**MODMUL - Modular Multiplication -** [**https://www.spoj.com/problems/MODMUL/**](https://www.spoj.com/problems/MODMUL/)

**🧠 Problem Understanding**

You're given two **non-negative 64-bit integers**, a and b. You need to:

* Multiply them: a \* b
* Take the result modulo 10000007: (a \* b) % 10000007
* Print the result with a case number.

**🧩 How to Think About It**

This is a classic **modular arithmetic** problem. The key idea is:

When numbers are very large, direct multiplication might overflow. But since we only care about the result modulo a small number (like 10000007), we can use modular properties to keep numbers manageable.

However, in C++, unsigned long long can handle up to 64-bit values, so direct multiplication is safe here.

**🛠️ Solution Approach**

1. Read input until EOF (end of file).
2. For each pair (a, b):
   * Compute (a \* b) % 10000007
   * Print the result in the format: Case #i: result

**💻 C++ Implementation**

#include <iostream>

using namespace std;

const long long MOD = 10000007;

int main() {

long long a, b;

int caseNum = 1;

while (cin >> a >> b) {

long long result = (a % MOD) \* (b % MOD) % MOD;

cout << "Case #" << caseNum++ << ": " << result << endl;

}

return 0;

}

**⏱️ Time & Space Complexity**

* **Time Complexity**: O(1) per test case (just a few arithmetic operations)
* **Space Complexity**: O(1) (no extra space used)

Another Approach :

#include <bits/stdc++.h>

using namespace std;

const long long MOD = 10000007;

// Function to do (a \* b) % mod safely without overflow

long long modMul(long long a, long long b, long long mod) {

    long long res = 0;

    a %= mod;

    while (b > 0) {

        if (b & 1) { // if b is odd

            res = (res + a) % mod;

        }

        a = (a \* 2) % mod;

        b >>= 1; // divide b by 2

    }

    return res;

}

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(NULL);

    long long a, b;

    int caseNo = 1;

    while (cin >> a >> b) {

        long long ans = modMul(a, b, MOD);

        cout << "Case #" << caseNo++ << ": " << ans << "\n";

    }

    return 0;

}

**Fibonacci Under Modulo -** [**https://vjudge.net/problem/Gym-248968Y**](https://vjudge.net/problem/Gym-248968Y)

**1. Problem Statement in Simple Words**

We are given:

* A number n (1 ≤ n ≤ 10⁵).
* Fibonacci sequence is defined as:
* f0 = 1
* f1 = 1
* f(i) = f(i-1) + f(i-2) for i ≥ 2
* We need to find the **n-th** Fibonacci number (using **0-based indexing**) but since it can be **huge**, we take the result **mod 998,244,353**.

**Example:**

n = 5

Sequence: 1, 1, 2, 3, 5, 8, ...

f5 = 8 (but in the example they use 1-based indexing, so be careful)

**2. Clarifying Indexing**

From the problem:

* It says f0 = f1 = 1.
* So:
* f0 = 1
* f1 = 1
* f2 = 2
* f3 = 3
* f4 = 5
* f5 = 8
* If **n = 5** (1-based in statement), they output 5. That means they are **using 1-based indexing** in the examples.

**3. How to Think About It**

We need to compute **nth Fibonacci mod 998244353** efficiently.  
Constraints:

* n ≤ 10⁵ → **O(n)** iterative solution is fine.
* No need for matrix exponentiation unless n is extremely large (like 10¹⁸).

**4. Approach**

**Iterative DP**

* Create an array fib of size n+1.
* Initialize:
* fib[0] = 1
* fib[1] = 1
* Loop from i = 2 to n:
* fib[i] = (fib[i-1] + fib[i-2]) % MOD
* Output fib[n].

**5. C++ Solution**

#include <bits/stdc++.h>

using namespace std;

const int MOD = 998244353;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int n;

cin >> n;

vector<int> fib(n + 2, 0);

fib[0] = 1;

fib[1] = 1;

for (int i = 2; i <= n; i++) {

fib[i] = (fib[i - 1] + fib[i - 2]) % MOD;

}

cout << fib[n] % MOD << "\n";

return 0;

}

**6. Complexity Analysis**

* **Time Complexity:** O(n) → we compute each Fibonacci number once.
* **Space Complexity:** O(n) → to store the sequence (can be reduced to O(1) if we store only the last two values).

**MODEX -** [**https://vjudge.net/problem/UVA-1230**](https://vjudge.net/problem/UVA-1230)

**🧠 Problem Understanding**

You're given multiple datasets. Each dataset contains three integers:

* x: the base
* y: the exponent
* n: the modulus

You need to compute:  
[ z = x^y \mod n ]

But here’s the catch:

* x and n are small (less than 32768)
* y is **very large** (up to 2 billion!)  
  So you **cannot** compute x^y directly — it would overflow and be extremely slow.

**🧩 How to Think About It**

This is a classic **modular exponentiation** problem.  
We use **Exponentiation by Squaring**, which reduces the time complexity from O(y) to O(log y).

**Key Idea:**

Instead of computing ( x^y ) directly, we:

* Square the base and halve the exponent recursively
* Apply modulo at each step to keep numbers small

**🛠️ C++ Implementation**

#include <iostream>

using namespace std;

int modExpo(int x, int y, int n) {

long long result = 1;

long long base = x % n;

while (y > 0) {

if (y % 2 == 1) {

result = (result \* base) % n;

}

base = (base \* base) % n;

y /= 2;

}

return result;

}

int main() {

int c;

cin >> c;

while (c--) {

int x, y, n;

cin >> x >> y >> n;

cout << modExpo(x, y, n) << endl;

}

int endSignal;

cin >> endSignal; // Read the final '0' line

return 0;

}

**⏱️ Time & Space Complexity**

* **Time Complexity**: O(log y) — efficient even for y ≈ 2×10⁹
* **Space Complexity**: O(1)

**✅ Sample Walkthrough**

**Input:**

2

2 3 5

2 2147483647 13

0

**Output:**

3 → because 2^3 = 8, and 8 % 5 = 3

11 → 2^2147483647 is huge, but mod 13 gives 11

**understand why fast modular exponentiation works in O(log y)** and see a **step-by-step test case example**. 🚀

**🔹 Problem Recap**

We need to compute:

z=xy mod nz = x^y \bmod n

where

* x and n are up to 2152^{15} (32,768),
* y is huge (up to 2312^{31} ≈ 2 billion).

**🔹 Why Naive Approach is Slow**

The **naive method** would multiply x with itself y times:

xy mod nx^y \bmod n

That’s **O(y)** multiplications.  
If y = 2,147,483,647 (≈ 2 billion), that many multiplications are **impossible** in time.

**🔹 Idea of Fast Modular Exponentiation**

Instead of multiplying x y times, we use **binary representation of y**:

Every number can be expressed as powers of 2.  
Example:

13=(1101)2=23+22+2013 = (1101)\_2 = 2^3 + 2^2 + 2^0

So:

x13=x(8+4+1)=x8⋅x4⋅x1x^{13} = x^{(8 + 4 + 1)} = x^8 \cdot x^4 \cdot x^1

This means we **only need log₂(y) steps** instead of y steps!

**🔹 Algorithm Steps**

1. Initialize result = 1.
2. While y > 0:
   * If y is odd → multiply current base into result.
   * Square the base (x = x \* x % n).
   * Divide y by 2 (shift right).

Each step halves y, so total steps = **O(log₂ y)**.

**🔹 Example Walkthrough**

Let’s test:

x=2,y=13,n=5x = 2, \quad y = 13, \quad n = 5

We want:

213 mod 52^{13} \bmod 5

**Step 1: Convert y (13) to binary**

13=(1101)2=8+4+113 = (1101)\_2 = 8 + 4 + 1

So:

213=28⋅24⋅212^{13} = 2^8 \cdot 2^4 \cdot 2^1

**Step 2: Algorithm Execution**

* result = 1, base = 2, y = 13

| **Iteration** | **y (binary)** | **Odd?** | **result** | **base → base² % 5** | **y → y/2** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1101 (13) | ✅ odd | (1\*2) % 5 = 2 | 2² % 5 = 4 | 6 |
| 2 | 110 (6) | ❌ even | 2 | 4² % 5 = 1 | 3 |
| 3 | 11 (3) | ✅ odd | (2\*1) % 5 = 2 | 1² % 5 = 1 | 1 |
| 4 | 1 (1) | ✅ odd | (2\*1) % 5 = 2 | 1² % 5 = 1 | 0 |

Final result = **3** ✅

**🔹 Complexity Analysis**

* Each iteration halves y.
* So total iterations = log⁡2(y)\log\_2(y).
* Each step has **O(1)** multiplications/mods.

👉 **Time Complexity = O(log y)**  
👉 **Space Complexity = O(1)**

✅ That’s why even for y = 2,147,483,647, the loop runs at most ~31 steps (since log₂(2 billion) ≈ 31). 🚀

**Modular division -** [**https://vjudge.net/problem/EOlymp-9606**](https://vjudge.net/problem/EOlymp-9606)

**🔹 Problem Statement (Rephrased)**

We are given three integers:

* aa, bb, and a prime modulus nn.

We need to compute:

ab   mod n\frac{a}{b} \; \bmod n

This is the modular division problem: finding xx such that

(b⋅x)≡a(modn)(b \cdot x) \equiv a \pmod{n}

**🔹 Key Idea**

Normally, division in modular arithmetic doesn’t work like real division. But **when the modulus nn is prime**, we can use **modular inverse**.

* The modular inverse of bb modulo nn is a number b−1b^{-1} such that:

b⋅b−1≡1(modn)b \cdot b^{-1} \equiv 1 \pmod{n}

* Then:

ab(modn)=(a⋅b−1)(modn)\frac{a}{b} \pmod{n} = (a \cdot b^{-1}) \pmod{n}

**🔹 How to Compute Modular Inverse?**

Since nn is prime, we can use **Fermat’s Little Theorem**:

bn−1≡1(modn)⇒bn−2≡b−1(modn)b^{n-1} \equiv 1 \pmod{n} \quad \Rightarrow \quad b^{n-2} \equiv b^{-1} \pmod{n}

So,

a/b mod n=(a⋅bn−2) mod na / b \bmod n = (a \cdot b^{n-2}) \bmod n

**🔹 Solution Approach**

1. **Read input**: integers a,b,na, b, n.
2. **Compute inverse of b**:

b−1=bn−2 mod nb^{-1} = b^{n-2} \bmod n

using fast exponentiation (binary exponentiation).

1. **Compute result**:

result=(a⋅b−1) mod nresult = (a \cdot b^{-1}) \bmod n

1. Print the result.

**🔹 Example Walkthrough**

**Example 1:**

Input: 3 4 7

* a=3,b=4,n=7a = 3, b = 4, n = 7
* Find b−1=47−2=45 mod 7b^{-1} = 4^{7-2} = 4^5 \bmod 7
  + 42=16≡24^2 = 16 \equiv 2
  + 44=22=44^4 = 2^2 = 4
  + 45=4⋅4=16≡2 mod 74^5 = 4 \cdot 4 = 16 \equiv 2 \bmod 7
* So, b−1=2b^{-1} = 2.
* Result = 3⋅2 mod 7=63 \cdot 2 \bmod 7 = 6. ✅

Output: 6

**Example 2:**

Input: 4 8 13

* a=4,b=8,n=13a = 4, b = 8, n = 13
* Find b−1=813−2=811 mod 13b^{-1} = 8^{13-2} = 8^{11} \bmod 13
  + Fast exponentiation → b−1=5b^{-1} = 5.
* Result = 4⋅5 mod 13=20 mod 13=74 \cdot 5 \bmod 13 = 20 \bmod 13 = 7. ✅

Output: 7

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

using ll = long long;

// Fast exponentiation (a^b mod m)

ll mod\_pow(ll base, ll exp, ll mod) {

ll result = 1;

base %= mod;

while (exp > 0) {

if (exp & 1) result = (result \* base) % mod;

base = (base \* base) % mod;

exp >>= 1;

}

return result;

}

int main() {

ll a, b, n;

cin >> a >> b >> n;

// Compute modular inverse of b

ll b\_inv = mod\_pow(b, n - 2, n);

// Compute (a \* b^-1) % n

ll ans = (a \* b\_inv) % n;

cout << ans << "\n";

return 0;

}

**🔹 Complexity Analysis**

* **Fast exponentiation**: O(log⁡n)O(\log n)
* Multiplication and modulo operations are O(1)O(1).
* So total **time complexity**:

O(log⁡n)O(\log n)

* **Space complexity**: O(1)O(1)