13 RSA & Digital Signatures

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] RSA can be used as a building block for public-key encryption and digital signatures. In this chapter we discuss only the application of RSA for digital signatures.

13.1 "Dividing" Mod n

to-do *I'm considering moving some of this material to Chapter 3 (secret sharing) — enough to understand that every nonzero element has a multiplicative inverses modulo a prime (totients, etc can stay here). That way, I don't have to say "trust me, this can be made to work" when describing Lagrange interpolation over a prime field, and students can play around with secret sharing using Sage. Also, students will see that there is "serious math" in the course already in chapter 3 so they don't get blindsided as we transition into public-key crypto. (Not to mention, this chapter is too long.)*

Please review the material from Section 0.2, to make sure your understanding of basic modular arithmetic is fresh. You should be comfortable with the definitions of $\mathbb{Z}_n$, congruence ($\equiv_n$), the modulus operator (%), and how to do addition, multiplication, and subtraction mod n.

Note that we haven't mentioned *division* mod n. Does it even make sense to talk about division mod n?

Example *Consider the following facts which hold mod 15:*

$$\begin{aligned} 2 \cdot 8 &\equiv_{15} 1 & \qquad 10 \cdot 8 &\equiv_{15} 5 \\ 4 \cdot 8 &\equiv_{15} 2 & 12 \cdot 8 &\equiv_{15} 6 \\ 6 \cdot 8 &\equiv_{15} 3 & 14 \cdot 8 &\equiv_{15} 7 \\ 8 \cdot 8 &\equiv_{15} 4 \end{aligned}$$

Now imagine replacing "$\cdot\, 8$" with "$\div\, 2$" in each of these examples:

$$\begin{aligned} 2 \div 2 &\equiv_{15} 1 & \qquad 10 \div 2 &\equiv_{15} 5 \\ 4 \div 2 &\equiv_{15} 2 & 12 \div 2 &\equiv_{15} 6 \\ 6 \div 2 &\equiv_{15} 3 & 14 \div 2 &\equiv_{15} 7 \\ 8 \div 2 &\equiv_{15} 4 \end{aligned}$$

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

Everything still makes sense! Somehow, multiplying by 8 mod 15 seems to be the same thing as "dividing by 2" mod 15.

The previous examples all used $x \cdot 8$ $(x \div 2)$ where x was an even number. What happens when x is an odd number?

$$3 \cdot 8 \equiv_{15} 9 \iff \text{“}3 \div 2 \equiv_{15} 9\text{” ??}$$

This might seem non-sensical, but if we make the substitutions $3 \equiv_{15} -12$ and $9 \equiv_{15} -6$, then we do indeed get something that makes sense:

$$-12 \cdot 8 \equiv_{15} -6 \iff -12 \div 2 \equiv_{15} -6$$

This example shows that there is surely some interesting relationship among the numbers 2, 8, and 15. It seems reasonable to interpret "multiplication by 8" as "division by 2" when working mod 15.

Is there a way we can do something similar for "division by 3" mod 15? Can we find some y where "multiplication by y mod 15" has the same behavior as "division by 3 mod 15?" In particular, we would seek a value y that satisfies $3 \cdot y \equiv_{15} 1$, but you can check for yourself that **no such value of y exists.**

Why can we "divide by 2" mod 15 but we apparently cannot "divide by 3" mod 15? We will explore this question in the remainder of this section.

Multiplicative Inverses

We usually don't directly use the terminology of "division" with modular arithmetic. Instead of saying "division by 2", we say "multiplication by 2^{-1}", where 2^{-1} is just another name for 8.

Definition 13.1 (x^{-1} mod n)

*The **multiplicative inverse** of x mod n is the integer y that satisfies $x \cdot y \equiv_n 1$ (if such a number exists). We usually refer to the multiplicative inverse of x as "x^{-1}."*

Example

Contuining to work mod 15, we have:

- $4^{-1} \equiv_{15} 4$ *since* $4 \cdot 4 = 16 \equiv_{15} 1$. *Hence 4 is its own multiplicative inverse! You can also understand this as:*

$$4^{-1} = (2^2)^{-1} = (2^{-1})^2 \equiv_{15} 8^2 = 64 \equiv_{15} 4$$

- $7^{-1} \equiv_{15} 13$ *since* $7 \cdot 13 = 91 \equiv_{15} 1$.

We are interested in which numbers have a multiplicative inverse mod n.

Definition 13.2 ($\mathbb{Z}_n^*$)

*The **multiplicative group**[2] **modulo** n is defined as:*

$$\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n \mid x \text{ has a multiplicative inverse mod } n\}$$

[2] "Group" is a technical term from abstract algebra.

For example, we have seen that $\mathbb{Z}_n^*$ contains the numbers 2, 4, and 7 (and perhaps others), but it doesn't contain the number 3 since 3 does not have a multiplicative inverse.

So which numbers have a multiplicative inverse mod n, in general? (Which numbers belong to $\mathbb{Z}*_n$?) The answer is quite simple:

Theorem 13.3 *x has a multiplicative inverse mod n **if and only if** $\gcd(x, n) = 1$. In other words, $\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n \mid \gcd(x, n) = 1\}$.*

We prove the theorem using another fact from abstract algebra which is often useful:

Theorem 13.4 (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.*

We won't prove Bezout's theorem, but we will show how it is used to prove Theorem 13.3:

Proof (of Theorem 13.3) ($\Leftarrow$) Suppose $\gcd(x, n) = 1$. We will show that x has a multiplicative inverse mod n. From Bezout's theorem, there exist integers a, b satisfying $ax + bn = 1$. By reducing both sides of this equation modulo n, we have

$$1 = ax + bn \equiv_n ax + b \cdot 0 = ax.$$

Thus the integer a that falls out of Bezout's theorem is the multiplicative inverse of x modulo n.

($\Rightarrow$) Suppose x has a multiplicative inverse mod n, so $xx^{-1} \equiv_n 1$. We need to show that $\gcd(x, n) = 1$. From the definition of $\equiv_n$, we know that 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 $xx^{-1} - kn = 1$. Since we can write 1 as an integral linear combination of x and n, Bezout's theorem says that we must have $\gcd(x, n) = 1$. ■

Example *$\mathbb{Z}_{15} = \{0, 1, \ldots, 14\}$, and to obtain $\mathbb{Z}_{15}^*$ we exclude any of the numbers that share a common factor with 15. In other words, we exclude the multiples of 3 and multiples of 5. The remaining numbers are $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$.*

Since 11 is a prime, 0 is the only number in $\mathbb{Z}_{11}$ that shares a common factor with 11. All the rest satisfy $\gcd(x, 11) = 1$. Hence, $\mathbb{Z}_{11}^* = \{1, 2, \cdots, 10\}$.

Example *We can use **Sage**[3] to play around with these concepts. Sage supports the `%` operator for modulus:*

```
sage: 2*8 % 15
1
```

It also supports a convenient way to generate "$\mathbb{Z}_n$-objects," or `Mod`-objects as they are called. An object like `Mod(2,15)` represents the value $2 \in \mathbb{Z}_{15}$, and all of its operations are overloaded to be the mod-15 operations:

[3] https://www.sagemath.org

```
sage: Mod(2,15)*8
1
sage: Mod(2,15)+31415926
3
sage: Mod(-1,15)
14
```

In Sage, you can compute multiplicative inverses in a few different ways:

```
sage: Mod(2,15)^-1
8
sage: 1/Mod(2,15)
8
sage: 2.inverse_mod(15)
8
sage: (1/2) % 15
8
```

Sage is smart enough to know when a multiplicative inverse doesn't exist:

```
sage: Mod(3,15)^-1
ZeroDivisionError: inverse of Mod(3, 15) does not exist
```

Sage supports huge integers, with no problem:

```
sage: n = 3141592653589793238462643383279502884197169399375105820974944
sage: x = 1234567890123456789012345678901234567890123456789012345678901
sage: 1/Mod(x,n)
2234412539909122491686747985730075304931040310346724620855837
```

The relationship between multiplicative inverses and GCD goes even farther than Theorem 13.3. Recall that we can compute $\gcd(x, n)$ efficiently using Euclid's algorithm. There is a relatively simple modification to Euclid's algorithm that also computes the corresponding Bezout coefficients with little extra work. In other words, given x and n, it is possible to efficiently compute integers a, b, and d such that

$$ax + bn = d = \gcd(x, n)$$

In the case where $\gcd(x, n) = d = 1$, the integer a is a multiplicative inverse of x mod n. The "extended Euclidean algorithm" for GCD is given below:

$\underline{\textsc{extgcd}(x, y)}$:
  // *returns* (d, a, b) *such that* $\gcd(x, y) = d = ax + by$
  if $y = 0$:
    return $(x, 1, 0)$
  else:
    $(d, a, b) := \textsc{extgcd}(y, x \% y)$
    return $(d, b, a - b\lfloor x/y \rfloor)$

Example *Sage implements the extended Euclidean algorithm as "xgcd":*

```
sage: xgcd(427,529)
(1, 223, -180)
sage: 223*427 + (-180)*529
1
```

You can then see that 223 and 427 are multiplicative inverses mod 529:

```
sage: 427*223 % 529
1
```

The Totient Function

Euler's **totient** function is defined as $\phi(n) \stackrel{\text{def}}{=} |\mathbb{Z}_n^*|$; that is, the number of elements of Z_n that have multiplicative inverses.

As an example, if n 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 n. Therefore, $\phi(n) = n - 1$ in this case.

RSA involves a modulus n that 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 since p and q are relatively prime, their least common multiple is pq, which is larger than any item in these lists.

We count $p + q - 1$ elements of $\mathbb{Z}_{pq}$ which share a common divisor with pq. The rest belong to $\mathbb{Z}_{pq}^*$, 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.

The reason we consider $\phi(n)$ at all is this fundamental theorem from abstract algebra:

Theorem 13.5 (Euler's Theorem) *If $x \in \mathbb{Z}_n^*$ then $x^{\phi(n)} \equiv_n 1$.*

Example *Using the formula for $\phi(n)$, we can see that $\phi(15) = \phi(3 \cdot 5) = (3-1)(5-1) = 8$. Euler's theorem says that raising any element of $\mathbb{Z}_{15}^*$ to the 8 power results in 1: We can use Sage to verify this:*

```
sage: for i in range(15):
....:     if gcd(i,15) == 1:
....:         print("%d^8 mod 15 = %d" % (i, i^8 % 15))
....:
1^8 mod 15 = 1
2^8 mod 15 = 1
4^8 mod 15 = 1
7^8 mod 15 = 1
8^8 mod 15 = 1
11^8 mod 15 = 1
13^8 mod 15 = 1
14^8 mod 15 = 1
```

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!
- The RSA function is: $x \mapsto x^e \,\%\, N$, where $x \in \mathbb{Z}_N$.
- The inverse RSA function is: $y \mapsto y^d \,\%\, N$, where $x \in \mathbb{Z}_N$.

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 x and y "live" in $\mathbb{Z}_N$.

$$\begin{tikzcd} x \arrow[r, bend left, "\text{raise to } e \text{ power (mod } N\text{)}"] & y \arrow[l, bend left, "\text{raise to } d \text{ power (mod } N\text{)}"] \end{tikzcd}$$

Let's make sure the function we called the "inverse RSA function" is actually an inverse of the RSA function. Let's start with an example:

Example *In Sage, we can sample a random prime between 1 and k by using* `random_prime(k)`*. We use it to sample the prime factors p and q:*

```
sage: p = random_prime(10^5)
sage: q = random_prime(10^5)
sage: N = p*q
sage: N
36486589
```

Then we can compute the exponents e and d. Recall that they must be multiplicative inverses mod $\phi(N)$, so they cannot share any common factors with $\phi(N)$. An easy way to ensure this is to choose e to be a prime:

```
sage: phi = (p-1)*(q-1)
sage: e = random_prime(phi)
sage: e
28931431
sage: d = 1/Mod(e,phi)
sage: d
31549271
```

We can now raise something to the e power and again to the d power:

```
sage: x = 31415926
sage: y = x^e % N
sage: y
1798996
sage: y^d % N
31415926
```

As you can see, raising to the e power and then d power (mod N) seems to bring us back to where we started (x).

We can argue that raising-to-the-e-power and raising-to-the-d-power are inverses in general: Since $ed \equiv_{\phi(N)} 1$, we can write $ed = t\phi(N) + 1$ for some integer t. Then:

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

Note that we have used the fact that $x^{\phi(N)} \equiv_N 1$ from Euler's theorem.[4]

How [Not] to Exponentiate Huge Numbers

When you see an expression like "$x^e \% N$", you might be tempted to implement it with the following algorithm:

$\underline{\textsc{NaiveExponentiate}(x, e, N)\text{:}}$
- $result = 1$
- for $i = 1$ to e: // *compute x^e*
  - $result = result \times x$
- return $result \% N$

While this algorithm would indeed give the correct answer, it is a really bad way of doing it. In practice, we use RSA with numbers that are thousands of bits long. Suppose we run the NaiveExponentiate algorithm with arguments x, e, and N which are around a thousand bits each (so the magnitude of these numbers is close to 2^{1000}):

[4]However, see Exercise 13.15.

1. The algorithm will spend approximately 2^{1000} iterations in the for-loop!
2. The algorithm computes x^e as an *integer* first, and then reduces that integer mod N. Observe that x^2 is roughly 2000 bits long, x^3 is roughly 3000 bits long, etc. So it would take about $2^{1000} \cdot 1000$ bits just to write down the integer x^e.

As you can see, there is neither enough time nor storage capacity in the universe to use this algorithm. So how can we actually compute values like $x^e \% N$ on huge numbers?

1. Suppose you were given an integer x and were asked to compute x^{17}. You can compute it as:

$$x^{17} = \underbrace{x \cdot x \cdot x \cdots x}_{\text{16 multiplications}}.$$

   But a more clever way is to observe that:

$$x^{17} = x^{16} \cdot x = (((x^2)^2)^2)^2 \cdot x.$$

   This expression can be evaluated with only 5 multiplications (squaring is just muliplying a number by itself).

   More generally, you can compute an expression like x^e by following the recurrence below. The method is called **exponentiation by repeated squaring**, for reasons that are hopefully clear:

$$x^e = \begin{cases} 1 & \text{if } e = 0 \\ (x^{\frac{e}{2}})^2 & \text{if } e \text{ even} \\ (x^{\frac{e-1}{2}})^2 \cdot x & \text{if } e \text{ odd} \end{cases}$$

```
BetterExp(x, e):
  if e = 0: return 1
  if e even:
    return BetterExp(x, e/2)^2
  if e odd:
    return BetterExp(x, (e-1)/2)^2 · x
```

   BetterExp divides the e argument by two (more or less) each time it recurses, until reaching the base case. Hence, the number of recursive calls is $O(\log e)$. In each recursive call there are only a constant number of multiplications (including squarings). So overall this algorithm requires only $O(\log e)$ multiplications (compared to $e - 1$ multiplications by just multiplying m by itself e times). In the case where $e \sim 2^{1000}$, this means only a few thousand multiplications.

2. We care about only $x^e \% N$, not the intermediate *integer* value x^e. One of the most fundamental features of modular arithmetic is that you can **reduce any intermediate values mod N** if you care about the final result only mod N.

   Revisiting our previous example:

$$x^{17} \% N = x^{16} \cdot x \% N = (((x^2 \% N)^2 \% N)^2 \% N)^2 \cdot x \% N.$$

   More generally, we can reduce all intermediate value mod N:

$\underline{\textsc{ModExp}(x, e, N)}$: // *compute* x^e % N
  if $e = 0$: return 1
  if e even:
    return $\textsc{ModExp}(x, \frac{e}{2}, N)^2$ % N
  if e odd:
    return $\textsc{ModExp}(x, \frac{e-1}{2}, N)^2 \cdot x$ % N

This algorithm avoids the problem of computing the astronomically huge integer x^e. It never needs to store any value (much) larger than N.

Warning: *Even this* ModExp *algorithm isn't an ideal way to implement exponentiation for cryptographic purposes. Exercise 13.10 explores some unfortunate properties of this exponentiation algorithm.*

Example *Most math libraries implement exponentiation using repeated squaring. For example, you can use Sage to easily calculate numbers with huge exponents:*

```
sage: 427^31415926 % 100
89
```

However, this expression still tells Sage to compute $427^{31415926}$ ***as an integer****, before reducing it mod 100. As such, it takes some time to perform this computation.*

If you try an expression like `x^e % N` *with a larger exponent, Sage will give a memory error. How can we tell Sage to perform modular reduction at every intermediate step during repeated squaring? The answer is to use Sage's* `Mod` *objects, for example:*

```
sage: Mod(427,100)^314159265358979
63
```

This expression performs repeated squaring on the object `Mod(427,100)`*. Since a* `Mod`*-object's operations are all overloaded (to give the answer only mod n), this has the result of doing a modular reduction after every squaring and multiplication. This expression runs instantaneously, even with very large numbers.*

Security Properties & Discussion

RSA is what is called a **trapdoor function.**

- One user generates the RSA parameters (primarily N, e, and d) and makes N and e public, while keeping d private.
- Functionality properties: Given only the public information N and e, it is easy to compute the RSA function ($x \mapsto x^e \% N$). Given the private information (d) it clearly easy to compute the RSA inverse ($y \mapsto y^d \% N$).
- **Security property:** Given only the public information, it should be hard to compute the RSA inverse ($y \mapsto y^d \% N$) on randomly chosen values. In other words, the only person who is able to compute the RSA inverse function is the person who generated the RSA parameters.

to-do *The security property is not natural to express in our language of security definitions (libraries).*

Currently the best known attacks against RSA (*i.e.*, ways to compute the inverse RSA function given only the public information) involve factoring the modulus. If we want to ensure that RSA is secure as a trapdoor function, we must understand the state of the art for factoring large numbers.

Before discussing the performance of factoring algorithms, remember that we measure performance as a function of the **length** of the input — how many bits it takes to write the input. In a factoring algorithm, the input is a large number N, and it takes roughly $n = \log_2 N$ bits to write down that number. We will discuss the running time of algorithms as a function of n, not N. Just keep in mind the difference in cost between *writing down* a 1000-bit number ($n = 1000$) vs *counting up to* a 1000-bit number ($N = 2^{1000}$)

Everyone knows the "trial division" method of factoring: given a number N, check whether i divides N, for every $i \in \{2, \ldots \sqrt{N}\}$. This algorithm requires $\sqrt{N} = 2^{n/2}$ divisions in the worst case. It is an exponential-time algorithm since we measure performance in terms of the bit-length n.

If this were the best-known factoring algorithm, then we would need to make N only as large as 2^{256} to make factoring require 2^{128} effort. But there are much better factoring algorithms than trial division. The fastest factoring algorithm today is called the Generalized Number Field Sieve (GNFS), and its complexity is something like $O\left(n^{(\frac{n}{\log n})^{\frac{1}{3}}}\right)$. This is not a polynomial-time algorithm, but it's much faster than trial division.

Example *Sage can easily factor reasonably large numbers. Factoring the following 200-bit RSA modulus on my modest computer takes about 10 seconds:*

```
sage: p = random_prime(2^100)
sage: q = random_prime(2^100)
sage: N = p*q
sage: factor(N)
206533721079613722225064934611 * 517582080563726621130111418123
```

As of January 2020, the largest RSA modulus that has been (publically) factored is a 795-bit modulus.[5] Factoring this number required the equivalent of 900 CPU-core-years, or roughly 2^{66} total clock cycles.

All of this is to say, the numbers involved in RSA need to be quite large to resist factoring attacks (*i.e.*, require 2^{128} effort for state-of-the-art factoring algorithms). Current best practices suggest to use 2048- or 4096-bit RSA moduli, meaning that p and q are each 1024 or 2048 bits.

to-do *"What about quantum computers?" is a common FAQ that I should address here.*

[5] https://en.wikipedia.org/wiki/RSA_numbers#RSA-240

13.3 Digital Signatures

MACs are a cryptographic primitive that provide authenticity. A valid MAC tag on m is "proof" that someone who knows the key has vouched for m. MACs are a symmetric-key primitive, in the sense that generating a MAC tag and verifying a MAC tag both require the same key (in fact, a tag is verified by re-computing it).

Digital signatures are similar to MACs, but with separate keys for signing and verification. A digital signature scheme consists of the following algorithms:

- KeyGen: outputs a **pair** of keys (sk, vk), where sk is the **signing key** and vk is the **verification key**.
- Sign: takes the signing key sk and a message m as input, and outputs a **signature** σ.
- Ver: takes the verification key vk, message m, and a potential signature σ as input; outputs a boolean.

If indeed σ is an output of $\text{Sign}(sk, m)$, then $\text{Ver}(vk, m, \sigma)$ should output `true`. Intuitively, it should be hard for an attacker to find any other (m, σ) pairs that cause Ver to output `true`.

The idea behind digital signatures is to make vk public. In other words, anyone (even the attacker) should be able to verify signatures. But only the holder of sk (the person who generated vk and sk) should be able to generate valid signatures. Furthermore, this guarantee should hold even against an attacker who sees many examples of valid signatures. The attacker should not be able to generate *new* valid signatures.

We formalize this security property in a similar way that we formalized the security of MACs: "only the secret-key holder can generate valid tags, even after seeing chosen examples of valid tags."

Definition 13.6 *Let Σ be a signature scheme. We say that Σ is a **secure signature** if $\mathcal{L}^{\Sigma}_{\text{sig-real}} \approx \mathcal{L}^{\Sigma}_{\text{sig-fake}}$, where:*

$\mathcal{L}^{\Sigma}_{\text{sig-real}}$
$(vk, sk) \leftarrow \Sigma.\text{KeyGen}$
$\underline{\text{GETVK}()}$:
return vk
$\underline{\text{GETSIG}(m)}$:
return $\Sigma.\text{Sign}(sk, m)$
$\underline{\text{VERSIG}(m, \sigma)}$:
return $\Sigma.\text{Ver}(vk, m, \sigma)$

$\mathcal{L}^{\Sigma}_{\text{sig-fake}}$
$(vk, sk) \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$
$\underline{\text{GETVK}()}$:
return vk
$\underline{\text{GETSIG}(m)}$:
$\sigma := \Sigma.\text{Sign}(sk, m)$
$\mathcal{S} := \mathcal{S} \cup \{(m, \sigma)\}$
return σ
$\underline{\text{VERSIG}(m, \sigma)}$:
return $(m, \sigma) \stackrel{?}{\in} \mathcal{S}$

Similar to the security definition for MACs, the libraries differ only in how they verify signatures provided by the attacker (VERSIG). If the attacker can generate a message-signature pair (m, σ) that (1) verifies correctly, but (2) was not generated previously by the library itself, then VERSIG from the $\mathcal{L}_{\text{sig-real}}$ library will return `true`, while the $\mathcal{L}_{\text{sig-fake}}$ library would return `false`. By asking for the libraries to be indistinguishable, we are really asking that the attacker cannot find any such message-signature pair (forgery).

The main difference to the MAC definition is that, unlike for the MAC setting, we intend to make a verification key public. The library can run $(vk, sk) \leftarrow \text{KeyGen}$, but these values remain private by default. To make vk public, we explicitly provide an accessor GETVK to the attacker.

"Textbook" RSA Signatures

Signatures have an asymmetry: everyone should be able to verify a signature, but only the holder of the signing key should be able to generate a valid signature. The RSA function has a similar asymmetry: if N and e are public, then anyone can raise things to the e power, but only someone with d can raise things to the d power.

This similarity suggests that we can use RSA for signatures in the following way:

- The verification key is (N, e) and the signing key is (N, d), where these values have the appropriate RSA relationship.
- A signature of message m (here m is an element of $\mathbb{Z}_N$) is the value $\sigma = m^d \,\%\, N$. Intuitively, only someone with the signing key can generate this value for a given m.
- To verify a signature σ on a message m, our goal is to check whether $\sigma \equiv_N m^d$. However, we are given only N and e, not d. Consider raising both sides of this equation to the e power:
$$\sigma^e \equiv_N (m^d)^e \equiv_N m$$
The second equality is from the standard RSA property. Now this check can be done given only the public information N and e.

A formal description of this scheme is given below:

Construction 13.7 (Textbook RSA)

The key generation algorithm is not listed here, but N, e, d are generated in the usual way for RSA. The signing key is $sk = (N, d)$ and the verification key is $vk = (N, e)$.

$\text{Sign}\big(sk = (N, d), m\big)$:	$\text{Ver}\big(vk = (N, e), m, \sigma\big)$:
return $m^d \,\%\, N$	$m' := \sigma^e \,\%\, N$ return $m \stackrel{?}{=} m'$

Unfortunately, textbook RSA signatures are useful only as a first intuition. They are **not secure**! A simple attack is the following:

Suppose an attacker knows the verification key (N, e) and sees a valid signature $\sigma \equiv_N m^d$ for some message m. Then σ^2 is also a valid signature for the message m^2, since:

$$\sigma^2 \equiv_n (m^d)^2 = (m^2)^d$$

The attacker can easily generate a forged signature on a new message m^2, making the scheme insecure.

Hashed RSA Signatures

The problem with textbook RSA signatures is that the signatures and plaintexts had a very strong algebraic relationship. Squaring the signature had the effect of squaring the underlying message. One way to fix the problem is to "break" this algebraic relationship. Hashed RSA signatures break the algebraic structure by applying the RSA function not to m directly, but to $H(m)$, where H is a suitable hash function (with outputs interpreted as elements of $\mathbb{Z}_N$).

Construction 13.8 (Textbook RSA)

$$\begin{array}{l} \underline{\text{Sign}\big(sk = (N,d), m\big):} \\ \text{return } H(m)^d \,\%\, N \end{array} \qquad \begin{array}{l} \underline{\text{Ver}\big(vk = (N,e), m, \sigma\big):} \\ y := \sigma^e \,\%\, N \\ \text{return } H(m) \stackrel{?}{=} y \end{array}$$

Let's see how this change thwarts the attack on textbook signatures. If σ is a valid signature of m, we have $\sigma \equiv_N H(m)^d$. Squaring both sides leads to $\sigma^2 \equiv_N (H(m)^2)^d$. Is this the valid signature of any m'? An attacker would have to identify some m' that has $H(m') = H(m)^2$. If the hash function is a good one, then this should be hard.

Of course, this is not a formal proof. It is possible to formally prove the security of hashed RSA signatures. The precise statement of security is: "if RSA is a secure trapdoor function and H is modeled as a random oracle, then hashed RSA signatures are a secure signature scheme." Since we have not given formal definitions for either trapdoor functions or random oracles, we won't see the proof in this book.

to-do *Write a chapter on random oracle and other idealized models.*

13.4 Chinese Remainder Theorem

今有物不知其數三三數之賸二五五數之賸三七七數之賸二問物幾何

When doing arithmetic mod N, we can sometimes use knowledge of the factors $N = pq$ to speed things up. This section discusses the math behind these speedups.

History. In the *Sunzi Suanjing*, written some time around the 4th century CE, Chinese mathematician Sunzi posed an interesting puzzle involving remainders:

> *"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?"*[6]

Sunzi's puzzle is the first known instance of a system of simultaneous equations involving modular arithmetic: In our notation, he is asking us to solve for x in the following system of congruences:

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

We can solve such systems of equations using what is called (in the West) the **Chinese Remainder Theorem** (CRT). Below is one of the simpler formations of the Chinese Remainder Theorem, involving only two equations/moduli (unlike the example above, which has three moduli 3, 5, and 7):

Theorem 13.9 (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 multiplicative 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. Using similar reasoning mod s, we can see that $x \equiv_s v$, so x is a solution to both equations.

Now we argue that this solution is *unique* modulo rs. Suppose x and x' are two solutions to the system of equations, so we have:

$$x \equiv_r x' \equiv_r u$$

[6]Chinese text is from an old manuscript of *Sunzi Suanjing*, but my inability to speak the language prevents me from identifying the manuscript more precisely. English translation is from Joseph Needham, *Science and Civilisation in China, vol. 3: Mathematics and Sciences of the Heavens and Earth*, 1959.

$$x \equiv_s x' \equiv_s v$$

Since $x \equiv_r x'$ and $x \equiv_s x'$, it must be that $x - x'$ is a multiple of r and a multiple of s. Since r and s are relatively prime, their least common multiple is rs, so $x - x'$ must be a multiple of rs. Hence, $x \equiv_{rs} x'$. So any two solutions to this system of equations are congruent mod rs. ■

Example *Sage implements the `crt` function to solve for x in these kinds of systems of equations. Suppose we want to solve for x:*

$$x \equiv_{427} 42$$
$$x \equiv_{529} 123$$

In Sage, the solution can be found as follows:

```
sage: crt( 42,123, 427,529 )
32921
```

We can check the solution:

```
sage: 32921 % 427
42
sage: 32921 % 529
123
```

CRT Encodings Preserve Structure

Let's call $(u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s$ the **CRT encoding** of $x \in \mathbb{Z}_{rs}$ if they satisfy the usual relationship:

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

We can convert any $x \in \mathbb{Z}_{rs}$ into its CRT encoding quite easily, via $x \mapsto (x \,\%\, r, x \,\%\, s)$. The Chinese Remainder Theorem says that any $(u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s$ is a valid CRT encoding of a unique $x \in \mathbb{Z}_{rs}$; and the proof of the theorem shows how to convert from the CRT encoding into the "usual $\mathbb{Z}_{rs}$ encoding."

The amazing thing about these CRT encodings is that they preserve all sorts of arithmetic structure.

Claim 13.10 *If (u, v) is the CRT encoding of x, and (u', v') is the CRT encoding of x', then $(u+u' \% r, v+v' \% s)$ is the CRT encoding of $x + x' \,\%\, rs$.*

Example *Taking $r = 3$ and $s = 5$, let's write down the CRT encodings of every element in $\mathbb{Z}_{15}$. In this table, every column contains x and its CRT encoding (u, v):*

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
u	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
v	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

Highlight the columns for $x = 3$ and $x' = 7$ and their sum $x + x' = 10$.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
u	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
v	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

Focusing on only the highlighted cells, the top row shows a true addition expression $3 + 7 \equiv_{15} 10$; the second row shows a true addition expression $0 + 1 \equiv_3 1$; the third row shows a true addition expression $3 + 2 \equiv_5 0$.

This pattern holds for any x and x', and I encourage you to **try it!**

As if that weren't amazing enough, the same thing holds for multiplication:

Claim 13.11 *If (u, v) is the CRT encoding of x, and (u', v') is the CRT encoding of x', then $(u \cdot u' \% r, v \cdot v' \% s)$ is the CRT encoding of $x \cdot x' \% rs$.*

Example *Let's return to the $r = 3$, $s = 5$ setting for CRT and highlight $x = 6$, $x' = 7$, and their product $x \cdot x' \equiv_{15} 12$.*

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
u	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
v	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

The top row shows a true multiplication expression $6 \cdot 7 \equiv_{15} 12$; the second row shows a true multiplication expression $0 \cdot 1 \equiv_3 0$; the third row shows a true multiplication expression $1 \cdot 2 \equiv_5 2$.

This pattern holds for any x and x', and I encourage you to **try it!**

The CRT suggests a different, perhaps more indirect, way to do things mod rs. Suppose x has CRT encoding (u, v) and x' has CRT encoding (u', v'), and we want to compute $x + y$ mod rs. One wild idea is to first *directly compute the CRT encoding of this answer*, and then convert that encoding to the normal integer representation in $\mathbb{Z}_{rs}$.

In this case, we know that the answer $x + x'$ has the CRT encoding $(u + u' \% r, v + v' \% s)$. But this is the same as $(x + x' \% r, x + x' \% s)$ — do you see why? So, to add $x + x'$ mod rs, we just need to add $x + x'$ mod r, and then add $x + x'$ mod s. This gives us the CRT encoding of the answer we want, and we can convert that CRT encoding back into a normal $\mathbb{Z}_{rs}$-integer.

The same idea works for multiplication as well, giving us the following:

CRT method for doing some operation[s] mod rs

1. *Do the operation[s] you want, but mod r instead of mod rs.*
2. *Do the operation[s] you want, but mod s instead of mod rs.*
3. *Those two results are the CRT encoding of the final answer, so convert them back to the normal representation.*

Example *Let's take the example $r = 3359$ and $s = 2953$, which are relatively prime (so the CRT applies). Suppose we want to compute $3141592 + 6535897 \,\%\, rs$. Doing it the usual way in Sage looks like this:*

```
sage: r = 3359
sage: s = 2953
sage: (3141592 + 6535897) % (r*s)
9677489
```

Doing it in the CRT way looks like this.

```
sage: u = (3141592 + 6535897) % r
sage: v = (3141592 + 6535897) % s
sage: crt( u,v, r,s )
9677489
```

Both methods give the same answer!

Application to RSA

You might be wondering what the point of all of this is.[7] The CRT method seems like a very indirect and wasteful way to compute anything. This impression might be true for simple operations like addition and single multiplications. However, the CRT method is *faster* for exponentiation mod N, which is the main operation in RSA!

Example *In Sage, we can do basic exponentiation mod n as follows:*

```
sage: def modexp(x,e,n): # x^e mod n
....:     return Mod(x,n)^e
```

If we are working over an RSA modulus and know its factorization $p \times q$, then we use the CRT method for exponentiation mod pq as follows. We simply do the exponentiation mod p and (separately) mod q, then use the `crt` function to convert back to $\mathbb{Z}_{pq}$.

```
sage: def crtmodexp(x,e,p,q): # x^e mod pq, using CRT speedup
....:     u = Mod(x,p)^e
....:     v = Mod(x,q)^e
....:     return crt(u.lift(),v.lift(),p,q)
```

We need to use `u.lift()` and `v.lift()` to convert u and v from `Mod`-objects into integers, because that is what `crt` expects.

We can use both methods to perform an exponentiation, and measure how long it takes with the `timeit` function. In this example, N is about 2000 bits long, and the difference in speed is noticeable:

[7] I'm talking about the CRT method for arithmetic mod rs, not life in general.

```
sage: p = random_prime(2^1000)
sage: q = random_prime(2^1000)
sage: N = p*q
sage: x = 12345678901234567
sage: e = randint(0,N) # random integer between 0 & N-1
sage: timeit('modexp(x,e,N)')
125 loops, best of 3: 5.34 ms per loop
sage: timeit('crtmodexp(x,e,p,q)')
125 loops, best of 3: 2.86 ms per loop
```

And just for good measure, we can check that both approaches give the same answer:

```
sage: modexp(x,e,N) == crtmodexp(x,e,p,q)
True
```

To understand why the CRT method is faster, it's important to know that the cost of standard modular exponentiation over a k-bit modulus is $O(k^3)$. For simplicity, let's pretend that exponentiation takes *exactly* k^3 steps. Suppose p and q are each k bits long, so that the RSA modulus N is $2k$ bits long. Hence, a standard exponentiation mod N takes $(2k)^3 = 8k^3$ steps.

With the CRT method, we do an exponentiation mod p and an exponentiation mod q. Each of these exponentiations takes k^3 steps, since p and q are only k bits long. Overall, we are only doing $2k^3$ steps in this approach, which is $4\times$ faster than the standard exponentiation mod N. In this simple analysis, we are not counting the cost of converting the CRT encoding back to the typical mod-N representation. But this cost is much smaller than the cost of an exponentiation (both in practice and asymptotically).

It's worth pointing out that this speedup can only be done for RSA *signing*, and not *verification*. In order to take advantage of the CRT method to speed up exponentiation mod N, it's necessary to know the prime factors p and q. Only the person who knows the signing key knows these factors.

13.5 The Hardness of Factoring N

As previously mentioned, the best known way to break the security of RSA as a trapdoor function (*i.e.*, to compute the inverse RSA function given only the public information N and e) involves factoring the RSA modulus.

Factoring integers (or, more specifically, factoring RSA moduli) is believed to be a hard problem for classical computers. 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:

Theorem 13.12 *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 the corresponding d (satisfying $ed \equiv_{\phi(N)} 1$).*

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 d exactly how it is computed in the key generation 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 13.12 concerns a new concept known as square roots of unity:

Definition 13.13 (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.*

Since $(\pm 1)^2 = 1$ *over the integers*, it is also true that $(\pm 1)^2 \equiv_N 1$. In other words, ± 1 are always square roots of unity modulo N, for any N. But some values of N have even more square roots of unity. If N is the product of distinct odd primes, then N has 4 square roots of unity: two trivial and two non-trivial ones (and you are asked to prove this fact in an exercise).

Claim 13.14 *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 13.12.)*

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:

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

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 $\{p,q\}$. In other words, $\{\gcd(pq, x-1), \gcd(pq, x+1)\} = \{p,q\}$. ■

Finding square roots of unity

Claim 13.15 *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 13.12.)*

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

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

Although we don't prove it here, it is possible to show that the algorithm 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 $1 - 2^{-n}$. ■

Exercises

13.1. Prove by induction the correctness of the EXTGCD algorithm. That is, whenever EXTGCD(x, y) outputs (d, a, b), 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 $x^a \% n = x^{a \% \phi(n)} \% n$ for any $x \in \mathbb{Z}_n^*$. In other words, when working modulo n, you can reduce exponents modulo $\phi(n)$.

13.5. How many fractions a/b **in lowest terms** are there, where $0 < a/b < 1$ and $b \leqslant n$? For $n = 5$ the answer is 9 since the relevant fractions are:

$$\frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}$$

Write a formula in terms of n. What is the answer for $n = 100$?

Hint: How many are there with denominator *exactly* equal to n (in terms of n)?

13.6. 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 kth 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 $k \leqslant \log_\phi N$ 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.[8]

13.7. Consider the following **symmetric-key** encryption scheme with plaintext space $\mathcal{M} = \{0,1\}^\lambda$. To encrypt a message m, we "pad" m into a prime number by appending a zero and then random non-zero bytes. We then mulitply by the secret key. To decrypt, we divide off the key and then strip away the "padding."

The idea is that decrypting a ciphertext without knowledge of the secret key requires factoring the product of two large primes, which is a hard problem.

[8] 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.

<u>KeyGen:</u>
  choose random λ-bit prime k
  return k

<u>$\text{Dec}(k, c)$:</u>
  $m' := c/k$
  while m' not a multiple of 10:
    $m' := \lfloor m'/10 \rfloor$
  return $m'/10$

<u>$\text{Enc}(k, m \in \{0,1\}^\lambda)$:</u>
  $m' := 10 \cdot m$
  while m' not prime:
    $d \leftarrow \{1, \ldots, 9\}$
    $m' := 10 \cdot m' + d$
  return $k \cdot m'$

Show an attack breaking CPA-security of the scheme. That is, describe a distinguisher and compute its bias.

Hint: Ask for any two ciphertexts.

13.8. Explain why the RSA exponents e and d must always be odd numbers.

13.9. Why must p and q be *distinct* primes? Why is it a bad idea to choose $p = q$?

13.10. A **simple power analysis (SPA)** attack is a physical attack on a computer, where the attacker monitors precisely how much electrical current the processor consumes while performing a cryptographic algorithm. In this exercise, we will consider an SPA attack against the ModExp algorithm shown in Section 13.2.

The ModExp algorithm consists mainly of squarings and multiplications. Suppose that by monitoring a computer it is easy to tell when the processor is running a squaring vs. a multiplication step (this is a very realistic assumption). This assumption is analogous to having access to the printed output of this modified algorithm:

<u>$\text{ModExp}(m, e, N)$:</u> *// compute m^e % N*
  if $e = 0$: return 1
  if e even:
    $res := \text{ModExp}(m, \frac{e}{2}, N)^2 \% N$
    print "square"
  if e odd:
    $res := \text{ModExp}(m, \frac{e-1}{2}, N)^2 \cdot m \% N$
    print "square"
    print "mult"
  return res

Describe how the printed output of this algorithm lets the attacker *completely* learn the value e. Remember that in RSA it is indeed the exponent that is secret, so this attack leads to key recovery for RSA.

Hint: Think about what "$e/2$," "$(e-1)/2$," and "e is odd" mean, in terms of the *bits* of e.

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

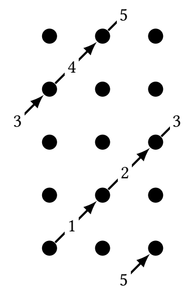
$$x \equiv_r u$$

$$x \equiv_s v$$

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.

13.12. Prove Claims 13.10 and 13.11.

13.13. Consider a rectangular grid of points, with width w and height h. Starting in the lower-left of the grid, start walking diagonally northeast. When you fall off end the grid, wrap around to the opposite side (*i.e.*, Pac-Man topology). Below is an example of the first few steps you take on a grid with $w = 3$ and $h = 5$:



Show that if $\gcd(w, h) = 1$ then you will eventually visit every point in the grid.

Hint: Derive a formula for the coordinates of the point you reach after n steps.

13.14. Suppose $(u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s$ is a CRT encoding of $x \in \mathbb{Z}_{rs}$. Prove that $x \in \mathbb{Z}^*_{rs}$ if and only if $u \in \mathbb{Z}^*_r$ and $v \in \mathbb{Z}^*_s$.

Note: this problem implies that $\phi(rs) = \phi(r)\phi(s)$ when $\gcd(r, s) = 1$. A special case of this identity is the familiar expression $\phi(pq) = (p-1)(q-1)$ when p and q are distinct primes.

13.15. There is a bug (or at least an oversight) in the proof that $x \mapsto x^e \% N$ and $y \mapsto y^d \% N$ are inverses. We used the fact that $x^{\phi(N)} \equiv_N 1$, but this is only necessarily true for $x \in \mathbb{Z}^*_N$. Using the Chinese Remainder Theorem, show that the RSA function and its inverse are truly inverses, even when applied to $x \notin \mathbb{Z}^*_N$.

13.16. We are supposed to choose RSA exponents e and d such that $ed \equiv_{\phi(N)} 1$. Let $N = pq$ and define the value $L = \mathsf{lcm}(p-1, q-1)$. Suppose we choose e and d such that $ed \equiv_L 1$. Show that RSA still works for this choice of e and d — in other words, $x \mapsto x^e \% N$ and $y \mapsto y^d \% N$ are inverses.

Hint: You'll have to use the Chinese Remainder Theorem.

★ 13.17. If $y^e \equiv_N x$ then we call y an "e-th root" of x. One way to think about RSA is that raising something to the d power is equivalent to computing an e-th root. Our assumption about RSA is that it's hard to compute e-th roots given only public e and N.

In this problem, show that if you are given an a-th root of x and b-th root of the same x, and $\gcd(a, b) = 1$, then you can easily compute an ab-th root of x.

More formally, given x, y, z and N where $y^a \equiv_N x$ and $z^b \equiv_N x$, show how to efficiently compute a value w such that $w^{ab} \equiv_N x$.

Compute w for the following values (after verifying that y is an a-th root and z is a b-th root of x mod N):

```
N = 318753895014839414391833197387495582828703628009180678460009
x = 183418622076108277295248802695684859123490073011079896375192
a = 56685394747281296805145649774065693442016512301628946051059
b = 178205100585526989632998577959780764157496762062661723119813
y = 185557583864994472527185541352084631165296327724386727334885
z = 20697550065842164169278024507041536884260713996371572807344
```

Hint: It is important that $\gcd(a, b) = 1$. Use Bezout's theorem.

13.18. Suppose Alice uses the CRT method to sign some message m in textbook RSA. In other words, she computes $m^d \,\%\, p$, then $m^d \,\%\, q$, and finally converts this CRT encoding back to $\mathbb{Z}_N$. But suppose Alice is using faulty hardware (or Eve is bombarding her hardware with electromagnetic pulses), so that she computes the **wrong value** mod q. The rest of the computation happens correctly, and Alice publishes m and the (incorrect) signature σ.

Show that, no matter what m is, and no matter what Alice's computational error was, Eve can factor N (upon seeing m, σ, and the public RSA information N and e).

Hint: $m \equiv_p \sigma^e$ but $m \not\equiv_q \sigma^e$.

13.19. (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 Sage function that takes as input an RSA modulus N and $\phi(N)$ and outputs the prime factors of 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 = 1331402728893351929221084092606621744763038316523836716885470094 84
      2532359405869171404826691822563682852609928294472079801831701748 67
```

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

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

13.21. Discuss the computational difficulty of the following problem:

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

If you can, relate its difficulty to that of other problems we've discussed (factoring N or inverting RSA).

13.22. (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 Sage 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

$$105295498644227198587577819266 3 \times 611174539744122090068393470777.$$

★ 13.23. 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.

13.24. 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.

13.25. Why are ± 1 the only square roots of unity modulo p, when p is an odd prime?

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

13.27. Implement a Sage 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.

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

```
    6010670771037423260302596293224586203119494535840455383059452175640
    0274610132250099809986731601449677193744267641167218611384967800008
    6366258360757218165973
d = 1384769997342637754981004435671327591821445734744740141950210912 72
```

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

(a) Let $N = pq$ be an RSA modulus. Show that if you know N and $\delta = |p - q|$ then you can efficiently factor N.

(b) Alice generated the following RSA modulus $N = pq$ and lets you know that $|p - q| < 10000$. Factor N:

```
N = 874677518388996663638698301429866315858010681593301504361505917406
    679600338654753978646639928231278257025792316921962329748948203153
    633013718175380969169006125249183547099230845322374618855425387176
    952865483432804575895177869626746459878695728149786382697571962961
    898331255405534657194681056148437649091612403258304084081171824215
    469594984981192162710052121535309254024720635781955739713239334398
    494465828323810812843582187587256744901184016546638718414715249093
    757039375585896257839327987501216755865353444704506441078034811012
    930282857089819030160822729139768982546143104625315700571887037795
    31855302859423676881
```

13.29. Here is a slightly better method to factor RSA moduli whose factors are too close together. As before, let $N = pq$.

(a) Define $t = (p + q)/2$. Note that when p and q are close, t is not much larger than $\sqrt{N}$. Show that:

- $t^2 - N$ is a perfect square.
- Given t, it is possible to efficiently factor N.

Hint: Write $t^2 - N = s^2$ for some s.

(b) Write a Sage function that factors RSA moduli whose prime factors are close. Use it to factor the following 2048-bit number. How close were the factors (how large was $|p - q|$)?

Hint: Sage has an `is_square` method. Also, be sure to do exact square roots over the integers, not the reals.

```
N = 514202868664266501986736340226343880193216864011643244558701956114
    553317880043289827487456460284103951463512024249329243228109624011
```

```
915392411888724026403127686707255825056081890692595715828380690811
131686383180282330775572385822102181209569411961125753242467971879
131305986986525600110340790595987975345573842266766492356686762134
653833064511337433089249621257629107825681429573934949101301135200
918606211394413498735486599678541369375887840013842439026159037108
043724221865116794034194812236381299786395457277559879575752254116
612726596118528071785474551058540599198869986780286733916614335663
37230032465696303733233
```