Computational Discrete Mathematics: Handout 04

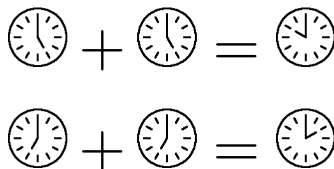
Curtis Bright

September 23, 2021

1 Modular Arithmetic

Modular arithmetic is a system for performing arithmetic in which computations are limited to a finite set with a "wrap-around" effect.

An everyday example of modular arithmetic is used as the time displayed on a clock. For example, 1 hour past 1 o'clock is 2 o'clock and 5 hours past 5 o'clock is 10 o'clock, but 7 hours past 7 o'clock is 2 o'clock. In symbols:



This is because 12 o'clock is treated as "zero"; if you go past tweleve o'clock then what matters is *how much* you went past 12 o'clock. In other words, you can remove all multiples of 12 hours from the time to get the "clock" time.

In modular arithmetic this is written as:

$$5 + 5 \equiv 10 \pmod{12}$$
$$7 + 7 \equiv 2 \pmod{12}$$

While at first this might seem like a mere curiosity, in fact modular arithmetic has an enormous number of applications. For example, it underlies the mathematics used to secure credit card information on the internet.

1.1 Formal Definition

Given integers a, b, and m we say that a and b are *congruent modulo m* and write

$$a \equiv b \pmod{m}$$

if $b - a$ is a multiple of m. Alternatively, if both a and b have the same remainder when divided by m.

For example, $14 \equiv 26 \pmod{12}$ because $26 - 14 = 2 \cdot 12$.

Using the notation $a \bmod m$ to mean the remainder produced by dividing a by m we can check that $14 \equiv 26 \pmod{12}$ by verifying that

$$26 \bmod 12 = 14 \bmod 12 = 2.$$

1.1.1 Note on notation

Note the notations $a = b \bmod m$ and $a \equiv b \pmod{m}$ are quite similar and this can be useful because they mean similar things. However, it is important to note that they are saying different things.

- **In the case $a = b \bmod m$:**

  Here "mod" is being used as a *function* meaning that $b \bmod m$ has a single fixed value for fixed values of b and m.

  For example, $14 \bmod 12 = 2$.

- **In the case $a \equiv b \pmod{m}$:**

  Here "mod" is not a function; the expression $a \equiv b \pmod{m}$ does not define a to be a unique integer but *any* integer that has a remainder of b when divided by m.

  For example, $2 \equiv 14 \pmod{12}$ but also $-10 \equiv 14 \pmod{12}$.

In mathematics the second notation defines what is known as an *equivalence relation* that partitions integers into "equivalence classes". Every integer belongs to one and only one equivalence class mod m.

For example, mod 2 there are exactly two equivalence classes: the class of even numbers and the class of odd numbers. Equivalence classes in modular arithmetic are also known as *residue classes*.

1.1.2 Unique representative

It is often useful to have a single unique *representative* for each equivalence class. In the case of modular arithmetic over the integers it is usually conveinient to define the representative of a class to be the smallest nonnegative integer in the class.

For example, consider the class of integers equivalent to $14 \pmod{12}$ which is $\{\ldots, -22, -10, 2, 14, 26, \ldots\}$. This class has the representative 2.

A list containing all possible representatives is known as a *system of representatives*. For example, with the above choice of representatives we have that every class is represented by one of the following integers:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.$$

12 is not a representative as it is in the class represented by 0.

1.1.3 Computing modular expressions

Given an expression involving integers and $+$, $-$, and $\times$ we can evaluate it modulo m by:

- Reducing each integer to its representative modulo m so that it lies in the range between 0 and $m-1$.
- Evaluating each operation on the reduced integers.
- Immediately after each operation reduce its result modulo m.

For example:

$$\begin{aligned} 20 \times (-89) + 32 &\equiv 6 \cdot 2 + 4 && \pmod 7 \\ &\equiv 12 + 4 && \pmod 7 \\ &\equiv 5 + 4 && \pmod 7 \\ &\equiv 9 && \pmod 7 \\ &\equiv 2 && \pmod 7 \end{aligned}$$

This works because $x \equiv y \pmod m$ implies that $c * x \equiv c * y \pmod m$ where $*$ is addition, subtraction, or multiplication.

How large can the intermediate results get? When working modulo m the intermediate results will never become larger than m^2. The worst case occurs when multiplying $(m-1) \cdot (m-1)$ but this is smaller than m^2.

This holds *regardless* of how large the starting integers are or how many operations there are. This allows us to quickly evaluate complicated expressions in modular arithmetic that would be impractical to evaluate using normal arithmetic.

What about division? You **cannot** arbitrarily divide both sides of an equivalence in modular arithmetic by a constant. Even if x, y, and m are all divisible by c, $x \equiv y \pmod m$ **does NOT** imply that $x/c \equiv y/c \pmod m$. For example, $0 \cdot 2 \equiv 2 \cdot 2 \pmod 4$ but $0 \not\equiv 2 \pmod 4$.

The "correct" way to perform this division would be to cancel c from x, y, and m simultaneously, e.g., $0 \cdot 2 \equiv 2 \cdot 2 \pmod 4$ implies $0 \equiv 2 \pmod 2$.

1.2 Modular inverses

As demonstrated above, cancellation using division has to be done carefully. At first it would seem as if we cannot even define division in modular arithmetic. For example, what would $1/3 \pmod{10}$ even mean? However, we will see it *will* be possible to give meaning to this expression using modular inverses.

1.2.1 Defining division

In modular arithmetic we don't deal with *fractions* of a number or *numerical approximations* of a number. For example, an equation like $1/3 = 0.333\ldots$ is meaningless in modular arithmetic.

However, we can still *algebraically* deal with expressions like $1/3$. How can we make sense of this? Let's say we denote the quantity $1/3$ (whatever that means) by x. If this expression is to make

sense then $3x$ should be 1. In other words, to determine if we can make sense of $1/3$ we would to find x (if it exists) which satisfies the congruence

$$3x \equiv 1 \pmod{m}.$$

By definition, this means that $3x - 1$ is a multiple of m (let's say that it is k times m for some integer k).

In other words, in order to find the modular inverse of 3 (modulo m) we want to solve

$$3x - 1 = km \qquad \text{for integers } x \text{ and } k.$$

1.2.2 Look familiar?

The Euclidean algorithm now comes to the rescue! Rewriting this with $y = -k$, we want to solve

$$3x + ym = 1.$$

Recall that the Euclidean algorithm gives us a method to find integers x and y such that

$$3x + ym = \gcd(3, m). \tag{$*$}$$

Thus, if $\gcd(3, m) = 1$ then an x exists so that $3x$ reduces to 1 modulo m; x is said to be the *modular inverse* of 3 and is typically denoted by 3^{-1}. Note that this assumes the modulus m is clear from the context; to be unambiguous it should be denoted $3^{-1} \pmod{m}$.

1.2.3 Example

Note that 3 and 10 share no common factors, so their gcd is 1; numbers that share no common factor are said to be *coprime*. The above reasoning indicates that we should be able to determine the value of 3^{-1} mod 10.

To do this, we run the Euclidean algorithm on 3 and 10:

$$\begin{array}{llll}
10 \cdot 1 & + 3 \cdot 0 & = 10 & \text{initialization} \\
10 \cdot 0 & + 3 \cdot 1 & = 3 & \text{initialization} \\
10 \cdot 1 & + 3 \cdot (-3) & = 1 & \text{subtract } \lfloor 10/3 \rfloor = 3 \text{ times the second from the first}
\end{array}$$

Thus, $y = 1$ and $x = -3$ is a solution of ($*$).

Note that 7 is the unique representative (as previously defined) of the equivalence class containing $x = -3$.

Indeed, $3 \times 7 = 21 \equiv 1 \pmod{10}$ and we have found an inverse of 3! Thus, we say that $3^{-1} \equiv 7 \pmod{10}$.

1.2.4 Uniqueness

Say a and m are coprime. Could it be that there are *multiple* possible values for $a^{-1} \pmod{m}$?

At first glance this isn't crazy, since there *will* be infinitely many integer solutions (x, y) to the equation

$$ax + my = 1. \tag{1}$$

However, all solutions x will belong to the same congruence class modulo m; thus there is indeed only a single solution modulo m and modular inverses (if they exist) are unique.

Formally, if (x_0, y_0) is a solution of (1) then *all* solutions are given by

$$(x_0 - mk, y_0 + ak) \qquad \text{for } k \in \mathbb{Z}.$$

1.2.5 Non-existence

When a and m are not coprime, $a^{-1} \pmod{m}$ does not exist because there is no solution of

$$ax + my = 1. \tag{2}$$

Indeed, any common divisor of a and m will divide the left-hand side of (2) so must also divide the right-hand side, i.e., 1—meaning the divisor must be trivial.

For example, $x = 2^{-1} \pmod{10}$ does not exist, since $2x + 10y$ must be even and cannot ever be 1.

1.2.6 Analysis

We've already seen that addition, subtraction, and multiplication of numbers at most m can be done in $O(n^2)$ word operations where $n = \text{len}(m) = O(\log m)$.

Furthermore, we've also seen that the extended Euclidean algorithm when run on numbers at most m uses $O(n^2)$ word operations.

Assuming that the operands are specified using the standard unique representative (i.e., by numbers at most m) then we can perform addition, subtraction, multiplication, and division (by a number coprime to m) in $O(n^2)$ word operations.

1.3 Modular exponentiation

What about computing $a^k \pmod{m}$? Of course, this could be computed by multiplying a by itself $k - 1$ times (and reducing the result modulo m after each multiplication so that the number does not grow extremely large).

1.3.1 Cost analysis

As usual, suppose that $n = \text{len}(m)$. We can also assume that a is reduced modulo m. Then multiplying by a uses $O(n^2)$ word operations and reducing the result modulo m also uses $O(n^2)$ word operations.

Since this must be done $k-1$ times, computing $a^k \pmod m$ using this method requires $O(kn^2)$ word operations.

Is this acceptable? At first, this may seem like a decent computational cost, as it is linear in k. Linear running times are usually good—assuming they are in the *size of the input*. But in this case the size of the input is $\text{len}(k)$, not k itself. An $O(k)$ running time is actually **exponential** in the size of the input—which is very bad.

For example, imagine trying to compute $2^{1234567890} \pmod{10}$. In this case the result will be a single digit and the exponent 1,234,567,890 fits in a single 32-bit word. However, using the simple algorithm of repeated multiplication will require over 1.2 billion multiplications!

Of course, you really don't want to be doing over a billion multiplications to compute a single digit if you don't have to!

1.3.2 Can we do better?

Can we compute $a^k \pmod m$ faster than using repeated mutliplication?

The answer is yes! In fact, we can compute it using $O(\log k)$ operations modulo m which is very fast—even for k like $k = 1{,}234{,}567{,}890$.

Moreover, the algorithm essentially only uses a single fairly simple trick.

The fact that this trick exists makes modular exponentiation a very important and useful operation that has a wealth of applications. For example, later we'll see how it underlies how private information is kept secure on the internet.

1.3.3 Repeated squaring

Fast modular exponentiation relies on a trick known as *repeated squaring*. The idea is that the sequence of numbers

$$\begin{gathered} a \pmod m \\ a^2 \pmod m \\ a^4 \pmod m \\ a^8 \pmod m \\ \vdots \\ a^{2^i} \pmod m \\ \vdots \\ a^{2^{\lfloor \lg(k) \rfloor}} \pmod m \end{gathered}$$

can be computed quickly, because each number in the sequence is the square of the previous number. For example, once $a^4 \pmod m$ is known, one can compute $a^8 \pmod m = a^4 \cdot a^4 \pmod m$ using a single multiplication mod m. Since there are $O(\text{len}(k))$ numbers in the sequence, computing them all takes $O(\text{len}(k)\,\text{len}(m)^2)$ word operations.

How does this help? Okay, so we are quickly able to compute $a^{2^i} \pmod{m}$ for $i = 0, \ldots, \lfloor \lg(k) \rfloor$.

However, it's not immediately clear how this helps us compute $a^k \pmod{m}$ for arbitrary k—since unless it happens to be the case that k is a power of 2 the quantity that we *want* to compute will not be in the list of numbers that we *did* compute.

The solution The solution is to "build" the number that we want (i.e., $a^k \pmod{m}$) out of the numbers that we computed (i.e., $a^{2^i} \pmod{m}$ for i up to $l := \lfloor \lg(k) \rfloor$).

How can we do this? Note that k can always be represented as a sum of powers of 2:

$$k = \sum_{i=0}^{l} k_i \cdot 2^i$$

where k_i is the ith bit in the binary representation of k. For example, $19 = 2^4 + 2^1 + 2^0$ because 10011 is the binary representation of 19.

Then we have that

$$\begin{aligned} a^k &\equiv a^{k_0 2^0 + k_1 2^1 + k_2 2^2 + \cdots + k_l 2^l} \pmod{m} \\ &\equiv a^{k_0 2^0} \cdot a^{k_1 2^1} \cdot a^{k_2 2^2} \cdots a^{k_l 2^l} \pmod{m} \\ &\equiv \prod_{\substack{i=0 \\ k_i = 1}}^{l} a^{2^i} \pmod{m}. \end{aligned}$$

Thus once $a^{2^i} \pmod{m}$ have been determined for $i \le l$ we can compute $a^k \pmod{m}$ by multiplying together the $a^{2^i} \pmod{m}$ for which $k_i = 1$ (i.e., when the ith bit of k in binary is 1).

For example, $a^{19} \equiv a^{16} \cdot a^2 \cdot a^1 \pmod{m}$ because we have that $k_4 = k_1 = k_0 = 1$ and $k_2 = k_3 = 0$.

Cost analysis First, note that the binary representation of a number k can be computed in $O(\text{len}(k))$ word operations.

Once we have that and the sequence of numbers from before we have to perform at most $l - 1 = O(\text{len}(k))$ multiplications modulo m. Note that the worst case occurs when the binary representation of k contains nothing but 1s and the best case occurs when k is a power of 2.

Thus, the cost of computing $a^k \pmod{m}$ via repeated squaring is the sum of:

- The cost of performing repeated squaring: $O(\text{len}(k)\,\text{len}(m)^2)$
- The cost of finding the binary representation of k: $O(\text{len}(k))$
- The cost of building the result out of the numbers computed via repeated squaring: $O(\text{len}(k)\,\text{len}(m)^2)$

In total, this requires $O(\text{len}(k)\,\text{len}(m)^2)$ word operations.

1.3.4 Built-in Sage function

The Sage function `power_mod` performs modular exponentiation using repeated squaring.

`power_mod(a, k, m)` will produce the same result as `a^k % m` but in general `power_mod` will be much faster than using `^`.

We can demonstrate this using the `timeit` command:

```
[ ]: timeit('2 ^ 1234567890 % 10')
```

```
[ ]: timeit('power_mod(2, 1234567890, 10)')
```

1.4 Simultaneous linear congruences

Finally, we'll cover a classic number theory problem (whose solution dates back to the 3rd century by the Chinese mathematician Sun-tzu).

Suppose we want to solve the following system of congruences for x:

$$\begin{aligned} x &\equiv 2 \pmod 3 \\ x &\equiv 3 \pmod 5 \\ x &\equiv 2 \pmod 7 \end{aligned}$$

How could you find a solution x?

1.4.1 Brute force

The obvious answer: try $x = 0, 1, 2, \ldots$ one at a time and stop if you find an x which satisfies each of the congruences simultaneously.

However, what if this method runs indefinitely?

How far do you need to go? Note that if x is a solution then $x + k \cdot 3 \cdot 5 \cdot 7$ is also a solution for all integers k. This follows because $k \cdot 3 \cdot 5 \cdot 7$ reduces to 0 modulo 3, 5, and 7.

Thus, in the worst case you have to examine all x from 0 up to $104 = 3 \cdot 5 \cdot 7 - 1$.

Analysis If $P = \prod_i m_i$ is the product of the moduli m_i then in the worst case P possibilities for x will have to be tried, so this method runs in $O(P)$ arithmetic operations modulo m_i.

Again, this might sound reasonable at first but it is exponential in $\operatorname{len}(m_i)$. Just imagine increasing the first modulus from 3 to 3 trillion. This doesn't even increase the length of the moduli (assuming a 64-bit word size) but increases the running time of this method by a factor of trillion!

1.4.2 Example solution

Consider the first two congruences. Translating them into equations over the integers, we want to find integers x, k, and l which satisfy

$$x = 2 + 3k \qquad\qquad x = 3 + 5l$$

Equating these, we want to solve $2 + 3k = 3 + 5l$ which is just a slightly-disguised linear Diophantine equation that can be rewritten in the form

$$3k - 5l = 1. \tag{3}$$

Since $\gcd(3, 5) = 1$, Bézout's identity tells us this equation has a solution in the integers and the extended Euclidean algorithm allows us to find the solution $(k, l) = (2, 1)$ leading to $x = 8$ which satisfies the first two congruences. Recall that *all* solutions of (3) are given by $(k, l) = (2 + 5a, 1 + 3a)$ for $a \in \mathbb{Z}$ leading to $x = 8 + 15a$ as the general solution of the first two congruences.

Of course, this is only a solution of the first two congruences and we still have a third congruence to consider. However, $x = 8 + 15a$ for $x \in \mathbb{Z}$ is equivalent to the congruence $x \equiv 8 \pmod{15}$ so we can rewrite the system as:

$$\begin{aligned} x &\equiv 8 \pmod{15} \\ x &\equiv 2 \pmod{7} \end{aligned}$$

Now we can apply the same procedure as before! To solve these we want to solve $x = 8 + 15l = 2 + 7k$ in integers which can be rewritten as $15l - 7k = -6$. This has a solution since -6 is a multiple of $\gcd(15, 7) = 1$. The extended Euclidean algorithm produces

$$15 \cdot 1 - 7 \cdot 2 = 1 \qquad \xrightarrow{\text{multiply by } -6} \qquad 15 \cdot (-6) - 7 \cdot (-12) = -6$$

so $(l, k) = (-6, -12)$ and $x = -82$ is a solution. As previously noted, all numbers of the form $-82 + 105a$ for $a \in \mathbb{Z}$ are also solutions. The canonical representation of x is then 23. Indeed, we find that:

$$\begin{aligned} 23 &\equiv 2 \pmod{3} \\ 23 &\equiv 3 \pmod{5} \\ 23 &\equiv 2 \pmod{7} \end{aligned}$$

1.5 Chinese remainder theorem

The formal mathematical result is known as the "Chinese remainder theorem". It says that if $m_0, \ldots, m_{r-1}$ are pairwise coprime ($\gcd(m_i, m_j) = 1$ for $i \neq j$) then

$$x \equiv a_0 \pmod{m_0}$$
$$\vdots$$
$$x \equiv a_{r-1} \pmod{m_{r-1}}$$

has a unique solution x modulo $m = m_0 \cdots m_{r-1}$.

1.5.1 General solution

Given $a_0, \ldots, a_{r-1}$ and $m_0, \ldots, m_{r-1}$ how can we find the unique solution x?

In fact, the general solution is of the form

$$x \equiv a_0 L_0 + a_1 L_1 + \cdots + a_{r-1} L_{r-1} \pmod{m}$$

where the L_i are computed so that

$$\begin{aligned} L_i &\equiv 1 \pmod{m_i} \\ L_i &\equiv 0 \pmod{m_k} \qquad \text{for all } k \neq i. \end{aligned}$$

Reducing the expression for x modulo m_i shows that $x \equiv a_i \pmod{m_i}$, i.e., x is indeed a solution. However, how can we compute values for the L_i?

Since L_i is a multiple of m_k for all $k \neq i$ it follows that L_i is also a multiple of the product $\prod_{k \neq i} m_k = \frac{m}{m_i}$. Thus we want to find L_i such that

$$\begin{aligned} L_i &\equiv 1 \pmod{m_i} \\ L_i &\equiv 0 \pmod{m/m_i} \end{aligned}$$

which is equivalent to the linear Diophantine equation $s \cdot m_i + t \cdot (m/m_i) = 1$. Since $\gcd(m_i, m/m_i) = 1$ we know the extended Euclidean algorithm allows us to find a satisfying (s, t). Setting $L_i := t \cdot (m/m_i)$ gives $L_i \equiv 1 \pmod{m_i}$ and $L_i \equiv 0 \pmod{n/m_i}$.

1.5.2 A general formula

Explicitly, if you'd like a general formula, note that the t from above is defined to be $(m/m_i)^{-1} \bmod m_i$. Since $L_i = t \cdot (m/m_i)$ the general solution for x is

$$x = \sum_{i=0}^{r-1} a_i \cdot ((m/m_i)^{-1} \bmod m_i) \cdot (m/m_i). \tag{4}$$

1.5.3 Cost analysis

First, we find the cost of computing $m = \prod_i m_i$. Note that $\mathrm{len}(\prod_{i=0}^{k} m_i) = O(\sum_{i=0}^{k} \mathrm{len}(m_i)) = O(\mathrm{len}(m))$. Computing each successive product for $k = 1, \ldots, r-1$ results in a total word operation cost of

$$O\left(\sum_{k=1}^{r-1} \mathrm{len}\left(\prod_{i=0}^{k} m_i\right) \mathrm{len}(m_i)\right) = O(\mathrm{len}(m)) \cdot \sum_{k=1}^{r-1} O(\mathrm{len}(m_i)) = O(\mathrm{len}(m)^2).$$

Next, once m is computed we can compute m/m_i for $i = 0, \ldots, r-1$ using a total word operation cost of

$$O\left(\sum_{i=0}^{r-1} \mathrm{len}(m/m_i)\,\mathrm{len}(m_i)\right) = O(\mathrm{len}(m)) \cdot \sum_{i=0}^{r-1} O(\mathrm{len}(m_i)) = O(\mathrm{len}(m)^2).$$

Similarly, we can compute $(m/m_i) \bmod m_i$ in the same cost. Next, we can compute $(m/m_i)^{-1} \bmod m_i$ for $i = 0, \ldots, r-1$ using a total word operation cost of

$$O\left(\sum_{i=0}^{r-1} \mathrm{len}(m_i)^2\right) = O\left(\left(\sum_{i=0}^{r-1} \mathrm{len}(m_i)\right)^2\right) = O(\mathrm{len}(m)^2).$$

Similarly, we can compute $L_i = ((m/m_i)^{-1} \bmod m_i) \cdot (m/m_i)$ in a word operation cost of

$$O\left(\sum_{i=0}^{r-1} \mathrm{len}(m_i)\,\mathrm{len}(m/m_i)\right) = \sum_{i=0}^{r-1} O(\mathrm{len}(m_i)) \cdot O(\mathrm{len}(m)) = O(\mathrm{len}(m)^2).$$

Finally, to compute x from (4) assuming that $\mathrm{len}(a_i) = O(\mathrm{len}(m_i))$ uses a total word operation cost of

$$O\left(\sum_{i=0}^{r-1} \mathrm{len}(m_i) \cdot \mathrm{len}(m)\right) = \sum_{i=0}^{r-1} O(\mathrm{len}(m_i)) \cdot O(\mathrm{len}(m)) = O(\mathrm{len}(m)^2).$$

In summary, we can solve simulaneous linear congruences using the formula (4) provided by the Chinese remainder theorem and this requires $O(\mathrm{len}(m)^2)$ word operations.

1.5.4 Takeaway

In summary, the Chinese remainder theorem provides a way of solving simultaneous linear congruences when the moduli of the congruences are pairwise coprime. It also allows us to efficiently compute the unique solution of such congruences—namely, quadratic in the length of the product of the moduli.

Applications of the Chinese remainder theorem reach far beyond what it may seem like at first glance. In fact, the Chinese remainder theorem essentially says that doing computations modulo $m = m_0 \cdots m_{r-1}$ is in a sense equivalent to doing computations modulo each of $m_0, \ldots, m_{r-1}$. This is useful because computations modulo m_i are cheaper than computations modulo m.

It also opens the possibility for parallelization: a computation that needs to be done modulo m can instead be done modulo m_i for each $i = 0, \ldots, r-1$. Since each computation modulo m_i is independent, the entire computation can be distributed across r processors. Once they've all finished the final result modulo m can be computed using the Chinese remainder theorem.

For example, "Chinese remaindering" methods are very useful for linear system solving or determinant computation on integer matrices.

1.6 Fermat's Little Theorem

A classic theorem of the mathematician Fermat (known as his "little" theorem—not his more famous "last" theorem) is the following:

If p is a prime and a is not a multiple of p, then

$$a^{p-1} \equiv 1 \pmod{p}. \tag{5}$$

It is easy to see that the restriction on a is necessary: if a *is* a multiple of p then $a \equiv 0 \pmod{p}$ meaning $a^{p-1} \equiv 0 \pmod{p}$.

However, by multiplying both sides of (5) by a gives an easier-to-state form that works for all a. If p is a prime and a is an integer then

$$a^p \equiv a \pmod{p}.$$

1.6.1 Applications

First, this theorem can often be used to detect prime numbers—or more accurately to detect *composite* numbers (numbers that have more than one prime factor).

If you have a number p that you want to test for primality then take some random a and compute $a^p \bmod p$; if the result is something other than a then p cannot be prime. Conversely, if the result is a then p may or may not be prime.

Second, Fermat's little theorem provides a simple way of computing modular inverses, because an easy consequence of it is that if a and p are coprime then

$$a^{p-2} \equiv a^{-1} \pmod{p}.$$

How fast is this? We know that a^{p-2} can be computed using repeated squaring with $O(\text{len}(p-2) \cdot \text{len}(p)^2) = O(\text{len}(p)^3)$ word operations.

Although this is a simple method it is not the most efficient, since we saw that modular inverses can be computed using the extended Euclidean algorithm which uses only $O(\text{len}(p)^2)$ word operations.

1.6.2 Generalization

Finally, we'll note one way in which the Fermat's little theorem can be generalized and which will be particularly useful in cryptography.

Suppose p and q are distinct prime numbers. We can use Fermat's little theorem and the Chinese remainder theorem to derive a version of Fermat's little theorem that works modulo pq.

Suppose a is coprime with both p and q. We can derive the following congruences by raising a^{p-1} by $q-1$ (in the first case) and a^{q-1} by $p-1$ (in the second case):

$$a^{(p-1)(q-1)} \equiv 1 \pmod{p}$$
$$a^{(p-1)(q-1)} \equiv 1 \pmod{q}$$

Since p and q are coprime, the Chinese remainder theorem says that $x = a^{(p-1)(q-1)}$ has a unique solution modulo pq. However, one easily sees that $x = 1$ is a solution (after all, $1 \bmod p = 1$ and $1 \bmod q = 1$). Thus the general solution is

$$x = a^{(p-1)(q-1)} \equiv 1 \pmod{pq}. \tag{6}$$

For example, with $(p, q) = (3, 5)$ we have that $2^{(3-1)(5-1)} \equiv 2^8 \equiv 256 \equiv 1 \pmod{3 \cdot 5}$. However, we also know the congruence will hold even if p and q are replaced with primes with 100s of digits.

Note: This is a special case of Euler's theorem, which says that if a and n are coprime then

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

where $\varphi(n)$ counts the number of positive integers up to n which are coprime to n. When $n = pq$ for distinct primes p and q one has

$$\varphi(pq) = (p-1)(q-1)$$

which is consistent with (6).