1 Mathematics

1.1 Modular Arithmetic

1.1.1 Chinese Remainder Theorem

```cpp
#include "../../headers/headers.h"

ll inline mod(ll x, ll m) { return ((x %= m) < 0) ? x + m : x; }
ll inline mul(ll x, ll y, ll m) { return (x * y) % m; }
ll inline add(ll x, ll y, ll m) { return (x + y) % m; }

// extended euclidean algorithm
// finds g, x, y such that
//    a * x + b * y = g = GCD(a,b)
ll gcdext(ll a, ll b, ll &x, ll &y)
{
    ll r2, x2, y2, r1, x1, y1, r0, x0, y0, q;
    r2 = a, x2 = 1, y2 = 0;
    r1 = b, x1 = 0, y1 = 1;
    while (r1)
    {
        q = r2 / r1;
        r0 = r2 % r1;
        x0 = x2 - q * x1;
        y0 = y2 - q * y1;
        r2 = r1, x2 = x1, y2 = y1;
        r1 = r0, x1 = x0, y1 = y0;
    }
    ll g = r2;
    x = x2, y = y2;
    if (g < 0)
        g = -g, x = -x, y = -y; // make sure g > 0
    // for debugging (in case you think you might have bugs)
    // assert (g == a * x + b * y);
    // assert (g == __gcd(abs(a),abs(b)));
    return g;
}

// ==============================================
// CRT for a system of 2 modular linear equations
// ==============================================
// We want to find X such that:
//   1) x = r1 (mod m1)
//   2) x = r2 (mod m2)
// The solution is given by:
//    sol = r1 + m1 * (r2-r1)/g * x' (mod LCM(m1,m2))
// where x' comes from
//   m1 * x' + m2 * y' = g = GCD(m1,m2)
//   where x' and y' are the values found by extended euclidean algorithm (gcdext)
// Useful references:
//   https://codeforces.com/blog/entry/61290
//   https://forthright48.com/chinese-remainder-theorem-part-1-coprime-moduli
//   https://forthright48.com/chinese-remainder-theorem-part-2-non-coprime-moduli
// ** Note: this solution works if lcm(m1,m2) fits in a long long (64 bits)
pair<ll, ll> CRT(ll r1, ll m1, ll r2, ll m2)
{
    ll g, x, y;
    g = gcdext(m1, m2, x, y);
    if ((r1 - r2) % g != 0)
        return {-1, -1}; // no solution
    ll z = m2 / g;
    ll lcm = m1 * z;
    ll sol = add(mod(r1, lcm), m1 * mul(mod(x, z), mod((r2 - r1) / g, z), z), lcm);
    // for debugging (in case you think you might have bugs)
    // assert (0 <= sol and sol < lcm);
    // assert (sol % m1 == r1 % m1);
    // assert (sol % m2 == r2 % m2);
    return {sol, lcm}; // solution + lcm(m1,m2)
}

// ==============================================
// CRT for a system of N modular linear equations
// ==============================================
//  Args:
//      r = array of remainders
//      m = array of modules
//      n = length of both arrays
//  Output:
//      a pair {X, lcm} where X is the solution of the sytemm
//          X = r[i] (mod m[i]) for i = 0 ... n-1
//      and lcm = LCM(m[0], m[1], ..., m[n-1])
//      if there is no solution, the output is {-1, -1}
// ** Note: this solution works if LCM(m[0],...,m[n-1]) fits in a long long (64 bits)
pair<ll, ll> CRT(ll *r, ll *m, int n)
{
    ll r1 = r[0], m1 = m[0];
    repx(i, 1, n)
    {
        ll r2 = r[i], m2 = m[i];
        ll g, x, y;
        g = gcdext(m1, m2, x, y);
        if ((r1 - r2) % g != 0)
            return {-1, -1}; // no solution
        ll z = m2 / g;
        ll lcm = m1 * z;
        ll sol = add(mod(r1, lcm), m1 * mul(mod(x, z), mod((r2 - r1) / g, z), z), lcm);
        r1 = sol;
        m1 = lcm;
    }
    // for debugging (in case you think you might have bugs)
    // assert (0 <= r1 and r1 < m1);
    // rep(i, n) assert (r1 % m[i] == r[i]);
```

```
    return {r1, m1};
}
```

1.2 Primality Checks

1.2.1 Miller Rabin

```
#include "../../../headers/headers.h"

ll mulmod(ull a, ull b, ull c)
{
    ull x = 0, y = a % c;
    while (b)
    {
        if (b & 1)
            x = (x + y) % c;
        y = (y << 1) % c;
        b >>= 1;
    }
    return x % c;
}

ll fastPow(ll x, ll n, ll MOD)
{
    ll ret = 1;
    while (n)
    {
        if (n & 1)
            ret = mulmod(ret, x, MOD);
        x = mulmod(x, x, MOD);
        n >>= 1;
    }
    return ret;
}

bool isPrime(ll n)
{
    vi a = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

    if (binary_search(a.begin(), a.end(), n))
        return true;

    if ((n & 1) == 0)
        return false;

    int s = 0;
    for (ll m = n - 1; !(m & 1); ++s, m >>= 1)
        ;

    int d = (n - 1) / (1 << s);

    for (int i = 0; i < 7; i++)
    {
        ll fp = fastPow(a[i], d, n);
        bool comp = (fp != 1);
        if (comp)
            for (int j = 0; j < s; j++)
            {
                if (fp == n - 1)
                {
                    comp = false;
                    break;
                }

                fp = mulmod(fp, fp, n);
            }
        if (comp)
            return false;
    }
    return true;
}
```

1.2.2 Sieve of Eratosthenes

```
#include "../../../headers/headers.h"

// O(n log log n)
vi sieve(int n)
{
    vi primes;

    vector<bool> is_prime(n + 1, true);
    int limit = (int)floor(sqrt(n));
    repx(i, 2, limit + 1) if (is_prime[i]) for (int j = i * i; j <= n; j
         += i)
        is_prime[j] = false;

    repx(i, 2, n + 1) if (is_prime[i]) primes.eb(i);

    return primes;
}
```

1.2.3 trialDivision

```
#include "../../../headers/headers.h"

// O(sqrt(n)/log(sqrt(n))+log(n))
vi trialDivision(int n, vi &primes)
{
    vi factors;
    for (auto p : primes)
    {
        if (p * p > n)
            break;
        while (n % p == 0)
        {
            primes.pb(p);
            if ((n /= p) == 1)
                return factors;
        }
```

```
    }
    if (n > 1)
        factors.pb(n);

    return factors;
}
```

1.3 Others

1.3.1 Polynomials

```
#include "../../headers/headers.h"

template <class T>
class Pol
{
private:
    vector<T> cofs;
    int n;

public:
    Pol(vector<T> cofs) : cofs(cofs)
    {
        this->n = cofs.size() - 1;
    }

    Pol<T> operator+(const Pol<T> &o)
    {
        vector<T> n_cofs;
        if (n > o.n)
        {
            n_cofs = cofs;
            rep(i, o.n + 1)
            {
                n_cofs[i] += o.cofs[i];
            }
        }
        else
        {
            n_cofs = o.cofs;
            rep(i, n + 1)
            {
                n_cofs[i] += cofs[i];
            }
        }
        return Pol(n_cofs);
    }

    Pol<T> operator-(const Pol<T> &o)
    {
        vector<T> n_cofs;
        if (n > o.n)
        {
            n_cofs = cofs;
            rep(i, o.n + 1)
            {
                n_cofs[i] -= o.cofs[i];
            }
        }
        else
        {
            n_cofs = o.cofs;
            rep(i, n + 1)
            {
                n_cofs[i] *= -1;
                n_cofs[i] += cofs[i];
            }
        }
        return Pol(n_cofs);
    }

    Pol<T> operator*(const Pol<T> &o) //Use Fast Fourier Transform when
         we implement it
    {
        vector<T> n_cofs(n + o.n + 1);
        rep(i, n + 1)
        {
            rep(j, o.n + 1)
            {
                n_cofs[i + j] += cofs[i] * o.cofs[j];
            }
        }
        return Pol(n_cofs);
    }

    Pol<T> operator*(const T &o)
    {
        vector<T> n_cofs = cofs;
        for (auto &cof : n_cofs)
        {
            cof *= o;
        }
        return Pol(n_cofs);
    }

    double operator()(double x)
    {
        double ans = 0;
        double temp = 1;
        for (auto cof : cofs)
        {
            ans += (double)cof * temp;
            temp *= x;
        }
        return ans;
    }

    Pol<T> integrate()
    {
        vector<T> n_cofs(n + 2);
```

```
        repx(i, 1, n_cofs.size())
        {
            n_cofs[i] = cofs[i - 1] / T(i);
        }
        return Pol<T>(n_cofs);
    }

    double integrate(T a, T b)
    {
        Pol<T> temp = integrate();
        return temp(b) - temp(a);
    }

    friend ostream &operator<<(ostream &str, const Pol &a);
};

ostream &operator<<(ostream &strm, const Pol<double> &a)
{
    bool flag = false;
    rep(i, a.n + 1)
    {
        if (a.cofs[i] == 0)
            continue;

        if (flag)
            if (a.cofs[i] > 0)
                strm << " + ";
            else
                strm << " - ";
        else
            flag = true;
        if (i > 1)
        {
            if (abs(a.cofs[i]) != 1)
                strm << abs(a.cofs[i]);
            strm << "x^" << i;
        }
        else if (i == 1)
        {
            if (abs(a.cofs[i]) != 1)
                strm << abs(a.cofs[i]);
            strm << "x";
        }
        else
        {
            strm << a.cofs[i];
        }
    }
    return strm;
}
```

1.3.2 Factorial Factorization

```
#include "../../headers/headers.h"

// O(n)
umap<int, int> factorialFactorization(int n, vi &primes)
{
    umap<int, int> p2e;
    for (auto p : primes)
    {
        if (p > n)
            break;
        int e = 0;
        int tmp = n;
        while ((tmp /= p) > 0)
            e += tmp;
        if (e > 0)
            p2e[p] = e;
    }
    return p2e;
}
```