**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