Euler's totient and RSA

Application of discrete models

1 Euler's totient function

Earlier (see Congruences) we have seen that $a \in \mathbb{Z}$ is invertible modulo m if and only if a and m are coprime, i.e. $\gcd(a, m) = 1$. For example, modulo 10, the invertible remainders are 1, 3, 7 and 9 (because everything else is divisible by either 2 or 5).

An important property about the invertible numbers is that their product is also invertible:

Theorem 1.1. *If a and b are invertible modulo m, then ab is also invertible modulo m.*

Example. For $m = 10$, $3 \cdot 9 \equiv 7\ (10)$, $3 \cdot 3 \equiv 9\ (10)$, $3 \cdot 7 \equiv 1\ (10)$ etc.

We have seen a special case of this in the previous document (Discrete logarithm), where the modulus was a prime number, because then all nonzero remainders are invertible, and we have seen that the product of two nonzero remainders are also nonzero. But we have also seen that it is *not* true if the modulus is not prime (e.g. $4 \cdot 5 \equiv 0\ (10)$), and now we can see that it is because the proper way to generalize this is to change "nonzero" to "invertible".

We have also introduced the notion of *complete residue systems*, e.g. $\{0, 1, 2, \ldots, m-1\}$ (see Congruences). Now let's see a similar notion, but only with invertible elements:

Definition 1.2. If S is a complete residue system (modulo m), then $\{a \in S \mid \gcd(a, m) = 1\}$ is a **reduced residue system** (modulo m).

Example. All of the following sets are reduced residue systems modulo 10:
$\{1, 3, 7, 9\}$, $\{11, 13, 17, 19\}$, $\{1, 53, 77, -21\}$ etc.

Similarly to complete residue systems, there are infinitely many reduced residue systems for any m (but usually we are only interested in the one that contains the remainders, e.g. $\{1, 3, 7, 9\}$). Also, for any given m, they have the same number of elements, e.g. for $m = 10$, they have always 4 elements.

But how many elements do these sets have in general, i.e. how many invertible remainders are there for a given m? This is measured by the following function:

Definition 1.3. For $n \in \mathbb{N}^+$, let $\boldsymbol{\varphi(n)}$ be the number of integers between 0 and n–1 that are coprime to n, i.e. $\varphi(n) := |\{k \in \mathbb{Z} \mid 0 \leq k \leq n-1 \ \wedge\ \gcd(n, k) = 1\}|$. It is called **Euler's totient function** (or *Euler's φ function* – spelled "phi").

Example. $\varphi(10) = |\{1, 3, 7, 9\}| = 4$ and $\varphi(7) = |\{1, 2, 3, 4, 5, 6\}| = 6$.

In Sage, $\varphi(n)$ is called `euler_phi(n)`. Let's print its first few values:

```
sage: matrix([[n, euler_phi(n)] for n in range(1, 26)]).transpose()
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25]
[ 1  1  2  2  4  2  6  4  6  4 10  4 12  6  8  8 16  6 18  8 12 10 22  8 20]
```

We can also plot them:

```
sage: plot(euler_phi, 1, 100)
```

The most important properties of $\varphi(n)$:

1. $\varphi(1) = 1$ (since $\gcd(1, 0) = 1$).
2. If $n \geq 2$, then $\varphi(n) \leq n-1$ (since $\gcd(n, 0) = n \neq 1$).
3. If (and only if) p is prime, then $\varphi(p) = p-1$, because all numbers between 1 and $p-1$ are coprime to p. (So the highest possible values of $\varphi(n)$ are for prime numbers.)
4. If p is prime, then $\varphi(p^k) = p^k - p^{k-1} = (p-1)p^{k-1}$, e.g. $\varphi(16) = \varphi(2^4) = 2^4 - 2^3 = 8$. (Proof: from the total p^k numbers, we need to remove the multiples of p, whose number is $p^k/p = p^{k-1}$.)
5. If a and b are coprime, then $\varphi(ab) = \varphi(a) \cdot \varphi(b)$. For example, $\varphi(10) = \varphi(2) \cdot \varphi(5) = 1 \cdot 4 = 4$. (The proof is complicated, so it is omitted.) Important: this is only true if a and b are coprime! Otherwise for example $\varphi(2) \cdot \varphi(4) = 1 \cdot 2 = 2$, but $\varphi(2 \cdot 4) = \varphi(8) = 2^3 - 2^2 = 4 \neq 2$.

How can we calculate $\varphi(n)$ in general? If we know the prime factorization of n, then the last two properties give a simple method. For example, if $n = 24 = 2^3 \cdot 3$, then, since 2^3 and 3 are coprime:

$$\varphi(24) = \varphi(2^3) \cdot \varphi(3) = (2^3 - 2^2) \cdot (3 - 1) = 4 \cdot 2 = 8.$$

Since powers of different prime numbers are always coprime, we can use the prime factorization of n to split $\varphi(n)$ into $\varphi()$'s of prime powers, and then use the formula for $\varphi(p^k)$. Another example:

$$\varphi(4200) = \varphi(2^3 \cdot 3 \cdot 5^2 \cdot 7) = \varphi(2^3) \cdot \varphi(3) \cdot \varphi(5^2) \cdot \varphi(7) = (2^3 - 2^2) \cdot 2 \cdot (5^2 - 5) \cdot 6 = 960.$$

The general formula is the following:

Theorem 1.4 (calculation of φ)**.** *If the canonical representation of n is $n = p_1^{k_1} p_2^{k_2} \cdots p_m^{k_m}$, then:*

$$\varphi(n) = \left(p_1^{k_1} - p_1^{k_1-1}\right)\left(p_2^{k_2} - p_2^{k_2-1}\right) \cdots \left(p_m^{k_m} - p_m^{k_m-1}\right).$$

As a special case, if n is a product of two prime numbers, $n = pq$, then $\varphi(n) = (p-1)(q-1)$.

Note: we can see that it is very easy to calculate $\varphi(n)$ if we know the prime factorization of n. But if we don't know that – and as we will see, factoring a large n is very difficult –, then we cannot calculate $\varphi(n)$ efficiently either. This will be of crucial importance later.

2 Modular exponentiation II

In the previous document (Discrete logarithm), we have discussed modular exponentiation, but mainly for prime modulus. In this section, we generalize it for arbitrary modulus.

For example, let's calculate the first few powers of all remainders modulo 10:

```
sage: matrix([[power_mod(a, k, 10) for k in range(30)] for a in range(10)])
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 2 4 8 6 2 4 8 6 2 4 8 6 2 4 8 6 2 4 8 6 2 4 8 6 2 4 8 6 2]
[1 3 9 7 1 3 9 7 1 3 9 7 1 3 9 7 1 3 9 7 1 3 9 7 1 3 9 7 1 3]
[1 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4]
[1 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5]
[1 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6]
[1 7 9 3 1 7 9 3 1 7 9 3 1 7 9 3 1 7 9 3 1 7 9 3 1 7 9 3 1 7]
[1 8 4 2 6 8 4 2 6 8 4 2 6 8 4 2 6 8 4 2 6 8 4 2 6 8 4 2 6 8]
[1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9]
```

Comparing this with the case of prime modulus from eariler, we can observe the following:

- Here 1 doesn't always reappear in all rows (after $a^0 = 1$). For some rows, it appears periodically, but for others, $a^0 = 1$ is the only 1.
- Eventually, each row repeats periodically, but the first reappearing power is not always 1.
- For which bases does 1 repeat? For 1, 3, 7 and 9, i.e. exactly for the invertible numbers.
- The *order* of these numbers (i.e. the first positive exponent for which the power is 1) can be 1, 2 or 4 (e.g. $\mathrm{ord}_{10}(7) = 4$). Notice that they are all divisors of 4.
- Just like for prime modulus, it is also true here that $\mathrm{ord}_{10}(1) = 1$ and $\mathrm{ord}_{10}(10-1) = 2$.

In order to generalize most of the above statements, we need a general theorem about modular exponentiation, similar to Fermat's little theorem, which said for a prime modulus that $a^{p-1} \equiv 1 \pmod{p}$ if $a \not\equiv 0 \pmod{p}$. The general version for arbitirary modulus is the following:

Theorem 2.1 (Euler's totient theorem, a.k.a. Fermat–Euler theorem, or simply Euler's theorem)**.** *If $m \in \mathbb{N}^+$, $a \in \mathbb{Z}$ and $\gcd(a, m) = 1$, then $a^{\varphi(m)} \equiv 1 \pmod{m}$.*

Example. For $m = 10$, $\varphi(10) = 4$, so $3^4 \equiv 1 \pmod{10}$, $7^4 \equiv 1 \pmod{10}$ etc.

So in general, it is not the $p-1$'st power that is 1, but the $\varphi(m)$'th power (and for prime numbers, these two values are the same: $\varphi(p) = p-1$).

Very important: this theorem works only if a and m are coprime. If they are not, then not only the $\varphi(m)$'th power will not be 1, but no positive power at all. For example, for $m = 10$, all powers of even numbers are even, and all powers of 5 are 5. This is true in general: it is easy to see that any $a^n \bmod m$ is divisible by $\gcd(a, m)$. So if $\gcd(a, m) \neq 1$, i.e. they are not coprime, then 1 cannot appear among the powers. But if they *are* coprime, then Euler's theorem guarantees that 1 appears.

Another consequence of the theorem is that the order of any of these numbers ($\mathrm{ord}_m(a)$) divides $\varphi(m)$. For example, for $m = 10$, the order is either 1, 2 or 4 (see above), and indeed, they all divide $\varphi(10) = 4$. (This is a generalization of Lagrange's theorem, which was stated only for prime modulus.)

In certain cases, Euler's theorem can be used to simplify modular exponentiation. Consider for example $137^{75} \bmod 10$. First, of course the base can be taken modulo 10, i.e. $137 \equiv 7 \pmod{10}$, therefore $137^{75} \equiv 7^{75} \ (10)$. Now for the exponent, the same thing doesn't work: $7^{75} \not\equiv 7^5 \ (10)$. Instead, we can use Euler's theorem, which states that $7^{\varphi(10)} \equiv 1 \ (10)$, so we can take the remainder of 75, not modulo 10, but modulo $\varphi(10) = 4$, i.e. $7^{75} \equiv 7^{18 \cdot 4 + 3} \equiv \left(7^4\right)^{18} 7^3 \equiv 1^{18} \cdot 7^3 \equiv 7^3 \ (10)$. In this way, we simplified 137^{75} to 7^3, which is much easier to calculate (e.g. by using fast exponentiation on these smaller numbers, see the previous document).

But this simplification doesn't always work: first, Euler's theorem requires that the base must be coprime to the modulus (which was true here: $\gcd(7, 10) = 1$), and second, we need to calculate $\varphi(m)$, i.e. we need the prime factorization of m, which is difficult to calculate for very big m.

Summary: if we can calculate $\varphi(m)$ for the modulus m, and $\gcd(a, m) = 1$, then the modular power $a^n \bmod m$ can be calculated by taking the base modulo m, and the exponent modulo $\varphi(m)$: $a^n \equiv (a \bmod m)^{n \bmod \varphi(m)} \pmod{m}$, and then performing fast exponentiation from these values.

But what if the base is not coprime to the modulus? Consider for example $138^{75} \bmod 10$. The first step still works, i.e. $138^{75} \equiv 8^{75} \ (10)$. But we can't use Euler's theorem directly, because $\gcd(8, 10) \neq 1$. The idea is to split the modulus into two parts: $10 = 5 \cdot 2$, and split the problem along with it: instead of $8^{75} \bmod 10$, calculate $8^{75} \bmod 5$ and $8^{75} \bmod 2$. Then the first part can be solved using Euler's theorem, because $\gcd(8, 5) = 1$, so $8^{75} \equiv 3^{75} \equiv 3^{75 \bmod 4} \equiv 3^3 \equiv 2 \ (5)$, and the second part is trivial: $8^{75} \equiv 0^{75} \equiv 0 \ (2)$. These two results ($138^{75} \equiv 2 \ (5)$ and $138^{75} \equiv 0 \ (2)$) can be combined into the final result (i.e. modulo 10) using the Chinese remainder theorem (see Congruences).

3 The RSA algorithm

RSA is a commonly used cryptosystem, i.e. a method to encrypt and decrypt messages. Its name comes from the initials of its creators (Rivest, Shamir and Adleman). It is used in various places such as internet communication (e.g. HTTPS, SSH) and bank security.

3.1 Asymmetric encryption

RSA is an *asymmetric* encryption algorithm (also known as *public-key* encryption), which means that it uses two different keys: one for encryption, and one for decryption (i.e. to decipher the encrypted message). This is in contrast to *symmetric-key* encryptions (such as the commonly used AES, which will be discussed in a later document), which use the same key for both encryption and decryption. Symmetric-key encryptions are usually simpler and much faster than the asymmetric ones, but their drawback is that both parties must know the same secret key. So they either have to meet in private to agree on this shared key, or use an algorithm such as the Diffie–Hellman key exchange to create one (see the previous document: Discrete logarithm).

Asymmetric encryptions however, such as RSA, have two different, but related keys:

- **Public key**: it can be shared publicly, and it is used to encrypt messages.
- **Private key**: it must be kept secret by its owner, and it is used to decrypt the messages that were encrypted by the corresponding public key.

This means that the two parties, say Alice and Bob, don't have to agree on a secret key, instead, Alice publishes her public key, and Bob uses it to encrypt a message for her, which can only be decrypted by Alice's private key. Therefore, anyone can send encrypted messages to Alice, but only she can decrypt them.

In order for an asymmetric encryption to work, the two keys must be connected, because one must decrypt what the other has encrypted; but also, they must be different enough so that one cannot be calculated from the other (or at least the private key from the public key), otherwise the whole system would be pointless. These two requirements seemingly contradict each other. Also, if one key cannot be calculated from the other, then how are they created in the first place?

3.2 How RSA works

The RSA algorithm is based on the fact that prime factorization is a very difficult task for large numbers. There is no known algorithm that can efficiently calculate the prime factors of a sufficiently large integer. So if we have two big prime numbers, p and q, we can easily multiply them: $n := p \cdot q$, but noone will be able to calculate p and q from n. (At least if the they are big enough, and they satisfy certain other mathematical requirements, which are not detailed here.) Note: this is similar to the discrete logarithm (see the previous document) in that it also had an operation, the modular exponentiation, which is easy to perform, but its inverse, the discrete logarithm is hard. Such an operation is called a *one-way function*.

The main operation of RSA, just like for the Diffie–Hellman key exchange, is modular exponentiation, but in this case, the modulus is not a prime number, but the product of two primes: $n = pq$. Both the encryption and the decryption are exponentiations modulo n, but to different exponents:

for a message m (the *plaintext*), the encryption calculates $c \equiv m^e \pmod{n}$, where c is the encrypted message (the *ciphertext*) and e is the *public exponent*; then, the decryption restores m by calculating $m \equiv c^d \pmod{n}$, where d is the *private exponent*. The question is of course how to create e and d so that the above calculation works.

In order for the decryption to restore the same m that was encrypted, we need

$$m \equiv c^d \equiv (m^e)^d \equiv m^{ed} \pmod{n}.$$

We know from Euler's totient theorem (see above) that the exponent can be taken modulo $\varphi(n)$, i.e. $m^{ed} \equiv m^{ed \bmod \varphi(n)} \pmod{n}$. So, if e and d satisfy $ed \equiv 1 \pmod{\varphi(n)}$ – i.e. d is the inverse of e modulo $\varphi(n)$ –, then we get $m^{ed} \equiv m \pmod{n}$ as desired. (Note: if $\gcd(m, n) \neq 1$, then we can't use Euler's theorem, but it can still be proven in other ways that $m^{ed} \equiv m \pmod{n}$.)

The private exponent (d) can only be calculated from the public exponent (e) if we know $\varphi(n)$. But in order to calculate $\varphi(n)$, we need the prime factorization of n, i.e. p and q. So the security of RSA is indeed based on the fact that prime factorization is difficult.

The first step of RSA is **key generation**:

1. Generate two different random secret prime numbers: p and q.
2. Calculate their product: $n := pq$.
3. Calculate $\varphi(n)$: $\varphi(n) = \varphi(pq) = \varphi(p)\varphi(q) = (p-1)(q-1)$.
4. Choose a public exponent e for which $2 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
5. Solve the congruence $ed \equiv 1 \pmod{\varphi(n)}$ for d, i.e. calculate the inverse of e modulo $\varphi(n)$.
6. The public key is the pair (n, e), which can be published.
7. The private key is the pair (n, d), which must be kept secret.

(Note: not only d, but also p, q and $\varphi(n)$ are secret values, though they are not needed anymore, so they can be deleted.)

Then, RSA performs **encryption** and **decryption** as follows:

1. Write the plaintext message as an integer m between 0 and $n-1$. (If the message is too long, then split it up into smaller chunks, and encrypt them separately.)
2. Encryption: calculate $c \equiv m^e \pmod{n}$ using the public key (n, e).
3. Decryption: calculate $m \equiv c^d \pmod{n}$ using the private key (n, d).

Just like for the Diffie–Hellman key exchange, it is very important that the two prime numbers p and q must be generated by a cryptographically secure pseudorandom number generator (see the previous document), so that an outside listener can't possibly predict them. It is also important that they must be large enough (and appropriately chosen), so that n cannot be factored into p and q. Nowadays, it is common to use 2048-bit or 4096-bit RSA, which means that the binary representation of n has 2048 or 4096 bits (so p and q each have ca. 1024 or 2048 bits, respectively).

And how do we generate e, the public exponent? Since it is public, it doesn't need to be chosen randomly, but it can be chosen to make the calculation more efficient. In practice, it is common to use the value $e = 2^{16} + 1 = 65537$, because it is simple enough to make modular exponentiation efficient (remember that fast exponentiation depends on the length and the number of 1 bits in the binary representation), but it is also complex enough that it doesn't risk the security of RSA.

3.3 Digital signature

As we have seen, cryptography with RSA works by encrypting the message using the public key, and decrypting it using the private key. In this way, anyone can send an encrypted message to a recipient, but only that person, i.e. the owner of the private key can decrypt it.

But what happens if we reverse that? What happens if we encrypt a message using the private key, and decrypt it using the public key? Then only the owner of the private key can encrypt messages, but anyone can decrypt them. Does it make any sense?

This is called *digital signature*, which is another important application of RSA besides encryption. In this case, the message m is "encrypted" using the private key, i.e. the value $c \equiv m^d \pmod{n}$ is calculated, and this c is attached to the message m. Then the recipient calculates $m \equiv c^e \pmod{n}$ from c, and if it matches the received m, then the signature verification is successful, i.e. the recipient can be certain that the message was indeed written by a person in possession of the private key, who is (hopefully) only the sender.

3.4 Improvement using CRT

We have seen that the public exponent, e can be chosen to make encryption as efficient as possible. But the private exponent, d cannot be influenced directly, so it can be very big – which is of course desirable from the security point of view, but it means that decryption, i.e. raising to the power d is much slower than encryption, i.e. raising to e.

One way to improve the speed of decryption is as follows. Instead of calculating $m \equiv c^d \ (n)$ directly, we do it in two parts: since $n = pq$, we can calculate this exponentiation separately for p and q, i.e. $m_p \equiv c^d \ (p)$ and $m_q \equiv c^d \ (q)$, and then combine them using the Chinese remainder theorem (see Congruences). The point of this is that not only can we calculate with smaller numbers (because of the smaller modulus), but we can also reduce the exponents using Fermat's little theorem (see Discrete logarithm): we only need to raise to the powers $d_p := d \bmod (p-1)$ and $d_q := d \bmod (q-1)$, respectively. So although we perform two modular exponentiations instead of one, each one is more than four times faster than the original one: the exponents are half as long, so we only need half as many multiplications, and the multiplied numbers are also half as long.

So the improved version calculates $m \equiv c^d \pmod{n}$ as follows:

1. Calculate the following values beforehand:
   - $d_p := d \bmod (p-1)$,
   - $d_q := d \bmod (q-1)$,
   - the inverse of q (i.e. q^{-1}) modulo p.
2. $m_p := c^{d_p} \bmod p$.
3. $m_q := c^{d_q} \bmod q$.
4. Calculate m from m_p and m_q using the Chinese remainder theorem (CRT):
   - $m_d := q^{-1}(m_p - m_q) \bmod p$.
   - $m := m_q + m_d q$.

Because of this, the private key is often not only (n, d), but the tuple (p, q, d_p, d_q, q^{-1}).

4 Primality tests

There is one more important aspect of RSA that needs to be addressed: how do we generate the two prime numbers p and q? We can use a cryptographically secure random number generator (see the previous document) but how do we check whether the result is a prime number? The whole point of RSA is that we can't factor large numbers – but if we can't find their divisors, then how do we check whether a big random integer is a prime number or not?

Fortunately, deciding whether a number is prime is much easier than factoring it. A method that does the former is called a *primality test*. There are two main kinds of primality tests:

1. A *deterministic primality test* can always tell correctly whether a number is prime.
2. A *probabilistic primality test* can't tell this for 100%, sometimes it makes mistakes. The commonly used tests only make mistakes in one direction: they might classify composite numbers as prime numbers, but they always give correct answer for prime numbers.

But why would we use a test that sometimes gives wrong answers? Because probabilistic primality tests are usually much faster than the deterministic ones. For example, trial divison, which checks all numbers between 2 and $\sqrt{n}$ whether they divide n (see Divisors, prime numbers), is a deterministic primality test, but it is very slow for large numbers (as it is essentially a prime factorization method as well). However, the methods described below are much more efficient, and they make mistakes rarely enough so that they are still acceptable.

The Sage function `is_prime()` uses a deterministic primality test, so it is always correct – by default. But we can tell it to use a probabilistic test instead, to make it faster, but maybe incorrect:

```
sage: n = 2^2048 + 981
sage: is_prime(n)    # deterministic: SLOW!
True
sage: proof.arithmetic(False)
sage: is_prime(n)    # probabilistic: fast, but unreliable
True
```

4.1 Fermat primality test

Fermat's little theorem is only about prime numbers: if p is prime and $a \not\equiv 0 \pmod{p}$, then $a^{p-1} \equiv 1 \pmod{p}$. But can it be true if p is not a prime number?

Let's print the values of $a^{n-1} \bmod n$ for various n's and a's (where $1 \leq a \leq n-1$):

```
sage: for n in range(9, 16):
....:     print(n, [power_mod(a, n-1, n) for a in range(1, n)])
9 [1, 4, 0, 7, 7, 0, 4, 1]
10 [1, 2, 3, 4, 5, 6, 7, 8, 9]
11 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
12 [1, 8, 3, 4, 5, 0, 7, 8, 9, 4, 11]
13 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
14 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
15 [1, 4, 9, 1, 10, 6, 4, 4, 6, 10, 1, 9, 4, 1]
```

Notice the following:

1. If n is a prime number, then all values are 1, as guaranteed by Fermat's little theorem.
2. If $a = 1$ (first values), then of course its powers are always 1.
3. If $a = n-1$ (last values), then we still often get 1, because $n-1 \equiv -1 \pmod{n}$, and every second power of -1 is 1.
4. In all other cases however, i.e. if n is composite and $2 \leq a \leq n-2$, we almost never get 1.
5. But sometimes we do: e.g. for $n = 15$ and $a = 4$, we get $4^{14} \equiv 1 \pmod{15}$.

These observations can be turned into a probabilistic primality test: calculate $a^{n-1} \bmod n$ for some a, and if the result is different from 1, then n is definitely not a prime number; and if it is 1, then it is *probably* prime. This is called the **Fermat primality test**, and it works as follows:

Fermat primality test(n):

> **Input:** $n \in \mathbb{N}, n \geq 4$, the number to be tested
> Choose a random base $a \in \{2, 3, \ldots, n-2\}$.
> $x := a^{n-1} \bmod n$
> **if** $x \neq 1$ **then** $\rightarrow$ *false* (= "not prime")
> **if** $x = 1$ **then** $\rightarrow$ *true* (= "probably prime")

This is a very efficient algorithm, because the modular exponentiation can be calculated quickly using fast exponentiation. (Compare this to trial division, which requires $\sim \sqrt{n}$ divisions, while the Fermat primality test needs only $\leq 2 \log_2 n$ multiplications (see the fast exponentiation). E.g. if $n \approx 1000000$, then the former is around 1000, while the latter is only 40.)

The error probability can be reduced by repeating the test multiple times (for different random a's), and only accepting n as a prime number if it passes all tests (i.e. they all return true). For example, if $n = 15$ were tested only for $a = 4$, then n would pass the test as a "prime number", whereas if we repeat the test, e.g. for $a = 2$, then n is revealed to be a composite number.

Definition 4.1. For an $a \in \mathbb{N}^+$, a composite number n is called a **Fermat pseudoprime** (to base a) if $a^{n-1} \equiv 1 \pmod{n}$. (It is sometimes simply called a **pseudoprime**.)

Example. $n = 15$ is a Fermat pseudoprime to base $a = 4$, because $4^{14} \equiv 1 \pmod{15}$.

So Fermat pseudoprimes are those numbers which "fool" the Fermat primality test.

Note that this notion depends on the base a, i.e. the same number can be a pseudoprime for some a, but not for another a. To emphasize this, we have the following terms:

Definition 4.2. For a composite number n, the base $a \in \mathbb{N}^+$ is called a **Fermat liar** for n if n is a Fermat pseudoprime to base a, otherwise it is called a **Fermat witness** for n.

Example. $a = 4$ is a Fermat liar for $n = 15$, while $a = 2$ is a Fermat witness: $2^{14} \equiv 4 \not\equiv 1 \ (15)$.

Fortunately, pseudoprimes are rare. For example, between 1 and 10000, there are only 22 Fermat pseudoprimes to base $a = 2$ (the smallest being 341), whereas there are 1229 prime numbers in the same range.

But there are certain pseudoprimes which really resist the Fermat primality test:

Definition 4.3. A composite number n is called a **Carmichael number** if for every base $a \in \mathbb{N}$ that is coprime to n, $a^{n-1} \equiv 1 \pmod{n}$.

That is, Carmichael numbers are those numbers which are pseudoprimes to the most possible bases. (It cannot be true for more a's than this, because if a and n are not coprime, then the power cannot possibly be 1, as we have seen earlier in this document.) The smallest Carmichael number is 561. Fortunately, these numbers are even rarer than regular pseudoprimes (there are only 7 Carmichael numbers between 1 and 10000).

The good news is that unless we hit a Carmichael number, it is quite easy to catch a pseudoprime:

Theorem 4.4. *If n is a composite number but not a Carmichael number, then for at least half of the a's between 1 and $n-1$, we have $a^{n-1} \not\equiv 1 \pmod{n}$.*

Which means that there are three cases with the Fermat primality test:

1. If n is a prime number, then the test is always correct.
2. If n is a Carmichael number (which is rare), then the test is very unreliable.
3. Otherwise, the test is correct in at least 50% of the cases. This may seem low at first glance, but this is only a conservative lower bound, and we can increase the accuracy by repeating the test for different a's: when repeting k times, the error probability is $< 1/2^k$, e.g. for $k = 10$, the result is incorrect in less than 0.001 (= 0.1%) of the cases.

4.2 Miller–Rabin primality test

For real numbers, we know that the equation $x^2 = 1$ has only two solutions: $x = 1$ and $x = -1$. The following theorem states that in certain cases, this is also true for modular arithmetic:

Theorem 4.5. *If p is a prime number and $x^2 \equiv 1 \pmod{p}$, then $x \equiv 1 \pmod{p}$ or $x \equiv -1 \pmod{p}$.*

Example. If the modulus is $p = 5$, then the squares of $0, 1, 2, 3, 4$ are, respectively, $0, 1, 4, 4, 1$, i.e. the only solutions of $x^2 \equiv 1\ (5)$ are indeed $x \equiv 1\ (5)$ and $x \equiv 4 \equiv -1\ (5)$. But this is not necessarily true for composite numbers, e.g. $3^2 \equiv 1\ (8)$.

(Proof: the first congruence implies that $p \mid x^2 - 1 = (x-1)(x+1)$, but a prime number can only divide a product if it divides one of its factors, i.e. either $p \mid (x-1)$ or $p \mid (x+1)$.)

Going back to Fermat's little theorem, if p is a prime number (and $a \not\equiv 0\ (p)$), then $a^{p-1} \equiv 1\ (p)$. For example, if $p = 29$, then $a^{28} \equiv 1\ (29)$. Now apply the above theorem for $x = a^{14}$ (so $x^2 = a^{28}$), i.e. halve the exponent, and get that either $a^{14} \equiv 1\ (29)$ or $a^{14} \equiv -1\ (29)$. If $a^{14} \equiv 1\ (29)$, then we can halve the exponent again, and get that either $a^7 \equiv 1\ (29)$ or $a^7 \equiv -1\ (29)$. But now the exponent is odd, so we can't halve it further.

In each step of the above process, we used that p is a prime number. If it is not, than one of the steps may very easily fail (but not necessarily). This gives us another probabilistic primality test. For example, if $n = 15$ and $a = 4$, then $a^{n-1} \equiv 1\ (n)$, i.e. $4^{14} \equiv 1\ (15)$ is true (as we have seen at the Fermat primality test), but if we halve the exponent, then $4^7 \equiv 4\ (15)$, which is neither 1 nor -1; so this reveals that 15 cannot be a prime number.

The **Miller–Rabin primality test** is based on the above logic, but it performs the steps in reverse: instead of first calculating a^{n-1} and then halving the exponent, it first finds the last exponent that cannot be halved further (e.g. 7 for the above $p = 29$), calculates the power with that (e.g. a^7), and then doubles the exponent back until it reaches $n-1$ (e.g. $a^7 \to a^{14} \to a^{28}$). The point of this is that the powers with smaller exponents are easier to calculate, and in many cases, the test can stop sooner than $n-1$ if the result is already clear (e.g. if the above "halving theorem" is violated).

Miller–Rabin primality test(n):

Input: $n \in \mathbb{N}, n \geq 4$, the number to be tested
if n *is even* **then** $\rightarrow$ *false* (= "not prime")
Choose a random base $a \in \{2, 3, \ldots, n-2\}$.
Halve $d := n-1$ until it becomes odd, and count the halvings by k (i.e. then $n-1 = 2^k d$).
$x := a^d \bmod n$
if $x = 1$ **then** $\rightarrow$ *true* (= "probably prime")
loop k *times* **do**
  if $x = n-1$ **then** $\rightarrow$ *true* (= "probably prime")
  $x := x^2 \bmod n$
  if $x = 1$ **then** $\rightarrow$ *false* (= "not prime")
$\rightarrow$ *false* (= "not prime")

The algorithm doubles the exponent until one of three conditions happen:

1. The power (x) becomes -1 (or $n-1$, as $n-1 \equiv -1$ (n)): then n passes the test (i.e. it is "probably prime"), because the square of -1 is 1, and so all subsequent squares will also be 1.
2. The power (x) becomes 1: then n is not a prime, because if it were, then the previous power would have been 1 or -1, and so the test would have already stopped then.
3. The exponent reaches $n-1$ (i.e. the loop terminates after doubling k times): then n is not a prime, because if it were, then the power would be 1, so we would have stopped already.

Definition 4.6. For an $a \in \mathbb{N}^+$, a composite number n is called a **strong pseudoprime** (to base a) if the Miller–Rabin primality test accepts n as a prime number for that a.

Since the Miller–Rabin primality test extends the Fermat primality test, strong pseudoprimes are also Fermat pseudoprimes, but the converse it not true (e.g. 15 is a Fermat pseudoprime for $a = 4$, but not a strong pseudoprime, as demonstrated above).

The Miller–Rabin primality test is better than the Fermat primality test for multiple reasons:

1. It doesn't have equivalents to the Carmichael numbers, i.e. numbers for which the Miller–Rabin test would be almost always wrong.
2. Furthermore, it is correct in at least 75% of the cases (compared to the Fermat test's 50%). This means that if we repeat the test k times, then the error probability is $< 1/4^k$ (compared to $1/2^k$), so the same accuracy can be achieved in half as many steps ($1/4^k = 1/2^{2k}$).
3. But even one step of the test is usually faster than that of the Fermat test, because the Miller–Rabin test doesn't always calculate the whole a^{n-1} power, as it can often stop earlier.

5 Author

This document was written by Uray M. János, partly using the following sources:

[1] https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa (Hungarian)
[2] https://en.wikipedia.org/wiki/Euler%27s_totient_function
[3] https://en.wikipedia.org/wiki/RSA_cryptosystem
[4] Rivest, Shamir, Adleman (1977): A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. https://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-082.pdf
[5] https://en.wikipedia.org/wiki/Fermat_primality_test

Errors and suggestions can be reported here: uray.janos@inf.elte.hu.