# 13 The RSA Function

RSA was among the first public-key cryptography developed. It was first described in 1978, and is named after its creators, Ron Rivest, Adi Shamir, and Len Adleman.[1] Although "textbook" RSA by itself is not a secure encryption scheme, it is a fundamental ingredient for public-key cryptography.

## 13.1 Modular Arithmetic & Number Theory

In general, public-key cryptography relies on computational problems from abstract algebra. Of the techniques currently known for public-key crypto, RSA uses some of the simplest mathematical ideas, so it's an ideal place to start.

We will be working with modular arithmetic, so please review the section on modular arithmetic from the first lecture! We need to understand the behavior of the four basic arithmetic operations in the set $\mathbb{Z}_n = \{0, \ldots, n-1\}$.

Every element $x \in \mathbb{Z}_n$ has an inverse with respect to addition mod $n$: namely $-x \,\%\, n$. For example, the additive inverse of 11 mod 14 is $-11 \equiv_{14} 3$. However, multiplicative inverses are not so straight-forward.

**Greatest common divisors.** If $d \mid x$ and $d \mid y$, then $d$ is a **common divisor** of $x$ and $y$. The largest possible such $d$ is called the **greatest common divisor (GCD),** denoted $\gcd(x,y)$. If $\gcd(x,y) = 1$, then we say that $x$ and $y$ are **relatively prime**. The oldest "algorithm" ever documented is the one Euclid described for computing GCDs (ca. 300 BCE):

```
GCD(x,y): // Euclid's algorithm
  if y = 0 then return x
  else return GCD(y, x % y)
```

**Inverses under multiplication.** We let $\mathbb{Z}_n^*$ denote the set $\{x \in \mathbb{Z}_n \mid \gcd(x,n) = 1\}$, the **multiplicative group modulo** $n$. This group is *closed under multiplication mod n*, which just means that if $x, y \in \mathbb{Z}_n^*$ then $xy \in \mathbb{Z}_n^*$, where $xy$ denotes multiplication mod $n$. Indeed, if $\gcd(x,n) = \gcd(y,n) = 1$, then $\gcd(xy,n) = 1$ and thus $\gcd(xy \,\%\, n, n) = 1$ by Euclid's algorithm.

In abstract algebra, a *group* is a set that is closed under its operation (in this case multiplication mod $n$), and is also closed under inverses. So if $\mathbb{Z}_n^*$ is really a group under multiplication mod $n$, then for every $x \in \mathbb{Z}_n^*$ there must be a $y \in \mathbb{Z}_n^*$ so that $xy \equiv_n 1$. In other words, $y$ is the **multiplicative inverse** of $x$ (and we would give it the name $x^{-1}$.

[1] Clifford Cocks developed an equivalent scheme in 1973, but it was classified since he was working for British intelligence.

The fact that we can always find a multiplicative inverse for elements of $\mathbb{Z}_n^*$ is due to the following theorem:

**Theorem 13.1 (Bezout's Theorem)** *For all integers $x$ and $y$, there exist integers $a$ and $b$ such that $ax + by = \gcd(x,y)$. In fact, $\gcd(x,y)$ is the smallest positive integer that can be written as an integral linear combination of $x$ and $y$.*

What does this have to do with multiplicative inverses? Take any $x \in \mathbb{Z}_n^*$; we will show how to find its multiplicative inverse. Since $x \in \mathbb{Z}_n^*$, we have $\gcd(x,n) = 1$. From Bezout's theorem, there exist integers $a, b$ satisfying $ax + bn = 1$. By reducing both sides of this equation modulo $n$, we have

$$1 \equiv_n ax + bn \equiv_n ax + 0$$

(since $bn \equiv_n 0$). Thus the integer $a$ guaranteed by Bezout's theorem is the multiplicative inverse of $x$ modulo $n$.

We have shown that every $x \in \mathbb{Z}_n^*$ has a multiplicative inverse mod $n$. That is, if $\gcd(x,n) = 1$, then $x$ has a multiplicative inverse. But might it be possible for $x$ to have a multiplicative inverse mod $n$ even if $\gcd(x,n) \neq 1$?

Suppose that we have an element $x$ with a multiplicative inverse; that is, $xx^{-1} \equiv_n 1$. Then $n$ divides $xx^{-1} - 1$, so we can write $xx^{-1} - 1 = kn$ (as an expression over the integers) for some integer $k$. Rearranging, we have that $xx^{-1} - kn = 1$. That is to say, we have a way to write 1 as an integral linear combination of $x$ and $n$. From Bezout's theorem, this must mean that $\gcd(x,n) = 1$. Hence, $x \in \mathbb{Z}_n^*$. We conclude that:

$$\mathbb{Z}_n^* \stackrel{\text{def}}{=} \{x \in \mathbb{Z}_n \mid \gcd(x,n) = 1\} = \{x \in \mathbb{Z}_n \mid \exists y \in \mathbb{Z}_n : xy \equiv_n 1\}.$$

The elements of $\mathbb{Z}_n^*$ are *exactly* those elements with a multiplicative inverse mod $n$.

Furthermore, multiplicative inverses can be computed efficiently using an extended version of Euclid's GCD algorithm. While we are computing the GCD, we can also keep track of integers $a$ and $b$ from Bezout's theorem at every step of the recursion; see below:

```
EXTGCD(x, y):
  // returns (d, a, b) such that gcd(x, y) = d = ax + by
  if y = 0:
    return (x, 1, 0)
  else:
    (d, a, b) := EXTGCD(y, x % y)
    return (d, b, a − b⌊x/y⌋)
```

**Example** *Below is a table showing the computation of EXTGCD$(35, 144)$. Note that the columns $x$, $y$ are computed from the top down (as recursive calls to EXTCD are made), while the columns $d$, $a$, and $b$ are computed from bottom up (as recursive calls return). Also note that in each row, we indeed have $d = ax + by$.*

| $x$ | $y$ | $\lfloor \frac{x}{y} \rfloor$ | $x \,\%\, y$ | $d$ | $a$ | $b$ |
|---|---|---|---|---|---|---|
| *35* | *144* | *0* | *35* | *1* | *-37* | *9* |
| *144* | *35* | *4* | *4* | *1* | *9* | *-37* |
| *35* | *4* | *8* | *3* | *1* | *-1* | *9* |
| *4* | *3* | *1* | *1* | *1* | *1* | *-1* |
| *3* | *1* | *3* | *0* | *1* | *0* | *1* |
| *1* | *0* | *-* | *-* | *1* | *1* | *0* |

*The final result demonstrates that* $35^{-1} \equiv_{144} -37 \equiv_{144} 107$. ◆

**The totient function.** Euler's **totient** function is defined as $\phi(n) = |\mathbb{Z}_n^*|$, in other words, the number of elements of $Z_n$ which are relatively prime to $n$.

As an example, if $p$ is a prime, then $\mathbb{Z}_n^* = \mathbb{Z}_n \setminus \{0\}$ because every integer in $\mathbb{Z}_n$ apart from zero is relatively prime to $p$. Therefore, $\phi(p) = p - 1$.

We will frequently work modulo $n$ where $n$ is the product of two distinct primes $n = pq$. In that case, $\phi(n) = (p-1)(q-1)$. To see why, let's count how many elements in $\mathbb{Z}_{pq}$ share a common divisor with $pq$ (i.e., are *not* in $\mathbb{Z}_{pq}^*$).

- The multiples of $p$ share a common divisor with $pq$. These include $0, p, 2p, 3p, \ldots, (q-1)p$. There are $q$ elements in this list.
- The multiples of $q$ share a common divisor with $pq$. These include $0, q, 2q, 3q, \ldots, (p-1)q$. There are $p$ elements in this list.

We have clearly double-counted element 0 in these lists. But no other element is double counted. Any item that occurs in both lists would be a common multiple of both $p$ and $q$, but the least common multiple of $p$ and $q$ is $pq$ since $p$ and $q$ are relatively prime. But $pq$ is larger than any item in these lists.

We count $p+q-1$ elements in $\mathbb{Z}_{pq}$ which share a common divisor with $pq$. That leaves the rest to reside in $\mathbb{Z}_{pq}^*$, and there are $pq - (p+q-1) = (p-1)(q-1)$ of them. Hence $\phi(pq) = (p-1)(q-1)$.

General formulas for $\phi(n)$ exist, but they typically rely on knowing the prime factorization of $n$. We will see more connections between the difficulty of computing $\phi(n)$ and the difficulty of factoring $n$ later in this part of the course.

Here's an important theorem from abstract algebra:

Theorem 13.2 (LaGrange) *If $G$ is a multiplicative group with $k$ elements, then $x^k = 1$ for all $x \in G$.*

As a corollary, we have that:

$$\text{If } \gcd(x, n) = 1 \text{ then } x^{\phi(n)} \equiv_n 1$$

And as a final corollary, we can deduce Fermat's "little theorem," that $x^p \equiv_p x$ for all $x$, when $p$ is prime.[2]

[2] You have to handle the case of $x \equiv_p 0$ separately, since zero is not in the multiplicative group.

## 13.2 The RSA Function

The RSA function is defined as follows:

- Let $p$ and $q$ be distinct primes (later we will say more about how they are chosen), and let $N = pq$. $N$ is called the **RSA modulus**.
- Let $e$ and $d$ be integers such that $ed \equiv_{\phi(N)} 1$. That is, $e$ and $d$ are multiplicative inverses mod $\phi(N)$ — not mod $N$! $e$ is called the **encryption exponent**, and $d$ is called the **decryption exponent**. These names are historical, but not entirely precise since RSA by itself does not achieve CPA security.
- The RSA function is: $m \mapsto m^e \% N$, where $m \in \mathbb{Z}_N$.
- The inverse RSA function is: $c \mapsto c^d \% N$, where $c \in \mathbb{Z}_N$.

Essentially, the RSA function (and its inverse) is a simple modular exponentiation. The most confusing thing to remember about RSA is that $e$ and $d$ "live" in $\mathbb{Z}^*_{\phi(N)}$, while $m$ and $c$ "live" in $\mathbb{Z}_N$.

Let's make sure the the function we called the "inverse RSA function" is actually in inverse of the RSA function. The RSA function raises its input to the $e$ power, and the inverse RSA function raises its input to the $d$ power. So it suffices to show that raising to the $ed$ power has no effect modulo $N$.

Since $ed \equiv_{\phi(N)} 1$, we can write $ed = t\phi(N) + 1$ for some integer $t$. Then:

$$(m^e)^d = m^{ed} = m^{t\phi(N)+1} = (m^{\phi(N)})^t m \equiv_N 1^t m = m$$

Note that we have used the fact that $m^{\phi(N)} \equiv_N 1$ from LaGrange's theorem.

to-do *Discuss computational aspects of modular exponentiation, and in general remind readers that efficiency of numerical algorithms is measured in terms of the* number of bits *needed to write the input.*

### Security Properties

In these notes we will not formally define a desired security property for RSA. Roughly speaking, the idea is that even when $N$ and $e$ can be made public, it should be hard to compute the operation $c \mapsto c^d \% N$. In other words, the RSA function $m \mapsto m^e \% N$ is:

- easy to compute given $N$ and $e$
- hard to invert given $N$ and $e$ but not $d$
- easy to invert given $d$

to-do *more details*

## 13.3 Chinese Remainder Theorem

The multiplicative group $\mathbb{Z}_N^*$ has some interesting structure when $N$ is the product of distinct primes. We can use this structure to optimize some algorithms related to RSA.

**History.** Some time around the 4th century CE, Chinese mathematician Sun Tzu Suan Ching discussed problems relating to simultaneous equations of modular arithmetic:

> *"We have a number of things, but we do not know exactly how many. If we count them by threes we have two left over. If we count them by fives we have three left over. If we count them by sevens we have two left over. How many things are there?"*[3]

In our notation, he is asking for a solution $x$ to the following system of equations:

$$x \equiv_3 2$$
$$x \equiv_5 3$$
$$x \equiv_7 2$$

A generalized way to solve equations of this kind was later given by mathematician Qin Jiushao in 1247 CE. For our eventual application to RSA, we will only need to consider the case of two simultaneous equations.

Theorem 13.3 (CRT) Suppose $\gcd(r,s) = 1$. Then for all integers $u, v$, there is a solution for $x$ in the following system of equations:

$$x \equiv_r u$$
$$x \equiv_s v$$

Furthermore, this solution is *unique* modulo $rs$.

Proof Since $\gcd(r,s) = 1$, we have by Bezout's theorem that $1 = ar + bs$ for some integers $a$ and $b$. Furthermore, $b$ and $s$ are multiplictaive inverses modulo $r$. Now choose $x = var + ubs$. Then,

$$x = var + ubs \equiv_r (va)0 + u(s^{-1}s) = u$$

So $x \equiv_r u$, as desired. By a symmetric argument, we can see that $x \equiv_s v$, so $x$ is a solution to the system of equations.

To see why the solution is *unique* modulo $rs$, suppose there are two solutions $x$ and $x'$ with:

$$x \equiv_r u \qquad x' \equiv_r u$$
$$x \equiv_s v \qquad x' \equiv_s v$$

Subtracting similar equations gives:

$$x - x' \equiv_r u - u = 0$$

[3] Translation due to Joseph Needham, *Science and Civilisation in China, vol. 3: Mathematics and Sciences of the Heavens and Earth*, 1959.

$$x - x' \equiv_s v - v = 0$$

We see that $x - x'$ must be both a multiple of $r$ and a multiple of $s$. Since $r$ and $s$ are relatively prime, $x - x'$ must be a multiple of $rs$. That is, $x \equiv_{rs} x'$. So any two solutions to this system of equations are congruent mod $rs$. ■

We can associate every pair $(u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s$ with its corresponding system of equations of the above form (with $u$ and $v$ as the right-hand-sides). The CRT suggests a relationship between these pairs $(u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s$ and elements of $\mathbb{Z}_{rs}$.

For $x \in \mathbb{Z}_{rs}$, and $(u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s$, let us write

$$x \overset{\text{crt}}{\longleftrightarrow} (u, v)$$

to mean that $x$ is a solution to $x \equiv_r u$ and $x \equiv_s v$. The CRT says that the $\overset{\text{crt}}{\longleftrightarrow}$ relation is a 1-to-1 correspondence between elements of $\mathbb{Z}_{rs}$ and elements of $\mathbb{Z}_r \times \mathbb{Z}_s$.

In fact, the relationship is even deeper than that. Consider the following observations:

1. If $x \overset{\text{crt}}{\longleftrightarrow} (u, v)$ and $x' \overset{\text{crt}}{\longleftrightarrow} (u', v')$, then $x + x' \overset{\text{crt}}{\longleftrightarrow} (u + u', v + v')$. You can see this by adding relevant equations together from the system of equations.

2. If $x \overset{\text{crt}}{\longleftrightarrow} (u, v)$ and $x' \overset{\text{crt}}{\longleftrightarrow} (u', v')$, then $xx' \overset{\text{crt}}{\longleftrightarrow} (uu', vv')$. You can see this by multiplying relevant equations together from the system of equations.

3. Suppose $x \overset{\text{crt}}{\longleftrightarrow} (u, v)$. Then $\gcd(x, rs) = 1$ if and only if $\gcd(u, r) = \gcd(v, s) = 1$. In other words, the $\overset{\text{crt}}{\longleftrightarrow}$ relation is a 1-to-1 correspondence between elements of $\mathbb{Z}^*_{rs}$ and elements of $\mathbb{Z}^*_r \times \mathbb{Z}^*_s$.[4]

The bottom line is that the CRT demonstrates that $\mathbb{Z}_{rs}$ and $\mathbb{Z}_r \times \mathbb{Z}_s$ **are essentially the same mathematical object.** In the terminology of abstract algebra, the two structures are *isomorphic.*

Think of $\mathbb{Z}_{rs}$ and $\mathbb{Z}_r \times \mathbb{Z}_s$ being two different sets of *names* for the same set of items. If we know the "$\mathbb{Z}_{rs}$-names" of two items, we can add them (mod $rs$) to get the $\mathbb{Z}_{rs}$-name of the result. If we know the "$\mathbb{Z}_r \times \mathbb{Z}_s$-names" of two items, we can add them (component-by-component) to get the $\mathbb{Z}_r \times \mathbb{Z}_s$-name of the result. The CRT says that both of these ways of adding result in the same results.

Additionally, the proof of the CRT shows us how to convert between these styles of names for a given object. So given $x \in \mathbb{Z}_{rs}$, we can compute $(x \% r, x \% s)$, which is the corresponding element/name in $\mathbb{Z}_r \times \mathbb{Z}_s$. Given $(u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s$, we can compute $x = var + ubs$ (where $a$ and $b$ are computed from the extended Euclidean algorithm) to obtain the corresponding element/name $x \in \mathbb{Z}_{rs}$.

From a **mathematical** perspective, $\mathbb{Z}_{rs}$ and $\mathbb{Z}_r \times \mathbb{Z}_s$ are the same object. However, from a **computational** perspective, there might be reason to favor one over the other. In fact, it turns out that doing computations in the $\mathbb{Z}_r \times \mathbb{Z}_s$ realm is significantly cheaper.

[4] Fun fact: this yields an alternative proof that $\phi(pq) = (p-1)(q-1)$ when $p$ and $q$ are prime. That is, $\phi(pq) = |\mathbb{Z}^*_{pq}| = |\mathbb{Z}^*_p \times \mathbb{Z}^*_q| = (p-1)(q-1)$.

### Application to RSA

In the context of RSA decryption, we are interested in taking $c \in \mathbb{Z}_{pq}$ and computing $c^d \in \mathbb{Z}_{pq}$. Since $p$ and $q$ are distinct primes, $\gcd(p, q) = 1$ and the CRT is in effect.

Thinking in terms of $\mathbb{Z}_{pq}$-arithmetic, raising $c$ to the $d$ power is rather straightforward. However, the CRT suggests that another approach is possible: We could convert $c$ into its $\mathbb{Z}_p \times \mathbb{Z}_q$ representation, do the exponentiation under that representation, and then convert back into the $\mathbb{Z}_{pq}$ representation. This approach corresponds to the bold arrows in Figure 13.1, and the CRT guarantees that the result will be the same either way.

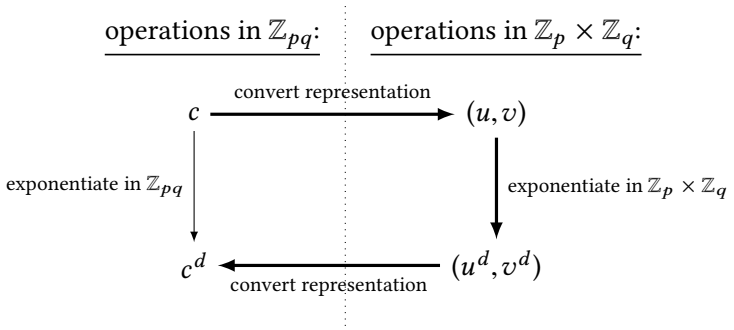


**Figure 13.1:** *Two ways to compute $c \mapsto c^d$ in $\mathbb{Z}_{pq}$.*

Now why would we ever want to compute things this way? Performing an exponentiation modulo an $n$-bit number requires about $n^3$ steps. Let's suppose that $p$ and $q$ are each $n$ bits long, so that the RSA modulus $N$ is $2n$ bits long. Performing $c \mapsto c^d$ modulo $N$ therefore costs about $(2n)^3 = 8n^3$ total.

The CRT approach involves two modular exponentiations — one mod $p$ and one mod $q$. Each of these moduli are only $n$ bits long, so the total cost is $n^3 + n^3 = 2n^3$. **The CRT approach is 4 times faster!** Of course, we are neglecting the cost of converting between representations, but that cost is very small in comparison to the cost of exponentiation.

It's worth pointing out that this speedup can only be done for the RSA *inverse* function. One must know $p$ and $q$ in order to exploit the Chinese Remainder Theorem, and only the party performing the RSA inverse function typically knows this.

## 13.4 The Hardness of Factoring $N$

Clearly the hardness of RSA is related to the hardness of factoring the modulus $N$. Indeed, if you can factor $N$, then you can compute $\phi(N)$, solve for $d$, and easily invert RSA. So factoring must be *at least as hard as* inverting RSA.

Factoring integers (or, more specifically, factoring RSA moduli) is believed to be a hard problem for classical computers.[5] In this section we show that some other problems related to RSA are "as hard as factoring." What does it mean for a computational problem to be "as hard as factoring?" More formally, in this section we will show the following:

[5] A polynomial-time algorithm for factoring is known for quantum computers.

Theorem 13.4 *Either **all** of the following problems can be solved in polynomial-time, or **none** of them can:*

1. *Given an RSA modulus $N = pq$, compute its factors $p$ and $q$.*
2. *Given an RSA modulus $N = pq$ compute $\phi(N) = (p-1)(q-1)$.*
3. *Given an RSA modulus $N = pq$ and value $e$, compute its inverse $d$, where $ed \equiv_{\phi(N)} 1$.*
4. *Given an RSA modulus $N = pq$, find any $x \not\equiv_N \pm 1$ such that $x^2 \equiv_N 1$.*

To prove the theorem, we will show:

- *if* there is an efficient algorithm for (1), *then* we can use it as a subroutine to construct an efficient algorithm for (2). This is straight-forward: if you have a subroutine factoring $N$ into $p$ and $q$, then you can call the subroutine and then compute $(p-1)(q-1)$.
- *if* there is an efficient algorithm for (2), *then* we can use it as a subroutine to construct an efficient algorithm for (3). This is also straight-forward: if you have a subroutine computing $\phi(N)$ given $N$, then you can compute the multiplicative inverse of $e$ using the extended Euclidean algorithm.
- *if* there is an efficient algorithm for (3), *then* we can use it as a subroutine to construct an efficient algorithm for (4).
- *if* there is an efficient algorithm for (4), *then* we can use it as a subroutine to construct an efficient algorithm for (1).

Below we focus on the final two implications.

### Using square roots of unity to factor $N$

Problem (4) of Theorem 13.4 concerns a new concept known as square roots of unity:

Definition 13.5 (Sqrt of unity) *$x$ is a **square root of unity modulo** $N$ if $x^2 \equiv_N 1$. If $x \not\equiv_N 1$ and $x \not\equiv_N -1$, then we say that $x$ is a **non-trivial** square root of unity.*

Note that $\pm 1$ are always square roots of unity modulo $N$, for any $N$. But if $N$ is the product of distinct odd primes, then $N$ has 4 square roots of unity: two trivial and two non-trivial ones (see the exercises in this chapter).

Claim 13.6 *Suppose there is an efficient algorithm for computing nontrivial square roots of unity modulo $N$. Then there is an efficient algorithm for factoring $N$. (This is the (4) $\Rightarrow$ (1) step in Theorem 13.4.)*

Proof The reduction is rather simple. Suppose NTSRU is an algorithm that on input $N$ returns a non-trivial square root of unity modulo $N$. Then we can factor $N$ with the following algorithm:

```
FACTOR(N):
  x := NTSRU(N)
  return gcd(N, x + 1) and gcd(N, x − 1)
```

The algorithm is simple, but we must argue that it is correct. When $x$ is a nontrivial square root of unity modulo $N$, we have the following:

$$\begin{aligned} x^2 &\equiv_{pq} 1 && \Rightarrow pq \mid x^2 - 1 && \Rightarrow pq \mid (x+1)(x-1) \\ x &\not\equiv_{pq} 1 && && \Rightarrow pq \nmid (x-1) \\ x &\not\equiv_{pq} -1 && && \Rightarrow pq \nmid (x+1) \end{aligned}$$

So the prime factorization of $(x+1)(x-1)$ contains a factor of $p$ and a factor of $q$. But neither $x+1$ nor $x-1$ contain factors of *both* $p$ and $q$. Hence $x+1$ and $x-1$ must each contain factors of exactly one of $\{p,q\}$, and $\{\gcd(pq, x-1), \gcd(pq, x+1)\} = \{p,q\}$. ■

### Finding square roots of unity

Claim 13.7 *If there is an efficient algorithm for computing $d \equiv_{\phi(N)} e^{-1}$ given $N$ and $e$, then there is an efficient algorithm for computing nontrivial square roots of unity modulo $N$. (This is the (3) ⇒ (4) step in Theorem 13.4.)*

Proof Suppose we have an algorithm FIND_D that on input $(N,e)$ returns the corresponding exponent $d$. Then consider the following algorithm which uses FIND_D as a subroutine:

```
SRU(N):
  choose e as a random n-bit prime
  d := FIND_D(N, e)
  write ed − 1 = 2^s r, with r odd
  // i.e., factor out as many 2s as possible
  w ← Z_N
  if gcd(w, N) ≠ 1: // w ∉ Z*_N
    use gcd(w, N) to factor N = pq
    compute a nontrivial square root of unity using p & q
  x := w^r % N
  if x ≡_N 1 then return 1
  for i = 0 to s:
    if x^2 ≡_N 1 then return x
    x := x^2 % N
```

There are several return statements in this algorithm, and it should be clear that all of them indeed return a square root of unity. Furthermore, the algorithm does eventually return within the main for-loop, because $x$ takes on the sequence of values:

$$w^r, w^{2r}, w^{4r}, w^{8r}, \ldots, w^{2^s r}$$

and the final value of that sequence satisfies

$$w^{2^s r} = w^{ed-1} \equiv_N w^{(ed-1)\%\phi(N)} = w^{1-1} = 1.$$

Conditioned on $w \in \mathbb{Z}_N^*$, it is possible to show that SqrtUnity$(N,e,d)$ returns a square root of unity *chosen uniformly at random* from among the four possible square roots of unity. So with probability 1/2, the output is a nontrivial square root. We can repeat this basic process $n$ times, and eventually encounter a nontrivial square root of unity with probability $1 - 2^{-n}$. ■

to-do *more complete analysis*

## 13.5 Malleability of RSA, and Applications

We now discuss several surprising problems that turn out to be equivalent to the problem of inverting RSA. The results in this section rely on the following *malleability* property of RSA: Suppose you are given $c = m^e$ for an unknown message $m$. Assuming $e$ is public, you can easily compute $c \cdot x^e = (mx)^e$. In other words, given the RSA function applied to $m$, it is possible to obtain the RSA function applied to a related message $mx$.

### Inverting RSA on a small subset

Suppose you had a subroutine INVERT$(N,e,c)$ that inverted RSA (*i.e.*, returned $c^d \bmod N$) but only for, say, 1% of all possible $c$'s. That is, there exists some subset $G \subseteq \mathbb{Z}_N$ with $|G| \geq N/100$, such that for all $c \in G$ we have $c \equiv_N (\text{INVERT}(N,e,c))^e$.

If you happen to have a value $c \notin G$, then it's not so clear how useful such a subroutine INVERT could be to you. However, it turns out that the subroutine can be used to invert RSA on *any input whatsoever.* Informally, if inverting RSA is easy on 1% of inputs, then inverting RSA is easy *everywhere.*

Assuming that we have such an algorithm INVERT, then this is how we can use it to invert RSA on any input:

```
REALLYINVERT(N,e,c):
  do:
    r ← Z_N
    c' := c · r^e % N
    m' := INVERT(N,e,c')
    m := m'/r
  repeat if m^e ≢_N c
  return m
```

Suppose the input to REALLYINVERT involves $c = (m^*)^e$ for some unknown $m^*$. The goal is to output $m^*$.

In the main loop, $c'$ is constructed to be an RSA encoding of $m^* \cdot r$. Since $r$ is uniformly distributed in $\mathbb{Z}_N$, so is $c'$. So the probability of $c'$ being in the "good set" $G$ is 1%. Furthermore, when $c'$ is indeed in the good set, INVERT inverts $c'$ correctly, returning $m^* \cdot r$. And in that case, REALLYINVERT outputs the correct answer $m^*$.

Each time through the main loop incurs a 1% chance of successfully inverting the given $c$. Therefore the expected running time of REALLYINVERT is $1/0.01 = 100$ times through the main loop.

### Determining high-order bits of $m$

Consider the following problem: Given $c = m^e \bmod N$ for an unknown $m$, determine whether $m > N/2$ or $m < N/2$. That is, does $m$ live in the top half or bottom half of $\mathbb{Z}_N$?

We show a surprising result that even this limited amount of information is enough to completely invert RSA. Equivalently, if inverting RSA is hard, then it is not possible to tell whether $m$ is in the top half or bottom half of $\mathbb{Z}_N$ given $m^e \,\%\, N$.

The main idea is that we can do a kind of binary search in $\mathbb{Z}_N$. Suppose TOPHALF$(N, e, c)$ is a subroutine that can tell whether $c^d \bmod N$ is in $\{0, \ldots, \frac{N-1}{2}\}$ or in $\{\frac{N+1}{2}, \ldots, N-1\}$. Given a candidate $c$, we can call TOPHALF to reduce the possible range of $m$ from $\mathbb{Z}_N$ to either the top or bottom half. Consider the ciphertext $c' = c \cdot 2^e$, which encodes $2m$. We can use TOPHALF to now determine whether $2m$ is in the top half of $\mathbb{Z}_N$. If $2m$ is in the top half of $\mathbb{Z}_N$, then $m$ is in the top half of its current range. Using this approach, we can repeatedly query TOPHALF to reduce the search space for $m$ by half each time. In only $\log N$ queries we can uniquely identify $m$.

```
BSEARCH(N, e, c):
  lo := 0;    hi := N − 1
  for i = 1 to log N:
    mid := (hi + lo)/2
    if TOPHALF(N, e, c):
      hi := mid
    else:
      lo := mid
    c := c · 2^e
  return ⌊hi⌋
```

to-do *more complete analysis*

## Exercises

13.1. Prove by induction the correctness of the EXTGCD algorithm. That is, whenever $(d, a, b) =$ EXTGCD$(x, y)$, we have $\gcd(x, y) = d = ax + by$. You may use the fact that the original Euclidean algorithm correctly computes the GCD.

13.2. Prove that if $g^a \equiv_n 1$ and $g^b \equiv_n 1$, then $g^{\gcd(a,b)} \equiv_n 1$.

13.3. Prove that $\gcd(2^a - 1, 2^b - 1) = 2^{\gcd(a,b)} - 1$.

13.4. Prove that $x^a \,\%\, n = x^{a \% \phi(n)} \,\%\, n$. In other words, when working modulo $n$, you can reduce exponents modulo $\phi(n)$.

13.5. In this problem we determine the efficiency of Euclid's GCD algorithm. Since its input is a pair of numbers $(x, y)$, let's call $x + y$ the *size* of the input. Let $F_k$ denote the $k$th Fibonacci number, using the indexing convention $F_0 = 1$; $F_1 = 2$. Prove that $(F_k, F_{k-1})$ is the smallest-*size* input on which Euclid's algorithm makes $k$ recursive calls. *Hint:* Use induction on $k$.

Note that the *size* of input $(F_k, F_{k-1})$ is $F_{k+1}$, and recall that $F_{k+1} \approx \phi^{k+1}$, where $\phi \approx 1.618\ldots$ is the golden ratio. Thus, for any inputs of *size* $N \in [F_k, F_{k+1})$, Euclid's algorithm will make less than $k \leq \log_\phi N$ recursive calls. In other words, the worst-case number of recursive calls made by Euclid's algorithm on an input of *size* $N$ is $O(\log N)$, which is linear in the number of bits needed to write such an input.[6]

13.6. In 1937, Alan Turing proposed the following cipher, based on the difficulty of factoring:[7]

- Beforehand, Alice and Bob agree on a secret key $k$, which is a **large prime**.
- To encrypt an English plaintext $m$, Alice encodes $m$ as a large prime number $\widehat{m}$ in the following way:
  1. Replace A with "01", B with "02", etc.
  2. Append all these numbers together. For instance, the message VICTORY would become 22090320151825:

| V | I | C | T | O | R | Y |
|---|---|---|---|---|---|---|
| 22 | 09 | 03 | 20 | 15 | 18 | 25 |

  3. Append several digits to the resulting number so that the result is prime. For instance, $22090320151825 \rightsquigarrow 2209032015182513$ is prime. We assume this can be done efficiently, perhaps by adding sufficient 0s and then using the `nextprime` function in PARI.
  4. Call the resulting encoding $\widehat{m}$. So if $m$ = VICTORY, then $\widehat{m}$ could be 2209032015182513

  Then Alice computes the integer $c = \widehat{m} \cdot k$. The ciphertext is $c$.
- To decrypt $c$, Bob computes $\widehat{m} = c/k$, which is guaranteed to be a (prime) integer, and then decodes $\widehat{m}$ into the corresponding English message $m$.

Show that if Alice encrypts and sends two messages to Bob, then the eavesdropper can easily recover the secret key. The eavesdropper need not know the two messages sent.

13.7. Explain why the RSA encryption exponent $e$ must always be an odd number.

13.8. The Chinese Remainder Theorem states that there is always a solution for $x$ in the following system of equations, when $\gcd(r, s) = 1$:

$$x \equiv_r u$$

[6] A more involved calculation that incorporates the cost of each division (modulus) operation shows the worst-case overall efficiency of the algorithm to be $O(\log^2 N)$ — quadratic in the number of bits needed to write the input.

[7] He may have been the first to recognize that number theory and intractibility could be used for practical purposes.

$$x \equiv_s v$$

Give an example $u$, $v$, $r$, $s$, with $\gcd(r,s) \neq 1$ for which the equations have no solution. Explain why there is no solution.

13.9. Bob chooses an RSA plaintext $m \in \mathbb{Z}_N$ and encrypts it under Alice's public key as $c \equiv_N m^e$. To decrypt, Alice first computes $m_p \equiv_p c^d$ and $m_q \equiv_q c^d$, then uses the CRT conversion to obtain $m \in \mathbb{Z}_N$, just as expected. But suppose Alice is using faulty hardware, so that she computes a **wrong value** for $m_q$. The rest of the computation happens correctly, and Alice computes the (wrong) result $\hat{m}$. Show that, no matter what $m$ is, and no matter what Alice's computational error was, Bob can factor $N$ if he learns $\hat{m}$.

*Hint:* Bob knows $m$ and $\hat{m}$ satisfying the following:

$$m \equiv_p \hat{m}$$
$$m \not\equiv_q \hat{m}$$

13.10. (a) Show that given an RSA modulus $N$ and $\phi(N)$, it is possible to factor $N$ easily.
*Hint:* you have two equations (involving $\phi(N)$ and $N$) and two unknowns ($p$ and $q$).

(b) Write a `pari` function that takes as input an RSA modulus $N$ and $\phi(N)$ and factors $N$. Use it to factor the following 2048-bit RSA modulus. *Note:* take care that there are no precision issues in how you solve the problem; double-check your factorization!

```
N = 133140272889335192922108409260662174476303831652383671688547009484
    253235940586917140482669182256368285260992829447207980183170174867
    620358952230969986447559330583492429636627298640338596531894556546
    013113154346823212271748927859647994534586133553218022983848108421
    465442089919090610542344768294481725103757222421917115971063026806
    587141287587037265150653669094323116686574536558866591647361053311
    046516013069669036866734126558017744393751161611219195769578488559
    882902397248309033911661475005854696820021069072502248533328754832
    698616238405221381252145137439919090800085955274389382721844956661
    1138745095472005761807
phi = 133140272889335192922108409260662174476303831652383671688547009484
    253235940586917140482669182256368285260992829447207980183170174867
    620358952230969986447559330583492429636627298640338596531894556546
    013113154346823212271748927859647994534586133553218022983848108421
    465442089919090610542344768294481725103757214932292046538867218497
    635256772227370109066785312096589779622355495419006049974567895189
    687318110498058692315630856693672069320529062399681563590382015177
    322909744749330702607931428154183726552004527201956226396835500346
    779062494259638983191178915027835134527751607017859064511731520440
    2981816860178885028680
```

13.11. True or false: if $x^2 \equiv 1 \pmod{N}$ then $x \in \mathbb{Z}_N^*$. Prove or give a counterexample.

13.12. Discuss the computational difficulty of the following problem:

*Given an integer $N$, find an element of $\mathbb{Z}_N \setminus \mathbb{Z}_N^*$.*

If you can, relate its difficulty to that of other problems we've discussed.

13.13. (a) Show that it is possible to efficiently compute all four square roots of unity modulo $pq$, given $p$ and $q$. *Hint:* CRT!

(b) Implement a `pari` function that takes distinct primes $p$ and $q$ as input and returns the four square roots of unity modulo $pq$. Use it to compute the four square roots of unity modulo

$$N = 105295498644227198587577819263 \times 611174539744122090068393470777.$$

★ 13.14. Show that, conditioned on $w \in \mathbb{Z}_N^*$, the SqrtUnity subroutine outputs a square root of unity chosen uniformly at random from the 4 possible square roots of unity. *Hint:* use the Chinese Remainder Theorem.

13.15. Suppose $N$ is an RSA modulus, and $x^2 \equiv_N y^2$, but $x \not\equiv_N \pm y$. Show that $N$ can be efficiently factored if such a pair $x$ and $y$ are known.

13.16. Why are $\pm 1$ the only square roots of unity modulo $p$, when $p$ is an odd prime?

13.17. When $N$ is an RSA modulus, why is squaring modulo $N$ a 4-to-1 function, but raising to the $e^{\text{th}}$ power modulo $N$ is 1-to-1?

13.18. Implement a `pari` function that efficiently factors an RSA modulus $N$, given only $N$, $e$, and $d$. Use your function to factor the following 2048-bit RSA modulus. *Note:* `pari` function `valuation(n,p)` returns the largest number $d$ such that $p^d \mid n$.

```
N = 157713892705550064909750632475691896977526767652833932128735618711
    213662561319634033137058267272367265499003291937716454788882499492
    311117065951077245304317542978715216577264400048278064574204140564
    709253009840166821302184014310192765595015483588878761062406993721
    851190041888790873152584082212461847511180066690936944585390792304
    663763886417861546718283897613617078370412411019301687497005038294
    389148932398661048471814117247898148030982257697888167001010511378
    647288478239379740416388270380035364271593609513220655573614212415
    962670795230819103845127007912428958291134064942068225836213242131
    15022256956985205924967
e = 327598866483920224268285375349315001772252982661926675504591773242
    501030864502336359508677092544631083799700755236766113095163469666
    905258066495934057774395712118774014408282455244138409433389314036
    198045263991986560198273156037233588691392913730537367184867549274
    682884119866630822924707702796323546327425328705958528315517584489
    590815901470874024949798420173098581333151755836650797037848765578
    433873141626191257009250151327378074817106208930064676608134109788
    601067077103742326030259629322458620311949453584045538305945217564
    027461013225009980998673160144967719374426764116721861138496780008
    6366258360757218165973
```

```
d = 13847699973426377549810044356713275918214457347447401419502109127 2
    75520780316201948448712786667542260840199088894265939341938452825 7
    46243463373868617660155575584218998643172533503162009785496229596 8
    39116109082638045896923641858596338471740670471483734950380878608 6
    70157376571482578304229734405052889825974575774123309929795233201 2
    74989728109037839800133705786918948873495185374832763188350213513 9
    52366499029633402032771390040868326423266464543889917844263334243 8
    19832998312120731543644704191589754444540250555842013850665510601 5
    21545014025612997738247606236651908738657687488693858578987418632 6
    69265500594424847344765
```

13.19. In this problem we'll see that it's bad to choose RSA prime factors $p$ and $q$ too close together.

(a) Let $s = (p - q)/2$ and $t = (p + q)/2$. Then $t$ is an integer greater than $\sqrt{N}$ such that $t^2 - N$ is a perfect square. When $p$ and $q$ are close, $t$ is not much larger than $\sqrt{N}$, so by testing successive integers, it is possible to find $t$ and $s$, and hence $p$ and $q$. Describe the details of this attack and how it works.

(b) Implement a `pari` function that factors RSA moduli using this approach. Use it to factor the following 2048-bit number (whose two prime factors are guaranteed to be close enough for the factoring approach to work in a reasonable amount of time, but far enough apart that you can't do the trial-and-error part by hand). What qualifies as "close prime factors" in this problem? How close was $t$ to $\sqrt{N}$?

*Hint:* `pari` has an `issquare` function. Also, be sure to do exact square roots over the integers, not the reals.

```
N = 51420286866426650198673634022634388019321686401164324455870195611 4
    55331788004328982748745646028410395146351202424932924322810962401 1
    91539241188872402640312768670725582505608189069259571582838069081 1
    13168638318028233077557238582210218120956941196112575324246797187 9
    13130598698652560011034079059598797534557384226676649235668676213 4
    65383306451133743308924962125762910782568142957393494910130113520 0
    91860621139441349873548659967854136937588784001384243902615903710 8
    04372422186511679403419481223638129978639545727755987957575225411 6
    61272659611852807178547455105854059919886998678028673391661433566 3
    3723003246569630373323
```