Modular Arithmetic for Beginners - Codeforces

Introduction

If you're new to the world of competitive programming, you may have noticed that some tasks, typically combinatorial and probability tasks, have this funny habit of asking you to calculate a huge number, then tell you that "because this number can be huge, please output it modulo $10^9 + 7$". Like, it's not enough that they ask you to calculate a number they know will overflow basic integer data types, but now you need to apply the modulo operation after that? Even worse are those that say you need to calculate a fraction $\frac{p}{q}$ and ask you to output r where $r \cdot q \equiv p \pmod{m}$... not only do you have to calculate a fraction with huge numbers, how in the *world* are you going to find r?

Actually, the modulo is there to make the calculation *easier*, not *harder*. This may sound counterintuitive, but once you know how modular arithmetic works, you'll see why too. Soon you'll be solving these problems like second nature.

Terminology and notation

For convenience, I will define the notation $n \bmod m$ (for integers n and m) to mean $n - \left\lfloor \frac{n}{m} \right\rfloor \cdot m$, where $\lfloor x \rfloor$ is the largest integer that doesn't exceed x. (This should always produce an integer between 0 and $m - 1$ inclusive.) This may or may not correspond to the expression `n % m` in your programming language (`%` is often called the "modulo operator" but in some instances, it's more correct to call it the "remainder operator"). If `-8 % 7 == 6`, you're fine, but if it is `-1`, you'll need to adjust it by adding m to any negative results. If you use Java/Kotlin, the standard library function `Math.floorMod(n, m)` does what we need.

Also for convenience, I will also define the $\bmod$ operator to have *lower* precedence than addition or subtraction, thus $ax + b \bmod m \Rightarrow (ax + b) \bmod m$. This probably does *not* correspond with the precedence of the `%` operator.

The value m after the modulo operator is known as the *modulus*. The result of the expression $n \bmod m$ is known as n's *residue* modulo m.

You may also sometimes see the notation $expr_1 \equiv expr_2 \pmod{m}$. This is read as "$expr_1$ is congruent to $expr_2$ modulo m", and is

shorthand for $expr_1 \bmod m = expr_2 \bmod m$.

"Basic" arithmetic

First off, some important identities about the modulo operator:

$$(a \bmod m) + (b \bmod m) \bmod m = a + b \bmod m$$

$$(a \bmod m) - (b \bmod m) \bmod m = a - b \bmod m$$

$$(a \bmod m) \cdot (b \bmod m) \bmod m = a \cdot b \bmod m$$

These identities have the very important consequence in that you generally don't need to ever store the "true" values of the large numbers you're working with, only their residues $\bmod m$. You can then add, subtract, and multiply with them as much as you need for your problem, taking the modulo as often as needed to avoid integer overflow. You may even decide to wrap them into their own object class with overloaded operators if your language supports them, though you may have to be careful of any object allocation overhead. If you use Kotlin like I do, consider using the `inline class` feature.

But what about division and fractions? That's slightly more complicated, and requires a concept called the "modular multiplicative inverse". The modular multiplicative inverse of a number a is the number a^{-1} such that $a \cdot a^{-1} \bmod m = 1$. You may notice that this is similar to the concept of a reciprocal, but here we don't want a fraction; we want an integer, specifically an integer between 0 and $m - 1$ inclusive.

But how do you actually find such a number? Bruteforcing all numbers to a prime number close to a billion will usually cause you to exceed the time limit. There are two faster ways to calculate the inverse: the extended GCD algorithm, and Fermat's little theorem. Though the extended GCD algorithm is more versatile and sometimes slightly faster, the Fermat's little theorem method is more popular, simply because it's almost "free" once you implement exponentiation, which is also often a useful operation in itself, so that's what we'll cover here.

Fermat's little theorem says that as long as the modulus m is a prime number ($10^9 + 7$ is prime, and so is 998 244 353, another common modulus in these problems), then $a^m \bmod m = a \bmod m$. Working backwards, $a^{m-1} \bmod m = 1 = a \cdot a^{m-2} \bmod m$, therefore the number we need is $a^{m-2} \bmod m$.

Note that this only works for $a \bmod m \neq 0$, because there is no number x such that $0 \cdot x \bmod m = 1$. In other words, you still can't divide by 0, sorry.

Multiplying $m - 2$ times would still take too long; therefore a trick

known as *exponentiation by squaring* is needed. It's based on the observation that for positive integer n, that if n is odd, $x^n = x(x^2)^{\frac{n-1}{2}}$, while if n is even, $x^n = (x^2)^{\frac{n}{2}}$. It can be implemented recursively by the following pseudocode:

```
function pow_mod(x, n, m):
    if n = 0 then return 1
    t := pow_mod(x, ⌊n/2⌋, m)
    if n is even:
        return t · t  mod m
    else:
        return t · t · x  mod m
```

Or iteratively as follows:

```
function pow_mod(x, n, m):
    y := 1
    while n > 0:
        if n is odd:
            y := y · x  mod m
        n := ⌊n/2⌋
        x := x · x  mod m
    return y
```

Now that you know how to compute the *modular multiplicative inverse* (to refresh, $a^{-1} = a^{m-2} \bmod m$ when m is prime), you can now define the division operator:

$$a \,/\, b \mod m = a \cdot b^{-1} \mod m$$

This also extends the $\bmod$ operator to rational numbers (i.e. fractions), as long as the denominator is coprime to m. (Thus the reason for choosing a fairly large prime; that way puzzle writers can avoid denominators with m as a factor). The four basic operations, as well as exponentiation, will still work on them as usual. Again, you generally never need to store the fractions as their "true" values, only their residues modulo m.

Congratulations! You have now mastered $\mathbb{Z} \,/\, p\,\mathbb{Z}$ field arithmetic! A "field" is just a fancy term from abstract algebra theory for a set with the four basic operators (addition, subtraction, multiplication, division) defined in a way that works just like you've learned in high-school for the rational and real numbers (however division by zero is still undefined), and $\mathbb{Z} \,/\, p\,\mathbb{Z}$ is just a fancy term meaning the set of integers from 0 to $p - 1$ treated as residues modulo p.

This also means that algebra works much like the way you learned in high school. How to solve $3 = 4x + 5 \mod 10^9 + 7$? Simply pretend that x is a real number and get $x = -1 \,/\, 2 \mod 10^9 + 7 = 500000003$. (Technically, all x whose residue is 500000003, including rationals, will satisfy the equation.)

You can also now take advantage of combinatoric identities, like $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. The factorials can be too big to store in their true form, but you can store their modular residues instead, then use modular multiplicative inverse to do the "division".

There are only a few things you need to be careful of, like:

- divisions through modular multiplicative inverse would be slower than the other operations ($O(\log m)$ instead of $O(1)$), so you may want to cache/memoize the inverses you use frequently in your program.
- comparisons (once you represent a number by its modulo residue, comparisons are generally meaningless, as your 1 might "really" be $m+1$, $10^{100}m+1$, $-5m+1$, or even $\frac{1}{m+1}$)
- exponentiation (when evaluating $x^n \bmod m$, you can't store n as $n \bmod m$. If n turns out to be really huge, you need to calculate it modulo $\varphi(m)$ instead, where φ stands for Euler's totient function. If m is prime, $\varphi(m) = m-1$. Note that this new modulus will then usually not be prime, thus "division" in it will not be reliable (you can still use the extended GCD algorithm, but only for numbers coprime to the new modulus), but you can still use the other three operators. In abstract algebra theory, $\mathbb{Z}/n\mathbb{Z}$ is a "ring" rather than a "field" when n isn't prime due to this loss). Do be careful about the special case 0^0, which should typically be defined as 1, while $0^{\varphi(m)}$ would still be 0.

Puzzles

Here are some simpler puzzles that require a modulo answer:

1281C - Вырезание и вставка

1279D - Робот Деда Мороза

1178C - Плитки

1248C - Иванушка-дурачок и теория вероятностей

935D - Фафа и древний алфавит

300C - Прекрасные числа