Algorithms with Numbers[1]

1 Algorithms

On Books and Algorithms

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

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

The decimal representation was developed in India around 500 AD, but it was first studied seriously by a man who lived in Baghdad in the 800s. Al Khwarizmi developed methods to add, multiply, divide numbers — even extract square roots and calculate the digits of π. The important advantage of these methods was that they did not need expensive equipment like an abacus, or sophisticated knowledge and insight, or even special intelligence or skill. They were precise, unambiguous, mechanical, efficient, correct. They were *algorithms* — a term coined to honor the wise man after his ideas reached Europe, eight centuries later.

1.1 Addition and Multiplication

Today, computers everywhere perform sophisticated tasks by running all kinds of algorithms — we'll see many of them in this book. But the original use of the term "algorithms" applied only to the methods we now learn at school for adding, subtracting, multiplying, etc. decimal numbers.

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 simple fact makes it possible to add two numbers 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 carry is a single digit, so that at any given step, three single-digit numbers are added.

If we are given two n-bit numbers x and y, how long does it take to add them together by the standard algorithm? This is the kind of question we shall be asking throughout this book. We want the answer expressed as a function of the size of the input, the number of bits of x and y. Since we are assuming that x and y are n bits long, their sum is $n + 1$ bits, and each

[1]Copyright ©2004 S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani.

Bases and Logs

Naturally, there is nothing special about the number ten — we just happen to have ten fingers, and so ten was an obvious place to pause and take counting to the next level. The Mayans had developed a similar positional system based on the number twenty (no shoes, see?). And of course today numbers are represented in computers in binary. To represent the number $N \geq 0$ in base b we need about $\log_b N$ digits. To be precise, $\lceil \log_b (N+1) \rceil$.
Recall that $\log_b N = \log N / \log b$: to change the base of a logarithm you divide by the logarithm of the new base. So, representing an integer N in any base we need $O(\log N)$ digits, and only the constant differs from one base to the other (see the box about the $O(\cdot)$ notation). Incidentally, this function, $\log N$, is very important for our subject. When we do not specify a base (and we almost always will not), we mean $\log_2 N$. The reason $\log N$ is important is because it has many interpretations:

1. $\log N$ is, of course, the power to which you need to raise 2 in order to obtain N.
2. Going backwards, it can also be seen as the number of times you must halve N to get down to one. This is useful when a number is halved at each iteration of an algorithm (as in the multiplication algorithm below).
3. It is the number of bits in the binary representation of N.
4. It is also the depth of the full binary tree with N leaves.
5. It is even the sum $1 + \frac{1}{2} + + \frac{1}{3} \cdots + + \frac{1}{N}$ (give or take a constant, see Problem ???).

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. Notice that, by using the $O(\cdot)$ notation we suppress the details of the analysis and focus on the big picture.

Can we do better? (Another often-asked question.) For addition, the answer is trivial: Since to add two n-bit numbers we must at least look at the input and write down the answer, n operations are definitely needed. So, the addition algorithm is optimal!

Some readers may be confused at this point: Why $O(n)$ operations? Isn't binary addition something that computers today perform by just one instruction? Two answers: When we want to understand algorithms, it makes sense to study even the basic algorithms that are now encoded in hardware. In doing so, we shall focus on the *bit complexity* of the algorithm, the number of elementary operations on individual bits —because this accounting reflects the amount of hardware, transistors and wires, necessary for implementing the algorithm. Second, it is certainly true that in one instruction we can add integers with a number of bits equal to the word length of today's computers — 32 perhaps. But later in this chapter we'll see that it is often useful and necessary to handle numbers that are much larger than this, several thousand bits long. Adding and multiplying such numbers on real computers is very much like performing these operations bit by bit.

Onwards to multiplication! 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. Because one of the great advantages of the positional system is that to multiply by the base is to left-shift. (And as we shall see, to divide by two, rounding down if needed, is a right shift.) Suppose for instance that we want to multiply 13×11, or, in binary notation, $x = 1101$ and $y = 1011$. The multiplication would proceed thus:

$$\begin{array}{ccccccccl}
 & & & & 1 & 1 & 0 & 1 & \\
 & & & \times & 1 & 0 & 1 & 1 & \\
\hline
 & & & & 1 & 1 & 0 & 1 & \text{(1101 times 1)} \\
 & & & 1 & 1 & 0 & 1 & & \text{(1101 times 1, shifted once)} \\
 & & 0 & 0 & 0 & 0 & & & \text{(1101 times 0, shifted twice)} \\
+ & 1 & 1 & 0 & 1 & & & & \text{(1101 times 1, shifted thrice)} \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{(answer, 143 in binary)}
\end{array}$$

In binary this is particularly easy since each intermediate row is either zero or x itself, left-shifted an appropriate amount of times. If x and y were both n bits long, we now have to add up to n numbers, each with up to $2n$ bits. 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 (as we had all suspected at elementary school).

But Al Khwarizmi knew also another way to multiply decimal numbers, and this method is used today in some European countries. To multiply two decimal numbers x and y, write them next to each other (see the example below). Then repeat the following: Divide the first number by two (rounding down the result, that is, dropping the ".5" if the number was odd), and double the second number. Keep going till the first number is 1, then add the numbers in the second row that correspond to *odd* numbers in the first row (in our example below, only 52 is omitted).

$$\begin{array}{ccl}
11 & 13 & \\
5 & 26 & \\
2 & 52 & \text{(omit)} \\
1 & 104 & \\
\hline
 & 143 & \text{(answer)}
\end{array}$$

By comparing the two algorithms, binary multiplication and multiplication by repeated halvings of the multiplier, we notice that they are doing the same thing! The three numbers added in the second algorithm are precisely the multiples of 13 by powers of two that were added in the binary method. Only this time 11 was not given to us explicitly in binary, and so we had to extract its binary representation by looking at the parity of the numbers obtained from it by successive divisions by two. Al Khwarizmi's second algorithm is an fascinating mixture of decimal and binary!

The algorithm (Figure 1.1) will terminate after n recursive calls, because at each recursive call y is halved — i.e., its number of bits is decreased by one. And each recursive call requires these operations: A test for odd/even (looking up the last bit); a division by two (a right shift); a multiplication by 2 (a left shift); and at most one addition, a total of $O(n)$ bit operations. The total time taken is thus $O(n^2)$, just as before.

Can we do better? Intuitively, it seems that multiplication cannot avoid adding about n multiples of the multiplicant, and we know that each addition is linear, so it would appear that n^2 bit operations are necessary. Astonishingly, in the next chapter we'll see that this is not so.

Figure 1.1 Multiplication à la Français.

```
function multiply($x, y$)
Input:  Two $n$-bit integers $x, y$
Output:  Their product

if $y = 0$:  return $0$
$z = $ multiply($x, \lceil y/2 \rceil$)
if $y$ is even:
   return $z + z$
else:
   return $x + z + z$
```

Figure 1.2 Division.

```
function divide($x, y$)
Input:  Two $n$-bit integers $x, y$, where $y \geq 1$
Output:  The quotient and remainder of $x$ by $y$

if $x = 0$:  return $(q, r) = (0, 0)$
$(q, r) = $ divide($\lfloor x/2 \rfloor, y$)
$q = 2 \cdot q$,  $r = 2 \cdot r$
if $x$ is odd:  $r = r + 1$
if $r \geq y$:  $r = r - y$,  $q = q + 1$
return $(q, r)$
```

Division is next. To divide x by $y \neq 0$ means to find a quotient q and a remainder r, where $x = yq + r$ and $r < y$. We show the recursive version of division in Figure 1.2. Like multiplication, it takes quadratic time.

2 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 N* if they differ by a multiple of N, or in symbols,

$$x \equiv y \pmod{N} \quad \Leftrightarrow \quad m \mid (x - y)$$

("|" 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 modular arithmetic is that it limits numbers to a predefined range $\{0, 1, \ldots, N-1\}$, and wraps around whenever you try to leave this range — like the hand of a clock. In this system, the usual associative, commutative, and distributive properties of addition and multiplication continue to apply. For instance,

$$\begin{array}{cl} x + (y+z) \equiv (x+y) + z \pmod N & \text{Associativity} \\ xy \equiv yx \pmod N & \text{Commutativity} \\ x(y+z) \equiv xy + yz \pmod N & \text{Distributivity} \end{array}$$

Another interpretation is that modular arithmetic deals with all of the integers, but divides them into N *equivalence classes*, each of the form $\{i + km : k \in \mathbf{Z}\}$ for some i between 0 and $N-1$. Any member of an equivalence class is substitutable for any other (see Exercise???). In any sequence of arithmetic operations, therefore, it is legal to reduce intermediate results modulo N at any stage. This can tremendously simplify big calculations. Witness:

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

Two's complement

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.

2.1 Modular addition and multiplication

To add two numbers x, y modulo N, we start with regular addition. The result is between 0 and $2(N-1)$; if it exceeds $N-1$, we just subtract off N. The overall computation therefore consists of at most one addition and one subtraction, of numbers which never exceed $O(N)$. Its running time is linear in the sizes of these numbers: $O(n)$, where by n we denote as usual $\log N$, the number of bits of N.

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

Division is not quite so easy. In ordinary arithmetic there is just one tricky case – division by zero. It turns out that in modular arithmetic there are potentially other such cases as

Figure 2.1 Modular exponentiation.

```
function mod_exp($x, y, N$)
Input:  Two $n$-bit integers $x, N$, integer exponent y
Output:  $x^y \pmod N$

if $y = 0$ return 1
$z =$ mod_exp($x, \lfloor y/2 \rfloor, N$)
if $y$ is even:
   return $z \cdot z \pmod N$
else:
   return $x \cdot z \cdot z \pmod N$
```

well, which we will characterize towards the end of this section. Whenever division is legal, however, it can be managed in cubic time, $O(n^3)$.

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.

2.2 Modular exponentiation

Suppose we need to compute $x^y \bmod N$. One way to do this is to repeatedly multiply by x modulo N, generating the sequence of intermediate products $x^i \bmod N$, $i = 2, 3 \ldots, y$. Each multiplication takes $O(n^2)$ bit operations to compute, and so the overall running time to compute the $y - 1$ products is $O(yn^2)$. This might not look too bad at first –until we notice that *it is exponential in the size (the number of bits) of y.*[2]

And we *can* do better. To see how, notice the following analogy: We do not compute $x \cdot y$ by adding x to itself y times. Then why do we do something as inefficient when computing x^y? This suggests that $x^y \pmod N$ can be computed by a recursive algorithm very similar to multiplication:

In order to calculate $13^{11} \pmod{31}$, we just multiply together (modulo 31, of course), the powers 13^{2^i} for those powers of two that correspond to ones in the binary representation of 11:

$$13^{11} = 13^8 \cdot 13^2 \cdot 13^1 \pmod{31}.$$

2.3 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 x and y: 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, factoring large integers is a notoriously difficult problem, so we must somehow sidestep this necessity.

[2]If we were not working in modular arithmetic, we couldn't hope to do much better than this, because then the answer x^y would itself be so large that it would take time proportional to y just to write down. However, in our case, the final answer is just $\log N$ bits long, so a better running time is at least plausible.

Figure 2.2 Euclid's algorithm for finding the greatest common divisor of two numbers.

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

if $b = 0$ then return $a$
return Euclid($b, a \bmod b$)
```

Euclid's algorithm uses the following simple rule.

Property. If $x > y$ then $\gcd(x, y) = \gcd(x \bmod y, y)$.

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

Here we go: Any integer which divides both x and y must also divide $x - y$, so $\gcd(x, y) \leq \gcd(x - y, y)$. Likewise, any integer which divides both $x - y$ and y must also divide both x and y, so $\gcd(x, y) \geq \gcd(x - y, y)$.

Euclid's rule allows us to write down an elegant recursive algorithm (Figure 2.2). 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 $a \bmod b \leq a/2$.

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 $O(n^3)$.

2.4 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 $d \,|\, a$ and $d \,|\, b$ and $d = ax + by$ for some integers x, y, then necessarily $d = \gcd(a, b)$.

The reason we are sure d is the greatest common divisor of a and b is simple: Certainly it *is* a common divisor of a and b, since it was assumed to divide both. And any common divisor of a and b must also divide $ax + by$, which is d. Therefore d is the largest common divisor!

So, if we can supply two numbers x and y such that $d = ax + by$, then we can be sure that $d = \gcd(a, b)$. For instance, we know that $gcd(13, 4) = 1$ because $13 \cdot 1 + 4 \cdot (-3) = 1$. But

Figure 2.3 A simple extension of Euclid's algorithm.

```
function Extended-Euclid($a,b$)
Input:  Two positive integers $a,b$ with $a \geq b$
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($b, a \bmod b$)
return $(y', x' - \lfloor a/b \rfloor y', d)$
```

when can we find such numbers? Under what circumstances can $\gcd(a,b)$ be expressed in this checkable form? It turns out that it *always* can. What is even better, the coefficients x, y can be found by a small extension to Euclid's algorithm; see Figure 2.3.

> **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).

2.5 Modular division

In real arithmetic, every number $a \neq 0$ has an inverse $1/a$, and dividing by a is the same as multiplying by this inverse. In modular arithmetic, we can similarly define the multiplicative inverse of a to be a number x such that $ax \equiv 1 \pmod{N}$. There can be at most one such x modulo N (why?) and we shall denote it by a^{-1}. However, this inverse does not always exist. Notice that $ax \bmod N$ is of the form $ax + kN$ for some integer k, and is therefore divisible by $\gcd(a,N)$. 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 $\gcd(a,N) = 1$ (that is, when a and N are *relatively prime*), the Extended Euclid algorithm gives us integers x, y such that $ax + Ny = 1$, which means that $ax = 1 \pmod{N}$. Thus x is a's sought inverse!

> **Property.** For any $a < N$, a has a multiplicative inverse modulo N if and only if it is relatively prime to N. When this inverse exists, it can be found in time $O(n^3)$ (where by n we denote, as usual, the number of bits of N).

In short: When working modulo N, we can divide by numbers relatively prime to N — and only by these.

Is Your Social Security Number a Prime?

2, 7, and 19 are primes, but how about 719? Telling whether a reasonably large number is a prime can be tedious: there are too many candidate factors to try. You can speed up the process by clever tricks: For example, you may omit even candidates after you have tried 2. You can actually omit all candidates except those that are themselves primes.
In fact, a little further thought will convince you that you can proclaim N a prime as soon as you have rejected all candidates *up to* $\sqrt{N}$. (Reason: If N can indeed be factored as $N = K \cdot L$, then it is impossible that both factors exceed $\sqrt{N} \ldots$)
We seem to be making progress! Perhaps by omitting more and more candidate factors, a truly fast primality test can be discovered.
Unfortunately, there is no fast primality test down this road. The reason is that we have been trying to tell if a number is a prime *by factoring it.* And factoring is a hard problem!
Modern cryptography (as well as the balance of this chapter) is about the following important idea: ***Factoring is hard and primality is easy.*** We cannot factor large numbers, but we can easily test huge numbers for primality! (Presumably, if a number is composite, such a test will detect this is the case *without finding a factor.*)

3 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 Test. If N is prime then for every $1 \leq a < N$,

$$a^{N-1} \equiv 1 \pmod{N}.$$

We'll now prove this important property, which is also known as "Fermat's Little Theorem."

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

The numbers $a \cdot i \bmod N$ are distinct because if $a \cdot i \equiv a \cdot j \pmod{N}$ then dividing both sides by a gives $i \equiv j \pmod{N}$. They are non-zero because $a \cdot i \equiv 0$ similarly implies $i \equiv 0$. (And we *can* divide by a, because that number was assumed nonzero, and therefore relatively prime to N.)

We now have two ways to write set S. It is the set $\{1, 2, \ldots, N-1\}$; and it is the set $\{a \cdot 1 \bmod N, a \cdot 2 \bmod N, \ldots, a \cdot (N-1) \bmod N\}$. Let's multiply together its elements in each of these representations. We get

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

Dividing by $(N-1)!$ (which we can do because this number is relatively prime to N, since N is assumed prime), gives the theorem.

This property seems promising viz. our project of devising a "factorless" primality test. But it is not an "if and only if" condition; it doesn't say what happens if N is *not* prime, so let's to try to understand that case.

For a composite N, it is certainly possible that $a^{N-1} \equiv 1 \bmod N$ 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 $a^{N-1} \equiv 1 \bmod N$. This motivates the algorithm of Figure 3.1.

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

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

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

More formally, let $A = \{a < N : a^{N-1} \equiv 1 \bmod N\}$ 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 N and thus invertible modulo N. Moreover these values all fail Fermat's test with respect to N, 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 N is prime, then $a^{N-1} \equiv 1 \bmod N$ for all $a < N$.
If N is not prime, then $a^{N-1} \equiv 1 \bmod N$ for at most half the values of $a < N$.

The algorithm of Figure 3.1 therefore has the following property, stated in the language of probabilities:

$$\begin{aligned} \Pr(\text{Algorithm 3.1 returns } \texttt{yes} \text{ when } N \text{ is prime}) &= 1 \\ \Pr(\text{Algorithm 3.1 returns } \texttt{yes} \text{ when } N \text{ is not prime}) &\le \tfrac{1}{2} \end{aligned}$$

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 3.2).

$$\Pr(\text{Algorithm 3.2 returns } \texttt{yes} \text{ when } N \text{ is not prime}) \le \frac{1}{2^k}$$

Hey, that was Group Theory!

The previous claim can also be explained using group theory. The set $A = \{a : a^{N-1} = 1 \bmod m\}$ is a *subgroup* of the multiplicative group G of numbers modulo N which are relatively prime to N. 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 $|G|/2$.

Figure 3.1 An algorithm for testing primality.

```
Primality($N$)
Input:  A positive integer $N$
Output:  yes/no

Pick an integer $a < N$ at random
If $a^{N-1} \equiv 1 \pmod{N}$
  then return yes
  else return no
```

Figure 3.2 An algorithm for testing primality, with low error probability.

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

Pick integers $a_1, a_2, \ldots, a_k < N$ at random
If $a_i^{N-1} \equiv 1 \pmod{N}$ for all $i = 1, 2, \ldots, k$
  then return yes
  else return no
```

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!

3.1 Generating random primes

In modern cryptography, we routinely set up a cryptosystem by equipping it with large prime numbers. This would be impractical if primes were rare; fortunately quite the reverse is true.

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,

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

This theorem says that a random n-bit number has about a one-in-n chance of being prime (actually about $2^n/(\ln 2^n) \approx 1.44/n$). *About one in twenty social security numbers are primes!*

Such abundance makes it simple to generate a random n-bit prime:

- Pick a random n-bit number N.
- Run a primality test on N (Algorithm 3.2).

Carmichael Numbers

(To add: Examples, Ramanujan, definition and proof that they are precisely the exceptions) There *is* a way around Carmichael numbers, using a slightly more refined primality test due to Rabin and Miller. Write $N-1$ in the form $2^t u$. As before we'll choose a random base a and check the value of $a^{N-1} \bmod N$. Perform this computation by first determining $a^u \bmod N$ and then repeatedly squaring. If we find that $a^{N-1} \not\equiv 1 \bmod N$, then N is composite by Fermat's Little Theorem, and we're done. Suppose instead that $a^{N-1} \equiv 1 \bmod N$. Somewhere along the way, while computing $a^u \bmod m, a^{2u} \bmod N, \ldots, a^{2^t u} = a^{N-1} \equiv 1 \bmod N$, we ran into a 1 for the first time. If $a^u \bmod N \neq 1$, look at the value in the list before that first 1. If that value is not $-1 \bmod N$, then we have found a *non-trivial square root* of 1 modulo N: a number which is not $\pm 1 \bmod N$, but which when squared is equal to $1 \bmod N$. Such a number can only exist if N 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 $N-1$ will reveal a composite m, even if it is a Carmichael number.

- If it passes the test, output N; else repeat the process.

If the randomly chosen N 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 N is prime, given that it passes the primality test?

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

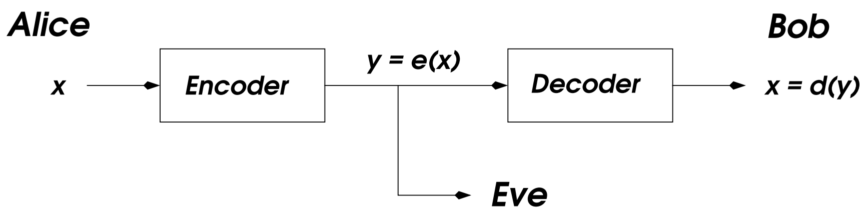
Using Lagrange's bound, this works out to approximately $1/(1+n/2^k)$ (you should check this), where $1/2^k$ is the one-sided error of the primality tester. Therefore k should be $\Omega(\log n)$.

4 Cryptography

Our closing topic, the Rivest-Shamir-Adelman (RSA) cryptosystem, uses all the ideas we have introduced in this chapter! It 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 4.1). 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

Figure 4.1 Alice wants to send Bob a message.



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 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.

4.1 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 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.

4.2 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.

4.3 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

Figure 4.2 RSA.

$\boxed{\text{Bob chooses his public and secret keys.}}$

- 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.

$\boxed{\text{Alice wishes to send message } x \text{ to Bob.}}$

- 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 $y^d \bmod N$.

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 4.2.

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.