whose elements are $0, 1, 2, \dots, n-1$ and in which we do addition, subtraction, and multiplication modulo n . This ring is called \mathbf{Z}_n . For example, in Table 4.5.1 we show the addition and multiplication tables of \mathbf{Z}_6 .

	+	0	1	2	3	4	5		*	0	1	2	3	4	5
0		0	1	2	3	4	5		0	0	0	0	0	0	0
1		1	2	3	4	5	0		1	0	1	2	3	4	5
2		2	3	4	5	0	1		2	0	2	4	0	2	4
3		3	4	5	0	1	2		3	0	3	0	3	0	3
4		4	5	0	1	2	3		4	0	4	2	0	4	2
5		5	0	1	2	3	4		5	0	5	4	3	2	1

Table 4.5.1: Arithmetic in the ring \mathbf{Z}_6

Notice that while \mathbf{Z}_n is a ring, it certainly need not be a *field*, because there will usually be some noninvertible elements. Reference to Table 4.5.1 shows that 2, 3, 4 have no multiplicative inverses in \mathbf{Z}_6 , while 1, 5 do have such inverses. The difference, of course, stems from the fact that 1 and 5 are relatively prime to the modulus 6 while 2, 3, 4 are not. We learned, in corollary 4.3.1, that an element m of \mathbf{Z}_n is invertible if and only if m and n are relatively prime.

The invertible elements of \mathbf{Z}_n form a multiplicative *group*. We will call that group the *group of units* of \mathbf{Z}_n and will denote it by U_n . It has exactly $\phi(n)$ elements, by lemma 4.5.1, where ϕ is the Euler function of (4.1.5).

The multiplication table of the group U_{18} is shown in Table 4.5.2.

	*	1	5	7	11	13	17
1		1	5	7	11	13	17
5		5	7	17	1	11	13
7		7	17	13	5	1	11
11		11	1	5	13	17	7
13		13	11	1	17	7	5
17		17	13	11	7	5	1

Table 4.5.2: Multiplication modulo 18

Notice that U_{18} contains $\phi(18) = 6$ elements, that each of them has an inverse and that each row (column) of the multiplication table contains a permutation of all of the group elements.

Let's look at the table a little more closely, with a view to finding out if the group U_{18} is *cyclic*. In a cyclic group there is an element a whose powers $1, a, a^2, a^3, \dots$ run through all of the elements of the group.

If we refer to the table again, we see that in U_{18} the powers of 5 are $1, 5, 7, 17, 13, 11, 1, \dots$. Thus the *order* of the group element 5 is equal to the order of the group, and the powers of 5 exhaust all group elements. The group U_{18} is indeed cyclic, and 5 is a generator of U_{18} .

A number (like 5 in the example) whose powers run through all elements of U_n is called a *primitive root* modulo n . Thus 5 is a primitive root modulo 18. The reader should now find, from Table 4.5.2, *all* of the primitive roots modulo 18.

Alternatively, since the order of a group element must always divide the order of the group, every element of U_n has an order that divides $\phi(n)$. The primitive roots are exactly the elements, if they exist, of maximum possible order $\phi(n)$.

We pause to note two corollaries of these remarks, namely

Theorem 4.5.1 ('Fermat's theorem'). *For every integer b that is relatively prime to n we have*

$$b^{\phi(n)} \equiv 1 \pmod{n}. \quad (4.5.1)$$

In particular, if n is a prime number then $\phi(n) = n - 1$, and we have

Theorem 4.5.2 ('Fermat's little theorem'). *If n is prime, then for all $b \not\equiv 0 \pmod{n}$ we have $b^{n-1} \equiv 1 \pmod{n}$.*

It is important to know which groups U_n are cyclic, *i.e.*, which integers n have primitive roots. The answer is given by

Theorem 4.5.3. *An integer n has a primitive root if and only if $n = 2$ or $n = 4$ or $n = p^a$ (p an odd prime) or $n = 2p^a$ (p an odd prime). Hence, the groups U_n are cyclic for precisely such values of n . ■*

The proof of theorem 4.5.3 is a little lengthy and is omitted. It can be found, for example, in the book of LeVeque that is cited at the end of this chapter.

According to theorem 4.5.3, for example, U_{18} is cyclic, which we have already seen, and U_{12} is not cyclic, which the reader should check.

Further, we state as an immediate consequence of theorem 4.5.3,

Corollary 4.5.3. *If n is an odd prime, then U_n is cyclic, and in particular the equation $x^2 = 1$, in U_n , has only the solutions $x = \pm 1$.*

Next we will discuss the fact that if the integer n can be factored in the form $n = p_1^{a_1} p_2^{a_2} \cdots p_r^{a_r}$ then the full ring \mathbf{Z}_n can also be factored, in a certain sense, as a 'product' of $\mathbf{Z}_{p_i^{a_i}}$.

Let's take \mathbf{Z}_6 as an example. Since $6 = 2 \cdot 3$, we expect that somehow $\mathbf{Z}_6 = \mathbf{Z}_2 \otimes \mathbf{Z}_3$. What this means is that we consider *ordered pairs* x_1, x_2 , where $x_1 \in \mathbf{Z}_2$ and $x_2 \in \mathbf{Z}_3$.

Here is how we do the arithmetic with the ordered pairs.

First, $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$, in which the two '+' signs on the right are different: the first ' $x_1 + y_1$ ' is done in \mathbf{Z}_2 while the ' $x_2 + y_2$ ' is done in \mathbf{Z}_3 .

Second, $(x_1, x_2) \cdot (y_1, y_2) = (x_1 \cdot y_1, x_2 \cdot y_2)$, in which the two multiplications on the right side are different: the ' $x_1 \cdot y_1$ ' is done in \mathbf{Z}_2 and the ' $x_2 \cdot y_2$ ' in \mathbf{Z}_3 .

Therefore the 6 elements of \mathbf{Z}_6 are

$$(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2).$$

A sample of the addition process is

$$\begin{aligned} (0, 2) + (1, 1) &= (0 + 1, 2 + 1) \\ &= (1, 0) \end{aligned}$$

where the addition of the first components was done modulo 2 and of the second components was done modulo 3.

A sample of the multiplication process is

$$\begin{aligned} (1, 2) \cdot (1, 2) &= (1 \cdot 1, 2 \cdot 2) \\ &= (1, 1) \end{aligned}$$

in which multiplication of the first components was done modulo 2 and of the second components was done modulo 3.

In full generality we can state the factorization of \mathbf{Z}_n as

Theorem 4.5.4. Let $n = p_1^{a_1} p_2^{a_2} \cdots p_r^{a_r}$. The mapping which associates with each $x \in \mathbf{Z}_n$ the r -tuple (x_1, x_2, \dots, x_r) , where $x_i = x \bmod p_i^{a_i}$ ($i = 1, r$), is a ring isomorphism of \mathbf{Z}_n with the ring of r -tuples (x_1, x_2, \dots, x_r) in which

- (a) $x_i \in \mathbf{Z}_{p_i^{a_i}}$ ($i = 1, r$) and
- (b) $(x_1, \dots, x_r) + (y_1, \dots, y_r) = (x_1 + y_1, \dots, x_r + y_r)$ and
- (c) $(x_1, \dots, x_r) \cdot (y_1, \dots, y_r) = (x_1 \cdot y_1, \dots, x_r \cdot y_r)$
- (d) In (b), the i^{th} ‘+’ sign on the right side is the addition operation of $\mathbf{Z}_{p_i^{a_i}}$ and in (c) the i^{th} ‘ \cdot ’ sign is the multiplication operation of $\mathbf{Z}_{p_i^{a_i}}$, for each $i = 1, 2, \dots, r$.

The proof of theorem 4.5.4 follows at once from the famous

Theorem 4.5.5 (‘The Chinese Remainder Theorem’). Let m_i ($i = 1, r$) be pairwise relatively prime positive integers, and let

$$M = m_1 m_2 \cdots m_r.$$

Then the mapping that associates with each integer x ($0 \leq x \leq M - 1$) the r -tuple (b_1, b_2, \dots, b_r) , where $b_i = x \bmod m_i$ ($i = 1, r$), is a bijection between \mathbf{Z}_M and $\mathbf{Z}_{m_1} \times \cdots \times \mathbf{Z}_{m_r}$.

A good theorem deserves a good proof. An outstanding theorem deserves two proofs, at least, one existential, and one constructive. So here are one of each for the Chinese Remainder Theorem.

Proof 1: We must show that each r -tuple (b_1, \dots, b_r) such that $0 \leq b_i < m_i$ ($i = 1, r$) occurs exactly once. There are obviously M such vectors, and so it will be sufficient to show that each of them occurs *at most* once as the image of some x .

In the contrary case we would have x and x' both corresponding to (b_1, b_2, \dots, b_r) , say. But then $x - x' \equiv 0$ modulo each of the m_i . Hence $x - x'$ is divisible by $M = m_1 m_2 \cdots m_r$. But $|x - x'| < M$, hence $x = x'$. ■

Proof 2: Here’s how to compute a number x that satisfies the simultaneous congruences $x \equiv b_i \pmod{m_i}$ ($i = 1, r$). First, by the extended Euclidean algorithm we can quickly find $t_1, \dots, t_r, u_1, \dots, u_r$, such that $t_j(M/m_j) + u_j m_j = 1$ for $j = 1, \dots, r$. Then we claim that the number $x = \sum_j b_j t_j (M/m_j)$ satisfies all of the given congruences. Indeed, for each $k = 1, 2, \dots, r$ we have

$$\begin{aligned} x &= \sum_{j=1}^r b_j t_j (M/m_j) \\ &\equiv b_k t_k (M/m_k) \pmod{m_k} \\ &\equiv b_k \pmod{m_k} \end{aligned}$$

where the first congruence holds because each M/m_j ($j \neq k$) is divisible by m_k , and the second congruence follows since

$$t_k (M/m_k) = 1 - u_k m_k \equiv 1 \pmod{m_k},$$

completing the second proof of the Chinese Remainder Theorem. ■

Now the proof of theorem 4.5.4 follows easily, and is left as an exercise for the reader.

The factorization that is described in detail in theorem 4.5.4 will be written symbolically as

$$\mathbf{Z}_n \cong \bigotimes_{i=1}^r \mathbf{Z}_{p_i^{a_i}}. \quad (4.5.2)$$

The factorization (4.5.2) of the ring \mathbf{Z}_n induces a factorization

$$U_n \cong \bigotimes_{i=1}^r U_{p_i^{a_i}} \quad (4.5.3)$$

of the group of units. Since U_n is a group, (4.5.3) is an isomorphism of the multiplicative structure only. In \mathbf{Z}_{12} , for example, we find

$$U_{12} \cong U_4 U_3$$

where $U_4 = \{1, 3\}$, $U_3 = \{1, 2\}$. So U_{12} can be thought of as the set $\{(1, 1), (1, 2), (3, 1), (3, 2)\}$, together with the componentwise multiplication operation described above.

Exercises for section 4.5

1. Give a complete proof of theorem 4.5.4.
2. Find all primitive roots modulo 18.
3. Find all primitive roots modulo 27.
4. Write out the multiplication table of the group U_{27} .
5. Which elements of \mathbf{Z}_{11} are squares?
6. Which elements of \mathbf{Z}_{13} are squares?
7. Find all $x \in U_{27}$ such that $x^2 = 1$. Find all $x \in U_{15}$ such that $x^2 = 1$.
8. Prove that if there is a primitive root modulo n then the equation $x^2 = 1$ in the group U_n has only the solutions $x = \pm 1$.
9. Find a number x that is congruent to 1, 7 and 11 to the respective moduli 5, 11 and 17. Use the method in the second proof of the remainder theorem 4.5.5.
10. Write out the complete proof of the ‘immediate’ corollary 4.5.3.

4.6 Pseudoprimality tests

In this section we will discuss various tests that might be used for testing the compositeness of integers probabilistically.

By a *pseudoprimality test* we mean a test that is applied to a pair (b, n) of integers, and that has the following characteristics:

- (a) The possible outcomes of the test are ‘ n is composite’ or ‘inconclusive.’
- (b) If the test reports ‘ n is composite’ then n is composite.
- (c) The test runs in a time that is polynomial in $\log n$.

If the test result is ‘inconclusive’ then we say that n is *pseudoprime to the base b* (which means that n is so far acting like a prime number, as far as we can tell).

The outcome of the test of the primality of n depends on the base b that is chosen. In a good pseudoprimality test there will be many bases b that will give the correct answer. More precisely, a good pseudoprimality test will, with high probability (*i.e.*, for a large number of choices of the base b) declare that a composite n is composite. In more detail, we will say that a pseudoprimality test is ‘good’ if there is a fixed positive number t such that every composite integer n is declared to be composite for at least tn choices of the base b , in the interval $1 \leq b \leq n$.

Of course, given an integer n , it is silly to say that ‘there is a high probability that n is prime.’ Either n is prime or it isn’t, and we should not blame our ignorance on n itself. Nonetheless, the abuse of language is sufficiently appealing that we will define the problem away: we will say that a given integer n is *very probably prime* if we have subjected it to a good pseudoprimality test, with a large number of different bases b , and have found that it is pseudoprime to all of those bases.

Here are four examples of pseudoprimality tests, only one of which is ‘good.’

Test 1. Given b, n . Output ‘ n is composite’ if b divides n , else ‘inconclusive.’

This isn’t the good one. If n is composite, the probability that it will be so declared is the probability that we happen to have found a b that divides n , where b is not 1 or n . The probability of this event, if b is chosen uniformly at random from $[1, n]$, is

$$p_1 = (d(n) - 2)/n$$

where $d(n)$ is the number of divisors of n . Certainly p_1 is not bounded from below by a positive constant t , if n is composite.

Test 2. Given b, n . Output ‘ n is composite’ if $\gcd(b, n) \neq 1$, else output ‘inconclusive.’

This one is a little better, but not yet good. If n is composite, the number of bases $b \leq n$ for which Test 2 will produce the result ‘composite’ is $n - \phi(n)$, where ϕ is the Euler totient function, of (4.1.5). This number of useful bases will be large if n has some small prime factors, but in that case it’s easy to find out that n is composite by other methods. If n has only a few large prime factors, say if $n = p^2$, then the proportion of useful bases is very small, and we have the same kind of inefficiency as in Test 1 above.

Now we can state the third pseudoprimality test.

Test 3. Given b, n . (If b and n are not relatively prime or) if $b^{n-1} \not\equiv 1 \pmod{n}$ then output ‘ n is composite,’ else output ‘inconclusive.’

Regrettably, the test is still not ‘good,’ but it’s a lot better than its predecessors. To cite an extreme case of its un-goodness, there exist composite numbers n , called *Carmichael numbers*, with the property that the pair (b, n) produces the output ‘inconclusive’ for every integer b in $[1, n-1]$ that is relatively prime to n . An example of such a number is $n = 1729$, which is composite ($1729 = 7 \cdot 13 \cdot 19$), but for which Test 3 gives the result ‘inconclusive’ on every integer $b < 1729$ that is relatively prime to 1729 (i.e., that is not divisible by 7 or 13 or 19).

Despite such misbehavior, the test usually seems to perform quite well. When $n = 169$ (a difficult integer for tests 1 and 2) it turns out that there are 158 different b ’s in $[1, 168]$ that produce the ‘composite’ outcome from Test 3, namely every such b except for 19, 22, 23, 70, 80, 89, 99, 146, 147, 150, 168.

Finally, we will describe a good pseudoprimality test. The familial resemblance to Test 3 will be apparent.

Test 4. (the strong pseudoprimality test): Given (b, n) . Let $n - 1 = 2^q m$, where m is an odd integer. If either

- (a) $b^m \equiv 1 \pmod{n}$ or
- (b) there is an integer i in $[0, q-1]$ such that

$$b^{m2^i} \equiv -1 \pmod{n}$$

then return ‘inconclusive’ else return ‘ n is composite.’

First we validate the test by proving the

Proposition. If the test returns the message ‘ n is composite,’ then n is composite.

Proof: Suppose not. Then n is an odd prime. We claim that

$$b^{m2^i} \equiv 1 \pmod{n}$$

for all $i = q, q-1, \dots, 0$. If so then the case $i = 0$ will contradict the outcome of the test, and thereby complete the proof. To establish the claim, it is clearly true when $i = q$, by Fermat’s theorem. If true for i , then it is true for $i-1$ also, because

$$\begin{aligned} (b^{m2^{i-1}})^2 &= b^{m2^i} \\ &\equiv 1 \pmod{n} \end{aligned}$$

implies that the quantity being squared is $+1$ or -1 . Since n is an odd prime, by corollary 4.5.3 U_n is cyclic, and so the equation $x^2 = 1$ in U_n has only the solutions $x = \pm 1$. But -1 is ruled out by the outcome of the test, and the proof of the claim is complete. ■

What is the computational complexity of the test? Consider first the computational problem of raising a number to a power. We can calculate, for example, $b^m \pmod{n}$ with $O(\log m)$ integer multiplications, by successive squaring. More precisely, we compute b, b^2, b^4, b^8, \dots by squaring, and reducing modulo n immediately after each squaring operation, rather than waiting until the final exponent is reached. Then we use the binary expansion of the exponent m to tell us which of these powers of b we should multiply together in order to compute b^m . For instance,

$$b^{337} = b^{256} \cdot b^{64} \cdot b^{16} \cdot b.$$

The complete power algorithm is recursive and looks like this:

```

function power(b, m, n);
{returns  $b^m \bmod n$ }
if  $m = 0$ 
  then
    power := 1
  else
     $t := \text{sqr}(\text{power}(b, \lfloor m/2 \rfloor, n));$ 
    if  $m$  is odd then  $t := t \cdot b;$ 
    power :=  $t \bmod n$ 
end.{power}

```

Hence part (a) of the strong pseudoprimality test can be done in $O(\log m) = O(\log n)$ multiplications of integers of at most $O(\log n)$ bits each. Similarly, in part (b) of the test there are $O(\log n)$ possible values of i to check, and for each of them we do a single multiplication of two integers each of which has $O(\log n)$ bits (this argument, of course, applies to Test 3 above also).

The entire test requires, therefore, some low power of $\log n$ bit operations. For instance, if we were to use the most obvious way to multiply two B bit numbers we would do $O(B^2)$ bit operations, and then the above test would take $O((\log n)^3)$ time. This is a polynomial in the number of bits of input.

In the next section we are going to prove that Test 4 is a good pseudoprimality test in that if n is composite then at least half of the integers b , $1 \leq b \leq n-1$ will give the result ‘ n is composite.’

For example, if $n = 169$, then it turns out that for 157 of the possible 168 bases b in $[1, 168]$, Test 4 will reply ‘169 is composite.’ The only bases b that 169 can fool are 19, 22, 23, 70, 80, 89, 99, 146, 147, 150, 168. For this case of $n = 169$ the performances of Test 4 and of Test 3 are identical. However, there are no analogues of the Carmichael numbers for Test 4.

Exercises for section 4.6

1. Given an odd integer n . Let $T(n)$ be the set of all $b \in [1, n]$ such that $\gcd(b, n) = 1$ and $b^{n-1} \equiv 1 \pmod{n}$. Show that $|T(n)|$ divides $\phi(n)$.
2. Let H be a cyclic group of order n . How many elements of each order r are there in H (r divides n)?
3. If $n = p^a$, where p is an odd prime, then the number of $x \in U_n$ such that x has exact order r , is $\phi(r)$, for all divisors r of $\phi(n)$. In particular, the number of primitive roots modulo n is $\phi(\phi(n))$.
4. If $n = p_1^{a_1} \cdots p_m^{a_m}$, and if r divides $\phi(n)$, then the number of $x \in U_n$ such that $x^r \equiv 1 \pmod{n}$ is

$$\prod_{i=1}^m \gcd(\phi(p_i^{a_i}), r).$$

5. In a group G suppose f_m and g_m are, respectively, the number of elements of order m and the number of solutions of the equation $x^m = 1$, for each $m = 1, 2, \dots$. What is the relationship between these two sequences? That is, how would you compute the g 's from the f 's? the f 's from the g 's? If you have never seen a question of this kind, look in any book on the theory of numbers, find ‘Möbius inversion,’ and apply it to this problem.

4.7 Proof of goodness of the strong pseudoprimality test

In this section we will show that if n is composite, then at least half of the integers b in $[1, n-1]$ will yield the result ‘ n is composite’ in the strong pseudoprimality test. The basic idea of the proof is that a subgroup of a group that is not the entire group can consist of at most half of the elements of that group.

Suppose n has the factorization

$$n = p_1^{a_1} \cdots p_s^{a_s}$$

and let $n_i = p_i^{a_i}$ ($i = 1, s$).

Lemma 4.7.1. *The order of each element of U_n is a divisor of $e^* = \text{lcm}\{\phi(n_i); i = 1, s\}$.*

Proof: From the product representation (4.5.3) of U_n we find that an element x of U_n can be regarded as an s -tuple of elements from the cyclic groups U_{n_i} ($i = 1, s$). The order of x is equal to the lcm of the orders of the elements of the s -tuple. But for each $i = 1, \dots, s$ the order of the i^{th} of those elements is a divisor of $\phi(n_i)$, and therefore the order of x divides the lcm shown above. ■

Lemma 4.7.2. *Let $n > 1$ be odd. For each element u of U_n let $C(u) = \{1, u, u^2, \dots, u^{e-1}\}$ denote the cyclic group that u generates. Let B be the set of all elements u of U_n for which $C(u)$ either contains -1 or has odd order (e odd). If B generates the full group U_n then n is a prime power.*

Proof: Let $e^* = 2^t m$, where m is odd and e^* is as shown in lemma 4.7.1. Then there is a j such that $\phi(n_j)$ is divisible by 2^t .

Now if n is a prime power, we are finished. So we can suppose that n is divisible by more than one prime number. Since $\phi(n)$ is an even number for all $n > 2$ (proof?), the number e^* is even. Hence $t > 0$ and we can define a mapping ψ of the group U_n to itself by

$$\psi(x) = x^{2^{t-1}m} \quad (x \in U_n)$$

(note that $\psi(x)$ is its own inverse).

This is in fact a group homomorphism:

$$\forall x, y \in U_n : \psi(xy) = \psi(x)\psi(y).$$

Let B be as in the statement of lemma 4.7.2. For each $x \in B$, $\psi(x)$ is in $C(x)$ and

$$\psi(x)^2 = \psi(x^2) = 1.$$

Since $\psi(x)$ is an element of $C(x)$ whose square is 1, $\psi(x)$ has order 1 or 2. Hence if $\psi(x) \neq 1$, it is of order 2. If the cyclic group $C(x)$ is of odd order then it contains no element of even order. Hence $C(x)$ is of even order and contains -1 . Then it can contain no other element of order 2, so $\psi(x) = -1$ in this case.

Hence for every $x \in B$, $\psi(x) = \pm 1$.

Suppose B generates the full group U_n . Then not only for every $x \in B$ but for every $x \in U_n$ it is true that $\psi(x) = \pm 1$.

Suppose n is not a prime power. Then $s > 1$ in the factorization (4.5.2) of U_n . Consider the element v of U_n which, when written out as an s -tuple according to that factorization, is of the form

$$v = (1, 1, 1, \dots, 1, y, 1, \dots, 1)$$

where the ‘ y ’ is in the j^{th} component, $y \in U_{n_j}$ (recall that j is as described above, in the second sentence of this proof). We can suppose y to be an element of order exactly 2^t in U_{n_j} since U_{n_j} is cyclic.

Consider $\psi(v)$. Clearly $\psi(v)$ is not 1, for otherwise the order of y , namely 2^t , would divide $2^{t-1}m$, which is impossible because m is odd.

Also, $\psi(v)$ is not -1 , because the element -1 of U_n is represented uniquely by the s -tuple all of whose entries are -1 . Thus $\psi(v)$ is neither 1 nor -1 in U_n , which contradicts the italicized assertion above. Hence $s = 1$ and n is a prime power, completing the proof. ■

Now we can prove the main result of Solovay, Strassen and Rabin, which asserts that Test 4 is good.

Theorem 4.7.1. *Let B' be the set of integers $b \bmod n$ such that (b, n) returns ‘inconclusive’ in Test 4.*

- (a) *If B' generates U_n then n is prime.*
- (b) *If n is composite then B' consists of at most half of the integers in $[1, n - 1]$.*

Proof: Suppose $b \in B'$ and let m be the odd part of $n - 1$. Then either $b^m \equiv 1$ or $b^{m2^i} \equiv -1$ for some $i \in [0, q - 1]$. In the former case the cyclic subgroup $C(b)$ has odd order, since m is odd, and in the latter case $C(b)$ contains -1 .

Hence in either case $B' \subseteq B$, where B is the set defined in the statement of lemma 4.7.2 above. If B' generates the full group U_n then B does too, and by lemma 4.7.2, n is a prime power, say $n = p^k$.

Also, in either of the above cases we have $b^{n-1} \equiv 1$, so the same holds for all $b \in B'$, and so for all $x \in U_n$ we have $x^{n-1} \equiv 1$, since B' generates U_n .

Now U_n is cyclic of order

$$\phi(n) = \phi(p^k) = p^{k-1}(p-1).$$

By theorem 4.5.3 there are primitive roots modulo $n = p^k$. Let g be one of these. The order of g is, on the one hand, $p^{k-1}(p-1)$ since the set of all of its powers is identical with U_n , and on the other hand is a divisor of $n-1 = p^k - 1$ since $x^{n-1} \equiv 1$ for all x , and in particular for $x = g$.

Hence $p^{k-1}(p-1)$ (which, if $k > 1$, is a multiple of p) divides $p^k - 1$ (which is one less than a multiple of p), and so $k = 1$, which completes the proof of part (a) of the theorem.

In part (b), n is composite and so B' cannot generate all of U_n , by part (a). Hence B' generates a proper subgroup of U_n , and so can contain at most half as many elements as U_n contains, and the proof is complete. ■

Another application of the same circle of ideas to computer science occurs in the generation of random numbers on a computer. A good way to do this is to choose a primitive root modulo the word size of your computer, and then, each time the user asks for a random number, output the next higher power of the primitive root. The fact that you started with a primitive root insures that the number of 'random numbers' generated before repetition sets in will be as large as possible.

Now we'll summarize the way in which the primality test is used. Suppose there is given a large integer n , and we would like to determine if it is prime.

We would do

```
function testn(n, outcome);
times := 0;
repeat
    choose an integer b uniformly at random in [2, n - 1];
    apply the strong pseudoprimalty test (Test 4) to the
        pair (b, n);
    times := times + 1
until {result is 'n is composite' or times = 100};
if times = 100 then outcome := 'n probably prime'
else outcome := 'n is composite'
end{testn}
```

If the procedure exits with 'n is composite,' then we can be certain that n is not prime. If we want to see the factors of n then it will be necessary to use some factorization algorithm, such as the one described below in section 4.9.

On the other hand, if the procedure halts because it has been through 100 trials without a conclusive result, then the integer n is very probably prime. More precisely, the chance that a composite integer n would have behaved like that is less than 2^{-100} . If we want certainty, however, it will be necessary to apply a test whose outcome will *prove* primality, such as the algorithm of Adleman, Rumely and Pomerance, referred to earlier.

In section 4.9 we will discuss a probabilistic factoring algorithm. Before doing so, in the next section we will present a remarkable application of the complexity of the factoring problem, to cryptography. Such applications remind us that primality and factorization algorithms have important applications beyond pure mathematics, in areas of vital public concern.

Exercises for section 4.7

1. For $n = 9$ and for $n = 15$ find all of the cyclic groups $C(u)$, of lemma 4.7.2, and find the set B .
2. For $n = 9$ and $n = 15$ find the set B' , of theorem 4.7.1.

4.8 Factoring and cryptography

A computationally intractable problem can be used to create secure codes for the transmission of information over public channels of communication. The idea is that those who send the messages to each other will have extra pieces of information that will allow them to solve the intractable problem rapidly, whereas an aspiring eavesdropper would be faced with an exponential amount of computation.

Even if we don't have a provably computationally intractable problem, we can still take a chance that those who might intercept our messages won't know any polynomial-time algorithms if we don't know any. Since there are precious few *provably* hard problems, and hordes of *apparently* hard problems, it is scarcely surprising that a number of sophisticated coding schemes rest on the latter rather than the former. One should remember, though, that an adversary might discover fast algorithms for doing these problems and keep that fact secret while deciphering all of our messages.

A remarkable feature of a family of recently developed coding schemes, called 'Public Key Encryption Systems,' is that the 'key' to the code lies in the public domain, so it can be easily available to sender and receiver (and eavesdropper), and can be readily changed if need be. On the negative side, the most widely used Public Key Systems lean on computational problems that are only *presumed* to be intractable, like factoring large integers, rather than having been *proved* so.

We are going to discuss a Public Key System called the RSA scheme, after its inventors: Rivest, Shamir and Adleman. This particular method depends for its success on the seeming intractability of the problem of finding the factors of large integers. If that problem could be done in polynomial time, then the RSA system could be 'cracked.'

In this system there are three centers of information: the sender of the message, the receiver of the message, and the Public Domain (for instance, the 'Personals' ads of the *New York Times*). Here is how the system works.

(A) Who knows what and when

Here are the items of information that are involved, and who knows each item:

p, q : two large prime numbers, chosen by the receiver, and told to nobody else (not even to the sender!).

n : the product pq is n , and this is placed in the Public Domain.

E : a random integer, placed in the Public Domain by the receiver, who has first made sure that E is relatively prime to $(p-1)(q-1)$ by computing the g.c.d., and choosing a new E at random until the g.c.d. is 1. This is easy for the receiver to do because p and q are known to him, and the g.c.d. calculation is fast.

P : a message that the sender would like to send, thought of as a string of bits whose value, when regarded as a binary number, lies in the range $[0, n-1]$.

In addition to the above, one more item of information is computed by the receiver, and that is the integer D that is the multiplicative inverse mod $(p-1)(q-1)$ of E , i.e.,

$$DE \equiv 1 \pmod{(p-1)(q-1)}.$$

Again, since p and q are known, this is a fast calculation for the receiver, as we shall see.

To summarize,

The receiver knows p, q, D
The sender knows P
Everybody knows n and E

In Fig. 4.8.1 we show the interiors of the heads of the sender and receiver, as well as the contents of the Public Domain.

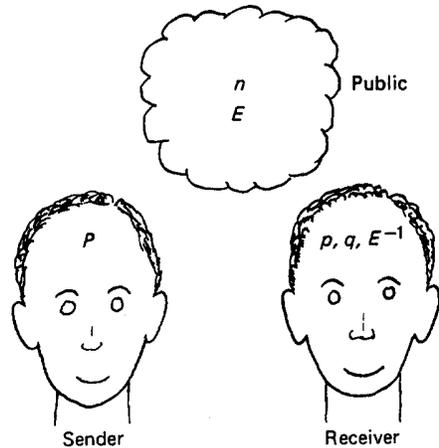


Fig. 4.8.1: Who knows what

(B) How to send a message

The sender takes the message P , looks at the public keys E and n , computes $C \equiv P^E \pmod{n}$, and transmits C over the public airwaves.

Note that the sender has no private codebook or anything secret other than the message itself.

(C) How to decode a message

The receiver receives C , and computes $C^D \pmod{n}$. Observe, however, that $(p-1)(q-1)$ is $\phi(n)$, and so we have

$$\begin{aligned} C^D &\equiv P^{DE} \\ &= P^{(1+t\phi(n))} \quad (t \text{ is some integer}) \\ &\equiv P \pmod{n} \end{aligned}$$

where the last equality is by Fermat's theorem (4.5.1). The receiver has now recovered the original message P .

If the receiver suspects that the code has been broken, *i.e.*, that the adversaries have discovered the primes p and q , then the sender can change them without having to send any secret messages to anyone else. Only the public numbers n and E would change. The sender would not need to be informed of any other changes.

Before proceeding, the reader is urged to construct a little scenario. Make up a short (very short!) message. Choose values for the other parameters that are needed to complete the picture. Send the message as the sender would, and decode it as the receiver would. Then try to intercept the message, as an eavesdropper would, and see what the difficulties are.

(D) How to intercept the message

An eavesdropper who receives the message C would be unable to decode it without (inventing some entirely new decoding scheme or) knowing the inverse D of $E \pmod{(p-1)(q-1)}$. The eavesdropper, however, does not even know the modulus $(p-1)(q-1)$ because p and q are unknown (only the receiver knows them), and knowing the product $pq = n$ alone is insufficient. The eavesdropper is thereby compelled to derive a polynomial-time factoring algorithm for large integers. May success attend those efforts!

The reader might well remark here that the receiver has a substantial computational problem in creating two large primes p and q . To a certain extent this is so, but two factors make the task a good deal easier. First, p and q will need to have only half as many bits as n has, so the job is of smaller size. Second, there

are methods that will produce large prime numbers very rapidly as long as one is not too particular about which primes they are, as long as they are large enough. We will not discuss those methods here.

The elegance of the RSA cryptosystem prompts a few more remarks that are intended to reinforce the distinction between exponential- and polynomial-time complexities.

How hard is it to factor a large integer? At this writing, integers of up to perhaps a couple of hundred digits can be approached with some confidence that factorization will be accomplished within a few hours of the computing time of a very fast machine. If we think in terms of a message that is about the length of one typewritten page, then that message would contain about 8000 bits, equivalent to about 2400 decimal digits. This is in contrast to the largest feasible length that can be handled by contemporary factoring algorithms of about 200 decimal digits. A one-page message is therefore well into the zone of computational intractability.

How hard is it to find the multiplicative inverse, $\text{mod } (p-1)(q-1)$? If p and q are *known* then it's easy to find the inverse, as we saw in corollary 4.3.1. Finding an inverse $\text{mod } n$ is no harder than carrying out the extended Euclidean algorithm, *i.e.*, it's a linear time job.

4.9 Factoring large integers

The problem of finding divisors of large integers is in a much more primitive condition than is primality testing. For example, we don't even know a *probabilistic* algorithm that will return a factor of a large composite integer, with probability $> 1/2$, in polynomial time.

In this section we will discuss a probabilistic factoring algorithm that finds factors in an *average* time that is only moderately exponential, and that's about the state of the art at present.

Let n be an integer whose factorization is desired.

Definition. By a *factor base* B we will mean a set of distinct nonzero integers $\{b_0, b_1, \dots, b_h\}$.

Definition. Let B be a factor base. An integer a will be called a B -number if the integer c that is defined by the conditions

$$\begin{aligned} (a) \quad & c \equiv a^2 \pmod{n} \text{ and} \\ (b) \quad & -n/2 \leq c < n/2 \end{aligned}$$

can be written as a product of factors from the factor base B .

If we let $e(a, i)$ denote the exponent of b_i in that product, then we have

$$a^2 \equiv \prod_{i=0}^h b_i^{e(a,i)} \pmod{n}.$$

Hence, for each B -number we get an $(h+1)$ -vector of exponents $\mathbf{e}(\mathbf{a})$.

Suppose we can find enough B -numbers so that the resulting collection of exponent vectors is a linearly dependent set, *mod* 2. For instance, a set of $h+2$ B -numbers would certainly have that property.

Then we could nontrivially represent the zero vector as a sum of a certain set A of exponent vectors, say

$$\sum_{\mathbf{a} \in A} \mathbf{e}(\mathbf{a}) \equiv (0, 0, \dots, 0) \pmod{2}.$$

Now define the integers

$$\begin{aligned} r_i &= (1/2) \sum_{\mathbf{a} \in A} e(\mathbf{a}, i) \quad (i = 0, 1, \dots, h) \\ u &= \prod_A a \pmod{n} \\ v &= \prod_i b_i^{r_i}. \end{aligned}$$

It then would follow, after an easy calculation, that $u^2 \equiv v^2 \pmod{n}$. Hence either $u - v$ or $u + v$ has a factor in common with n . It may be, of course, that $u \equiv \pm v \pmod{n}$, in which case we would have

learned nothing. However if neither $u \equiv v \pmod{n}$ nor $u \equiv -v \pmod{n}$ is true then we will have found a nontrivial factor of n , namely $\gcd(u - v, n)$ or $\gcd(u + v, n)$.

Example:

Take as a factor base $B = \{-2, 5\}$, and let it be required to find a factor of $n = 1729$. Then we claim that 186 and 267 are B -numbers. To see that 186 is a B -number, note that $186^2 = 20 \cdot 1729 + (-2)^4$, and similarly, since $267^2 = 41 \cdot 1729 + (-2)^4 5^2$, we see that 267 is a B -number, for this factor base B .

The exponent vectors of 186 and 267 are $(4, 0)$ and $(4, 2)$ respectively, and these sum to $(0, 0) \pmod{2}$, hence we find that

$$\begin{aligned} u &= 186 \times 267 \equiv 1250 \pmod{1729} \\ r_1 &= 4; \quad r_2 = 1 \\ v &= (-2)^4(5)^1 = 80 \\ \gcd(u - v, n) &= \gcd(1170, 1729) = 13 \end{aligned}$$

and we have found the factor 13 of 1729. ■

There might have seemed to be some legerdemain involved in plucking the B -numbers 186 and 267 out of the air, in the example above. In fact, as the algorithm has been implemented by its author, J. D. Dixon, one simply chooses integers uniformly at random from $[1, n - 1]$ until enough B -numbers have been found so their exponent vectors are linearly dependent modulo 2. In Dixon's implementation the factor base that is used consists of -1 together with the first h prime numbers.

It can then be proved that if n is not a prime power then with a correct choice of h relative to n , if we repeat the random choices until a factor of n is found, the average running time will be

$$\exp\{(2 + o(1))(\log \log \log n)^{.5}\}.$$

This is not polynomial time, but it is moderately exponential only. Nevertheless, it is close to being about the best that we know how to do on the elusive problem of factoring a large integer.

4.10 Proving primality

In this section we will consider a problem that sounds a lot like primality testing, but is really a little different because the rules of the game are different. Basically the problem is to convince a skeptical audience that a certain integer is prime, requiring them to do only a small amount of computation in order to be so persuaded.

First, though, suppose you were writing a 100-decimal-digit integer n on the blackboard in front of a large audience and you wanted to prove to them that n was *not* a prime.

If you simply wrote down two smaller integers whose product was n , the job would be done. Anyone who wished to be certain could spend a few minutes multiplying the factors together and verifying that their product was indeed n , and all doubts would be dispelled.

Indeed*, a speaker at a mathematical convention in 1903 announced the result that $2^{67} - 1$ is not a prime number, and to be utterly convincing all he had to do was to write

$$2^{67} - 1 = 193707721 \times 761838257287.$$

We note that the speaker probably had to work very hard to *find* those factors, but having found them it became quite easy to convince others of the truth of the claimed result.

A pair of integers r, s for which $r \neq 1$, $s \neq 1$, and $n = rs$ constitute a *certificate* attesting to the compositeness of n . With this certificate $\mathcal{C}(n)$ and an auxiliary checking algorithm, *viz.*

- (1) Verify that $r \neq 1$, and that $s \neq 1$
- (2) Verify that $rs = n$

we can prove, in polynomial time, that n is not a prime number.

* We follow the account given in V. Pratt, Every prime has a succinct certificate, *SIAM J. Computing*, 4 (1975), 214-220.

Now comes the hard part. How might we convince an audience that a certain integer n is a prime number? The rules are that we are allowed to do any immense amount of calculation beforehand, and the results of that calculation can be written on a certificate $\mathcal{C}(n)$ that accompanies the integer n . The audience, however, will need to do only a polynomial amount of further computation in order to convince themselves that n is prime.

We will describe a primality-checking algorithm \mathcal{A} with the following properties:

- (1) Inputs to \mathcal{A} are the integer n and a certain certificate $\mathcal{C}(n)$.
- (2) If n is prime then the action of \mathcal{A} on the inputs $(n, \mathcal{C}(n))$ results in the output ‘ n is prime.’
- (3) If n is not prime then *for every possible certificate* $\mathcal{C}(n)$ the action of \mathcal{A} on the inputs $(n, \mathcal{C}(n))$ results in the output ‘primality of n is not verified.’
- (4) Algorithm \mathcal{A} runs in polynomial time.

Now the question is, does such a procedure exist for primality verification? The answer is affirmative, and we will now describe one. The fact that primality can be quickly verified, if not quickly discovered, is of great importance for the developments of Chapter 5. In the language of section 5.1, what we are about to do is to show that the problem ‘Is n prime?’ belongs to the class NP.

The next lemma is a kind of converse to ‘Fermat’s little theorem’ (theorem 4.5.2).

Lemma 4.10.1. *Let p be a positive integer. Suppose there is an integer x such that $x^{p-1} \equiv 1 \pmod{p}$ and such that for all divisors d of $p-1$, $d < p-1$, we have $x^d \not\equiv 1 \pmod{p}$. Then p is prime.*

Proof: First we claim that $\gcd(x, p) = 1$, for let $g = \gcd(x, p)$. Then $x = gg'$, $p = gg''$. Since $x^{p-1} \equiv 1 \pmod{p}$ we have $x^{p-1} = 1 + tp$ and $x^{p-1} - tp = (gg')^{p-1} - tgg'' = 1$. The left side is a multiple of g . The right side is not, unless $g = 1$.

It follows that $x \in U_p$, the group of units of \mathbf{Z}_p . Thus x is an element of order $p-1$ in a group of order $\phi(p)$. Hence $(p-1) | \phi(p)$. But always $\phi(p) \leq p-1$. Hence $\phi(p) = p-1$ and p is prime. ■

Lemma 4.10.1 is the basis for V. Pratt’s method of constructing certificates of primality. The construction of the certificate is actually *recursive* since step 3⁰ below calls for certificates of smaller primes. We suppose that the certificate of the prime 2 is the trivial case, and that it can be verified at no cost.

Here is a complete list of the information that is on the certificate $\mathcal{C}(p)$ that accompanies an integer p whose primality is to be attested to:

- 1⁰: a list of the primes p_i and the exponents a_i for the canonical factorization $p-1 = \prod_{i=1}^r p_i^{a_i}$
- 2⁰: the certificates $\mathcal{C}(p_i)$ of each of the primes p_1, \dots, p_r
- 3⁰: a positive integer x .

To verify that p is prime we could execute the following algorithm \mathcal{B} :

- (B1) Check that $p-1 = \prod p_i^{a_i}$.
- (B2) Check that each p_i is prime, using the certificates $\mathcal{C}(p_i)$ ($i = 1, r$).
- (B3) For each divisor d of $p-1$, $d < p-1$, check that $x^d \not\equiv 1 \pmod{p}$.
- (B4) Check that $x^{p-1} \equiv 1 \pmod{p}$.

This algorithm \mathcal{B} is correct, but it might not operate in polynomial time. In step B3 we are looking at every divisor of $p-1$, and there may be a lot of them.

Fortunately, it isn’t necessary to check *every* divisor of $p-1$. The reader will have no trouble proving that *there is a divisor d of $p-1$ ($d < p-1$) for which $x^d \equiv 1 \pmod{p}$ if and only if there is such a divisor that has the special form $d = (p-1)/p_i$.*

The primality checking algorithm \mathcal{A} now reads as follows.

- (A1) Check that $p-1 = \prod p_i^{a_i}$.
- (A2) Check that each p_i is prime, using the certificates $\mathcal{C}(p_i)$ ($i = 1, r$).
- (A3) For each $i := 1$ to r , check that

$$x^{(p-1)/p_i} \not\equiv 1 \pmod{p}.$$

(A4) Check that $x^{p-1} \equiv 1 \pmod{p}$.

Now let's look at the complexity of algorithm \mathcal{A} .

We will measure its complexity by the number of times that we have to do a computation of either of the types (a) 'is $m = \prod q_j^{b_j}$?' or (b) 'is $y^s \equiv 1 \pmod{p}$?'

Let $f(p)$ be that number. Then we have (remembering that the algorithm calls itself r times)

$$f(p) = 1 + \sum_{i=2}^r f(p_i) + r + 1 \tag{4.10.1}$$

in which the four terms, as written, correspond to the four steps in the checking algorithm. The sum begins with ' $i = 2$ ' because the prime 2, which is always a divisor of $p - 1$, is 'free.'

Now (4.10.1) can be written as

$$g(p) = \sum_{i=2}^r g(p_i) + 4 \tag{4.10.2}$$

where $g(p) = 1 + f(p)$. We claim that $g(p) \leq 4 \log_2 p$ for all p .

This is surely true if $p = 2$. If true for primes less than p then from (4.10.2),

$$\begin{aligned} g(p) &\leq \sum_{i=2}^r \{4 \log_2 p_i\} + 4 \\ &= 4 \log_2 \left\{ \prod_{i=2}^r p_i \right\} + 4 \\ &\leq 4 \log_2 \{(p-1)/2\} + 4 \\ &= 4 \log_2 (p-1) \\ &\leq 4 \log_2 p. \end{aligned}$$

Hence $f(p) \leq 4 \log_2 p - 1$ for all $p \geq 2$. ■

Since the number of bits in p is $\Theta(\log p)$, the number $f(p)$ is a number of executions of steps that is a polynomial in the length of the input bit string. We leave to the exercises the verification that each of the steps that $f(p)$ counts is also executed in polynomial time, so the entire primality-verification procedure operates in polynomial time. This yields

Theorem 4.10.1. (*V. Pratt, 1975*) *There exist a checking algorithm and a certificate such that primality can be verified in polynomial time.*

Exercises for section 4.10

1. Show that two positive integers of b bits each can be multiplied with at most $O(b^2)$ bit operations (multiplications and carries).
2. Prove that step A1 of algorithm \mathcal{A} can be executed in polynomial time, where time is now measured by the number of bit operations that are implied by the integer multiplications.
3. Same as exercise 2 above, for steps A3 and A4.
4. Write out the complete certificate that attests to the primality of 19.
5. Find an upper bound for the total number of bits that are in the certificate of the integer p .
6. Carry out the complete checking algorithm on the certificate that you prepared in exercise 4 above.
7. Let $p = 15$. Show that there is no integer x as described in the hypotheses of lemma 4.10.1.
8. Let $p = 17$. Find all integers x that satisfy the hypotheses of lemma 4.10.1.

Bibliography

The material in this chapter has made extensive use of the excellent review article

John D. Dixon, Factorization and primality tests, *The American Mathematical Monthly*, **91** (1984), 333-352.

A basic reference for number theory, Fermat's theorem, etc. is

G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, Oxford University Press, Oxford, 1954

Another is

W. J. LeVeque, *Fundamentals of Number Theory*, Addison-Wesley, Reading, MA, 1977

The probabilistic algorithm for compositeness testing was found by

M. O. Rabin, Probabilistic algorithms, in *Algorithms and Complexity, New Directions and Recent Results*, J. Traub ed., Academic Press, New York, 1976

and at about the same time by

R. Solovay and V. Strassen, A fast Monte Carlo test for primality, *SIAM Journal of Computing*, **6** (1977), pp. 84-85; *erratum ibid.*, **7** (1978), 118.

Some empirical properties of that algorithm are in

C. Pomerance, J. L. Selfridge and S. Wagstaff Jr., The pseudoprimes to $25 \cdot 10^9$, *Mathematics of Computation*, **35** (1980), 1003-1026.

The fastest nonprobabilistic primality test appeared first in

L. M. Adleman, On distinguishing prime numbers from composite numbers, *IEEE Abstracts*, May 1980, 387-406.

A more complete account, together with the complexity analysis, is in

L. M. Adleman, C. Pomerance and R. S. Rumely, On distinguishing prime numbers from composite numbers, *Annals of Mathematics* **117** (1983), 173-206.

A streamlined version of the above algorithm was given by

H. Cohen and H. W. Lenstra Jr., Primality testing and Jacobi sums, Report 82-18, Math. Inst. U. of Amsterdam, Amsterdam, 1982.

The idea of public key data encryption is due to

W. Diffie and M. E. Hellman, New directions in cryptography, *IEEE Transactions on Information Theory*, **IT-22**, 6 (1976), 644-654.

An account of the subject is contained in

M. E. Hellman, The mathematics of public key cryptography, *Scientific American*, **241**, 2 (August 1979), 146-157.

The use of factoring as the key to the code is due to

R. L. Rivest, A. Shamir and L. M. Adleman, A method for obtaining digital signatures and public key cryptosystems, *Communications of the A.C.M.*, **21**, 2 (February 1978), 120-126

The probabilistic factoring algorithm in the text is that of

John D. Dixon, Asymptotically fast factorization of integers, *Mathematics of Computation*, **36** (1981), 255-260.