Modular Arithmetic

Modulus Operator

Basics of Modulus Operator

The `a % b` gives remainder when a is divided by b.

If `a % b = c`, then it can also be written as $a \equiv c\ mod(b)$, which says a will give remainder of c when divided by b. Eg: `7 % 3 = 1` can also be written as $7 \equiv 1\ mod(3)$

The Behavior of Modulus with Negative Integers

The expression `-4 % 3` gives different results in different languages:

- In python, it gives output as 2
- In C++, it gives output as -1

Ideally, if we look according to periodicity, then it should have been 2 only as we always consider range of expression `a % b` from 0 to (b-1).

But C++ treats modulus in a different way, i.e. C++ treats `-a % b` as `-(a % b)`
C++ is not so good at handling negative things, it treats even `((-a) % (-b))` as `-(a % b)`
Thatswhy, we don't deal much with negative numbers in modulo part.

Modulo Arithmetic

The need of Modulo Arithmetic

Since, too large numbers like $100!$ can't be stored, thatswhy people move into a realm of remainder.

Thatswhy, sometimes different coding platforms ask you to return `ans % p` instead of directly returning ans, where p is any random number. They do so to shorten the range of answer in $[0, p-1]$.

Example :
A set S contains 'n' elements. If A and B are subsets of set S, then find number of different combinations of A & B generated such that $A \cap B = \emptyset$.
Since, each element in Set S has 3 options: either go to A, or to B, or no where. So, the total number of different combinations of A & B would be 3^n. But the problem is 3^n becomes large very very fast, so instead of printing 3^n you'll be asked to print $3^n \% (10^9 + 7)$.

Significance of using $10^9 + 7$ as Modulus

1. It's a Prime number
   On using prime numbers as modulus, you get more diversified values as possible outputs.
   To majorly avoid hard-coded solutions getting accepted.
   So that Inverses would always exist.
2. It's the first prime after 10^9

3. Because on choosing $10^9 + 7$ as modulus, sum of 2 remainders just fits in `int` range.
   r1 & r2 both could range from $[0, 10^9 + 6]$
   So, their sum would range from $[0, 2 \times 10^9 + 12]$ and the last value `int` can store is 2.1×10^9

Rules of Modulo Arithmetic

To ensure that intermediate values remains small, Modulo Arithmetic follow some set of rules :

1. $(a + b) \mod c \equiv ((a \mod c) + (b \mod c)) \mod c$
2. $(a \times b) \mod c \equiv ((a \mod c) \times (b \mod c)) \mod c$
3. $(a - b) \mod c \equiv ((a \mod c) - (b \mod c)) \mod c$
4. $(k \times a) \mod c \equiv (k \times (a \mod c)) \mod c$, for any integer k
5. $a^b \mod c \equiv (a \mod c)^b \mod c$
6. $((a \times b) \times c) \mod p \equiv (((a \times b) \mod p) \times c) \mod p$

Note : If you take modulus of 2 numbers, then both of their remainders would be around at maximum of 10^9, and if you multiply both of them then the result would be around 10^{18} which will not fit in the integer range. So whenever something around multiplication comes, you have to use `long long` datatype.

Example - 1

It is currently 7:00 PM. What time (in AM or PM) will it be in 1000 hours ?

Time "repeats" every 24 hours, so we work with modulo 24

$$1000 \equiv 16 + (24 \times 41) \equiv 16 \pmod{24}$$

Since the time in 1000 hours is equivalent to time in 16 hours
So 1000 hours forward from 7:00 PM is like 16 hours forward from 7:00 PM
Therefore, it'll be 11:00 AM in 1000 hours.

Example - 2

Calculate the value of expression $(a + b - c \times e^d + f) \mod (10^9 + 7)$
It's given that all values of all variables is less than 10^9

Add brackets, $((a + (b - (c \times (e^d)))) + f) \mod (10^9 + 7)$

If the intermediate values are negative, then that's not wrong, it's just that we should've printed their positive counter part.
For example, let's consider expression

$$(3 \times -4) \mod 5$$

If it stayed as it is,

$$(3 \times -4) \mod 5 \implies -12 \mod 5 = -2$$

If we've maded it correct,

$$(3 \times -4) \mod 5 \implies (3 \times 1) \mod 5 = 3$$

They're still both are the same thing, you just not have corrected -2 to be 3 in realm of (mod 5)

-2 and 3 are same thing in realm of (mod 5). So, it's just about correcting the final value, if you let the intermediate value be negative and still not overflow, then that's fine.

So, to get the positive counter part, while printing do $(ans \mod m + m) \mod m$

```
#include<bits/stdc++.h>
using namespace std;

#define int long long
int mod = 1e9 + 7;

signed main() {
        int a, b, c, d, e, f;
        cin >> a >> b >> c >> d >> e >> f;

        int ans = 1;
        for(int i=0; i<d; i++) {
                ans = (ans * e) % mod;
        }

        ans = (c * ans) % mod;
        ans = (b - ans) % mod; // ans may become negative here
        ans = (a + ans) % mod;
        ans = (ans + f) % mod;

        cout << (ans%mod + mod) % mod << '\n';

        return 0;
}
```

Binary Exponentiation

Binary Exponentiation

It's a Algorithm that calculates a^b in $O(\log b)$ time complexity.

Intution :

$$a^b = \begin{cases} a^{b-1} \times a & \text{if } b \text{ is odd} \\ a^{b/2} \times a^{b/2} & \text{if } b \text{ is even} \end{cases}$$

It's like mostly at every step we're doubling the power, infact in every 2 steps it will get half for sure
In 2 steps you're getting half, so in $2 \times \log n$ steps you will get to 1
Hence, it's time complexity is $O(\log n)$

Code :

```
#include<bits/stdc++.h>
using namespace std;

#define int long long
int mod = 1e9 + 7;
int binpow(int a, int b) {
        // Base case
        if(b == 0) return 1;

        if(b % 2 == 1) return (a * binpow(a, b-1)) % mod;
        else {
                int x = binpow(a, b/2);
                return (x * x) % mod;
        }
}

signed main() {
        int a, b;
        cin >> a >> b;

        int ans = binpow(a, b);
        cout << (ans%mod + mod) % mod << '\n';

        return 0;
}
```

Note :
If there's modulo, we add modulo
Else, we don't add modulo

Inverses

Need of Inverses

Unlike addition, subtraction & multiplication, in case of division modulus($\%$) can't be broken down

$$(\frac{a}{b}) \mod c \neq \frac{(a \mod c)}{(b \mod c)}$$

That's where the concept of Inverses came into picture

Inverses

Let's assume value of above expression to be 'x'

$$x \equiv (\frac{a}{b}) \pmod c$$

On multiplying both sides with 'b', maybe numbers won't match but their remainders will

$$(b \times x) \mod c \equiv a \mod c$$

On putting a = 1, we get

$$x \equiv \frac{1}{b} \pmod c$$

From above, we can say that there might be a possible value (i.e. x) which signifies $\frac{1}{b}$ such that

$$(b \times x) \mod c \equiv 1 \mod c$$

If we are able to find an x that satisfies above condition, then we will call that x as **Inverse of b**
This x is denoted as b^{-1}

Example
Let's understand this with a numeric example,

$$b \cdot b^{-1} \equiv 1 \pmod 7$$

So,

b	1	2	3	4	5	6
b^{-1}	1	4	5	2	3	6

Now,

$$(\frac{4}{2}) \mod 7 = 2$$

$$(\frac{4}{2}) \mod 7 \equiv (4 \cdot 2^{-1}) \mod 7$$

$$(4 \cdot 2^{-1}) \mod 7 = (4 \cdot 4) \mod 7 = 16 \mod 7 = 2$$

From above, inverse results in same answer as original

So, when we're doing mod with certain fraction & if we're able to find inverse of number in denominator, then at time of dividing, instead of dividing we will multiply with the number's inverse

Does Inverse always exists ?

Let's assume $b \cdot b^{-1} \equiv 1 \mod 6$, then

b	1	2	3	4	5
b^{-1}	1	DNE	DNE	DNE	5

It's not necessary that inverse will always exist

Condition for Inverse existence

For existence of inverse, the values of b and m should be co-prime

Note : This is one of the reasons that why the modulo's given in problems are always a prime number (It's because prime number will always be co-prime with any other number). Thatswhy modulo's given in the problems are prime numbers, so that Inverses always exists.

Fermat's Little Theorem

If p is a prime number and a is an integer not divisible by p, then:

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

In other words, a^{p-1} leaves a remainder of 1 when divided by p.

Note : Prime modulo is only needed for existence of inverses that are required in division operations, but for any other operations (i.e. multiplication, addition & subtraction) modulo need not be prime number.

Program for finding Inverse

Approach :
From Fermat's Little Theorem,

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

From Inverses,

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

By direct comparison between above equations, we get

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

Use Binary Exponentiation to find a^{p-2}

Example :
If we want 2^{-1} in $\mod 7$, then

$$2^{-1} = 2^{7-2} \mod 7 = 4$$

Code :

```
#include<bits/stdc++.h>
using namespace std;

#define int long long
int mod = 1e9 + 7;

int binpow(int a, int b) {
        // Base case
        if(b == 0) return 1;

        if(b % 2 == 1) return (a * binpow(a, b-1)) % mod;
        else {
                int x = binpow(a, b/2);
                return (x * x) % mod;
        }
}

int inverse(int x) {
        return binpow(x, mod-2);
}
```

Rules of Inverses

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

How to write Safe Code ?

Rules to write Safe code

1. Always use `long long` datatype
   Trick to convert all `int` to `long long` :
   > - Convert `int main()` to `signed main()`
   > - Use Macro, `#define int long long`
2. Ensure that all expressions are enclosed in brackets explicitly
3. Use only 1 operator at a time
   So that it never overflows in intermediate calculations
4. Instead of printing `x`, print `((x % m) + m) % m`
   To print the positive counter part in case of negative intermediate values
   Let's say before printing,

$$\begin{gathered} x \rightarrow (-\infty, \infty) \\ x \mod 4 \rightarrow [-3, 3] \\ ((x \mod 4) + 4) \rightarrow [1, 7] \\ ((x \mod 4) + 4) \mod 4 \rightarrow [0, 3] \end{gathered}$$

   All we did was adding the number, and taking the remainder again
   So, that can never change the answer because when you're taking remainder with x and if you add x, then that doesn't change the answer
   So, if the module had been m, then the final range would have been $[0, m-1]$
5. Always use Binary Exponentiation or Loops, never use the pow() function of STL
   Because :
   > - pow() function calculates in floating-point
   >   So, 5^2 can come out to be 25.000001 or 24.9999999 also
   >   Typecasting 24.9999999 to integer will lead to 24, which is wrong
   > - How will you take remainder of floating-point numbers ?
6. Never perform division, always find Inverses

$$\frac{a}{b} \rightarrow a \times b^{-1}$$