OI Templates

v1.2.0

DarkChemist

2023-12-7

Contents

1 C++ Syntax

1.1 Operator Overloading

Operator overloading is a powerful method that allows users to overload operators to perform different operatorations. We can use this to overload input and outupt of `std::vector` and `std::pair`.

```
// pair IO
template <class T, class V>
istream& operator>>(istream& is, pair<T, V>& obj) {
    is >> obj.first >> obj.second;
    return is;
}

template <class T, class V>
ostream& operator<<(ostream& os, const pair<T, V>& obj) {
    os << obj.first << ' ' << obj.second;
    return os;
}

// vector IO
template <class T>
istream& operator>>(istream& is, vector<T>& obj) {
    for (int i = 0; i < int(obj.size()); i++) is >> obj[i];
    return is;
}

template <class T>
ostream& operator<<(ostream& os, const vector<T>& obj) {
    for (int i = 0; i < int(obj.size()); i++) {
        os << obj[i]; if (i != int(obj.size()) - 1) os << ' ';
    }
    return os;
}
```

1.2 String Functions

To construct a string with a specific character of length `n`:

```
string str(n, chr);
```

To obtain a substring `t` from index `i` to `j` inclusively:

```
string t = s.substr(i, j - i + 1);
```

To find if a character exists in a string `str`:

```
if (str.find(chr) != string::npos) {
    cout << "Yes\n";
} else {
    cout << "No\n";
}
```

To access front element / back element of a vector:

```
vec.front();
vec.back();
```

To insert an element `x` in position `idx` into a vector:

```
vec.insert(vec.begin() + idx, x);
```

1.3 Regular Expression

1.4 Random Generator

```
mt19937_64 rng(chrono::system_clock::now().time_since_epoch().count());
```

1.5 Clock

```
clock_t begin = clock();
double end = (double) (clock() - begin) / CLOCKS_PER_SEC;
```

2 Mathematics

2.1 Basic Modular Arithmetic

Modular arithmetic is very important especially when dealing with counting problems. Modulear arithmetic is essential to solve them.
Here are some common identities.

- If $a \equiv b \pmod{n}$, then $b \equiv a \pmod{n}$
- If $a \equiv b \pmod{n}$, and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$
- $(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$
- $(a - b) \bmod m = (a \bmod m - b \bmod m) \bmod m$
- $(a \times b) \bmod m = (a \bmod m \times b \bmod m) \bmod m$

2.2 Binary Exponentiation

Using this method, we can calculate $a^b \bmod m$ in $O(\log n)$

Iterative method:

```
long long bigmd(long long x, long long y, long long z) {
    long long ans = 1;
    while (y){
        if (y & 1) ans = ans * x % z;
        y >>= 1;
        x = x * x % z;
    }
    return ans;
}
```

Recursive method:

```
long long bigmd(int n, int p, int mod){
    if (p == 0) return 1;
    long long tt = bigmd(n, p / 2, mod);
    return (p & 1 ? (n % mod) : 1) * (tt * tt % mod);
}
```

2.3 Modular Inverse

Fermat's little theorem states that

$$a^p \equiv a \pmod{p}$$

where a is an integer and p is a prime number.
The formula can be also represented as

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

Normally, we call a number b as an inverse of a when $a \times b = 1$.
e.g. 0.5 is the inverse of 2.

Similarly, we call a number b as the modular inverse of a modulo n when $a \times b \equiv 1 \pmod{n}$
e.g. 4 is the inverse of 2 modulo 7. $(4 \times 2 \bmod 7 = 1)$

If we recall the above formula, we can come into a conclusion that the modular inverse of $a \bmod p$ is $a^{p-2} \bmod p$ (for prime number p only).

Code example:

```
// use the above bigmd function
long long modinv(long long a, long long p){
    return bigmd(a, p - 2, p);
}
```

2.4 Matrix

Matrices can be useful in solving linear recurrences.

```
struct matrix {
    int row_size, col_size;
    vector<vector<long long>> container;
    matrix(int n, int m) : row_size(n), col_size(m), container(n, vector<long long>(m))
        {};
    matrix(vector<vector<long long>> mtx) : row_size(mtx.size()), col_size(mtx.size() ==
        0 ? 0 : mtx[0].size()), container(mtx) {};

    matrix& operator *= (const matrix& rhs) {
        if (col_size != rhs.row_size) {
            cerr << "This two matrices cannot be multiplied together.\n";
            exit(EXIT_FAILURE);
        }

        vector<vector<long long>> new_container(row_size, vector<long long>(rhs.col_size
            ));
        int common_side = col_size;
        for (int i = 0; i < row_size; i++) {
            for (int j = 0; j < rhs.col_size; j++) {
                long long result = 0;
                for (int k = 0; k < common_side; k++) {
                    result += container[i][k] * rhs.container[k][j];
                }
                new_container[i][j] = result;
            }
        }

        container = new_container;
        return *this;
    }

    friend matrix operator * (matrix lhs, const matrix& rhs) {
        lhs *= rhs;
        return lhs;
    }

    matrix& operator += (const matrix& rhs) {
        if (col_size != rhs.col_size || row_size != rhs.row_size) {
            cerr << "This two matrices cannot be added together.\n";
            exit(EXIT_FAILURE);
        }

        for (int i = 0; i < row_size; i++) {
            for (int j = 0; j < col_size; i++) {
                container[i][j] += rhs.container[i][j];
            }
        }

        return *this;
    }

    matrix operator + (matrix lhs, const matrix& rhs) {
        return lhs + rhs;
    }

    matrix& operator %= (const int mod) {
```

```
        for (int i = 0; i < row_size; i++) {
            for (int j = 0; j < col_size; j++) {
                container[i][j] %= mod;
            }
        }
        return *this;
    }

    friend matrix operator % (matrix lhs, const int mod) {
        lhs %= mod;
        return lhs;
    }
};

matrix power(matrix n, long long p, long long mod) {
    matrix curr = n;
    matrix result(vector<vector<long long>>({{1, 0}, {0, 1}}));
    while (p > 0) {
        if (p & 1) {
            result *= curr;
            result %= mod;
        }
        p >>= 1;
        curr *= curr;
        curr %= mod;
    }
    return result;
}
```

2.5 Extended Euclidean Algorithm

Find a pair of integers (x, y) such that $a \cdot x + b \cdot y = \gcd(a, b)$.

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

2.6 Newton's Method

Note that $x_{i+1} = x_i - \frac{f(x_i)}{f\prime(x_i)}$.

```
double sqrt_newton(double n) {
    const double eps = 1E-15;
    double x = 1;
    for (;;) {
        double nx = x - f(x) / f_prime(x);
        if (abs(x - nx) < eps)
            break;
        x = nx;
    }
    return x;
```

```
}
```

2.7 Polynomial Operations

Here, you can find polynomial operations.

```
/* Verified on https://judge.yosupo.jp:
- N = 500'000:
-- Convolution, 440ms (https://judge.yosupo.jp/submission/85695)
-- Convolution (mod 1e9+7), 430ms (https://judge.yosupo.jp/submission/85696)
-- Inv of power series, 713ms (https://judge.yosupo.jp/submission/85694)
-- Exp of power series, 2157ms (https://judge.yosupo.jp/submission/85698)
-- Log of power series, 1181ms (https://judge.yosupo.jp/submission/85699)
-- Pow of power series, 3275ms (https://judge.yosupo.jp/submission/85703)
-- Sqrt of power series, 1919ms (https://judge.yosupo.jp/submission/85705)
-- P(x) -> P(x+a), 523ms (https://judge.yosupo.jp/submission/85706)
-- Division of polynomials, 996ms (https://judge.yosupo.jp/submission/85707)
- N = 100'000:
-- Multipoint evaluation, 2161ms (https://judge.yosupo.jp/submission/85709)
-- Polynomial interpolation, 2551ms (https://judge.yosupo.jp/submission/85711)
-- Kth term of Linear Recurrence, 2913ms (https://judge.yosupo.jp/submission/85727)
- N = 50'000:
-- Inv of Polynomials, 1691ms (https://judge.yosupo.jp/submission/85713)
- N = 10'000:
-- Find Linear Recurrence, 346ms (https://judge.yosupo.jp/submission/85025)
/////////

The main goal of this library is to implement common polynomial functionality in a
reasonable from competitive programming POV complexity, while also doing it in as
straight-forward way as possible.

Therefore, primary purpose of the library is educational and most of constant-time
optimizations that may significantly harm the code readability were not used.

The library is reasonably fast and generally can be used in most problems where
polynomial operations constitute intended solution. However, it is recommended to
seek out other implementations when the time limit is tight or you really want to
squeeze a solution when it is probably not the intended one.
*/

#include <bits/stdc++.h>

using namespace std;

namespace algebra {
    const int maxn = 1 << 20;
    const int magic = 250; // threshold for sizes to run the naive algo
    mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

    template<typename T>
    T bpow(T x, int64_t n) {
        if(n == 0) {
            return T(1);
        } else {
            auto t = bpow(x, n / 2);
            t = t * t;
            return n % 2 ? x * t : t;
        }
    }
```

```
template<int m>
struct modular {
    // https://en.wikipedia.org/wiki/Berlekamp-Rabin_algorithm
    // solves x^2 = y (mod m) assuming m is prime in O(log m).
    // returns nullopt if no sol.
    optional<modular> sqrt() const {
        static modular y;
        y = *this;
        if(r == 0) {
            return 0;
        } else if(bpow(y, (m - 1) / 2) != modular(1)) {
            return nullopt;
        } else {
            while(true) {
                modular z = rng();
                if(z * z == *this) {
                    return z;
                }
                struct lin {
                    modular a, b;
                    lin(modular a, modular b): a(a), b(b) {}
                    lin(modular a): a(a), b(0) {}
                    lin operator * (const lin& t) {
                        return {
                            a * t.a + b * t.b * y,
                            a * t.b + b * t.a
                        };
                    }
                } x(z, 1); // z + x
                x = bpow(x, (m - 1) / 2);
                if(x.b != modular(0)) {
                    return x.b.inv();
                }
            }
        }
    }

    int r;
    constexpr modular(): r(0) {}
    constexpr modular(int64_t rr): r(rr % m) {if(r < 0) r += m;}
    modular inv() const {return bpow(*this, m - 2);}
    modular operator - () const {return r ? m - r : 0;}
    modular operator * (const modular &t) const {return (int64_t)r * t.r % m;}
    modular operator / (const modular &t) const {return *this * t.inv();}
    modular operator += (const modular &t) {r += t.r; if(r >= m) r -= m; return *
        this;}
    modular operator -= (const modular &t) {r -= t.r; if(r < 0) r += m; return *this
        ;}
    modular operator + (const modular &t) const {return modular(*this) += t;}
    modular operator - (const modular &t) const {return modular(*this) -= t;}
    modular operator *= (const modular &t) {return *this = *this * t;}
    modular operator /= (const modular &t) {return *this = *this / t;}

    bool operator == (const modular &t) const {return r == t.r;}
    bool operator != (const modular &t) const {return r != t.r;}

    explicit operator int() const {return r;}
    int64_t rem() const {return 2 * r > m ? r - m : r;}
```

```
};

template<int T>
istream& operator >> (istream &in, modular<T> &x) {
    return in >> x.r;
}

template<int T>
ostream& operator << (ostream &out, modular<T> const& x) {
    return out << x.r;
}

template<typename T>
T fact(int n) {
    static T F[maxn];
    static bool init = false;
    if(!init) {
        F[0] = T(1);
        for(int i = 1; i < maxn; i++) {
            F[i] = F[i - 1] * T(i);
        }
        init = true;
    }
    return F[n];
}

template<typename T>
T rfact(int n) {
    static T F[maxn];
    static bool init = false;
    if(!init) {
        F[maxn - 1] = T(1) / fact<T>(maxn - 1);
        for(int i = maxn - 2; i >= 0; i--) {
            F[i] = F[i + 1] * T(i + 1);
        }
        init = true;
    }
    return F[n];
}

template<typename T>
T small_inv(int n) {
    static T F[maxn];
    static bool init = false;
    if(!init) {
        for(int i = 1; i < maxn; i++) {
            F[i] = rfact<T>(i) * fact<T>(i - 1);
        }
        init = true;
    }
    return F[n];
}

namespace fft {
    using ftype = double;
    struct point {
        ftype x, y;

        ftype real() {return x;}
```

```
    ftype imag() {return y;}

    point(): x(0), y(0){}
    point(ftype x, ftype y = 0): x(x), y(y){}

    static point polar(ftype rho, ftype ang) {
        return point{rho * cos(ang), rho * sin(ang)};
    }

    point conj() const {
        return {x, -y};
    }

    point operator +=(const point &t) {x += t.x, y += t.y; return *this;}
    point operator +(const point &t) const {return point(*this) += t;}
    point operator -(const point &t) const {return {x - t.x, y - t.y};}
    point operator *(const point &t) const {return {x * t.x - y * t.y, x * t.y +
        y * t.x};}
};

point w[maxn]; // w[2^n + k] = exp(pi * k / (2^n))
int bitr[maxn];// b[2^n + k] = bitreverse(k)
const ftype pi = acos(-1);
bool initiated = 0;
void init() {
    if(!initiated) {
        for(int i = 1; i < maxn; i *= 2) {
            int ti = i / 2;
            for(int j = 0; j < i; j++) {
                w[i + j] = point::polar(ftype(1), pi * j / i);
                if(ti) {
                    bitr[i + j] = 2 * bitr[ti + j % ti] + (j >= ti);
                }
            }
        }
        initiated = 1;
    }
}

void fft(auto &a, int n) {
    init();
    if(n == 1) {
        return;
    }
    int hn = n / 2;
    for(int i = 0; i < n; i++) {
        int ti = 2 * bitr[hn + i % hn] + (i > hn);
        if(i < ti) {
            swap(a[i], a[ti]);
        }
    }
    for(int i = 1; i < n; i *= 2) {
        for(int j = 0; j < n; j += 2 * i) {
            for(int k = j; k < j + i; k++) {
                point t = a[k + i] * w[i + k - j];
                a[k + i] = a[k] - t;
                a[k] += t;
            }
        }
```

```
    }
}

void mul_slow(vector<auto> &a, const vector<auto> &b) {
    if(a.empty() || b.empty()) {
        a.clear();
    } else {
        int n = a.size();
        int m = b.size();
        a.resize(n + m - 1);
        for(int k = n + m - 2; k >= 0; k--) {
            a[k] *= b[0];
            for(int j = max(k - n + 1, 1); j < min(k + 1, m); j++) {
                a[k] += a[k - j] * b[j];
            }
        }
    }
}

template<int m>
struct dft {
    static constexpr int split = 1 << 15;
    vector<point> A;

    dft(vector<modular<m>> const& a, size_t n): A(n) {
        for(size_t i = 0; i < min(n, a.size()); i++) {
            A[i] = point(
                a[i].rem() % split,
                a[i].rem() / split
            );
        }
        if(n) {
            fft(A, n);
        }
    }

    auto operator * (dft const& B) {
        assert(A.size() == B.A.size());
        size_t n = A.size();
        if(!n) {
            return vector<modular<m>>();
        }
        vector<point> C(n), D(n);
        for(size_t i = 0; i < n; i++) {
            C[i] = A[i] * (B[i] + B[(n - i) % n].conj());
            D[i] = A[i] * (B[i] - B[(n - i) % n].conj());
        }
        fft(C, n);
        fft(D, n);
        reverse(begin(C) + 1, end(C));
        reverse(begin(D) + 1, end(D));
        int t = 2 * n;
        vector<modular<m>> res(n);
        for(size_t i = 0; i < n; i++) {
            modular<m> A0 = llround(C[i].real() / t);
            modular<m> A1 = llround(C[i].imag() / t + D[i].imag() / t);
            modular<m> A2 = llround(D[i].real() / t);
            res[i] = A0 + A1 * split - A2 * split * split;
        }
```

```
            return res;
        }

        point& operator [](int i) {return A[i];}
        point operator [](int i) const {return A[i];}
    };

    size_t com_size(size_t as, size_t bs) {
        if(!as || !bs) {
            return 0;
        }
        size_t n = as + bs - 1;
        while(__builtin_popcount(n) != 1) {
            n++;
        }
        return n;
    }

    template<int m>
    void mul(vector<modular<m>> &a, vector<modular<m>> b) {
        if(min(a.size(), b.size()) < magic) {
            mul_slow(a, b);
            return;
        }
        auto n = com_size(a.size(), b.size());
        auto A = dft<m>(a, n);
        if(a == b) {
            a = A * A;
        } else {
            a = A * dft<m>(b, n);
        }
    }
}

template<typename T>
struct poly {
    vector<T> a;

    void normalize() { // get rid of leading zeroes
        while(!a.empty() && a.back() == T(0)) {
            a.pop_back();
        }
    }

    poly(){}
    poly(T a0) : a{a0}{normalize();}
    poly(const vector<T> &t) : a(t){normalize();}

    poly operator -() const {
        auto t = *this;
        for(auto &it: t.a) {
            it = -it;
        }
        return t;
    }

    poly operator += (const poly &t) {
        a.resize(max(a.size(), t.a.size()));
        for(size_t i = 0; i < t.a.size(); i++) {
```

```
        a[i] += t.a[i];
    }
    normalize();
    return *this;
}

poly operator -= (const poly &t) {
    a.resize(max(a.size(), t.a.size()));
    for(size_t i = 0; i < t.a.size(); i++) {
        a[i] -= t.a[i];
    }
    normalize();
    return *this;
}
poly operator + (const poly &t) const {return poly(*this) += t;}
poly operator - (const poly &t) const {return poly(*this) -= t;}

poly mod_xk(size_t k) const { // get first k coefficients
    return vector<T>(begin(a), begin(a) + min(k, a.size()));
}

poly mul_xk(size_t k) const { // multiply by x^k
    auto res = a;
    res.insert(begin(res), k, 0);
    return res;
}

poly div_xk(size_t k) const { // drop first k coefficients
    return vector<T>(begin(a) + min(k, a.size()), end(a));
}

poly substr(size_t l, size_t r) const { // return mod_xk(r).div_xk(l)
    return vector<T>(
        begin(a) + min(l, a.size()),
        begin(a) + min(r, a.size())
    );
}

poly operator *= (const poly &t) {fft::mul(a, t.a); normalize(); return *this;}
poly operator * (const poly &t) const {return poly(*this) *= t;}

poly reverse(size_t n) const { // computes x^n A(x^{-1})
    auto res = a;
    res.resize(max(n, res.size()));
    return vector<T>(res.rbegin(), res.rbegin() + n);
}

poly reverse() const {
    return reverse(deg() + 1);
}

pair<poly, poly> divmod_slow(const poly &b) const { // when divisor or quotient
   is small
    vector<T> A(a);
    vector<T> res;
    T b_lead_inv = b.a.back().inv();
    while(A.size() >= b.a.size()) {
        res.push_back(A.back() * b_lead_inv);
        if(res.back() != T(0)) {
```

```
            for(size_t i = 0; i < b.a.size(); i++) {
                A[A.size() - i - 1] -= res.back() * b.a[b.a.size() - i - 1];
            }
        }
        A.pop_back();
    }
    std::reverse(begin(res), end(res));
    return {res, A};
}

pair<poly, poly> divmod_hint(poly const& b, poly const& binv) const { // when
    inverse is known
    assert(!b.is_zero());
    if(deg() < b.deg()) {
        return {poly{0}, *this};
    }
    int d = deg() - b.deg();
    if(min(d, b.deg()) < magic) {
        return divmod_slow(b);
    }
    poly D = (reverse().mod_xk(d + 1) * binv.mod_xk(d + 1)).mod_xk(d + 1).
        reverse(d + 1);
    return {D, *this - D * b};
}

pair<poly, poly> divmod(const poly &b) const { // returns quotiend and remainder
    of a mod b
    assert(!b.is_zero());
    if(deg() < b.deg()) {
        return {poly{0}, *this};
    }
    int d = deg() - b.deg();
    if(min(d, b.deg()) < magic) {
        return divmod_slow(b);
    }
    poly D = (reverse().mod_xk(d + 1) * b.reverse().inv(d + 1)).mod_xk(d + 1).
        reverse(d + 1);
    return {D, *this - D * b};
}

// (ax+b) / (cx+d)
struct transform {
    poly a, b, c, d;
    transform(poly a, poly b = T(1), poly c = T(1), poly d = T(0)): a(a), b(b),
        c(c), d(d){}

    transform operator *(transform const& t) {
        return {
            a*t.a + b*t.c, a*t.b + b*t.d,
            c*t.a + d*t.c, c*t.b + d*t.d
        };
    }

    transform adj() {
        return transform(d, -b, -c, a);
    }

    auto apply(poly A, poly B) {
        return make_pair(a * A + b * B, c * A + d * B);
```

```
    }
};

template<typename Q>
static void concat(vector<Q> &a, vector<Q> const& b) {
    for(auto it: b) {
        a.push_back(it);
    }
}

// finds a transform that changes A/B to A'/B' such that
// deg B' is at least 2 times less than deg A
static pair<vector<poly>, transform> half_gcd(poly A, poly B) {
    assert(A.deg() >= B.deg());
    int m = (A.deg() + 1) / 2;
    if(B.deg() < m) {
        return {{}, {T(1), T(0), T(0), T(1)}};
    }
    auto [ar, Tr] = half_gcd(A.div_xk(m), B.div_xk(m));
    tie(A, B) = Tr.adj().apply(A, B);
    if(B.deg() < m) {
        return {ar, Tr};
    }
    auto [ai, R] = A.divmod(B);
    tie(A, B) = make_pair(B, R);
    int k = 2 * m - B.deg();
    auto [as, Ts] = half_gcd(A.div_xk(k), B.div_xk(k));
    concat(ar, {ai});
    concat(ar, as);
    return {ar, Tr * transform(ai) * Ts};
}

// return a transform that reduces A / B to gcd(A, B) / 0
static pair<vector<poly>, transform> full_gcd(poly A, poly B) {
    vector<poly> ak;
    vector<transform> trs;
    while(!B.is_zero()) {
        if(2 * B.deg() > A.deg()) {
            auto [a, Tr] = half_gcd(A, B);
            concat(ak, a);
            trs.push_back(Tr);
            tie(A, B) = trs.back().adj().apply(A, B);
        } else {
            auto [a, R] = A.divmod(B);
            ak.push_back(a);
            trs.emplace_back(a);
            tie(A, B) = make_pair(B, R);
        }
    }
    trs.emplace_back(T(1), T(0), T(0), T(1));
    while(trs.size() >= 2) {
        trs[trs.size() - 2] = trs[trs.size() - 2] * trs[trs.size() - 1];
        trs.pop_back();
    }
    return {ak, trs.back()};
}

static poly gcd(poly A, poly B) {
    if(A.deg() < B.deg()) {
```

```
        return full_gcd(B, A);
    }
    auto Tr = fraction(A, B);
    return Tr.d * A - Tr.b * B;
}

// Returns the characteristic polynomial
// of the minimum linear recurrence for the sequence
poly min_rec_slow(int d) const {
    auto R1 = mod_xk(d + 1).reverse(d + 1), R2 = xk(d + 1);
    auto Q1 = poly(T(1)), Q2 = poly(T(0));
    while(!R2.is_zero()) {
        auto [a, nR] = R1.divmod(R2); // R1 = a*R2 + nR, deg nR < deg R2
        tie(R1, R2) = make_tuple(R2, nR);
        tie(Q1, Q2) = make_tuple(Q2, Q1 + a * Q2);
        if(R2.deg() < Q2.deg()) {
            return Q2 / Q2.lead();
        }
    }
    assert(0);
}

static transform convergent(auto L, auto R) { // computes product on [L, R)
    if(R - L == 1) {
        return transform(*L);
    } else {
        int s = 0;
        for(int i = 0; i < R - L; i++) {
            s += L[i].a.size();
        }
        int c = 0;
        for(int i = 0; i < R - L; i++) {
            c += L[i].a.size();
            if(2 * c > s) {
                return convergent(L, L + i) * convergent(L + i, R);
            }
        }
        assert(0);
    }
}

poly min_rec(int d) const {
    if(d < magic) {
        return min_rec_slow(d);
    }
    auto R2 = mod_xk(d + 1).reverse(d + 1), R1 = xk(d + 1);
    if(R2.is_zero()) {
        return poly(1);
    }
    auto [a, Tr] = full_gcd(R1, R2);
    int dr = (d + 1) - a[0].deg();
    int dp = 0;
    for(size_t i = 0; i + 1 < a.size(); i++) {
        dr -= a[i + 1].deg();
        dp += a[i].deg();
        if(dr < dp) {
            auto ans = convergent(begin(a), begin(a) + i + 1);
            return ans.a / ans.a.lead();
        }
```

```
    }
    auto ans = convergent(begin(a), end(a));
    return ans.a / ans.a.lead();
}

// calculate inv to *this modulo t
// quadratic complexity
optional<poly> inv_mod_slow(poly const& t) const {
    auto R1 = *this, R2 = t;
    auto Q1 = poly(T(1)), Q2 = poly(T(0));
    int k = 0;
    while(!R2.is_zero()) {
        k ^= 1;
        auto [a, nR] = R1.divmod(R2);
        tie(R1, R2) = make_tuple(R2, nR);
        tie(Q1, Q2) = make_tuple(Q2, Q1 + a * Q2);
    }
    if(R1.deg() > 0) {
        return nullopt;
    } else {
        return (k ? -Q1 : Q1) / R1[0];
    }
}

optional<poly> inv_mod(poly const &t) const {
    assert(!t.is_zero());
    if(false && min(deg(), t.deg()) < magic) {
        return inv_mod_slow(t);
    }
    auto A = t, B = *this % t;
    auto [a, Tr] = full_gcd(A, B);
    auto g = Tr.d * A - Tr.b * B;
    if(g.deg() != 0) {
        return nullopt;
    }
    return -Tr.b / g[0];
};

poly operator / (const poly &t) const {return divmod(t).first;}
poly operator % (const poly &t) const {return divmod(t).second;}
poly operator /= (const poly &t) {return *this = divmod(t).first;}
poly operator %= (const poly &t) {return *this = divmod(t).second;}
poly operator *= (const T &x) {
    for(auto &it: a) {
        it *= x;
    }
    normalize();
    return *this;
}
poly operator /= (const T &x) {
    return *this *= x.inv();
}
poly operator * (const T &x) const {return poly(*this) *= x;}
poly operator / (const T &x) const {return poly(*this) /= x;}

poly conj() const { // A(x) -> A(-x)
    auto res = *this;
    for(int i = 1; i <= deg(); i += 2) {
        res.a[i] = -res[i];
```

```
    }
    return res;
}

void print(int n) const {
    for(int i = 0; i < n; i++) {
        cout << (*this)[i] << ' ';
    }
    cout << "\n";
}

void print() const {
    print(deg() + 1);
}

T eval(T x) const { // evaluates in single point x
    T res(0);
    for(int i = deg(); i >= 0; i--) {
        res *= x;
        res += a[i];
    }
    return res;
}

T lead() const { // leading coefficient
    assert(!is_zero());
    return a.back();
}

int deg() const { // degree, -1 for P(x) = 0
    return (int)a.size() - 1;
}

bool is_zero() const {
    return a.empty();
}

T operator [](int idx) const {
    return idx < 0 || idx > deg() ? T(0) : a[idx];
}

T& coef(size_t idx) { // mutable reference at coefficient
    return a[idx];
}

bool operator == (const poly &t) const {return a == t.a;}
bool operator != (const poly &t) const {return a != t.a;}

poly deriv(int k = 1) { // calculate derivative
    if(deg() + 1 < k) {
        return poly(T(0));
    }
    vector<T> res(deg() + 1 - k);
    for(int i = k; i <= deg(); i++) {
        res[i - k] = fact<T>(i) * rfact<T>(i - k) * a[i];
    }
    return res;
}
```

```
poly integr() { // calculate integral with C = 0
    vector<T> res(deg() + 2);
    for(int i = 0; i <= deg(); i++) {
        res[i + 1] = a[i] * small_inv<T>(i + 1);
    }
    return res;
}

size_t trailing_xk() const { // Let p(x) = x^k * t(x), return k
    if(is_zero()) {
        return -1;
    }
    int res = 0;
    while(a[res] == T(0)) {
        res++;
    }
    return res;
}

poly log(size_t n) { // calculate log p(x) mod x^n
    assert(a[0] == T(1));
    return (deriv().mod_xk(n) * inv(n)).integr().mod_xk(n);
}

poly exp(size_t n) { // calculate exp p(x) mod x^n
    if(is_zero()) {
        return T(1);
    }
    assert(a[0] == T(0));
    poly ans = T(1);
    size_t a = 1;
    while(a < n) {
        poly C = ans.log(2 * a).div_xk(a) - substr(a, 2 * a);
        ans -= (ans * C).mod_xk(a).mul_xk(a);
        a *= 2;
    }
    return ans.mod_xk(n);
}

poly pow_bin(int64_t k, size_t n) { // O(n log n log k)
    if(k == 0) {
        return poly(1).mod_xk(n);
    } else {
        auto t = pow(k / 2, n);
        t = (t * t).mod_xk(n);
        return (k % 2 ? *this * t : t).mod_xk(n);
    }
}

// Do not compute inverse from scratch
poly powmod_hint(int64_t k, poly const& md, poly const& mdinv) {
    if(k == 0) {
        return poly(1);
    } else {
        auto t = powmod_hint(k / 2, md, mdinv);
        t = (t * t).divmod_hint(md, mdinv).second;
        if(k % 2) {
            t = (t * *this).divmod_hint(md, mdinv).second;
        }
```

```
        return t;
    }
}

poly circular_closure(size_t m) const {
    if(deg() == -1) {
        return *this;
    }
    auto t = *this;
    for(size_t i = t.deg(); i >= m; i--) {
        t.a[i - m] += t.a[i];
    }
    t.a.resize(min(t.a.size(), m));
    return t;
}

static poly mul_circular(poly const& a, poly const& b, size_t m) {
    return (a.circular_closure(m) * b.circular_closure(m)).circular_closure(m);
}

poly powmod_circular(int64_t k, size_t m) {
    if(k == 0) {
        return poly(1);
    } else {
        auto t = powmod_circular(k / 2, m);
        t = mul_circular(t, t, m);
        if(k % 2) {
            t = mul_circular(t, *this, m);
        }
        return t;
    }
}

poly powmod(int64_t k, poly const& md) {
    int d = md.deg();
    if(d == -1) {
        return k ? *this : poly(T(1));
    }
    if(md == xk(d)) {
        return pow(k, d);
    }
    if(md == xk(d) - poly(T(1))) {
        return powmod_circular(k, d);
    }
    auto mdinv = md.reverse().inv(md.deg() + 1);
    return powmod_hint(k, md, mdinv);
}

// O(d * n) with the derivative trick from
// https://codeforces.com/blog/entry/73947?#comment-581173
poly pow_dn(int64_t k, size_t n) {
    if(n == 0) {
        return poly(T(0));
    }
    assert((*this)[0] != T(0));
    vector<T> Q(n);
    Q[0] = bpow(a[0], k);
    auto a0inv = a[0].inv();
    for(int i = 1; i < (int)n; i++) {
```

```
        for(int j = 1; j <= min(deg(), i); j++) {
            Q[i] += a[j] * Q[i - j] * (T(k) * T(j) - T(i - j));
        }
        Q[i] *= small_inv<T>(i) * a0inv;
    }
    return Q;
}

// calculate p^k(n) mod x^n in O(n log n)
// might be quite slow due to high constant
poly pow(int64_t k, size_t n) {
    if(is_zero()) {
        return k ? *this : poly(1);
    }
    int i = trailing_xk();
    if(i > 0) {
        return k >= int64_t(n + i - 1) / i ? poly(T(0)) : div_xk(i).pow(k, n - i
            * k).mul_xk(i * k);
    }
    if(min(deg(), (int)n) <= magic) {
        return pow_dn(k, n);
    }
    if(k <= magic) {
        return pow_bin(k, n);
    }
    T j = a[i];
    poly t = *this / j;
    return bpow(j, k) * (t.log(n) * T(k)).exp(n).mod_xk(n);
}

// returns nullopt if undefined
optional<poly> sqrt(size_t n) const {
    if(is_zero()) {
        return *this;
    }
    int i = trailing_xk();
    if(i % 2) {
        return nullopt;
    } else if(i > 0) {
        auto ans = div_xk(i).sqrt(n - i / 2);
        return ans ? ans->mul_xk(i / 2) : ans;
    }
    auto st = (*this)[0].sqrt();
    if(st) {
        poly ans = *st;
        size_t a = 1;
        while(a < n) {
            a *= 2;
            ans -= (ans - mod_xk(a) * ans.inv(a)).mod_xk(a) / 2;
        }
        return ans.mod_xk(n);
    }
    return nullopt;
}

poly mulx(T a) const { // component-wise multiplication with a^k
    T cur = 1;
    poly res(*this);
    for(int i = 0; i <= deg(); i++) {
```

```
        res.coef(i) *= cur;
        cur *= a;
    }
    return res;
}

poly mulx_sq(T a) const { // component-wise multiplication with a^{k choose 2}
    T cur = 1, total = 1;
    poly res(*this);
    for(int i = 0; i <= deg(); i++) {
        res.coef(i) *= total;
        cur *= a;
        total *= cur;
    }
    return res;
}

// be mindful of maxn, as the function
// requires multiplying polynomials of size deg() and n+deg()!
poly chirpz(T z, int n) const { // P(1), P(z), P(z^2), ..., P(z^(n-1))
    if(is_zero()) {
        return vector<T>(n, 0);
    }
    if(z == T(0)) {
        vector<T> ans(n, (*this)[0]);
        if(n > 0) {
            ans[0] = accumulate(begin(a), end(a), T(0));
        }
        return ans;
    }
    auto A = mulx_sq(z.inv());
    auto B = ones(n+deg()).mulx_sq(z);
    return semicorr(B, A).mod_xk(n).mulx_sq(z.inv());
}

// res[i] = prod_{1 <= j <= i} 1/(1 - z^j)
static auto _1mzk_prod_inv(T z, int n) {
    vector<T> res(n, 1), zk(n);
    zk[0] = 1;
    for(int i = 1; i < n; i++) {
        zk[i] = zk[i - 1] * z;
        res[i] = res[i - 1] * (T(1) - zk[i]);
    }
    res.back() = res.back().inv();
    for(int i = n - 2; i >= 0; i--) {
        res[i] = (T(1) - zk[i+1]) * res[i+1];
    }
    return res;
}

// prod_{0 <= j < n} (1 - z^j x)
static auto _1mzkx_prod(T z, int n) {
    if(n == 1) {
        return poly(vector<T>{1, -1});
    } else {
        auto t = _1mzkx_prod(z, n / 2);
        t *= t.mulx(bpow(z, n / 2));
        if(n % 2) {
            t *= poly(vector<T>{1, -bpow(z, n - 1)});
```

```
        }
        return t;
    }
}

poly chirpz_inverse(T z, int n) const { // P(1), P(z), P(z^2), ..., P(z^(n-1))
    if(is_zero()) {
        return {};
    }
    if(z == T(0)) {
        if(n == 1) {
            return *this;
        } else {
            return vector{(*this)[1], (*this)[0] - (*this)[1]};
        }
    }
    vector<T> y(n);
    for(int i = 0; i < n; i++) {
        y[i] = (*this)[i];
    }
    auto prods_pos = _1mzk_prod_inv(z, n);
    auto prods_neg = _1mzk_prod_inv(z.inv(), n);

    T zn = bpow(z, n-1).inv();
    T znk = 1;
    for(int i = 0; i < n; i++) {
        y[i] *= znk * prods_neg[i] * prods_pos[(n - 1) - i];
        znk *= zn;
    }

    poly p_over_q = poly(y).chirpz(z, n);
    poly q = _1mzkx_prod(z, n);

    return (p_over_q * q).mod_xk(n).reverse(n);
}

static poly build(vector<poly> &res, int v, auto L, auto R) { // builds
   evaluation tree for (x-a1)(x-a2)...(x-an)
    if(R - L == 1) {
        return res[v] = vector<T>{-*L, 1};
    } else {
        auto M = L + (R - L) / 2;
        return res[v] = build(res, 2 * v, L, M) * build(res, 2 * v + 1, M, R);
    }
}

poly to_newton(vector<poly> &tree, int v, auto l, auto r) {
    if(r - l == 1) {
        return *this;
    } else {
        auto m = l + (r - l) / 2;
        auto A = (*this % tree[2 * v]).to_newton(tree, 2 * v, l, m);
        auto B = (*this / tree[2 * v]).to_newton(tree, 2 * v + 1, m, r);
        return A + B.mul_xk(m - l);
    }
}

poly to_newton(vector<T> p) {
    if(is_zero()) {
```

```
        return *this;
    }
    int n = p.size();
    vector<poly> tree(4 * n);
    build(tree, 1, begin(p), end(p));
    return to_newton(tree, 1, begin(p), end(p));
}

vector<T> eval(vector<poly> &tree, int v, auto l, auto r) { // auxiliary
   evaluation function
    if(r - l == 1) {
        return {eval(*l)};
    } else {
        auto m = l + (r - l) / 2;
        auto A = (*this % tree[2 * v]).eval(tree, 2 * v, l, m);
        auto B = (*this % tree[2 * v + 1]).eval(tree, 2 * v + 1, m, r);
        A.insert(end(A), begin(B), end(B));
        return A;
    }
}

vector<T> eval(vector<T> x) { // evaluate polynomial in (x1, ..., xn)
    int n = x.size();
    if(is_zero()) {
        return vector<T>(n, T(0));
    }
    vector<poly> tree(4 * n);
    build(tree, 1, begin(x), end(x));
    return eval(tree, 1, begin(x), end(x));
}

poly inter(vector<poly> &tree, int v, auto l, auto r, auto ly, auto ry) { //
   auxiliary interpolation function
    if(r - l == 1) {
        return {*ly / a[0]};
    } else {
        auto m = l + (r - l) / 2;
        auto my = ly + (ry - ly) / 2;
        auto A = (*this % tree[2 * v]).inter(tree, 2 * v, l, m, ly, my);
        auto B = (*this % tree[2 * v + 1]).inter(tree, 2 * v + 1, m, r, my, ry);
        return A * tree[2 * v + 1] + B * tree[2 * v];
    }
}

static auto inter(vector<T> x, vector<T> y) { // interpolates minimum polynomial
    from (xi, yi) pairs
    int n = x.size();
    vector<poly> tree(4 * n);
    return build(tree, 1, begin(x), end(x)).deriv().inter(tree, 1, begin(x), end
       (x), begin(y), end(y));
}

static auto resultant(poly a, poly b) { // computes resultant of a and b
    if(b.is_zero()) {
        return 0;
    } else if(b.deg() == 0) {
        return bpow(b.lead(), a.deg());
    } else {
        int pw = a.deg();
```

```
        a %= b;
        pw -= a.deg();
        auto mul = bpow(b.lead(), pw) * T((b.deg() & a.deg() & 1) ? -1 : 1);
        auto ans = resultant(b, a);
        return ans * mul;
    }
}

static poly xk(size_t n) { // P(x) = x^n
    return poly(T(1)).mul_xk(n);
}

static poly ones(size_t n) { // P(x) = 1 + x + ... + x^{n-1}
    return vector<T>(n, 1);
}

static poly expx(size_t n) { // P(x) = e^x (mod x^n)
    return ones(n).borel();
}

static poly log1px(size_t n) { // P(x) = log(1+x) (mod x^n)
    vector<T> coeffs(n, 0);
    for(size_t i = 1; i < n; i++) {
        coeffs[i] = (i & 1 ? T(i).inv() : -T(i).inv());
    }
    return coeffs;
}

static poly log1mx(size_t n) { // P(x) = log(1-x) (mod x^n)
    return -ones(n).integr();
}

// [x^k] (a corr b) = sum_{i} a{(k-m)+i}*bi
static poly corr(poly a, poly b) { // cross-correlation
    return a * b.reverse();
}

// [x^k] (a semicorr b) = sum_i a{i+k} * b{i}
static poly semicorr(poly a, poly b) {
    return corr(a, b).div_xk(b.deg());
}

poly invborel() const { // ak *= k!
    auto res = *this;
    for(int i = 0; i <= deg(); i++) {
        res.coef(i) *= fact<T>(i);
    }
    return res;
}

poly borel() const { // ak /= k!
    auto res = *this;
    for(int i = 0; i <= deg(); i++) {
        res.coef(i) *= rfact<T>(i);
    }
    return res;
}

poly shift(T a) const { // P(x + a)
```

```
    return semicorr(invborel(), expx(deg() + 1).mulx(a)).borel();
}

poly x2() { // P(x) -> P(x^2)
    vector<T> res(2 * a.size());
    for(size_t i = 0; i < a.size(); i++) {
        res[2 * i] = a[i];
    }
    return res;
}

// Return {P0, P1}, where P(x) = P0(x) + xP1(x)
pair<poly, poly> bisect() const {
    vector<T> res[2];
    res[0].reserve(deg() / 2 + 1);
    res[1].reserve(deg() / 2 + 1);
    for(int i = 0; i <= deg(); i++) {
        res[i % 2].push_back(a[i]);
    }
    return {res[0], res[1]};
}

// Find [x^k] P / Q
static T kth_rec(poly P, poly Q, int64_t k) {
    while(k > Q.deg()) {
        int n = Q.a.size();
        auto [Q0, Q1] = Q.mulx(-1).bisect();
        auto [P0, P1] = P.bisect();

        int N = fft::com_size((n + 1) / 2, (n + 1) / 2);

        auto Q0f = fft::dft(Q0.a, N);
        auto Q1f = fft::dft(Q1.a, N);
        auto P0f = fft::dft(P0.a, N);
        auto P1f = fft::dft(P1.a, N);

        if(k % 2) {
            P = poly(Q0f * P1f) + poly(Q1f * P0f);
        } else {
            P = poly(Q0f * P0f) + poly(Q1f * P1f).mul_xk(1);
        }
        Q = poly(Q0f * Q0f) - poly(Q1f * Q1f).mul_xk(1);
        k /= 2;
    }
    return (P * Q.inv(Q.deg() + 1))[k];
}

poly inv(int n) const { // get inverse series mod x^n
    auto Q = mod_xk(n);
    if(n == 1) {
        return Q[0].inv();
    }
    // Q(-x) = P0(x^2) + xP1(x^2)
    auto [P0, P1] = Q.mulx(-1).bisect();

    int N = fft::com_size((n + 1) / 2, (n + 1) / 2);

    auto P0f = fft::dft(P0.a, N);
    auto P1f = fft::dft(P1.a, N);
```

```
    auto TTf = fft::dft(( // Q(x)*Q(-x) = Q0(x^2)^2 - x^2 Q1(x^2)^2
        poly(P0f * P0f) - poly(P1f * P1f).mul_xk(1)
    ).inv((n + 1) / 2).a, N);

    return (
        poly(P0f * TTf).x2() + poly(P1f * TTf).x2().mul_xk(1)
    ).mod_xk(n);
}

// compute A(B(x)) mod x^n in O(n^2)
static poly compose(poly A, poly B, int n) {
    int q = std::sqrt(n);
    vector<poly> Bk(q);
    auto Bq = B.pow(q, n);
    Bk[0] = poly(T(1));
    for(int i = 1; i < q; i++) {
        Bk[i] = (Bk[i - 1] * B).mod_xk(n);
    }
    poly Bqk(1);
    poly ans;
    for(int i = 0; i <= n / q; i++) {
        poly cur;
        for(int j = 0; j < q; j++) {
            cur += Bk[j] * A[i * q + j];
        }
        ans += (Bqk * cur).mod_xk(n);
        Bqk = (Bqk * Bq).mod_xk(n);
    }
    return ans;
}

// compute A(B(x)) mod x^n in O(sqrt(pqn log^3 n))
// preferrable when p = deg A and q = deg B
// are much less than n
static poly compose_large(poly A, poly B, int n) {
    if(B[0] != T(0)) {
        return compose_large(A.shift(B[0]), B - B[0], n);
    }

    int q = std::sqrt(n);
    auto [B0, B1] = make_pair(B.mod_xk(q), B.div_xk(q));

    B0 = B0.div_xk(1);
    vector<poly> pw(A.deg() + 1);
    auto getpow = [&](int k) {
        return pw[k].is_zero() ? pw[k] = B0.pow(k, n - k) : pw[k];
    };

    function<poly(poly const&, int, int)> compose_dac = [&getpow, &compose_dac](
       poly const& f, int m, int N) {
        if(f.deg() <= 0) {
            return f;
        }
        int k = m / 2;
        auto [f0, f1] = make_pair(f.mod_xk(k), f.div_xk(k));
        auto [A, B] = make_pair(compose_dac(f0, k, N), compose_dac(f1, m - k, N
           - k));
```

```
                return (A + (B.mod_xk(N - k) * getpow(k).mod_xk(N - k)).mul_xk(k)).
                    mod_xk(N);
            };

            int r = n / q;
            auto Ar = A.deriv(r);
            auto AB0 = compose_dac(Ar, Ar.deg() + 1, n);

            auto Bd = B0.mul_xk(1).deriv();

            poly ans = T(0);

            vector<poly> B1p(r + 1);
            B1p[0] = poly(T(1));
            for(int i = 1; i <= r; i++) {
                B1p[i] = (B1p[i - 1] * B1.mod_xk(n - i * q)).mod_xk(n - i * q);
            }
            while(r >= 0) {
                ans += (AB0.mod_xk(n - r * q) * rfact<T>(r) * B1p[r]).mul_xk(r * q).
                    mod_xk(n);
                r--;
                if(r >= 0) {
                    AB0 = ((AB0 * Bd).integr() + A[r] * fact<T>(r)).mod_xk(n);
                }
            }

            return ans;
        }
    };

    static auto operator * (const auto& a, const poly<auto>& b) {
        return b * a;
    }
};

using namespace algebra;

const int mod = 998244353;
typedef modular<mod> base;
typedef poly<base> polyn;

void solve() {
    int n;
    cin >> n;
    vector<base> f(n), g(n);
    copy_n(istream_iterator<base>(cin), n, begin(f));
    copy_n(istream_iterator<base>(cin), n, begin(g));
    polyn::compose(f, g, n).print(n);
}

signed main() {
    //freopen("input.txt", "r", stdin);
    ios::sync_with_stdio(0);
    cin.tie(0);
    int t;
    t = 1;// cin >> t;
    while(t--) {
        solve();
    }
```

```
}
```

2.8 Fraction

```
template <class T>
struct fraction{
  T numerator, denominator;
  fraction(T n, T d){
    numerator = n, denominator = d;
        simplify();
  }
  fraction(){
    numerator = 0, denominator = 1;
  }

  fraction& operator += (const fraction& rhs){
    numerator = numerator * rhs.denominator + rhs.numerator * denominator, denominator
       *= rhs.denominator;
        simplify();
    return *this;
    }

        fraction& operator *= (const fraction& rhs){
        numerator *= rhs.numerator, denominator *= rhs.denominator;
        simplify();
    return *this;
  }

  friend fraction operator * (fraction lhs, const fraction& rhs){
    lhs *= rhs;
        lhs.simplify();
    return lhs;
  }

  friend ostream& operator << (ostream& os, const fraction& obj){
    os << obj.numerator << ' ' << obj.denominator;
    return os;
  }

  void simplify(){
    T ndgcd = gcd(numerator, denominator);
    numerator /= ndgcd, denominator /= ndgcd;
  }
};
```

3 Data Structures

3.1 Disjoint-set Union

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The structure is very flexible as you can add different operations to it.

```
class disjoint_set_union {
    vector<int> parent; // ancestor
    vector<int> minn, maxx;
    vector<int> count;

    vector<bool> config;
    public:
        disjoint_set_union(int length) {
            parent.resize(length + 1);
            iota(parent.begin(), parent.end(), 0);
        }

        disjoint_set_union(int length) {
            parent.resize(length + 1);
            iota(parent.begin(), parent.end(), 0);
            count.resize(length + 1, 1);
        }

        int find(int node) {
            return parent[node] = parent[node] == node ? node : find(parent[node]);
        }

        void joint(int x, int y) {
            int rootx = find(x), rooty = find(y);

            if (rootx != rooty) {
                parent[rooty] = rootx;
                count[rootx] += count[rooty];
            }
        }
};
```

3.2 Binary Indexed Tree / Fenwick Tree

```
class fenwick {
    // must be one-based
    vector<long long> bit;
    int size = 0;

public:
    fenwick(int length) {
        bit.resize(length + 1);
        this->size = length + 1;
    }

    void update(int pos, int val) {
        for (; pos < size; pos += pos & (-pos)) bit[pos] += val;
    }

    long long query(int pos) {
        long long ans = 0;
```

```
        for (;pos > 0; pos -= pos & (-pos)) ans += bit[pos];
        return ans;
    }
};
```

3.3 Segment Tree

3.3.1 Basic Segment Tree

```
template <class T>
class segment_tree {
private:
    int n;
    vector<T> tree;
    T operation(T left, T right) {
        return left + right;
    }

    void pushup(int node) {
        tree[node] = operation(tree[node + node], tree[node + node + 1]);
    }

    void build(const vector<T>& v, int curr, int lo, int hi) {
        if (lo == hi) {
            tree[curr] = v[lo];
        } else {
            int mi = lo + (hi - lo) / 2;
            build(v, curr + curr, lo, mi);
            build(v, curr + curr + 1, mi + 1, hi);
            pushup(curr);
        }
    }

    void set(int idx, int val, int curr, int lo, int hi) {
        if (lo == hi) {
            tree[curr] = val;
        } else {
            int mi = lo + (hi - lo) / 2;
            if (idx <= mi) {
                set(idx, val, curr + curr, lo, mi);
            } else {
                set(idx, val, curr + curr + 1, mi + 1, hi);
            }
            pushup(curr);
        }
    }

    T query(int l, int r, int curr, int lo, int hi) { // l, r is the target range!
        if (lo == l && hi == r) {
            return tree[curr];
        } else {
            int mi = lo + (hi - lo) / 2;
            if (r <= mi) {
                return query(l, r, curr + curr, lo, mi);
            } else if (l >= mi + 1) {
                return query(l, r, curr + curr + 1, mi + 1, hi);
            } else {
                return operation(query(l, mi, curr + curr, lo, mi), query(mi + 1, r,
                    curr + curr + 1, mi + 1, hi));
```

```
            }
        }
    }

public:
    segment_tree(const vector<T>& v) : n(v.size()), tree(4 * v.size()) {
        build(v, 1, 0, n - 1);
    }

    void set(int idx, T val) {
        set(idx, val, 1, 0, n - 1);
    }

    T query(int l, int r) {
        return query(l, r, 1, 0, n - 1);
    }
};
```

3.3.2 Online Segment with Maximum Sum

```
class segment_tree {
    int _size = 0;
    int power2size = 1;
    vector<long long> v;
    vector<long long> pref, suf, sum;

    public:
        segment_tree(int length) {
            _size = length;
            v.resize(4 * _size, 0);
            pref.resize(4 * _size);
            suf.resize(4 * _size);
            sum.resize(4 * _size);

            length--;
            while (length > 0) power2size <<= 1, length >>= 1;
        }

        void set(int l, int r, int pos, int idx, long long x, int p2s) {
            if (inRange(pos, pos + 1, l, r) == 2) {
                int mid = p2s >> 1;
                if (r - l != 1) {
                    set(l, l + mid, pos, 2 * idx + 1, x, mid);
                    set(l + mid, r, pos, 2 * idx + 2, x, mid);
                    v[idx] = max(max(v[2 * idx + 1], v[2 * idx + 2]), suf[2 * idx + 1] +
                        pref[2 * idx + 2]);
                    pref[idx] = max(pref[2 * idx + 1], sum[2 * idx + 1] + pref[2 * idx +
                        2]);
                    suf[idx] = max(suf[2 * idx + 2], sum[2 * idx + 2] + suf[2 * idx +
                       1]);
                    sum[idx] = sum[2 * idx + 1] + sum[2 * idx + 2];
                } else {
                    v[idx] = max(0LL, x);
                    pref[idx] = max(0LL, x);
                    suf[idx] = max(0LL, x);
                    sum[idx] = x;
                }
            }
```

```
        }

        void set(int pos, long long x) {
            set(0, _size, pos, 0, x, power2size);
        }

        // current left, current right, target left, target right
        long long ans(int l, int r, int lt, int rt, int idx, int p2s) {
            int checker = inRange(l, r, lt, rt);
            int mid = p2s >> 1;
            if (checker == 0) return 0;
            else if (checker == 1) return max(ans(l, l + mid, lt, rt, 2 * idx + 1, mid),
                ans(l + mid, r, lt, rt, 2 * idx + 2, mid));
            else return v[idx];
        }

        // [l, r)
        long long ans(int l, int r) {
            return ans(0, _size, l, r, 0, power2size);
        }

        int size() {
            return _size;
        }

        friend istream& operator >> (istream& is, segment_tree& obj) {
            for (int i = 0; i < obj.size(); i++) {
                int x;
                is >> x;
                obj.set(i, x);
            }
            return is;
        }

    private:
        // 0 = completely out of range, 1 = partially in range, 2 = completely in range
        // current segment, target segment
        int inRange(int l, int r, int lt, int rt) {
            if (r <= lt || l >= rt) return 0;
            else if (l >= lt && r <= rt) return 2;
            else return 1;
        }
};
```

3.3.3 Lazy Segment Tree

```
template <class T>
class segment_tree {
private:
    int n;
    vector<T> tree, diff; // tree = real value, diff = modification value
    vector<bool> lazy; // lazy = is the current node lazied

    T operation(T left, T right) {
        return min(left, right);
    }

    void pushup(int node) {
```

```
        if (lazy[node]) {
            // un-lazy itself
            lazy[node] = false;
            // lazy its children
            tree[node + node] = tree[node + node + 1] = diff[node];
            diff[node + node] = diff[node + node + 1] = diff[node];
            lazy[node + node] = lazy[node + node + 1] = true;
        }
        tree[node] = operation(tree[node + node], tree[node + node + 1]);
    }

    void build(const vector<T>& v, int curr, int lo, int hi) {
        if (lo == hi) {
            tree[curr] = v[lo];
        } else {
            int mi = lo + (hi - lo) / 2;
            build(v, curr + curr, lo, mi);
            build(v, curr + curr + 1, mi + 1, hi);
            pushup(curr);
        }
    }

    void set(int l, int r, T val, int curr, int lo, int hi) {
        if (lo == l && hi == r) {
            tree[curr] = val;
            diff[curr] = val;
            lazy[curr] = true;
        } else {
            pushup(curr);
            int mi = lo + (hi - lo) / 2;
            if (r <= mi) {
                set(l, r, val, curr + curr, lo, mi);
            } else if (l >= mi + 1) {
                set(l, r, val, curr + curr + 1, mi + 1, hi);
            } else {
                set(l, mi, val, curr + curr, lo, mi);
                set(mi + 1, r, val, curr + curr + 1, mi + 1, hi);
            }
            pushup(curr);
        }
    }

    T query(int l, int r, int curr, int lo, int hi) {
        if (lo == l && hi == r) {
            return tree[curr];
        } else {
            pushup(curr);
            int mi = lo + (hi - lo) / 2;
            if (r <= mi) {
                return query(l, r, curr + curr, lo, mi);
            } else if (l >= mi + 1) {
                return query(l, r, curr + curr + 1, mi + 1, hi);
            } else {
                return operation(query(l, mi, curr + curr, lo, mi), query(mi + 1, r,
                    curr + curr + 1, mi + 1, hi));
            }
        }
    }
public:
```

```
    segment_tree(const vector<T>& v) : n(v.size()), tree(4 * v.size()), diff(4 * v.size
        ()), lazy(4 * v.size()) {
        build(v, 1, 0, n - 1);
    }

    void set(int l, int r, T val) {
        set(l, r, val, 1, 0, n - 1);
        // cout << '\n';
    }

    T query(int l, int r) {
        return query(l, r, 1, 0, n - 1);
    }
};
```

3.4 Sparse Table

```
template <class T>
class sparse_table {
    int n, k;
    vector<vector<T>> table;

    T operation(T left, T right) {
        return left + right;
    }

    int log2_floor(unsigned long long i) {
        return i ? __builtin_clzll(1) - __builtin_clzll(i) : -1;
    }

    void build(const vector<T>& v) {
        // copy(v.begin(), v.end(), table[0]);
        for (int i = 0; i < n; i++) {
            table[0][i] = v[i];
        }
        for (int i = 1; i < k; i++) {
            for (int j = 0; j + (1 << i) <= n; j++) {
                table[i][j] = operation(table[i - 1][j], table[i - 1][j + (1 << (i - 1))
                    ]);
            }
        }
    }
public:
    sparse_table(const vector<T>& v) : n(v.size()), k(log2_floor(v.size()) + 1), table(k
        , vector<T>(v.size())) {
        build(v);
    }

    T query(int l, int r) {
        int size = r - l + 1;
        T ans = 0;
        for (int i = 0; i < k; i++) {
            if (size & 1) {
                ans = operation(ans, table[i][l]);
                l += 1 << i;
            }
            size >>= 1;
        }
```

```
        return ans;
    }
};
```

3.5 Min Heap / Max Heap

```
template <class T>
using max_heap = priority_queue<T, vector<T>>;

template <class T>
using min_heap = priority_queue<T, vector<T>, greater<T>>;
```

3.6 Linked List

3.6.1 Singly Linked List

To initiate a linked list:

```
list<int> lst(5, 0)
```

To insert element `x` to index `idx`:

```
auto it = lst.begin();
advance(it, idx);
lst.insert(it, x);
```

To erase element of index `idx`;

```
auto it = lst.begin();
advance(it, idx);
lst.erase(it);
```

3.6.2 Doubly Linked List

3.7 Lazy Min Heap

```
template <class T>
class lazy_min_heap {
    min_heap<T> pq, lazy;

public:
    lazy_min_heap() {}
    void push(T obj) {
        pq.push(obj);
    }

    void pop(T obj) {
        lazy.push(obj);
        while (!lazy.empty() && !pq.empty() && pq.top() == lazy.top()) {
            pq.pop();
            lazy.pop();
        }
    }

    T top() {
        return pq.top();
    }

    bool empty() {
        return pq.empty();
```

```
    }

    int size() {
        return pq.size() - lazy.size();
    }
};
```

3.8 Treap

3.8.1 Normal Treap

```
struct item {
    int key, prior;
    item *l, *r;
    item () { }
    item (int key) : key(key), prior(rand()), l(NULL), r(NULL) { }
    item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) { }
};
typedef item* pitem;

void split (pitem t, int key, pitem & l, pitem & r) {
    if (!t)
        l = r = NULL;
    else if (t->key <= key)
        split (t->r, key, t->r, r),  l = t;
    else
        split (t->l, key, l, t->l),  r = t;
}

void insert (pitem & t, pitem it) {
    if (!t)
        t = it;
    else if (it->prior > t->prior)
        split (t, it->key, it->l, it->r),  t = it;
    else
        insert (t->key <= it->key ? t->r : t->l, it);
}

void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
}

void erase (pitem & t, int key) {
    if (t->key == key) {
        pitem th = t;
        merge (t, t->l, t->r);
        delete th;
    }
    else
        erase (key < t->key ? t->l : t->r, key);
}

pitem unite (pitem l, pitem r) {
    if (!l || !r)  return l ? l : r;
```

```
    if (l->prior < r->prior)  swap (l, r);
    pitem lt, rt;
    split (r, l->key, lt, rt);
    l->l = unite (l->l, lt);
    l->r = unite (l->r, rt);
    return l;
}

// add calls of upd_cnt() to the end of insert, erase, split and merge
// to keep cnt values up-to-date.
int cnt (pitem t) {
    return t ? t->cnt : 0;
}

void upd_cnt (pitem t) {
    if (t)
        t->cnt = 1 + cnt(t->l) + cnt (t->r);
}
```

3.8.2 Implicit Treap

```
typedef struct item * pitem;
struct item {
    int prior, value, cnt;
    bool rev;
    pitem l, r;
};
// new implementation of split and merge
void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    int cur_key = add + cnt(t->l); //implicit key
    if (key <= cur_key)
        split (t->l, l, t->l, key, add),  r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
    upd_cnt (t);
}

int cnt (pitem it) {
    return it ? it->cnt : 0;
}

void upd_cnt (pitem it) {
    if (it)
        it->cnt = cnt(it->l) + cnt(it->r) + 1;
}
```

```
void push (pitem it) {
    if (it && it->rev) {
        it->rev = false;
        swap (it->l, it->r);
        if (it->l)  it->l->rev ^= true;
        if (it->r)  it->r->rev ^= true;
    }
}

void merge (pitem & t, pitem l, pitem r) {
    push (l);
    push (r);
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    push (t);
    int cur_key = add + cnt(t->l);
    if (key <= cur_key)
        split (t->l, l, t->l, key, add),  r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
    upd_cnt (t);
}

void reverse (pitem t, int l, int r) {
    pitem t1, t2, t3;
    split (t, t1, t2, l);
    split (t2, t2, t3, r-l+1);
    t2->rev ^= true;
    merge (t, t1, t2);
    merge (t, t, t3);
}

void output (pitem t) {
    if (!t)  return;
    push (t);
    output (t->l);
    printf ("%d ", t->value);
    output (t->r);
}
```

4 String Algorithms

4.1 Hashing

```
struct custom_hash {
    int mod = 31;
    int mod2 = 1000003957;
    // 1000003957, 1000001957, 1000003469, 1000003283, 1000002431
    // 1000010611, 1000009739, 1000009567, 1000012253, 1000011421
    void prefixHash(vector<long long>& dest, string& s) {
        dest.resize(s.size());
        dest[0] = s[0] - 'a';

        for (int i = 1; i < s.size(); i++) {
            dest[i] = dest[i - 1] * mod % mod2 + (s[i] - 'a');
            dest[i] %= mod2;
        }
    }

    void prefixHashSize(vector<long long>& dest, string& s, int size) {
        dest.resize(s.size());
        dest[0] = s[0];

        long long power = bigmd(mod, size, mod2);

        for (int i = 1; i < s.size(); i++) {
            dest[i] = dest[i - 1] * mod % mod2 + (s[i]);

            if (i >= size){
                long long minus = (s[i - size]) * power % mod2;
                dest[i] -= minus;
            }

            if (dest[i] < 0){
                dest[i] += mod2;
            }

            dest[i] %= mod2;
        }
    }

    void suffixHash(vector<long long>& dest, string& s) {
        dest.resize(s.size());
        dest[s.size() - 1] = s[s.size() - 1] - 'a';

        for (int i = s.size() - 2; i >= 0; i--) {
            dest[i] = dest[i + 1] * mod + (s[i] - 'a');
            dest[i] %= mod2;
        }
    }

    long long singlePrefixHash(string& s) {
        long long result = s[0] - 'a';
        for (int i = 1; i < s.size(); i++) {
            result = result * mod + (s[i] - 'a');
            result %= mod2;
        }
        return result;
    }
```

```
    long long singleSuffixHash(string& s) {
        long long result = s[s.size() - 1] - 'a';
        for (int i = s.size() - 2; i >= 0; i--) {
            result = result * mod + (s[i] - 'a');
            result %= mod2;
        }
        return result;
    }
} hasher;
```

4.2 Trie

```
struct trie_node{
    int children[26]; // index of the next node
    int isWord = -1; // if it is a word
    int cnt = 0;
};

class trie {
    vector<trie_node> v;
    int size = 1;
    public:
        trie(){
            v.resize(1);
        }

        void insert(string& s, int idx) {
            int currpos = 0;
            v[currpos].cnt++;
            for (int i = 0; i < s.size(); i++) {
                // create a new node if it doesn't exist
                if (v[currpos].children[s[i] - 'a'] == 0) {
                    v[currpos].children[s[i] - 'a'] = size++;
                    v.push_back(trie_node());
                }
                currpos = v[currpos].children[s[i] - 'a'];
                v[currpos].cnt++;

                // mark it as a word if it is the end of the loop
                if (i == s.size() - 1) v[currpos].isWord = idx;
            }
        }

        void remove(string& s) {
            int currpos = 0;
            v[currpos].cnt--;
            for (int i = 0; i < s.size(); i++) {
                int next = v[currpos].children[s[i] - 'a'];
                currpos = next;
                v[currpos].cnt--;
            }
        }

        int traverse(string& p) {
            int currpos = 0;
            for (int i = 0; i < p.size(); i++) {
                int next = v[currpos].children[p[i] - 'a'];
```

```
                if (next == 0) return 0;
                currpos = next;
            }
            return v[currpos].cnt;
        }
};
```

4.3 KM(P)

```
int ans = 0;
int index = 1e9;
for (int i = 0; i < t.size(); i++) {
    vector<int> next(t.size());
    next[i] = i - 1;

    for (int j = i + 1; j < t.size(); j++) {
        int p = next[j - 1] + 1;
        if (t[j] == t[p]) {
            next[j] = p;
        } else {
            next[j] = i - 1;
        }
    }

    int ptr = i - 1;
    for (int j = 0; j < s.size(); j++) {
        while (ptr >= i && t[ptr + 1] != s[j]) {
            ptr = next[ptr];
        }

        if (t[ptr + 1] == s[j]) {
            ptr++;
        }

        if (ptr - i + 1 > ans) {
            ans = ptr - i + 1;
            index = j - ans + 1;
        } else if (ptr - i + 1 == ans && index > j - ans) {
            index = j - ans + 1;
        }

        if (ptr == t.size()) {
            break;
        }
    }
}
```

5 Graph

5.1 Bipartite Graph

5.1.1 Bipartite Graph Checking

```
bool isBipartite(vector<bool>& visited, int node, vector<int>& color) {
    visited[node] = true;

    for (auto a : graph[node]) {
        if (!visited[a]) {
            color[a] = 1 - color[node];
            return isBipartite(visited, a, color);
        }else if (color[a] == color[node]) {
            return false;
        }
    }

    return true;
}
```

5.1.2 Maximum Bipartite Matching

Idea: We keep finding valid augmentation to graph until we can find none.

1. We start at a vertex A, and it connects to vertices $B_1, B_2, \cdots, B_k$.
2. We try to include edge (A, B_1) in our maximum matching. If B_1 is not the endpoint of any edge in the current matching configuration, we are done.
3. Otherwise, B1 has been matched to some other vertex, lets say C. We have to find if C can match to some other vertices (not B). Only if it is possible, we match A with B_1.
4. If A cannot match with B_1, try next connected vertex (B_2) and repeat from step 2, until you find one augmentation, or you declare that A cannot be included.

```
bool dfs(vector<bool>& visited, int node) {
    if (visited[node]) return false;

    visited[node] = true;
    for (auto to : graph[node]) {
        if (matching[to] == -1 || dfs(visited, matching[to])) {
            matching[to] = cur;
            return true;
        }
    }

    return false;
}

int maxmatch(vector<vector<int>>& graph) {
    int cnt = 0;
    vector<bool> visited(graph.size());
    for (int i = 1; i < graph.size(); i++) {
        fill(visited.begin(), visited.end(), false);
        cnt += dfs(i);
    }
    return cnt;
}
```

5.2 Lowest Common Ancestor

```
class lowest_common_ancestor {
    vector<vector<int>> parent;
    vector<int> depth;
    void dfs(vector<vector<int>>& graph, int node, int steppies) {
        depth[node] = steppies;
        for (auto other : graph[node]) {
            parent[0][other] = node;
            dfs(graph, other, steppies + 1);
        }
    }

    public:
        lowest_common_ancestor(int n, int root, vector<vector<int>>& graph) {
            parent.resize(20, vector(n + 1, -1));
            depth.resize(n + 1);
            dfs(graph, root, 1);
            for (int i = 1; i < 20; i++) {
                for (int j = 0; j < n; j++) {
                    if (parent[i - 1][j] != -1) parent[i][j] = parent[i - 1][parent[i -
                        1][j]];
                }
            }
        }

        int lift(int node, int steppies) {
            for (int i = 19; i >= 0; i--) {
                if (steppies & (1 << i)) node = parent[i][node];
            }
            return node;
        }

        int query(int lhs, int rhs) {
            if (depth[lhs] > depth[rhs]) swap(lhs, rhs);
            int required_steppies = depth[rhs] - depth[lhs];
            rhs = lift(rhs, required_steppies);

            if (lhs == rhs) return lhs;

            for (int i = 19; i >= 0; i--) {
                if (parent[i][lhs] != parent[i][rhs]) {
                    lhs = parent[i][lhs], rhs = parent[i][rhs];
                }
            }

            return parent[0][lhs];
        }
};
```

5.3 Graph Traversal

5.3.1 Dijkstra's Algorithm

A single source weighted shortest path algorithm in $O(n \log n)$

```
void dijkstra(vector<int>& visited, int node){
    min_heap<pair<int, int>> pq;
    pq.push({0, node});
    visited[node] = 0;
```

```
    while (!pq.empty()){
        pair<int, int> curr = pq.top();
        pq.pop();

        if (curr.first > visited[curr.second]) continue;

        for (auto a : graph[curr.second]){
            if (curr.first + a.second < visited[a.first]){
                visited[a.first] = curr.first + a.second;
                pq.push({curr.first + a.second, a.first});
            }
        }
    }
}
```

5.3.2 Bellman-Ford Algorithm

A single source weighted shortest path algorithm which supports negative edges in $O(VE)$.

```
vector<long long> dist(n, inf * inf);
dist[root] = 0;

for (int t = 0; t < n - 1; t++) {
    for (int i = 0; i < n; i++) {
        for (pair<int, int> other : graph[i]) {
            int u = i, v = other.first;
            if (dist[u] != inf * inf && dist[v] > dist[u] + other.second) {
                dist[v] = dist[u] + other.second;
            }
        }
    }
}

// for checking negative cycles
for (int i = 0; i < n; i++) {
    for (pair<int, int> other : graph[i]) {
        int u = i, v = other.first;
        if (dist[u] != inf * inf && dist[v] > dist[u] + other.second) {
            cout << "NEGATIVE CYCLE\n";
            return 0;
        }
    }
}
```

5.3.3 Floyd-Warshall Algorithm

An all pairs weighted shortest path algorithm which supports negative edges in $O(V^3)$.

```
vector dist(n, vector(n, inf + inf));
for (int i = 0; i < n; i++) {
    dist[i][i] = 0;
}

for (int i = 0; i < m; i++) {
    long long u, v, w;
    cin >> u >> v >> w;
    dist[u][v] = w;
}
```

```
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dist[i][k] != inf + inf && dist[k][j] != inf + inf) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}

// for checking negative cycles
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            if (dist[i][k] + dist[k][j] + dist[j][i] < 0) {
                cout << "NEGATIVE CYCLE\n";
                return 0;
            }
        }
    }
}
```

5.3.4 0-1 BFS

If the weights of the edges are either 0 or 1, we do not have to use a priority queue. Instead, we can use a deque. When we do insertion, if the newly added edge is 0, place it in front of the deque; otherwise, place it at the back of the deque.

```
vector<int> dist(n, INF);
dist[s] = 0;
deque<int> dq;
dq.push_front(s);
while (!dq.empty()) {
    int node = dq.front();
    dq.pop_front();
    for (auto edge : graph[node]) {
        int other = edge.first;
        int weight = edge.second;
        if (dist[other] + w < dist[node]) {
            dust[other] = dist[node] + w;
            if (w == 1)
                q.push_back(other);
            else
                q.push_front(other);
        }
    }
}
```

6 Geometry

6.1 Convex Hull

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i], include_collinear)
           )
            st.pop_back();
        st.push_back(a[i]);
    }

    a = st;
}
```