**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 - 1] % 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:

^{13} = x^{(8 + 4 + 1)} = x^8 . x^4 . 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:

* a, b, and a prime modulus n.

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 n is prime**, we can use **modular inverse**.

* The modular inverse of b modulo n is a number b^{-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 = 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 = 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)

1. **Modular Exponentiation -** [**https://codeforces.com/problemset/problem/913/A**](https://codeforces.com/problemset/problem/913/A)

**🔹 Problem Statement (Rephrased)**

Normally, in modular exponentiation problems, we compute:

2n mod m2^n \bmod m

But here, the task is **the reverse problem**. Instead of computing 2n mod m2^n \bmod m, we are given two numbers nn and mm, and we must compute:

m mod 2nm \bmod 2^n

**🔹 How to Think**

* Recall that x mod yx \bmod y = the **remainder** when xx is divided by yy.
* So we simply need:

Answer=m%(2n)\text{Answer} = m \% (2^n)

* Example 1:  
  n=4,m=42n = 4, m = 42  
  2n=24=162^n = 2^4 = 16  
  42%16=1042 \% 16 = 10 ✅
* Example 2:  
  n=1,m=58n = 1, m = 58  
  21=22^1 = 2  
  58%2=058 \% 2 = 0 ✅
* Example 3:  
  n=98765432,m=23456789n = 98765432, m = 23456789  
  2n2^n is **enormous**, but notice:
  + If 2n>m2^n > m, then m%2n=mm \% 2^n = m.
  + Here, 2987654322^{98765432} is way larger than 2345678923456789, so answer = mm. ✅

**🔹 Key Observations**

1. Directly computing 2n2^n for large nn is impossible (too big).
2. But since we only need m mod 2nm \bmod 2^n:
   * If n≥31n \geq 31, then 2n≥231>109≥m2^n \geq 2^{31} > 10^9 \geq m.
   * Therefore, for all **large nn** (n≥31n \geq 31), answer = mm.

So we only need to compute 2n2^n explicitly when n<31n < 31.

**🔹 Solution Approach**

1. Read n,mn, m.
2. If n≥31n \geq 31, print mm.
   * Because 2n2^n will exceed mm.
3. Otherwise, compute 2n2^n safely (using 1LL << n) and print m % (1 << n).

**🔹 Example Walkthrough**

**Example 1**

Input:

4

42

* n=4n=4, so 24=162^4 = 16.
* 42%16=1042 \% 16 = 10.  
  Output: 10

**Example 2**

Input:

1

58

* n=1n=1, so 21=22^1 = 2.
* 58%2=058 \% 2 = 0.  
  Output: 0

**Example 3**

Input:

98765432

23456789

* n=98765432≥31n=98765432 \geq 31.
* Answer = m=23456789m = 23456789.  
  Output: 23456789

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

int main() {

long long n, m;

cin >> n >> m;

if (n >= 31) {

cout << m << "\n"; // 2^n > m, so remainder = m

} else {

long long modVal = 1LL << n; // compute 2^n

cout << m % modVal << "\n";

}

return 0;

}

**🔹 Complexity Analysis**

* **Time Complexity**: O(1)O(1)
  + Just one modulo operation.
* **Space Complexity**: O(1)O(1)

✅ Simple but elegant problem: reduce it to modulo with power of 2, and optimize by noticing that 2n2^n quickly surpasses mm.

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

**🔹 Problem Statement (Rephrased)**

We are given a single integer n (1 ≤ n ≤ 100000).  
We need to calculate **n! (n factorial)**:

n!=1×2×3×...×n

But since factorial grows very fast, we are asked to give the result **modulo 998244353** (a large prime number).

Formally:

Output=(n!) mod 998244353

**🔹 Key Observations**

1. Factorials grow **extremely large**:
   * 20! ≈ 2.4×10^18 (already too big for 64-bit integer).
   * 100000! has **~456,574 digits**!

So we **must** use modular arithmetic.

1. The modulo given (998244353) is a **special prime** often used in competitive programming (supports fast modular operations).
2. Since n can go up to 10^5, we can compute the factorial iteratively, applying the modulo at each step to prevent overflow.

**🔹 Thinking Process**

We want:

n!mod  998244353

That means:

* Multiply numbers from 1 to n.
* After each multiplication, take modulo 998244353.

Example for n=5:

5!=1⋅2⋅3⋅4⋅5=120

120mod  998244353=120

For n=45, the result becomes large, but since we apply modulo at every step, it remains manageable.

**🔹 Solution Approach**

1. Initialize result = 1.
2. Loop i from 1 to n.
3. Update:

result=(result×i)mod  998244353

1. After loop ends, print result.

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

const int MOD = 998244353;

int main() {

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

cin.tie(0);

long long n;

cin >> n;

long long result = 1;

for (long long i = 1; i <= n; i++) {

result = (result \* i) % MOD;

}

cout << result << "\n";

return 0;

}

**🔹 Complexity Analysis**

* **Time Complexity**:  
  O(n) → We loop from 1 to n once. With n ≤ 100000, it’s efficient.
* **Space Complexity**:  
  O(1) → Only uses constant extra memory.

**✅ Example Walkthrough**

Input: 45

* Iteratively multiply and apply modulo.
* Final result: 991610752.

Input: 2399

* Final result: 201967083.

**Big Mod UVA – 374 -** [**https://vjudge.net/problem/UVA-374**](https://vjudge.net/problem/UVA-374)

**🔹 Problem Statement (Rephrased)**

We need to compute:

R = B^P mod M

for multiple test cases, where:

* B (base), P (power) are integers in range [0, 2^{31}-1].
* M (modulus) is an integer in range [1,46340]

Input format:

* Each test case consists of 3 integers:
* B
* P
* M
* Several test cases are given (until end of input).

Output:

* For each test case, print one line with the result.

**🔹 How to Think About It**

1. **Naïve way**:  
   Just compute BPB^P then take modulo MM.  
   ❌ Impossible, because B^P becomes astronomically large when P is big (e.g., 3^{18132} has thousands of digits).
2. **Key Insight (Modulo Property)**:

(a⋅b) mod m=((a mod m)⋅(b mod m)) mod m

→ This allows us to keep numbers small throughout computation.

1. **Efficient Approach (Binary Exponentiation / Fast Power)**:  
   Instead of multiplying BB PP times (O(P)), we use the **divide & conquer trick**:
   * If P is even:

B^P mod M=(B^P/2 mod M)^2 mod M

* + If P is odd:

B^P mod M=(B mod M)⋅(B^(P−1) mod M) mod M

1. This reduces complexity to **O(log P)**.

**🔹 Solution Approach**

1. Read input until EOF (multiple test cases).
2. For each test case:
   * Reduce base B % M.
   * Use **binary exponentiation** to compute (B^P) % M efficiently.
   * Print the result.

**🔹 C++ Solution**

#include <iostream>

using namespace std;

// Fast Modular Exponentiation (Binary Exponentiation)

long long modExpo(long long b, long long p, long long m)

{

long long result = 1 % m; // safe if m=1

long long base = b % m;

while (p > 0)

{

if (p % 2 == 1) // if p is odd

{

result = (result \* base) % m;

}

base = (base \* base) % m; // square the base

p /= 2; // divide exponent by 2

}

return result;

}

int main()

{

long long b, p, m;

while (cin >> b >> p >> m) // read until EOF

{

cout << modExpo(b, p, m) << "\n";

}

return 0;

}

**🔹 Complexity Analysis**

* **Time Complexity**:  
  O(log P) per test case (very fast, even for P=2.1×10^9) .
* **Space Complexity**:  
  O(1), only uses a few variables.

**🔹 Example Walkthrough**

Input:

3

18132

17

17

1765

3

2374859

3029382

36123

Step by step:

1. 318132 mod 17=133^{18132} \bmod 17 = 13
2. 171765 mod 3=217^{1765} \bmod 3 = 2
3. 23748593029382 mod 36123=131952374859^{3029382} \bmod 36123 = 13195

Output:

13

2

13195

✅ Matches sample output.

**FACTMUL - Product of factorials -** [**https://vjudge.net/problem/SPOJ-FACTMUL**](https://vjudge.net/problem/SPOJ-FACTMUL)

**🔹 Problem Statement**

You are asked to compute the product of the first n factorials:

P(n)=1!×2!×3!×⋯×n!P(n) = 1! \times 2! \times 3! \times \dots \times n!

and return the result **modulo**

M=109546051211M = 109546051211

where

* Input: a single integer n (1 ≤ n ≤ 10^7).
* Output: P(n)mod  MP(n) \mod M.

Example:

* If n = 5

1!×2!×3!×4!×5!=1×2×6×24×120=345601! \times 2! \times 3! \times 4! \times 5! = 1 \times 2 \times 6 \times 24 \times 120 = 34560

Output = 34560.

**🔹 How to Think**

1. **Naive formula**  
   Directly compute all factorials, multiply them, and take modulo.  
   Problem: factorials grow *extremely fast* → overflow and inefficiency if not reduced by MOD.
2. **Observation**
   * i!=(i−1)!×ii! = (i-1)! \times i
   * So factorials can be computed incrementally.
   * Keep fact = i! mod M and update at each step.

Then:

P(n)=∏i=1n(i!)=(1!)×(2!)×(3!)…(n!)P(n) = \prod\_{i=1}^n (i!) = (1!) \times (2!) \times (3!) \dots (n!)

At each step:

* + Update fact = fact \* i % M
  + Update ans = ans \* fact % M

1. **Modulo Concern**
   * MOD = 109546051211 fits within 64-bit, but intermediate multiplication of two 64-bit numbers can exceed 64-bit range.
   * So we must use **128-bit integer type** (unsigned \_\_int128) for safe intermediate products.
2. **Early Exit Optimization**
   * If at some step fact % M == 0, then all following factorials are also 0 mod M.
   * So ans will remain 0, and we can **break early**.

**🔹 Solution Approach**

* Use a loop from 1 to n.
* Maintain two variables:
  + fact = i! mod M
  + ans = product of all factorials mod M
* Use \_\_int128 for intermediate multiplication.
* Return ans.

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

int main() {

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

cin.tie(nullptr);

const unsigned long long MOD = 109546051211ULL; // given modulo

int n;

if (!(cin >> n)) return 0;

unsigned long long fact = 1, ans = 1;

for (int i = 1; i <= n; ++i) {

// compute i! modulo MOD

fact = (unsigned \_\_int128)fact \* i % MOD;

// multiply into the product

ans = (unsigned \_\_int128)ans \* fact % MOD;

// early stop if factorial becomes 0 modulo MOD

if (fact == 0) break;

}

cout << ans << '\n';

return 0;

}

**🔹 Complexity Analysis**

* **Time Complexity:**
  + We do a single loop up to n.
  + Each iteration is O(1).
  + Total: O(n)
  + For n ≤ 10^7, this is feasible (≈ 10 million iterations).
* **Space Complexity:**
  + Only a few variables are used (fact, ans).
  + Total: O(1)

✅ This approach efficiently computes the answer, avoids overflow, and passes online judge constraints.

**LASTDIG - The last digit -** [**https://vjudge.net/problem/SPOJ-LASTDIG**](https://vjudge.net/problem/SPOJ-LASTDIG)

**🔹 Problem Restatement**

Nestor needs to find the **last digit** of aba^b (a raised to the power b).

* aa is the **base** (0 ≤ a ≤ 20)
* bb is the **exponent** (0 ≤ b ≤ 2,147,483,000)
* Both aa and bb are **not 0 at the same time**.
* There can be up to 30 test cases.

**Example:**

* Input: 3 10 → 310=590493^{10} = 59049, last digit = **9**
* Input: 6 2 → 62=366^2 = 36, last digit = **6**

**🔹 How to Think**

1. **Direct computation is impossible**:  
   For b=2,147,483,000b = 2,147,483,000, computing aba^b will overflow and is way too slow.
2. **Last digit repeats in cycles**:
   * Example: powers of 2  
     21=22^1 = 2 → last digit 2  
     22=42^2 = 4 → last digit 4  
     23=82^3 = 8 → last digit 8  
     24=162^4 = 16 → last digit 6  
     25=322^5 = 32 → last digit 2 → cycle repeats

So last digit cycle for 2 is [2, 4, 8, 6].

1. **Key observation**:  
   The last digit always **repeats in a cycle of at most length 4**.  
   So instead of computing full power, just:
   * Find last digit cycle of a
   * Use (b−1)%cycle length(b-1) \% \text{cycle length} to pick the correct element.

**🔹 Solution Approach**

1. Read number of test cases tt.
2. For each test case:
   * Read a,ba, b.
   * Special case: if b==0b == 0, return 1 (because a0=1a^0 = 1).
   * Compute the cycle of last digits for base a.
   * Use modulo to find the correct last digit.
   * Print result.

**🔹 C++ Solution (Your Code)**

#include <bits/stdc++.h>

using namespace std;

int main() {

int t; cin >> t;

while(t--) {

long long a, b;

cin >> a >> b;

if(b == 0) { // special case

cout << 1 << "\n";

continue;

}

int cycle[4], len = 0;

int last = a % 10; // only last digit matters

cycle[len++] = last;

// build the cycle of last digits

for(int i = 1; i < 4; i++) {

int d = (cycle[i-1] \* last) % 10;

if(d == cycle[0]) break; // cycle repeats

cycle[len++] = d;

}

cout << cycle[(b-1) % len] << "\n"; // pick correct index

}

}

**🔹 Example Walkthrough**

Input:

2

3 10

6 2

* Case 1: 3103^{10}
  + Cycle: [3, 9, 7, 1] (length 4)
  + Index: (10−1)(10-1) % 4 = 9 % 4 = 1 → cycle[1] = **9**
* Case 2: 626^2
  + Cycle: [6] (length 1, because it repeats immediately)
  + Index: (2−1)(2-1) % 1 = 0 → cycle[0] = **6**

Output:

9

6

✅ Matches expected.

**🔹 Complexity Analysis**

* Building the cycle: **O(1)** (max 4 multiplications).
* Answering each test case: **O(1)**.
* Overall: **O(t)** (at most 30).
* Memory: **O(1)**.

This is **super efficient** and works even for the largest bb.