Lecture 14: Modular Arithmetic

Eric Vigoda
Discrete Mathematics for Computer Science

March 2, 2026

14 Modular Arithmetic

Later, when we're looking at cryptography algorithms it will be important to work with huge numbers, e.g., several thousand digits long. We will also be taking powers with respect to these huge numbers. For example, suppose x and y are 1000 bit numbers, and we want to compute x^y. Say $x \approx 2^{1000}$ and $y \approx 2^{1000}$ then $x^y \approx \left(2^{1000}\right)^{2^{1000}} = 2^{1000 \cdot 2^{1000}}$, and therefore the number has approximately $1000 \cdot 2^{1000}$ bits. That's way too much to handle – writing it down would be more data than the total sum of what's been written in the history of humanity.

To deal with these huge numbers we look at modular arithmetic which corresponds to looking at the remainder when dividing. For example the hours on a clock are the remainder when dividing by 12, and we count the hours by $0, 1, \ldots, 11$, and then we return to 0 and continue again. This corresponds to counting modulo 12.

A simple example is modulo 2, which corresponds to the least significant bit of a number.

For integer x,

$$x \bmod 2 = \begin{cases} 1 & \text{if } x = 2k+1 \text{ for some integer } k \\ 0 & \text{if } x = 2k \text{ for some integer } k. \end{cases}$$

In other words, $x \bmod 2$ is 1 if x is odd and 0 if x is even.

More generally, for integers x and N where $N \geq 1$, $x \bmod N$ is the remainder when dividing x by N. Equivalently,

$$x \bmod N = r \quad \text{where } x = qN + r \text{ for integers } q, r \text{ with } 0 \leq r < N.$$

As a simple example, when working **mod 3** there are 3 possible values $0, 1$, or 2, and hence the integers are divided into 3 equivalence classes:

$$\ldots, -9,\ -6,\ -3,\ \mathbf{0},\ 3,\ 6,\ 9, \ldots$$
$$\ldots, -8,\ -5,\ -2,\ \mathbf{1},\ 4,\ 7,\ 10 \ldots$$
$$\ldots, -7,\ -4,\ -1,\ \mathbf{2},\ 5,\ 8,\ 11 \ldots$$

The numbers in the same equivalence class are not equal but they are equivalent when we are working in the setting of **mod 3**. Hence we mark it using the $\equiv$ symbol, and say that the numbers x and y are **congruent** modulo 3 if $x \equiv y \bmod 3$.

To reason formally about modular arithmetic, we first introduce the notion of divisibility.

Definition 14.1 (Divisibility)**.** For integers a and b with $a \neq 0$, we say that

$$a \mid b$$

(read "a divides b") if there exists an integer k such that

$$b = ak.$$

Equivalently, b is a multiple of a. Note that $a \mid b$ means a divides b, not the other way around; the divisor appears on the left.

If $a \mid b$, then dividing b by a leaves remainder 0. If no such integer k exists, we write $a \nmid b$.

Definition 14.2 (Congruence Modulo N)**.** For integers x, y and an integer $N \geq 1$, we say

$$x \equiv y \pmod{N}$$

if

$$N \mid (x - y).$$

In words, two integers are congruent modulo N precisely when they leave the same remainder upon division by N.

Why does this definition make sense? Suppose

$$x = q_1 N + r \qquad \text{and} \qquad y = q_2 N + r$$

for integers q_1, q_2 and the same remainder r with $0 \leq r < N$. Then

$$x - y = (q_1 - q_2)N,$$

so N divides $x - y$.

Thus the following two statements are equivalent:

$$x \equiv y \pmod{N} \quad \iff \quad x \bmod N = y \bmod N \quad \iff \quad N \mid (x - y),$$

where $N \mid (x-y)$ means N divides $(x-y)$ (so $x-y$ is a multiple of N). In other words, congruence modulo N means "equal after ignoring multiples of N." Thus modular arithmetic replaces exact equality with equality up to multiples of N.

Geometric intuition (clock arithmetic).
Working modulo N is like moving around a clock with N positions labeled $0, 1, \ldots, N-1$. Adding N corresponds to making one full revolution around the clock and returning to the same position. Thus numbers that differ by multiples of N (e.g., x, $x + N$, $x + 2N$, $x - N$) all land at the same point on the clock, which is why they are considered equivalent modulo N.

Here are some examples:

$$-9 \equiv 6 \bmod 3$$

$$5 \equiv 10 \bmod 3$$

$$-60 \equiv 16 \bmod 19$$

$$91 \equiv 3 \bmod 22$$

We can also have a string of equivalences, and we simply put **mod 3** at the end of the line to indicate that the entire line is with respect to **mod 3**, e.g.:

$$4 \equiv 1 \equiv -11 \bmod 3$$

$$-76 \equiv -16 \equiv 4 \bmod 20$$

The consequence of this equivalence is that the numbers can be interchanged.

Fact 14.3. *If $x \equiv x' \bmod N$ and $y \equiv y' \bmod N$ then:*

$$x + y \equiv x' + y' \bmod N$$

$$xy \equiv x'y' \bmod N$$

In other words, we may replace numbers by equivalent ones before performing arithmetic, which greatly simplifies computations.

> **Replacement Principle.** If two numbers are congruent modulo N, then one may be replaced by the other inside any arithmetic expression (addition, subtraction, multiplication, or exponentiation).

This basic fact can be used to simplify calculations, here is a simple example to illustrate such a simplification.

Example 1: Calculate

$$2^{345} \bmod 31$$

When performing a calculation **mod N** then we want to "reduce" our answer and report it in simplest terms which is an integer in the set $\{0, 1, \ldots, N-1\}$. Returning to Example 1, our strategy is to look for a power of 2 that is congruent to 1 modulo 31, since powers of 1 are easy to compute. Since $2^5 = 32 \equiv 1 \pmod{31}$ and $345 = 5 \cdot 69$, we obtain

$$2^{345} \equiv \left(2^5\right)^{69} \equiv (32)^{69} \equiv 1^{69} \equiv 1 \bmod 31.$$

Fact 14.4 (Reduction Rule)**.** *If $a \equiv b \pmod{N}$, then for any integer $k \geq 1$,*

$$a^k \equiv b^k \pmod{N}.$$

This allows us to reduce intermediate values at every step of a calculation, and will later be a key and essential step in performing fast modular exponentiation.

Example 2: Compute several powers of 3 modulo 7.

$$\begin{aligned}
3^1 &\equiv 3 \pmod{7},\\
3^2 &= 9 \equiv 2 \pmod{7},\\
3^3 &= 3 \cdot 2 = 6 \pmod{7},\\
3^4 &= 3 \cdot 6 = 18 \equiv 4 \pmod{7},\\
3^5 &= 3 \cdot 4 = 12 \equiv 5 \pmod{7},\\
3^6 &= 3 \cdot 5 = 15 \equiv 1 \pmod{7}.
\end{aligned}$$

We have discovered that

$$3^6 \equiv 1 \pmod{7}.$$

This is extremely useful, because higher powers now repeat. For example,

$$3^{100} = 3^{6 \cdot 16 + 4} = (3^6)^{16} 3^4 \equiv 1^{16} \cdot 3^4 \equiv 4 \pmod{7}.$$

The powers of 3 modulo 7 eventually repeat. This phenomenon is called *periodicity*: after some point, powers cycle through the same values. Finding such repetitions allows us to compute very large powers efficiently. This phenomenon is called *periodicity*: powers eventually repeat because there are only N possible remainders modulo N, namely $0, 1, \ldots, N-1$. Hence among the values $a^1, a^2, \ldots, a^{N+1}$ taken modulo N, two must have the same remainder, and from that point onward the sequence repeats.

We now introduce a general method that works even when no obvious periodicity is visible.

Example 3: Compute

$$7^{100} \bmod 13.$$

Instead of multiplying 7 one hundred times, we repeatedly square, reducing modulo 13 after each step.

First compute successive powers of 7:

$$
\begin{aligned}
7^1 &\equiv 7 \pmod{13},\\
7^2 &= 49 \equiv 10 \pmod{13},\\
7^4 &= (7^2)^2 \equiv 10^2 = 100 \equiv 9 \pmod{13},\\
7^8 &= (7^4)^2 \equiv 9^2 = 81 \equiv 3 \pmod{13},\\
7^{16} &= (7^8)^2 \equiv 3^2 = 9 \pmod{13},\\
7^{32} &= (7^{16})^2 \equiv 9^2 = 81 \equiv 3 \pmod{13},\\
7^{64} &= (7^{32})^2 \equiv 3^2 = 9 \pmod{13}.
\end{aligned}
$$

The key observation is that repeated squaring corresponds to writing the exponent in **binary**. Since computers represent integers in base 2, this is why fast exponentiation is an especially natural algorithmic approach and leads to efficient algorithms.

Write 100 in binary:

$$100 = 64 + 32 + 4 = 2^6 + 2^5 + 2^2.$$

Thus,

$$7^{100} = 7^{64} \cdot 7^{32} \cdot 7^4,$$

so once we know the powers corresponding to powers of two, we can multiply only the necessary terms together.

Substituting the values computed above,

$$7^{100} \equiv 9 \cdot 3 \cdot 9 \pmod{13}.$$

Now simplify:

$$9 \cdot 3 = 27 \equiv 1 \pmod{13},$$

so

$$7^{100} \equiv 1 \cdot 9 = 9 \pmod{13}.$$

Notice that we computed only powers corresponding to powers of two. This reduces the number of multiplications from 100 down to about $2\log_2(100)$ multiplications, since we used the binary expansion of 100 together with repeated squaring.

The previous example reveals an important pattern. Rather than multiplying x by itself many times, we repeatedly *square* smaller powers and reuse earlier computations. Each step replaces the problem of computing x^y with the smaller problem of computing a power whose exponent is approximately half as large.

This observation leads to a general recursive method for computing large powers efficiently. Instead of performing y multiplications, we reduce the exponent by a factor of two at every step, dramatically decreasing the amount of work required.

Fast Modular Exponentiation (Recursive Form)

To compute $x^y \bmod N$, we use the following recursive rule.

Base case:

$$x^0 \equiv 1 \pmod{N}.$$

Recursive step:

$$x^y \bmod N = \begin{cases} \left(x^{y/2} \bmod N\right)^2 \bmod N & \text{if } y \text{ is even},\\ x\left(x^{\lfloor y/2 \rfloor} \bmod N\right)^2 \bmod N & \text{if } y \text{ is odd}. \end{cases}$$

After every multiplication, we reduce modulo N so that the intermediate calculations don't blow up in size.

Key idea. Each step cuts the exponent roughly in half. Therefore computing $x^y \bmod N$ requires only about $\log_2 y$ recursive steps instead of y multiplications.

Running time intuition. Let $T(y)$ denote the number of multiplications needed to compute x^y. Each recursive step replaces y by $\lfloor y/2 \rfloor$, so

$$T(y) = T(\lfloor y/2 \rfloor) + O(1).$$

After about $\log_2 y$ steps we reach the base case, giving

$$T(y) = O(\log y).$$

If y has n bits (so $y \approx 2^n$), then the algorithm uses only $O(n)$ multiplications. Standard multiplication of two n-bit integers takes $O(n^2)$ time, and modular multiplication has the same asymptotic cost. Therefore, the total running time of the algorithm in terms of n, which is the number of bits in the input numbers, is $O(n^3)$.

In the next lecture we will study when numbers have multiplicative inverses modulo N, and how to compute them efficiently using the Euclidean algorithm.

14.1 Additional Exercises

Example 4: Compute

$$7^{402} \bmod 100$$

Example 5: Compute

$$10! \bmod 11$$