数学

取模类

```
using i64 = long long;
template<class T>
constexpr T power(T a, i64 b) {
    T res = 1;
    for (; b; b /= 2, a *= a) {
        if (b % 2) {
            res *= a;
        }
    }
    return res;
}

constexpr i64 mul(i64 a, i64 b, i64 p) {
    i64 res = a * b - i64(1.L * a * b / p) * p;
    res %= p;
    if (res < 0) {
        res += p;
    }
    return res;
}
template<i64 P>
struct MLong {
    i64 x;
    constexpr MLong() : x{} {}
    constexpr MLong(i64 x) : x{norm(x % getMod())} {}

    static i64 Mod;
    constexpr static i64 getMod() {
        if (P > 0) {
            return P;
        } else {
            return Mod;
        }
    }
    constexpr static void setMod(i64 Mod_) {
        Mod = Mod_;
    }
    constexpr i64 norm(i64 x) const {
        if (x < 0) {
            x += getMod();
        }
        if (x >= getMod()) {
            x -= getMod();
        }
        return x;
    }
    constexpr i64 val() const {
        return x;
    }
```

```
    explicit constexpr operator i64() const {
        return x;
    }
    constexpr MLong operator-() const {
        MLong res;
        res.x = norm(getMod() - x);
        return res;
    }
    constexpr MLong inv() const {
        assert(x != 0);
        return power(*this, getMod() - 2);
    }
    constexpr MLong &operator*=(MLong rhs) & {
        x = mul(x, rhs.x, getMod());
        return *this;
    }
    constexpr MLong &operator+=(MLong rhs) & {
        x = norm(x + rhs.x);
        return *this;
    }
    constexpr MLong &operator-=(MLong rhs) & {
        x = norm(x - rhs.x);
        return *this;
    }
    constexpr MLong &operator/=(MLong rhs) & {
        return *this *= rhs.inv();
    }
    friend constexpr MLong operator*(MLong lhs, MLong rhs) {
        MLong res = lhs;
        res *= rhs;
        return res;
    }
    friend constexpr MLong operator+(MLong lhs, MLong rhs) {
        MLong res = lhs;
        res += rhs;
        return res;
    }
    friend constexpr MLong operator-(MLong lhs, MLong rhs) {
        MLong res = lhs;
        res -= rhs;
        return res;
    }
    friend constexpr MLong operator/(MLong lhs, MLong rhs) {
        MLong res = lhs;
        res /= rhs;
        return res;
    }
    friend constexpr std::istream &operator>>(std::istream &is, MLong &a) {
        i64 v;
        is >> v;
        a = MLong(v);
        return is;
    }
    friend constexpr std::ostream &operator<<(std::ostream &os, const MLong &a) {
        return os << a.val();
    }
```

```
    friend constexpr bool operator==(MLong lhs, MLong rhs) {
        return lhs.val() == rhs.val();
    }
    friend constexpr bool operator!=(MLong lhs, MLong rhs) {
        return lhs.val() != rhs.val();
    }
};

template<>
i64 MLong<0LL>::Mod = i64(1E18) + 9;

template<int P>
struct MInt {
    int x;
    constexpr MInt() : x{} {}
    constexpr MInt(i64 x) : x{norm(x % getMod())} {}

    static int Mod;
    constexpr static int getMod() {
        if (P > 0) {
            return P;
        } else {
            return Mod;
        }
    }
    constexpr static void setMod(int Mod_) {
        Mod = Mod_;
    }
    constexpr int norm(int x) const {
        if (x < 0) {
            x += getMod();
        }
        if (x >= getMod()) {
            x -= getMod();
        }
        return x;
    }
    constexpr int val() const {
        return x;
    }
    explicit constexpr operator int() const {
        return x;
    }
    explicit constexpr operator i64() const {
        return x;
    }
    constexpr MInt operator-() const {
        MInt res;
        res.x = norm(getMod() - x);
        return res;
    }
    constexpr MInt inv() const {
        assert(x != 0);
        return power(*this, getMod() - 2);
    }
    constexpr MInt &operator*=(MInt rhs) & {
```

```
        x = 1LL * x * rhs.x % getMod();
        return *this;
    }
    constexpr MInt &operator+=(MInt rhs) & {
        x = norm(x + rhs.x);
        return *this;
    }
    constexpr MInt &operator-=(MInt rhs) & {
        x = norm(x - rhs.x);
        return *this;
    }
    constexpr MInt &operator/=(MInt rhs) & {
        return *this *= rhs.inv();
    }
    friend constexpr MInt operator*(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res *= rhs;
        return res;
    }
    friend constexpr MInt operator+(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res += rhs;
        return res;
    }
    friend constexpr MInt operator-(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res -= rhs;
        return res;
    }
    friend constexpr MInt operator/(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res /= rhs;
        return res;
    }
    friend constexpr std::istream &operator>>(std::istream &is, MInt &a) {
        i64 v;
        is >> v;
        a = MInt(v);
        return is;
    }
    friend constexpr std::ostream &operator<<(std::ostream &os, const MInt &a) {
        return os << a.val();
    }
    friend constexpr bool operator==(MInt lhs, MInt rhs) {
        return lhs.val() == rhs.val();
    }
    friend constexpr bool operator!=(MInt lhs, MInt rhs) {
        return lhs.val() != rhs.val();
    }
};

template<>
int MInt<0>::Mod = 998244353;

template<int V, int P>
constexpr MInt<P> CInv = MInt<P>(V).inv();
```

```
constexpr int P = 998244353;
using Z = MInt<P>;
```

多项式

标准

```
std::vector<int> rev;
template<int P>
std::vector<MInt<P>> roots{0, 1};

template<int P>
constexpr MInt<P> findPrimitiveRoot() {
    MInt<P> i = 2;
    int k = __builtin_ctz(P - 1);
    while (true) {
        if (power(i, (P - 1) / 2) != 1) {
            break;
        }
        i += 1;
    }
    return power(i, (P - 1) >> k);
}

template<int P>
constexpr MInt<P> primitiveRoot = findPrimitiveRoot<P>();

template<>
constexpr MInt<998244353> primitiveRoot<998244353> {31};

template<int P>
constexpr void dft(std::vector<MInt<P>> &a) {
    int n = a.size();

    if (int(rev.size()) != n) {
        int k = __builtin_ctz(n) - 1;
        rev.resize(n);
        for (int i = 0; i < n; i++) {
            rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
        }
    }

    for (int i = 0; i < n; i++) {
        if (rev[i] < i) {
            std::swap(a[i], a[rev[i]]);
        }
    }
    if (roots<P>.size() < n) {
        int k = __builtin_ctz(roots<P>.size());
        roots<P>.resize(n);
        while ((1 << k) < n) {
            auto e = power(primitiveRoot<P>, 1 << (__builtin_ctz(P - 1) - k -
1));
```

```
            for (int i = 1 << (k - 1); i < (1 << k); i++) {
                roots<P>[2 * i] = roots<P>[i];
                roots<P>[2 * i + 1] = roots<P>[i] * e;
            }
            k++;
        }
    }
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i += 2 * k) {
            for (int j = 0; j < k; j++) {
                MInt<P> u = a[i + j];
                MInt<P> v = a[i + j + k] * roots<P>[k + j];
                a[i + j] = u + v;
                a[i + j + k] = u - v;
            }
        }
    }
}

template<int P>
constexpr void idft(std::vector<MInt<P>> &a) {
    int n = a.size();
    std::reverse(a.begin() + 1, a.end());
    dft(a);
    MInt<P> inv = (1 - P) / n;
    for (int i = 0; i < n; i++) {
        a[i] *= inv;
    }
}

template<int P = ::P>
struct Poly : public std::vector<MInt<P>> {
    using Value = MInt<P>;

    Poly() : std::vector<Value>() {}
    explicit constexpr Poly(int n) : std::vector<Value>(n) {}

    explicit constexpr Poly(const std::vector<Value> &a) : std::vector<Value>(a)
{}
    constexpr Poly(const std::initializer_list<Value> &a) : std::vector<Value>(a)
{}

    template<class InputIt, class = std::_RequireInputIter<InputIt>>
    explicit constexpr Poly(InputIt first, InputIt last) : std::vector<Value>
(first, last) {}

    template<class F>
    explicit constexpr Poly(int n, F f) : std::vector<Value>(n) {
        for (int i = 0; i < n; i++) {
            (*this)[i] = f(i);
        }
    }

    constexpr Poly shift(int k) const {
        if (k >= 0) {
            auto b = *this;
```

```
            b.insert(b.begin(), k, 0);
            return b;
        } else if (this->size() <= -k) {
            return Poly();
        } else {
            return Poly(this->begin() + (-k), this->end());
        }
    }
    constexpr Poly trunc(int k) const {
        Poly f = *this;
        f.resize(k);
        return f;
    }
    constexpr friend Poly operator+(const Poly &a, const Poly &b) {
        Poly res(std::max(a.size(), b.size()));
        for (int i = 0; i < a.size(); i++) {
            res[i] += a[i];
        }
        for (int i = 0; i < b.size(); i++) {
            res[i] += b[i];
        }
        return res;
    }
    constexpr friend Poly operator-(const Poly &a, const Poly &b) {
        Poly res(std::max(a.size(), b.size()));
        for (int i = 0; i < a.size(); i++) {
            res[i] += a[i];
        }
        for (int i = 0; i < b.size(); i++) {
            res[i] -= b[i];
        }
        return res;
    }
    constexpr friend Poly operator-(const Poly &a) {
        std::vector<Value> res(a.size());
        for (int i = 0; i < int(res.size()); i++) {
            res[i] = -a[i];
        }
        return Poly(res);
    }
    constexpr friend Poly operator*(Poly a, Poly b) {
        if (a.size() == 0 || b.size() == 0) {
            return Poly();
        }
        if (a.size() < b.size()) {
            std::swap(a, b);
        }
        int n = 1, tot = a.size() + b.size() - 1;
        while (n < tot) {
            n *= 2;
        }
        if (((P - 1) & (n - 1)) != 0 || b.size() < 128) {

            Poly c(a.size() + b.size() - 1);
            for (int i = 0; i < a.size(); i++) {
                for (int j = 0; j < b.size(); j++) {
```

```
                    c[i + j] += a[i] * b[j];
                }
            }
            return c;
        }
        a.resize(n);
        b.resize(n);
        dft(a);
        dft(b);
        for (int i = 0; i < n; ++i) {
            a[i] *= b[i];
        }
        idft(a);
        a.resize(tot);
        return a;
    }
    constexpr friend Poly operator*(Value a, Poly b) {
        for (int i = 0; i < int(b.size()); i++) {
            b[i] *= a;
        }
        return b;
    }
    constexpr friend Poly operator*(Poly a, Value b) {
        for (int i = 0; i < int(a.size()); i++) {
            a[i] *= b;
        }
        return a;
    }
    constexpr friend Poly operator/(Poly a, Value b) {
        for (int i = 0; i < int(a.size()); i++) {
            a[i] /= b;
        }
        return a;
    }
    constexpr Poly &operator+=(Poly b) {
        return (*this) = (*this) + b;
    }
    constexpr Poly &operator-=(Poly b) {
        return (*this) = (*this) - b;
    }
    constexpr Poly &operator*=(Poly b) {
        return (*this) = (*this) * b;
    }
    constexpr Poly &operator*=(Value b) {
        return (*this) = (*this) * b;
    }
    constexpr Poly &operator/=(Value b) {
        return (*this) = (*this) / b;
    }
    template <class T>
    constexpr Value operator() ( T x ) {
        Value ans = 0 ;
        Value cnt = 1 ;
        for ( int i = 0 ; i < this->size () ; ++ i ) {
            ans += (* this) [ i ] * cnt ;
            cnt *= x ;
```

```
        }
        return ans ;
    }
    constexpr Poly deriv() const {
        if (this->empty()) {
            return Poly();
        }
        assert (this->size() != 0) ;
        Poly res(this->size() - 1);
        for (int i = 0; i < this->size() - 1; ++i) {
            res[i] = (i + 1) * (*this)[i + 1];
        }
        return res;
    }
    constexpr Poly integr() const {
        Poly res(this->size() + 1);
        for (int i = 0; i < this->size(); ++i) {
            res[i + 1] = (*this)[i] / (i + 1);
        }
        return res;
    }
    constexpr Poly inv(int m) const {
        Poly x{(*this)[0].inv()};
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly{2} - trunc(k) * x)).trunc(k);
        }
        return x.trunc(m);
    }
    constexpr Poly log(int m) const {
        return (deriv() * inv(m)).integr().trunc(m);
    }
    constexpr Poly exp(int m) const {
        Poly x{1};
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly{1} - x.log(k) + trunc(k))).trunc(k);
        }
        return x.trunc(m);
    }
    constexpr Poly pow(int k, int m) const {
        int i = 0;
        while (i < this->size() && (*this)[i] == 0) {
            i++;
        }
        if (i == this->size() || 1LL * i * k >= m) {
            return Poly(m);
        }
        Value v = (*this)[i];
        auto f = shift(-i) * v.inv();
        return (f.log(m - i * k) * k).exp(m - i * k).shift(i * k) * power(v, k);
    }
    constexpr Poly pow(int k, int m, int k2) const {
        int i = 0;
```

```
        while (i < this->size() && (*this)[i] == 0) {
            i++;
        }
        if (i == this->size() || 1LL * i * k >= m) {
            return Poly(m);
        }
        Value v = (*this)[i];
        auto f = shift(-i) * v.inv();
        return (f.log(m - i * k) * k).exp(m - i * k).shift(i * k) * power(v, k2);
    }
    constexpr Poly sqrt(int m) const {
        Poly x{1};
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x + (trunc(k) * x.inv(k)).trunc(k)) * CInv<2, P>;
        }
        return x.trunc(m);
    }
    constexpr Poly inv() const {
        return move (inv(this->size ())) ;
    }
    constexpr Poly log() const {
        return move(log(this->size ()));
    }
    constexpr Poly exp() const {
        return move(exp(this->size ()));
    }
    constexpr Poly pow(i64 b) const {
        Poly<> res (vector <Z> { 1 }) ;
        auto a = * this ;
        for (; b; b /= 2, a *= a) {
            if (b % 2) {
                res *= a;
            }
        }
        return res;
    }
    constexpr Poly sqrt() const {
        return move(sqrt(this->size()));
    }
    constexpr Poly mulT(Poly b) const {
        if (b.size() == 0) {
            return Poly();
        }
        int n = b.size();
        std::reverse(b.begin(), b.end());
        return ((*this) * b).shift(-(n - 1));
    }
    constexpr std::vector<Value> eval(std::vector<Value> x) const {
        if (this->size() == 0) {
            return std::vector<Value>(x.size(), 0);
        }
        const int n = std::max(x.size(), this->size());
        std::vector<Poly> q(4 * n);
        std::vector<Value> ans(x.size());
```

```
        x.resize(n);
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                q[p] = Poly{1, -x[l]};
            } else {
                int m = (l + r) / 2;
                build(2 * p, l, m);
                build(2 * p + 1, m, r);
                q[p] = q[2 * p] * q[2 * p + 1];
            }
        };
        build(1, 0, n);
        std::function<void(int, int, int, const Poly &)> work = [&](int p, int l,
int r, const Poly &num) {
            if (r - l == 1) {
                if (l < int(ans.size())) {
                    ans[l] = num[0];
                }
            } else {
                int m = (l + r) / 2;
                auto need = move(num.mulT(q[2 * p + 1]));
                need.resize ( m - l ) ;
                work(2 * p, l, m, need);
                need = move(num.mulT(q[2 * p]));
                need.resize ( r - m ) ;
                work(2 * p + 1, m, r, need);
            }
        };
        work(1, 0, n, mulT(q[1].inv(n)));
        return ans;
    }
};

template<int P = ::P>
Poly<P> berlekampMassey(const Poly<P> &s) {
    Poly<P> c;
    Poly<P> oldC;
    int f = -1;
    for (int i = 0; i < s.size(); i++) {
        auto delta = s[i];
        for (int j = 1; j <= c.size(); j++) {
            delta -= c[j - 1] * s[i - j];
        }
        if (delta == 0) {
            continue;
        }
        if (f == -1) {
            c.resize(i + 1);
            f = i;
        } else {
            auto d = oldC;
            d *= -1;
            d.insert(d.begin(), 1);
            MInt<P> df1 = 0;
            for (int j = 1; j <= d.size(); j++) {
                df1 += d[j - 1] * s[f + 1 - j];
```

```
            }
            assert(df1 != 0);
            auto coef = delta / df1;
            d *= coef;
            Poly<P> zeros(i - f - 1);
            zeros.insert(zeros.end(), d.begin(), d.end());
            d = zeros;
            auto temp = c;
            c += d;
            if (i - temp.size() > f - oldC.size()) {
                oldC = temp;
                f = i;
            }
        }
    }
    c *= -1;
    c.insert(c.begin(), 1);
    return c;
}


template<int P = ::P>
MInt<P> linearRecurrence(Poly<P> p, Poly<P> q, i64 n) {
    int m = q.size() - 1;
    while (n > 0) {
        auto newq = q;
        for (int i = 1; i <= m; i += 2) {
            newq[i] *= -1;
        }
        auto newp = p * newq;
        newq = q * newq;
        for (int i = 0; i < m; i++) {
            p[i] = newp[i * 2 + n % 2];
        }
        for (int i = 0; i <= m; i++) {
            q[i] = newq[i * 2];
        }
        n /= 2;
    }
    return p[0] / q[0];
}
```

c++版本修复

```
# include <vector>
# include <tuple>   // for std::tie
# include <utility> // for std::make_pair

using cp = complex<double>;
using _Tp = vector<cp>::iterator;

struct cxx20_It : public _Tp {
    using _Tp::_Tp;
    cxx20_It(const _Tp& it) : _Tp(it) {}
    cp &operator [](int x) {
```

```
        return *(*this + x);
    }
};
using It = cxx20_It;
```

FFT

```
const double PI = acos(-1.0);

struct Complex {
    double x, y;
    Complex(double _x = 0.0, double _y = 0.0) {
        x = _x;
        y = _y;
    }
    Complex operator-(const Complex &b) const {
        return Complex(x - b.x, y - b.y);
    }
    Complex operator+(const Complex &b) const {
        return Complex(x + b.x, y + b.y);
    }
    Complex operator*(const Complex &b) const {
        return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
    }
    friend ostream &operator<<(ostream &cout, Complex u) {
        return cout << "(" << u.x << ", " << u.y << ")";
    }
};

void change(vector<Complex> &y) {
    int len = y.size();
    for (int i = 1, j = len / 2; i < len - 1; i++) {
        if (i < j) std::swap(y[i], y[j]);
        int k = len / 2;
        while (j >= k) {
            j = j - k;
            k = k / 2;
        }
        if (j < k) j += k;
    }
}

void fft(vector<Complex> &y, int on) {
    int len = y.size();
    change(y);
    for (int h = 2; h <= len; h <<= 1) {
        Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));
        for (int j = 0; j < len; j += h) {
            Complex w(1, 0);
            for (int k = j; k < j + h / 2; k++) {
                Complex u = y[k];
                Complex t = w * y[k + h / 2];
                y[k] = u + t;
                y[k + h / 2] = u - t;
                w = w * wn;
```

```
            }
        }
    }
    if (on == -1) {
        for (int i = 0; i < len; i++) {
            y[i].x /= len;
        }
    }
}

struct Poly : public vector<double> {
    using std::vector<double>::vector;
    friend Poly operator+(Poly x, Poly y) {
        int n = std::max(x.size(), y.size());
        Poly z(n);
        for (int i = 0; i < x.size(); i += 1) {
            z[i] += x[i];
        }
        for (int i = 0; i < y.size(); i += 1) {
            z[i] += y[i];
        }
        return z;
    }
    friend Poly operator*(Poly x, Poly y) {
        int len = x.size() + y.size();
        int n = 1;
        while (n < len) {
            n *= 2;
        }
        vector<Complex> _x(n), _y(n);
        for (int i = 0; i < x.size(); i += 1) {
            _x[i].x = x[i];
        }
        for (int i = 0; i < y.size(); i += 1) {
            _y[i].x = y[i];
        }
        fft(_x, 1), fft(_y, 1);
        for (int i = 0; i < n; i += 1) {
            _x[i] = _x[i] * _y[i];
        }
        fft(_x, -1);
        Poly ans(n);
        for (int i = 0; i < n; i += 1) {
            ans[i] = _x[i].x;
        }
        return ans;
    }
    Poly operator-() {
        Poly a = *this;
        for (auto &i : a) {
            i = -i;
        }
        return a;
    }
    friend Poly operator-(Poly x, Poly y) {
        return x + -y;
```

```
    }

    friend Poly operator*(Poly x, int _mul) {
        for(int i = 0; i < x.size(); ++i) {
            x[i] *= _mul;
        }
        return x;
    };
    friend Poly operator*(int _mul, Poly x) {
        return x * _mul;
    };

    Poly &operator+=(Poly y) {
        return *this = *this + y;
    }
    Poly &operator-=(Poly y) {
        return *this = *this - y;
    }
    Poly &operator*=(Poly y) {
        return *this = *this * y;
    }

    Poly &operator*=(int y) {
        return *this = *this * y;
    }
    template<typename T>
    operator vector<T>() {
        vector<T> a(size());
        for (int i = 0; i < size(); i += 1) {
            a[i] = T(this->operator[](i) + 0.5);
        }
        return a;
    }
};
```

更快的FFT

```
constexpr double pi = 3.14159265358979311599796346854418516159057617187 5, pi2 = 2
* pi;

using cp = complex<double>;
using It = vector<cp>::iterator;
vector<cp> a;

template<int on, int n>
void fft(It a) {
    if (n == 1)
        return;
    if (n == 2) {
        tie(a[0], a[1]) = make_pair(a[0] + a[1], a[0] - a[1]);
        return;
    }
```

```
    constexpr int h = n >> 1, q = n >> 2;
    if (on == -1) {
        fft<on, h>(a);
        fft<on, q>(a + h);
        fft<on, q>(a + h + q);
    }
    cp w(1, 0), w3(1, 0);
    constexpr cp wn(cos(pi2 / n), on * sin(pi2 / n)),
                 wn3(cos(pi2 * 3 / n), on * sin(pi2 * 3 / n));
    for (int i = 0; i < q; i++) {
        if (on == -1) {
            cp tmp1 = w * a[i + h], tmp2 = w3 * a[i + h + q],
               x = a[i], y = tmp1 + tmp2,
               x1 = a[i + q], y1 = tmp1 - tmp2;
            y1 = cp(y1.imag(), -y1.real());
            a[i] += y;
            a[i + q] += y1;
            a[i + h] = x - y;
            a[i + h + q] = x1 - y1;
        } else {
            cp x = a[i] - a[i + h], y = a[i + q] - a[i + h + q];
            y = cp(y.imag(), -y.real());
            a[i] += a[i + h];
            a[i + q] += a[i + h + q];
            a[i + h] = (x - y) * w;
            a[i + h + q] = (x + y) * w3;
        }
        w *= wn;
        w3 *= wn3;
    }
    if (on == 1) {
        fft<on, h>(a);
        fft<on, q>(a + h);
        fft<on, q>(a + h + q);
    }
}
template<>
void fft<1, 0> (It a) {}
template<>
void fft<-1, 0> (It a) {}


template<int on>
void FFT(It a, int n) {
    # define C(x)\
        case 1 << x:\
            fft<on, 1 << x>(a);\
            break
    switch (n) {
        C(1);
        C(2);
        C(3);
        C(4);
        C(5);
        C(6);
        C(7);
```

```
        C(8);
        C(9);
        C(10);
        C(11);
        C(12);
        C(13);
        C(14);
        C(15);
        C(16);
        C(17);
        C(18);
        C(19);
        C(20);
        C(21);
    }
    # undef C
}

vector<cp> _x;
struct Poly : public vector<double> {
    using std::vector<double>::vector;
    friend Poly operator+(Poly x, Poly y) {
        int n = std::max(x.size(), y.size());
        Poly z(n);
        for (int i = 0; i < x.size(); i += 1) {
            z[i] += x[i];
        }
        for (int i = 0; i < y.size(); i += 1) {
            z[i] += y[i];
        }
        return z;
    }
    friend Poly operator*(Poly &x, Poly &y) {
        int len = x.size() + y.size() + 1;
        int n = 1;
        while (n < len) {
            n *= 2;
        }
        _x.assign(n, {});
        for (int i = 0; i < x.size(); i += 1) {
            _x[i].real(x[i]);
        }
        for (int i = 0; i < y.size(); i += 1) {
            _x[i].imag(y[i]);
        }
        FFT<1>(_x.begin(), n);
        for (int i = 0; i < n; i += 1) {
            _x[i] *= _x[i];
        }
        FFT<-1>(_x.begin(), n);
        Poly ans(n);
        const double inv = 0.5 / n;
        for (int i = 0; i < n; i += 1) {
            ans[i] = _x[i].imag() * inv;
        }
        return ans;
```

```
    }
    Poly operator-() {
        Poly a = *this;
        for (auto &i : a) {
            i = -i;
        }
        return a;
    }
    friend Poly operator-(Poly x, Poly y) {
        return x + -y;
    }
    Poly &operator+=(Poly y) {
        return *this = *this + y;
    }
    Poly &operator-=(Poly y) {
        return *this = *this - y;
    }
    Poly &operator*=(Poly y) {
        return *this = *this * y;
    }
    template<typename T>
    operator vector<T>() {
        vector<T> a(size());
        for (int i = 0; i < size(); i += 1) {
            a[i] = T(this->operator[](i) + 0.5);
        }
        return a;
    }
};
```

更快的NTT

```
#include <bits/stdc++.h>
#include <immintrin.h>
using namespace std;

using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using ci = const int;

constexpr int inSZ = 1 << 17, outSZ = 1 << 21;
char ibuf[inSZ], *in1 = ibuf, *in2 = ibuf;
char obuf[outSZ], *out1 = obuf, *out2 = obuf + outSZ;

inline char gc() {
    if (__builtin_expect(in1 == in2 && (in2 = (in1 = ibuf) +
        fread(ibuf, 1, inSZ, stdin), in1 == in2), 0)) return EOF;
    return *in1++;
}

inline void flush() {
    fwrite(obuf, 1, out1 - obuf, stdout);
    out1 = obuf;
}
```

```
#define pc(c) (*out1++ = c)

inline void read(int &x) {
    x = 0; static char c;
    while (!isdigit((c = gc())));
    while (x = 10 * x + (c ^ 48), isdigit(c = gc()));
}

inline void write(int x) {
    if (__builtin_expect(out1 + 20 > out2, 0)) flush();
    int tot = 0;
    do { pc(x % 10 + 48); } while (++tot, x /= 10);
    reverse(out1 - tot, out1);
}

constexpr int N = 1 << 21;
constexpr int mod = 998244353, g = 3;
inline int dil(int x) { return x >> 31 ? x + mod : x; }
inline int mu(int x, int y) { return u64(x) * y % mod; }

inline int qpow(int x, int y) {
    int z = 1;
    do { if (y & 1) z = mu(z, x); x = mu(x, x); } while (y >>= 1);
    return z;
}

inline int bceil(int x) { return 1 << __lg(x - 1) + 1; }

int w[N >> 1], iw[N >> 1];
void preNTT(int n) {
    int l = bceil(n) >> 1;
    w[0] = iw[0] = 1;
    for (int i = 1; i < l; i <<= 1) {
        w[i] = qpow(g, (mod - 1 >> 2) / i);
        iw[i] = qpow(g, mod - 1 - (mod - 1 >> 2) / i);
    }
    for (int i = 1; i < l; ++i) {
        w[i] = mu(w[i & (i - 1)], w[i & -i]);
        iw[i] = mu(iw[i & (i - 1)], iw[i & -i]);
    }
}

struct poly : vector<int> {
    friend void dif(poly &f, int lim) {
        f.resize(lim);
        for (int l = lim >> 1, r = lim; l; l >>= 1, r >>= 1)
            for (int i = 0, *o = w; i != lim; i += r, ++o)
                for (int j = i, x, y; j != i + l; ++j)
                    x = dil(f[j] - mod), y = mu(f[j + l], *o), f[j] = x + y, f[j
+ l] = x - y + mod;
        for (int i = 0; i < lim; ++i) f[i] = dil(f[i] - mod);
    }
    friend void dit(poly &f, int lim) {
        f.resize(lim);
```

```
        for (int l = 1, r = 2; l < lim; l <<= 1, r <<= 1)
            for (int i = 0, *o = iw; i != lim; i += r, ++o)
                for (int j = i, x, y; j != i + l; ++j)
                    x = f[j], y = mod - f[j + l], f[j] = dil(x - y), f[j + l] =
mu(x + y, *o);
        ci iv = mod - (mod - 1) / lim;
        for (int i = 0; i < lim; ++i) f[i] = mu(f[i], iv);
    }
    friend poly operator*(poly f, poly g) {
        int len = f.size() + g.size() - 1;
        int lim = bceil(len);
        f.resize(lim), g.resize(lim);
        preNTT(lim); dif(f, lim); dif(g, lim);
        for (int i = 0; i < lim; ++i) f[i] = mu(f[i], g[i]);
        dit(f, lim); f.resize(len);
        return forward<poly>(f);
    }
};
poly F, G;

int main() {
    int n, m, lim;
    read(n), read(m);
    F.resize(n + 1), G.resize(m + 1);
    lim = bceil(n + m + 1);
    for (int i = 0; i <= n; ++i) read(F[i]);
    for (int i = 0; i <= m; ++i) read(G[i]);
    F = F * G;
    for (int i = 0; i <= n + m; ++i) write(F[i]), pc(' ');

    return flush(), 0;
}
```

多项式扩展包

```
/**
 * 多项式扩展包
*/
namespace ExPoly {
    template<int P = ::P, class T1, class T2>
    constexpr static Poly <P> Lagrange(T1 x, T2 y) {
        int n = x.size();
        vector <Poly<>> M(4 * n);
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                M[p] = Poly{(int) -x[l], 1};
            } else {
                int m = (l + r) / 2;
                build(2 * p, l, m);
                build(2 * p + 1, m, r);
                M[p] = M[2 * p] * M[2 * p + 1];
            }
        };
```

```
        build(1, 0, n);
        auto M_ = M[1].deriv().eval(x);
        for (int i = 0; i < n; ++i) {
            M_[i] = y[i] * M_[i].inv();
        }
        vector <Poly<>> f(4 * n);
        std::function<void(int, int, int)> work = [&](int p, int l, int r) ->
void {
            if (r - l == 1) {
                if (l < n) {
                    f[p] = Poly{(int) M_[l]};
                }
            } else {
                int m = (l + r) / 2;
                work(2 * p, l, m);
                work(2 * p + 1, m, r);
                f[p] = f[2 * p] * M[2 * p + 1] + f[2 * p + 1] * M[2 * p];
            }
        };
        work(1, 0, n);
        return f[1];
    }

/**
 *作用：对多项式进行平移操作
 *时间复杂度O(nlog(n))
 */
    template<int P = ::P>
    constexpr static Poly <P> Polynomial_translation(Poly <P> f, int k) {
        i64 n = (i64) f.size() - 1;
        Poly <P> g(n + 1);
        Z res = 1;
        for (int i = 0; i <= n; ++i) {
            g[n - i] = res * comb.invfac(i);
            res *= k;
            f[i] *= comb.fac(i);
        }
        Poly <P> here = g * f;
        here = here.shift(-n);
        for (int i = 0; i <= n; ++i) {
            here[i] *= comb.invfac(i);
        }
        return here;
    }

/**
 *作用：对相同的n对i \in ( 0 , n ) 求出将n个不同的元素划分为i个非空集的方案数
 *第二类Stirling数
 *时间复杂度O(nlog(n))
 */
    template<int P = ::P>
    constexpr static Poly <P> Second_Stirling_Same_N(int n) {
        Poly <P> f(n + 1), g(n + 1);
        for (int i = 0; i <= n; ++i) {
            g[i] = (i & 1 ? (Z) - 1 : Z(1)) * comb.invfac(i);
            f[i] = power((Z) i, n) * comb.invfac(i);
```

```
        }
        f *= g;
        f.resize(n + 1);
        return f;
    }

/**
 *作用：对相同的k对不同n 求出将n个不同的元素划分为k个非空集的方案数
 *第二类Stirling数
 *时间复杂度O(nlog(n))
 */
    template<int P = ::P>
    constexpr static Poly <P> Second_Stirling_Same_K(int Max_n, int k) {
        comb.init(Max_n + 1);
        Poly <P> f(vector<Z>(comb._invfac.begin(), comb._invfac.begin() + Max_n +
1));
        f[0] = 0;
        f = f.pow(k, Max_n + 1);
        for (int i = 0; i <= Max_n; ++i) {
            f[i] = f[i] * comb.fac(i) * comb.invfac(k);
        }
        return f;
    }

/**
 *作用：对相同的n对i \in ( 0 , n ) 求出将n个不同的元素划分为i个非空轮换的方案数
 *第一类Stirling数
 *时间复杂度O(nlog(n))
 */
    template<int P = ::P>
    constexpr static Poly <P> First_Stirling_Same_N(int n) {
        ll len = __lg(n);
        Poly <P> f = {1};
        ll cnt = 0;
        for (int i = len; i >= 0; --i) {
            f *= Polynomial_translation(f, cnt);
            cnt <<= 1;
            if (n >> i & 1) f *= Poly{cnt, 1}, cnt += 1;
        }
        return f;
    }

/**
 *作用：对相同的k对不同n 求出将n个不同的元素划分为k个非轮换的方案数
 *第一类Stirling数
 *时间复杂度O(nlog(n))
 */
    template<int P = ::P>
    constexpr static Poly <P> First_Stirling_Same_K(int Max_n, int k) {
        comb.init(Max_n + 1);
        Poly <P> f(comb._inv.begin(), comb._inv.begin() + Max_n + 1);
        f = f.pow(k, Max_n + 1);
        for (int i = 0; i <= Max_n; ++i) {
            f[i] *= comb.fac(i) * comb.invfac(k);
        }
        return f;
```

```
    }
};
```

矩阵

```
namespace matrix {
    using i64 = long long;

    template<typename T>
    struct Matrix : public std::vector<std::vector<T>> {
        using std::vector<std::vector<T>>::vector;

        Matrix(int x) : std::vector<std::vector<T>>(x, std::vector<T>(x)) {};
        Matrix(int x, int y) : std::vector<std::vector<T>>(x, std::vector<T>(y))
{};
        Matrix(int x, int y, T c) : std::vector<std::vector<T>>(x, std::vector<T>
(y, c)) {};

        constexpr Matrix operator+(Matrix a);
        constexpr Matrix operator-(Matrix a);
        constexpr Matrix operator*(Matrix a);

        template <typename T1, typename T2>
        friend constexpr Matrix<T1> operator*(Matrix<T1> x, T2 a);

        constexpr Matrix& operator+=(Matrix a);
        constexpr Matrix& operator-=(Matrix a);
        constexpr Matrix& operator*=(Matrix a);

        template <typename T1, typename T2>
        friend constexpr Matrix<T1>& operator*=(Matrix<T1>& x, T2 a);

        constexpr Matrix pow(i64 b);
        constexpr Matrix Transpose();
        constexpr Matrix inv();
    };

    template <typename T>
    constexpr Matrix<T> Matrix<T>::operator+(Matrix<T> a) {
        auto it = *this;
        int n = (int)a.size();
        int m = (int)a.back().size();
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < m; ++j)
                it[i][j] += a[i][j];
        return it;
    }

    template <typename T>
    constexpr Matrix<T> Matrix<T>::operator-(Matrix<T> a) {
        auto it = *this;
        int n = (int)a.size();
        int m = (int)a.back().size();
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < m; ++j)
```

```
            it[i][j] -= a[i][j];
    return it;
}

template <typename T>
constexpr Matrix<T> Matrix<T>::operator*(Matrix<T> a) {
    int n = (int)this->size();
    int mid = (int)a.size();
    int m = (int)a.back().size();
    Matrix<T> it(n, m);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            for (int k = 0; k < mid; ++k)
                it[i][j] += (*this)[i][k] * a[k][j];
    return it;
}

template <typename T1, typename T2>
constexpr Matrix<T1> operator*(Matrix<T1> x, T2 a) {
    int n = (int)x.size();
    int m = (int)x.back().size();
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            x[i][j] *= a;
    return x;
}

template <typename T>
constexpr Matrix<T>& Matrix<T>::operator+=(Matrix<T> a) {
    return *this = *this + a;
}

template <typename T>
constexpr Matrix<T>& Matrix<T>::operator-=(Matrix<T> a) {
    return *this = *this - a;
}

template <typename T>
constexpr Matrix<T>& Matrix<T>::operator*=(Matrix<T> a) {
    return *this = *this * a;
}

template <typename T1, typename T2>
constexpr Matrix<T1>& operator*=(Matrix<T1>& x, T2 a) {
    return x = x * a;
}

template <typename T>
constexpr Matrix<T> Matrix<T>::pow(i64 b) {
    auto res = Matrix(this->size(), this->size());
    for (int i = 0; i < (int)this->size(); ++i)
        res[i][i] = 1;
    auto a = *this;
    for (; b; b /= 2, a *= a)
        if (b % 2) res *= a;
    return res;
```

```
    }

    template <typename T>
    constexpr Matrix<T> Matrix<T>::Transpose() {
        int n = this->back().size(), m = this->size();
        auto it = Matrix(n, m);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < m; ++j)
                it[i][j] = (*this)[j][i];
        return it;
    }

    template <typename T>
    constexpr Matrix<T> Matrix<T>::inv() {
        int n = this->size();
        Matrix<T> it(n, 2 * n);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                it[i][j] = (*this)[i][j];
        for (int i = 0; i < n; ++i)
            it[i][i + n] = 1;
        for (int i = 0; i < n; ++i) {
            int r = i;
            for (int k = i; k < n; ++k)
                if ((i64)it[k][i]) { r = k; break; }
            if (r != i)
                swap(it[r], it[i]);
            if (!(i64)it[i][i])
                return Matrix<T>();

            T x = (T) 1 / it[i][i];
            for (int k = 0; k < n; ++k) {
                if (k == i)
                    continue;
                T t = it[k][i] * x;
                for (int j = i; j < 2 * n; ++j)
                    it[k][j] -= t * it[i][j];
            }
            for (int j = 0; j < 2 * n; ++j)
                it[i][j] *= x;
        }
        Matrix<T> ans(n, n);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                ans[i][j] = it[i][j + n];
        return ans;
    }
    // namespace Matrix
};
```

数学类

base

```
int mul(int a, int b, int P) {
    return 1ll * a * b % P;
}

template<typename T>
T power(T a, i64 b, i64 P) {
    T res = 1;
    for (; b; b >>= 1) {
        if (b & 1) {
            res = 1ll * res * a % P;
        }
        a = 1ll * a * a % P;
    }
    return res;
}

int sum2(int a) {
    return a * (a + 1) * (2 * a + 1) / 6;
}
```

Exgcd

```
/**
 * 算法：扩展欧几里得算法
 * 作用：求解 ax + by = gcd(a, b)
 * 返回：gcd, x, y
 */
template<typename T = i64>
array<T, 3> _Exgcd(T a, T b) {
    T x1 = 1, x2 = 0, x3 = 0, x4 = 1;
    while (b != 0) {
        T c = a / b;
        std::tie(x1, x2, x3, x4, a, b)
            = std::make_tuple(x3, x4, x1 - x3 * c, x2 - x4 * c, b, a - b * c);
    }
    return {a, x1, x2}; //x = x1, y = x2;
}

/**
 * 算法：扩展欧几里得算法
 * 作用：求解 ax + by = res
 * 限制：gcd(a, b) | res
 */
template<typename T = i64>
array<T, 3> Exgcd(T a, T b, T res) {
    assert(res % __gcd(a, b) == 0);
    auto [gcd, x, y] = _Exgcd(a, b);
    return {gcd, res / gcd * x, res / gcd * y};
}
```

```cpp
/**
 * 算法：线性同余方程
 * 作用：求解 ax == b (mod P)
 *       的最小整数解
 * 要求：gcd(a, P) | b
 */
template<typename T>
T linearCongruenceEquation(i64 a, i64 b, i64 P) {
    auto [gcd, x, k] = Exgcd<T>(a, P, b);
    T t = P / gcd;
    return (x % t + t) % t;
}

/**
 * 算法：扩展欧几里得算法求逆元
 * 作用：求解 ax == 1 ( mod n )的最小整数解
 * 要求：a 与 n 互质
 */
template<typename T>
T inv(i64 a, i64 P) {
    auto [gcd, x, k] = _Exgcd(a, P);
    return (x % P + P) % P;
}
```

中国剩余定理

```cpp
template<typename T = i64>
array<T, 3> _Exgcd(T a, T b) {
    T x1 = 1, x2 = 0, x3 = 0, x4 = 1;
    while (b != 0) {
        T c = a / b;
        std::tie(x1, x2, x3, x4, a, b)
            = std::make_tuple(x3, x4, x1 - x3 * c, x2 - x4 * c, b, a - b * c);
    }
    return {a, x1, x2}; //x = x1, y = x2;
}

template<typename T>
T inv(i64 a, i64 P) {
    auto [gcd, x, k] = _Exgcd(a, P);
    return (x % P + P) % P;
}

/**
 * 算法：中国剩余定理
 * 作用：求解一元线性同余方程 (x == a (mod P)) 在模n (所有的模积) 的解
 * 限制：所有模互质
 */

template<typename T>
T chineseRemainderTheorem(vector<i64> &a, vector<i64> &P) {
    T n = accumulate(P.begin(), P.end(), (T) 1, multiplies<T>()), ans = 0;

    for (int i = 0; i < (i64) a.size(); ++i) {
        T P1 = n / P[i], b;
```

```
        b = inv<T>(P1, P[i]);
        ans = (ans + a[i] * P1 * b % n) % n;
    }
    return (ans % n + n) % n;
}

template<typename T = i64>
array<T, 3> Exgcd(T a, T b, T res) {
    assert(res % __gcd(a, b) == 0);
    auto [gcd, x, y] = _Exgcd(a, b);
    return {gcd, res / gcd * x, res / gcd * y};
}

/**
 * 算法：扩展中国剩余定理
 * 作用：求解一元线性同余方程（ x == a （ mod m ））在模n（所有模的最小公倍数）的解
 * 无限制：所有模互质
*/
template<typename T>
T extendTheChineseRemainderTheorem(vector<i64> &a, vector<i64> &P) {
    T P1 = P[0], a1 = a[0];
    for (int i = 1; i < a.size(); ++i) {
        T P2 = P[i], a2 = a[i];
        auto [gcd, p, q] = Exgcd(P1, P2, a2 - a1);
        a1 = P1 * p + a1;
        P1 = P1 * P2 / gcd;
        a1 = (a1 % P1 + P1) % P1;
    }
    return a1;
}
```

质因数分解，素数检验

```
i64 mul(i64 a, i64 b, i64 m) {
    return static_cast<__int128>(a) * b % m;
}
i64 power(i64 a, i64 b, i64 m) {
    i64 res = 1 % m;
    for (; b; b >>= 1, a = mul(a, a, m))
        if (b & 1)
            res = mul(res, a, m);
    return res;
}
/**
 * 算法：Miller_Rabin_Test
 * 作用：在long long范围内快速判断质数
 * 时间复杂度：O(log^3(n))
 */
bool isprime(i64 n) {
    if (n < 2)
        return false;
    static constexpr int A[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
    int s = __builtin_ctzll(n - 1);
    i64 d = (n - 1) >> s;
    for (auto a : A) {
```

```
        if (a == n)
            return true;
        i64 x = power(a, d, n);
        if (x == 1 || x == n - 1)
            continue;
        bool ok = false;
        for (int i = 0; i < s - 1; ++i) {
            x = mul(x, x, n);
            if (x == n - 1) {
                ok = true;
                break;
            }
        }
        if (!ok)
            return false;
    }
    return true;
}
/**
 * 时间复杂度 : O(n ^ (1 / 4))
 */
std::vector<i64> factorize(i64 n) {
    std::vector<i64> p;
    std::function<void(i64)> f = [&](i64 n) {
        if (n <= 10000) {
            for (int i = 2; i * i <= n; ++i)
                for (; n % i == 0; n /= i)
                    p.push_back(i);
            if (n > 1)
                p.push_back(n);
            return;
        }
        if (isprime(n)) {
            p.push_back(n);
            return;
        }
        auto g = [&](i64 x) {
            return (mul(x, x, n) + 1) % n;
        };
        i64 x0 = 2;
        while (true) {
            i64 x = x0;
            i64 y = x0;
            i64 d = 1;
            i64 power = 1, lam = 0;
            i64 v = 1;
            while (d == 1) {
                y = g(y);
                ++lam;
                v = mul(v, std::abs(x - y), n);
                if (lam % 127 == 0) {
                    d = std::gcd(v, n);
                    v = 1;
                }
                if (power == lam) {
                    x = y;
```

```
                    power *= 2;
                    lam = 0;
                    d = std::gcd(v, n);
                    v = 1;
                }
            }
            if (d != n) {
                f(d);
                f(n / d);
                return;
            }
            ++x0;
        }
    };
    f(n);
    std::sort(p.begin(), p.end());
    return p;
}

void dfs(int i, auto &b, vector<i64>& ans, i64 c) {
    if (i == b.size()) {
        ans.push_back(c);
        return;
    }
    auto [u, siz] = b[i];
    i64 t = 1;
    for (int j = 0; j <= siz; j += 1) {
        dfs(i + 1, b, ans, c * t);
        t *= u;
    }
}

/**
 * 时间复杂度 : O(siz * P.size())
 */
vector<i64> factorize2(i64 n) {
    auto p = factorize(n);
    std::map<i64, int> map;
    for (auto u : p) {
        map[u] ++;
    }
    auto v = vector(map.begin(), map.end());
    vector<i64> ans;
    dfs(0, v, ans, 1);
    sort(ans.begin(), ans.end());
    return ans;
}
```

扩展欧拉定理

```
template<typename T>
T power(T a, i64 b, i64 P) {
    T res = 1;
    for (; b; b >>= 1) {
        if (b & 1) {
```

```
            res = 1ll * res * a % P;
        }
        a = 1ll * a * a % P;
    }
    return res;
}

/**
 * 算法：欧拉函数
 * 作用：求欧拉函数
 * 时间复杂度：O(sqrt(n))
 */
template<typename T = i64>
T Phi(T n) {
    T ans = n;
    for (i64 i = 2; i * i <= n; i++)
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
}

/**
 * 算法：扩展欧拉定理（欧拉降幂）
 * 作用：大指数快速幂
 * 时间复杂度：O(sqrt(m))
 */

i64 exEulertheorem(i64 a, string b, i64 P) {
    i64 gcd = __gcd(a, P);
    i64 phi = Phi(P);
    i64 res = 0;
    bool ok = 0;
    for (auto u: b) {
        res = res * 10 + u - '0';
        while (res >= phi) {
            res -= phi;
            if (!ok) ok = 1;
        }
    }
    if (gcd != 1 && ok) res += phi;
    return power(a, res, P);
}
```

扩展卢卡斯定理

```
/**
 * 算法：扩展lucas
 * 作用：在p为非质数情况下，大数组合数C(n,m)
 * 必要情况下，预处理降低复杂度，复杂度O(p logp)
*/
using i64 = long long;
i64 mul(i64 a, i64 b, i64 P) {
```

```
    return static_cast<__int128>(a) * b % P;
}
i64 power(i64 a, i64 b, i64 P) {
    i64 res = 1 % P;
    for (; b; b >>= 1, a = mul(a, a, P))
        if (b & 1)
            res = mul(res, a, P);
    return res;
}
template<typename T = i64>
constexpr array<T, 3> Exgcd(T a, T b) {
    T x1 = 1, x2 = 0, x3 = 0, x4 = 1;
    while (b != 0) {
        T c = a / b;
        std::tie(x1, x2, x3, x4, a, b) =
                std::make_tuple(x3, x4, x1 - x3 * c, x2 - x4 * c, b, a - b * c);
    }
    return {a, x1, x2}; //x = x1, y = x2;
}
template<typename T = i64>
constexpr array<T, 3> __Exgcd(T a, T b, T res) {
    assert(res % __gcd(a, b) == 0);
    auto [gcd, x, y] = Exgcd(a, b);
    return {gcd, res / gcd * x, res / gcd * y};
}
template<typename T = i64>
constexpr T inv(i64 a, i64 mod) {
    auto [gcd, x, k] = Exgcd<T>((T) a, (T) mod);
    return (x % mod + mod) % mod;
}
template<typename T = i64>
constexpr T Extend_the_Chinese_remainder_theorem
        (vector <i64> &a, vector <i64> &m) {
    T m1 = m[0], a1 = a[0];
    for (int i = 1; i < (i64) a.size(); ++i) {
        T m2 = m[i], a2 = a[i];
        auto [gcd, p, q] = __Exgcd(m1, m2, a2 - a1);
        a1 = m1 * p + a1;
        m1 = m1 * m2 / gcd;
        a1 = (a1 % m1 + m1) % m1;
    }
    return a1;
}
i64 Exlucas(i64 n, i64 m, i64 P) {
    std::vector <i64> p, a;
    function <i64(i64, i64, i64)> calc = [&](i64 n, i64 x, i64 P) mutable -> i64
{
        if (!n) return 1;
        i64 s = 1;
        for (i64 i = 1; i <= P; ++i)  //求阶乘，可预处理降低复杂度
            if (i % x != 0) s = mul(s, i, P);
        s = power(s, n / P, P);
        for (i64 i = n / P * P + 1; i <= n; ++i)
            if (i % x != 0) s = mul(i, s, P);
        return mul(s, calc(n / x, x, P), P);
    };
```

```
    function <i64(i64, i64, i64, i64)> multilucas = [&](i64 n, i64 m, i64 x, i64
P) -> i64 {
        i64 cnt = 0;
        for (i64 i = n; i != 0; i /= x) cnt += i / x;
        for (i64 i = m; i != 0; i /= x) cnt -= i / x;
        for (i64 i = n - m; i != 0; i /= x) cnt -= i / x;
        return static_cast<__int128>(1) * power(x, cnt, P) % P * calc(n, x, P) %
P
                * inv(calc(m, x, P), P) % P * inv(calc(n - m, x, P), P) % P;
    };
    for (i64 i = 2; i * i <= P; ++i) {
        if (P % i == 0) {
            p.emplace_back(1);
            while (P % i == 0) p.back() *= i, P /= i;
            a.emplace_back(multilucas(n, m, i, p.back()));
        }
    }
    if (P > 1) p.emplace_back(P), a.emplace_back(multilucas(n, m, P, P));
    return Extend_the_Chinese_remainder_theorem(a, p);
}
```

扩展大步小步算法

```
/**
 * 算法：扩展BSGS
 * 作用：求解 a ^ x = b ( mod m )
 * 无要求： a与m互质
 * 返回：问题的最小非负x，无解返回-1
 * 建议使用自定义Hash
*/
using i64 = long long;
using ui64 = unsigned long long;
constexpr i64 exBSGS(i64 a, i64 b, i64 m, i64 k = 1) {
    constexpr i64 inf = 1e15;
    auto BSGS = [&](i64 a, i64 b, i64 m, i64 k = 1) {
# ifdef _Hash
        unordered_map <ui64, ui64, Hash> map;
# else
        std::map <ui64, ui64> map;
# endif
        i64 cur = 1, t = sqrt(m) + 1;
        for (i64 B = 1; B <= t; ++B) {
            (cur *= a) %= m;
            map[b * cur % m] = B;
        }
        ll now = cur * k % m;
        for (i64 A = 1; A <= t; ++A) {
            auto it = map.find(now);
            if (it != map.end())
                return A * t - (i64) it->second;
            (now *= cur) %= m;
        }
        return -inf; // 无解
    };
```

```
    i64 A = a %= m, B = b %= m, M = m;
    if (b == 1) return 0;
    i64 cur = 1 % m;
    for (int i = 0;; i++) {
        if (cur == B) return i;
        cur = cur * A % M;
        i64 d = __gcd(a, m);
        // if (b % d) return -inf;
        if(b % d) return -1;
        if (d == 1) {
            auto ans = BSGS(a, b, m, k * a % m);
            if (ans == -inf) return -1;
            else return ans + i + 1;
        }
        k = k * a / d % m, b /= d, m /= d;
    }
}
```

n次剩余

```
/**
 * 算法：n次剩余
 * 作用：求解 x ^ a = b ( mod m )
 * 要求：m是质数
 * 返回：x，无解返回-1e15
 * 建议使用自定义Hash
*/
using i64 = long long;
i64 mul(i64 a, i64 b, i64 m) {
    return static_cast<__int128>(a) * b % m;
}
template<class T = i64>
constexpr T power(T a, i64 b) {
    T res = 1;
    for (; b; b /= 2, a *= a)
        if (b % 2) res *= a;
    return res;
}

i64 power(i64 a, i64 b, i64 m) {
    i64 res = 1 % m;
    for (; b; b >>= 1, a = mul(a, a, m))
        if (b & 1)
            res = mul(res, a, m);
    return res;
}
std::vector <i64> n_times_remaining(i64 a, i64 b, i64 m) {
    b %= m;
    vector<array<i64, 3>> fs;
    [&] (i64 m) {
        for (i64 i = 2; i * i <= m; i += 1) {
            if (m % i == 0) {
                array<i64, 3> f{i, 1, 0};
                while(m % i == 0) m /= i, f[1] *= i, f[2] += 1;
```

```
                fs.push_back(f);
            }
        }
        if (m > 1) fs.push_back({m, m, 1});
    }(m);
    auto get_Step = [&] (i64 a, i64 n, i64 mod) {//求阶
        i64 ans = n;
        for (i64 i = 2; i * i <= n; i++)
            if (n % i == 0) {
                while (ans % i == 0 && power(a, ans / i, mod) == 1) ans /= i;
                for (; n % i == 0; n /= i);
            }
        if (power(a, ans / n, mod) == 1)ans /= n;
        return ans;
    };

    i64 ans = 1;
    auto cntor = [&] (i64 A, i64 B, i64 m, i64 phi) {
        i64 c = get_Step(B, phi, m), y = phi / c, G = __gcd(A, phi);
        if (y % G) ans = 0; ans *= G;
    };
    for (auto [p, pt, t] : fs) {
        if (!ans) break;
        if (b % pt == 0) ans *= power(p, t - (t + a - 1) / a, 1e9);
        else {
            i64 Z = 0, b0 = b;
            for (; b0 % p == 0; Z ++, pt /= p, t--, b0 /= p);
            if (Z % a) ans = 0;
            else {
                cntor(a, b0, pt, pt - pt / p);
                ans *= power(p, Z - Z / a, 1e9);
            }
        }
    }
    return std::vector<i64>{ans};
}
```

原根

```
template<typename T = i64>
T Phi(T n) {
    T ans = n;
    for (i64 i = 2; i * i <= n; i++)
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
}

template<typename T>
T power(T a, i64 b, i64 P) {
    T res = 1;
```

```
    for (; b; b >>= 1) {
        if (b & 1) {
            res = 1ll * res * a % P;
        }
        a = 1ll * a * a % P;
    }
    return res;
}

i64 min_primitive_root(i64 m) {
    i64 phi = Phi(m);
    auto div = [&](i64 x) {
        vector <i64> f;
        for (i64 i = 2; i * i <= x; ++i) {
            if (x % i != 0) continue;
            f.push_back(i);
            while (x % i == 0) x /= i;
        }
        if (x != 1 && x != phi) f.push_back(x);
        return f;
    };
    auto d = div(phi);
    i64 root = -1;
    auto check = [&](i64 x) {
        for (auto u: d)
            if (power(x, u, m) == 1)
                return false;
        root = x;
        return true;
    };
    for (i64 i = 1;; ++i) {
        if (__gcd(i, m) != 1)
            continue;
        if (check(i)) break;
    }
    return root;
}
```

原根2

```
struct Sieves {
    int n;
    vector<int> Prime, Euler, Morbius, Approximate, Approximate_cnt;
    vector<bool> notprime;
    vector<array<i64, 2>> div;

    Sieves() {};

    Sieves(int _n) { init(_n); };

    void init(int _n) {
        n = _n;
        Prime_work();
    }
```

```
    void Prime_work() {
        notprime.assign(n + 1, 0);
        notprime[0] = 1;
        notprime[1] = 1;
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) {
                Prime.push_back(i);
            }
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                notprime[i * Prime[j]] = 1;

                if (i % Prime[j] == 0) break;
            }
        }
    }

    void Euler_work() {
        Euler.assign(n + 1, 0);
        Euler[1] = 1;
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) Euler[i] = i - 1;
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                i64 now = i * Prime[j];
                if (i % Prime[j] != 0) {
                    Euler[now] = (Prime[j] - 1) * Euler[i];
                } else {
                    Euler[now] = Prime[j] * Euler[i];
                    break;
                }
            }
        }
    }

    void Morbius_work() {
        Morbius.assign(n + 1, 0);
        Morbius[1] = 1;
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) Morbius[i] = -1;
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                i64 now = i * Prime[j];
                if (i % Prime[j] != 0) {
                    Morbius[now] = -Morbius[i];
                } else break;
            }
        }
    }

    void Div_work() {
        div.resize(n + 1);
        div[0] = {1, 1};
        div[1] = {1, 1};
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) {
                div[i] = {1, i};
            }
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
```

```
                div[i * Prime[j]] = {i, Prime[j]};
                if (i % Prime[j] == 0) break;
            }
        }
    }

/**
 * 求约数个数
*/
    void Approximate_work() {
        Approximate.assign(n + 1, 0);
        Approximate_cnt.assign(n + 1, 0);
        Approximate[1] = 1;
        Approximate_cnt[1] = 0;
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) {
                Approximate[i] = 2;
                Approximate_cnt[i] = 1;
            }
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                i64 now = i * Prime[j];
                if (i % Prime[j] != 0) {
                    Approximate_cnt[now] = 1;
                    Approximate[now] = Approximate[i] * 2;
                } else {
                    Approximate_cnt[now] = Approximate_cnt[i] + 1;
                    Approximate[now] = Approximate[i] / Approximate_cnt[now] *
(Approximate_cnt[now] + 1);
                    break;
                }
            }
        }
    }

    std::vector<i64> get_frac(i64 x) {
        vector<i64> f;
        for (; x > 1; f.push_back(div[x][0]), x = div[x][1]);
        return f;
    }

    i64 size() { return (i64) Prime.size(); }

    bool isprime(int n) { return !notprime[n]; }

    i64 eu(int n) { return Euler[n]; }

    i64 mo(int n) { return Morbius[n]; }
};

template<typename T>
T power(T a, i64 b, i64 P) {
    T res = 1;
    for (; b; b >>= 1) {
        if (b & 1) {
            res = 1ll * res * a % P;
        }
```

```
        a = 1ll * a * a % P;
    }
    return res;
}

/**
 * 求一个数的所有原根
 * 时间复杂度：O(sqrt(m))
 */
vector<i64> primitive_root(i64 n) {
    Sieves s(n);
    s.Euler_work();
    vector<bool> exist(n + 1);
    exist[2] = 1;
    exist[4] = 1;
    for (i64 p : s.Prime) {
        if ((p & 1) == 0) continue;
        for (i64 now = p; now < exist.size(); now *= p) {
            exist[now] = 1;
            if (now * 2 < exist.size())
                exist[now * 2] = 1;
        }
    }
    if (!exist[n]) return vector<i64>();
    vector <i64> f;
    i64 phi = s.eu(n);
    i64 pphi = s.eu(phi);
    i64 m = phi;
    for (int i = 2; i * i <= m; ++i) {
        if (m % i == 0) {
            f.push_back(i);
            while (m % i)
                m /= i;
        }
    }
    if (m != 1) f.push_back(m);
    i64 root = -1;
    auto check = [&](i64 x) {
        for (auto u: f)
            if (power(x, phi / u, n) == 1)
                return false;
        root = x;
        return true;
    };
    for (i64 i = 1;; ++i) {
        if (__gcd(i, n) != 1) continue;
        if (check(i)) break;
    }
    vector <i64> ans;
    for (i64 now = root, i = 1; i <= phi; ++i) {
        if (__gcd(phi, i) == 1)
            ans.push_back(now);
        now = (now * root) % n;
    }
    sort(ans.begin(), ans.end());
    return ans;
```

```
}
```

旧版参考

```
/**
 * 数学工具箱
*/

namespace Math {
    using i64 = long long;
    using Int = __int128;
    using ui64 = unsigned long long;
    std::mt19937
rng(std::chrono::system_clock::now().time_since_epoch().count());

    struct math {

/**
 * @brief   带模乘
 * @return  (a ^ b)% m
*/
        i64 static mul(i64 a, i64 b, i64 m);


/**
 * @brief   快速幂
*/
        template<class T>
        constexpr static T power(T a, i64 b);

        i64 static power(i64 a, i64 b, i64 m);

/**
 * @brief   求和
*/
        template<typename T>
        constexpr static T __sum1(T it);

        template<typename T>
        constexpr static T __sum2(T it);

/**
 * 欧几里得算法相关
*/

/**
 * 算法：扩展欧几里得算法
 * 作用：求解 ax + by = gcd ( a , b )
 * 返回：gcd,x,y
 */
        template<typename T = i64>
        constexpr array<T, 3> static Exgcd(T a, T b);
```

```
/**
 * 算法：扩展欧几里得算法
 * 作用：求解 ax + by = res
 * 限制：gcd(a, b) | res
 */
        template<typename T = i64>
        constexpr array<T, 3> static __Exgcd(T a, T b, T res);

/**
 * 算法：线性同余方程
 * 作用：求解 ax == b ( mod n )
 *       的最小整数解
 * 要求：gcd ( a , n ) | b
 */
        template<typename T = i64>
        constexpr T static Linear_congruence_equation(i64 a, i64 b, i64 mod);

/**
 * 算法：扩展欧几里得算法求逆元
 * 作用：求解 ax == 1 ( mod n )的最小整数解
 * 要求：a 与 n 互质
 */
        template<typename T = i64>
        constexpr T static inv(i64 a, i64 mod);

/**
 * 扩展欧几里得结束
*/

/**
 * 算法：Miller_Rabin_Test
 * 作用：在long long范围内快速判断质数
 * 时间复杂度：O(log^3(n))
 */
        constexpr static bool Miller_Rabin_Test(i64 n);

/**
 * 算法：Pollard_Rho
 * 作用：能快速找到大整数的一个非1、非自身的因子的算法
 * 时间复杂度：O(n^{1/4}log(n))
 */
        static i64 Pollard_Rho(i64 N);

/**
 * 算法：使用Pollard_Rho进行质因数分解
 * 返回：顺序所有质因子(重复)
*/
        std::vector <i64> static factorize(i64 n);

/**
 * 算法：中国剩余定理
 * 作用：求解一元线性同余方程（ x == a （ mod m ））在模n（所有的模积）的解
 * 限制：所有模互质
 */
        template<typename T = i64>
        constexpr static T Chinese_remainder_theorem
```

```
                (vector <i64> &a, vector <i64> &m);

/**
 * 算法：扩展中国剩余定理
 * 作用：求解一元线性同余方程（ x == a （ mod m ））在模n（所有模的最小公倍数）的解
 * 无限制：所有模互质
*/
        template<typename T = i64>
        constexpr static T Extend_the_Chinese_remainder_theorem
                (vector <i64> &a, vector <i64> &m);

/**
 * 算法：欧拉函数
 * 作用：求欧拉函数
 * 时间复杂度：O(sqrt ( n ))
 */
        template<typename T = i64>
        constexpr static T Euler_phi(T n);

/**
 * 算法：扩展欧拉定理（欧拉降幂）
 * 作用：大指数快速幂
 * 时间复杂度：O(sqrt ( m ))
 */
        static i64 Extending_Euler_theorem(i64 a, string b, i64 m);


/**
 * 算法：求最小原根
 * 要求：请自行保证这个数有原根(2,4,p^q,2*p^q)
 * 时间复杂度：O(sqrt(n))
*/
        static i64 min_primitive_root(i64 m);

/**
 * 求一个数的所有原根
 * 注意提前使用质数筛，名称为s，开到n，并筛出欧拉函数
 * 需要Linear_sieves_max、s
 * 时间复杂度：O(sqrt ( m ))
 */
# ifdef _Linear_sieves
        std::vector <i64> static primitive_root(i64 n);
# endif


/**
 * 算法：扩展BSGS
 * 作用：求解 a ^ x = b ( mod m )
 * 无要求： a与m互质
 * 返回：问题的最小非负x，无解返回-1
 * 建议使用自定义Hash
*/
        constexpr i64 static exBSGS(i64 a, i64 b, i64 m, i64 k = 1);

/**
 * 算法：n次剩余
```

```
 * 作用：求解 x ^ a = b ( mod m )
 * 要求：m是质数
 * 返回：x，无解返回-1e15
 * 建议使用自定义Hash
*/
        static std::vector <i64> n_times_remaining(i64 a, i64 b, i64 m);


/**
 * 算法：扩展lucas
 * 作用：在p为非质数情况下，大数组合数C(n,m)
 * 必要情况下，预处理降低复杂度
*/

        static i64 Exlucas(i64 n, i64 m, i64 P);

        //struct math
    };


    i64 math::mul(i64 a, i64 b, i64 m) {
        return static_cast<__int128>(a) * b % m;
    }


    template<class T>
    constexpr T math::power(T a, i64 b) {
        T res = 1;
        for (; b; b /= 2, a *= a)
            if (b % 2) res *= a;
        return res;
    }

    i64 math::power(i64 a, i64 b, i64 m) {
        i64 res = 1 % m;
        for (; b; b >>= 1, a = mul(a, a, m))
            if (b & 1)
                res = mul(res, a, m);
        return res;
    }


    template<typename T>
    constexpr T math::__sum1(T it) { return (it * (it + 1)) / ((T) 2); }

    template<typename T>
    constexpr T math::__sum2(T it) { return it * (it + 1) * (2 * it + 1) / ((T)
6); }


    template<typename T>
    constexpr array<T, 3> math::Exgcd(T a, T b) {
        T x1 = 1, x2 = 0, x3 = 0, x4 = 1;
        while (b != 0) {
            T c = a / b;
            std::tie(x1, x2, x3, x4, a, b) =
```

```
                    std::make_tuple(x3, x4, x1 - x3 * c, x2 - x4 * c, b, a - b *
c);
        }
        return {a, x1, x2}; //x = x1, y = x2;
    }

    template<typename T>
    constexpr array<T, 3> math::__Exgcd(T a, T b, T res) {
        assert(res % __gcd(a, b) == 0);
        auto [gcd, x, y] = Exgcd(a, b);
        return {gcd, res / gcd * x, res / gcd * y};
    }

    template<typename T>
    constexpr T math::Linear_congruence_equation(i64 a, i64 b, i64 mod) {
        auto [gcd, x, k] = __Exgcd<T>((T) a, (T) mod, (T) b);
        T t = mod / gcd;
        return (x % t + t) % t;
    }

    template<typename T>
    constexpr T math::inv(i64 a, i64 mod) {
        auto [gcd, x, k] = Exgcd<T>((T) a, (T) mod);
        return (x % mod + mod) % mod;
    }


    constexpr bool math::Miller_Rabin_Test(i64 n) {
        if (n < 3 || n % 2 == 0) return n == 2;//特判
        i64 u = n - 1, t = 0;
        while (u % 2 == 0) u /= 2, ++t;
        constexpr std::array<i64, 7> ud = {2, 325, 9375, 28178, 450775, 9780504,
1795265022};
        for (i64 a: ud) {
            i64 v = power(a, u, n);
            if (v == 1 || v == n - 1 || v == 0) continue;
            for (int j = 1; j <= t; j++) {
                v = mul(v, v, n);
                if (v == n - 1 && j != t) {
                    v = 1;
                    break;
                }//出现一个n-1，后面都是1，直接跳出
                if (v == 1) return 0;//这里代表前面没有出现n-1这个解，二次检验失败
            }
            if (v != 1) return 0;//Fermat检验
        }
        return 1;
    }

    i64 math::Pollard_Rho(i64 N) {
        if (N == 4) // 特判4
            return 2;
        if (Miller_Rabin_Test(N)) // 特判质数
            return N;
        auto randint = [&](i64 l, i64 r) {
            return l + rng() % (r - l + 1);
```

```
        };
        while (true) {
            i64 c = randint(1, N - 1); // 生成随机的c
            auto f = [=](i64 x) { return ((Int) x * x + c) % N; }; // Int表示
__int128，防溢出
            i64 t = f(0), r = f(f(0));
            while (t != r) {
                i64 d = gcd(abs(t - r), N);
                if (d > 1)
                    return d;
                t = f(t), r = f(f(r));
            }
        }
    }

    std::vector <i64> math::factorize(i64 n) {
        std::vector <i64> p;
        std::function<void(i64)> f = [&](i64 n) {
            if (n <= 10000) {
                for (int i = 2; i * i <= n; ++i)
                    for (; n % i == 0; n /= i)
                        p.push_back(i);
                if (n > 1)
                    p.push_back(n);
                return;
            }
            if (Miller_Rabin_Test(n)) {
                p.push_back(n);
                return;
            }
            auto g = [&](i64 x) {
                return (mul(x, x, n) + 1) % n;
            };
            i64 x0 = 2;
            while (true) {
                i64 x = x0;
                i64 y = x0;
                i64 d = 1;
                i64 power = 1, lam = 0;
                i64 v = 1;
                while (d == 1) {
                    y = g(y);
                    ++lam;
                    v = mul(v, std::abs(x - y), n);
                    if (lam % 127 == 0) {
                        d = std::gcd(v, n);
                        v = 1;
                    }
                    if (power == lam) {
                        x = y;
                        power *= 2;
                        lam = 0;
                        d = std::gcd(v, n);
                        v = 1;
                    }
                }
```

```
            if (d != n) {
                f(d);
                f(n / d);
                return;
            }
            ++x0;
        }
    };
    f(n);
    std::sort(p.begin(), p.end());
    return p;
}

template<typename T>
constexpr T math::Chinese_remainder_theorem
        (vector <i64> &a, vector <i64> &m) {
    T n = accumulate(m.begin(), m.end(), (T) 1, multiplies<T>()), ans = 0;

    for (int i = 0; i < (i64) a.size(); ++i) {
        T m1 = n / m[i], b;
        b = inv(m1, m[i]);
        ans = (ans + a[i] * m1 * b % n) % n;
    }
    return (ans % n + n) % n;
}

template<typename T>
constexpr T math::Extend_the_Chinese_remainder_theorem
        (vector <i64> &a, vector <i64> &m) {
    T m1 = m[0], a1 = a[0];
    for (int i = 1; i < (i64) a.size(); ++i) {
        T m2 = m[i], a2 = a[i];
        auto [gcd, p, q] = __Exgcd(m1, m2, a2 - a1);
        a1 = m1 * p + a1;
        m1 = m1 * m2 / gcd;
        a1 = (a1 % m1 + m1) % m1;
    }
    return a1;
}

template<typename T>
constexpr T math::Euler_phi(T n) {
    T ans = n;
    for (i64 i = 2; i * i <= n; i++)
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
}

i64 math::Extending_Euler_theorem(i64 a, string b, i64 m) {
    i64 gcd = __gcd(a, m);
    i64 phi = Euler_phi(m);
    i64 res = 0;
```

```
        bool flag = 0;
        for (auto u: b) {
            res = res * 10 + u - '0';
            while (res >= phi) {
                res -= phi;
                if (!flag) flag = 1;
            }
        }
        if (gcd != 1 && flag) res += phi;
        return power(a, res, m);
    }

    i64 math::min_primitive_root(i64 m) {
        i64 phi = math::Euler_phi(m);
        auto div = [&](i64 x) {
            vector <i64> f;
            for (i64 i = 2; i * i <= x; ++i) {
                if (x % i != 0) continue;
                f.push_back(i);
                while (x % i == 0) x /= i;
            }
            if (x != 1 && x != phi) f.push_back(x);
            return f;
        };
        auto d = div(phi);
        i64 root = -1;
        auto check = [&](i64 x) {
            for (auto u: d)
                if (math::power(x, u, m) == 1)
                    return false;
            root = x;
            return true;
        };
        for (i64 i = 1;; ++i) {
            if (__gcd(i, m) != 1)
                continue;
            if (check(i)) break;
        }
        return root;
    }

# ifdef _Linear_sieves
    std::vector <i64> math::primitive_root(i64 n) {
        static vector<bool> exist(Linear_sieves_max + 1);
        auto __exist = [&]() {
            static bool __existed = 0;
            if (__existed) return;
            __existed = 1;
            exist[2] = 1;
            exist[4] = 1;
            for (ll p: s.Prime) {
                if ((p & 1) == 0) continue;
                for (ll now = p; now <= (ll) exist.size() - 1; now *= p) {
                    exist[now] = 1;
                    if (now * 2 <= (ll) exist.size() - 1)
                        exist[now * 2] = 1;
```

```
                }
            }
        };
        __exist();
        if (!exist[n]) return vector<i64>();
        vector <ll> f;
        ll phi = s.eu(n);
        ll pphi = s.eu(phi);
        ll m = phi;
        for (int i = 2; i * i <= m; ++i) {
            if (m % i == 0) {
                f.push_back(i);
                while (m % i)
                    m /= i;
            }
        }
        if (m != 1) f.push_back(m);
        // Debug ( f ) ;
        ll root = -1;
        auto check = [&](ll x) {
            for (auto u: f)
                if (power(x, phi / u, n) == 1)
                    return false;
            root = x;
            return true;
        };
        for (i64 i = 1;; ++i) {
            if (__gcd(i, n) != 1) continue;
            if (check(i)) break;
        }
        vector <ll> ans;
        for (i64 now = root, i = 1; i <= phi; ++i) {
            if (__gcd(phi, i) == 1)
                ans.push_back(now);
            now = (now * root) % n;
        }
        sort(ans.begin(), ans.end());
        return ans;
    }

# endif

    constexpr i64 math::exBSGS(i64 a, i64 b, i64 m, i64 k) {
        constexpr i64 inf = 1e15;
        auto BSGS = [&](i64 a, i64 b, i64 m, i64 k = 1) {
# ifdef _Hash
            unordered_map <ui64, ui64, Hash> map;
# else
            std::map <ui64, ui64> map;
# endif
            i64 cur = 1, t = sqrt(m) + 1;
            for (i64 B = 1; B <= t; ++B) {
                (cur *= a) %= m;
                map[b * cur % m] = B;
            }
            ll now = cur * k % m;
```

```
            for (i64 A = 1; A <= t; ++A) {
                auto it = map.find(now);
                if (it != map.end())
                    return A * t - (i64) it->second;
                (now *= cur) %= m;
            }
            return -inf; // 无解
        };
        i64 A = a %= m, B = b %= m, M = m;
        if (b == 1) return 0;
        i64 cur = 1 % m;
        for (int i = 0;; i++) {
            if (cur == B) return i;
            cur = cur * A % M;
            i64 d = __gcd(a, m);
            if (b % d) return -inf;
            if (d == 1) {
                auto ans = BSGS(a, b, m, k * a % m);
                if (ans == -inf) return -1;
                else return ans + i + 1;
            }
            k = k * a / d % m, b /= d, m /= d;
        }
    }

    std::vector <i64> math::n_times_remaining(i64 a, i64 b, i64 m) {
        auto root = min_primitive_root(m);
        i64 now = math::power(root, a, m);
        i64 c = math::exBSGS(now, b, m);
        if (c == -1) return vector<i64>();
        i64 x0 = math::power(root, c, m);
        i64 phi = math::Euler_phi(m);
        i64 gcd = __gcd(a, phi);
        vector <i64> ans;
        i64 cnt = math::power(root, phi / gcd, m);
        for (int i = 0; i < gcd; ++i) {
            ans.push_back(x0);
            x0 = math::mul(x0, cnt, m);
        }
        return ans;
    }

    i64 math::Exlucas(i64 n, i64 m, i64 P) {
        std::vector <i64> p, a;
        function <i64(i64, i64, i64)> calc = [&](i64 n, i64 x, i64 P) mutable ->
i64 {
            if (!n) return 1;
            i64 s = 1;
            for (i64 i = 1; i <= P; ++i)  //求阶乘，可预处理降低复杂度
                if (i % x != 0) s = math::mul(s, i, P);
            s = math::power(s, n / P, P);
            for (i64 i = n / P * P + 1; i <= n; ++i)
                if (i % x != 0) s = math::mul(i, s, P);
            return math::mul(s, calc(n / x, x, P), P);
        };
```

```
        function <i64(i64, i64, i64, i64)> multilucas = [&](i64 n, i64 m, i64 x,
i64 P) -> i64 {
            i64 cnt = 0;
            for (i64 i = n; i != 0; i /= x) cnt += i / x;
            for (i64 i = m; i != 0; i /= x) cnt -= i / x;
            for (i64 i = n - m; i != 0; i /= x) cnt -= i / x;
            return static_cast<__int128>(1) * math::power(x, cnt, P) % P *
calc(n, x, P) % P
                    * math::inv(calc(m, x, P), P) % P * math::inv(calc(n - m, x,
P), P) % P;
        };
        for (i64 i = 2; i * i <= P; ++i) {
            if (P % i == 0) {
                p.emplace_back(1);
                while (P % i == 0) p.back() *= i, P /= i;
                a.emplace_back(multilucas(n, m, i, p.back()));
            }
        }
        if (P > 1) p.emplace_back(P), a.emplace_back(multilucas(n, m, P, P));
        return math::Extend_the_Chinese_remainder_theorem(a, p);
    }
    // namespace Math
}

using namespace Math;
```

线性基

```
struct Linear_Base {
    int siz;
    vector<int> a;
    Linear_Base(int _siz = 61) {
        siz = _siz;
        a.resize(siz + 1);
    }
    void insert(int x) {//插入
        for (int i = siz; i >= 0; i--) if (x & (1ll << i)) {
            if (!a[i]) { a[i] = x; return; }
            else x ^= a[i];
        }
    }
    bool check(int x) {//查询x是否能被异或出来
        for (int i = siz; i >= 0; i--) if (x & (1ll << i)) {
            if (!a[i]) break;
            x ^= a[i];
        }
        return x == 0;
    }
    int querymax(int res) {//查询最大异或和
        for (int i = siz; i >= 0; i--) if ((res ^ a[i]) > res) res ^= a[i];
        return res;
    }
    int querymin(int res) {//查询最小
        for (int i = siz; i >= 0; i--) if (res & (1ll << i)) res ^= a[i];
        return res;
```

```
    }
    int querykth(int k) {//查询第k大的异或和
        vector<int> tmp(siz + 10);
        int res = 0, cnt = 0;
        for (int i = 0; i <= siz; i++) {
            for (int j = i - 1; j >= 0; j--) if (a[i] & (1ll << j)) a[i] ^= a[j];
            if(a[i]) tmp[cnt++] = a[i];
        }
        for (int i = 0; i < cnt; i++) if (k & (1ll << i)) res ^= tmp[i];
        return res;
    }
    void merge(const Linear_Base& other)//合并
    {
        for (int i = 0; i <= siz; i++) insert(other.a[i]);
    }
};
```

线性筛

```
struct Linear_sieves {
# define _Linear_sieves
    int n;
    vector<int> Prime, Euler, Morbius, Approximate, Approximate_cnt;
    vector<bool> notprime;
    vector<array<i64, 2>> div;

    Linear_sieves() {};

    Linear_sieves(int _n) { init(_n); };

    void init(int _n) {
        n = _n;
        Prime_work();
    }

    void Prime_work() {
        notprime.assign(n + 1, 0);
        notprime[0] = 1;
        notprime[1] = 1;
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) {
                Prime.push_back(i);
            }
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                notprime[i * Prime[j]] = 1;

                if (i % Prime[j] == 0) break;
            }
        }
    }

    void Euler_work() {
        Euler.assign(n + 1, 0);
        Euler[1] = 1;
```

```
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) Euler[i] = i - 1;
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                i64 now = i * Prime[j];
                if (i % Prime[j] != 0) {
                    Euler[now] = (Prime[j] - 1) * Euler[i];
                } else {
                    Euler[now] = Prime[j] * Euler[i];
                    break;
                }
            }
        }
    }

    void Morbius_work() {
        Morbius.assign(n + 1, 0);
        Morbius[1] = 1;
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) Morbius[i] = -1;
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                i64 now = i * Prime[j];
                if (i % Prime[j] != 0) {
                    Morbius[now] = -Morbius[i];
                } else break;
            }
        }
    }

    void Div_work() {
        div.resize(n + 1);
        div[0] = {1, 1};
        div[1] = {1, 1};
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) {
                div[i] = {1, i};
            }
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                div[i * Prime[j]] = {Prime[j], i};
                if (i % Prime[j] == 0) break;
            }
        }
    }

/**
 * 求约数个数
*/
    void Approximate_work() {
        Approximate.assign(n + 1, 0);
        Approximate_cnt.assign(n + 1, 0);
        Approximate[1] = 1;
        Approximate_cnt[1] = 0;
        for (i64 i = 2; i <= n; ++i) {
            if (notprime[i] == 0) {
                Approximate[i] = 2;
                Approximate_cnt[i] = 1;
            }
```

```
            for (i64 j = 0; i * Prime[j] <= n; ++j) {
                i64 now = i * Prime[j];
                if (i % Prime[j] != 0) {
                    Approximate_cnt[now] = 1;
                    Approximate[now] = Approximate[i] * 2;
                } else {
                    Approximate_cnt[now] = Approximate_cnt[i] + 1;
                    Approximate[now] = Approximate[i] / Approximate_cnt[now] *
(Approximate_cnt[now] + 1);
                    break;
                }
            }
        }
    }

    std::vector<i64> get_frac(i64 x) {
        vector<i64> f;
        for (; x > 1; f.push_back(div[x][0]), x = div[x][1]);
        return f;
    }

    i64 size() { return (i64) Prime.size(); }

    bool isprime(int n) { return !notprime[n]; }

    i64 eu(int n) { return Euler[n]; }

    i64 mo(int n) { return Morbius[n]; }
};
```

线性筛轻量级

```
vector<int> minp, primes;

void seive(int n) {
    minp.assign(n + 1, 0);
    primes.clear();

    for (int i = 2; i <= n; i += 1) {
        if (minp[i] == 0) {
            minp[i] = i;
            primes.push_back(i);
        }

        for (auto p : primes) {
            if (i * p > n) {
                break;
            }
            minp[i * p] = p;
            if (p == minp[i]) {
                break;
            }
        }
    }
```

```
}
```

组合数学

```
template<class T>
struct Comb {
    int n;
    std::vector <T> _fac;
    std::vector <T> _invfac;
    std::vector <T> _inv;

    Comb() : n{0}, _fac{1}, _invfac{1}, _inv{0} {}

    Comb(int n) : Comb() {
        init(n);
    }

    void init(int m) {
        m = std::min(m, T::getMod() - 1);
        if (m <= n) return;
        _fac.resize(m + 1);
        _invfac.resize(m + 1);
        _inv.resize(m + 1);

        for (int i = n + 1; i <= m; i++) {
            _fac[i] = _fac[i - 1] * i;
        }
        _invfac[m] = _fac[m].inv();
        for (int i = m; i > n; i--) {
            _invfac[i - 1] = _invfac[i] * i;
            _inv[i] = _invfac[i] * _fac[i - 1];
        }
        n = m;
    }

    T fac(int m) {
        if (m > n) init(2 * m);
        return _fac[m];
    }

    T invfac(int m) {
        if (m > n) init(2 * m);
        return _invfac[m];
    }

    T inv(int m) {
        if (m > n) init(2 * m);
        return _inv[m];
    }

    T binom(int n, int m) {
        if (n < m || m < 0) return 0;
        return fac(n) * invfac(m) * invfac(n - m);
```

```
    }

/**
 * 第二类斯特林数
 * 时间复杂度 : O (m * log (m))
*/
    T Stirling(int n, int m) {
        T ans = 0;
        for (int i = 0; i <= m; ++i) {
            ans += (((m - i) & 1) == 1 ? -1 : 1) * power((T) i, n) * invfac(i) *
invfac(m - i);
        }
        return ans;
    }

    T Catalan(int n) {
        return binom(2 * n, n) * inv(n + 1);
    }

/**
 * 算法：卢卡斯定理
 * 作用：大数组合数
 * 注意在p较小时使用p
 * p为Z的质数
 * 时间复杂度为O(logp)
*/
    T lucas(i64 n, i64 m) {
        if (m == 0) return T(1);
        return binom(n % T::getMod(), m % T::getMod()) * lucas(n / T::getMod(), m
/ T::getMod());
    }
};

Comb<Z> comb;
```

行列式

```
using i64 = long long;
// 时间复杂度 O(n^3 + n^2 * logp)
// 行列式 mod p， a[1, n][1, n]
constexpr int calcDet(vector<vector<int>> &a, int n, const int p) {
    i64 zf = 1, ans = 1, tmp = 0;

    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= n; ++j)
            a[i][j] %= p;

    for (int i = 1; i <= n; i++) {
        int k = i;
        for (int j = i + 1; j <= n; j++)
            if (a[j][i] > a[k][i]) {
                k = j;
            }
        if (!a[k][i]) return 0;
```

```
        if (k != i) swap(a[i], a[k]), zf = -zf;

        for (int j = i + 1; j <= n; j++) {
            if (a[j][i] > a[i][i]) swap(a[i], a[j]), zf = -zf;
            while (a[j][i]) {
                tmp = a[i][i] / a[j][i];
                for (int k = i; k <= n; k++)
                    a[i][k] = (a[i][k] + a[j][k] * (p - tmp) % p) % p;
                swap(a[i], a[j]), zf = -zf;
            }
        }
        ans = ans * a[i][i] % p;
    }

    if (zf == -1) ans = (-ans + p) % p;
    return ans;
}

// 时间复杂度 O(n^3)
// 行列式 a[0, n)[0, n)
constexpr double calcDet(vector<vector<double>> &a, int n, const double eps = 1e-
9) {
    double det = 1;
    for (int i = 0; i < n; ++i) {
        int k = i;
        for (int j = i + 1; j < n; ++j)
            if (abs(a[j][i]) > abs(a[k][i])) k = j;
        if (abs(a[k][i]) < eps) {
            det = 0;
            break;
        }
        swap(a[i], a[k]);
        if (i != k) det = -det;
        det *= a[i][i];
        for (int j = i + 1; j < n; ++j) a[i][j] /= a[i][i];
        for (int j = 0; j < n; ++j)
            if (j != i && abs(a[j][i]) > eps)
                for (int k = i + 1; k < n; ++k) a[j][k] -= a[i][k] * a[j][i];
    }
    return det;
}
```

高斯消元

```
using f64 = double;

// 时间复杂度: O(m * n^2), -1无解, 0唯一解, 否则无穷解
// a为增广矩阵 行r:[0, n) 列c:[0, m), a[i][m]为b[0, n), 求解答案为 x[0, m)
int Gauss(vector<vector<f64>> &a, vector<f64> &x, int n, int m, f64 eps = 1e-7){
    int r = 0, c = 0;
    for(r = 0; r < n && c < m; r++, c++) {
        int maxr = r;
        for(int i = r + 1; i < n; i++) {
            if(abs(a[i][c]) > abs(a[maxr][c]))
```

```
                maxr = i;
        }
        if(maxr != r) std::swap(a[r], a[maxr]);
        if(fabs(a[r][c]) < eps) {
            r--;
            continue;
        }
        for(int i = r + 1; i < n; i++) {
            if(fabs(a[i][c]) > eps){
                f64 k = a[i][c] / a[r][c];
                for(int j = c; j < m + 1; j++) a[i][j] -= a[r][j] * k;
                a[i][c] = 0;
            }
        }
    }
    for(int i = r; i < m; i++) {
        if(fabs(a[i][c]) > eps) return -1;//无解
    }
    if(r < m) return m - r;//返回自由元个数
    for(int i = m-1; i >= 0; i--) {
        for(int j = i + 1; j < m; j++) a[i][m] -= a[i][j] * x[j];
        x[i] = a[i][m] / a[i][i];
    }
    return 0;//有唯一解
}

struct Complex{
    f64 x, y;
    Complex operator+(const Complex &b) const {
        return Complex({x + b.x, y + b.y});
    }
    Complex operator-(const Complex &b) const {
        return Complex({x - b.x, y - b.y});
    }
    Complex operator*(const Complex &b) const {
        return Complex({x * b.x - y * b.y, x * b.y + y * b.x});
    }
    Complex operator/(const Complex &b) const {
        return Complex({(x * b.x + y * b.y) / (b.x * b.x + b.y * b.y), (y * b.x -
x * b.y) / (b.x * b.x + b.y * b.y)});
    }
    f64 mo() {
        return x * x + y * y;
    }
    bool iszero(f64 eps = 1e-7) {
        return mo() < eps;
    }
};
// 时间复杂度：O(m * n^2), -1无解，0唯一解，否则无穷解
// a为增广矩阵 行r:[0, n) 列c:[0, m)，a[i][m]为b[0, n)，求解答案为 x[0, m)
int Gauss(vector<vector<Complex>> &a, vector<Complex> &x, int n, int m, f64 eps =
1e-7){
    int r = 0, c = 0;
    for(r = 0; r < n && c < m; r++, c++) {
        int maxr = r;
        for(int i = r + 1; i < n; i++) {
```

```
            if(a[i][c].mo() > a[maxr][c].mo())
                maxr = i;
        }
        if(maxr != r) std::swap(a[r], a[maxr]);
        if(a[r][c].iszero(eps)) {
            r--;
            continue;
        }
        for(int i = r + 1; i < n; i++) {
            if(!a[i][c].iszero(eps)) {
                Complex k = a[i][c] / a[r][c];
                for(int j = c; j < m + 1; ++j) a[i][j] = a[i][j] - a[r][j] * k;
                a[i][c] = {0, 0};
            }
        }
    }
    for(int i = r; i < m; i++) {
        if(!a[i][c].iszero(eps)) return -1;
    }
    if(r < m) return m - r;//返回自由元个数
    for(int i = m-1; i >= 0; i--) {
        for(int j = i + 1; j < m; j++) a[i][m] = a[i][m] - a[i][j] * x[j];
        x[i] = a[i][m] / a[i][i];
    }
    return 0;//有唯一解
}
```