Number-theoretic algorithms[1]

1 Factoring versus primality testing

One of the most fundamental dichotomies in the study of algorithms is between problems which can be solved efficiently and those which cannot. In the present chapter, we illustrate this divide with two number-theoretic tasks: factoring and deciding whether a number is prime. Despite the close connection between these two problems, they differ vastly in computational complexity. For factoring no polynomial-time algorithm is known or believed to exist. On the other hand, we will derive a fast and elegant procedure for testing primality, by using some classical results of number theory. We will then see how the juxtaposition of these two intimately-related problems, one intractable and the other efficiently solvable, makes possible some of the most beautiful and important cryptosystems.

The path to these results is paved with many glorious landmarks in the development of numeric algorithms. The first and most basic of them has to do with the representation of numbers.

2 Basic arithmetic

We are so used to writing numbers in decimal, or binary, or other bases $b \geq 2$, that it seems strange that these representations have not always been around, and that in fact they took great pains to discover. Through the ages, people have had to contend with many less convenient alternatives. In *unary*, for example, the number n is denoted by n ones, and therefore occupies space proportional to n. This is exponentially more wasteful than base b, in which the same number takes up only $\lceil \log_b n \rceil = O(\log n)$ space. The Roman number system, which still survives to a small extent, is just a minor improvement over unary, incorporating a few extra symbols which reduce number sizes only by a constant factor.

In base b, the number 374 means $3 \times b^2 + 7 \times b + 4$. This representation, in which different positions denote different powers of the base, is extremely convenient for arithmetic algorithms. For instance, it makes it possible to multiply a number by b simply by left-shifting it (giving 3740 above). More generally, for all the basic arithmetic tasks there are simple procedures that can be carried out by hand, and which are familiar to us from childhood. Unfortunately we learn these long before we can appreciate why they work, so it worth pinpointing the principles behind them.

2.1 Addition

The standard method for adding numbers relies on one observation – in any base $b \geq 2$, if you add up three single-digit numbers, then you get back at most a two-digit number. This is because the maximum possible sum is $3(b-1)$, which is strictly less than b^2 (why?), the smallest three-digit number.

This primitive fact makes it possible to add two numbers x, y by aligning their right-hand ends, and then performing a single right-to-left pass in which the sum is computed digit by digit, maintaining the overflow in a carry digit. The invariant is that the overflow is a single digit, so that at any given step, three single-digit numbers are added.

Figure 2.1 Multiplication.

```
function multiply(x, y)
Input:  Two n-bit integers x[1...n] and y[1...n]
Output:  Their product z[1...2n]

z = 0
for i = 1 to n:
   if y[i] = 1:
      z = 2z + x
   else:
      z = 2z
return z
```

If x, y are n bits long, their sum is $n + 1$ bits, and each individual bit of this sum gets computed in a fixed amount of time. The total running time for the addition algorithm is therefore $O(n)$, *linear* in the size of the inputs.

Subtraction can be done in much the same way, and is also linear.

2.2 Multiplication and division

The grade school algorithm for multiplying two numbers x, y is to create an array of intermediate sums, each representing the product of x by a single digit of y. These values are appropriately left-shifted and then added up. To see the rationale behind this procedure, suppose for instance that $x = 10011$ and $y = 101$, in binary notation.

$$10011 \times 101 \quad = \quad (10011 \times 1) \times 4 + (10011 \times 0) \times 2 + (10011 \times 1) \times 1$$

Each term in parentheses is the product of x with a single bit of y. The corresponding power of two specifies how much that term needs to be left-shifted.

		1	0	0	1	1	$= 10011 \times 1$
	0	0	0	0	0		$= 10011 \times 0$
1	0	0	1	1			$= 10011 \times 1$
1	0	1	1	1	1	1	(answer)

In binary this is particularly easy since each intermediate row is either zero or x. The pseudocode of Figure 2.1 mimics this procedure, but rather than explicitly constructing the entire array of partial sums, it starts with the last of the sums (the one at the bottom of the array) and works its way up, keeping track of the total so far. For each row of the array it does $O(n)$ work; the total time taken is thus $O(n^2)$, *quadratic* in the size of the inputs. This is certainly polynomial but is much slower than addition.

The division of x by y results in a quotient q and a remainder r, where $x = yq + r$ and $r < y$. The "long division" procedure for finding q and r can once again be streamlined considerably when the numbers are in binary (Figure 2.2). Like multiplication, it takes quadratic time.

Figure 2.2 Division.

```
function divide(x, y)
Input:  Two n-bit integers x[1...n] and y[1...n]
Output:  Integers q[1...n], r[1...n] such that x = yq + r, r < y

q = r = 0
for i = 1 to n:
    r = 2r + x[i]
    if r ≥ y:
        r = r − y
        q[i] = 1
return (q, r)
```

3 Modular arithmetic

With repeated addition or multiplication, numbers can get cumbersomely large. This is why we reset the hour to zero whenever it reaches twenty-four, and the month to January after every stretch of twelve months. Similarly, for the built-in arithmetic operations of computer processors, numbers are restricted to some size – 32 bits, say – which is considered generous enough for most purposes.

Modular arithmetic is a system for dealing with restricted ranges of integers. It is based upon an enhanced notion of equivalence between numbers: x and y are *congruent modulo* m if they differ by a multiple of m, or in symbols,

$$x \equiv y \pmod{m} \quad \Leftrightarrow \quad m \,|\, (x - y)$$

("|" means "divides"). For instance, $253 \equiv 13 \bmod 60$ because $253 - 13$ is a multiple of sixty: 253 minutes is 4 hours and 13 minutes.

One way to think of modulo arithmetic is that it limits numbers to a predefined range $\{0, 1, \ldots, m-1\}$, and wraps around whenever you try to leave this range, like the hands of a clock. In this system, the usual associative, commutative, and distributive properties of addition and multiplication continue to apply. For instance,

(Associative)	$x + (y + z) \equiv (x + y) + z \pmod{m}$.
(Commutative)	$xy \equiv yx \pmod{m}$.
(Distributive)	$x(y + z) \equiv xy + yz \pmod{m}$.

Another interpretation is that modular arithmetic allows all integers but divides them into m *equivalence classes*, each of the form $\{i + km : k \in \mathbf{Z}\}$ for some i between 0 and $m - 1$. Any member of an equivalence class is substitutable for any other (see exercise); for example, $xy \bmod m$ is the same as $(x + k_1 m)(y + k_2 m) \bmod m$. In any sequence of arithmetic operations, therefore, it is legal to reduce intermediate results modulo m at any stage. This can tremendously simplify big calculations. Witness:

$$2^{140} + 258 \cdot 34 \equiv (2^5)^{28} + 10 \cdot 3 \equiv 32^{28} - 9 \equiv 1^{28} + 30 \equiv 1 - 1 \equiv 0 \pmod{31}.$$

Some of the features of modular arithmetic are nicely illustrated in *two's complement*, the most common format for storing signed integers. It uses n bits to represent numbers in the range $[-2^{n-1}, 2^{n-1}-1]$, and is usually described as follows:

- Positive integers, in the range 0 to $2^{n-1}-1$, are stored in regular binary, and have a leading bit of zero.
- Negative integers $-x$, with $1 \leq x \leq 2^{n-1}$, are stored by first constructing x in binary, then flipping all the bits, and finally adding 1. The leading bit in this case is one.

Here's a much simpler description: any number in the range $-2^{n-1} \ldots 2^{n-1}-1$ is stored modulo 2^n. Negative numbers $-x$ therefore end up as $2^n - x$. Arithmetic operations like addition and subtraction can be performed directly in this format, ignoring any overflow bits that arise.

3.1 Modular addition and multiplication

To add two numbers x, y modulo m, we start with regular addition. The result is between 0 and $2(m-1)$; if it exceeds $m-1$, we need to subtract off m. The overall computation therefore consists of at most one addition and one subtraction, of numbers which never exceed $O(m)$. Its running time is linear in the sizes of these numbers, $O(\log m)$. Modular subtraction is much the same, using the relation $-y \equiv m - y$ to convert it into an addition problem.

The product of two mod-m numbers x, y can be as large as $(m-1)^2$, but this is still just $O(\log m)$ bits long. To reduce it modulo m, all we have to do is compute the remainder upon dividing it by m. Therefore multiplication remains a quadratic operation.

Division is not quite so easy. Normally there is just one tricky case – division by zero. It turns out that in modular arithmetic there are potentially other such cases as well, which we will characterize towards the end of this section. Whenever division is legal, however, it can be managed in cubic time, $O(\log^3 m)$.

For our next two tasks – modular exponentiation and greatest common divisor – the most obvious procedures take exponentially long, but with some ingenuity polynomial-time solutions can be found. A careful choice of algorithm makes all the difference.

3.2 Modular exponentiation

Modular exponentiation consists of computing $a^b \bmod m$. One way to do this is to repeatedly multiply by a modulo m, generating the sequence of intermediate products $a^i \bmod m$, $i = 1, \ldots, b$. They each take $O(\log^2 m)$ time to compute, and so the overall running time to compute the $b-1$ products is $O(b \log^2 m)$. This might not look too bad, but notice that it is exponential in the size of b. If we were not working in modulo arithmetic, we couldn't hope to do much better than this, because then the answer a^b would itself be so large that it would take time proportional to b just to write down. However, in our case, the final answer is just $O(\log m)$ bits long, so a better running time is at least plausible.

The key to an efficient algorithm is make use of an additional operation which rapidly increases the exponent of a number a^i:

> To double the exponent (or equivalently, to left-shift its binary representation), *square* the number.

We still need the operation we were using earlier, which we now rephrase thus:

Figure 3.1 Modular exponentiation.

```
function ModExp(a, b, m)
Input:  Integers 0 < a < m and a positive exponent b = b[1...n]
Output:  a^b mod m

r = 1
for i = 1 to n:
  r = r^2 mod m
  if b[i] = 1 then r = ra mod m
return r
```

To increment the exponent by one, multiply the number by a.

Starting with an exponent of zero, we iteratively perform increment and left-shift operations until the exponent is b. If $b = 1011_2$ (the subscript means binary), this looks like

$$a^0 \xrightarrow{\text{incr}} a^1 \xrightarrow{\text{shift}} a^{10} \xrightarrow{\text{shift}} a^{100} \xrightarrow{\text{incr}} a^{101} \xrightarrow{\text{shift}} a^{1010} \xrightarrow{\text{incr}} a^{1011}.$$

The algorithm is shown in Figure 3.1, and has a running time of $O(\log b \log^2 m)$. Typically $b < m$, so this is *cubic* in the size of the input.

3.3 Euclid's algorithm for greatest common divisor

Our next algorithm was discovered well over two thousand years ago by the mathematician Euclid, in ancient Greece. It computes the *greatest common divisor* of two integers a and b: the largest integer which divides both of them. The most obvious approach is to first factor a and b, and then multiply together their common factors. For instance, $1035 = 3^2 \cdot 5 \cdot 23$ and $759 = 3 \cdot 11 \cdot 23$, so their gcd is $3 \cdot 23 = 69$. However, no polynomial-time procedure is known for factoring, so we must somehow sidestep this necessity.

Euclid's algorithm uses the following simple rule.

Property. If $a > b$ then $\gcd(a, b) = \gcd(a \bmod b, b)$.

In terms of proof, it is enough to show the slightly simpler rule $\gcd(a, b) = \gcd(a - b, b)$ from which the one above can be derived by repeatedly subtracting b from a.

Any integer which divides both a and b must also divide both $a - b$ and b, so $\gcd(a, b) \leq \gcd(a - b, b)$. Likewise, any integer which divides both $a - b$ and b must also divide both a and b, so $\gcd(a, b) \geq \gcd(a - b, b)$.

Euclid's rule allows us to write down an elegant recursive algorithm (Figure 3.2). In order to figure out its running time, we need to understand how quickly the arguments (a, b) decrease with each successive recursive call. In a single round, they become $(b, a \bmod b)$: their order is swapped, and the larger of them, a, gets reduced to $a \bmod b$. This is a substantial reduction:

Claim. If $a \geq b$ then $a \bmod b \leq a/2$.

Figure 3.2 Euclid's algorithm for finding the greatest common divisor of two numbers.

```
function Euclid(a, b)
Input:  Two positive integers a, b with a ≥ b
Output:  gcd(a, b)

if b = 0 then return a
return Euclid(b, a mod b)
```

To see this, consider two possible ranges for the value of b. Either (i) $b \leq a/2$, in which case $a \bmod b < b \leq a/2$, or (ii) $b > a/2$, in which case $a \bmod b = a - b \leq a/2$.

This means that after any two consecutive rounds, both arguments are at the very least halved in value. Within $2\lceil \log_2 b \rceil$ recursive calls, therefore, the second argument will get down to zero. In terms of the input size $n = \lceil \log_2 a \rceil + \lceil \log_2 b \rceil$, there are $O(n)$ rounds and each involves a quadratic-time division, for a total time of $O(n^3)$.

3.4 An extension of Euclid's algorithm

Suppose someone claims that d is the greatest common divisor of a and b: how can we check this? It is not enough to verify that d divides both a and b, because this only shows d to be a common factor, not necessarily the largest one. Here's a test that can be used if d is of a particular form.

> **Claim.** If $d \mid a$ and $d \mid b$ and $d = ax + by$ for some integers x, y, then d must be $\gcd(a, b)$.

Certainly d is a common divisor of a and b. Let d' be any other common divisor; since d' divides both a and b, it must also divide $d = ax + by$, so $d' \leq d$. Therefore d is the largest common divisor.

But when can $\gcd(a, b)$ be expressed in this checkable form? It turns out that it *always* can. For instance, the greatest common divisor of 5 and 7 is $1 = 5 \times 3 - 7 \times 2$. The coefficients x, y can moreover be found by a small extension to Euclid's algorithm; see Figure 3.3.

> **Claim.** For any inputs a, b, the Extended Euclid algorithm returns integers x, y, d such that $\gcd(a, b) = d = ax + by$.

The first thing to confirm is that if you ignore the x's and y's, the extended algorithm is exactly the same as the original. Therefore at least $d = \gcd(a, b)$ is correctly computed.

For the rest, the recursive nature of the algorithm suggests a proof by induction. The recursion ends when $b = 0$, so it is convenient to do induction on the value of b.

The base case $b = 0$ is easy enough to check directly. Now pick any larger value of b. The algorithm finds $\gcd(a, b)$ by calling $\gcd(b, a \bmod b)$. Since $a \bmod b < b$, we can apply the inductive hypothesis to this recursive call and conclude that the x', y' it returns are correct:

$$\gcd(b, a \bmod b) = bx' + (a \bmod b)y'.$$

Writing $(a \bmod b)$ as $(a - \lfloor a/b \rfloor b)$, we find

$$\gcd(a, b) \;=\; \gcd(b, a \bmod b) \;=\; bx' + (a - \lfloor a/b \rfloor b)y' \;=\; ay' + b(x' - \lfloor a/b \rfloor y').$$

This validates the algorithm's behavior on input (a, b).

Figure 3.3 A simple extension of Euclid's algorithm.

```
function Extended-Euclid(a, b)
Input:  Two positive integers a, b with a ≥ b
Output:  Integers x, y, d such that d = gcd(a, b) and ax + by = d

if b = 0 then return (1, 0, a)
(x', y', d) = Extended-Euclid(b, a mod b)
return (y', x' − ⌊a/b⌋y', d)
```

3.5 Modular division

In regular arithmetic, every number $a \neq 0$ has an inverse $1/a$, and dividing by a is the same as multiplying by this inverse. In modulo arithmetic, we can similarly define the multiplicative inverse of a to be a number x such that $ax \equiv 1 \pmod{m}$. There can be at most one such x modulo m (why?) and we shall denote it by a^{-1}. However, this inverse does not always exist. Notice that $ax \bmod m$ is of the form $ax + km$ for some integer k, and is therefore divisible by $\gcd(a, m)$. If this gcd is more than 1, then $ax \not\equiv 1$, no matter what x might be.

In fact, this is the only circumstance in which a is not invertible. When $\gcd(a, m) = 1$ (we say a and m are *relatively prime*), the Extended Euclid algorithm gives us integers x, y such that $ax + my = 1$. Reducing modulo m, we find x is the sought inverse.

> **Property.** For any $a < m$, a has a multiplicative inverse modulo m if and only if it is relatively prime to m. When this inverse exists, it can be found in time $O(\log^3 m)$.

When working modulo m, we can thus always divide by numbers relatively prime to m.

4 Primality testing

In the previous section, our first try at a gcd algorithm ran into intractability problems because of its dependence upon factoring. We are now in for an even closer brush with this particular barrier, as our next task is to determine whether a number is prime. Is there some litmus test which will answer this without actually trying to factor the number?

We place our hopes in a classic theorem from the year 1640.

> **Fermat's Little Theorem.** If m is prime then for every $1 \leq a < m$,
>
> $$a^{m-1} \equiv 1 \pmod{m}.$$

First of all, since a is relatively prime to m, we are allowed to divide by it.

Let S be the set of numbers $\{1, 2, \ldots, m-1\}$. Multiply each of these by a modulo m; we will show that the resulting numbers are all distinct and non-zero. Since they lie in the range $[1, m-1]$, they must again constitute the set S, that is, $S = \{a \cdot i \bmod m : 1 \leq i < m\}$.

The numbers $a \cdot i \bmod m$ are distinct because if $a \cdot i \equiv a \cdot j \pmod{m}$ then dividing both sides by a gives $i \equiv j \pmod{m}$. They are non-zero because $a \cdot i \equiv 0$ similarly implies $i \equiv 0$.

We now have two ways to write set S. Let's multiply together its elements in each of these representations. We get

$$(m-1)! \equiv a^{m-1} \cdot (m-1)! \pmod{m}.$$

Figure 4.1 An algorithm for testing primality.

```
Primality1(s)
Input:  A positive integer m
Output:  yes/no

Pick an integer a < m at random
If a^{m-1} ≡ 1 (mod m)
  then return yes
  else return no
```

Dividing by $(m-1)!$, which is relatively prime to m, gives the theorem.

This is extremely promising, but it is not an "if and only if" condition; it doesn't say what happens if m is *not* prime, so let's to try to understand that case. For a composite m, it is certainly possible that $a^{m-1} \equiv 1 \bmod m$ for certain choices of a. For instance, $341 = 11 \cdot 31$ is not prime, and yet $2^{340} \equiv 1 \bmod 341$. Nevertheless, if a is picked *randomly*, it is unlikely that $a^{m-1} \equiv 1 \bmod m$. This motivates the algorithm of Figure 4.1.

Let's make this more precise. First of all, there is a bit of bad news in that certain composite values of m, called *Carmichael numbers*, satisfy $a^{m-1} \equiv 1 \bmod m$ for *all* a. But these numbers are very rare, and we will later fix our algorithm to handle them, so let's ignore them for the time being. For the remaining values of m, there is at least one a for which $a^{m-1} \not\equiv 1 \bmod m$. It turns out that if there is some such a, there must be lots of them.

> **Claim.** If $a^{m-1} \not\equiv 1 \bmod m$ for some a relatively prime to m, then it must hold for at least half the choices of $a < m$.

Let $a_0 < m$ be relatively prime to m and satisfy $a_0^{m-1} \not\equiv 1 \bmod m$. The key is to notice that for every element $a < m$ which passes Fermat's test with respect to m (that is, $a^{m-1} \equiv 1 \bmod m$) there is a corresponding element $a_0 \cdot a \bmod m$ which fails the test, $(a_0 \cdot a)^{m-1} \not\equiv 1 \bmod m$. Therefore at most half the possible values of a can pass the test.

More formally, let $A = \{a < s : a^{m-1} \equiv 1 \bmod m\}$ be the set of values a which pass Fermat's test. Then, as in the previous theorem, the set $\{a_0 \cdot a : a \in A\}$ consists of $|A|$ distinct values, since a_0 is relatively prime to m and thus invertible modulo m. Moreover these values all fail Fermat's test with respect to m, since for any $a \in A$,

$$(a_0 \cdot a)^{m-1} \equiv a_0^{m-1} \cdot a^{m-1} \equiv a_0^{m-1} \cdot 1 \not\equiv 1 \pmod m.$$

This proves the claim: at least as many values of a fail the test as do succeed.

We are ignoring Carmichael numbers, so we can now assert

> If m is prime, then $a^{m-1} \equiv 1 \bmod m$ for all $a < m$.
> If m is not prime, then $a^{m-1} \equiv 1 \bmod m$ for at most half the values of $a < m$.

The algorithm of Figure 4.1 therefore has the following guarantee:

$$\begin{aligned} \Pr(\text{Algorithm 4.1 returns } \texttt{yes} \text{ when } m \text{ is prime}) &= 1 \\ \Pr(\text{Algorithm 4.1 returns } \texttt{yes} \text{ when } m \text{ is not prime}) &\leq \tfrac{1}{2} \end{aligned}$$

Figure 4.2 An algorithm for testing primality, with lower error probability.

```
Primality2(m)
Input:  A positive integer m
Output:  yes/no

Pick integers a_1, a_2, ..., a_k < m at random
If a_i^{m-1} ≡ 1 (mod m) for all i = 1, 2, ..., k
  then return yes
  else return no
```

We can reduce this *one-sided error* by repeating the procedure many times, by randomly picking several values of a and testing them all (Figure 4.2).

$$\Pr(\text{Algorithm 4.2 returns } \texttt{yes} \text{ when } m \text{ is not prime}) \quad \leq \quad \tfrac{1}{2^k}$$

This probability of error drops exponentially fast, and can be driven arbitrarily low by choosing k large enough. Testing $k = 100$ different values of a makes the probability of failure at most 2^{-100}, which is miniscule: far less, for instance, than the probability that a random cosmic ray will sabotage the computer during the computation!

Advanced Notes

Theorem 4 can also be explained using group theory. The set $A = \{a : a^{m-1} = 1 \bmod m\}$ is a *subgroup* of the multiplicative group G of numbers modulo m which are relatively prime to m. The size of a subgroup must divide the size of the group. So if A doesn't contain all of G, the next largest size it can have is $|G|/2$.

What about the problem of Carmichael numbers? There is a way around them, using a slightly more refined primality test due to Rabin and Miller. Write $m-1$ in the form $2^t u$. As before we'll choose a random base a and check the value of $a^{m-1} \bmod m$. Perform this computation by first determining $a^u \bmod m$ and then repeatedly squaring. If we find that $a^{m-1} \not\equiv 1 \bmod m$, then m is composite by Fermat's Little Theorem, and we're done. Suppose instead that $a^{m-1} \equiv 1 \bmod m$. Somewhere along the way, while computing $a^u \bmod m, a^{2u} \bmod m, \ldots, a^{2^t u} = a^{m-1} \equiv 1 \bmod m$, we ran into a 1 for the first time. If $a^u \bmod m \neq 1$, look at the value in the list before that first 1. If that value is not $-1 \bmod m$, then we have found a *non-trivial square root* of 1 modulo m: a number which is not $\pm 1 \bmod m$, but which when squared is equal to $1 \bmod m$. Such a number can only exist if m is composite (you should try proving this). It turns out that if we combine this square-root check with our earlier Fermat test, then at least $3/4$ of the possible values of a between 1 and $m-1$ will reveal a composite m, even if it is a Carmichael number.

4.1 Generating random primes

For some cryptographic applications we will need to choose primes randomly. Our algorithm for doing so relies on the fact that primes are actually quite common.

Lagrange's Prime Number Theorem. Let $\pi(x)$ be the number of primes $< x$. Then $\pi(x) \approx \frac{x}{\ln x}$, or more precisely,

$$\lim_{x \to \infty} \frac{\pi(x)}{\frac{x}{\ln x}} = 1.$$

According to this theorem, a random n-bit number has about a one-in-n chance of being prime (actually about $2^n/(\ln 2^n) \approx 1.44/n$). The approximation gets better for larger n.

This suggests a simple way to generate a random n-bit prime:

- Pick a random n-bit number m.
- Run a primality test on m (Algorithm 4.2).
- If it passes the test, output m; else repeat the process.

If the randomly chosen m is truly prime, which happens with probability at least $1/n$, then it will certainly pass the test. So on each iteration, this procedure has at least a $1/n$ chance of halting. Therefore on average it will halt after $O(n)$ rounds.

When the procedure stops, what is the chance that it outputs a number which really is prime? To put it differently, what is the probability that a randomly chosen n-bit number m is prime, given that it passes the primality test?

$$\begin{aligned} \mathbf{P}(m \text{ is prime} \mid m \text{ passes test}) &= \frac{\mathbf{P}(m \text{ is prime and passes test})}{\mathbf{P}(m \text{ passes test})} \\ &= \frac{\mathbf{P}(m \text{ is prime})}{\mathbf{P}(m \text{ passes test})} \end{aligned}$$

Using Lagrange's bound, this works out to approximately $1/(1+n/2^k)$ (you should check this), where $1/2^k$ is the one-sided error of the primality tester. Therefore k should be $\Omega(\log n)$.

5 Cryptography

We end this chapter by describing the Rivest-Shamir-Adelman (RSA) cryptosystem, which derives very strong guarantees of security by ingeniously exploiting the wide gulf between the polynomial-time computability of certain number-theoretic tasks (modular exponentiation, greatest common divisor, primality testing) and the intractability of others (factoring).

The typical setting for cryptography can be described via a cast of three characters: Alice and Bob, who wish to communicate in private, and Eve, an eavesdropper who will go to great lengths to find out what they are saying. For concreteness, let's say Alice wants to send a specific message x, written in binary (why not), to her friend Bob. She encodes it as $e(x)$, sends it over, and then Bob applies his decryption function $d(\cdot)$ to decode it: $d(e(x)) = x$ (Figure 5.1). Eve is able to intercept $e(x)$: for instance, she might be a sniffer on the network. Ideally the encryption function $e(\cdot)$ is so chosen that without knowing $d(\cdot)$, Eve cannot do anything with the information she has picked up. In other words, knowing $e(x)$ tells her little or nothing about what x might be.

Cryptographic schemes can be broadly classified as *private-key* or *public-key*. In the former, Alice and Bob meet beforehand and together choose a secret codebook, with which they

Figure 5.1 Alice wants to send Bob a message.

encrypt all future correspondence between them. Eve's only hope, then, is to collect some encoded messages and use them to at least partially figure out the codebook. *Public-key* schemes such as RSA are significantly more subtle and tricky: they allow Alice to send Bob a message without ever having seen him before. This almost sounds implausible, because it seems like Bob is no better off than Eve in terms of being able to understand Alice's message. The way around this problem is for Bob to publish some public information about himself, which Alice can use while sending her message. For instance, Bob could make available a public locker which, once shut, can only be opened by his key. RSA has revolutionized cryptography by digitally implementing such a system, with a compelling computational guarantee of security. In this protocol, Alice and Bob only need to perform the simplest of calculations, like multiplication, which any pocket computing device could handle. On the hand, in order for Eve to be successful, she needs to perform operations like factoring large numbers, which requires more computational power than would be afforded by the world's most powerful computers combined.

It is ironic that the thoroughly practical field of cryptography should rest so heavily upon number theory, which has long been regarded as one of the purest areas of mathematics, untarnished by material consequence. The renowned number theorist G. H. Hardy once declared of his work: "I have never done anything useful". Yet theorems of the kind he is alluding to are now crucial to the operation of web browsers and cell phones, and to the security of financial transactions worldwide.

We start with a private-key scheme.

5.1 The one-time pad

In this setting, Alice and Bob meet beforehand and pick a random binary string r, called a *one-time pad*, of the same length as the message x which will later be sent. Alice's encryption function is a bitwise exclusive-or, $e(x) = x \oplus r$, and Bob's decryption function is the same thing, $d(y) = e(y)$, so $d(e(x)) = x \oplus r \oplus r = x$.

Eve gains nothing by intercepting $e(x)$, because it is a completely random string:

> **Property.** Fix any n-bit string x, and let r be a random n-bit string. Then, over the possible choices of r, $x \oplus r$ is uniformly distributed over $\{0,1\}^n$; that is, for any $z \in \{0,1\}^n$, $\mathbf{P}(x \oplus r = z) = 1/2^n$.

(Can you prove this?) Therefore, instead of looking at $e(x)$, Eve could do just as well by generating a random n-bit string herself!

A major snag is that the one-time pad can only be used to send one message (hence the

Figure 5.2 RSA.

Bob chooses his public and secret keys.

- He starts by picking two large (n-bit) random primes p and q.
- His public key is (N, e) where $N = pq$ and e is a $2n$-bit number relatively prime to $(p-1)(q-1)$. A common choice is $e = 3$ because it permits fast encoding.
- His secret key is d, the inverse of e modulo $(p-1)(q-1)$, computed using the `Extended-Euclid` algorithm.

Alice wishes to send message x to Bob.

- She looks up his public key (N, e) and sends him $y = (x^e \bmod N)$, computed using an efficient modular exponentiation algorithm.
- He decodes the message by computing $y^d \bmod N$.

name). The second message would not be secure, because if Eve knew $x \oplus r$ and $y \oplus r$ for two messages x and y, then she could take the exclusive-or to get $x \oplus y$, which might be important information. Therefore the random string which Alice and Bob share has to be the combined length of all the messages they will send.

5.2 DES

The one-time pad is a toy cryptographic scheme whose behavior and theoretical properties are completely clear. At the other end of the spectrum lies the *data encryption standard* (DES). This cryptographic protocol, established in 1976, is once again private-key: Alice and Bob have to agree on a shared random string. But this time the string is of a small fixed size, fixty-six to be precise, and can be used repeatedly. The scheme is very widely used but is quite complicated and so a lot of things about it remain unknown. For instance, there are suspicions that it was specifically engineered to be transparent to the US government. It does, however, have some guarantees of security, and certainly the general public does not know how to break the code (that is, how to get x from $e(x)$) except using techniques which are not very much more efficient than the brute-force approach of trying all possibilities for the shared string.

5.3 RSA

Unlike the previous two protocols, the RSA scheme is an example of *public-key cryptography*: anybody can send a message to anybody else using publicly available information, rather like addresses or phone numbers. Each person has a public key known to the whole world, and a secret key known only to himself. When Alice wants to send message x to Bob, she encodes it using his public key. He decrypts it using his secret key, to retrieve x. Eve is welcome to see as many encrypted messages for Bob as she likes, but she will not be able to decode them, under certain simple assumptions.

To understand the RSA protocol, consider any two primes p, q and let $N = pq$. We will shortly establish the following.

> **Property.** For any e relatively prime to $(p-1)(q-1)$, the mapping $x \mapsto x^e \bmod N$ is a bijection on $\{0, 1, \ldots, N-1\}$.

Therefore this mapping is a reasonable way to encode messages x. If Bob publishes (N, e) as his *public key*, then everyone else can use it to send him encrypted messages.

How can Bob decrypt these messages? Another related property serves us here.

> **Property.** For any e relatively prime to $(p-1)(q-1)$, there is a corresponding d such that for all $x \in \{0, \ldots, N-1\}$,
>
> $$(x^e)^d \equiv x \bmod N.$$

(Notice that this implies the previous property.) If Bob has this mystery value d in his possession, as his *secret key*, he can readily decrypt all the messages that come to him. In fact d is easy to describe: it is the multiplicative inverse of e mod $(p-1)(q-1)$. We will prove this a bit later, but for the time being notice that using p and q, it is easy for Bob to figure out d. However, the rest of the world knows only (N, e) (not p, q), and it is (believed) intractable to determine d from these; hence this key truly remains secret. The resulting protocol is shown in Figure 5.2.

To show the correctness of RSA, we need to verify the decrypting properties of d that we claimed above. Since de is of the form $k(p-1)(q-1)+1$ for some integer k, it follows that

$$(x^e)^d \equiv x^{k(p-1)(q-1)+1} \equiv x \pmod{N},$$

where the last equivalence is a generalization of Fermat's theorem to which we now turn.

> **Lemma.** If N is the product of two distinct primes p and q then for any z and k, $z^{k(p-1)(q-1)+1} \equiv z \bmod N$.

Proof. We first check that this holds modulo p. If $p \mid z$ we are immediately done; otherwise, we appeal to Fermat's little theorem and get

$$z^{k(p-1)(q-1)+1} \;\equiv\; (z^{p-1})^{k(q-1)} \cdot z \;\equiv\; 1^{k(q-1)} \cdot z \;\equiv\; z \pmod{p}.$$

The same holds for q, and since $z^{k(p-1)(q-1)+1} - z$ is divisible by both p and q, it must also be divisible by $N = pq$. ∎

The security of RSA hinges upon a simple assumption:

> Given $N, e,$ and $y = x^e \bmod N$, it is computationally intractable to determine x.

How might Eve try to guess x? She could experiment with all possible values of x, each time checking whether $x^e \equiv y \bmod N$, but this would take exponential time. Or she could try to factor N to retrieve p and q, and then figure out d by inverting e modulo $(p-1)(q-1)$, but we believe factoring to be hard. This intractability is normally a source of dismay; the insight of RSA lies in using it to advantage.