My Codebook

Felix Huang

September 9, 2022

Contents

1 Data-structures

1.1 DSU.h

```
class DSU {
public:
    DSU() : DSU(0) {}

    DSU(int _n) : n(_n), _size(vector<int>(n, -1)) {}

    inline int leader(int u) {
        assert(0 <= u && u < n);
        return (_size[u] < 0 ? u : (_size[u] = leader(_size[u])));
    }

    bool merge(int a, int b) {
        assert(0 <= a && a < n);
        assert(0 <= b && b < n);
        a = leader(a);
        b = leader(b);
        if(a == b) {
            return false;
        }
        if(-_size[a] < -_size[b]) {
            swap(a, b);
        }
        _size[a] += _size[b];
        _size[b] = a;
        return true;
```

```cpp
    }

    inline int size(int u) {
        assert(0 <= u && u < n);
        return -_size[leader(u)];
    }

    inline bool same(int a, int b) {
        assert(0 <= a && a < n);
        assert(0 <= b && b < n);
        return leader(a) == leader(b);
    }

    vector<vector<int>> groups() {
        vector<int> leader_buf(n), group_size(n);
        for(int i = 0; i < n; i++) {
            leader_buf[i] = leader(i);
            group_size[leader_buf[i]]++;
        }
        vector<vector<int>> result(n);
        for(int i = 0; i < n; i++) {
            result[i].reserve(group_size[i]);
        }
        for(int i = 0; i < n; i++) {
            result[leader_buf[i]].push_back(i);
        }
        result.erase(remove_if(result.begin(), result.end(), [](const vector<int>& v) {
            return v.empty();
        }), result.end());
        return result;
    }

private:
    int n;
    vector<int> _size;
};

```

1.2 Fenwick.h

```cpp
// 0-based index
template<class T>
class fenwick {
public:
    fenwick() : fenwick(0) {}

    fenwick(int _n) : n(_n), data(_n) {}

    void add(int p, T x) {
        assert(0 <= p && p < n);
        while(p < n) {
            data[p] += x;
            p |= (p + 1);
        }
    }

    T get(int p) {
        assert(0 <= p && p < n);
        T res{};
        while(p >= 0) {
            res += data[p];
            p = (p & (p + 1)) - 1;
        }
        return res;
    }

    T sum(int l, int r) {
        return get(r) - (l ? get(l - 1) : T{});
    }

private:
    int n;
    vector<T> data;
};

```

1.3 HashMap.h

```cpp
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

struct splitmix64_hash {
    static unsigned long long splitmix64(unsigned long long x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    unsigned long long operator()(unsigned long long x) const {
        static const unsigned long long FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count
        return splitmix64(x + FIXED_RANDOM);
    }
};

template<class T, class U, class H = splitmix64_hash> using hash_map = gp_hash_table<T, U, H>;
template<class T, class H = splitmix64_hash> using hash_set = hash_map<T, null_type, H>;

```

1.4 Segtree.h

```cpp
template<class T, T (*e)(), T (*op)(T, T)>
class segtree {
public:
    segtree() : segtree(0) {}

    segtree(int _n) : segtree(vector<T>(_n, e())) {}

    segtree(const vector<T>& a): n(int(a.size())) {
        log = 31 - __builtin_clz(2 * n - 1);
        size = 1 << log;
        st.resize(size << 1, e());
        for(int i = 0; i < n; ++i) {
            st[size + i] = a[i];
        }
        for(int i = size - 1; i; --i) {
            update(i);
        }
    }

    void set(int p, T val) {
        assert(0 <= p && p < n);
        p += size;
        st[p] = val;
        for(int i = 1; i <= log; ++i) {
            update(p >> i);
        }
    }

    inline T get(int p) const {
        assert(0 <= p && p < n);
        return st[p + size];
    }

    inline T operator[](int p) const {
        return get(p);
    }

    T prod(int l, int r) const {
        assert(0 <= l && l <= r && r <= n);
        T sml = e(), smr = e();
        l += size;
        r += size;
        while(l < r) {
            if(l & 1) {
                sml = op(sml, st[l++]);
            }
            if(r & 1) {
                smr = op(st[--r], smr);
            }
            l >>= 1;
            r >>= 1;
        }
        return op(sml, smr);
    }

    inline T all_prod() const { return st[1]; }

    template<bool (*f)(T)> int max_right(int l) const {
        return max_right(l, [](T x) { return f(x); });
    }

    template<class F> int max_right(int l, F f) const {
        assert(0 <= l && l <= n);
        assert(f(e()));
        if(l == n) {
            return n;
        }
        l += size;
        T sm = e();
        do {
            while(!(l & 1)) {
                l >>= 1;
            }
            if(!f(op(sm, st[l]))) {
                while(l < size) {
                    l <<= 1;
                    if(f(op(sm, st[l]))) {
                        sm = op(sm, st[l]);
                        l++;
                    }
                }
                return l - size;
            }
            sm = op(sm, st[l]);
            l++;
        } while((l & -l) != l);
        return n;
    }

    template<bool (*f)(T)> int min_left(int r) const {
        return min_left(r, [](T x) { return f(x); });
    }

    template<class F> int min_left(int r, F f) const {
        assert(0 <= r && r <= n);
        assert(f(e()));
        if(r == 0) {
            return 0;
        }
        r += size;
        T sm = e();
        do {
            r--;
            while(r > 1 && (r & 1)) {
                r >>= 1;
            }
            if(!f(op(st[r], sm))) {
                while(r < size) {
                    r = r << 1 | 1;
                    if(f(op(st[r], sm))) {
                        sm = op(st[r], sm);
                        r--;
                    }
                }
                return r + 1 - size;
            }
            sm = op(st[r], sm);
        } while((r & -r) != r);
        return 0;
    }

private:
    int n, size, log;
    vector<T> st;

    inline void update(int v) { st[v] = op(st[v << 1], st[v << 1 | 1]); }
};

```

1.5 LazySegtree.h

```
template<class S,
         S (*e)(),
         S (*op)(S, S),
         class F,
         F (*id)(),
         S (*mapping)(F, S),
         F (*composition)(F, F)>
class lazy_segtree {
public:
    lazy_segtree() : lazy_segtree(0) {}

    explicit lazy_segtree(int _n) : lazy_segtree(vector<S>(_n, e())) {}

    explicit lazy_segtree(const vector<S>& v) : n(int(v.size())) {
        log = 31 - __builtin_clz(2 * n - 1);
        size = 1 << log;
        d = vector<S>(size << 1, e());
        lz = vector<F>(size, id());
        for(int i = 0; i < n; i++) {
            d[size + i] = v[i];
        }
        for(int i = size - 1; i; --i) {
            update(i);
        }
    }

    void set(int p, S x) {
        assert(0 <= p && p < n);
        p += size;
        for(int i = log; i; --i) {
            push(p >> i);
        }
        d[p] = x;
        for(int i = 1; i <= log; ++i) {
            update(p >> i);
        }
    }

    S get(int p) {
        assert(0 <= p && p < n);
        p += size;
        for(int i = log; i; i--) {
            push(p >> i);
        }
        return d[p];
    }

    S operator[](int p) {
        return get(p);
    }

    S prod(int l, int r) {
        assert(0 <= l && l <= r && r <= n);
        if(l == r) {
            return e();
        }
        l += size;
        r += size;
        for(int i = log; i; i--) {
            if(((l >> i) << i) != l) {
                push(l >> i);
            }
            if(((r >> i) << i) != r) {
                push(r >> i);
            }
        }
        S sml = e(), smr = e();
        while(l < r) {
            if(l & 1) {
                sml = op(sml, d[l++]);
            }
            if(r & 1) {
                smr = op(d[--r], smr);
            }
            l >>= 1;
            r >>= 1;
        }
        return op(sml, smr);
    }

    S all_prod() const { return d[1]; }

    void apply(int p, F f) {
        assert(0 <= p && p < n);
        p += size;
        for(int i = log; i; i--) {
            push(p >> i);
        }
        d[p] = mapping(f, d[p]);
        for(int i = 1; i <= log; i++) {
            update(p >> i);
        }
    }
    void apply(int l, int r, F f) {
        assert(0 <= l && l <= r && r <= n);
        if(l == r) {
            return;
        }
        l += size;
        r += size;
        for(int i = log; i; i--) {
            if(((l >> i) << i) != l) {
                push(l >> i);
            }
            if(((r >> i) << i) != r) {
                push((r - 1) >> i);
            }
        }
        {
            int l2 = l, r2 = r;
            while(l < r) {
                if(l & 1) {
                    all_apply(l++, f);
                }
                if(r & 1) {
                    all_apply(--r, f);
                }
                l >>= 1;
                r >>= 1;
            }
            l = l2;
            r = r2;
        }
        for(int i = 1; i <= log; i++) {
            if(((l >> i) << i) != l) {
                update(l >> i);
            }
            if(((r >> i) << i) != r) {
                update((r - 1) >> i);
            }
        }
    }

    template<bool (*g)(S)> int max_right(int l) {
        return max_right(l, [](S x) { return g(x); });
    }

    template<class G> int max_right(int l, G g) {
        assert(0 <= l && l <= n);
        assert(g(e()));
        if(l == n) {
            return n;
        }
        l += size;
        for(int i = log; i; i--) {
            push(l >> i);
        }
        S sm = e();
        do {
            while(!(l & 1)) {
                l >>= 1;
            }
            if(!g(op(sm, d[l]))) {
                while(l < size) {
                    push(l);
                    l <<= 1;
                    if(g(op(sm, d[l]))) {
                        sm = op(sm, d[l]);
                        l++;
                    }
                }
                return l - size;
            }
            sm = op(sm, d[l]);
            l++;
        } while((l & -l) != l);
        return n;
    }

    template<bool (*g)(S)> int min_left(int r) {
        return min_left(r, [](S x) { return g(x); });
    }

    template<class G> int min_left(int r, G g) {
        assert(0 <= r && r <= n);
        assert(g(e()));
        if(r == 0) {
            return 0;
        }
        r += size;
        for(int i = log; i >= 1; i--) {
            push((r - 1) >> i);
        }
        S sm = e();
        do {
            r--;
            while(r > 1 && (r & 1)) {
                r >>= 1;
            }
            if(!g(op(d[r], sm))) {
                while(r < size) {
                    push(r);
                    r = r << 1 | 1;
                    if(g(op(d[r], sm))) {
                        sm = op(d[r], sm);
                        r--;
                    }
                }
                return r + 1 - size;
            }
            sm = op(d[r], sm);
        } while((r & -r) != r);
        return 0;
    }

private:
    int n, size, log;
    vector<S> d;
    vector<F> lz;

    inline void update(int k) { d[k] = op(d[k << 1], d[k << 1 | 1]); }

    void all_apply(int k, F f) {
        d[k] = mapping(f, d[k]);
        if(k < size) {
            lz[k] = composition(f, lz[k]);
        }
    }

    void push(int k) {
        all_apply(k << 1, lz[k]);
        all_apply(k << 1 | 1, lz[k]);
        lz[k] = id();
    }
};

```

1.6 OrderStatisticTree.h

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

template<class T, class Comp = less<T>> using ordered_set = tree<T, null_type, Comp, rb_tree_tag, tree_order_statistics_node_update>;
template<class T> using ordered_multiset = ordered_set<T, less_equal<T>>;
// Use `s.erase(s.find(x))` when using `ordered_multiset`
```

1.7 SparseTable.h

```
template<class T, T (*op)(T, T)>
class sparse_table {
public:
    sparse_table() : n(0) {}

    sparse_table(const vector<T>& a) {
        n = static_cast<int>(a.size());
        int max_log = 32 - __builtin_clz(n);
        mat.resize(max_log);
        mat[0] = a;
        for(int j = 1; j < max_log; ++j) {
            mat[j].resize(n - (1 << j) + 1);
            for(int i = 0; i <= n - (1 << j); ++i) {
                mat[j][i] = op(mat[j - 1][i], mat[j - 1][i + (1 << (j - 1))]);
            }
        }
    }

    inline T prod(int from, int to) const {
        assert(0 <= from && from <= to && to <= n - 1);
        int lg = 31 - __builtin_clz(to - from + 1);
        return op(mat[lg][from], mat[lg][to - (1 << lg) + 1]);
    }

    inline T operator[](int p) const {
        assert(0 <= p && p < n);
        return mat[0][p];
    }

private:
    int n;
    vector<vector<T>> mat;
};

```

1.8 ConvexHullTrick.h

```
struct Line_t {
    mutable long long k, m, p;

    inline bool operator<(const Line_t& o) const { return k < o.k; }
    inline bool operator<(long long x) const { return p < x; }
};

// returns maximum (with minimum use negative coefficient and constant)
struct CHT : multiset<Line_t, less<>> {
    // (for doubles, use INF = 1/.0, div(a,b) = a/b)
    static const long long INF = LLONG_MAX;
    long long div(long long a, long long b) { // floored division
        return a / b - ((a ^ b) < 0 && a % b);
    }

    bool isect(iterator x, iterator y) {
        if(y == end()) {
            x->p = INF;
            return 0;
        }
        if(x->k == y->k) {
            x->p = (x->m > y->m ? INF : -INF);
        } else {
            x->p = div(y->m - x->m, x->k - y->k);
        }
        return x->p >= y->p;
    }

    void insert_line(long long k, long long m) {
        auto z = insert({k, m, 0}), y = z++, x = y;
        while(isect(y, z)) {
            z = erase(z);
        }
        if(x != begin() && isect(--x, y)) {
            isect(x, y = erase(y));
        }
        while((y = x) != begin() && (--x)->p >= y->p) {
            isect(x, erase(y));
        }
    }

    long long eval(long long x) {
        assert(!empty());
        auto l = *lower_bound(x);
        return l.k * x + l.m;
    }
};

```

1.9 Treap.h

```
mt19937_64 rng(48763);

struct Node {
    long long val;
    long long sum;
    bool rev;
    int size;
    int pri;

    Node* l;
    Node* r;

    Node(long long x) : val(x), sum(x), rev(false), size(1), pri(rng()), l(NULL), r(NULL) {}
};

inline int size(Node*& v) {
    return (v ? v->size : 0);
}

void pull(Node*& v) {
    v->size = 1 + size(v->l) + size(v->r);
    v->sum = v->val + (v->l ? v->l->sum : 0) + (v->r ? v->r->sum : 0);
}

void push(Node*& v) {
    if(v->rev) {
        swap(v->l, v->r);
        if(v->l) {
            v->l->rev = !v->l->rev;
        }
        if(v->r) {
            v->r->rev = !v->r->rev;
        }
        v->rev = false;
    }
}

Node* merge(Node* a, Node* b) {
    if(!a || !b) {
        return (a ? a : b);
    }
    push(a);
    push(b);
    if(a->pri > b->pri) {
        a->r = merge(a->r, b);
        pull(a);
        return a;
    } else {
        b->l = merge(a, b->l);
        pull(b);
        return b;
    }
}

void split(Node* v, Node*& a, Node*& b, int k) {
    if(k == 0) {
        a = NULL;
        b = v;
        return;
    }
    push(v);
    if(size(v->l) >= k) {
        b = v;
        split(v->l, a, v->l, k);
        pull(b);
    } else {
        a = v;
        split(v->r, v->r, b, k - size(v->l) - 1);
        pull(a);
    }
}

```

2 Combinatorial

2.1 Combination.h

```
vector<mint> fact{1}, inv_fact{1};

void init_fact(int n) {
    while((int) fact.size() <= n) {
        fact.push_back(fact.back() * (int) fact.size());
    }
    int sz = (int) inv_fact.size();
    if(sz >= n + 1) {
        return;
    }
    inv_fact.resize(n + 1);
    inv_fact[n] = 1 / fact.back();
    for(int i = n - 1; i >= sz; --i) {
        inv_fact[i] = inv_fact[i + 1] * (i + 1);
    }
}

mint C(int n, int k) {
    if(k < 0 || k > n) {
        return 0;
    }
    init_fact(n);
    return fact[n] * inv_fact[k] * inv_fact[n - k];
}

mint P(int n, int k) {
    if(k < 0 || k > n) {
        return 0;
    }
    init_fact(n);
    return fact[n] * inv_fact[n - k];
}

```

2.2 CountInversions.h

```
// @return the number of inversions s.t i < j, a_i > a_j
template<class T>
long long countInversions(const vector<T>& a) {
    int n = (int) a.size();
    auto b = a;
    sort(b.begin(), b.end());
    b.erase(unique(b.begin(), b.end()), b.end());
    fenwick<int> fenw((int) b.size() + 1);
    long long ans = 0;
    for(int i = 0; i < n; ++i) {
        int x = lower_bound(b.begin(), b.end(), a[i]) - b.begin();
        ans += fenw.sum(x + 1, (int) b.size());
        fenw.add(x, 1);
    }
    return ans;
}

```

3 Number-theory

3.1 ExtendGCD.h

```
// @return x,y s.t. ax + by = gcd(a,b)
long long 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 x2, y2;
    long long c = a % b;
    if(c < 0) {
        c += b;
    }
    long long g = ext_gcd(b, c, x2, y2);
    x = y2;
    y = x2 - (a / b) * y2;
    return g;
}

```

3.2 InvGCD.h

```cpp

// @param $1 \leq b$
// @return $g, x$ s.t.
//     $g = \gcd(a, b)$
//     $ax = g \pmod{b}$
//     $0 \leq x < \frac{b}{g}$
constexpr pair<long long, long long> inv_gcd(long long a, long long b) {
    a %= b;
    if(a < 0) {
        a += b;
    }

    if(a == 0) return {b, 0};

    long long s = b, t = a;
    long long m0 = 0, m1 = 1;

    while(t) {
        long long u = s / t;
        s -= t * u;
        m0 -= m1 * u;

        // swap(s, t);
        // swap(m0, m1);
        auto tmp = s;
        s = t;
        t = tmp;
        tmp = m0;
        m0 = m1;
        m1 = tmp;
    }
    if(m0 < 0) m0 += b / s;
    return {s, m0};
}

```

3.3 StaticModint.h

```cpp
template<int m>
class static_modint {
public:
    static constexpr int mod() {
        return m;
    }

    static_modint() : value(0) {}

    static_modint(long long v) {
        v %= mod();
        if(v < 0) {
            v += mod();
        }
        value = v;
    }

    const int& operator()() const {
        return value;
    }

    template<class T>
    explicit operator T() const {
        return static_cast<T>(value);
    }

    static_modint& operator+=(const static_modint& rhs) {
        value += rhs.value;
        if(value >= mod()) {
            value -= mod();
        }
        return *this;
    }

    static_modint& operator-=(const static_modint& rhs) {
        value -= rhs.value;
        if(value < 0) {
            value += mod();
        }
        return *this;
    }

    static_modint& operator*=(const static_modint& rhs) {
        value = (long long) value * rhs.value % mod();
        return *this;
    }

    static_modint& operator/=(const static_modint& rhs) {
        auto eg = inv_gcd(rhs.value, mod());
        assert(eg.first == 1);
        return *this *= eg.second;
    }

    template<class T>
    static_modint& operator+=(const T& rhs) {
        return *this += static_modint(rhs);
    }

    template<class T>
    static_modint& operator-=(const T& rhs) {
        return *this -= static_modint(rhs);
    }

    template<class T>
    static_modint& operator*=(const T& rhs) {
        return *this *= static_modint(rhs);
    }

    template<class T>
    static_modint& operator/=(const T& rhs) {
        return *this /= static_modint(rhs);
    }

    static_modint operator+() const {
        return *this;
    }

    static_modint operator-() const {
        return static_modint() - *this;
    }

    static_modint& operator++() {
        return *this += 1;
    }

    static_modint& operator--() {
        return *this -= 1;
    }

    static_modint operator++(int) {
        static_modint res(*this);
        *this += 1;
        return res;
    }

    static_modint operator--(int) {
        static_modint res(*this);
        *this -= 1;
        return res;
    }

    static_modint operator+(const static_modint& rhs) {
        return static_modint(*this) += rhs;
    }

    static_modint operator-(const static_modint& rhs) {
        return static_modint(*this) -= rhs;
    }

    static_modint operator*(const static_modint& rhs) {
        return static_modint(*this) *= rhs;
    }

    static_modint operator/(const static_modint& rhs) {
        return static_modint(*this) /= rhs;
    }

    inline bool operator==(const static_modint& rhs) const {
        return value == rhs();
    }

    inline bool operator!=(const static_modint& rhs) const {
        return !(*this == rhs);
    }

private:
    int value;
};

template<int m, class T> static_modint<m> operator+(const T& lhs, const static_modint<m>& rhs) {
    return static_modint<m>(lhs) += rhs;
}

template<int m, class T> static_modint<m> operator-(const T& lhs, const static_modint<m>& rhs) {
    return static_modint<m>(lhs) -= rhs;
}

template<int m, class T> static_modint<m> operator*(const T& lhs, const static_modint<m>& rhs) {
    return static_modint<m>(lhs) *= rhs;
}

template<int m, class T> static_modint<m> operator/(const T& lhs, const static_modint<m>& rhs) {
    return static_modint<m>(lhs) /= rhs;
}

template<int m>
istream& operator>>(istream& in, static_modint<m>& num) {
    long long x;
    in >> x;
    num = static_modint<m>(x);
    return in;
}

template<int m>
ostream& operator<<(ostream& out, const static_modint<m>& num) {
    return out << num();
}

using modint998244353 = static_modint<998244353>;
using modint1000000007 = static_modint<1000000007>;

```

3.4 DynamicModint.h

```
template<int id>
class dynamic_modint {
public:
    static int mod() {
        return int(bt.umod());
    }

    static void set_mod(int m) {
        assert(1 <= m);
        bt = barrett(m);
    }

    dynamic_modint() : value(0) {}

    dynamic_modint(long long v) {
        v %= mod();
        if(v < 0) {
            v += mod();
        }
        value = v;
    }

    const unsigned int& operator()() const {
        return value;
    }

    template<class T>
    explicit operator T() const {
        return static_cast<T>(value);
    }

    dynamic_modint& operator+=(const dynamic_modint& rhs) {
        value += rhs.value;
        if(value >= umod()) {
            value -= umod();
        }
        return *this;
    }

    template<class T>
    dynamic_modint& operator+=(const T& rhs) {
        return *this += dynamic_modint(rhs);
    }

    dynamic_modint& operator-=(const dynamic_modint& rhs) {
        value += mod() - rhs.value;
        if(value >= umod()) {
            value -= umod();
        }
        return *this;
    }

    template<class T>
    dynamic_modint& operator-=(const T& rhs) {
        return *this -= dynamic_modint(rhs);
    }

    dynamic_modint& operator*=(const dynamic_modint& rhs) {
        value = bt.mul(value, rhs.value);
        return *this;
    }

    template<class T>
    dynamic_modint& operator*=(const T& rhs) {
        return *this *= dynamic_modint(rhs);
    }

    dynamic_modint& operator/=(const dynamic_modint& rhs) {
        auto eg = inv_gcd(rhs.value, mod());
        assert(eg.first == 1);
        return *this *= eg.second;
    }

    template<class T>
    dynamic_modint& operator/=(const T& rhs) {
        return *this /= dynamic_modint(rhs);
    }

    dynamic_modint operator+() const {
        return *this;
    }

    dynamic_modint operator-() const {
        return dynamic_modint() - *this;
    }

    dynamic_modint& operator++() {
        ++value;
        if(value == umod()) {
            value = 0;
        }
        return *this;
    }

    dynamic_modint& operator--() {
        if(value == 0) {
            value = umod();
        }
        --value;
        return *this;
    }

    dynamic_modint operator++(int) {
        dynamic_modint res(*this);
        ++*this;
        return res;
    }

    dynamic_modint operator--(int) {
        dynamic_modint res(*this);
        --*this;
        return res;
    }

    dynamic_modint operator+(const dynamic_modint& rhs) {
        return dynamic_modint(*this) += rhs;
    }

    dynamic_modint operator-(const dynamic_modint& rhs) {
        return dynamic_modint(*this) -= rhs;
    }

    dynamic_modint operator*(const dynamic_modint& rhs) {
        return dynamic_modint(*this) *= rhs;
    }

    dynamic_modint operator/(const dynamic_modint& rhs) {
        return dynamic_modint(*this) /= rhs;
    }

    inline bool operator==(const dynamic_modint& rhs) const {
        return value == rhs();
    }

    inline bool operator!=(const dynamic_modint& rhs) const {
        return !(*this == rhs);
    }

private:
    unsigned int value;
    static barrett bt;
    static unsigned int umod() { return bt.umod(); }
};

template<int id, class T> dynamic_modint<id> operator+(const T& lhs, const dynamic_modint<id>& rhs) {
    return dynamic_modint<id>(lhs) += rhs;
```

```
}

template<int id, class T> dynamic_modint<id> operator-(const T& lhs, const dynamic_modint<id>& rhs) {
    return dynamic_modint<id>(lhs) -= rhs;
}

template<int id, class T> dynamic_modint<id> operator*(const T& lhs, const dynamic_modint<id>& rhs) {
    return dynamic_modint<id>(lhs) *= rhs;
}

template<int id, class T> dynamic_modint<id> operator/(const T& lhs, const dynamic_modint<id>& rhs) {
    return dynamic_modint<id>(lhs) /= rhs;
}

template<int id> barrett dynamic_modint<id>::bt(998244353);

template<int id>
istream& operator>>(istream& in, dynamic_modint<id>& num) {
    long long x;
    in >> x;
    num = dynamic_modint<id>(x);
    return in;
}

template<int id>
ostream& operator<<(ostream& out, const dynamic_modint<id>& num) {
    return out << num();
}

```

3.5 CRT.h

```
// @return
//     remainder, modulo
//             or
//     0, 0 if do not exist
pair<long long, long long> crt(const vector<long long>& r, const vector<long long>& m) {
    assert(r.size() == m.size());
    int n = (int) r.size();
    // Contracts: 0 <= r0 < m0
    long long r0 = 0, m0 = 1;
    for(int i = 0; i < n; i++) {
        assert(1 <= m[i]);
        long long r1 = r[i] % m[i];
        if(r1 < 0) r1 += m[i];
        long long m1 = m[i];
        if(m0 < m1) {
            swap(r0, r1);
            swap(m0, m1);
        }
        if(m0 % m1 == 0) {
            if(r0 % m1 != r1) return {0, 0};
            continue;
        }
        long long g, im;
        tie(g, im) = inv_gcd(m0, m1);

        long long u1 = (m1 / g);
        if((r1 - r0) % g) return {0, 0};

        long long x = (r1 - r0) / g % u1 * im % u1;

        r0 += x * m0;
        m0 *= u1;
        if(r0 < 0) r0 += m0;
    }
    return {r0, m0};
}

```

3.6 LinearSieve.h

```
vector<bool> isprime;
vector<int> primes;
vector<int> phi;
vector<int> mobius;
void linear_sieve(int n) {
    n += 1;
    isprime.resize(n);
    fill(isprime.begin() + 2, isprime.end(), true);
    phi.resize(n);
    mobius.resize(n);
    phi[1] = mobius[1] = 1;
    for(int i = 2; i < n; ++i) {
        if(isprime[i]) {
            primes.push_back(i);
            phi[i] = i - 1;
            mobius[i] = -1;
        }
        for(auto& j : primes) {
            if(i * j >= n) {
                break;
            }
            isprime[i * j] = false;
            if(i % j == 0) {
                mobius[i * j] = 0;
                phi[i * j] = phi[i] * j;
                break;
            } else {
                mobius[i * j] = mobius[i] * mobius[j];
                phi[i * j] = phi[i] * phi[j];
            }
        }
    }
}

```

3.7 ModInverses.h

```
// @return array A of length N s.t
//     i · A_i = 1  (mod m)
vector<int> mod_inverse(int m, int n = -1) {
    assert(n < m);
    if(n == -1) {
        n = m - 1;
    }
    vector<int> inv(n + 1);
    inv[0] = inv[1] = 1;
    for(int i = 2; i <= n; ++i) {
        inv[i] = m - (long long) (m / i) * inv[m % i] % m;
    }
    return inv;
}

```

3.8 PowMod.h

```
// @param 0 ≤ n
// @param 1 ≤ m
// @return x^n (mod m)
constexpr long long pow_mod_constexpr(long long x, long long n, int m) {
    if(m == 1) return 0;
    unsigned int _m = (unsigned int)(m);
    unsigned long long r = 1;
    x %= m;
    if(x < 0) {
        x += m;
    }
    unsigned long long y = x;
    while(n) {
        if(n & 1) r = (r * y) % _m;
        y = (y * y) % _m;
        n >>= 1;
    }
    return r;
}

```

3.9 IsPrime.h

```
// Reference:
// M. Forisek and J. Jancina,
// Fast Primality Testing forIntegers That Fit into a Machine Word
// @param n `0 <= n`
constexpr bool is_prime_constexpr(int n) {
    if(n <= 1) return false;
    if(n == 2 || n == 7 || n == 61) return true;
    if(n % 2 == 0) return false;
    long long d = n - 1;
    while(d % 2 == 0) d /= 2;
    constexpr long long bases[3] = {2, 7, 61};
    for(long long a : bases) {
        long long t = d;
        long long y = pow_mod_constexpr(a, t, n);
        while(t != n - 1 && y != 1 && y != n - 1) {
            y = y * y % n;
            t <<= 1;
        }
        if(y != n - 1 && t % 2 == 0) {
            return false;
        }
    }
    return true;
}
template<int n> constexpr bool is_prime = is_prime_constexpr(n);

```

3.10 Factorize.h

```
template<class T>
vector<pair<T, int>> MergeFactors(const vector<pair<T, int>>& a, const vector<pair<T, int>>& b) {
    vector<pair<T, int>> c;
    int i = 0, j = 0;
    while(i < (int) a.size() || j < (int) b.size()) {
        if(i < (int) a.size() && j < (int) b.size() && a[i].first == b[j].first) {
            c.emplace_back(a[i].first, a[i].second + b[j].second);
            ++i, ++j;
            continue;
        }
        if(j == (int) b.size() || (i < (int) a.size() && a[i].first < b[j].first)) {
            c.push_back(a[i++]);
        } else {
            c.push_back(b[j++]);
        }
    }
    return c;
}

template<class T>
vector<pair<T, int>> RhoC(const T& n, const T& c) {
    if(n <= 1) {
        return {};
    }
    if(n % 2 == 0) {
        return MergeFactors({{2, 1}}, RhoC(n / 2, c));
    }
    if(is_prime_constexpr(n)) {
        return {{n, 1}};
    }
    T x = 2;
    T saved = 2;
    T p = 1;
    T lam = 1;
    while(true) {
        x = (x * x % n + c) % n;
        T g = __gcd(((x - saved) + n) % n, n);
        if(g != 1) {
            return MergeFactors(RhoC(g, c + 1), RhoC(n / g, c + 1));
        }
        if(p == lam) {
            saved = x;
            p <<= 1;
            lam = 0;
        }
        lam += 1;
    }
    return {};
}

template<class T>
vector<pair<T, int>> Factorize(T n) {
    if(n <= 1) {
        return {};
    }
    return RhoC(n, T(1));
}

template<class T>
vector<T> BuildDivisorsFromFactors(const vector<pair<T, int>>& factors) {
    int total = 1;
    for(int i = 0; i < (int) factors.size(); ++i) {
        total *= factors[i].second + 1;
    }
    vector<T> divisors;
    divisors.reserve(total);
    divisors.push_back(1);
    for(auto& [p, cnt] : factors) {
        int sz = (int) divisors.size();
        for(int i = 0; i < sz; ++i) {
            T cur = divisors[i];
            for(int j = 0; j < cnt; ++j) {
                cur *= p;
                divisors.push_back(cur);
            }
        }
    }
    // sort(divisors.begin(), divisors.end());
    return divisors;
}

```

3.11 PrimitiveRoot.h

```
// Compile time primitive root
// @param m must be prime
// @return primitive root (and minimum in now)
constexpr int primitive_root_constexpr(int m) {
    if(m == 2) return 1;
    if(m == 167772161) return 3;
    if(m == 469762049) return 3;
    if(m == 754974721) return 11;
    if(m == 998244353) return 3;
    int divs[20] = {};
    divs[0] = 2;
    int cnt = 1;
    int x = (m - 1) / 2;
    while(x % 2 == 0) x /= 2;
    for(int i = 3; (long long)(i)*i <= x; i += 2) {
        if(x % i == 0) {
            divs[cnt++] = i;
            while(x % i == 0) {
                x /= i;
            }
        }
    }
    if(x > 1) {
        divs[cnt++] = x;
    }
    for(int g = 2;; g++) {
        bool ok = true;
        for(int i = 0; i < cnt; i++) {
            if(pow_mod_constexpr(g, (m - 1) / divs[i], m) == 1) {
                ok = false;
                break;
            }
        }
        if(ok) return g;
    }
}
template<int m> constexpr int primitive_root = primitive_root_constexpr(m);

```

3.12 FloorSum.h

```
// @param n < 2^{32}
// @param 1 <= m < 2^{32}
// @return sum_{i=0}^{n-1} \lfloor \frac{ai + b}{m} \rfloor \pmod{2^{64}}
unsigned long long floor_sum_unsigned(unsigned long long n, unsigned long long m, unsigned long long a, unsigned long long b) {
    unsigned long long ans = 0;
    while(true) {
        if(a >= m) {
            ans += n * (n - 1) / 2 * (a / m);
            a %= m;
        }
        if(b >= m) {
            ans += n * (b / m);
            b %= m;
        }
        unsigned long long y_max = a * n + b;
        if(y_max < m) {
            break;
        }
        // y_max < m * (n + 1)
        // floor(y_max / m) <= n
        n = (unsigned long long)(y_max / m);
        b = (unsigned long long)(y_max % m);
        swap(m, a);
    }
    return ans;
}

long long floor_sum(long long n, long long m, long long a, long long b) {
    assert(0 <= n && n < (1LL << 32));
    assert(1 <= m && m < (1LL << 32));
    unsigned long long ans = 0;
    if(a < 0) {
        unsigned long long a2 = safe_mod(a, m);
        ans -= 1ULL * n * (n - 1) / 2 * ((a2 - a) / m);
        a = a2;
    }
    if(b < 0) {
        unsigned long long b2 = safe_mod(b, m);
        ans -= 1ULL * n * ((b2 - b) / m);
        b = b2;
    }
    return ans + floor_sum_unsigned(n, m, a, b);
}

```

4 Numerical

4.1 Barrett.h

```
// Fast modular multiplication by barrett reduction
// Reference: https://en.wikipedia.org/wiki/Barrett_reduction
class barrett {
public:
    unsigned int m;
    unsigned long long im;

    explicit barrett(unsigned int _m) : m(_m), im((unsigned long long)(-1) / _m + 1) {}

    unsigned int umod() const { return m; }

    unsigned int mul(unsigned int a, unsigned int b) const {
        unsigned long long z = a;
        z *= b;
#ifdef _MSC_VER
        unsigned long long x;
        _umul128(z, im, &x);
#else
        unsigned long long x = (unsigned long long)(((unsigned __int128)(z) * im) >> 64);
#endif
        unsigned int v = (unsigned int)(z - x * m);
        if(m <= v) {
            v += m;
        }
        return v;
    }
};

```

4.2 BitTransform.h

```
template<class T>
void OrTransform(vector<T>& a) {
    const int n = (int) a.size();
    assert((n & -n) == n);
    for(int i = 1; i < n; i <<= 1) {
        for(int j = 0; j < n; j += i << 1) {
            for(int k = 0; k < i; ++k) {
                a[i + j + k] += a[j + k];
            }
        }
    }
}

template<class T>
void OrInvTransform(vector<T>& a) {
    const int n = (int) a.size();
    assert((n & -n) == n);
    for(int i = 1; i < n; i <<= 1) {
        for(int j = 0; j < n; j += i << 1) {
            for(int k = 0; k < i; ++k) {
                a[i + j + k] -= a[j + k];
            }
        }
    }
}

template<class T>
void AndTransform(vector<T>& a) {
    const int n = (int) a.size();
    assert((n & -n) == n);
    for(int i = 1; i < n; i <<= 1) {
        for(int j = 0; j < n; j += i << 1) {
            for(int k = 0; k < i; ++k) {
                a[j + k] += a[i + j + k];
            }
        }
    }
}

template<class T>
void AndInvTransform(vector<T>& a) {
    const int n = (int) a.size();
    assert((n & -n) == n);
    for(int i = 1; i < n; i <<= 1) {
        for(int j = 0; j < n; j += i << 1) {
            for(int k = 0; k < i; ++k) {
                a[j + k] -= a[i + j + k];
            }
        }
    }
}

template<class T>
void XorTransform(vector<T>& a) {
    const int n = (int) a.size();
    assert((n & -n) == n);
    for(int i = 1; i < n; i <<= 1) {
        for(int j = 0; j < n; j += i << 1) {
            for(int k = 0; k < i; ++k) {
                T x = move(a[j + k]), y = move(a[i + j + k]);
                a[j + k] = x + y;
                a[i + j + k] = x - y;
            }
        }
    }
}

template<class T>
void XorInvTransform(vector<T>& a) {
    XorTransform(a);
    T inv2 = T(1) / T((int) a.size());
    for(auto& x : a) {
        x *= inv2;
    }
}

// Compute c[k] = sum(a[i] * b[j]) for (i or j) = k.
// Complexity: O(n log n)
```

```
template<class T>
vector<T> OrConvolution(vector<T> a, vector<T> b) {
    const int n = (int) a.size();
    assert(n == int(b.size()));
    OrTransform(a);
    OrTransform(b);
    for(int i = 0; i < n; ++i) {
        a[i] *= b[i];
    }
    OrInvTransform(a);
    return a;
}

// Compute c[k] = sum(a[i] * b[j]) for (i and j) = k.
// Complexity: O(n log n)
template<class T>
vector<T> AndConvolution(vector<T> a, vector<T> b) {
    const int n = (int) a.size();
    assert(n == int(b.size()));
    AndTransform(a);
    AndTransform(b);
    for(int i = 0; i < n; ++i) {
        a[i] *= b[i];
    }
    AndInvTransform(a);
    return a;
}

// Compute c[k] = sum(a[i] * b[j]) for (i xor j) = k.
// Complexity: O(n log n)
template<class T>
vector<T> XorConvolution(vector<T> a, vector<T> b) {
    const int n = (int) a.size();
    assert(n == int(b.size()));
    XorTransform(a);
    XorTransform(b);
    for (int i = 0; i < n; ++i) {
        a[i] *= b[i];
    }
    XorInvTransform(a);
    return a;
}

template<class T>
void ZetaTransform(vector<T>& a) {
    OrTransform(a);
}

template<class T>
void MobiusTransform(vector<T>& a) {
    OrInvTransform(a);
}

template<class T>
vector<T> SubsetSumConvolution(const vector<T>& f, const vector<T>& g) {
    const int n = (int) f.size();
    assert(n == int(g.size()));
    assert((n & -n) == n);
    const int N = __lg(n);
    vector<vector<T>> fhat(N + 1, vector<T>(n));
    vector<vector<T>> ghat(N + 1, vector<T>(n));
    for(int mask = 0; mask < n; ++mask) {
        fhat[__builtin_popcount(mask)][mask] = f[mask];
        ghat[__builtin_popcount(mask)][mask] = g[mask];
    }
    for(int i = 0; i <= N; ++i) {
        ZetaTransform(fhat[i]);
        ZetaTransform(ghat[i]);
    }
    vector<vector<T>> h(N + 1, vector<T>(n));
    for(int mask = 0; mask < n; ++mask) {
        for(int i = 0; i <= N; ++i) {
            for(int j = 0; j <= i; ++j) {
                h[i][mask] += fhat[j][mask] * ghat[i - j][mask];
            }
        }
    }
    for(int i = 0; i <= N; ++i) {
        MobiusTransform(h[i]);
    }
    vector<T> result(n);
    for(int mask = 0; mask < n; ++mask) {
        result[mask] = h[__builtin_popcount(mask)][mask];
    }
    return result;
}

```

4.3 FFT.h

```
// Fast-Fourier-Transform
using cd = complex<double>;

const double PI = acos(-1);

void fft(vector<cd>& a, bool inv) {
    int n = (int) a.size();
    for(int i = 1, j = 0; i < n; ++i) {
        int bit = n >> 1;
        for(; j & bit; bit >>= 1) {
            j ^= bit;
        }
        j ^= bit;
        if(i < j) {
            swap(a[i], a[j]);
        }
    }
    for(int len = 2; len <= n; len <<= 1) {
        const double ang = 2 * PI / len * (inv ? -1 : +1);
        cd rot(cos(ang), sin(ang));
        for(int i = 0; i < n; i += len) {
            cd w(1);
            for(int j = 0; j < len / 2; ++j) {
                cd u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= rot;
            }
        }
    }
    if(inv) {
        for(auto& x : a) {
            x /= n;
        }
    }
}

```

4.4 Poly.h

```
vector<int> __bit_reorder;

template<class T>
class Poly {
public:
    static constexpr int R = primitive_root<T::mod()>;

    Poly() {}

    Poly(int n) : coeff(n) {}

    Poly(const vector<T>& a) : coeff(a) {}

    Poly(const initializer_list<T>& a) : coeff(a) {}

    static constexpr int mod() {
        return (int) T::mod();
    }

    inline int size() const {
        return (int) coeff.size();
    }

    void resize(int n) {
        coeff.resize(n);
    }

    T at(int idx) const {
        if(idx < 0 || idx >= size()) {
            return 0;
        }
        return coeff[idx];
    }

    T& operator[](int idx) {
        return coeff[idx];
    }

    Poly mulxk(int k) const {
        auto b = coeff;
        b.insert(b.begin(), k, T(0));
        return Poly(b);
    }

    Poly modxk(int k) const {
        k = min(k, size());
        return Poly(vector<T>(coeff.begin(), coeff.begin() + k));
    }

    Poly divxk(int k) const {
        if(size() <= k) {
            return Poly<T>();
        }
        return Poly(vector<T>(coeff.begin() + k, coeff.end()));
    }

    friend Poly operator+(const Poly& a, const Poly& b) {
        vector<T> res(max(a.size(), b.size()));
        for(int i = 0; i < (int) res.size(); ++i) {
            res[i] = a.at(i) + b.at(i);
        }
        return Poly(res);
    }

    friend Poly operator-(const Poly& a, const Poly& b) {
        vector<T> res(max(a.size(), b.size()));
        for(int i = 0; i < (int) res.size(); ++i) {
            res[i] = a.at(i) - b.at(i);
        }
        return Poly(res);
    }

    static void ensure_base(int n) {
        if((int) __bit_reorder.size() != n) {
            int k = __builtin_ctz(n) - 1;
            __bit_reorder.resize(n);
            for(int i = 0; i < n; ++i) {
                __bit_reorder[i] = __bit_reorder[i >> 1] >> 1 | (i & 1) << k;
            }
        }
        if((int) roots.size() < n) {
            int k = __builtin_ctz(roots.size());
            roots.resize(n);
            while((1 << k) < n) {
                T e = pow_mod_constexpr(R, (T::mod() - 1) >> (k + 1), T::mod());
                for(int i = 1 << (k - 1); i < (1 << k); ++i) {
                    roots[2 * i] = roots[i];
                    roots[2 * i + 1] = roots[i] * e;
                }
                k += 1;
            }
        }
    }

    static void dft(vector<T>& a) {
        const int n = (int) a.size();
        assert((n & -n) == n);
        ensure_base(n);
        for(int i = 0; i < n; ++i) {
            if(__bit_reorder[i] < i) {
                swap(a[i], a[__bit_reorder[i]]);
            }
        }
        for(int k = 1; k < n; k *= 2) {
            for(int i = 0; i < n; i += 2 * k) {
```

```cpp
                for(int j = 0; j < k; ++j) {
                    T u = a[i + j];
                    T v = a[i + j + k] * roots[k + j];
                    a[i + j] = u + v;
                    a[i + j + k] = u - v;
                }
            }
        }
    }

    static void idft(vector<T>& a) {
        const int n = (int) a.size();
        reverse(a.begin() + 1, a.end());
        dft(a);
        T inv = (1 - T::mod()) / n;
        for(int i = 0; i < n; ++i) {
            a[i] *= inv;
        }
    }

    friend Poly operator*(Poly a, Poly b) {
        if(a.size() == 0 || b.size() == 0) {
            return Poly();
        }
        if(min(a.size(), b.size()) < 250) {
            vector<T> 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 Poly(c);
        }
        int tot = a.size() + b.size() - 1;
        int sz = 1;
        while(sz < tot) {
            sz <<= 1;
        }
        a.coeff.resize(sz);
        b.coeff.resize(sz);
        dft(a.coeff);
        dft(b.coeff);
        for(int i = 0; i < sz; ++i) {
            a.coeff[i] = a[i] * b[i];
        }
        idft(a.coeff);
        a.resize(tot);
        return a;
    }

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

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

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

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

    Poly& operator*=(Poly b) {
        return *this = *this * b;
    }

    Poly deriv() const {
        if(coeff.empty()) {
            return Poly<T>();
        }
        vector<T> res(size() - 1);
        for(int i = 0; i < size() - 1; ++i) {
            res[i] = (i + 1) * coeff[i + 1];
        }
        return Poly(res);
    }

    Poly integr() const {
        vector<T> res(size() + 1);
        for(int i = 0; i < size(); ++i) {
            res[i + 1] = coeff[i] / T(i + 1);
        }
        return Poly(res);
    }

    Poly inv(int m) const {
        Poly x{T(1) / coeff[0]};
        int k = 1;
        while(k < m) {
            k *= 2;
            x = (x * (Poly{T(2)} - modxk(k) * x)).modxk(k);
        }
        return x.modxk(m);
    }

    Poly log(int m) const {
        return (deriv() * inv(m)).integr().modxk(m);
    }

    Poly exp(int m) const {
        Poly x{T(1)};
        int k = 1;
        while(k < m) {
            k *= 2;
            x = (x * (Poly{T(1)} - x.log(k) + modxk(k))).modxk(k);
        }
        return x.modxk(m);
    }

    Poly pow(int k, int m) const {
        if(k == 0) {
            vector<T> a(m);
            a[0] = 1;
            return Poly(a);
        }
        int i = 0;
        while(i < size() && coeff[i]() == 0) {
            i++;
        }
        if(i == size() || 1LL * i * k >= m) {
            return Poly(vector<T>(m));
        }
        T v = coeff[i];
        auto f = divxk(i) * (1 / v);
        return (f.log(m - i * k) * T(k)).exp(m - i * k).mulxk(i * k) * power(v, k);
    }

    Poly sqrt(int m) const {
        Poly<T> x{1};
        int k = 1;
        while(k < m) {
            k *= 2;
            x = (x + (modxk(k) * x.inv(k)).modxk(k)) * T((mod() + 1) / 2);
        }
        return x.modxk(m);
    }

    Poly mulT(Poly b) const {
        if(b.size() == 0) {
            return Poly<T>();
        }
        int n = b.size();
        reverse(b.coeff.begin(), b.coeff.end());
        return ((*this) * b).divxk(n - 1);
    }

    vector<T> eval(vector<T> x) const {
        if(size() == 0) {
            return vector<T>(x.size(), 0);
        }
        const int n = max((int) x.size(), size());
        vector<Poly<T>> q(4 * n);
        vector<T> ans(x.size());
        x.resize(n);
        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);
        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;
                work(2 * p, l, m, num.mulT(q[2 * p + 1]).modxk(m - l));
                work(2 * p + 1, m, r, num.mulT(q[2 * p]).modxk(r - m));
            }
        };
        work(1, 0, n, mulT(q[1].inv(n)));
        return ans;
    }

private:
    vector<T> coeff;
    static vector<T> roots;
};

template<class T> vector<T> Poly<T>::roots{0, 1};

```

5 Geometry

5.1 Point.h

```
template<class T>
class Point {
public:
    T x, y;

    Point() : x(0), y(0) {}

    Point(const T& a, const T& b) : x(a), y(b) {}

    template<class U>
    explicit Point(const Point<U>& p) : x(static_cast<T>(p.x)), y(static_cast<T>(p.y)) {}

    Point(const pair<T, T>& p) : x(p.first), y(p.second) {}

    Point(const complex<T>& p) : x(real(p)), y(imag(p)) {}

    explicit operator pair<T, T>() const {
        return pair<T, T>(x, y);
    }

    explicit operator complex<T>() const {
        return complex<T>(x, y);
    }

    inline Point& operator+=(const Point& rhs) {
        x += rhs.x;
        y += rhs.y;
        return *this;
    }

    inline Point& operator-=(const Point& rhs) {
        x -= rhs.x;
        y -= rhs.y;
        return *this;
    }

    inline Point& operator*=(const T& rhs) {
        x *= rhs;
        y *= rhs;
        return *this;
    }

    inline Point& operator/=(const T& rhs) {
        x /= rhs;
        y /= rhs;
        return *this;
    }

    template<class U>
    inline Point& operator+=(const Point<U>& rhs) {
        return *this += Point<T>(rhs);
    }

    template<class U>
    inline Point& operator-=(const Point<U>& rhs) {
        return *this -= Point<T>(rhs);
    }

    inline Point operator+() const {
        return *this;
    }

    inline Point operator-() const {
        return Point(-x, -y);
    }

    inline Point operator+(const Point& rhs) {
        return Point(*this) += rhs;
    }

    inline Point operator-(const Point& rhs) {
        return Point(*this) -= rhs;
    }

    inline Point operator*(const T& rhs) {
        return Point(*this) *= rhs;
    }

    inline Point operator/(const T& rhs) {
        return Point(*this) /= rhs;
    }

    inline bool operator==(const Point& rhs) {
        return x == rhs.x && y == rhs.y;
    }

    inline bool operator!=(const Point& rhs) {
        return !(*this == rhs);
    }

    inline T dist2() const {
        return x * x + y * y;
    }

    inline long double dist() const {
        return sqrt(dist2());
    }

    inline Point unit() const {
        return *this / this->dist();
    }

    inline long double angle() const {
        return atan2(y, x);
    }

    inline friend T dot(const Point& lhs, const Point& rhs) {
        return lhs.x * rhs.x + lhs.y * rhs.y;
    }

    inline friend T cross(const Point& lhs, const Point& rhs) {
        return lhs.x * rhs.y - lhs.y * rhs.x;
    }

    inline friend Point dot_cross(const Point& lhs, const Point& rhs) {
        return Point(dot(lhs, rhs), cross(lhs, rhs));
    }
};

template<class T>
istream& operator>>(istream& in, Point<T>& p) {
    return in >> p.x >> p.y;
}

```

5.2 ConvexHull.h

```
// @return the points of the convex hull in clock-wise order
template<class T>
vector<Point<T>> ConvexHull(vector<Point<T>> points) {
    const int n = (int) points.size();
    sort(points.begin(), points.end(), [](const Point<T>& a, const Point<T>& b) {
        if(a.x == b.x) {
            return a.y < b.y;
        }
        return a.x < b.x;
    });
    auto build = [&]() {
        vector<Point<T>> upper;
        upper.push_back(points[0]);
        upper.push_back(points[1]);
        for(int i = 2; i < n; ++i) {
            while((int) upper.size() >= 2) {
                if(cross(upper.end()[-1] - upper.end()[-2], points[i] - upper.end()[-1]) > 0) {
                    upper.pop_back();
                } else {
                    break;
                }
            }
            upper.push_back(points[i]);
        }
        return upper;
    };
    vector<Point<T>> upper = build();
    reverse(points.begin(), points.end());
    vector<Point<T>> lower = build();
    lower.pop_back();
    upper.insert(upper.end(), lower.begin() + 1, lower.end());
    return upper;
}

```

6 Graph

6.1 LCA.h

```
class LCA {
public:
    LCA() : LCA(0) {}
    LCA(int _n) : n(_n), g(_n) {}

    static pair<int, int> __lca_op(pair<int, int> a, pair<int, int> b) {
        return min(a, b);
    }

    void add_edge(int u, int v) {
        assert(0 <= u && u < n);
        assert(0 <= v && v < n);
        g[u].push_back(v);
        g[v].push_back(u);
    }

    void build(int root = 0) {
        assert(0 <= root && root < n);
        depth.assign(n, 0);
        parent.assign(n, -1);
        subtree_size.assign(n, 1);
        euler.reserve(2 * n - 1);
        first_occurrence.assign(n, 0);
        tour_list.reserve(n);
        tour_start.assign(n, 0);
        function<void(int, int, int)> dfs = [&](int u, int p, int d) {
            parent[u] = p;
            depth[u] = d;
            first_occurrence[u] = (int) euler.size();
            euler.push_back(u);
            pair<int, int> heavy = {-1, -1};
            for(auto& v : g[u]) {
                if(v == p) {
                    continue;
                }
                dfs(v, u, d + 1);
                subtree_size[u] += subtree_size[v];
                if(subtree_size[v] > heavy.first) {
                    heavy = {subtree_size[v], v};
                }
                euler.push_back(u);
            }
            sort(g[u].begin(), g[u].end(), [&](int a, int b) {
                return subtree_size[a] > subtree_size[b];
            });
        };
        dfs(root, -1, 0);
        heavy_root.assign(n, 0);
        function<void(int, bool)> dfs2 = [&](int u, bool is_heavy) {
            tour_start[u] = (int) tour_list.size();
            tour_list.push_back(u);
            heavy_root[u] = (is_heavy ? heavy_root[parent[u]] : u);
            bool heavy = true;
            for(auto& v : g[u]) {
                if(v == parent[u]) {
                    continue;
                }
                dfs2(v, heavy);
                heavy = false;
            }
        };
        dfs2(root, false);
        {
            vector<pair<int, int>> route;
            route.reserve((int) euler.size());
            for(auto& u : euler) {
                route.emplace_back(depth[u], u);
            }
            st = sparse_table<pair<int, int>, __lca_op>(route);
        }
    }

    inline int dist(int u, int v) const {
        return depth[u] + depth[v] - 2 * depth[lca(u, v)];
    }

    pair<int, array<int, 2>> get_diameter() const {
        pair<int, int> u_max = {-1, -1};
        pair<int, int> ux_max = {-1, -1};
        pair<int, array<int, 2>> uxv_max = {-1, {-1, -1}};
        for(int u : euler) {
            u_max = max(u_max, {depth[u], u});
            ux_max = max(ux_max, {u_max.first - 2 * depth[u], u_max.second});
            uxv_max = max(uxv_max, {ux_max.first + depth[u], {ux_max.second, u}});
        }
        return uxv_max;
    }

    inline int kth_ancestor(int u, int k) const {
        if(depth[u] < k) {
            return -1;
        }
        while(k > 0) {
            int root = heavy_root[u];
            if(depth[root] <= depth[u] - k) {
                return tour_list[tour_start[u] - k];
            }
            k -= depth[u] - depth[root] + 1;
            u = parent[root];
        }
        return u;
    }

    inline int kth_node_on_path(int a, int b, int k) const {
        int z = lca(a, b);
        int fi = depth[a] - depth[z];
        int se = depth[b] - depth[z];
        assert(0 <= k && k <= fi + se);
        if(k < fi) {
            return kth_ancestor(a, k);
        } else {
            return kth_ancestor(b, fi + se - k);
        }
    }

    int lca(int u, int v) const {
        assert(0 <= u && u < n);
        assert(0 <= v && v < n);
        int l = first_occurrence[u];
        int r = first_occurrence[v];
        return st.prod(min(l, r), max(l, r)).second;
    }

public:
    int n;
    vector<vector<int>> g;
    vector<int> parent;
    vector<int> depth;
    vector<int> subtree_size;

protected:
    vector<int> euler;
    vector<int> first_occurrence;
    vector<int> tour_list;
    vector<int> tour_start;
    vector<int> heavy_root;
    sparse_table<pair<int, int>, __lca_op> st;
};

```

6.2 HLD.h

```
class HLD : LCA {
public:
    using LCA::add_edge;
    using LCA::build;
    using LCA::dist;
    using LCA::get_diameter;
    using LCA::kth_ancestor;
    using LCA::kth_node_on_path;
    using LCA::lca;

    HLD() : HLD(0) {}
    HLD(int _n) : LCA(_n) {}

    inline int get(int u) const {
        return tour_start[u];
    }

    // return $path_{[u,\dots,p)}$ where $p$ is an ancestor of $u$
    vector<pair<int, int>> path_up(int u, int p) const {
        vector<pair<int, int>> seg;
        while(heavy_root[u] != heavy_root[p]) {
            seg.emplace_back(get(heavy_root[u]), get(u) + 1);
            u = parent[heavy_root[u]];
        }
        // $id_p$ is smaller than $id_u$ but we don't want $id_p$
        seg.emplace_back(get(p) + 1, get(u) + 1);
        return seg;
    }

    vector<pair<int, int>> path(int u, int v) const {
        int z = lca(u, v);
        auto lhs = path_up(u, z);
        auto rhs = path_up(v, z);
        lhs.emplace_back(get(z), get(z) + 1);
        lhs.insert(lhs.end(), rhs.begin(), rhs.end());
        return lhs;
    }
};

```

6.3 TwoSat.h

```
// Under construction
```

6.4 Dinic.h

```
template<class T>
class Dinic {
public:
    struct Edge {
        int to;
        T cap;
        Edge(int _to, T _cap) : to(_to), cap(_cap) {}
    };

    static constexpr T INF = numeric_limits<T>::max() / 2;

    int n;
    vector<Edge> e;
    vector<vector<int>> g;
    vector<int> cur, h;

    Dinic() {}
    Dinic(int _n) : n(_n), g(_n) {}

    void add_edge(int u, int v, T c) {
        assert(0 <= u && u < n);
        assert(0 <= v && v < n);
        g[u].push_back(e.size());
        e.emplace_back(v, c);
        g[v].push_back(e.size());
        e.emplace_back(u, 0);
    }

    bool bfs(int s, int t) {
        h.assign(n, -1);
        queue<int> que;
        h[s] = 0;
        que.push(s);
        while(!que.empty()) {
            int u = que.front();
            que.pop();
            for(int i : g[u]) {
                int v = e[i].to;
                T c = e[i].cap;
                if(c > 0 && h[v] == -1) {
                    h[v] = h[u] + 1;
                    if(v == t) {
                        return true;
                    }
                    que.push(v);
                }
            }
        }
        return false;
    }

    T dfs(int u, int t, T f) {
        if(u == t) {
            return f;
        }
        T r = f;
        for(int &i = cur[u]; i < int(g[u].size()); ++i) {
            int j = g[u][i];
            int v = e[j].to;
            T c = e[j].cap;
            if(c > 0 && h[v] == h[u] + 1) {
                T a = dfs(v, t, min(r, c));
                e[j].cap -= a;
                e[j ^ 1].cap += a;
                r -= a;
                if (r == 0) {
                    return f;
                }
            }
        }
        return f - r;
    }

    T flow(int s, int t) {
        assert(0 <= s && s < n);
        assert(0 <= t && t < n);
        T ans = 0;
        while(bfs(s, t)) {
            cur.assign(n, 0);
            ans += dfs(s, t, INF);
        }
        return ans;
    }
};

```

6.5 MCMF.h

```
template<class Cap_t, class Cost_t>
class MCMF {
public:
    struct Edge {
        int from;
        int to;
        Cap_t cap;
        Cost_t cost;
        Edge(int u, int v, Cap_t _cap, Cost_t _cost) : from(u), to(v), cap(_cap), cost(_cost) {}
    };

    static constexpr Cap_t EPS = static_cast<Cap_t>(1e-9);

    int n;
    vector<Edge> edges;
    vector<vector<int>> g;
    vector<Cost_t> d;
    vector<bool> in_queue;
    vector<int> previous_edge;

    MCMF(int _n) : n(_n), g(_n), d(_n), in_queue(_n), previous_edge(_n) {}

    void add_edge(int u, int v, Cap_t cap, Cost_t cost) {
        assert(0 <= u && u < n);
        assert(0 <= v && v < n);
        g[u].push_back(edges.size());
        edges.emplace_back(u, v, cap, cost);
        g[v].push_back(edges.size());
        edges.emplace_back(v, u, 0, -cost);
    }

    bool bfs(int s, int t) {
        bool found = false;
        fill(d.begin(), d.end(), numeric_limits<Cost_t>::max());
        d[s] = 0;
        in_queue[s] = true;
        queue<int> que;
        que.push(s);
        while(!que.empty()) {
            int u = que.front();
            que.pop();
            if(u == t) {
                found = true;
            }
            in_queue[u] = false;
            for(auto& id : g[u]) {
                const Edge& e = edges[id];
                if(e.cap > EPS && d[u] + e.cost < d[e.to]) {
                    d[e.to] = d[u] + e.cost;
                    previous_edge[e.to] = id;
                    if(!in_queue[e.to]) {
                        que.push(e.to);
                        in_queue[e.to] = true;
                    }
                }
            }
        }
        return found;
    }

    pair<Cap_t, Cost_t> flow(int s, int t) {
        assert(0 <= s && s < n);
        assert(0 <= t && t < n);
        Cap_t cap = 0;
        Cost_t cost = 0;
        while(bfs(s, t)) {
            Cap_t send = numeric_limits<Cap_t>::max();
            int u = t;
            while(u != s) {
                const Edge& e = edges[previous_edge[u]];
                send = min(send, e.cap);
                u = e.from;
            }
            u = t;
            while(u != s) {
                Edge& e = edges[previous_edge[u]];
                e.cap -= send;
                Edge& b = edges[previous_edge[u] ^ 1];
                b.cap += send;
                u = e.from;
            }
            cap += send;
            cost += send * d[t];
        }
        return make_pair(cap, cost);
    }
};

```

7 String

7.1 SuffixArray.h

```
vector<int> sa_naive(const vector<int>& s) {
    int n = int(s.size());
    vector<int> sa(n);
    iota(sa.begin(), sa.end(), 0);
    sort(sa.begin(), sa.end(), [&](int l, int r) {
        if(l == r) {
            return false;
        }
        while(l < n && r < n) {
            if(s[l] != s[r]) {
                return s[l] < s[r];
            }
            l++;
            r++;
        }
        return l == n;
    });
    return sa;
}

vector<int> sa_doubling(const vector<int>& s) {
    int n = int(s.size());
    vector<int> sa(n), rnk = s, tmp(n);
    iota(sa.begin(), sa.end(), 0);
    for(int k = 1; k < n; k *= 2) {
        auto cmp = [&](int x, int y) {
            if(rnk[x] != rnk[y]) return rnk[x] < rnk[y];
            int rx = x + k < n ? rnk[x + k] : -1;
            int ry = y + k < n ? rnk[y + k] : -1;
            return rx < ry;
        };
        sort(sa.begin(), sa.end(), cmp);
        tmp[sa[0]] = 0;
        for(int i = 1; i < n; i++) {
            tmp[sa[i]] = tmp[sa[i - 1]] + (cmp(sa[i - 1], sa[i]) ? 1 : 0);
        }
        swap(tmp, rnk);
    }
    return sa;
}

// SA-IS, linear-time suffix array construction
// Reference:
// G. Nong, S. Zhang, and W. H. Chan,
// Two Efficient Algorithms forLinear Time Suffix Array Construction
template<int THRESHOLD_NAIVE = 10, int THRESHOLD_DOUBLING = 40>
vector<int> sa_is(const vector<int>& s, int upper) {
    int n = int(s.size());
    if(n == 0) {
        return {};
    }
    if(n == 1) {
        return {0};
    }
    if(n == 2) {
        if(s[0] < s[1]) {
            return {0, 1};
        } else {
            return {1, 0};
        }
    }
    if(n < THRESHOLD_NAIVE) {
        return sa_naive(s);
    }
    if(n < THRESHOLD_DOUBLING) {
        return sa_doubling(s);
    }
    vector<int> sa(n);
    vector<bool> ls(n);
    for(int i = n - 2; i >= 0; i--) {
        ls[i] = (s[i] == s[i + 1]) ? ls[i + 1] : (s[i] < s[i + 1]);
    }
    vector<int> sum_l(upper + 1), sum_s(upper + 1);
    for(int i = 0; i < n; i++) {
        if(!ls[i]) {
            sum_s[s[i]]++;
        } else {
            sum_l[s[i] + 1]++;
        }
    }
    for(int i = 0; i <= upper; i++) {
        sum_s[i] += sum_l[i];
        if(i < upper) {
            sum_l[i + 1] += sum_s[i];
        }
    }

    auto induce = [&](const vector<int>& lms) {
        fill(sa.begin(), sa.end(), -1);
        vector<int> buf(upper + 1);
        copy(sum_s.begin(), sum_s.end(), buf.begin());
        for(auto d : lms) {
            if(d == n) {
                continue;
            }
            sa[buf[s[d]]++] = d;
        }
        copy(sum_l.begin(), sum_l.end(), buf.begin());
        sa[buf[s[n - 1]]++] = n - 1;
        for(int i = 0; i < n; i++) {
            int v = sa[i];
            if(v >= 1 && !ls[v - 1]) {
                sa[buf[s[v - 1]]++] = v - 1;
            }
        }
        copy(sum_l.begin(), sum_l.end(), buf.begin());
        for(int i = n - 1; i >= 0; i--) {
            int v = sa[i];
            if(v >= 1 && ls[v - 1]) {
                sa[--buf[s[v - 1] + 1]] = v - 1;
            }
        }
    };

    vector<int> lms_map(n + 1, -1);
    int m = 0;
    for(int i = 1; i < n; i++) {
        if(!ls[i - 1] && ls[i]) {
            lms_map[i] = m++;
        }
    }
    vector<int> lms;
    lms.reserve(m);
    for(int i = 1; i < n; i++) {
        if(!ls[i - 1] && ls[i]) {
            lms.push_back(i);
        }
    }

    induce(lms);

    if(m) {
        vector<int> sorted_lms;
        sorted_lms.reserve(m);
        for(int v : sa) {
            if(lms_map[v] != -1) {
                sorted_lms.push_back(v);
            }
        }
        vector<int> rec_s(m);
        int rec_upper = 0;
        rec_s[lms_map[sorted_lms[0]]] = 0;
        for(int i = 1; i < m; i++) {
            int l = sorted_lms[i - 1], r = sorted_lms[i];
            int end_l = (lms_map[l] + 1 < m) ? lms[lms_map[l] + 1] : n;
            int end_r = (lms_map[r] + 1 < m) ? lms[lms_map[r] + 1] : n;
            bool same = true;
            if(end_l - l != end_r - r) {
                same = false;
            } else {
                while(l < end_l) {
                    if(s[l] != s[r]) {
                        break;
                    }
                    l++;
                    r++;
                }
                if(l == n || s[l] != s[r]) {
                    same = false;
                }
            }
            if(!same) {
                rec_upper++;
            }
            rec_s[lms_map[sorted_lms[i]]] = rec_upper;
        }

        auto rec_sa = sa_is<THRESHOLD_NAIVE, THRESHOLD_DOUBLING>(rec_s, rec_upper);

        for(int i = 0; i < m; i++) {
            sorted_lms[i] = lms[rec_sa[i]];
        }
        induce(sorted_lms);
    }
    return sa;
}

vector<int> suffix_array(const vector<int>& s, int upper) {
    assert(0 <= upper);
    for(int d : s) {
        assert(0 <= d && d <= upper);
    }
    auto sa = sa_is(s, upper);
    return sa;
}

template<class T>
vector<int> suffix_array(const vector<T>& s) {
    int n = int(s.size());
    vector<int> idx(n);
    iota(idx.begin(), idx.end(), 0);
    sort(idx.begin(), idx.end(), [&](int l, int r) { return s[l] < s[r]; });
    vector<int> s2(n);
    int now = 0;
    for(int i = 0; i < n; i++) {
        if(i && s[idx[i - 1]] != s[idx[i]]) {
            now++;
        }
        s2[idx[i]] = now;
    }
    return sa_is(s2, now);
}

vector<int> suffix_array(const string& s) {
    int n = int(s.size());
    vector<int> s2(n);
    for(int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return sa_is(s2, 255);
}

```

7.2 LCP.h

```
// Reference:
// T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park,
// Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its
// Applications
template<class T>
vector<int> lcp_array(const vector<T>& s, const vector<int>& sa) {
    int n = int(s.size());
    assert(n >= 1);
    vector<int> rnk(n);
    for(int i = 0; i < n; i++) {
        rnk[sa[i]] = i;
    }
    vector<int> lcp(n - 1);
    int h = 0;
    for(int i = 0; i < n; i++) {
        if(h > 0) {
            h--;
        }
        if(rnk[i] == 0) {
            continue;
        }
        int j = sa[rnk[i] - 1];
        for(; j + h < n && i + h < n; h++) {
            if(s[j + h] != s[i + h]) {
                break;
            }
        }
        lcp[rnk[i] - 1] = h;
    }
    return lcp;
}

vector<int> lcp_array(const string& s, const vector<int>& sa) {
    int n = int(s.size());
    vector<int> s2(n);
    for(int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return lcp_array(s2, sa);
}

```

7.3 KMP.h

```
template<class T>
vector<int> KMP(const vector<T>& a) {
    int n = (int) a.size();
    vector<int> k(n);
    for(int i = 1; i < n; ++i) {
        int j = k[i - 1];
        while(j > 0 && a[i] != a[j]) {
            j = k[j - 1];
        }
        if(a[i] == a[j]) {
            j += 1;
        }
        k[i] = j;
    }
    return k;
}

vector<int> KMP(const string& s) {
    vector<int> s2(s.begin(), s.end());
    return KMP(s2);
}

```

7.4 DynamicKMP.h

```
template<int ALPHABET, int (*f)(char)>
class DynamicKMP {
public:
    DynamicKMP() {}

    DynamicKMP(const string& s) {
        reserve(s.size());
        for(const char& c : s) {
            push(c);
        }
    }

    void push(char c) {
        int v = f(c);
        dp.emplace_back();
        dp.back()[v] = (int) dp.size();
        if(p.empty()) {
            p.push_back(0);
            return;
        }
        int i = (int) p.size();
        for(int j = 0; j < ALPHABET; ++j) {
            if(j == v) {
                p.push_back(dp[p[i - 1]][j]);
            } else {
                dp.back()[j] = dp[p[i - 1]][j];
            }
        }
    }

    void pop() {
        p.pop_back();
        dp.pop_back();
    }

    int query() const {
        return p.back();
    }

    vector<int> query_all() const {
        return p;
    }

    void reserve(int sz) {
        p.reserve(sz);
        dp.reserve(sz);
    }

private:
    vector<int> p;
    vector<array<int, ALPHABET>> dp;
};
```

7.5 Zfunc.h

```
template<class T>
vector<int> z_algorithm(const vector<T>& a) {
    int n = (int) a.size();
    vector<int> z(n);
    for(int i = 1, j = 0; i < n; ++i) {
        if(i <= j + z[j]) {
            z[i] = min(z[i - j], j + z[j] - i);
        }
        while(i + z[i] < n && a[i + z[i]] == a[z[i]]) {
            z[i] += 1;
        }
        if(i + z[i] > j + z[j]) {
            j = i;
        }
    }
    return z;
}

vector<int> z_algorithm(const string& s) {
    vector<int> s2(s.begin(), s.end());
    return z_algorithm(s2);
}

```

7.6 RollingHash.h

```
// @param m `1 <= m`
// @return x mod m
constexpr long long safe_mod(long long x, long long m) {
    x %= m;
    if(x < 0) {
        x += m;
    }
    return x;
}

// @param n `0 <= n`
// @param m `1 <= m`
// @return `(x ** n) % m`
constexpr long long pow_mod_constexpr(long long x, long long n, int m) {
    if(m == 1) return 0;
    unsigned int _m = (unsigned int)(m);
    unsigned long long r = 1;
    unsigned long long y = safe_mod(x, m);
    while(n) {
        if(n & 1) r = (r * y) % _m;
        y = (y * y) % _m;
        n >>= 1;
    }
    return r;
}

template<class T>
class Rolling_Hash {
public:
    Rolling_Hash() {}

    Rolling_Hash(int _A, string _s): A(_A), n((int) _s.size()), s(_s), pref(n) {
        pref[0] = s[0];
        for(int i = 1; i < n; ++i) {
            pref[i] = pref[i - 1] * A + s[i];
        }
    }

    inline int size() const {
        return n;
    }

    inline T get(int l, int r) const {
        assert(0 <= l && l <= r && r < n);
        if(l == 0) {
            return pref[r];
        }
        return pref[r] - pref[l - 1] * pow_mod_constexpr(A, r - l + 1, T::mod());
    }

    inline T id() const {
        return pref.back();
    }

private:
    int A;
    int n;
    string s;
    vector<T> pref;
};

```

7.7 Manacher.h

```
template<class T>
vector<int> manacher_odd(const vector<T>& a) {
    vector<T> b(1, -87);
    b.insert(b.end(), a.begin(), a.end());
    b.push_back(-69);
    int n = (int) b.size();
    vector<int> z(n);
    z[0] = 1;
    for(int i = 1, l = -1, r = 1; i <= n; ++i) {
        if(i < r) {
            z[i] = min(z[l + r - i], r - i);
        }
        while(b[i - z[i]] == b[i + z[i]]) {
            z[i] += 1;
        }
        if(i + z[i] - 1 > r) {
            l = i - z[i] + 1;
            r = i + z[i] - 1;
        }
    }
    return vector<int>(z.begin() + 1, z.end() - 1);
}

template<class T>
vector<int> manacher(const vector<T>& a) {
    int n = (int) a.size();
    vector<int> idx(n);
    iota(idx.begin(), idx.end(), 0);
    sort(idx.begin(), idx.end(), [&](int l, int r) { return s[l] < s[r]; });
    vector<int> b(n);
    int now = 0;
    for(int i = 0; i < n; i++) {
        if(i && s[idx[i - 1]] != s[idx[i]]) {
            now++;
        }
        b[idx[i]] = now;
    }
    vector<int> s2;
    s2.reserve((int) b.size() * 2);
    for(auto& x : b) {
        s2.push_back(x);
        s2.push_back(-1);
    }
    s2.pop_back();
    return manacher_odd(s2);
}

vector<int> manacher(const string& s) {
    vector<int> s2;
    s2.reserve((int) s.size() * 2);
    for(const auto& c : s) {
        s2.push_back(c);
        s2.push_back(-1);
    }
    s2.pop_back();
    return manacher_odd(s2);
}
```

7.8 Trie.h

```
template<int ALPHABET, int (*f)(char)>
class Trie {
public:
    struct Node {
        int answer = 0;
        int next[ALPHABET];

        Node() {
            memset(next, -1, sizeof(next));
        }
    };

    Trie() : Trie(vector<string>()) {}

    Trie(const vector<string>& strs) {
        clear();
        for(const string& s : strs) {
            insert(s);
        }
    }

    void insert(const string& s, int p = 0) {
        for(const char& c : s) {
            int v = f(c);
            if(nodes[p].next[v] == -1) {
                nodes[p].next[v] = newNode();
            }
            p = nodes[p].next[v];
        }
        nodes[p].answer += 1;
    }

    int count(const string& s, int p = 0) {
        for(const char& c : s) {
            int v = f(c);
            if(nodes[p].next[v] == -1) {
                return 0;
            }
            p = nodes[p].next[v];
        }
        return nodes[p].answer;
    }

    void clear() {
        nodes.clear();
        newNode();
    }

    void reserve(int n) {
        nodes.reserve(n);
    }

private:
    vector<Node> nodes;

    inline int newNode() {
        nodes.emplace_back();
        return (int) nodes.size() - 1;
    }
};
```

7.9 AhoCorasick.h

```
template<int ALPHABET, int (*f)(char)>
class AhoCorasick {
public:
    struct Node {
        int fail = -1;
        int answer = 0;
        int next[ALPHABET];

        Node() {
            memset(next, -1, sizeof(next));
        }
    };

    AhoCorasick() : AhoCorasick(vector<string>()) {}

    AhoCorasick(const vector<string>& strs) {
        clear();
        for(const string& s : strs) {
            query_index.push_back(insert(s));
        }
    }

    int insert(const string& s) {
        int p = 0;
        for(int i = 0; i < (int) s.size(); ++i) {
            int v = f(s[i]);
            if(nodes[p].next[v] == -1) {
                nodes[p].next[v] = newNode();
            }
            p = nodes[p].next[v];
        }
        return p;
    }

    vector<int> solve(const string& s) {
        build_failure_all();
        int p = 0;
        for(int i = 0; i < (int) s.size(); ++i) {
            int v = f(s[i]);
            while(p > 0 && nodes[p].next[v] == -1) {
                p = nodes[p].fail;
            }
            if(nodes[p].next[v] != -1) {
                p = nodes[p].next[v];
                nodes[p].answer += 1;
            }
        }
        for(int i = (int) que.size() - 1; i >= 0; --i) {
            nodes[nodes[que[i]].fail].answer += nodes[que[i]].answer;
        }
        vector<int> res(query_index.size());
        for(int i = 0; i < (int) res.size(); ++i) {
            res[i] = nodes[query_index[i]].answer;
        }
        return res;
    }

    void clear() {
        nodes.clear();
        que.clear();
        query_index.clear();
        newNode();
        nodes[0].fail = 0;
    }

    void reserve(int n) {
        nodes.reserve(n);
    }

private:
    vector<Node> nodes;
    vector<int> que;
    vector<int> query_index;

    inline int newNode() {
        nodes.emplace_back();
        return (int) nodes.size() - 1;
    }

    void build_failure(int p) {
        for(int i = 0; i < ALPHABET; ++i) {
            if(nodes[p].next[i] != -1) {
                int tmp = nodes[p].fail;
                while(tmp > 0 && nodes[tmp].next[i] == -1) {
                    tmp = nodes[tmp].fail;
                }
                if(nodes[tmp].next[i] != nodes[p].next[i] && nodes[tmp].next[i] != -1) {
                    tmp = nodes[tmp].next[i];
                }
                nodes[nodes[p].next[i]].fail = tmp;
                que.push_back(nodes[p].next[i]);
            }
        }
    }

    void build_failure_all() {
        que.clear();
        que.reserve(nodes.size());
        que.push_back(0);
        for(int i = 0; i < (int) que.size(); ++i) {
            build_failure(que[i]);
        }
    }
};

```

8 Misc

8.1 Aliens.h

```
// find minimum
int aliens(int l, int r, int k) {
    while(l < r) {
        int m = l + (r - l) / 2;
        auto [score, op] = f(m);
        if(op == k) {
            return score - m * k;
        }
        if(op < k) {
            r = m;
        } else {
            l = m + 1;
        }
    }
    return f(l).first - l * k;
}
```

8.2 Timer.h

```
const clock_t startTime = clock();
inline double getCurrentTime() {
    return (double) (clock() - startTime) / CLOCKS_PER_SEC;
}

```

8.3 Random.h

```cpp
class random_t {
public:
    mt19937_64 rng;
    unsigned long long seed;

    random_t() : random_t(chrono::steady_clock::now().time_since_epoch().count()) {}

    random_t(unsigned long long s) : rng(s), seed(s) {}

    inline void set_seed(unsigned long long s) {
        seed = s;
        rng = mt19937_64(s);
    }

    inline void reset() {
        set_seed(seed);
    }

    inline unsigned long long next() {
        return uniform_int_distribution<unsigned long long>(0, ULLONG_MAX)(rng);
    }

    inline unsigned long long next(unsigned long long a) {
        return uniform_int_distribution<unsigned long long>(0, a - 1)(rng);
    }

    inline unsigned long long next(unsigned long long a, unsigned long long b) {
        return uniform_int_distribution<unsigned long long>(a, b)(rng);
    }

    inline long double nextDouble() {
        return uniform_real_distribution<long double>(0.0, 1.0)(rng);
    }

    inline long double nextDouble(long double a) {
        return nextDouble() * a;
    }

    inline long double nextDouble(long double a, long double b) {
        return uniform_real_distribution<long double>(a, b)(rng);
    }

    template<class T>
    void shuffle(vector<T>& a) {
        for(int i = (int) a.size() - 1; i >= 0; --i) {
            swap(a[i], a[next(i + 1)]);
        }
    }
};

```

8.4 Debug.h

```cpp
const string NONE = "\033[m", RED = "\033[0;32;31m", LIGHT_RED = "\033[1;31m", GREEN = "\033[0;32;32m", LIGHT_GREEN = "\033[1;32m", BLUE = "\033[0;32;34m", LIGHT_BLUE = "\033[1;34m", DARK_GRAY = "\033[1;30m", CYAN = "\033[0;36m", LIGHT_CYAN = "\033[1;36m", PURPLE = "\033[0;35m", LIGHT_PURPLE = "\033[1;35m", BROWN = "\033[0;33m", YELLOW = "\033[1;33m", LIGHT_GRAY = "\033[0;37m", WHITE = "\033[1;37m";
template<class c> struct rge { c b, e; };
template<class c> rge<c> range(c i, c j) { return rge<c>{i, j}; }
template<class c> auto dud(c* x)->decltype(cerr << *x, 0);
template<class c> char dud(...);
struct debug {
#ifdef LOCAL
    ~debug() { cerr << endl; }
    template<class c> typename enable_if<sizeof dud<c>(0) != 1, debug&>::type operator<<(c i) { cerr << boolalpha << i; return *this; }
    template<class c> typename enable_if<sizeof dud<c>(0) == 1, debug&>::type operator<<(c i) { return *this << range(begin(i), end(i)); }
    template<class c, class b> debug& operator<<(pair<b, c> d) { return *this << "(" << d.first << ", " << d.second << ")"; }
    template<class a, class b, class c> debug& operator<<(tuple<a, b, c> tp) { return *this << "(" << get<0>(tp) << ", " << get<1>(tp) << ", " << get<2>(tp) << ")"; };
    template<class a, class b, class c, class d> debug& operator<<(tuple<a, b, c, d> tp) { return *this << "(" << get<0>(tp) << ", " << get<1>(tp) << ", " << get<2>(tp) << ", " << get<3>(tp) << ")"; };
    template<class c> debug& operator<<(rge<c> d) {
        *this << "{";
        for(auto it = d.b; it != d.e; ++it) {
            *this << ", " + 2 * (it == d.b) << *it;
        }
        return *this << "}";
    }
#else
    template<class c> debug& operator<<(const c&) { return *this; }
#endif
};
#define show(...) "" << LIGHT_RED << " [" << NONE << #__VA_ARGS__ ": " << (__VA_ARGS__) << LIGHT_RED << "] " << NONE << ""

```

8.5 Discrete.h

```cpp
template<class T>
vector<int> discrete(const vector<T>& a, int OFFSET = 0) {
    vector<T> b(a);
    sort(b.begin(), b.end());
    b.erase(unique(b.begin(), b.end()), b.end());
    vector<int> c(a.size());
    for(int i = 0; i < (int) a.size(); ++i) {
        c[i] = int(lower_bound(b.begin(), b.end(), a[i]) - b.begin()) + OFFSET;
    }
    return c;
}

```

8.6 Template.h

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
using namespace std;
using namespace __gnu_pbds;

using uint = unsigned int;
using ll = long long;
using ull = unsigned long long;
using ld = long double;
template<class T> using pair2 = pair<T, T>;
using pii = pair2<int>;
using pll = pair2<ll>;
using pdd = pair2<ld>;
using vi = vector<int>;
using vl = vector<ll>;
template<class T> using PQ = priority_queue<T>;
template<class T> using PQG = priority_queue<T, vector<T>, greater<T>>;
template<class T, class Comp = less<T>> using ordered_set = tree<T, null_type, Comp, rb_tree_tag, tree_order_statistics_node_update>;
template<class T> using ordered_multiset = ordered_set<T, less_equal<T>>;

struct splitmix64_hash {
    static ull splitmix64(ull x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    ull operator()(ull x) const {
        static const ull FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};

template<class T, class U, class H = splitmix64_hash> using hash_map = gp_hash_table<T, U, H>;
template<class T, class H = splitmix64_hash> using hash_set = hash_map<T, null_type, H>;

template<class T> inline bool chmin(T& a, const T& b) { if(a > b) { a = b; return true; } return false; }
template<class T> inline bool chmax(T& a, const T& b) { if(a < b) { a = b; return true; } return false; }

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    ;
    return 0;
}

```