Competitive Programming

Science Honors Program (SHP)

Session 7

Guest: Rain Yang (Christian Yongwhan Lim)

Saturday, November 14, 2026

Overview

- Modular Arithmetic
  - Extended Euclidean Algorithm
  - Modular Inverse
  - Linear Congruence Equation
  - Chinese Remainder Theorem

Extended Euclidean Algorithm

- The extended Euclidean algorithm finds a way to represent GCD in terms of a and b. So, it finds coefficients x and y for which:

$$a \cdot x + b \cdot y = \gcd(a, b)$$

Extended Euclidean Algorithm (con't)

- The changes to the Euclidean algorithm are very simple:
  - We can see that the algorithm ends with b=0 and a=g.
  - For these parameters we can easily find coefficients, namely $g \cdot 1 + 0 \cdot 0 = g$.
  - Starting from these coefficients (x,y)=(1,0), we can go backwards up the recursive calls.
  - All we need to do is to figure out how the coefficients x and y change during the transition from (a,b) to (b,a%b).

Extended Euclidean Algorithm (con't)

- Suppose, we found (x_1,y_1) for (b,a%b):

$$b \cdot x_1 + (a \bmod b) \cdot y_1 = g$$

- And want to find (x,y) such that:

$$a \cdot x + b \cdot y = g$$

Extended Euclidean Algorithm (con't)

- Since:

$$a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b$$

- We get:

$$g = b \cdot x_1 + (a \bmod b) \cdot y_1 = b \cdot x_1 + \left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right) \cdot y_1$$

Extended Euclidean Algorithm: Implementation #1

```
int gcd(int a, int b, int& x, int& y) {
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }
  int x1, y1;
  int d = gcd(b, a % b, x1, y1);
  x = y1;
  y = x1 - y1 * (a / b);
  return d;
}
```

Extended Euclidean Algorithm: Implementation #2

```
int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0;
  int x1 = 0, y1 = 1, a1 = a, b1 = b;
  while (b1) {
    int q = a1 / b1;
    tie(x, x1) = make_tuple(x1, x - q * x1);
    tie(y, y1) = make_tuple(y1, y - q * y1);
    tie(a1, b1) = make_tuple(b1, a1 - q * b1);
  }
  return a1;
}
```

Modular Inverse

- A modular multiplicative inverse of an integer a is an integer x such that ax is congruent to 1 modular some modulus m.

- We want to find an integer x so that:

$$a \cdot x \equiv 1 \mod m$$

- We will also denote x simply with a^{-1}.

Modular Inverse: Extended Euclidean algorithm

- Consider the following equation with unknown x and y:

$$a \cdot x + m \cdot y = 1$$

- This is a Linear Diophantine equation in two variables. When gcd(a, m)=1, the equation has a solution which can be found using the extended Euclidean algorithm. Note that gcd(a, m)=1 is also the condition for the modular inverse to exist.

Modular Inverse: Extended Euclidean algorithm (con't)

- Now, if we take modulo m of both sides, we can get rid of m·y and the equation becomes:

$$a \cdot x \equiv 1 \mod m$$

- Thus, the modular inverse of a is x.

Modular Inverse: Extended Euclidean algorithm (con't)

```
int x, y;
int g = extended_euclidean(a, m, x, y);
if (g != 1) {
  cout << "No solution!";
}
else {
  x = (x % m + m) % m;
  cout << x << endl;
}
```

Modular Inverse: Binary Exponentiation

- Another method for finding modular inverse is to use **Euler's theorem**, which states that the following congruence is true if a and m are relatively prime:

$$a^{\phi(m)} \equiv 1 \mod m$$

- where ϕ is totient function.

Modular Inverse: Binary Exponentiation (con't)

- Note that a and m being relative prime is the condition for the modular inverse to exist.

- If m is a prime number, this simplifies to Fermat's little theorem:

$$a^{m-1} \equiv 1 \mod m$$

Modular Inverse: Binary Exponentiation (con't)

- For an arbitrary (but coprime) modulus m:

$$a^{\phi(m)-1} \equiv a^{-1} \mod m$$

- For a prime modulus m:

$$a^{m-2} \equiv a^{-1} \mod m$$

- From here, we can use the binary exponentiation.

Modular Inverse: Binary Exponentiation (con't)

```
ll inv(ll a, ll m) {
    return exp(a, phi(m)-1, m);
}

ll invp(ll a, ll m) {
    return exp(a, m-2, m);
}
```

Linear Congruence Equation

- We just need to solve:

$$a \cdot x \equiv b \pmod{n}$$

- When gcd(a,n) = 1 (coprime), we just need to find the multiplicative inverse:

$$x \equiv b \cdot a^{-1} \pmod{n}$$

Linear Congruence Equation (con't)

- When gcd(a,n) = 1 (coprime), we just need to find the multiplicative inverse:

$$x \equiv b \cdot a^{-1} \pmod{n}$$

- If not, let g := gcd(a,n). If b is not divisible by g, then there is **no solution**! If g divides b, then by dividing both sides of the equation by g (i.e. dividing a, b and n by g), we get:

$$a' \cdot x \equiv b' \pmod{n'}$$

Linear Congruence Equation (con't)

- We get x' as solution for x.
- It is clear that x' will also be a solution of the original equation. However, it will not be the only solution. It can be shown that the original equation has exactly g solutions, given by:

$$x_i \equiv (x' + i \cdot n') \pmod{n} \quad \text{for } i = 0 \ldots g - 1$$

- Summarizing, we can say that the number of solutions of the linear congruence equation is equal to either gcd(a,n) or 0.

Chinese Remainder Theorem

- Where m_i are pairwise coprime, let:

$$m = m_1 \cdot m_2 \cdots m_k$$

- Where a_i are some given constants, suppose we have:

$$\begin{cases} a \equiv a_1 \pmod{m_1} \\ a \equiv a_2 \pmod{m_2} \\ \vdots \\ a \equiv a_k \pmod{m_k} \end{cases}$$

Chinese Remainder Theorem (con't)

- The original form of CRT then states that the given system of congruences always has ***one and exactly one solution*** modulo m.

- For example,

$$\begin{cases} a & \equiv & 2 \pmod{3} \\ a & \equiv & 3 \pmod{5} \\ a & \equiv & 2 \pmod{7} \end{cases}$$

- has the solution 23 modulo 105.

Chinese Remainder Theorem (con't)

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



$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ \vdots \\ x \equiv a_k \pmod{m_k} \end{cases}$$

Chinese Remainder Theorem: Solution for Two Moduli

$$\begin{cases} a \equiv a_1 \pmod{m_1} \\ a \equiv a_2 \pmod{m_2} \end{cases}$$

- We want:

$$a \pmod{m_1 m_2}$$

- Using the **Extended Euclidean Algorithm** we can find **Bézout** coefficients n_1, n_2 such that

$$n_1 m_1 + n_2 m_2 = 1$$

Chinese Remainder Theorem: Solution for Two Moduli

$$n_1 \equiv m_1^{-1} \pmod{m_2}$$
$$n_2 \equiv m_2^{-1} \pmod{m_1}$$

Chinese Remainder Theorem: Solution for Two Moduli

$$n_1 \equiv m_1^{-1} \pmod{m_2}$$
$$n_2 \equiv m_2^{-1} \pmod{m_1}$$

$$a = a_1 n_2 m_2 + a_2 n_1 m_1 \bmod m_1 m_2$$

Chinese Remainder Theorem: Solution for Two Moduli

- We can check:

$$\begin{aligned} a &\equiv a_1 n_2 m_2 + a_2 n_1 m_1 && \pmod{m_1} \\ &\equiv a_1(1 - n_1 m_1) + a_2 n_1 m_1 && \pmod{m_1} \\ &\equiv a_1 - a_1 n_1 m_1 + a_2 n_1 m_1 && \pmod{m_1} \\ &\equiv a_1 && \pmod{m_1} \end{aligned}$$

- The same holds for m_2 by symmetry!

Chinese Remainder Theorem: Solution for Two Moduli

- **Uniqueness**

$$x \equiv a_i \pmod{m_i} \qquad y \equiv a_i \pmod{m_i}$$

Chinese Remainder Theorem: Solution for Two Moduli

- **Uniqueness**

$$x \equiv a_i \pmod{m_i} \quad y \equiv a_i \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_i}$$

Chinese Remainder Theorem: Solution for Two Moduli

- **Uniqueness**

$$x \equiv a_i \pmod{m_i} \quad y \equiv a_i \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_1 m_2}$$

Chinese Remainder Theorem: Solution for Two Moduli

- **Uniqueness**

$$x \equiv a_i \pmod{m_i} \quad y \equiv a_i \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_1 m_2}$$

$$x \equiv y \pmod{m_1 m_2}$$

Chinese Remainder Theorem: Solution for General Case

- **Inductive Solution**
  - As m_1m_2 is coprime to m_3, we can inductively repeatedly apply the solution for two moduli for any number of moduli.

Chinese Remainder Theorem: Solution for General Case

- **Inductive Solution**
  - As m_1m_2 is coprime to m_3, we can inductively repeatedly apply the solution for two moduli for any number of moduli.
- **Direct Construction**

$$M_i := \prod_{i \neq j} m_j \qquad N_i := M_i^{-1} \bmod m_i$$

Chinese Remainder Theorem: Solution for General Case

- **Inductive Solution**
  - As m_1m_2 is coprime to m_3, we can inductively repeatedly apply the solution for two moduli for any number of moduli.
- **Direct Construction**

$$M_i := \prod_{i \neq j} m_j \qquad N_i := M_i^{-1} \bmod m_i$$

$$a \equiv \sum_{i=1}^{k} a_i M_i N_i \pmod{m_1 m_2 \cdots m_k}$$

Chinese Remainder Theorem: Solution for General Case

- We can check this is indeed a solution:

$$\begin{aligned} a &\equiv \sum_{j=1}^{k} a_j M_j N_j \pmod{m_i} \\ &\equiv a_i M_i N_i \pmod{m_i} \\ &\equiv a_i M_i M_i^{-1} \pmod{m_i} \\ &\equiv a_i \pmod{m_i} \end{aligned}$$

Chinese Remainder Theorem: Implementation

```
struct Congruence { ll a, m; };
ll crt(vector<Congruence> const& cs) {
  ll M = 1, ret = 0;
  for (auto const& c : cs)
    M *= c.m;
  for (auto const& c : cs) {
    ll a_i=c.a, M_i=M/c.m, N_i = mod_inv(M_i,c.m);
    ret=(ret+a_i*M_i%M*N_i) % M;
  }
  return ret;
}
```

CRT: Solution for not coprime moduli

- In the not coprime case, a system of congruences has **exactly one** solution modulo **lcm**(m_1, m_2, ..., m_k) or has **no** solution at all.
- Where

$$p_1^{n_1} p_2^{n_2} \cdots p_k^{n_k}$$

- is a prime factorization of m_i, the following are equivalent!

$$a \equiv a_i \pmod{m_i} \qquad a \equiv a_i \pmod{p_j^{n_j}}$$

CRT: Solution for not coprime moduli (con't)

- Because originally some moduli had common factors, we will get some congruences moduli based on the same prime, however possibly with **different** prime powers.
- The congruence with the **highest** prime power modulus will be the strongest congruence of all congruences based on the same prime number.
- If there are no contradictions, then the system of equation has a solution.
    - We can ignore all congruences except the ones with the highest prime power moduli. These moduli are now coprime. So, we are back to coprime moduli!


BREAK

Group Exercise

- You may work in groups with those sitting near you.
- Each person must submit their own solutions.
- You have **75 minutes**.
- Afterward, you will present your solution!


THANK YOU