6.857 R01: Review of Modular Arithmetic

Andrew He

February 8, 2019

1 Introduction

We're going to cover modular arithmetic and a few useful theorems. We'll also take note of how to implement these operations.

2 Modular Arithmetic

We'll start with some motivation.

Example 1 (Last digits). **Q:** What is the last digit of $298753 + 98398$? How about $287124 \cdot 17643$?

A: The last digit of the sum/product only depends on the last digits of the operands; thus, they are $3 + 8 = 1\boxed{1}$ and $4 \cdot 3 = 1\boxed{2}$.

Note that the last digit is just the number modulo 10. This generalizes to become modular arithmetic. We'll say "a is congruent to b modulo m" and write $a \equiv b \pmod{m}$ if and only if:

- $a\%m = b\%m$, or equivalently,
- $m \mid (b - a)$

. I'm using % like in code. The second form is the most useful for proving things, but somewhat cumbersome to use otherwise.

We can check that addition, subtraction, and multiplication "work properly" modulo m (e.g. you get consistent results whether adding 2 or 12 modulo 10).

Remark. For a more formal definition, we observe that modular congruence is an equivalence relationship on the integers, so we define addition/subtraction/multiplication on the equivalence classes. For those who know more math, we're defining an addition group and a multiplication group. For those who know even more math, we're just taking a quotient group of the integers.

For the rest of this lecture, we'll mostly work with prime modulo; they have some particularly nice properties, and we'll show how to generalize them near the end.

2.1 Implementation

Modular addition and subtraction modulo m can be in $O(\log m)$ time, just like normal addition and subtraction, just using grade-school formulas, as there are $O(\log m)$ digits.

Likewise, multiplication takes $O((\log m)^2)$ time, using grade-school multiplication, or $O(\log m \log \log m)$ with FFT-based techniques.

Either way these techniques are all fast; that means we can write algorithms using these operations.

3 Modular Division

We've seen modular addition and subtraction (which are inverses), and modular multiplication. What about division? Division is very useful; it allows us solve linear equations and reverse multiplication, which gives a ton of power.

Claim 2. Division is pretty much equivalent to the existence of multiplicative inverses; if a^{-1} is the multiplicative inverse of a so that $a \cdot a^{-1} = 1$, then $b/a = b \cdot a^{-1}$.

Theorem 3. *Given any prime modulo p and residue $a \not\equiv 0 \pmod{p}$, there exists a unique value $b \pmod{p}$ such that $ab \equiv ba \equiv 1 \pmod{p}$. We'll define and write $a^{-1} = b$.*

One proof of this theorem is by considering the arithmetic sequence

$$0, a, 2a, 3a, \ldots, (p-1)a\ .$$

This sequence must have all distinct residues: otherwise, if $ia \equiv ja$, then $p \mid (ia - ja) = (i-j)a$, which isn't possible, as $p \nmid i - j$ and $a \not\equiv 0 \pmod{p}$. The sequence has p distinct residues, so one of them must be 1, so there's some k such that $ak \equiv 1 \pmod{p}$.

This means that modulo p, we can divide by any non-zero element! Division works in all the ways that normal division does. Even "fractions" work like we'd expect.

Example 4. Let's work in modulo 7. Note that over rationals, $1/2+1/3 = 1/6$. How about over modulo 7? Well, $2^{-1} = 4$, $3^{-1} = 5$, and $6^{-1} = 6$, and indeed, $4 + 5 \equiv 6$. Statements like these still work out because we can multiply both sides by 6 and clear denominators, as we'd expect.

3.1 Implementation

We've shown that there exist modular inverses, but we haven't shown a way to find them. The standard technique is the extended Euclidean algorithm, which you can find by Googling. In SAGE (a Python-like compute algebra system), you can just call `inverse_mod(a, m)`. This is also fast: it runs in $O((\log m)^2)$ or $O(\log m)$ multiplications, which is also polynomial time.

4 Modular Exponentiation

Modular exponentiation is where we start to get real cryptographic power.

Unlike multiplication, we don't define exponentiation in some special way modulo m. Exponentiation is simply repeated multiplication: $g^3 \equiv g \cdot g \cdot g \pmod{m}$. This means that any standard identities like $g^{a+b} = g^a g^b$ and $g^{ab} = (g^a)^b$ work. Note that the exponents are *not* taken modulo m, unlike g.

The most useful structure comes from iterated multiplication or the exponents of a number.

Example 5. Consider powers of 2 modulo 7. We have:

- $2^0 \equiv 1$.
- $2^1 \equiv 2$.
- $2^2 \equiv 4$.
- $2^3 \equiv 1$.
- $2^4 \equiv 2$.
- $2^5 \equiv 4$.

- $2^6 \equiv 1$.

Note that, after we hit $2^3 \equiv 1$, we continue to cycle through the same values, because 1 is the multiplicative identity. Also, the multiplicative inverse of 2 exists, so this sequence is always "reversible": given 2^a, we know 2^{a-1} uniquely. Thus, we can see that this sequence is actually cyclic going forwards and backwards, and each cycle contains only unique elements. There are only $p-1$ different residues, so it has to cycle within $p-1$ elements.

This turns out to be very useful. We'll call the cycle length the *order* of 2 modulo 7, and sometimes will write it as $\text{ord}_7(2) = 3$.

Theorem 6 (Fermat's Little Theorem). *Given a prime modulo p, and a residue $a \not\equiv 0 \pmod{p}$,*

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

Equivalently, $\text{ord}_p(a) \mid (p-1)$ for all a.

The biggest takeaway from this theorem is that we can essentially take exponents modulo $p-1$, as $a^{k(p-1)+r} \equiv (a^{p-1})^k a^r \equiv a^r \pmod{p}$.

4.1 Generators and Primitive Roots

Fermat's Little Theorem gives an upper bound on the order, but it would be great if we could find a value with actually high order. It turns out we can!

Theorem 7. *For any prime modulo p, there exists an element g such that* $\text{ord}_p(g) = p-1$.

That means, if we look at $1, g, g^2, \dots g^{p-2}$, these $p-1$ elements are necessarily distinct, so they cover all of the non-zero residues modulo p. That means we've defined a nice cyclic structure over $1, \dots, p-1$!

This also means that we can find an element of order d for any $d \mid p-1$: just take $g^{(p-1)/d}$. This is sometimes useful; we often pick primes $p = 2q+1$ where q is prime, and then find an element of order q because prime cycle-length is nice and has less room for vulnerability.

4.2 Implementation

This is where the cool crypto comes from: modular exponentiation is fast. We can take a^{2k} by recursively computing $(a^k)^2$ and $a^{2k+1} = a \cdot (a^k)^2$, so taking the kth power takes only $\log(k)$ multiplications.

However, taking the inverse operation - a "discrete logarithm", seems to be very hard. Given a generator g and a value v, finding k so that $g^k \equiv v$ seems to require essentially brute forcing k in $O(k)$ time. Cryptography relies heavily on this. Yay!

5 Non-prime Modular Arithmetic

Finally, we'll quickly cover some non-prime modular arithmetic. The key theorem here is the Chinese Remainder Theorem.

Theorem 8 (Chinese Remainder Theorem (CRT))**.** *For any two relatively prime modulo m and n, and constants a and b, given that*

$$x = a \pmod{m}$$

$$x = b \pmod{n}$$

there is a unique residue v such that

$$x = v \pmod{mn}$$

In other words, if we know x mod m and x mod n, we can uniquely determine x mod mn. Thus, we can think of mod mn as just a combination of the information of mod m and mod n.

Theorem 9 (Euler's Theorem for Semiprimes)**.** *Given two primes p and q,*

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

Proof. Note that $x^{(p-1)(q-1)} \equiv (x^{p-1})^{q-1} \equiv 1 \pmod{p}$, and likewise modulo q. By the CRT, this uniquely determines the value modulo pq, so it must be 1, as desired. □

This is useful for RSA.

5.1 Implementation

We can find the value from CRT in $O(\log(m))$ multiplications using the extended Euclidean algorithm. In SAGE, this is the method `crt`.