

Level 1 Training

Session 6

Number Theory Part 2

Modular arithmetic, binary power,
modular inverse, GCD, and LCM

Keep the congruence. Prove the shortcut.
Check that division is allowed.

How to use this reference

This is a reference to study **after the session**. It explains why the mathematics works, what each variable means, and where a short piece of code stops being safe. Learn one argument at a time; a memorized formula without its conditions is not yet a reliable tool.

A study routine for this final session

1. **Keep the original question visible.** Is the output an exact integer, a remainder, an inverse, or a solution of an equation? These are different requests.
2. **Trace a small instance.** Use a modulus such as 7 or 12, a short Euclidean chain, or an exponent such as 13. Write the state before each update.
3. **State the condition before the formula.** A modulus may be composite. An inverse may not exist. A factorial may be zero modulo a prime. Check the condition first.
4. **Test and rebuild.** Try zero, one, a negative input, a non-coprime pair, and a boundary product. Use the hints before the solutions, then rewrite the method independently.

How the material is labelled

A **complete example** or **complete solution** has headers and its own **main**. Its listing comes directly from a supplied `.cpp` file. Compile one file at a time. Fragments and deliberately wrong examples are labelled separately.

Pause and predict answers appear nearby; cover them first. The optional extensions and Practice 18 to 20 are a second-reading route. Main topics and coding solutions begin on fresh pages.

One distinction that prevents many wrong answers

Taking a remainder preserves addition and multiplication, but it does not preserve ordering or ordinary integer division. Taking `% mod` after an overflowing multiplication does not repair that multiplication either.

Language and prerequisites

All complete programs use **standard C++23**, with no GNU extension headers. You should know functions, references, vectors, recursion, binary representation, and Big-O notation. Session 4 supplies combinations; Session 5 supplies prime factorization. We introduce GCD and LCM before inverses because coprimality is the condition that makes modular division possible.

The Level 1 learning path

Session	Topics
1	Two Pointers and Binary Search. Part A: two pointers and sliding windows. Part B: sorted lookup, boundaries, and search on the answer.
2	Recursion and Backtracking. Calls, decision trees, enumeration, pruning, and state restoration.
3	Bit Manipulation and Bitmasking. Binary values, bitwise operations, XOR, compact sets, and subsets.
4	Counting Principles. Addition, multiplication, nCr , nPr , and inclusion-exclusion.
5	Number Theory, Part 1. Divisors, prime factorization, and the sieve.
6	Number Theory, Part 2. Modular arithmetic, binary power (exponentiation), modular inverse, GCD, and LCM.

The six-session series is complete

This handout contains **Session 6: Number Theory, Part 2**. Its section numbers begin with 6 and match the complete collected book, which now contains **all six sessions**. Two Pointers and Binary Search remain combined in Session 1.

What you should be able to do afterwards

Compute common divisors and multiples, justify Euclid's algorithm, maintain canonical residues, evaluate enormous powers, determine whether an inverse exists, compute inverses under general or prime moduli, and use factorial tables for modular combinations under the correct bounds. Explain which products fit in the chosen type.

Conventions used throughout

A modulus is positive. Inverse tasks use $m \geq 2$. Canonical residues lie in $[0, m)$, even when inputs are negative. Exponents are nonnegative unless a task explicitly says otherwise; the programs use the empty-product convention $a^0 = 1$, including $0^0 = 1$ in these counting tasks.

Core modular products use $1 \leq m \leq 10^9 + 7$ and `long long`; this bound is part of the implementation, not a theorem about every modulus. An optional method handles wider products. Inputs follow their stated contracts and programs print no prompts. `NONE` means no requested inverse or solution exists; `OVER` means an exact positive LCM exceeds a supplied cap.

Complexity counts fixed-width operations. Bounds include preprocessing, per-query work, and output when relevant. Recursion space counts active calls, not all calls ever made.

Contents

How to use this reference	i
The Level 1 learning path	ii
6 Number Theory, Part 2	1
6.1 GCD: find the largest common unit	2
6.2 Euclid: replace the pair without changing its common divisors	3
6.3 Implement Euclid and count its actual work	4
6.4 GCD across a sequence: keep a meaning, not a history	6
6.5 LCM: combine periods using maximum prime exponents	7
6.6 Protect an LCM with a cap before multiplying	9
6.7 Applications: fractions, exact pieces, and repeated-step cycles	10
6.8 Congruence: remember a residue instead of the whole integer	12
6.9 C++ remainder is not always the canonical residue	13
6.10 Add, subtract, and multiply residues without losing the answer	14
6.11 Ordinary division and modular cancellation are different	16
6.12 Process a huge decimal integer without storing its value	17
6.13 Modular prefix sums: subtract summaries, then normalize	18
6.14 Binary power: reuse squares instead of multiplying repeatedly	20
6.15 An iterative modular-power function with a precise invariant	21
6.16 Recursive binary power: calculate the half only once	23
6.17 Power boundaries: zero, composite moduli, and safe products	24
6.18 Application: count huge families with a short exponentiation	25
6.19 An inverse exists exactly when the value and modulus are coprime	27
6.20 Extended Euclid: return coefficients as well as a GCD	28
6.21 Compute a general-modulus inverse and handle failure explicitly	30
6.22 A prime modulus permits the Fermat inverse formula	32
6.23 A modular quotient is multiplication by an existing inverse	34
6.24 Linear congruences: zero, one, or several residue solutions	36
6.25 Modular combinations: factorial tables need invertible denominators	38
6.26 GCD summaries: combine a prefix and suffix instead of subtracting	41
6.27 Debug the conditions, arithmetic, and total workload	43
6.28 Optional extension: Euler's totient connects factors and residues	44
6.29 Optional extension: modular multiplication when the modulus is large	46
6.30 Optional extension: an exponent too long for an integer variable	48
6.31 Optional extension: combine two coprime congruences	50
6.32 Optional extension: a geometric sum without modular division	52
6.33 Practice problems: state the arithmetic contract before coding	54
6.34 Trace, correctness, and complexity drills	61
6.35 Hints before solutions	67
6.36 Explained practice solutions	72
6.37 Explained trace and reasoning drills	92
6.38 Quick reference, glossary, and readiness checklist	96
6.39 Technical notes, verification, and references	102
6.40 Completing Level 1: build an argument before a template	106

SESSION 6

Number Theory, Part 2

Keep the congruence. Prove the shortcut. Check that division is allowed.

Session 5 described the prime structure of an integer. This session connects several integers: their common factors, their common multiples, and their remainders under the same modulus. It also shows how repeated squaring turns an enormous exponent into a short computation.

The recurring question is not just “Can I write a formula?” It is “Which information does this transformation preserve?” Euclid preserves common divisors. Modular arithmetic preserves certain algebraic relationships. Binary exponentiation preserves the power still represented by its state.

Five tools and the questions they answer

Tool	Question	Important condition
GCD	What is the largest common unit?	Handle signs and zeros deliberately.
LCM	When do exact periods coincide?	The result can be much larger than either input.
Modular arithmetic	Which remainder is enough to retain?	Use one positive modulus consistently.
Binary power	How can a huge exponent be processed quickly?	Every intermediate product must be safe.
Modular inverse	Can multiplication by a value be undone?	The value and the modulus must be coprime.

Your route through the chapter

Stage A: exact common structure. Derive GCD, Euclid, LCM, fractions, and repeated-step cycles.

Stage B: remainders and powers. Learn congruence, normalized residues, safe operations, digit processing, and iterative and recursive binary exponentiation.

Stage C: division and counting. Derive inverse existence, extended Euclid, prime-modulus inverses, linear congruences, and modular combinations.

Stage D: practise and connect. Solve the problems and explain their assumptions. Optional extensions cover wider arithmetic, huge decimal exponents, Euler’s theorem, two congruences, and geometric sums without division.

Do not choose an algorithm from one word in the statement

“Modulo” does not mean the modulus is prime. “Power” does not mean the exponent should be reduced. “Division” does not mean an inverse exists. Read the full contract before selecting a helper.

6.1 GCD: find the largest common unit

Start with divisors you can list

The positive divisors of 18 are 1, 2, 3, 6, 9, 18. The positive divisors of 30 are 1, 2, 3, 5, 6, 10, 15, 30. Their common divisors are 1, 2, 3, 6, so their **greatest common divisor** is 6:

$$\gcd(18, 30) = 6.$$

Imagine two ribbons with lengths 18 and 30. A piece length d cuts both ribbons exactly when d divides both lengths. The largest such length is their GCD. This interpretation is more useful than memorizing a function name.

Prime exponents give another description

Session 5 gives $18 = 2^1 3^2$ and $30 = 2^1 3^1 5^1$. A common divisor cannot use more copies of a prime than either input contains. Take the **minimum exponent** of every prime, treating a missing prime as exponent zero:

$$\gcd(18, 30) = 2^{\min(1,1)} 3^{\min(2,1)} 5^{\min(0,1)} = 6.$$

This explains the answer, but factoring both inputs is not necessary. Euclid's algorithm will compute it directly from remainders.

Coprime does not mean prime

Two integers are **coprime** when their GCD is 1. Both 8 and 15 are composite, yet they are coprime. Two different primes are coprime; two equal primes are not.

Signs and zeros need a convention

Signs do not change positive divisors: $\gcd(-18, 30) = 6$. For $a > 0$, every divisor of a divides zero, so $\gcd(a, 0) = a$. Mathematically, all positive integers divide both zeros, so there is no greatest positive common divisor of $(0, 0)$. In this book's code, as in `std::gcd`, $\gcd(0, 0)$ **returns 0 by convention**.^[6-2]

A bound can make sign handling safe

Several examples accept only nonnegative inputs. Signed examples state smaller bounds. Negating the smallest representable signed integer is not generally safe, and `std::gcd` requires the magnitudes of its arguments to be representable in its result type. Do not silently extend a bounded helper to every possible signed value.^{[6-2][6-4]}

Pause and predict: 1

Are 6, 10, and 15 pairwise coprime because their combined GCD is 1?

Check your prediction. No. The combined GCD is 1, but $\gcd(6, 10) = 2$, $\gcd(6, 15) = 3$, and $\gcd(10, 15) = 5$. "Pairwise coprime" asks about every pair, not one GCD of the whole list.

6.2 Euclid: replace the pair without changing its common divisors

For nonnegative a and positive b , integer division gives

$$a = qb + r, \quad 0 \leq r < b.$$

The useful identity is

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

This is the core of Euclid's algorithm.^[6-1] We can prove the transformation without guessing from examples.

Why no common divisor is lost or added

Suppose d divides both a and b . It divides their difference $a - qb = r$, so it also divides both b and r .

Conversely, suppose d divides both b and r . It divides $qb + r = a$, so it also divides both a and b .

The two pairs have **exactly the same common divisors**. Therefore their greatest common divisor is the same. We preserve more than a sample answer: we preserve the entire condition that defines an answer.

Trace the changing pair

a	b	q	r	Next pair
252	198	1	54	(198, 54)
198	54	3	36	(54, 36)
54	36	1	18	(36, 18)
36	18	2	0	(18, 0)

The last nonzero value is 18. We stop at (18, 0) because the answer is now immediate. We do *not* attempt $18 \bmod 0$.

Why the algorithm must stop

After every nonterminal step, the new second value is a nonnegative remainder strictly smaller than the old second value. Such a sequence cannot decrease forever. The GCD is preserved until the terminal pair $(g, 0)$, whose answer is g .

If $a < b$, the first remainder is a , so the first update simply changes (a, b) into (b, a) . A preliminary swap is optional, not required for correctness.

Pause and predict: 2

For $(a, b) = (35, 22)$, what are the next two pairs? Can the smaller input appear first?

Check your prediction. The next pairs are (22, 13) and (13, 9). Starting with (22, 35) is also valid: the first update becomes (35, 22) and the same argument applies.

6.3 Implement Euclid and count its actual work

Task. Read $0 \leq a, b \leq 10^{18}$ and print their GCD, with the stated zero convention. Both input values fit in `long long`.

Complete example | C++23

```
// 0 <= a, b <= 10^18; gcd(0,0) is defined as 0 here.
#include <iostream>
long long gcdValue(long long a, long long b) {
    while (b != 0) {
        long long remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0;
    std::cin >> a >> b;
    std::cout << gcdValue(a, b) << '\n';
}
```

Sample input

Sample output

252 198

18

Read the assignments in order

The remainder must be computed using the *old* values of both variables. Only then do we replace a with b and b with the stored remainder. Writing `a = b;` first and then `b = a % b;` would use two equal values and produce zero immediately.

Invariant. The current pair has the same GCD as the original pair. The proof in the previous section justifies one iteration. At termination, `b == 0`, so `a` is the required answer.

Why the time is logarithmic

Consider two consecutive remainder steps after the pair is in decreasing order. If $r < b/2$, the smaller value has already at least halved. Otherwise, $b/2 \leq r < b$, so the next remainder is $b - r \leq b/2$. Thus every two steps reduce the smaller positive value by at least a factor of two.

The number of remainder operations is $O(1 + \log(\min(a, b) + 1))$. For two positive growing inputs, this is usually written $O(\log \min(a, b))$. The loop uses $O(1)$ extra memory. This is a **fixed-width operation model**, not a claim about constant-cost arithmetic on integers of arbitrary length.

Pause and predict: 3

Why store `remainder` before assigning `a = b`?

Check your prediction. It must represent the remainder of the original pair for this iteration. Once `a` is overwritten, that pair no longer exists in the two variables.

The same identity as a recursive contract

A recursive helper promises to return the GCD of its two arguments. The base case is `b == 0`; the recursive call uses the equivalent smaller pair.

Complete example | C++23

```
// 0 <= a, b <= 10^18; gcd(0,0) is 0.
#include <iostream>
long long gcdValue(long long a, long long b) {
    if (b == 0) return a;
    return gcdValue(b, a % b);
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0;
    std::cin >> a >> b;
    std::cout << gcdValue(a, b) << '\n';
}
```

Sample input

198 252

Sample output

18

The total number of calls is logarithmic, but calls remain active until deeper calls return. Without relying on compiler optimization, auxiliary stack space is $O(1 + \log(\min(a, b) + 1))$. The iterative version avoids that call stack.

The standard-library version

After understanding the algorithm, ordinary solutions can use the library helper:

```
// Inside main; include <numeric>. Bounds make magnitudes representable.
long long answer = std::gcd(a, b);
```

`std::gcd` handles signs, returns a nonnegative result, and returns zero for two zero arguments. The function's result type is derived from its *argument types*; assigning it afterwards to a wider variable does not change those types.^[6-2]

Do not confuse a library contract with one implementation

The logarithmic analysis above proves the cost of our Euclidean code. The C++23 specification states the mathematical behavior and input requirements of `std::gcd`; it does not require this exact loop or prescribe one particular implementation. Use the derived algorithm when an implementation-level argument is needed.

6.4 GCD across a sequence: keep a meaning, not a history

A divisor divides every number in a list exactly when it divides the GCD accumulated so far and the next number. This allows a left-to-right fold:

$$g_0 = 0, \quad g_{i+1} = \gcd(g_i, a_i).$$

The zero is useful because $\gcd(0, a) = a$. It is not an arbitrary initialization.

Task. Read $0 \leq n \leq 200000$ nonnegative values, each at most 10^{18} . Print the combined GCD. The empty-list result is defined as 0.

Complete example | C++23

```
// 0 <= n <= 200000; 0 <= values <= 10^18. Empty-list GCD is 0.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    long long answer = 0;
    for (int i = 0; i < n; ++i) {
        long long value = 0;
        std::cin >> value;
        answer = std::gcd(answer, value);
    }
    std::cout << answer << '\n';
}
```

Sample input

Sample output

4

42

84 126 210 0

Meaning of the state. After reading i values, **answer** is their combined GCD. The program never needs to revisit a previous value, so storing the full array would be unnecessary.

For a largest value A , $O(n(1 + \log(A + 1)))$ is a safe arithmetic bound for a fold using Euclidean GCDs. The program stores only $O(1)$ extra state. Input reading still costs $O(n)$, even if the running GCD becomes 1.

Pause and predict: 4

The running GCD becomes 1 before the input ends. Can it later become 7? Can you stop reading immediately in a multiple-test program?

Check your prediction. It cannot increase to 7: $\gcd(1, x) = 1$. But unread values still belong to this test case. Consume them so they are not mistaken for the next case's input.

6.5 LCM: combine periods using maximum prime exponents

For positive a, b , their **least common multiple** is the smallest positive integer divisible by both. A signal every 18 seconds and a signal every 30 seconds coincide again after 90 seconds, not after their GCD of 6.

Derive the formula from Session 5

A common multiple must contain enough copies of each prime to include both inputs. Its smallest possible exponent is their **maximum**. The GCD uses the minimum. Since $\min(x, y) + \max(x, y) = x + y$ for every pair of exponents,

$$\gcd(a, b) \operatorname{lcm}(a, b) = ab \quad \text{for } a, b > 0.$$

Therefore

$$\operatorname{lcm}(a, b) = \frac{a}{\gcd(a, b)} b.$$

The division is exact. Doing it first can avoid constructing a much larger intermediate product.

Task. Read $0 \leq a, b \leq 10^9$. Print their LCM; define the result as 0 when either value is 0, as the C++ library does.^[6-2]

Complete example | C++23

```
// 0 <= a, b <= 10^9; the LCM fits in long long.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0;
    std::cin >> a >> b;
    if (a == 0 || b == 0) std::cout << 0 << '\n';
    else std::cout << (a / std::gcd(a, b)) * b << '\n';
}
```

Sample input

18 30

Sample output

90

Numeric bound. For these inputs, the positive result is at most $ab \leq 10^{18}$, so both the final multiplication and its result fit in `long long`. The zero branch avoids division by $\gcd(0, 0) = 0$.

Pause and predict: 5

Is dividing first enough to make the LCM of any two 64-bit integers safe?

Check your prediction. No. It avoids one unnecessarily large intermediate, but the true LCM may still exceed the result type. A bound or an explicit cap check is still required.

Library convenience does not remove numeric preconditions

`std::lcm` is available in `<numeric>`. It requires the input magnitudes *and the LCM* to be representable in the common argument type. It is not an overflow detector.^[6-2]

```
// Inside main; a and b have type long long and are in [0, 10^9].  
long long period = std::lcm(a, b);
```

Wrong assumption. Declaring a wide destination does not widen a library call that receives narrower arguments:

```
// Wrong example on a 32-bit-int platform: true LCM exceeds int.  
int a = 1000000000;  
int b = 999999999;  
long long period = std::lcm(a, b);
```

The inputs should already have the intended wide type, or be converted before the call. Even then, check that the true result fits in that wide type.

Do not take remainders before computing exact GCD or LCM

Suppose the task asks for an exact LCM and only then its remainder. Replacing each input by its residue generally changes its prime structure. With modulus 5, the original pair (6, 10) has LCM 30 and remainder 0, while the residue pair (1, 0) does not preserve either original period.

An even clearer GCD example is (6, 10) modulo 7: the original GCD is 2, but the residue pair (6, 3) has GCD 3. Congruence alone is not enough to recover an exact GCD. Preserve the exact values while computing the exact property.

A list of LCMs can grow quickly

For positive inputs, fold from 1 using $L \leftarrow \text{lcm}(L, a_i)$. Every new result is a multiple of the previous result, so it never decreases. This permits an early “over the cap” state when the task needs only bounded results. Zero inputs would break this monotonicity because the LCM convention then becomes zero; the next example therefore explicitly requires positive values.

6.6 Protect an LCM with a cap before multiplying

Task. Read $1 \leq n \leq 200000$ positive integers and a cap $1 \leq C \leq 10^{18}$. Values are also at most 10^{18} . Print their exact LCM if it is at most C ; otherwise print **OVER**.

Let $u = L / \gcd(L, x)$. Since u, x, C are positive,

$$ux > C \iff u > \lfloor C/x \rfloor.$$

The right side can be checked without forming ux .

Complete example | C++23

```
// 1 <= n <= 200000; 1 <= values, limit <= 10^18.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    long long limit = 0;
    std::cin >> n >> limit;
    long long current = 1;
    bool tooLarge = false;
    for (int i = 0; i < n; ++i) {
        long long value = 0;
        std::cin >> value;
        if (tooLarge) continue; // Still consume the complete input.
        long long reduced = current / std::gcd(current, value);
        if (reduced > limit / value) tooLarge = true;
        else current = reduced * value;
    }
    if (tooLarge) std::cout << "OVER" << '\n';
    else std::cout << current << '\n';
}
```

Sample input

Sample output

3 100

30

6 10 15

Why it works. While the flag is false, **current** is the exact LCM of all processed values and is at most the cap. A successful check permits a safe multiplication. Once the true LCM exceeds the cap, later positive inputs cannot reduce it. We retain the flag but continue reading.

The algorithm uses $O(n(1 + \log(C + A + 1)))$ as a convenient safe Euclidean-operation bound, with largest input A , and $O(1)$ extra memory. A tighter description sums the individual GCD costs and includes the n reads.

Pause and predict: 6

The current LCM is 30, the next value is 7, and the cap is 100. Which check rejects the multiplication?

Check your prediction. The GCD is 1, so the reduced value is 30. Since $30 > \lfloor 100/7 \rfloor = 14$, the next LCM would exceed the cap. It need not be constructed.

6.7 Applications: fractions, exact pieces, and repeated-step cycles

Reduce a fraction without using floating point

A fraction a/b , with $b \neq 0$, is unchanged when numerator and denominator are divided by the same nonzero integer. Dividing both by $\gcd(a, b)$ removes every common factor. Put the sign in the numerator so the representation is consistent.

Task. Read numerator and denominator between -10^9 and 10^9 , with a nonzero denominator. Print the reduced numerator and a positive denominator.

Complete example | C++23

```
// -10^9 <= numerator, denominator <= 10^9; denominator != 0.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long top = 0, bottom = 0;
    std::cin >> top >> bottom;
    long long g = std::gcd(top, bottom);
    top /= g;
    bottom /= g;
    if (bottom < 0) {
        top = -top;
        bottom = -bottom;
    }
    std::cout << top << ' ' << bottom << '\n';
}
```

Sample input

-18 -24

Sample output

3 4

For a zero numerator, the GCD is the denominator's magnitude, so the normalized result is 0 1. There is no need for a special floating-point case. The sign changes and divisions are safe under the printed bounds.

Pause and predict: 7

What canonical pair represents $0/(-15)$? Why must the original denominator still be nonzero?

Check your prediction. The pair is $(0, 1)$. A denominator of zero would not describe a valid fraction, so the GCD calculation cannot make it one.

A circular walk has a GCD-based period

There are $n > 0$ positions numbered 0 through $n - 1$. Starting at 0, repeatedly advance by $k \geq 0$, wrapping around. After t moves, the position is the remainder of tk modulo n . A return to the start requires $n \mid tk$.

Write $g = \gcd(n, k)$, $n = gn'$, and $k = gk'$. Then n' and k' share no prime factors. The condition becomes $n' \mid tk'$. All the prime factors needed by n' must come from t , so the smallest positive return time is n' :

$$\text{number of visited positions} = \frac{n}{\gcd(n, k)}.$$

No position repeats earlier: such a repeat would give a smaller positive difference between move counts that also returns to 0.

For $n = 12$ and $k = 8$, the positions are 0, 8, 4, 0. The answer is $12/4 = 3$. With $k = 0$, only the starting position is visited, and the same formula gives $n/n = 1$.

Do not confuse a return period with a first meeting time

For two signals that *both start at time zero*, the positive simultaneous times are multiples of the LCM of their positive periods. Different starting offsets create congruence conditions instead; the plain LCM alone does not determine the first meeting. The optional two-congruence section will address such conditions under a coprimality promise.

A quick decision test

If the story asks for a largest equal unit that fits every quantity, think GCD. If it asks for the first positive shared multiple of exact periods, think LCM. If it asks how many positions a fixed step visits on a cycle, derive the return condition. The words “largest” and “smallest” alone are not enough: identify what must divide what.

6.8 Congruence: remember a residue instead of the whole integer

Division separates a multiple and a remainder

For an integer a and a positive modulus m , there is a unique representation

$$a = qm + r, \quad 0 \leq r < m.$$

The value r is the **canonical residue** of a modulo m . With $m = 7$, the integers 4, 11, 18, -3 , -10 all have residue 4. They differ by multiples of 7.

We write

$$a \equiv b \pmod{m}$$

when m divides $a - b$. Read this as “ a is congruent to b modulo m .” It says their residues are equal, not that the original integers are equal.

a	q	r	Check: $a = 7q + r$
18	2	4	$18 = 14 + 4$
4	0	4	$4 = 0 + 4$
-3	-1	4	$-3 = -7 + 4$
-10	-2	4	$-10 = -14 + 4$

Why a smaller state can be enough

If a task asks only for the remainder of a huge answer, we may never need to build that answer. We need transformations that preserve its congruence. The next sections derive which arithmetic operations have this property and which do not.

For example, the count of binary strings of length 1000 is much too large for a 64-bit integer. Its remainder modulo a supplied m can still be computed using small residues, provided each intermediate operation is safe.

The modulus is part of every stored meaning

A stored residue 4 is incomplete information without its modulus. A value congruent to 4 modulo 7 need not be congruent to 4 modulo 5. Do not combine residues computed under different moduli as though they referred to the same representation.

When $m = 1$, the only canonical residue is 0. Every integer, including 1, is congruent to zero modulo 1. Ordinary residue and power functions can support this case; inverse tasks in this chapter deliberately use $m \geq 2$.

Pause and predict: 8

Does $18 \equiv 4 \pmod{7}$ imply $18 = 4$? What is the canonical residue of -10 modulo 7?

Check your prediction. No: their difference, not their equality, matters. The difference is 14, which is divisible by 7. The canonical residue of -10 is 4.

6.9 C++ remainder is not always the canonical residue

For integer operands, C++ division truncates toward zero. The remainder satisfies the corresponding quotient-remainder identity. Consequently a negative dividend can have a negative nonzero remainder: $-10 \% 7$ is -3 , not 4 . A zero divisor is invalid.^[6-3]

A normalization helper

Reduce first, then add the positive modulus only when the result is negative:

```
// Above main. Preconditions: mod > 0; value and mod fit long long.
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
```

After the first statement, the remainder lies strictly between $-m$ and m . If negative, adding m moves it into $[0, m)$. If nonnegative, it is already canonical.

Why the order matters

For `value = -100` and `mod = 7`, adding the modulus once *before* reducing gives -93 , which is still negative. Normalization must first remove the possibly many whole multiples.

The familiar expression $(\text{value} \% \text{mod} + \text{mod}) \% \text{mod}$ is correct when its addition fits. The conditional form avoids adding two positive values when $\text{value} \% \text{mod}$ is already nonnegative, so it is easier to justify for a wide positive modulus.

A residue is useful only for the intended operation

Suppose a sorted array contains 6, 8, 13 and the modulus is 7. Its residues are 6, 1, 6, which are no longer sorted. Reducing values before a comparison or a binary search can destroy the order that justified the algorithm.

Likewise, equal residues do not identify equal original values. Congruence forgets whole multiples of the modulus on purpose. A later step cannot recover information that was discarded.

Keep modulus and divisor restrictions separate

The core examples promise $m \geq 1$, so they never use $\% 0$. A task that asks for an inverse additionally promises $m \geq 2$, but still does not promise that the value has an inverse. That is a different condition.

Pause and predict: 9

Evaluate $-17 \% 7$, then normalize it. Why is normalization useful before using a remainder as a vector index?

Check your prediction. C++ produces -3 ; adding 7 gives 4. A negative remainder is not a valid nonnegative frequency-table index.

6.10 Add, subtract, and multiply residues without losing the answer

Suppose $a = qm + r$ and $b = tm + s$, with r, s canonical. Then

$$a + b = (q + t)m + (r + s),$$

so the sum is congruent to $r + s$. Subtraction works by the same argument. For multiplication,

$$ab = (qtm + qs + tr)m + rs,$$

so all terms except rs are multiples of m .

The three safe mathematical rules

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

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

$$(ab) \bmod m = ((a \bmod m)(b \bmod m)) \bmod m.$$

Here $\bmod$ denotes a canonical mathematical residue. In C++, normalize negative results as needed.

Task. Read $|a|, |b| \leq 10^{18}$ and $1 \leq m \leq 10^9 + 7$. Print the canonical residues of $a + b$, $a - b$, and ab , one per line.

Complete example | C++23

```
// |a|, |b| <= 10^18; 1 <= mod <= 1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0, mod = 0;
    std::cin >> a >> b >> mod;
    a = normalize(a, mod);
    b = normalize(b, mod);
    std::cout << (a + b) % mod << '\n';
    std::cout << normalize(a - b, mod) << '\n';
    std::cout << a * b % mod << '\n';
}
```

Sample input

-17 9 7

Sample output

6

2

1

The mathematical identity does not prevent machine overflow

After normalization, both values are at most $m - 1$. With the stated bound,

$$(m - 1)^2 \leq (1000000006)^2 = 1000000012000000036,$$

which fits in a 64-bit signed `long long`. The sum is at most $2m - 2$, and the difference lies between $-(m - 1)$ and $m - 1$. Each expression is therefore safe *before* its final remainder operation.

By contrast, the original product ab can have magnitude 10^{36} . Computing `a * b % mod` before reducing a and b can overflow. Signed overflow is undefined behavior; a later remainder does not repair it.^[6-4]

```
// Wrong example under |a|, |b| <= 10^18:
long long answer = (a * b) % mod;

// Also wrong when x, y are int and their product exceeds int:
long long answer2 = (x * y) % mod;

// A cast must affect the multiplication, not only its destination:
long long answer3 = (1LL * x * y) % mod;
// This last line still requires the long long product to fit.
```

Reduce between products, not only at the end

Even three normalized values near 10^9 can have a triple product near 10^{27} . Use `a * b % mod * c % mod`: left-to-right evaluation reduces the first product before the next multiplication. Do not write `a * b * c % mod` under the same bounds.

Unsigned wraparound is defined modulo a power of two, but your problem's modulus may be something else. Replacing `long long` with an unsigned type is not a general solution to modular-product overflow. The optional wide-multiplication section derives a portable alternative.

Pause and predict: 10

Can $a, b < m$ with $m = 10^{18}$ justify a direct `long long` multiplication? Does using `% m` on the next token change that?

Check your prediction. No. The product can approach 10^{36} , far beyond the type. The remainder happens after multiplication. A larger arithmetic type or an overflow-safe multiplication algorithm is required.

6.11 Ordinary division and modular cancellation are different

Addition, subtraction, and multiplication preserve congruence. Ordinary integer division generally does not.

Take $8/2$ modulo 5. Computing the exact quotient first gives 4. Reducing first gives residues 3 and 2, whose C++ integer quotient is 1. Neither integer division nor floating-point division is the modular operation we need.

The question that modular division actually asks

To divide a residue a by b , we ask for a residue x such that

$$bx \equiv a \pmod{m}.$$

There may be one solution, several solutions, or no solution. A **modular inverse** is what guarantees a unique solution for every numerator. We will derive its existence condition using GCD.

Equation	Residues tested	Outcome
$3x \equiv 8 \pmod{11}$	$x = 10$ works	One solution modulo 11.
$2x \equiv 2 \pmod{6}$	$x = 1, 4$ work	Two solutions modulo 6.
$2x \equiv 1 \pmod{6}$	An even product cannot have residue 1	No solution.

Cancelling a factor can change the modulus

The congruence $2x \equiv 2 \pmod{6}$ is equivalent to $6 \mid 2(x-1)$, hence to $3 \mid x-1$. The correct reduced condition is $x \equiv 1 \pmod{3}$, not $x \equiv 1 \pmod{6}$. The latter would lose the valid solution 4.

Exact integer division can still be used *before* information is discarded, when the original integer is available and divisibility is guaranteed. This is how the LCM formula safely divides by a GCD. It is not the same as dividing two already-reduced residues.

A prime modulus is not required for ordinary modular operations

Everything in the preceding three arithmetic rules works for a positive composite modulus too. Only particular inverse formulas require primality. Always keep that condition beside the formula instead of attaching it to the word “modulo.”

Pause and predict: 11

Why does $(8 \% 5) / (2 \% 5)$ fail to compute $(8/2) \bmod 5$?

Check your prediction. It performs ordinary integer division on 3 and 2, giving 1. The desired modular result is 4, because $2 \cdot 4 \equiv 8 \pmod{5}$. Reducing discarded information that ordinary division needed.

6.12 Process a huge decimal integer without storing its value

Suppose the number already read is v . Appending a decimal digit d changes it to $10v + d$. If only its residue matters, keep

$$r \leftarrow (10r + d) \bmod m.$$

The multiplication and addition rules prove that this update preserves the remainder of every processed prefix.

Task. Read a nonempty string of at most 200000 decimal digits and $1 \leq m \leq 10^9 + 7$. Leading zeros are allowed. Print the number's remainder without converting the full string to an integer.

Complete example | C++23

```
// 1..200000 decimal digits; 1 <= mod <= 1000000007.
#include <iostream>
#include <string>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::string digits;
    long long mod = 0;
    std::cin >> digits >> mod;
    long long remainder = 0;
    for (char ch : digits)
        remainder = (remainder * 10 + (ch - '0')) % mod;
    std::cout << remainder << '\n';
}
```

Sample input

Sample output

12345678901234567890 97

3

Trace a short input

For digits 3142 and modulus 7:

Next digit	Old residue	Unreduced expression	New residue
3	0	$10 \cdot 0 + 3 = 3$	3
1	3	$10 \cdot 3 + 1 = 31$	3
4	3	$10 \cdot 3 + 4 = 34$	6
2	6	$10 \cdot 6 + 2 = 62$	6

The largest intermediate, $10(m - 1) + 9$, fits in `long long`. Time and stored-input space are $O(L)$ for L digits; the arithmetic state is $O(1)$.

Pause and predict: 12

Do leading zeros change the result? Can this method use modulus 12 rather than a prime?

Check your prediction. Leading zeros do not change the residue. Composite modulus 12 is valid because the update requires no inverse.

6.13 Modular prefix sums: subtract summaries, then normalize

Level 0 introduced prefix sums. When only query remainders are needed, define

$$P[0] = 0, \quad P[i] \equiv a_1 + \cdots + a_i \pmod{m}.$$

Store every $P[i]$ canonically. For a one-based inclusive range $[l, r]$,

$$\sum_{i=l}^r a_i \equiv P[r] - P[l-1] \pmod{m}.$$

The two stored residues are enough because subtraction preserves congruence.

Task. Read $1 \leq n \leq 200000$, $0 \leq q \leq 200000$, and $1 \leq m \leq 10^9 + 7$. Then read n integers of magnitude at most 10^{18} , followed by valid one-based inclusive query ranges. Print each range sum modulo m .

Complete example | C++23

```
// 1 <= n <= 200000; 0 <= q <= 200000; |values| <= 10^18; mod <= 1000000007
// and mod >= 1.
#include <iostream>
#include <vector>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    long long mod = 0;
    std::cin >> n >> q >> mod;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = (pref[i - 1] + normalize(value, mod)) % mod;
    }
    while (q--) {
        int left = 0, right = 0;
        std::cin >> left >> right;
        std::cout << normalize(pref[right] - pref[left - 1], mod)
                  << '\n';
    }
}
```

Sample input

Sample output

4 3 7	2
5 -8 9 3	1
1 4	6
2 3	
2 2	

A decreasing prefix is not a bug here

With array 5, −8, 9, 3 and modulus 7, the canonical prefixes are

$$P = [0, 5, 4, 6, 2].$$

The final prefix is 2 although the previous one is 6. Residues can wrap around. They are **not sorted cumulative totals**, even when original values are nonnegative.

For the single-element range $[2, 2]$, compute $P[2] - P[1] = 4 - 5 = -1$, then normalize it to 6. This equals $-8 \bmod 7$. Printing the raw C++ negative remainder would violate the requested output convention.

Correctness and cost

The construction preserves the sum of each processed prefix by the modular addition rule. Query subtraction cancels all positions before l , leaving the desired range. Normalization chooses its canonical representative.

Building the table costs $O(n)$; each query costs $O(1)$, so the program costs $O(n + q)$ and stores $O(n)$ residues. The original array is not stored because this task never needs it after preprocessing. Updates would require revisiting the summary, just as in the static-range-query lesson.

A residue prefix does not support every ordinary-prefix algorithm

It supports modular range sums. It does not preserve numerical order, exact sums, or the monotone prefix property needed by some binary searches. Decide what a query needs before choosing what to store.

Pause and predict: 13

Can you binary-search the first modular prefix whose value is at least a target merely because all original array values are positive?

Check your prediction. No. Remainders wrap around, so the modular prefixes need not be sorted. The ordering invariant required by binary search has been lost.

6.14 Binary power: reuse squares instead of multiplying repeatedly

The direct method for a^e performs e multiplications. It is reasonable for $e = 10$, but not for $e = 10^{18}$. Reducing after every multiplication controls the numeric values, but does not reduce this enormous iteration count.

Halving the exponent gives a smaller problem

For $k \geq 0$,

$$a^{2k} = (a^k)^2, \quad a^{2k+1} = (a^k)^2 a.$$

Both cases need one half-size power, one squaring, and at most one extra multiplication. The base case is $a^0 = 1$. The method is called **binary exponentiation**, **binary power**, or **exponentiation by squaring**.^[6-1]

Binary digits describe which powers are needed

Session 3 introduced binary representation. Since $13 = 8 + 4 + 1$,

$$a^{13} = a^8 a^4 a^1.$$

Successive squaring produces $a, a^2, a^4, a^8, \dots$. The bits of 13 tell us to include the first, third, and fourth of these powers.

For $a = 3$ modulo 17:

Exponent weight	Power residue	Bit in 13	Action
1	3	1	Multiply the answer by 3.
2	9	0	Skip this factor.
4	13	1	Multiply the answer by 13.
8	16	1	Multiply the answer by 16.

Multiplying the selected residues, reducing after each product, gives 12. There is no need to construct 3^{13} first.

What “binary” does not mean

The algorithm uses the binary structure of the exponent. It does not require the exponent to be read as a binary string, and it does not require bitwise syntax. Testing oddness with `exponent % 2` and halving with `exponent /= 2` are sufficient and easy to trace.

Pause and predict: 14

Which power weights are selected for exponent 22? How many repeated squarings are needed to reach the largest weight?

Check your prediction. $22 = 16 + 4 + 2$, so select a^{16}, a^4, a^2 . Starting from a , four squarings reach a^{16} . The number of stages grows with the number of binary digits, not with 22 itself.

6.15 An iterative modular-power function with a precise invariant

Task. Read $|a| \leq 10^{18}$, $0 \leq e \leq 10^{18}$, and $1 \leq m \leq 10^9 + 7$. Print $a^e \bmod m$ as a canonical residue. The task uses the empty-product convention when $e = 0$.

Complete example | C++23

```
// |base| <= 10^18; 0 <= exponent <= 10^18; 1 <= mod <= 1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long base = 0, exponent = 0, mod = 0;
    std::cin >> base >> exponent >> mod;
    std::cout << modPower(base, exponent, mod) << '\n';
}
```

Sample input

3 13 17

Sample output

12

Name the three moving values

result contains the product already selected. **base** is the current power-of-two contribution. **exponent** describes the contributions still needed. The original input exponent is intentionally reduced during the loop; store it separately only if the surrounding task needs it later.

Invariant. If the original inputs are A, E , then at the start of every iteration,

$$\text{result} \cdot \text{base}^{\text{exponent}} \equiv A^E \pmod{m}.$$

The stored residues stay in $[0, m)$. Their products fit under the printed modulus bound.

Prove the odd and even updates

Suppose the remaining exponent is $2k$. Squaring the base and replacing the exponent by k preserves base^{2k} .

Suppose it is $2k + 1$. First move one copy of the current base into **result**. Then square the base and replace the exponent by k . Algebraically,

$$RB^{2k+1} = (RB)(B^2)^k.$$

Taking remainders during these products preserves the congruence. When the exponent reaches zero, the remaining power is 1, so **result** is the required residue.

Trace the state before each iteration

For $A = 3$, $E = 13$, and $m = 17$:

result	base	exponent	What this iteration does
1	3	13	Select 3; square to 9; halve to 6.
3	9	6	Select nothing; square to 13; halve to 3.
3	13	3	Select 13, giving 5; square to 16.
5	16	1	Select 16, giving 12; exponent becomes 0.

Time, memory, and identities

A positive exponent loses one binary digit per iteration. The running time is $O(1 + \log(e + 1))$ and extra memory is $O(1)$. An exponent at most 10^{18} requires at most 60 iterations in this implementation.

Initialize with $1 \% \text{mod}$, not simply 1, so exponent zero also gives the canonical result for modulus 1. Squaring on the final iteration is harmless because the modular product is still bounded; an exact, nonmodular power routine would need a separate bound for even that unused last square.

Pause and predict: 15

Why must the odd case multiply by the old base before the base is squared?

Check your prediction. The odd exponent contains one copy of the current power B . Multiplying by B^2 instead would select the next power too early and break the invariant.

6.16 Recursive binary power: calculate the half only once

The exponent identities also give a recursive implementation. The helper receives an already normalized base so every multiplication is between bounded residues.

Complete example | C++23

```
// |base| <= 10^18; 0 <= exponent <= 10^18; 1 <= mod <= 1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long powerNormalized(long long base, long long exponent,
                          long long mod) {
    if (exponent == 0) return 1 % mod;
    long long half = powerNormalized(base, exponent / 2, mod);
    long long answer = half * half % mod;
    if (exponent % 2 == 1) answer = answer * base % mod;
    return answer;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long base = 0, exponent = 0, mod = 0;
    std::cin >> base >> exponent >> mod;
    std::cout << powerNormalized(normalize(base, mod), exponent, mod)
               << '\n';
}
```

Sample input

-3 5 13

Sample output

4

One recursive call is the difference between fast and slow

The single result `half` is used twice in the squaring. Writing two calls for the same half exponent would build two identical subtrees: **Wrong performance pattern:** calling `powerNormalized(base, exponent / 2, mod)` twice and multiplying the two returned values, rather than storing one call's result. An odd exponent would still require an extra factor of the base.

For powers of two, this duplicate-call shape has linear total call count in the exponent rather than logarithmic count. Small-looking code is not necessarily efficient code.

One stored half-result gives $O(1 + \log(e + 1))$ time and stack space. The iterative version instead uses constant extra space.

Pause and predict: 16

Does a recursion depth of $O(\log e)$ prove that total time is $O(\log e)$ when every call recursively calls the half-size problem twice?

Check your prediction. No. Depth counts one root-to-leaf path, not the whole branching tree. Store the half answer once to avoid repeated work.

6.17 Power boundaries: zero, composite moduli, and safe products

Five cases to understand before copying the helper

Input situation	Required reasoning
Exponent zero	The empty product is 1; print $1 \bmod m$. These tasks explicitly include $0^0 = 1$.
Base zero, exponent positive	Every selected product is zero modulo m ; the answer is 0.
Modulus 1	The only residue is 0, regardless of the base or nonnegative exponent.
Negative base	Normalize before multiplying. The sign is then handled by congruence.
Composite modulus	The ordinary binary-power algorithm still works. It uses no inverse or prime-specific theorem.

Do not replace exact integer arithmetic with a floating-point power

`std::pow` is not the modular exponentiation function. Converting a large floating-point result back to an integer cannot reliably recover an exact huge power. The operator `^` is bitwise XOR in C++, not exponentiation; Session 3 introduced that distinction.

Do not reduce the exponent merely because the base was reduced

The identity $a^e \equiv (a \bmod m)^e \pmod m$ always holds for nonnegative e . It does not say that e may be replaced by $e \bmod m$.

For example, $2^5 \bmod 5 = 2$, while $2^{5 \bmod 5} = 2^0 \bmod 5 = 1$. Reducing exponents requires a separate theorem and its coprimality conditions. For a machine-sized exponent, ordinary binary power is already fast enough and needs no such reduction.

The multiplication method is part of the complexity

Our core bound makes one modular product constant-cost. If a wider modulus requires an $O(\log m)$ repeated-doubling multiplication, a modular power using it costs $O((1 + \log(e + 1))(1 + \log m))$ rather than simply $O(\log e)$ under that implementation.

Pause and predict: 17

Does $6^{20} \bmod 8$ require a prime modulus or an inverse of 6? What about $6^{-1} \bmod 8$?

Check your prediction. The nonnegative power requires neither and is 0. The negative exponent would require an inverse, which does not exist because $\gcd(6, 8) = 2$. These are different tasks.

6.18 Application: count huge families with a short exponentiation

Session 4's multiplication rule says that a length- n word over k available symbols has k^n possibilities when every position independently chooses any symbol. Positions matter; repetition is allowed.

The exact count may be enormous, but a problem may ask only for its residue. Do not enumerate words and do not compute the exact power first. Evaluate the counting formula directly modulo the supplied modulus.

Task. Read $0 \leq k \leq 10^9$, $0 \leq n \leq 10^{18}$, and $1 \leq m \leq 10^9 + 7$. Print the number of length- n words modulo m . There is one empty word when $n = 0$, including when $k = 0$.

Separate the counting proof from the arithmetic proof

First prove the formula: every position has k choices and every complete sequence is generated exactly once, so multiply k a total of n times. Then prove the evaluator: binary exponentiation returns the remainder of that power using repeated squaring. A correct evaluator cannot repair an incorrect counting formula.

For nonempty subsets of n distinct positions, the count is $2^n - 1$: include or exclude each position, then remove the empty subset once. The modular implementation must normalize the final subtraction, particularly when $m = 1$ or $2^n \bmod m = 0$.

Pause and predict: 18

How many length-zero words exist over an empty alphabet? How many positive-length words exist over it?

Check your prediction. There is one empty word, but no positive-length word. The program's $0^0 = 1$ convention represents the first counting case, not a universal assertion about every mathematical context.

The model must match the formula

With three available symbols and a word length of two, there are $3^2 = 9$ words when repetition is allowed. Forbidding repeated symbols changes the count to $3 \cdot 2 = 6$. Choosing an unranked two-symbol set instead changes it to $\binom{3}{2} = 3$. The same input sizes can describe three different counting problems.

Evaluate the word-count formula modulo the input modulus

Complete example | C++23

```
// 0 <= alphabet <= 10^9; 0 <= length <= 10^18; 1 <= mod <= 1000000007.
// Empty word count is 1.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long alphabet = 0, length = 0, mod = 0;
    std::cin >> alphabet >> length >> mod;
    std::cout << modPower(alphabet, length, mod) << '\n';
}
```

Sample input

4 10 1000

Sample output

576

The sample computes $4^{10} \bmod 1000 = 576$. The modulus 1000 is composite, but that causes no problem: this algorithm uses no division. A zero-length word gives $1 \bmod m$; a positive-length word over zero symbols gives 0.

The input exponent may be as large as 10^{18} , but at most 60 loop iterations are needed. Time is $O(1 + \log(n + 1))$, and the iterative helper uses $O(1)$ extra space. Practice 10 changes the model to nonempty subsets and adds a normalized subtraction.

6.19 An inverse exists exactly when the value and modulus are coprime

For $m \geq 2$, a **modular inverse** of a is a residue x satisfying

$$ax \equiv 1 \pmod{m}.$$

The notation a^{-1} means this multiplicative inverse in the modular setting, not ordinary integer division $1 / a$. Under modulus 12, the inverse of 5 is 5 because $25 \equiv 1 \pmod{12}$. Under modulus 13, the inverse of 5 is 8.

Why a common factor makes inversion impossible

Let $g = \gcd(a, m)$. If $ax \equiv 1 \pmod{m}$, some integer y satisfies $ax + my = 1$. Since g divides both terms on the left, it must divide 1. Hence $g = 1$ is necessary.

For $a = 6, m = 12$, every product $6x$ has residue 0 or 6. It can never have residue 1. Repeatedly trying powers cannot create an inverse that does not exist.

Why coprimality is also enough

Bezout's identity says that integers u, v can be found with

$$au + mv = \gcd(a, m).$$

When the GCD is 1, reduction modulo m gives $au \equiv 1 \pmod{m}$, so the normalized value of u is an inverse. Extended Euclid, derived next, constructs these coefficients. Thus

$$a^{-1} \pmod{m} \text{ exists} \iff \gcd(a, m) = 1.$$

The inverse-existence criterion and extended Euclid are standard number-theory results; the next section derives the coefficient update used in our code.^{[6-7][6-8]}

Uniqueness is modulo the same modulus

If both x and y are inverses, then $m \mid a(x - y)$. Because a and m are coprime, all prime factors required by m must divide $x - y$. Therefore $x \equiv y \pmod{m}$. There is exactly one canonical inverse in $[0, m)$, although infinitely many integer representatives differ by multiples of m .

Pause and predict: 19

Which of 4, 5, and 6 have inverses modulo 12? Does a composite modulus rule out every inverse?

Check your prediction. Only 5 among these three values is coprime to 12, so only 5 is invertible. Composite moduli can have many invertible residues; not every nonzero residue is one.

6.20 Extended Euclid: return coefficients as well as a GCD

An ordinary Euclidean call returns $g = \gcd(a, b)$. An **extended** call additionally returns integers x, y such that

$$ax + by = g.$$

The coefficients may be negative. Their purpose is to satisfy an exact equation, not to count objects or positions.

Derive the update from the recursive promise

Assume the recursive call for $(b, a \bmod b)$ returns u, v with

$$bu + (a \bmod b)v = g.$$

Using $a \bmod b = a - \lfloor a/b \rfloor b$ gives

$$bu + (a - \lfloor a/b \rfloor b)v = av + b(u - \lfloor a/b \rfloor v) = g.$$

Therefore return

$$\boxed{x = v, \quad y = u - \lfloor a/b \rfloor v.}$$

At $b = 0$ and $a > 0$, choose $x = 1, y = 0$ because $a \cdot 1 + 0 \cdot 0 = a$. That supplies the base witness. Each return transforms a valid child witness into a valid parent witness, so the construction also proves Bezout's identity for these inputs.

A back-substitution example

For $a = 35, b = 22$, Euclid gives

$$35 = 22 + 13, \quad 22 = 13 + 9, \quad 13 = 9 + 4, \quad 9 = 2 \cdot 4 + 1.$$

Working backwards,

$$\begin{aligned} 1 &= 9 - 2 \cdot 4 = 3 \cdot 9 - 2 \cdot 13 \\ &= 3 \cdot 22 - 5 \cdot 13 = 8 \cdot 22 - 5 \cdot 35. \end{aligned}$$

Thus $(x, y) = (-5, 8)$ is a valid witness. Modulo 22, the coefficient -5 becomes 17, so $35 \cdot 17 \equiv 1 \pmod{22}$.

Pause and predict: 20

The child returns coefficients u, v for $(b, a \bmod b)$. Why is the parent's first coefficient v , not u ?

Check your prediction. The remainder term contains a once: $(a - qb)v = av - qbv$. Collecting coefficients of a leaves v . Collecting those of b leaves $u - qv$.

A complete witness-producing program

Task. Read $0 \leq a, b \leq 10^9 + 7$, not both zero. Print g, x, y such that $g = \gcd(a, b)$ and $ax + by = g$. Any correct coefficient pair is acceptable.

Complete example | C++23

```
// 0 <= a, b <= 1000000007; not both zero. Output g,x,y with ax+by=g.
#include <iostream>
long long extendedGcd(long long a, long long b,
                      long long& x, long long& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long nextX = 0, nextY = 0;
    long long g = extendedGcd(b, a % b, nextX, nextY);
    x = nextY;
    y = nextX - (a / b) * nextY;
    return g;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0, x = 0, y = 0;
    std::cin >> a >> b;
    long long g = extendedGcd(a, b, x, y);
    std::cout << g << ' ' << x << ' ' << y << '\n';
}
```

Sample input

35 22

Sample output

1 -5 8

Returning call (a, b)	x	y	Check
$(1, 0)$	1	0	1
$(4, 1)$	0	1	1
$(9, 4)$	1	-2	1
$(13, 9)$	-2	3	1
$(22, 13)$	3	-5	1
$(35, 22)$	-5	8	1

The returned scalar is the GCD. Reference parameters carry the two coefficients. Separate local variables `nextX` and `nextY` preserve the child values while the parent results are assigned.

The Euclidean chain has logarithmic length, so both time and active stack space are logarithmic in the input bound. This implementation deliberately uses bounded nonnegative arguments. Its coefficient arithmetic is *not* advertised as safe for arbitrary signed 64-bit inputs.

6.21 Compute a general-modulus inverse and handle failure explicitly

Normalize a first, run extended Euclid on (a, m) , and inspect its returned GCD. If it is not 1, there is no inverse. Otherwise normalize the coefficient of a to select the canonical inverse.

Task. Read $|a| \leq 10^{18}$ and $2 \leq m \leq 10^9 + 7$. Print the canonical inverse of a modulo m , or NONE when none exists.

Complete example | C++23

```
// |a| <= 10^18; 2 <= mod <= 1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long extendedGcd(long long a, long long b,
                     long long& x, long long& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long nextX = 0, nextY = 0;
    long long g = extendedGcd(b, a % b, nextX, nextY);
    x = nextY;
    y = nextX - (a / b) * nextY;
    return g;
}
bool inverse(long long a, long long mod, long long& result) {
    a = normalize(a, mod);
    long long x = 0, y = 0;
    if (extendedGcd(a, mod, x, y) != 1) return false;
    result = normalize(x, mod);
    return true;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, mod = 0, result = 0;
    std::cin >> a >> mod;
    if (inverse(a, mod, result)) std::cout << result << '\n';
    else std::cout << "NONE" << '\n';
}
```

Sample input

Sample output

-5 12

7

Read the success/failure contract

`inverse` returns a Boolean. Only on success does the output reference `result` contain an answer the caller is allowed to use. Printing that reference after a failed call would confuse a default or stale value with a valid inverse.

For input $-5, 12$, normalization gives 7. Extended Euclid produces a coefficient congruent to 7 modulo 12, and indeed $(-5) \cdot 7 = -35 \equiv 1 \pmod{12}$. For input $6, 12$, the GCD is 6 and the correct output is `NONE`.

Why normalize before extended Euclid?

The original a may have magnitude 10^{18} , but only its residue matters for inversion. After normalization, both inputs to `extendedGcd` are at most $10^9 + 7$. This keeps the Euclidean coefficient calculation within its stated contract while preserving the inverse problem.

A coarse bound using the usual Euclidean coefficient bounds makes the coefficient products comfortably fit in `long long` for these inputs. Arbitrary full-width signed inputs require a separate arithmetic analysis; simply removing the printed limit is not justified.

An inverse is cheaply checkable

For a claimed inverse x , verify $0 \leq x < m$ and $ax \bmod m = 1$ after normalizing a . Under the core modulus bound, the product is safe. This is useful for debugging and for accepting any equivalent coefficient witness returned by extended Euclid.

Pause and predict: 21

What should the inverse helper report for $a = 0, m = 13$? Is a stored result of zero a valid substitute for failure?

Check your prediction. It must report failure because the GCD is 13. Zero is not an inverse: $0 \cdot 0 \bmod 13 = 0$, not 1. Keep failure separate from a numeric answer.

6.22 A prime modulus permits the Fermat inverse formula

Fermat's little theorem states that if p is prime and a is not divisible by p , then

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

Consequently

$$a^{-1} \equiv a^{p-2} \pmod{p} \quad \text{when } p \text{ is prime and } a \not\equiv 0 \pmod{p}.$$

These hypotheses belong beside the formula every time it is used.^[6-9]

A short proof through rearranged nonzero residues

Multiply each of the nonzero residues $1, 2, \dots, p-1$ by a modulo p . None becomes zero. Two become equal only if $p \mid a(i-j)$; since p does not divide a , it must divide $i-j$, which forces $i=j$ in this range. The products therefore rearrange the same nonzero residues.

Multiplying them gives $a^{p-1}(p-1)! \equiv (p-1)! \pmod{p}$. The factorial contains no factor p , so it is coprime to p and can be cancelled. This leaves the theorem. Factoring out one a gives the inverse formula.

Task. Read $|a| \leq 10^{18}$ and a *guaranteed prime* $2 \leq p \leq 10^9 + 7$. Print the inverse or NONE if a is divisible by p .

Complete example | C++23

```
// |a| <= 10^18; prime is guaranteed prime and 2 <= prime <= 1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, prime = 0;
    std::cin >> a >> prime;
    a = normalize(a, prime);
    if (a == 0) std::cout << "NONE" << '\n';
    else std::cout << modPower(a, prime - 2, prime) << '\n';
}
```

Sample input	Sample output
5 13	8

Small prime, composite modulus, and false converses

For $p = 2$, the exponent is zero. The only nonzero residue is 1, whose inverse is indeed 1. The code checks the zero residue before using the exponentiation formula.

For a composite modulus, a^{m-2} is not a general inverse formula. With $a = 3, m = 8$, the inverse exists and equals 3, but $3^{8-2} \bmod 8 = 1$. The general extended-Euclid helper handles this case correctly.

The theorem does not say that a successful power test proves primality. For example, $2^{340} \equiv 1 \pmod{341}$ even though $341 = 11 \cdot 31$. A single congruence is not a replacement for a justified primality test.

When may an exponent be reduced?

If $a \not\equiv 0 \pmod{p}$ and p is prime, write $e = q(p-1) + r$. The theorem gives

$$a^e = (a^{p-1})^q a^r \equiv a^r \pmod{p}.$$

Thus reduction modulo $p-1$ is justified in that situation. It is not reduction modulo p .

If the base is zero modulo p , retain whether the original exponent is zero or positive. For $p = 7$, $7^6 \bmod 7 = 0$, but reducing the exponent to $6 \bmod 6 = 0$ and applying the empty-product convention would give 1. The missing condition changes the answer.

Pause and predict: 22

Can you use $a^{m-2} \bmod m$ for every nonzero a when $m = 12$? What should you use instead when a is coprime to 12?

Check your prediction. No. The formula requires a prime modulus. For a coprime value under modulus 12, extended Euclid computes a valid inverse without requiring factorization or primality.

6.23 A modular quotient is multiplication by an existing inverse

When b has an inverse modulo m , define the unique modular quotient by

$$\frac{a}{b} \pmod{m} := (a \bmod m)b^{-1} \bmod m.$$

It solves $bx \equiv a \pmod{m}$. It is not ordinary integer truncation. For example, the modular quotient of 8 by 3 under modulus 11 is 10 because $3 \cdot 10 \equiv 8 \pmod{11}$.

Task. Read $|a|, |b| \leq 10^{18}$ and $2 \leq m \leq 10^9 + 7$. Print this unique modular quotient when b is invertible; otherwise print NONE.

Complete example | C++23

```
// |top|, |bottom| <= 10^18; 2 <= mod <= 1000000007. Require a unique modular
// quotient.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long extendedGcd(long long a, long long b,
                     long long& x, long long& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long nextX = 0, nextY = 0;
    long long g = extendedGcd(b, a % b, nextX, nextY);
    x = nextY;
    y = nextX - (a / b) * nextY;
    return g;
}
bool inverse(long long a, long long mod, long long& result) {
    a = normalize(a, mod);
    long long x = 0, y = 0;
    if (extendedGcd(a, mod, x, y) != 1) return false;
    result = normalize(x, mod);
    return true;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long top = 0, bottom = 0, mod = 0, inv = 0;
    std::cin >> top >> bottom >> mod;
    if (!inverse(bottom, mod, inv)) std::cout << "NONE" << '\n';
    else std::cout << normalize(top, mod) * inv % mod << '\n';
}
```

Sample input

8 3 11

Sample output

10

Failure here does not always mean an equation has no solution

If b is not invertible, the quotient operation is not defined for every numerator. The particular congruence $bx \equiv a \pmod{m}$ might still have several solutions. For instance, $2x \equiv 2 \pmod{6}$ has two solutions, but no unique division-by-2 operation under modulus 6. The next section solves the equation rather than pretending there is an inverse.

Pause and predict: 23

The exact integer $10/2$ exists. Does that imply 2 has an inverse modulo 8?

Check your prediction. No. Exact integer divisibility and modular invertibility are different properties. The exact quotient is 5, but $\gcd(2, 8) = 2$, so 2 has no inverse modulo 8.

6.24 Linear congruences: zero, one, or several residue solutions

Consider

$$ax \equiv b \pmod{m}, \quad m \geq 2.$$

Let $g = \gcd(a, m)$. The equation requires $ax + my = b$ for some integer y , so g must divide b . If it does not, no solution exists.

Reduce the equation and its modulus together

When $g \mid b$, divide the *exact* equation by g :

$$\frac{a}{g}x \equiv \frac{b}{g} \pmod{m/g}.$$

The reduced coefficient and modulus are coprime. If $m/g > 1$, invert the reduced coefficient and obtain one canonical solution x_0 modulo m/g .

Within the original residue interval $[0, m)$, all solutions are

$$x_0, x_0 + m/g, x_0 + 2m/g, \dots, x_0 + (g-1)m/g.$$

There are exactly g of them. Each satisfies the reduced congruence; every solution must belong to that same reduced residue class.

Trace a noninvertible coefficient that still has solutions

For $6x \equiv 8 \pmod{14}$, the GCD is 2 and it divides 8. The reduced equation is $3x \equiv 4 \pmod{7}$. Since the inverse of 3 modulo 7 is 5, $x_0 = 4 \cdot 5 \pmod{7} = 6$.

The original solutions are 6 and $6 + 7 = 13$. Check both in the original equation: $36 \pmod{14} = 8$ and $78 \pmod{14} = 8$.

When the reduced modulus is 1

There is then no inversion task to perform. Once $g \mid b$ holds, every original residue is a solution, and the first is 0. For example, $0x \equiv 0 \pmod{12}$ has all 12 residue solutions, while $0x \equiv 1 \pmod{12}$ has none.

Pause and predict: 24

How many solutions in $[0, 6)$ satisfy $2x \equiv 2 \pmod{6}$? Which residues are they?

Check your prediction. The GCD is 2, so there are two solutions. Reduction gives $x \equiv 1 \pmod{3}$, producing 1 and 4. Cancelling 2 without reducing the modulus would miss one.

Return a compact description instead of printing every solution

Task. Read $0 \leq a, b \leq 10^9$ and $2 \leq m \leq 10^9 + 7$. Print the smallest canonical solution and the number of solutions, or NONE. This avoids potentially printing a billion residues.

Complete example | C++23

```
// 0 <= a, b <= 10^9; 2 <= mod <= 1000000007. Output smallest solution and
// count, or NONE.
#include <iostream>
#include <numeric>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long extendedGcd(long long a, long long b,
                     long long& x, long long& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long nextX = 0, nextY = 0;
    long long g = extendedGcd(b, a % b, nextX, nextY);
    x = nextY;
    y = nextX - (a / b) * nextY;
    return g;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0, mod = 0;
    std::cin >> a >> b >> mod;
    long long g = std::gcd(a, mod);
    if (b % g != 0) {
        std::cout << "NONE" << '\n';
        return 0;
    }
    long long reducedMod = mod / g;
    long long first = 0;
    if (reducedMod > 1) {
        long long x = 0, y = 0;
        extendedGcd(a / g, reducedMod, x, y);
        first = normalize(b / g, reducedMod)
            * normalize(x, reducedMod) % reducedMod;
    }
    std::cout << first << ' ' << g << '\n';
}
```

Sample input

6 8 14

Sample output

6 2

The computation takes logarithmic Euclidean time and logarithmic recursive stack space. If a different task asks to print all g residues, its time must also include $O(g)$ output work. Practice 17 uses a smaller modulus and asks for that enumeration.

6.25 Modular combinations: factorial tables need invertible denominators

Session 4 derived

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}.$$

To evaluate this formula modulo a **prime** p , assume $0 \leq r \leq n < p$. None of the factorial factors is divisible by p , so all three factorial residues are nonzero and invertible.

Define

$$F[i] = i! \bmod p, \quad I[i] = (i!)^{-1} \bmod p.$$

Then each valid query is answered by

$$\binom{n}{r} \bmod p = F[n]I[r]I[n-r] \bmod p.$$

Reduce between multiplications so a triple product is never constructed.

Build all inverse factorials using one exponentiation

First compute $F[0] = 1$ and $F[i] = F[i-1]i \bmod p$ through a limit $N < p$. Compute $I[N] = F[N]^{p-2} \bmod p$ once.

Because $i! = i(i-1)!$, their inverses satisfy

$$I[i-1] \equiv iI[i] \pmod{p}.$$

Walk *backwards* from N to 1 using that recurrence. There is no need to call binary power separately for every table entry.

i	0	1	2	3	4	5
$F[i] \pmod{7}$	1	1	2	6	3	1
$I[i] \pmod{7}$	1	1	4	6	5	1

For example, $\binom{5}{2} \bmod 7 = 1 \cdot 4 \cdot 6 \bmod 7 = 3$, matching $10 \bmod 7$.

Pause and predict: 25

Why does computing $I[N]$ once allow every smaller inverse factorial to be recovered?

Check your prediction. The exact identity $i! = i(i-1)!$ gives $I[i-1] = iI[i]$ modulo the prime. A backward multiplication therefore removes one factorial factor without performing a new inversion.

A many-query implementation

Task. Read $0 \leq N \leq 200000$ and $0 \leq q \leq 200000$. Each query has $0 \leq n \leq N$ and $-1 \leq r \leq N + 1$. Print $\binom{n}{r} \bmod 1000000007$; print 0 for $r < 0$ or $r > n$.

Complete example | C++23

```
// 0 <= limit <= 200000; 0 <= q <= 200000; 0 <= n <= limit; -1 <= r <=
// limit+1. Modulus 1000000007.
#include <iostream>
#include <vector>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int limit = 0, queries = 0;
    std::cin >> limit >> queries;
    const long long mod = 1000000007;
    std::vector<long long> fact(limit + 1, 1), invFact(limit + 1, 1);
    for (int i = 1; i <= limit; ++i)
        fact[i] = fact[i - 1] * i % mod;
    invFact[limit] = modPower(fact[limit], mod - 2, mod);
    for (int i = limit; i >= 1; --i)
        invFact[i - 1] = invFact[i] * i % mod;
    while (queries--) {
        int n = 0, r = 0;
        std::cin >> n >> r;
        if (r < 0 || r > n) std::cout << 0 << '\n';
        else std::cout << fact[n] * invFact[r] % mod
                      * invFact[n - r] % mod << '\n';
    }
}
```

Sample input

```
10 4
5 2
0 0
8 9
10 5
```

Sample output

```
10
1
0
252
```

Preparation takes $O(N + \log p)$ time and $O(N)$ storage. Each query costs $O(1)$, so total time is $O(N + q + \log p)$. The input bound guarantees $N < p$. The check for an impossible r happens *before* indexing either factorial table.

What breaks at or beyond the prime

If $n \geq p$, then $n! \equiv 0 \pmod{p}$, and zero has no inverse. The simple table method above is not valid through such an n .

The combination itself need not be zero. For example, $\binom{7}{7} = 1$, even though $7! \bmod 7 = 0$. The zero factorial representation has lost information about powers of the prime that cancel in the exact quotient.

A composite modulus creates another obstruction: a nonzero factorial residue may still share a factor with the modulus. Under modulus 8, $2!$ is 2, which is nonzero but not invertible. Do not use prime-factorial code merely because the residue is nonzero.

Choose a method from both constraints and modulus

Pascal's identity uses addition only, so it works modulo any positive modulus:

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}.$$

It is a useful alternative for smaller limits or a composite modulus. Large- n methods beyond the prime require additional techniques; they are not hidden features of the simple factorial table.

For an unblocked grid with r rows and c columns, a down/right route has $r - 1$ down moves among $r + c - 2$ total moves. The count is $\binom{r+c-2}{r-1}$. Check that the required factorial limit is below the prime before applying the modular formula.

Pause and predict: 26

A prime-modulus table was built through $N = 20$ with $p = 7$. Why is a zero inverse-factorial entry not a valid answer for queries such as $\binom{7}{7}$?

Check your prediction. The table violated $N < p$. Once the factorial contains the factor 7, its residue is zero and has no inverse. The exact combination can still be nonzero; $\binom{7}{7} = 1$.

6.26 GCD summaries: combine a prefix and suffix instead of subtracting

Not every prefix summary supports cancellation. A prefix sum supports subtraction, but a prefix GCD does not have an inverse operation that recovers the GCD of an arbitrary range.

For example, arrays $[6, 10]$ and $[6, 14]$ have the same prefix GCDs $[0, 6, 2]$, yet the GCD of their final single element is 10 in one case and 14 in the other. No formula based only on those identical prefixes can recover both different answers.

Removing one position needs a different combination

To remove a_i and combine all other values, summarize both sides:

$$P[i] = \gcd(a_0, \dots, a_{i-1}), \quad S[i] = \gcd(a_i, \dots, a_{n-1}).$$

Use $P[0] = 0$ and $S[n] = 0$. After removing position i , the answer is

$$\gcd(P[i], S[i + 1]).$$

No subtraction or inverse is required; GCD is associative over the surviving values.

Task. Read $2 \leq n \leq 200000$ values between 0 and 10^9 . Delete exactly one entry and print the largest possible GCD of the remaining entries.

Complete example | C++23

```
// 2 <= n <= 200000; 0 <= values <= 10^9.
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    std::vector<long long> a(n), pref(n + 1, 0), suff(n + 1, 0);
    for (long long& value : a) std::cin >> value;
    for (int i = 0; i < n; ++i)
        pref[i + 1] = std::gcd(pref[i], a[i]);
    for (int i = n - 1; i >= 0; --i)
        suff[i] = std::gcd(suff[i + 1], a[i]);
    long long answer = 0;
    for (int i = 0; i < n; ++i)
        answer = std::max(answer, std::gcd(pref[i], suff[i + 1]));
    std::cout << answer << '\n';
}
```

Sample input

Sample output

3

6

12 15 18

Trace the two summaries

For $a = [12, 15, 18]$,

$$P = [0, 12, 3, 3], \quad S = [3, 3, 18, 0].$$

Deleting positions 0, 1, and 2 gives candidate GCDs

$$\gcd(0, 3) = 3, \quad \gcd(12, 18) = 6, \quad \gcd(3, 0) = 3.$$

The maximum is 6. The two padded zeros handle deletion at an endpoint without an out-of-range access.

Building both tables and evaluating all candidates requires $O(n)$ GCD operations. With Euclidean implementations and largest value A , $O(n(1 + \log(A + 1)))$ is a safe total arithmetic bound. Storage is $O(n)$ for the input and two summaries.

A summary must match the operation you need

Prefix sums let you cancel an earlier prefix. Prefix GCDs let you combine already summarized groups, but not undo a group. This distinction is another example of the chapter's central question: what information does a transformation preserve?

Pause and predict: 27

Why are $P[0]$ and $S[n]$ initialized to 0 rather than 1?

Check your prediction. A missing side should not change the other side's GCD. Zero is the identity because $\gcd(0, x) = x$. Using 1 would force endpoint candidates to become 1 even when the surviving values share a larger divisor.

6.27 Debug the conditions, arithmetic, and total workload

Ask these questions before changing the formula

Symptom	Question to investigate
Negative printed residue	Was a C++ negative remainder normalized?
Wrong answer for a huge product	Did multiplication overflow before the remainder operation?
Inverse output is inconsistent	Are the value and modulus coprime? Was a prime-only formula used for a composite modulus?
Wrong result for exponent zero	Was the identity initialized as $1 \bmod m$? What convention does the problem use for 0^0 ?
LCM unexpectedly shrinks	Were values reduced modulo something, or did an exact multiplication overflow?
A modular combination is always zero	Did the factorial table reach the prime or use a noninvertible denominator?
Correct single query, wrong batch	Was state reset per query, and was all input consumed?

Count preparation once and helper work honestly

For q modular-power queries, total time is the sum of the logarithmic exponent costs, not the cost of one power. For q general inverses, count q Euclidean chains. For factorial-based combinations, count the forward table, the one initial inverse, the backward table, and the q outputs.

A table depends on its modulus. A factorial table prepared modulo one prime cannot be reused unchanged under another prime. The same applies to prefix residues and inverse values.

Use a tiny independent checker

For small m , test every possible inverse candidate. For small exponents, multiply directly using a safe small bound. For a claimed Bezout witness, substitute it into the exact equation. For a linear congruence, enumerate all residues and compare the solution set. These checks exercise the model independently of the optimized code.

Pause and predict: 28

A function computes $a^{m-2} \bmod m$ and its outputs match several random samples. What is still missing before calling it an inverse routine?

Check your prediction. Its assumptions and proof. The formula requires a prime modulus and a nonzero residue. Composite inputs or noninvertible values can fail even when many samples happen to pass.

6.28 Optional extension: Euler's totient connects factors and residues

For a positive integer n , Euler's totient $\varphi(n)$ counts the integers in $1, 2, \dots, n$ that are coprime to n . Thus $\varphi(1) = 1$. For $n = 12$, the counted integers are 1, 5, 7, 11, so $\varphi(12) = 4$.

Count by excluding multiples of each prime factor

If the distinct prime factors of n are $p_1, \dots, p_k$, a number is coprime to n exactly when none of those primes divides it. Inclusion-exclusion from Session 4 gives

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right).$$

Each intersection size is exactly n divided by the product of the selected distinct primes, because that product divides n . Expanding the product therefore reproduces the alternating inclusion-exclusion terms. Repeated copies of the same prime do not create additional exclusion rules.

For $36 = 2^2 \cdot 3^2$, the result is $36(1 - 1/2)(1 - 1/3) = 12$. Implement this using integer updates `result -= result / p`, once per distinct prime. Do not approximate the fractions with floating point.

Complete example | C++23

```
// 1 <= n <= 10^9; phi(1)=1 by convention.
#include <iostream>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long n = 0;
    std::cin >> n;
    long long remaining = n, answer = n;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        answer -= answer / p;
        while (remaining % p == 0) remaining /= p;
    }
    if (remaining > 1) answer -= answer / remaining;
    std::cout << answer << '\n';
}
```

Sample input

36

Sample output

12

Why the integer updates remain exact

Before processing a new prime p , the current result is n divided by some other distinct primes, then multiplied by integers. None of the divisions removed the factor p , so the result is still divisible by p . The update subtracts exactly one p th of that result. Trial factorization takes $O(\sqrt{n})$ time in the worst case and $O(1)$ auxiliary space here; the input limit is $n \leq 10^9$.

Euler's theorem and its essential condition

For $m \geq 2$ and $\gcd(a, m) = 1$, Euler's theorem states

$$a^{\varphi(m)} \equiv 1 \pmod{m}.$$

It generalizes Fermat's theorem, since $\varphi(p) = p - 1$ for a prime p . The coprimality condition is not optional. The theorem and its use for inverse powers are developed in the MIT number-theory reading.^[6-9]

To see the mechanism, consider all invertible residues modulo m . Multiplication by an invertible a rearranges them: two products cannot coincide unless their original residues coincide. Multiplying all the rearranged residues gives the original product times $a^{\varphi(m)}$. That original product is invertible, so it can be cancelled, leaving the claimed congruence.

Consequently $a^{\varphi(m)-1}$ is an inverse when a and m are coprime. This does not automatically make it the best implementation: first finding $\varphi(m)$ may require factorization. Extended Euclid finds an inverse without factoring the modulus.

Do not reduce an exponent without proving a period

Modulo 8, $\varphi(8) = 4$. However, $2^4 \bmod 8 = 0$, whereas $2^{4 \bmod 4} \bmod 8 = 1$. The base 2 is not coprime to 8, so Euler's exponent reduction is not available. A correct modular-power loop works without any such reduction.

6.29 Optional extension: modular multiplication when the modulus is large

Our core bound $m \leq 10^9 + 7$ makes a product of two normalized residues fit in `long long`. A modulus near $9 \cdot 10^{18}$ does not. Taking a remainder after an overflowing multiplication does not repair the calculation.

First make addition safe

Assume $0 \leq a, b < m \leq 9 \cdot 10^{18}$. Instead of forming $a + b$ unconditionally, compare a with $m - b$.

$$(a + b) \bmod m = \begin{cases} a - (m - b), & a \geq m - b, \\ a + b, & a < m - b. \end{cases}$$

In the first branch, the subtraction is nonnegative and smaller than m . In the second branch, $a + b < m$, so the addition fits. Neither branch evaluates an overflowing sum.

Then multiply by repeated doubling

Process the binary digits of the multiplier. Add the current value when its bit is set, double the value with safe addition, and halve the remaining multiplier. This is the same decomposition as binary exponentiation, but addition replaces multiplication.

Complete example | C++23

```
// 0 <= a, b <= 9*10^18; 1 <= mod <= 9*10^18.
#include <iostream>
// Preconditions: 0 <= a, b < mod <= 9*10^18.
long long addMod(long long a, long long b, long long mod) {
    if (a >= mod - b) return a - (mod - b);
    return a + b;
}
long long multiplyMod(long long a, long long b, long long mod) {
    a %= mod;
    b %= mod;
    long long result = 0;
    while (b > 0) {
        if (b % 2 == 1) result = addMod(result, a, mod);
        a = addMod(a, a, mod);
        b /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0, mod = 0;
    std::cin >> a >> b >> mod;
    std::cout << multiplyMod(a, b, mod) << '\n';
}
```


6.30 Optional extension: an exponent too long for an integer variable

Suppose the exponent is a decimal string with hundreds of thousands of digits. Parsing it into `long` is not possible. Reducing it modulo $m - 1$ is not a general solution either: the modulus may be composite and the base may not be invertible.

Process a decimal prefix directly

If the processed exponent prefix represents E and the next digit is d , the new exponent is $10E + d$. Therefore

$$a^{10E+d} = (a^E)^{10} \cdot a^d.$$

Maintain only $r = a^E \bmod m$. Replace it with $r^{10}a^d \bmod m$ using the ordinary modular-power helper. This identity holds without primality or coprimality assumptions.

Trace the exponent, not the enormous power

For base 2, modulus 1000, and exponent string 123:

Digit	Prefix exponent	Stored residue	Update
Before input	0	1	Empty exponent prefix.
1	1	2	$1^{10} \cdot 2^1 \bmod 1000$.
2	12	96	$2^{10} \cdot 2^2 \bmod 1000$.
3	123	608	$96^{10} \cdot 2^3 \bmod 1000$.

Leading zeros leave the numerical exponent unchanged. The program starts with $1 \bmod m$, so an all-zero exponent also works when $m = 1$. As in the core power helper, exponent zero denotes the empty product, even when the base is zero.

The helper powers use exponents at most 10, so each digit requires a constant number of fixed-width modular operations. For L decimal digits, the time is $O(L)$. The stored input string uses $O(L)$ space; the arithmetic state is $O(1)$.

A brute-force oracle for short exponent strings can convert them to an ordinary integer and compute the result independently. Large tests should include all zeros, many leading zeros, a non-coprime base, and modulus 1.

A complete decimal-exponent evaluator

Complete example | C++23

```
// |base| <= 10^18; 1..200000 exponent digits; 1 <= mod <= 1000000007.
#include <iostream>
#include <string>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}
long long decimalPower(long long base, const std::string& digits,
                      long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    for (char ch : digits) {
        int digit = ch - '0';
        result = modPower(result, 10, mod)
            * modPower(base, digit, mod) % mod;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long base = 0, mod = 0;
    std::string exponent;
    std::cin >> base >> exponent >> mod;
    std::cout << decimalPower(base, exponent, mod) << '\n';
}
```

Sample input

2 00123 1000

Sample output

608

The helper normalizes the base under the current modulus, then reads the exponent's characters without converting the full exponent to a machine integer. Each character is known to be a decimal digit from the input contract. Save this file independently; it includes every helper it calls.

The sample's leading zeros are intentionally preserved in the input. They change neither the exponent value nor the answer. Changing the modulus to a composite number does not invalidate the recurrence.

6.31 Optional extension: combine two coprime congruences

We want the unique x in $[0, mn)$ satisfying

$$x \equiv a \pmod{m}, \quad x \equiv b \pmod{n},$$

where $m, n \geq 2$ and $\gcd(m, n) = 1$. This is the two-modulus coprime case of the Chinese remainder theorem. We can construct and prove the result using the inverse machinery already developed here.

Normalize a modulo m and b modulo n . Every solution of the first congruence has the form $x = a + mt$. Substituting into the second gives

$$mt \equiv b - a \pmod{n}.$$

The inverse of m modulo n exists, so choose the canonical residue

$$t = (b - a) m^{-1} \pmod{n}, \quad x = a + mt.$$

Because $0 \leq a < m$ and $0 \leq t < n$, the constructed x is already in $[0, mn)$. If two answers satisfied both congruences, their difference would be divisible by both coprime moduli and therefore by mn . Two distinct values in this interval cannot differ by mn , proving uniqueness.

Complete example | C++23

```
// 2 <= m,n <= 10^9; gcd(m,n)=1; |a|,|b| <= 10^18.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long extendedGcd(long long a, long long b,
                     long long& x, long long& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long nextX = 0, nextY = 0;
    long long g = extendedGcd(b, a % b, nextX, nextY);
    x = nextY;
    y = nextX - (a / b) * nextY;
    return g;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, m = 0, b = 0, n = 0;
    std::cin >> a >> m >> b >> n;
    a = normalize(a, m);
    b = normalize(b, n);
    long long x = 0, y = 0;
    extendedGcd(m, n, x, y); // The input promises gcd(m,n) == 1.
    long long step = normalize(b - a, n) * normalize(x, n) % n;
    std::cout << a + m * step << '\n';
}
```

Sample input	Sample output
2 5 3 7	17

Check the bounds before forming the combined answer

The example guarantees $m, n \leq 10^9$. The product $mn \leq 10^{18}$ and all products of normalized intermediate values fit in `long long`. With arbitrary 64-bit moduli, the combined modulus might not fit even if each individual modulus fits.

For $x \equiv 2 \pmod{5}$ and $x \equiv 3 \pmod{7}$, the inverse of 5 modulo 7 is 3. Hence $t = (3 - 2) \cdot 3 \pmod{7} = 3$ and $x = 2 + 5 \cdot 3 = 17$. Direct checks give remainders 2 and 3.

Do not use this code for non-coprime moduli

The pair $x \equiv 1 \pmod{4}$, $x \equiv 2 \pmod{6}$ has no solution: the first requires an odd value and the second an even value. Some non-coprime systems do have solutions, but $m^{-1} \pmod{n}$ is unavailable. A generalized treatment needs a compatibility check and a different combined modulus; it is outside this optional example's contract.

Extended Euclid takes $O(1 + \log \max(m, n))$ time and recursive stack space. All other work is a constant number of bounded arithmetic operations. This is a useful example of reducing a new problem to a helper with a proved precondition.

6.32 Optional extension: a geometric sum without modular division

Consider

$$S_n = 1 + a + a^2 + \dots + a^{n-1}.$$

The familiar exact formula $(a^n - 1)/(a - 1)$ is dangerous modulo m : $a - 1$ may have no inverse. For $a = 3, m = 8$, the denominator would be 2, which is not invertible. We can avoid division entirely.

Return a power and a sum together

Let one call return $(P_n, S_n) = (a^n \bmod m, S_n \bmod m)$. At $n = 0$, return $(1 \bmod m, 0)$. For $n = 2h$, split the terms into two blocks:

$$P_{2h} = P_h^2, \quad S_{2h} = S_h + P_h S_h = S_h(1 + P_h).$$

For $n = 2h + 1$, first construct the even result, then append its next power:

$$S_{2h+1} = S_{2h} + P_{2h}, \quad P_{2h+1} = P_{2h}a.$$

All equalities can be reduced modulo m after safe arithmetic. The recursive call for h is made only once.

Complete example | C++23

```
// |base| <= 10^18; 0 <= n <= 10^18; 1 <= mod <= 1000000007.
#include <iostream>
#include <utility>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
// Return {base^n, 1 + base + ... + base^(n-1)}, modulo mod.
std::pair<long long, long long> powerAndSum(
    long long base, long long n, long long mod) {
    if (n == 0) return {1 % mod, 0};
    auto half = powerAndSum(base, n / 2, mod);
    long long power = half.first * half.first % mod;
    long long sum = half.second * (1 + half.first) % mod;
    if (n % 2 == 1) {
        sum = (sum + power) % mod;
        power = power * base % mod;
    }
    return {power, sum};
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long base = 0, n = 0, mod = 0;
    std::cin >> base >> n >> mod;
    auto answer = powerAndSum(normalize(base, mod), n, mod);
    std::cout << answer.first << ' ' << answer.second << '\n';
}
```

Sample input	Sample output
3 5 8	3 1

Why the two returned values are enough

The first block has h terms and sum S_h . Multiplying every term of that block by a^h shifts its exponents from $0, \dots, h-1$ to $h, \dots, 2h-1$. That gives the second block. When an extra term is needed, its exponent is exactly $2h$.

For $a = 3, n = 5, m = 8$, the exact power is 243 and the exact sum is 121, giving the output 3 1. No inverse is involved, so composite moduli, $a \equiv 1 \pmod{m}$, and modulus 1 are all valid.

n	$P_n \bmod 8$	$S_n \bmod 8$	Reason
0	1	0	Empty product and empty sum.
1	3	1	Append exponent 0.
2	1	4	$3^2 \bmod 8, 1 + 3$.
5	3	1	Double length 2, then append exponent 4.

Each level halves n and performs constant work, so both time and recursive stack space are $O(1 + \log n)$. The core modulus bound still matters: $S_h(1 + P_h)$ can be nearly $m(m-1)$, which fits under $m \leq 10^9 + 7$ but not for every signed 64-bit modulus.

What to take from the optional topics

You do not need to memorize five new templates. Reuse the same habits: factor only when factor information is needed; inspect a multiplication before evaluating it; derive a decimal-prefix update; turn simultaneous conditions into one equation; and return enough information to combine two halves without unsafe division.

6.33 Practice problems: state the arithmetic contract before coding

The tasks below use only the core material through Problem 17. Problems 18 to 20 are optional extensions. For each solution, identify the domain, derive the formula, check every division or multiplication, and explain the time and memory costs. All input follows the printed bounds; programs print no prompts.

Attempt the tasks before opening their hints or solutions. Samples are not a substitute for tests with zeros, negative inputs where allowed, non-invertible residues, and extreme bounds.

1. Cut two ropes into equal pieces

Two ropes have positive integer lengths a and b . Cut both completely into pieces of one common positive integer length, with no leftover rope. Make each piece as long as possible. Report that length and the total number of pieces.

Input and output. Read a, b with $1 \leq a, b \leq 10^{18}$. Print the longest piece length, then the total number of pieces.

Sample input	Sample output
84 126	42 5

Explained solution; try the hint first.

2. Write a fraction in canonical form

Reduce a rational number to lowest terms. Its denominator in the output must be positive. Represent zero as 0/1.

Input and output. Read numerator a and nonzero denominator b , with $|a|, |b| \leq 10^9$. Print the reduced numerator and denominator.

Sample input	Sample output
0 -15	0 1

Explained solution; try the hint first.

3. Count positive simultaneous signals

Two devices signal at time 0 and then every a and b time units respectively. Count the times when they signal together in the interval from 1 through T , inclusive. Time 0 must not be counted.

Input and output. Read a, b, T , where $1 \leq a, b \leq 10^9$ and $0 \leq T \leq 10^{18}$. Print the number of positive simultaneous times.

Sample input	Sample output
6 10 100	3

Explained solution; try the hint first.

4. Return to the starting position

Positions $0, 1, \dots, n-1$ lie on a circle. Start at position 0. Every move adds k modulo n . Find the smallest positive number of moves that returns you to 0. A move with $k = 0$ still counts as one move.

Input and output. Read n, k with $1 \leq n \leq 10^{18}$ and $0 \leq k \leq 10^{18}$. Print the minimum positive move count.

Sample input	Sample output
12 8	3

Explained solution; try the hint first.

5. A divisor shared by the entire list

Find the GCD of all entries in a nonempty list of nonnegative integers. The requested result for an all-zero list is 0.

Input and output. Read $1 \leq n \leq 200000$, then n integers $0 \leq a_i \leq 10^{18}$. Print one GCD.

Sample input	Sample output
3	12
0 36 60	

Explained solution; try the hint first.

6. Keep an LCM below a cap

Find the exact LCM of a nonempty list of positive integers, but only when it is at most a given limit. Otherwise report that it is over the limit. The exact LCM is not promised to fit in a machine integer.

Input and output. Read n, L , then n values. Bounds: $1 \leq n \leq 200000$ and $1 \leq L, a_i \leq 10^{18}$. Print the exact LCM if it is at most L ; otherwise print OVER.

Sample input	Sample output
3 50	OVER
6 10 21	

Explained solution; try the hint first.

7. Evaluate a signed expression modulo m

Compute the canonical residue of $ab + c$ modulo a positive modulus m . The exact product ab may not fit in `long long`.

Input and output. Read a, b, c, m with $|a|, |b|, |c| \leq 10^{18}$ and $1 \leq m \leq 10^9 + 7$. Print the answer in $[0, m)$.

Sample input	Sample output
-7 11 -5 13	9

Explained solution; try the hint first.

8. Remainder after joining two decimal strings

Join the decimal digits of s followed by the digits of t . Find the value of this concatenation modulo m . Preserve every digit, including leading zeros in either string.

Input and output. Read the nonempty digit strings s, t , then m . Each length is at most 100000; $1 \leq m \leq 10^9 + 7$. Print the canonical remainder.

Sample input	Sample output
12 003 97	72

Explained solution; try the hint first.

9. Answer independent power queries

For each query, compute $a^e \bmod m$. Exponent zero means the empty product, including when $a = 0$. Each query may use a different modulus.

Input and output. Read $0 \leq q \leq 100000$, then q triples a, e, m . Bounds: $|a| \leq 10^{18}$, $0 \leq e \leq 10^{18}$, $1 \leq m \leq 10^9 + 7$. Print one canonical residue per query.

Sample input	Sample output
3	24
2 10 1000	1
0 0 7	0
8 0 1	

Explained solution; try the hint first.

10. Count nonempty subsets modulo m

A set contains n distinct labelled objects. Count its nonempty subsets modulo m . Only the number of objects is given; do not generate the subsets.

Input and output. Read $0 \leq n \leq 10^{18}$ and $1 \leq m \leq 10^9 + 7$. Print the canonical residue of the count.

Sample input	Sample output
5 7	3

Explained solution; try the hint first.

11. Inverses under possibly composite moduli

For every pair a, m , find the inverse of a modulo m if it exists. Composite moduli are allowed. A failure must be reported rather than replaced by an arbitrary residue.

Input and output. Read $0 \leq q \leq 100000$, then q pairs. Bounds: $|a| \leq 10^{18}$ and $2 \leq m \leq 10^9 + 7$. Print the canonical inverse, or NONE, for each pair.

Sample input	Sample output
3	5
5 12	NONE
6 12	7
-3 11	

Explained solution; try the hint first.

12. Divide in a prime-modulus calculation

Under a guaranteed prime modulus p , compute a multiplied by the modular inverse of b . Report failure when b has no inverse. This is not ordinary integer division.

Input and output. Read a, b, p with $|a|, |b| \leq 10^{18}$ and prime $2 \leq p \leq 10^9 + 7$. Print the canonical quotient residue, or NONE.

Sample input
Sample output

8 3 11

10

 Explained solution; try the hint first.

13. Many committee-size queries

Each query describes n distinct people and asks for the number of unranked committees of size r , modulo 1000000007. Different committees are different sets of people. An impossible size has count 0.

Input and output. Read N, q , then q pairs n, r . Bounds: $0 \leq N, q \leq 200000$, $0 \leq n \leq N$, and $-1 \leq r \leq N + 1$. Print one answer per query.

Sample input
Sample output

6 3

20

6 3

0

4 -1

1

6 0

 Explained solution; try the hint first.

14. Count routes through an empty grid

In a grid with r rows and c columns, move from the top-left cell to the bottom-right cell using only moves one cell down or right. There are no blocked cells. Count the routes modulo 1000000007.

Input and output. Read $1 \leq r, c \leq 100000$. Print the route count.

Sample input
Sample output

3 4

10

 Explained solution; try the hint first.

15. Maximize the GCD after one deletion

Remove exactly one position from a list. Find the largest GCD obtainable from all remaining values. The remaining list is nonempty. Zeros are allowed.

Input and output. Read $2 \leq n \leq 200000$, then n integers $0 \leq a_i \leq 10^9$. Print the largest achievable GCD.

Sample input	Sample output
4 18 24 30 35	6

Explained solution; try the hint first.

16. Count subarrays with a divisible sum

Count nonempty contiguous subarrays whose sum is divisible by m . Values may be negative. Equal-valued subarrays at different positions count separately.

Input and output. Read n, m , then the n values. Bounds: $0 \leq n \leq 200000$, $1 \leq m \leq 200000$, and $|a_i| \leq 10^9$. Print the exact count, not its residue.

Sample input	Sample output
4 3 1 2 -3 4	4

Explained solution; try the hint first.

17. List every solution of a modular equation

Find every integer x with $0 \leq x < m$ satisfying $ax \equiv b \pmod{m}$. Print all solutions in increasing order; there may be more than one.

Input and output. Read a, b, m with $0 \leq a, b \leq 10^9$ and $2 \leq m \leq 100000$. Print the solutions separated by spaces, or NONE when there are none.

Sample input	Sample output
6 8 14	6 13

Explained solution; try the hint first.

18. Optional: a decimal exponent with many digits

Compute $a^E \bmod m$, where E is supplied as a nonempty decimal digit string. The modulus need not be prime and a need not be coprime to it. Leading zeros are allowed.

Input and output. Read a , the digit string for E , then m . Bounds: $|a| \leq 10^{18}$, the string length is at most 200000, and $1 \leq m \leq 10^9 + 7$. Print the residue. Exponent zero denotes the empty product.

Sample input	Sample output
6 00020 8	0

Explained solution; try the hint first.

19. Optional: recover a value from two remainders

Find the unique x in $[0, mn)$ satisfying $x \equiv a \pmod{m}$ and $x \equiv b \pmod{n}$. The two moduli are guaranteed to be coprime.

Input and output. Read a, m, b, n , where $|a|, |b| \leq 10^{18}$, $2 \leq m, n \leq 10^9$, and $\gcd(m, n) = 1$. Print x .

Sample input	Sample output
-1 4 2 9	11

Explained solution; try the hint first.

20. Optional: a long sum of successive powers

Compute $(1 + a + a^2 + \dots + a^{n-1}) \bmod m$. For $n = 0$, the sum is empty and equals 0. Do not assume that $a - 1$ has a modular inverse.

Input and output. Read a, n, m , with $|a| \leq 10^{18}$, $0 \leq n \leq 10^{18}$, and $1 \leq m \leq 10^9 + 7$. Print the canonical sum residue.

Sample input	Sample output
3 5 8	1

Explained solution; try the hint first.

6.34 Trace, correctness, and complexity drills

Work on paper first. State the relevant conditions, show the intermediate values, and explain why an alternative would fail. Drill 12 uses the optional extensions.

Drill 1: A Euclidean trace and a zero argument

Trace the successive pairs (a, b) for $\text{gcd}(252, 198)$. At which pair does the loop stop, and which variable is the answer? Then evaluate $\text{gcd}(0, 84)$ and the chapter's convention for $\text{gcd}(0, 0)$. Explain why checking $b \neq 0$ before taking a remainder is necessary.

Explained answer.

Drill 2: A whole-list GCD is not pairwise coprimality

The list is 6, 10, 15. Compute its whole-list GCD and the GCD of every pair. Is the statement “a list has GCD 1, therefore every pair is coprime” valid? Give a precise interpretation of the result of a GCD fold.

Explained answer.

Drill 3: Division first does not make every LCM representable

Take $a = 10^{18}$ and $b = 10^{18} - 1$. Their GCD is 1. Is $a / \gcd(a,b) * b$ safe in signed 64-bit arithmetic? The task only needs to know whether the LCM is at most $L = 10^{18}$. Show a check that answers without forming the product. Would the permanent OVER flag still be justified if later list values could be zero?

Explained answer.

Drill 4: Remainders and ordinary division

Evaluate the C++ expression `-17 % 7` and its canonical nonnegative normalization. Next compare these three expressions mathematically: $(8/2) \bmod 5$, integer division $(8 \bmod 5)/2$, and modular division $(8 \bmod 5) \cdot 2^{-1} \bmod 5$. Why do the last two mean different operations?

Explained answer.

Drill 5: The binary-power invariant in action

For the iterative modular-power algorithm, trace base 3, exponent 13, modulus 17. List the state (*result*, *base*, *exponent*) at the start of each iteration and the final result. How many loop iterations occur? What congruence must remain true after every update?

Explained answer.

Drill 6: Two identical recursive calls repeat the work

A correct recursive power helper calls itself once for $\lfloor e/2 \rfloor$, stores the result, and squares it. A modified version instead calls itself twice with that same smaller exponent. Compare their time and stack depth, ignoring compiler optimizations and counting the source-level recursive work. Use exponents $e = 2^k$ to make the call counts clear.

Explained answer.

Drill 7: Invertibility and a false prime-modulus shortcut

Which residues in $[0, 12)$ have modular inverses? Give their inverse residues. Then a programmer uses $3^{8-2} \bmod 8$ as the inverse of 3 modulo 8. Calculate that candidate, test it, and find the actual inverse.

Explained answer.

Drill 8: A coefficient is a witness, not a canonical residue

Extended Euclid returns $g = 1$, $x = -5$, $y = 8$ for inputs 35 and 22. Verify the exact identity. Which coefficient gives the inverse of 35 modulo 22, and what is its canonical value? Is a different pair of coefficients necessarily wrong?

Explained answer.

Drill 9: A modular equation can have several answers

Solve $6x \equiv 8 \pmod{14}$ for every x in $[0, 14)$. Compare it with $6x \equiv 7 \pmod{14}$. Finally, how many residues satisfy $0x \equiv 0 \pmod{14}$? Explain the special reduced-modulus case rather than asking an inverse helper to invert modulo 1.

Explained answer.

Drill 10: Factorial tables need an invertible denominator

A factorial-table implementation uses prime $p = 7$ but precomputes through $N = 7$. Explain why inverting `fact[7]` is invalid, even though $\binom{7}{7} = 1$. For a valid $N < p$, compare one backwards inverse-factorial precomputation with independently calling binary power for every inverse factorial.

Explained answer.

Drill 11: Two different prefix summaries

Arrays $[6, 10]$ and $[6, 14]$ have the same GCD of the first element and the same GCD of both elements. Can those two prefix-GCD values determine the GCD of the second element alone? Separately, trace the divisible-subarray counter for $[1, 2, -3, 4]$ modulo 3. Why is the residue-0 frequency initially 1, and why is the frequency increment after the answer update?

Explained answer.

Drill 12: Optional: choose a valid shortcut

For each calculation, justify a safe method and check the small result: (a) compute $6^{20} \bmod 8$ without assuming the base is invertible; (b) solve $x \equiv 2 \pmod{5}$, $x \equiv 3 \pmod{7}$ in $[0, 35)$; (c) compute $1 + 3 + 3^2 + 3^3 + 3^4$ modulo 8 without inverting 2. What assumption would break the tempting shortcut in (a) or (c)?

Explained answer.

6.35 Hints before solutions

A hint should help you make the next decision without replacing your attempt. Write the missing derivation, then implement and test it before looking at the complete program.

Problem 1: Cut two ropes into equal pieces

The common length must divide both rope lengths. Once you know that length, count the pieces from each rope separately.

Problem 2: Write a fraction in canonical form

Divide both values by the GCD of their magnitudes. Then decide whether both signs must change.

Problem 3: Count positive simultaneous signals

Find the first positive simultaneous time. Every later simultaneous time is its multiple.

Problem 4: Return to the starting position

After t moves the position is $tk \bmod n$. Divide n and k by their GCD.

Problem 5: A divisor shared by the entire list

The GCD accumulated so far is enough to process the next entry. What initial value leaves the first entry unchanged?

Problem 6: Keep an LCM below a cap

Before multiplying $current / \gcd(current, x)$ by x , compare the first factor with L/x .

Problem 7: Evaluate a signed expression modulo m

Normalize the operands before multiplying. Reduce the product before adding the last residue.

Problem 8: Remainder after joining two decimal strings

Use the same decimal-prefix recurrence for both strings. Do not reset the remainder before processing t .

Problem 9: Answer independent power queries

Call binary exponentiation with fresh local state. Its initial result is $1 \bmod m$, not an unconditional final answer of 1.

Problem 10: Count nonempty subsets modulo m

Each object has two choices. Remove exactly one empty subset and normalize the subtraction.

Problem 11: Inverses under possibly composite moduli

Normalize a first. Extended Euclid gives both the existence test and the coefficient that becomes the inverse.

Problem 12: Divide in a prime-modulus calculation

First test the denominator's residue. Only then may Fermat's inverse exponent be used.

Problem 13: Many committee-size queries

All queries share one prime and one maximum N . Precompute factorials and inverse factorials once.

Problem 14: Count routes through an empty grid

Every route has exactly $r - 1$ down moves and $c - 1$ right moves. Choose which move positions contain the down moves.

Problem 15: Maximize the GCD after one deletion

Precompute GCDs of prefixes and suffixes. The deleted position separates the retained values into two groups.

Problem 16: Count subarrays with a divisible sum

A subarray sum is divisible by m exactly when its two endpoint prefix sums have equal residues. Count earlier prefixes with the current residue.

Problem 17: List every solution of a modular equation

Let $g = \gcd(a, m)$. First test whether g divides b . Solve the reduced coprime equation, then restore all g residue classes.

Problem 18: Optional: a decimal exponent with many digits

Appending digit d changes the processed exponent from E to $10E + d$. Update the stored power directly.

Problem 19: Optional: recover a value from two remainders

After normalizing a , write $x = a + mt$ and solve one equation for t modulo n .

Problem 20: Optional: a long sum of successive powers

Have one recursive call return both a^n and the sum of n terms. Combine two equal-length blocks without division.

Hints for the reasoning drills

Drill 1

At each step replace the pair with $(b, a \bmod b)$, using the old values on the right.

Drill 2

A divisor shared by all entries is a stricter requirement than a divisor shared by just one selected pair.

Drill 3

Compare the first factor with L/b . Recall that the capped-list task explicitly requires positive entries.

Drill 4

C++ integer division truncates toward zero. Modulo 5, the inverse of 2 is 3.

Drill 5

Square the current base every iteration and halve the remaining exponent. Include the odd-exponent multiplication before the squaring.

Drill 6

For a single child, $T(e) = T(\lfloor e/2 \rfloor) + O(1)$. For two separately computed children, both subtrees must be counted.

Drill 7

List residues coprime to 12. For the modulus 8 calculation, check the defining product rather than trusting the exponent.

Drill 8

Reduce $35x + 22y = 1$ modulo 22. Adding a suitable multiple of the other coefficient's input can produce more witnesses.

Drill 9

Start with $g = \gcd(6, 14) = 2$. If the right side is divisible by g , divide the coefficient, right side, and modulus by it.

Drill 10

A factorial containing a factor p has residue zero modulo p . Count how many power calls each construction makes.

Drill 11

For GCD, compare the two possible suffix answers. For sums, write all five prefix residues, including the empty prefix.

Drill 12

Ordinary modular power needs no inverse. The two-modulus construction needs coprimality. The paired power-and-sum recurrence uses no division.

6.36 Explained practice solutions

These programs solve the exact input contracts in the practice section. Each is a complete independent program. Read the explanation, trace its sample, and then close the solution before writing your own version. A correct formula still needs safe arithmetic and the correct output convention.

For a library GCD call, the complexity bounds below use the usual Euclidean method analysed in this chapter. Numeric operations are counted in the fixed-width model. Recursion space is the maximum active depth, not the total number of calls.

Solution 1: Cut two ropes into equal pieces

Idea. Let $g = \gcd(a, b)$. Print g and $a/g + b/g$. Use a wide integer for both the rope lengths and the count.

Complete solution | C++23

```
// 1 <= a, b <= 1018.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0;
    std::cin >> a >> b;
    long long length = std::gcd(a, b);
    std::cout << length << ' ' << a / length + b / length << '\n';
}
```

Why it works. Every permitted length is a common divisor, so it is at most g . The length g divides both ropes and is attainable. The two exact quotients count all pieces without overlap.

Cost. Euclid takes $O(1 + \log \min(a, b))$ time and constant working space. The total count is at most $2 \cdot 10^{18}$, which fits the chosen type.

Check. Equal lengths, coprime lengths, one length equal to 1, and the largest lengths.

Solution 2: Write a fraction in canonical form

Idea. The library GCD handles these bounded signed values. Divide both terms by it; if the denominator is negative, negate both. The bound excludes the most-negative signed integer.

Complete solution | C++23

```
// |top|, |bottom| <= 10^9; bottom != 0.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long top = 0, bottom = 0;
    std::cin >> top >> bottom;
    long long g = std::gcd(top, bottom);
    top /= g;
    bottom /= g;
    if (bottom < 0) {
        top = -top;
        bottom = -bottom;
    }
    std::cout << top << ' ' << bottom << '\n';
}
```

Why it works. Dividing numerator and denominator by the same nonzero number preserves the fraction. Removing their GCD leaves coprime magnitudes. Flipping both signs preserves the value and makes the denominator positive.

Cost. $O(1 + \log \max(1, |a|, |b|))$ time with a Euclidean GCD and $O(1)$ working space.

Check. Zero numerator, negative denominator, two negative terms, and an already reduced fraction.

Solution 3: Count positive simultaneous signals

Idea. Compute $L = (a/\gcd(a,b))b$, then print $\lfloor T/L \rfloor$. Under the bounds, $L \leq 10^{18}$, so the exact period fits.

Complete solution | C++23

```
// 1 <= a, b <= 10^9; 0 <= end <= 10^18. Do not count time zero.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0, end = 0;
    std::cin >> a >> b >> end;
    long long period = (a / std::gcd(a, b)) * b;
    std::cout << end / period << '\n';
}
```

Why it works. A shared signal time is divisible by both periods, hence by their LCM. Conversely every multiple of their LCM is shared. The positive multiples at most T are $L, 2L, \dots, \lfloor T/L \rfloor L$.

Cost. $O(1 + \log \min(a, b))$ time and $O(1)$ space.

Check. $T = 0$, $T < L$, $T = L$, equal periods, and one period dividing the other.

Solution 4: Return to the starting position

Idea. Write $n = gn'$ and $k = gk'$, where $g = \gcd(n, k)$. The answer is $n' = n/g$; no simulation is needed.

Complete solution | C++23

```
// 1 <= n <= 10^18; 0 <= step <= 10^18; start at position 0.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long n = 0, step = 0;
    std::cin >> n >> step;
    std::cout << n / std::gcd(n, step) << '\n';
}
```

Why it works. Returning requires $n \mid tk$, equivalently $n' \mid tk'$. Since n' and k' are coprime, this is equivalent to $n' \mid t$. The smallest positive choice is n' . This also gives 1 when $k = 0$.

Cost. $O(1 + \log \max(n, k))$ time and $O(1)$ space.

Check. One position, zero step, step divisible by n , and a step coprime to n .

Solution 5: A divisor shared by the entire list

Idea. Start with result 0 and repeatedly replace it with $\gcd(\text{result}, a_i)$. Process values as they are read rather than storing the whole list.

Complete solution | C++23

```
// 1 <= n <= 200000; 0 <= values <= 10^18. All zero gives 0.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    long long answer = 0;
    for (int i = 0; i < n; ++i) {
        long long value = 0;
        std::cin >> value;
        answer = std::gcd(answer, value);
    }
    std::cout << answer << '\n';
}
```

Why it works. After each iteration, result is the GCD of exactly the values read so far. The identity $\gcd(0, x) = x$ starts the invariant, and combining the next value preserves precisely the common divisors.

Cost. A simple bound is $O(n(1 + \log V))$ time for values at most V , including input, and $O(1)$ working space.

Check. One value, all zeros, zeros between positive values, and a prefix whose GCD becomes 1.

Solution 6: Keep an LCM below a cap

Idea. Fold the LCM from 1. Set $r = \text{current} / \gcd(\text{current}, x)$. If $r > L/x$, mark the result as over the limit without multiplying. Continue consuming input.

Complete solution | C++23

```
// 1 <= n <= 200000; 1 <= values, limit <= 10^18.
#include <iostream>
#include <numeric>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    long long limit = 0;
    std::cin >> n >> limit;
    long long current = 1;
    bool tooLarge = false;
    for (int i = 0; i < n; ++i) {
        long long value = 0;
        std::cin >> value;
        if (tooLarge) continue; // Still consume the complete input.
        long long reduced = current / std::gcd(current, value);
        if (reduced > limit / value) tooLarge = true;
        else current = reduced * value;
    }
    if (tooLarge) std::cout << "OVER" << '\n';
    else std::cout << current << '\n';
}
```

Why it works. With positive integers, $rx > L$ exactly when $r > \lfloor L/x \rfloor$. A safe multiplication therefore produces the exact next LCM. Further positive entries cannot make an over-limit LCM smaller.

Cost. $O(n(1 + \log V))$ time is a simple bound; $V \leq 10^{18}$. Only a few scalars are stored.

Check. Limit 1, repeated entries, equality with the cap, and coprime values whose product would overflow.

Solution 7: Evaluate a signed expression modulo m

Idea. Reduce each signed operand to $[0, m)$. Multiply the first two, take its remainder, add the third, and reduce once more.

Complete solution | C++23

```
// |a|, |b|, |c| <= 10^18; 1 <= mod <= 1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0, c = 0, mod = 0;
    std::cin >> a >> b >> c >> mod;
    a = normalize(a, mod);
    b = normalize(b, mod);
    c = normalize(c, mod);
    std::cout << (a * b % mod + c) % mod << '\n';
}
```

Why it works. Replacing operands by congruent residues preserves multiplication and addition. The normalized product is at most $(m-1)^2$, and the final addition is below $2m$, so the code computes the intended residue safely.

Cost. $O(1)$ time and space in the fixed-width model.

Check. Negative terms, zero operands, modulus 1, and residues close to $m-1$.

Solution 8: Remainder after joining two decimal strings

Idea. Start with remainder 0. Process each digit of s , then each digit of t , using $(10r + d) \bmod m$. No full integer conversion is needed.

Complete solution | C++23

```
// Each string has 1..100000 decimal digits; 1 <= mod <= 1000000007.
#include <iostream>
#include <string>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::string first, second;
    long long mod = 0;
    std::cin >> first >> second >> mod;
    long long remainder = 0;
    for (char ch : first)
        remainder = (remainder * 10 + (ch - '0')) % mod;
    for (char ch : second)
        remainder = (remainder * 10 + (ch - '0')) % mod;
    std::cout << remainder << '\n';
}
```

Why it works. After every digit, the stored state is congruent to the numerical value of exactly the processed prefix. Continuing into t constructs the concatenation rather than adding the two original numbers.

Cost. $O(|s| + |t|)$ time; the input strings occupy $O(|s| + |t|)$ space, while the running arithmetic state is constant.

Check. Strings containing only zeros, t beginning with zeros, modulus 1, and maximum lengths.

Solution 9: Answer independent power queries

Idea. Normalize each base under that query's modulus, then repeatedly consume the exponent's lowest binary digit. Do not carry a result or modulus from the preceding query.

Complete solution | C++23

```
// 0 <= q <= 100000; |base| <= 10^18; 0 <= exponent <= 10^18; 1 <= mod <=
// 10000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int queries = 0;
    std::cin >> queries;
    while (queries--) {
        long long base = 0, exponent = 0, mod = 0;
        std::cin >> base >> exponent >> mod;
        std::cout << modPower(base, exponent, mod) << '\n';
    }
}
```

Why it works. The invariant $result \cdot base^{\text{remaining}} \equiv a^e \pmod{m}$ is established independently for every query and yields its result when the remaining exponent becomes zero.

Cost. $O(\sum_{i=1}^q (1 + \log(e_i + 1)))$ time and $O(1)$ auxiliary space when queries are processed as read.

Check. Empty query batch, zero exponent, negative base, modulus 1, and consecutive queries with different moduli.

Solution 10: Count nonempty subsets modulo m

Idea. Compute $2^n \bmod m$ using binary power, subtract 1, and normalize. The subtraction takes place after reduction without changing its final congruence.

Complete solution | C++23

```
// 0 <= n <= 10^18; 1 <= mod <= 1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long n = 0, mod = 0;
    std::cin >> n >> mod;
    std::cout << normalize(modPower(2, n, mod) - 1, mod) << '\n';
}
```

Why it works. There are 2^n subsets because each labelled object independently is selected or not. Exactly one outcome selects nothing, so the required count is $2^n - 1$.

Cost. $O(1 + \log(n + 1))$ time and $O(1)$ working space.

Check. $n = 0$ produces 0; modulus 1; and a power whose residue is 0 before subtracting 1.

Solution 11: Inverses under possibly composite moduli

Idea. For each query, run extended Euclid on normalized a and m . A GCD other than 1 means failure; otherwise normalize the coefficient of a .

Complete solution | C++23

```
// 0 <= q <= 100000; |a| <= 10^18; 2 <= mod <= 1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long extendedGcd(long long a, long long b,
                      long long& x, long long& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long nextX = 0, nextY = 0;
    long long g = extendedGcd(b, a % b, nextX, nextY);
    x = nextY;
    y = nextX - (a / b) * nextY;
    return g;
}
bool inverse(long long a, long long mod, long long& result) {
    a = normalize(a, mod);
    long long x = 0, y = 0;
    if (extendedGcd(a, mod, x, y) != 1) return false;
    result = normalize(x, mod);
    return true;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int q = 0;
    std::cin >> q;
    while (q--) {
        long long a = 0, mod = 0, answer = 0;
        std::cin >> a >> mod;
        if (inverse(a, mod, answer)) std::cout << answer << '\n';
        else std::cout << "NONE" << '\n';
    }
}
```

Why it works. The returned equation is $ax + my = g$. When $g = 1$, reducing it modulo m proves that x is an inverse. If $g > 1$, every combination is divisible by g , so no inverse can exist.

Cost. $O(\sum_i (1 + \log m_i))$ time; recursive stack space $O(1 + \log M)$, with M the largest modulus.

Check. A nonzero non-invertible value, a negative base, zero residue, and many changing moduli.

Solution 12: Divide in a prime-modulus calculation

Idea. Normalize both values. If the denominator is zero, print NONE; otherwise compute $b^{p-2} \bmod p$ and multiply it by the numerator residue.

Complete solution | C++23

```
// |top|, |bottom| <= 10^18; prime is guaranteed prime, 2..1000000007.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long top = 0, bottom = 0, prime = 0;
    std::cin >> top >> bottom >> prime;
    top = normalize(top, prime);
    bottom = normalize(bottom, prime);
    if (bottom == 0) std::cout << "NONE" << '\n';
    else std::cout << top * modPower(bottom, prime - 2, prime)
                    % prime << '\n';
}
```

Why it works. A nonzero residue under a prime modulus is coprime to that modulus. Fermat's theorem therefore makes b^{p-2} its inverse. Multiplication yields precisely the requested modular quotient.

Cost. $O(1 + \log p)$ time and $O(1)$ auxiliary space.

Check. Prime 2, denominator divisible by p , negative inputs, and zero numerator with invertible denominator.

Solution 13: Many committee-size queries

Idea. Build factorials up to N , invert only the last factorial by binary power, and fill inverse factorials backwards. Reject $r < 0$ or $r > n$ before indexing.

Complete solution | C++23

```
// 0 <= limit <= 200000; 0 <= q <= 200000; 0 <= n <= limit; -1 <= r <=
// limit+1.
#include <iostream>
#include <vector>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int limit = 0, queries = 0;
    std::cin >> limit >> queries;
    const long long mod = 1000000007;
    std::vector<long long> fact(limit + 1, 1), invFact(limit + 1, 1);
    for (int i = 1; i <= limit; ++i)
        fact[i] = fact[i - 1] * i % mod;
    invFact[limit] = modPower(fact[limit], mod - 2, mod);
    for (int i = limit; i >= 1; --i)
        invFact[i - 1] = invFact[i] * i % mod;
    while (queries--) {
        int n = 0, r = 0;
        std::cin >> n >> r;
        if (r < 0 || r > n) std::cout << 0 << '\n';
        else std::cout << fact[n] * invFact[r] % mod
                      * invFact[n - r] % mod << '\n';
    }
}
```

Why it works. Since $N < p$, every required factorial is invertible. The product $n! (r!)^{-1} ((n-r)!)^{-1}$ is the binomial coefficient modulo p . The backwards recurrence computes each needed inverse correctly.

Cost. $O(N + q + \log p)$ time and $O(N)$ table space; queries are not stored.

Check. $N = 0$, $r = 0$, $r = n$, invalid sizes, and repeated queries at the table limit.

Solution 14: Count routes through an empty grid

Idea. Compute $\binom{r+c-2}{r-1}$ with factorials and a modular inverse of the denominator. The largest factorial index is 199998, below the fixed prime.

Complete solution | C++23

```
// 1 <= rows, cols <= 100000; move only down or right, no blocked cells.
#include <iostream>
#include <vector>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int rows = 0, cols = 0;
    std::cin >> rows >> cols;
    int n = rows + cols - 2;
    const long long mod = 1000000007;
    std::vector<long long> fact(n + 1, 1);
    for (int i = 1; i <= n; ++i) fact[i] = fact[i - 1] * i % mod;
    long long denominator = fact[rows - 1] * fact[cols - 1] % mod;
    std::cout << fact[n] * modPower(denominator, mod - 2, mod)
               % mod << '\n';
}
```

Why it works. A route is uniquely determined by the positions of its down moves, and every such selection yields a valid route in an unblocked grid. The factorial quotient therefore counts each route exactly once.

Cost. $O(r + c + \log p)$ time and $O(r + c)$ table space for the printed implementation.

Check. A single cell has one empty route; one row; one column; and the largest rectangle.

Solution 15: Maximize the GCD after one deletion

Idea. For deletion at index i , combine the GCD before i with the GCD after i . Use padded zero entries for empty sides, and take the maximum over all positions.

Complete solution | C++23

```
// 2 <= n <= 200000; 0 <= values <= 10^9.
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    std::vector<long long> a(n), pref(n + 1, 0), suff(n + 1, 0);
    for (long long& value : a) std::cin >> value;
    for (int i = 0; i < n; ++i)
        pref[i + 1] = std::gcd(pref[i], a[i]);
    for (int i = n - 1; i >= 0; --i)
        suff[i] = std::gcd(suff[i + 1], a[i]);
    long long answer = 0;
    for (int i = 0; i < n; ++i)
        answer = std::max(answer, std::gcd(pref[i], suff[i + 1]));
    std::cout << answer << '\n';
}
```

Why it works. Those two groups contain exactly the retained entries. The GCD of their GCDs is therefore the GCD after that deletion. Checking every position includes an optimal deletion.

Cost. $O(n(1 + \log V))$ time is a simple bound for $V \leq 10^9$; the input and two summary arrays use $O(n)$ space.

Check. Two values, all zeros, deletion at either endpoint, and a single value that lowers every full-list GCD.

Solution 16: Count subarrays with a divisible sum

Idea. Initialize the frequency of residue 0 to 1 for the empty prefix. Read each value, update the normalized prefix residue, add its existing frequency to the answer, then increment that frequency.

Complete solution | C++23

```
// 0 <= n <= 200000; 1 <= mod <= 200000; |values| <= 10^9.
#include <iostream>
#include <vector>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, mod = 0;
    std::cin >> n >> mod;
    std::vector<long long> frequency(mod, 0);
    frequency[0] = 1;
    long long remainder = 0, answer = 0;
    for (int i = 0; i < n; ++i) {
        long long value = 0;
        std::cin >> value;
        remainder = normalize(remainder + value, mod);
        answer += frequency[remainder];
        ++frequency[remainder];
    }
    std::cout << answer << '\n';
}
```

Why it works. At a right endpoint, every earlier equal-residue prefix defines one valid nonempty subarray. Adding before incrementing excludes the current prefix paired with itself. Every subarray is counted once at its right endpoint.

Cost. $O(n + m)$ time including frequency initialization; $O(m)$ working space. The answer may reach $n(n + 1)/2$ and needs **long long**.

Check. Empty input, modulus 1, negative values, all zeros, and subarrays beginning at the first position.

Solution 17: List every solution of a modular equation

Idea. If $g \nmid b$, there is no solution. Otherwise solve $(a/g)x \equiv b/g \pmod{m/g}$ for the least residue x_0 , then print $x_0 + j(m/g)$ for $0 \leq j < g$.

Complete solution | C++23

```
// 0 <= a, b <= 10^9; 2 <= mod <= 100000. List all x in [0, mod), or NONE.
#include <iostream>
#include <numeric>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long extendedGcd(long long a, long long b,
                     long long& x, long long& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long nextX = 0, nextY = 0;
    long long g = extendedGcd(b, a % b, nextX, nextY);
    x = nextY;
    y = nextX - (a / b) * nextY;
    return g;
}
int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, b = 0, mod = 0;
    std::cin >> a >> b >> mod;
    long long g = std::gcd(a, mod);
    if (b % g != 0) {
        std::cout << "NONE" << '\n';
        return 0;
    }
    long long reducedMod = mod / g;
    long long first = 0;
    if (reducedMod > 1) {
        long long x = 0, y = 0;
        extendedGcd(a / g, reducedMod, x, y);
        first = normalize(b / g, reducedMod)
            * normalize(x, reducedMod) % reducedMod;
    }
    for (long long i = 0; i < g; ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << first + i * reducedMod;
    }
    std::cout << '\n';
}
```

Why it works. The reduced congruence is equivalent and has coprime coefficient and modulus. Its solutions differ by m/g , giving exactly g residues in $[0, m)$. If $m/g = 1$, start at 0 without inversion.

Cost. $O(1 + \log \max(a, m) + g)$ time, including output, and logarithmic recursive stack space. No solution list is stored.

Check. No solution, one solution, several solutions, and $a = b = 0$, when every residue is valid.

Solution 18: Optional: a decimal exponent with many digits

Idea. Maintain the power of the processed exponent prefix. For each digit d , replace r with $r^{10}a^d \bmod m$. Use only bounded-exponent calls to the core helper.

Complete solution | C++23

```
// |base| <= 10^18; 1..200000 exponent digits; 1 <= mod <= 1000000007.
#include <iostream>
#include <string>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long modPower(long long base, long long exponent,
                  long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) result = result * base % mod;
        base = base * base % mod;
        exponent /= 2;
    }
    return result;
}
long long decimalPower(long long base, const std::string& digits,
                      long long mod) {
    base = normalize(base, mod);
    long long result = 1 % mod;
    for (char ch : digits) {
        int digit = ch - '0';
        result = modPower(result, 10, mod)
            * modPower(base, digit, mod) % mod;
    }
    return result;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long base = 0, mod = 0;
    std::string exponent;
    std::cin >> base >> exponent >> mod;
    std::cout << decimalPower(base, exponent, mod) << '\n';
}
```

Why it works. The identity $a^{10E+d} = (a^E)^{10}a^d$ preserves the meaning of the stored residue for every new digit, regardless of coprimality. The initial empty prefix represents exponent zero.

Cost. $O(L)$ time for L digits and $O(L)$ space for the input string; the arithmetic state is constant.

Check. All-zero exponent, leading zeros, composite modulus, non-invertible base, and modulus 1.

Solution 19: Optional: recover a value from two remainders

Idea. Normalize the two residues. Find $m^{-1} \pmod n$ with extended Euclid, then set $t = (b - a)m^{-1} \pmod n$ and output $a + mt$.

Complete solution | C++23

```
// 2 <= m,n <= 10^9; gcd(m,n)=1; |a|,|b| <= 10^18.
#include <iostream>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
long long extendedGcd(long long a, long long b,
                     long long& x, long long& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long nextX = 0, nextY = 0;
    long long g = extendedGcd(b, a % b, nextX, nextY);
    x = nextY;
    y = nextX - (a / b) * nextY;
    return g;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long a = 0, m = 0, b = 0, n = 0;
    std::cin >> a >> m >> b >> n;
    a = normalize(a, m);
    b = normalize(b, n);
    long long x = 0, y = 0;
    extendedGcd(m, n, x, y); // The input promises gcd(m,n) == 1.
    long long step = normalize(b - a, n) * normalize(x, n) % n;
    std::cout << a + m * step << '\n';
}
```

Why it works. The construction satisfies the first congruence by its form and the second by the chosen t . Its range is $[0, mn)$. Any two solutions differ by a multiple of both coprime moduli and hence by mn , proving uniqueness there.

Cost. $O(1 + \log \max(m, n))$ time and recursive stack space. The bound $mn \leq 10^{18}$ makes the combined answer representable.

Check. Negative given residues, a zero result, one modulus larger than the other, and large coprime moduli.

Solution 20: Optional: a long sum of successive powers

Idea. Return a pair containing the power and the sum. Double a half-length pair with $P' = P^2$ and $S' = S(1 + P)$; for odd length append P' and multiply the power by a .

Complete solution | C++23

```
// |base| <= 10^18; 0 <= n <= 10^18; 1 <= mod <= 1000000007. Sum n terms
// from exponent 0.
#include <iostream>
#include <utility>
long long normalize(long long value, long long mod) {
    value %= mod;
    if (value < 0) value += mod;
    return value;
}
// Return {base^n, 1 + base + ... + base^(n-1)}, modulo mod.
std::pair<long long, long long> powerAndSum(
    long long base, long long n, long long mod) {
    if (n == 0) return {1 % mod, 0};
    auto half = powerAndSum(base, n / 2, mod);
    long long power = half.first * half.first % mod;
    long long sum = half.second * (1 + half.first) % mod;
    if (n % 2 == 1) {
        sum = (sum + power) % mod;
        power = power * base % mod;
    }
    return {power, sum};
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    long long base = 0, n = 0, mod = 0;
    std::cin >> base >> n >> mod;
    auto answer = powerAndSum(normalize(base, mod), n, mod);
    std::cout << answer.second << '\n';
}
```

Why it works. The second block is the first block multiplied by a^h , which shifts all its exponents by h . An odd final term has exponent $2h$. The base pair $(1 \bmod m, 0)$ represents exactly zero terms.

Cost. $O(1 + \log(n + 1))$ time and recursive stack space. All multiplications obey the core modulus bound.

Check. Zero terms, one term, base 0 or 1, a non-invertible $a - 1$, and modulus 1.

6.37 Explained trace and reasoning drills

Compare your argument with the reasoning below. A matching number is useful, but the input assumptions and the explanation of each step are equally important.

Drill 1: A Euclidean trace and a zero argument

The pairs are $(252, 198)$, $(198, 54)$, $(54, 36)$, $(36, 18)$, and $(18, 0)$. The loop stops at the last pair and returns $a = 18$, not the zero remainder. For $(0, 84)$, one step reaches $(84, 0)$, so the GCD is 84. The pair $(0, 0)$ stops immediately and returns the adopted value 0. Taking a remainder when the second operand is zero is undefined in C++; the guard is part of correctness, not merely an optimization.

Drill 2: A whole-list GCD is not pairwise coprimality

The whole-list GCD is $\gcd(\gcd(6, 10), 15) = \gcd(2, 15) = 1$. Yet the pairwise GCDs are $\gcd(6, 10) = 2$, $\gcd(6, 15) = 3$, and $\gcd(10, 15) = 5$. Thus none of these pairs is coprime. A fold returns the greatest nonnegative common divisor of every processed entry, with the zero conventions used in this chapter. It does not preserve enough information to answer every pairwise question.

Drill 3: Division first does not make every LCM representable

The exact LCM is $10^{18}(10^{18} - 1)$, which is far beyond the signed 64-bit range. Dividing first removes no factor here. Compute $r = a/g = a$ and compare $r > L/b$. The quotient L/b is 1, so the LCM is over the cap without multiplication. The permanent flag relies on positive inputs: adding positive values cannot reduce an LCM. If a later zero were allowed under the library zero convention, the final LCM could become 0, so that reasoning would no longer apply.

Drill 4: Remainders and ordinary division

C++ gives $-17/7 = -2$ and remainder -3 ; normalization adds 7 and produces 4. Exact division first gives $(8/2) \bmod 5 = 4$. Dividing the reduced numerator as an ordinary integer gives $3/2 = 1$. Modular division instead gives $3 \cdot 3 \bmod 5 = 4$. The modular inverse solves a multiplication equation; it does not mean truncating an integer quotient. Its existence also requires coprimality.

Drill 5: The binary-power invariant in action

The starting states are $(1, 3, 13)$, $(3, 9, 6)$, $(3, 13, 3)$, and $(5, 16, 1)$. The final state is $(12, 1, 0)$, so the answer is 12 after four iterations. Throughout the loop, $result \cdot base^{exponent} \equiv 3^{13} \pmod{17}$. The final base is not the answer; once the remaining exponent is zero, the accumulated result is the answer.

Drill 6: Two identical recursive calls repeat the work

With base case $e = 0$, the single-child version has $k + 2$ calls for $e = 2^k$, so its time and depth are $\Theta(1 + \log e)$. The repeated-child version has $2^{k+2} - 1 = 4e - 1$ calls under that same recurrence and base case, giving $\Theta(e)$ time. Its active depth is still only $k + 2$: the two subtrees are evaluated one after the other, not kept active side by side. A small stack does not imply little total work.

Drill 7: Invertibility and a false prime-modulus shortcut

The invertible residues modulo 12 are 1, 5, 7, 11, and each is its own inverse: their squares have remainder 1. The proposed modulo-8 candidate is $3^6 \bmod 8 = 1$. It fails because $3 \cdot 1 \bmod 8 = 3$. The actual inverse is 3 because $3 \cdot 3 \bmod 8 = 1$. Fermat's $p - 2$ exponent requires a prime modulus; the fact that an inverse exists under a composite modulus does not make that exponent valid.

Drill 8: A coefficient is a witness, not a canonical residue

The identity is $35(-5) + 22(8) = -175 + 176 = 1$. Reducing modulo 22 removes the second term, so $x = -5$ supplies the inverse; its canonical value is 17. Indeed $35 \cdot 17 \bmod 22 = 1$. The pair $(17, -27)$ is another valid witness because $35 \cdot 17 + 22(-27) = 1$. In general $(x + 22t, y - 35t)$ remains a witness for every integer t . A checker should test the identity and GCD, not require one particular pair.

Drill 9: A modular equation can have several answers

The first equation reduces to $3x \equiv 4 \pmod{7}$. Since $3^{-1} \equiv 5 \pmod{7}$, the least solution is $x_0 = 6$. Restore the two residues 6 and $6 + 7 = 13$. In the second equation, 2 does not divide 7, so no solution exists. For $0x \equiv 0 \pmod{14}$, all 14 residues satisfy the equation. Dividing by $g = 14$ gives reduced modulus 1, so set the base residue to 0 and enumerate all original residue classes.

Drill 10: Factorial tables need an invertible denominator

Here $7! \equiv 0 \pmod{7}$, and zero has no inverse. The exact factorial quotient for $\binom{7}{7}$ is meaningful over integers, but its numerator and denominator cannot be replaced by a division of zero residues. Under $N < p$, one inverse at the limit and a backwards loop take $O(N + \log p)$ time. Inverting each factorial separately takes $O((N + 1)(1 + \log p))$. Both use $O(N)$ table space, and each later valid binomial query takes constant time.

Drill 11: Two different prefix summaries

Both GCD-prefix pairs are $(6, 2)$, but the required suffix GCD is 10 for one array and 14 for the other. Thus no subtraction-like reconstruction from those two summaries is possible. The sum-prefix residues are 0, 1, 0, 0, 1. Their frequencies are three zeros and two ones, producing $\binom{3}{2} + \binom{2}{2} = 4$ nonempty divisible subarrays. During the scan, the answer increments are 0, 1, 2, 1. The initial zero represents subarrays beginning at the first element; incrementing after counting excludes a prefix paired with itself, which would be an empty subarray.

Drill 12: Optional: choose a valid shortcut

(a) Binary power gives 0. Reducing exponent 20 modulo $\varphi(8) = 4$ would incorrectly produce $6^0 \bmod 8 = 1$, because 6 is not coprime to 8. (b) Write $x = 2 + 5t$; $t \equiv 3 \pmod{7}$, so $x = 17$. (c) The exact sum is 121 and its residue is 1. The paired recurrence computes that residue directly. Dividing by $3 - 1 = 2$ modulo 8 is unavailable because $\gcd(2, 8) = 2$, not 1.

6.38 Quick reference, glossary, and readiness checklist

Use these pages for revision after studying the derivations. The condition beside a formula is part of the formula. In this section, m is a positive modulus, p is prime where explicitly stated, and $\text{norm}(x, m)$ is the canonical residue in $[0, m)$.

Exact common structure

Need	Rule	Check first
GCD	$\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$.	Take a remainder only when $b \neq 0$; normalize signs or use a valid library call.
Zero argument	$\text{gcd}(a, 0) = a $.	The chapter defines $\text{gcd}(0, 0) = 0$ for programming.
Positive LCM	$\text{lcm}(a, b) = (a / \text{gcd}(a, b))b$.	Divide first, but also prove the result fits. Handle zero before dividing.
Capped LCM	$r = a / \text{gcd}(a, b)$; reject when $r > \lfloor L/b \rfloor$.	$a, b, L > 0$; do not evaluate the product first.
Reduced fraction	Divide numerator and denominator by their GCD.	Denominator nonzero; output denominator positive.
Circle period	$n / \text{gcd}(n, k)$.	$n > 0$; answer is the smallest positive move count.

Modular operations and powers

Operation	Safe meaning or update
Normalize	Take $x \% m$; if negative, add m . Require $m > 0$.
Add/subtract	Normalize operands and normalize the final difference. Bound an addition before evaluating it.
Multiply	Multiply reduced wide operands only when their product fits. Core examples require $m \leq 10^9 + 7$.
Power	Start with $1 \bmod m$. For each binary exponent bit, optionally multiply the result, square the base, and halve the exponent.
Decimal number	$r \leftarrow (10r + d) \bmod m$ for each digit d .
Decimal exponent	$r \leftarrow r^{10^d} \bmod m$ for each digit d ; no prime-modulus assumption is needed.

Inverses, equations, and modular combinations

Need	Formula and conditions
Inverse existence	For $m \geq 2$, $a^{-1} \pmod{m}$ exists exactly when $\gcd(a, m) = 1$.
Extended Euclid	From child coefficients u, v for $(b, a \bmod b)$, return $x = v$, $y = u - \lfloor a/b \rfloor v$.
General inverse	Normalize a , find $ax + my = g$, reject $g \neq 1$, otherwise return $\text{norm}(x, m)$.
Prime inverse	$a^{p-2} \bmod p$, only for prime p and $a \not\equiv 0 \pmod{p}$.
Modular division	$a \cdot b^{-1} \bmod m$, only if b is invertible. Do not replace it with ordinary integer division.
Linear congruence	For $ax \equiv b \pmod{m}$, let $g = \gcd(a, m)$. No solution if $g \nmid b$. Otherwise there are g residues modulo m , spaced by m/g .
Combinations	$\binom{n}{r} \equiv F_n I_r I_{n-r} \pmod{p}$, with factorials F and inverse factorials I , provided $0 \leq r \leq n \leq N < p$. Reduce after each multiplication.
Backward table	$I_N = F_N^{p-2} \bmod p$; $I_{i-1} = i I_i \bmod p$. All entries depend on the same prime.

A choice guide

Use a GCD or LCM when the relation is exact divisibility. Use modular addition and multiplication when only a remainder matters. Use binary power when repeated multiplication is too long. Before dividing, ask whether the denominator is invertible under this exact modulus.

For many binomial queries under one prime and a bounded $N < p$, build tables once. For a different modulus, those tables are not reusable. For arbitrary composite moduli, a Pascal recurrence may be appropriate at small bounds; the factorial-inverse shortcut is not universally valid.

For all solutions of one linear congruence, compute a base residue and a spacing instead of pretending that every equation has one inverse-based answer. Include output size when a task asks to print many residues.

Time and memory at a glance

Let V bound exact numeric inputs, e be an exponent, L a decimal-string length, N a table limit, q a query count, and g the number of printed congruence solutions. These are fixed-width operation counts.

Method	Time	Extra space
Iterative Euclid	$O(1 + \log V)$	$O(1)$.
Recursive Euclid / extended Euclid	$O(1 + \log V)$	$O(1 + \log V)$ call stack.
GCD fold over n inputs	$O(n(1 + \log V))$ simple bound	$O(1)$ if streamed.
Iterative modular power	$O(1 + \log(e + 1))$	$O(1)$.
Recursive modular power	$O(1 + \log(e + 1))$	$O(1 + \log(e + 1))$.
One general inverse	$O(1 + \log m)$	Logarithmic recursive stack.
Factorial-table queries	$O(N + q + \log p)$ total	$O(N)$ tables.
Decimal remainder / decimal exponent	$O(L)$	$O(1)$ state, plus the stored string.
List all linear-congruence solutions	$O(1 + \log V + g)$	Logarithmic stack; output streamed.
GCD after each possible deletion	$O(n(1 + \log V))$	$O(n)$ summaries.
Count divisible-sum subarrays	$O(n + m)$	$O(m)$ residue frequencies.
Wide modular product	$O(1 + \log m)$	$O(1)$.
Paired power and geometric sum	$O(1 + \log(n + 1))$	Logarithmic stack.

The bounds for GCD calls describe the Euclidean algorithm analysed here. The standard-library contract specifies the result and its preconditions, rather than requiring this exact implementation. Counting fixed-width operations is not the same as claiming constant cost for multiplication of arbitrary-length integers.

Glossary

Common divisor

An integer that divides each of the given integers exactly.

GCD

Greatest common divisor, with nonnegative output and deliberate zero conventions in code.

Coprime

Two integers whose GCD is 1. A whole-list GCD of 1 does not imply pairwise coprimality.

LCM

Least positive common multiple for positive inputs; the library convention gives 0 if either input is 0.

Congruence

The relation $a \equiv b \pmod{m}$, meaning m divides $a - b$.

Canonical residue

The unique representative in $0, 1, \dots, m - 1$ for positive m .

Invariant

A statement preserved as an algorithm changes its state, used to connect intermediate values to the final answer.

Binary exponentiation

Computing a power by repeated squaring and the binary digits of an exponent; also called binary power.

Bezout coefficients

Integers x, y that witness $ax + by = \gcd(a, b)$.

Modular inverse

A residue x satisfying $ax \equiv 1 \pmod{m}$. It is not an ordinary reciprocal stored with integer division.

Unit

Another name for an invertible residue under a modulus.

Linear congruence

A modular equation such as $ax \equiv b \pmod{m}$; it may have zero, one, or several residues as solutions.

Inverse factorial

The modular inverse of a factorial under conditions that guarantee it exists.

Totient

$\varphi(n)$, the number of integers in 1 through n coprime to n ; used in the optional Euler extension.

Readiness checklist

You are ready to move on when you can explain these statements without reading a template.

1. Euclid preserves exactly the common divisors, reaches a zero remainder, and returns the other value.
2. An LCM formula needs both a zero rule and a product bound; dividing first is not a universal overflow guarantee.
3. C++ may give a negative remainder for a negative input. Normalization restores the canonical representative.
4. Multiplication is reduced only after its intermediate product has already been made safe.
5. Binary power preserves a stated invariant and handles exponent zero and modulus 1 deliberately.
6. A modular inverse exists precisely under a coprimality condition; a nonzero value under a composite modulus can still fail.
7. Extended Euclid returns a verifiable exact identity. Fermat's inverse shortcut requires a prime modulus.
8. A linear congruence with a non-invertible coefficient is not necessarily impossible; the GCD determines compatibility and the number of solutions.
9. Factorial and inverse-factorial tables share one prime and must remain below it in this implementation.
10. Prefix residues can count divisible subarrays, but prefix GCDs cannot be subtracted to recover an arbitrary interval.

A small test set you can reuse

Test the smallest allowed input, an all-zero case where permitted, a negative value, a prime and a composite modulus, an absent inverse, exponent zero, and an input near each numeric bound. Compare against a direct small-input solution, and check returned witnesses by substitution.

Practice planning and revision notes

Use the space below to plan the sheet with your mentors. Dates and links are deliberately left blank rather than inventing a timetable.

Sheet link

Start date and time

End date and time

Contest / review
time

A suggested study order

First solve the exact GCD and LCM tasks, then the residue and power tasks, then inverse and equation tasks. Return to table-based combinations and prefix applications after the basic helpers feel familiar. Leave the optional problems for a second pass.

After each attempt, record the smallest input that exposed a mistake. Write the missing condition next to your solution: positive modulus, prime modulus, coprimality, a representable product, or a valid table index. Re-solving that small case from the invariant is more useful than copying a corrected line.

My explanation of the most important condition

One mistake to test for next time

6.39 Technical notes, verification, and references

These notes document the domain of the code and the checks used for this edition. They are a reference for a second reading, not extra syntax to memorize before attempting the core practice.

Language version and arithmetic model

All 44 complete Session 6 programs use standard C++23 headers. No GNU policy-based data structures, arbitrary-precision package, or extended integer type is required. Standard-language references below use the fixed N4950 C++23 working draft rather than a moving post-C++23 draft.

The tested environment has a 32-bit `int` and a 64-bit `long long`. C++ specifies minimum integer widths rather than requiring exactly those widths. The printed programs use explicit bounds compatible with this environment. Signed overflow is not a modular-arithmetic strategy; it is undefined behavior. Unsigned arithmetic wraps modulo a power of two, not an arbitrary input modulus.^[6-4]

The core modular bound is $1 \leq m \leq 1000000007$. Two normalized residues have product at most 10000000120000000036 , below the signed 64-bit maximum. Intermediate products are reduced before another factor is multiplied. The optional wide-product program uses a different contract and avoids unsafe sums and products explicitly.

Signs, zeros, and helper contracts

Exact GCD programs use nonnegative inputs unless a signed-input task states its bound. The fraction example permits small signed values; it does not apply `abs` to the most-negative representable integer. Standard `gcd` requires the argument magnitudes to be representable in its common result type. Standard `lcm` additionally requires the result to be representable.^[6-2]

Power examples use nonnegative exponents. Exponent zero denotes the empty product and produces $1 \bmod m$, including the inputs $a = 0$ or $m = 1$. Inverse examples use $m \geq 2$ and require or test coprimality. A supplied prime modulus is an input promise, not a fact inferred from its appearance or size.

Extended-Euclid examples operate on bounded nonnegative inputs; general inverse helpers normalize a signed base before recursion. The displayed coefficient updates and final witness checks fit under the stated bounds. These are not unrestricted full-range signed-64-bit number-theory APIs.

Scope and interpretation

The book teaches contest algorithms with valid input, not a general input-validation library. GCD/LCM require exact values, so reducing those values modulo an unrelated number changes the problem. The optional CRT construction handles two coprime moduli only. The factorial-inverse construction does not cover $n \geq p$ or arbitrary composite moduli.

What was checked for this edition

There are **24 complete examples and 20 complete practice solutions**. All 44 compiled with GCC 14.2.0 in C++23 mode and passed **17,551 automated input/output checks**. Checks include every printed program sample, exhaustive or generated small cases, and boundary cases within the stated contracts.

Independent expected results use Python's exact integer arithmetic, GCD, LCM, powers, and binomial coefficients. Small inverse and congruence cases are checked by enumeration; divisible-subarray and deletion tasks are checked by direct methods. Large geometric sums are compared with an independently structured matrix calculation. A witness-producing extended-GCD program is checked by its defining equation and GCD, rather than by requiring one coefficient pair.

Tests cover zero values, all-zero exponent strings, leading zeros, modulus 1 where allowed, negative bases, missing inverses, composite moduli, several solutions to an equation, table endpoints, long strings, wide products, and large exact subarray counts. A separate script passed **29 exact assertions** covering all 12 reasoning drills and additional printed numerical claims.

The checking build uses:

```
-std=c++23 -pedantic-errors -Wall -Wextra  
-Wconversion -Wshadow -Werror -O1 -g  
-D_GLIBCXX_ASSERTIONS -fsanitize=undefined  
-fno-sanitize-recover=all
```

These diagnostic and sanitizer options support the verification process; they do not prove correctness on unexecuted paths. The mathematical explanations and input contracts remain necessary.^{[6-5][6-6]}

Earlier sessions and reproducibility

The six earlier topic suites were run again: Session 1 has two suites because it contains Two Pointers and Binary Search. Those **264 earlier programs** remain byte-for-byte unchanged. Together with the new 44 programs, the complete six-session collection contains **308 complete programs** and passed **95,674 input/output checks**. Additional drill assertions are reported separately, not added to that I/O count.

Run `tests/verify_number_theory_2.py` and `tests/verify_number_theory_2_drills.py` from the project root to reproduce the new checks. The README lists every earlier runner. Reports record compiler flags, counts, and source hashes. Build `session_06.tex` for this handout or `book.tex` for the complete book; both include this same chapter and the same C++ files.

Primary references and further reading

The reference markers identify supporting language rules and established mathematical results. The instructional explanations, tasks, traces, and programs in this chapter are written for this training; the linked sources are not prerequisite reading.

[6-1] Antti Laaksonen: Competitive Programmer's Handbook, number theory

<https://cses.fi/book/book.pdf>

Chapter 21 provides a compact reference for GCD, modular powers, inverses, equations, and congruence systems. Read it after the step-by-step explanations here, not instead of them.

[6-2] C++23 N4950: greatest common divisor and least common multiple

<https://timsong-cpp.github.io/cppwp/n4950/numeric.ops>

See the `numeric.ops.gcd` and `numeric.ops.lcm` sections for return conventions, common types, and representability preconditions.

[6-3] C++23 N4950: multiplication, division, and remainder

<https://timsong-cpp.github.io/cppwp/n4950/expr.mul>

[6-4] C++23 N4950: fundamental integer types and arithmetic

<https://timsong-cpp.github.io/cppwp/n4950/basic.fundamental>

[6-5] GCC: instrumentation and undefined-behavior checking

<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

[6-6] GCC: language-standard selection

<https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html>

[6-7] MIT Mathematics for Computer Science: GCD and extended Euclid

https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-spring-2015/mit6_042js15_session12.pdf

Chapter 8.1 to 8.5, especially Section 8.2, develops common divisors and linear-combination witnesses. The text calls extended Euclid the Pulverizer.

[6-8] MIT Mathematics for Computer Science: congruences and inverses

https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-spring-2015/mit6_042js15_session13.pdf

Chapter 8.6 to 8.9 explains modular operations, inverse existence, and cancellation conditions.

[6-9] MIT Mathematics for Computer Science: Euler's theorem and Fermat's theorem

https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-spring-2015/mit6_042js15_session14.pdf

Chapter 8.10 connects coprime residues, totients, powers, and inverse powers.

How to use a reference responsibly

Look up the exact condition you need. A theorem stated for a prime modulus does not automatically apply to a composite one. A library function with a representability precondition is not a safe substitute for a bound check. A formula that holds over exact integers may need a different implementation after taking remainders.

Earlier chapters retain their own topic-specific verification records. The counts above describe this complete release and include the rerun earlier suites. Updated front matter supplies the final six-session organization without rewriting the historical claims in each chapter.

6.40 Completing Level 1: build an argument before a template

The six sessions now form one study sequence. Session 1 exploits order and monotonicity. Session 2 explores and restores choices. Session 3 represents small sets and binary state. Session 4 counts outcomes without enumerating them. Sessions 5 and 6 use integer structure and congruences to replace repeated work with justified arithmetic.

A repeatable problem-solving process

1. **Define the required object.** Is the answer a position, a set, a count, a minimum, a divisor, or a residue? Distinguish an exact answer from an answer modulo a number.
2. **Write a small correct solution.** Enumerate possibilities or simulate a tiny input. This becomes a source of counterexamples and an independent checker.
3. **Identify the reusable structure.** Sorted order, a monotone condition, independent choices, a compact state, a prime factorization, or an invariant may remove repeated work.
4. **Prove the shortcut.** Explain why a discarded candidate is unnecessary, a branch partition is complete, a count is unique, or a transformation preserves the intended relation.
5. **Protect the implementation.** State bounds, index conventions, sign rules, empty cases, and arithmetic preconditions before choosing types and helpers.
6. **Test and revisit.** Compare against the small method, examine the simplest failing input, and rewrite the final solution later without looking at the template.

What progress looks like

You do not need to recall every optional formula immediately. You should be able to recognize an idea, find its conditions, reconstruct its invariant, and explain why your implementation meets them. That is a stronger foundation than memorizing a long collection of code snippets.

Keep practising problems that combine two sessions. A number-theory formula may need binary search over a bound. A counting argument may rely on a frequency map. A brute-force bitmask program may be the best checker for a faster mathematical solution. The tools are most useful when you can justify how they fit together.