Module 6 of 12

Number theory: sieve, modular arithmetic, nCr

The Sieve of Eratosthenes with prefix counts, exponentiation by squaring, inverses by Fermat's little theorem, nCr mod 1e9+7 from factorial tables, and __int128 for moduli near 10^18.

Contest number theory is a small toolkit used everywhere: answers "modulo $10^9 + 7$", primes up to 10^7, binomial coefficients for counting problems, powers with exponents in the billions. This module teaches the four tools that cover most of it and the overflow rules that keep them correct.

The lesson animates the **Sieve of Eratosthenes** crossing out multiples from p^2, and **modular exponentiation** squaring its way through the bits of the exponent; then it turns division into multiplication by an inverse (Fermat's little theorem, since $10^9 + 7$ is prime) and builds factorial and inverse-factorial tables for O(1) nCr. The exercises answer 3 000 prime-count range queries with a sieve and a prefix count, and 3 000 nCr queries with n up to 10^6. The timed checkpoint raises numbers up to 10^{18} to powers up to 10^{18} modulo values up to 10^{18} — where every product needs `__int128`.

CONTENTS

1	The sieve, modular arithmetic and nCr mod p	15 min
✓	Checkpoint — Number theory	45 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 15 MIN

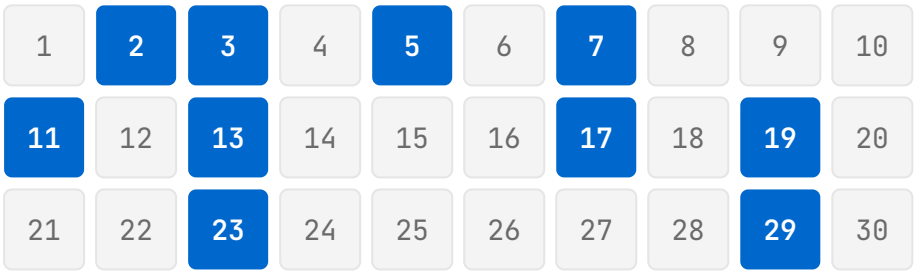
The sieve, modular arithmetic and nCr mod p

Three tools cover most of the number theory a contest asks for: a **sieve** for every prime at once, **modular exponentiation** for huge powers, and **factorial tables** for binomial coefficients under a prime modulus.

The sieve

Testing each number by trial division costs O(n√n). The Sieve of Eratosthenes marks every composite up to n in O(n log log n): each prime crosses out its multiples, starting at its square.

The Sieve of Eratosthenes on 1..30



primes ≤ 30: 10

1. Write 1 to 30. 1 is not prime; every other number starts unmarked. The first unmarked number is the next prime.
2. 2 is unmarked, so it is prime. Cross out its multiples from 4 = 2×2: 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30. Smaller multiples of 2 have a smaller prime factor and are gone already.
3. 3 is unmarked, so it is prime. Cross out its multiples from 9 = 3×3: 9, 15, 21, 27 (4 already crossed by a smaller prime). Smaller multiples of 3 have a smaller prime factor and are gone already.
4. 5 is unmarked, so it is prime. Cross out its multiples from 25 = 5×5: 25 (1 already crossed by a smaller prime). Smaller multiples of 5 have a smaller prime factor and are gone already.
5. The next prime, 7, has $7^2 = 49 > 30$: every composite up to 30 has a prime factor ≤ √30 and is crossed. The 10 numbers left are the primes. In all, O(n log log n).

```
vector<bool> composite(N + 1);
composite[0] = composite[1] = true;
for (ll p = 2; p * p ⩽ N; ++p)
    if (!composite[p])
        for (ll m = p * p; m ⩽ N; m += p) composite[m] = true;
```

- Range questions — "how many primes in [l, r]" — become a **prefix count** over the sieve: O(1) each.
- Keep the smallest prime factor instead of a bool and you can factor any number up to N in O(log N).

Powers by squaring

`a^e mod m` with `e` up to 10^{18} needs about 60 steps, not 10^{18}.

Modular exponentiation by squaring

e (binary)	bit	result	base
start	–	1	3
1101	1	1·3 → 3	3^2 mod 1000 = 9
110	0	3	9^2 mod 1000 = 81
11	1	3·81 → 243	81^2 mod 1000 = 561
1	1	243·561 → 323	561^2 mod 1000 = 721

1. 3^13 mod 1000 without 13 multiplications. 13 in binary is 1101, so 3^13 = 3^1 · 3^4 · 3^8. Start with result 1 and base 3.
2. Lowest bit of 1101 is 1: multiply the result by the base, 1 · 3 mod 1000 = 3. Then square the base (9) and shift the exponent right.
3. Lowest bit of 110 is 0: the result stays 3. Square the base to 81 and shift the exponent right.
4. Lowest bit of 11 is 1: multiply the result by the base, 3 · 81 mod 1000 = 243. Then square the base (561) and shift the exponent right.
5. Lowest bit of 1 is 1: multiply the result by the base, 243 · 561 mod 1000 = 323. Then square the base (721) and shift the exponent right.
6. The exponent is 0: 3^13 mod 1000 = 323, in 4 squarings, one per bit — O(log e). Reduce after every product, so each one is below 1000^2 and never overflows.

```
ll power(ll b, ll e, ll m) {
    ll r = 1 % m;
    b %= m;
    for (; e > 0; e >>= 1, b = b * b % m)
        if (e & 1) r = r * b % m;
    return r;
}
```

- **Reduce after every product.** Below `MOD = 1e9+7` two values multiply to under 2^{60}, safe in `long long`.
- **A modulus near 10^{18}?** Multiply in `__int128`: `(__int128)a * b % m`.
- `1 % m`, not `1`: when `m = 1` the answer is 0.

Division mod a prime, and nCr

You cannot divide under a modulus, but you can multiply by an **inverse**. For a prime `p`, Fermat gives `a^-1 = a^(p-2) mod p`.

```
fact[0] = 1;
for (int i = 1; i ⩽ N; ++i) fact[i] = fact[i - 1] * i % MOD;
inv[N] = power(fact[N], MOD - 2, MOD);
for (int i = N; i > 0; --i) inv[i - 1] = inv[i] * i % MOD;
// C(n, r) = fact[n] * inv[r] % MOD * inv[n - r] % MOD, and 0 when r > n
```

One modpow and two O(N) loops, then every query is O(1).

The traps

- **Three factors, one reduction.** `a * b * c % MOD` overflows even when each is reduced; reduce after every product.
- **Negative results.** `(a - b) % MOD` can be negative in C++; write `(a - b + MOD) % MOD`.
- **A composite modulus.** Fermat's inverse needs a prime. Modulo a composite `m`, `b` has an inverse only when `gcd(b, m) = 1`.
- **Tables too small.** Size them by the largest `n` any query can ask, not the first one you read.

GAME · MATCH THE PAIRS

The tool for the job

Match each task to the technique that solves it in time.

Choose a term on the left, then its partner on the right.

A Count primes in a million ranges up to 10^6	· multiply by b^(MOD-2)
B 2^(10^18) mod 1e9+7	· factorial and inverse factorial tables

C a / b mod 1e9+7

D C(n, r) mod 1e9+7 for 10^5 queries

E a * b mod m with m near 10^18

· exponentiation by squaring

· sieve plus a prefix count

· multiply in __int128

Check

More practice

- Count Primes — the sieve itself.
- Closest Prime Numbers in Range — sieve, then scan.
- Count Good Numbers — modpow with a huge exponent.
- Prime Arrangements — a sieve and factorials mod p.
- Count Anagrams — multinomials with inverse factorials.

EXERCISE 1

Primes in ranges

Answer `q` questions: how many primes lie in the range `[l, r]`, both ends included?

Input: `q` (`1 ≤ q ≤ 10^5`), then `q` lines `l r` (`1 ≤ l ≤ r ≤ 10^6`). **Output:** one line per question, the count.

```
5
1 10
10 20
2 2
1 1
14 16
```

prints

```
4
4
1
0
0
```

Trial division per question is far too slow at the limits; sieve once.

INPUT 1

```
5
1 10
10 20
2 2
1 1
14 16
```

OUTPUT 1

```
4
4
1
0
0
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1'000'000;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    vector<bool> composite(N + 1, false);
    composite[0] = composite[1] = true;
    for (ll p = 2; p * p <= N; ++p)
        if (!composite[p])
            for (ll m = p * p; m <= N; m += p) composite[m] = true;

    // cnt[i] = primes in [1, i]; a range is a difference of two prefixes.
    vector<int> cnt(N + 1, 0);
    for (int i = 1; i <= N; ++i) cnt[i] = cnt[i - 1] + (composite[i] ? 0 : 1);

    int q;
    cin >> q;
    while (q--) {
        int l, r;
        cin >> l >> r;
        cout << cnt[r] - cnt[l - 1] << '\n';
    }
    return 0;
}
```

EXERCISE 2

nCr modulo 1e9+7

Answer `q` questions: print `C(n, r)` — the number of ways to choose `r` items from `n` — modulo `1 000 000 007`. When `r > n` the answer is `0`.

Input: `q` (`1 ≤ q ≤ 10^5`), then `q` lines `n r` (`0 ≤ n, r ≤ 10^6`). **Output:** one line per question.

```
5
5 2
10 0
10 10
3 5
0 0
```

prints

```
10
1
1
0
1
```

Precompute factorials and inverse factorials up to 10^6 once; each question is then O(1).

INPUT 1

```
5
5 2
10 0
10 10
3 5
0 0
```

OUTPUT 1

```
10
1
1
0
1
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

const ll MOD = 1'000'000'007;
const int N = 1'000'000;

ll power(ll b, ll e) {
    ll r = 1;
    b %= MOD;
    for (; e > 0; e >>= 1, b = b * b % MOD)
        if (e & 1) r = r * b % MOD;
    return r;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    vector<ll> fact(N + 1), inv(N + 1);
    fact[0] = 1;
    for (int i = 1; i <= N; ++i) fact[i] = fact[i - 1] * i % MOD;
    // MOD is prime, so fact[N]^(MOD-2) is its inverse (Fermat); walk down for the r
    inv[N] = power(fact[N], MOD - 2);
    for (int i = N; i > 0; --i) inv[i - 1] = inv[i] * i % MOD;

    int q;
    cin >> q;
    while (q--) {
        int n, r;
        cin >> n >> r;
        if (r > n) {
            cout << 0 << '\n';
            continue;
        }
        cout << fact[n] * inv[r] % MOD * inv[n - r] % MOD << '\n';
    }
    return 0;
}
```

QUIZ

1. Why does the sieve's inner loop start at `p * p` rather than `2 * p`?

A. It is required for the prefix count to work

B. Every smaller multiple of p has a smaller prime factor and is already crossed out

C. Starting at 2 * p would cross out p itself

D. p * p is the only multiple that matters

2. `MOD = 1e9+7`. Which expression can overflow `long long` when `a`, `b` and `c` are already reduced mod MOD?

A. a * b % MOD * c % MOD

B. (a + b) % MOD

C. a * b * c % MOD

D. a * b % MOD

3. What is `power(5, 0, 1)` with the lesson's modpow, which starts from `r = 1 % m`?

A. 1

B. 5

C. Undefined

D. 0

4. Why may a / b mod p be computed as `a * power(b, p - 2, p) % p`?

A. Because p is prime and b is not a multiple of p, b^(p-1) ≡ 1, so b^(p-2) is b's inverse

B. Because p - 2 is always even

C. It works for any modulus

D. Because power rounds the division

ANSWER KEY · THE SIEVE, MODULAR ARITHMETIC AND NCR MOD P

1. B k·p with k < p is divisible by some prime ≤ k < p, which crossed it out earlier. Starting at p·p skips repeated work — and lets the outer loop stop when p·p > N.

2. C Two reduced values multiply to under 1.0e18, which fits; multiplying that by a third before reducing can reach 10^27. Reduce after every product.

3. D Everything is 0 modulo 1. Starting from `1 % m` gives 0 when m = 1; starting from a bare `1` would wrongly print 1.

4. A Fermat's little theorem: for prime p and b not divisible by p, b^(p−1) ≡ 1 (mod p), so b · b^(p−2) ≡ 1. For a composite modulus you need the extended Euclidean algorithm, and an inverse exists only when gcd(b, m) = 1.

CHECKPOINT · 45 MIN

Checkpoint — Number theory

A timed checkpoint on contest number theory: the sieve and prefix counts, exponentiation by squaring, inverses under a prime modulus, factorial tables for nCr, and keeping every product inside its type.

How it works. Ten questions and one program, in **45 minutes** from when you start the timer. Clear 70% of the questions and get the program accepted on every hidden case — moduli up to 10^{18} included — to complete the module.

Before you start, make sure you can answer these from memory:

- Why the sieve starts at p^2, and its complexity.
- The modpow loop, and why the result starts at `1 % m`.
- Why reduced residues below 1e9+7 multiply safely in `long long`, and what changes near 10^{18}.
- How one modpow and two loops give every inverse factorial up to N.
- When an inverse exists modulo a composite number.

EXERCISE

Huge powers, huge modulus

Answer `q` questions: print `a^b mod m`. Take `0^0 = 1`, so `a^0 mod m` is `1 mod m`.

Input: `q` (`1 ≤ q ≤ 10^4`), then `q` lines `a b m` (`0 ≤ a, b ≤ 10^18`, `1 ≤ m ≤ 10^18`). **Output:** one line per question.

```
5
2 10 1000
3 13 1000
5 0 7
7 3 1
0 0 10
```

prints

```
24
323
1
0
1
```

With `m` near 10^{18}, the product of two residues reaches 10^{36} — no 64-bit type holds it.

INPUT 1

```
5
2 10 1000
3 13 1000
5 0 7
7 3 1
0 0 10
```

OUTPUT 1

```
24
323
1
0
1
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

// Residues below 10^18 multiply to below 10^36: only a 128-bit product holds it.
ll mulmod(ll a, ll b, ll m) {
    return static_cast<ll>(static_cast<__int128>(a) * b % m);
}

ll power(ll b, ll e, ll m) {
    ll r = 1 % m;   // m = 1 makes every answer 0
    b %= m;
    while (e > 0) {
        if (e & 1) r = mulmod(r, b, m);
        b = mulmod(b, b, m);
        e >>= 1;
    }
    return r;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int q;
    cin >> q;
    while (q--) {
        ll a, b, m;
        cin >> a >> b >> m;
        cout << power(a, b, m) << '\n';
    }
    return 0;
}
```

CHECKPOINT QUESTIONS

1. What is the time complexity of the Sieve of Eratosthenes up to n?

A. O(√n)

B. O(n log log n)

C. O(n √n)

D. O(n log n) per prime

2. How many multiplications does exponentiation by squaring use for an exponent near 10^18?

A. About 10^18

B. About 18

C. About 120 (two per bit, 60 bits)

D. About 10^9

3. Why is `(__int128)a * b % m` needed when m is near 10^18, but not when m = 1e9+7?

A. __int128 is faster

B. long long cannot hold 10^18

C. Because % is undefined on long long

D. Residues below 1e9+7 multiply to under 2^60, which long long holds; residues below 10^18 multiply to up to 10^36

4. With factorial tables up to N, `inv[N] = power(fact[N], MOD - 2)`. How is `inv[N - 1]` obtained?

A. inv[N] * N % MOD

B. inv[N] / N

C. power(fact[N - 1], MOD - 2) is the only way

D. inv[N] * (N - 1) % MOD

5. What is C(10, 3) mod 1e9+7?

A. 1000

B. 120

C. 720

D. 30

6. How do you count the primes in many ranges [l, r] with r ≤ 10^6?

A. Sieve each range separately

B. Binary search for l and r in the sieve array

C. Sieve once, build a prefix count, answer cnt[r] − cnt[l − 1]

D. Trial-divide every number in each range

7. Why can't you compute C(n, r) mod p as `fact[n] / (fact[r] * fact[n - r]) % p` with reduced factorials?

A. Division is slower than multiplication

B. fact[n] overflows before the division

C. It works, but only for even n

D. Reduced factorials are no longer divisible in the integers; division must be a multiplication by the modular inverse

8. Fermat's little theorem gives inverses modulo a prime p. When does an inverse of b modulo a composite m exist?

A. Exactly when gcd(b, m) = 1

B. Always

C. Never

D. Only when m is even

9. In `power`, what goes wrong if the base is not reduced (`b %= m`) before the loop, with b up to 10^18 and m = 1e9+7?

A. The result is always 1

B. The first squaring b * b overflows long long

C. Nothing; the answer is the same

D. The loop runs forever

10. A smallest-prime-factor sieve stores `spf[x]` for every x ≤ N. How long does factoring one x ≤ N take with it?

A. O(N)

B. O(1)

C. O(log x)

D. O(√x)

ANSWER KEY · CHECKPOINT — NUMBER THEORY

1. B Each prime p crosses about n/p numbers, and the sum of 1/p over primes up to n grows like log log n. Trial division of every number would be O(n√n).

2. C 10^18 < 2^60, so there are at most 60 bits: one squaring per bit and at most one multiply into the result.

3. D long long holds up to about 9.2 × 10^18. (10^9)² = 10^18 fits; (10^18)² does not. __int128 holds about 1.7 × 10^38.

4. A 1/(N−1)! = N / N! = N · inv[N]. Walking down this way computes every inverse factorial with one modpow and N multiplications.

5. B 10·9·8 / 3! = 720 / 6 = 120, already below the modulus.

6. C The sieve is O(N log log N) once; the prefix count makes each range O(1). Trial division per range is O(range · √r) every time.

7. D After reducing mod p, fact[n] may be smaller than the divisor or not a multiple of it, so integer division gives nonsense. Multiply by the inverse instead.

8. A b has an inverse mod m iff b and m are coprime; the extended Euclidean algorithm finds it. Fermat's shortcut b^(m−2) only works for prime m.

9. B b · b with b near 10^18 is 10^36: undefined behaviour in long long. Reducing first keeps every factor below m, so with m = 1e9+7 each product stays under 2^60.

10. C Repeatedly divide x by spf[x]. Each division at least halves x, so there are at most log2(x) steps.