1 ******** general matching

```
// _mm_setcsr( _mm_getcsr() | 0x0040 | 0x8000 );
#include <bits/stdc++.h>

std::vector<std::pair<int, int>> maximum_matching(
    int n,
    const std::vector<std::pair<int, int>> &edg) {
    struct DSU {
        std::vector<int> prv;
        DSU(int n = 0) : prv(n, -1) { ; }
        void clear(int n) { prv.assign(n, -1); }
        int get(int i) { return prv[i] == -1 ? i : prv[i] = get(prv[i]); }
    };
    std::vector<int> vec(n, -1);
    std::vector<std::vector<int>> gr(n);
    {
        std::vector<int> cnt(n);
        for (auto [u, v] : edg) {
            cnt[u]++, cnt[v]++;
        }
        for (int i = 0; i < n; i++) {
            gr[i].reserve(cnt[i]);
        }
        for (auto [u, v] : edg) {
            gr[u].push_back(v);
            gr[v].push_back(u);
        }
    }

    DSU dsu(n), dsu2(n + 1);
    std::vector<std::array<int, 2>> prv(n, {-1, -1});
    std::vector<std::array<int, 3>> fwd(n, {-1, -1, -1}), bwd(n, {-1, -1, -1});
    std::vector<int> bl_dir(n, -1), bl_prv(n, -1), bl_lvl(n, -1);
    std::vector<int> bl_depth(n, -1), bl_jump(n, -1), bl_ord(n, -1);
    std::vector<int> depth(n, -1);
    std::vector<int64_t> used(n, -1);
    int64_t used_mark = -1;
    std::deque<int> deque;
    std::vector<std::pair<int, int>> aug_vec;
    int total_bl = 0;

    auto find_lca = [&](int u, int v) {
        used_mark++;
        u = dsu.get(u), v = dsu.get(v);
        while (u >= 0 || v >= 0) {
            if (u != -1) {
                if (used[u] == used_mark) {
                    return u;
                }
                used[u] = used_mark, (u = prv[u][0] >= 0 ? dsu.get(prv[u][0]) : prv[u][0]);
            }
            std::swap(u, v);
        }
        return -1;
    };

    auto contract_blossom = [&](int u0, int v0, int l) {
        bl_lvl[l] = ++total_bl;
        bl_ord.push_back(l);
        for (int ch = 0; ch < 2; ch++) {
            int x0 = ch == 0 ? u0 : v0;
            int y0 = ch == 0 ? v0 : u0;
            int x = dsu.get(x0);
            while (x != l) {
                int p0 = prv[x][0];
                int p = dsu.get(p0);

                if (depth[x] == 1) {
                    deque.push_back(x);
                }
                bl_dir[x] = depth[x] ^ ch;
                bl_prv[x] = l;
                fwd[x] = {dsu.get(p0), p0, prv[x][1]};
                bwd[x] = {dsu.get(y0), y0, x0};

                if (ch) {
                    std::swap(fwd[x], bwd[x]);
                }
                x0 = p0, y0 = prv[x][1], x = p;
            }
        }
        for (auto x : std::array<int, 2>{dsu.get(u0), dsu.get(v0)}) {
            for (; x != l; x = dsu.get(prv[x][0])) {
                dsu.prv[x] = l;
            }
        }
    };
    auto init_jump = [&]() {
        for (int i = bl_ord.size() - 1; i >= 0; i--) {
            int v = bl_ord[i];
            if (bl_depth[v] == -1) {
                int f = bl_prv[v];
                bl_depth[v] = bl_depth[f] + 1;
                bl_jump[v] = f;
                if (bl_depth[v] > 1 && bl_depth[f] - bl_depth[bl_jump[f]] ==
                                           bl_depth[bl_jump[f]] - bl_depth[bl_jump[bl_jump[f]]]) {
                    bl_jump[v] = bl_jump[bl_jump[f]];
                }
            }
        }
    };
    auto jump = [&](int x, int d) {
        while (bl_lvl[bl_prv[x]] < d) {
            if (bl_lvl[bl_jump[x]] < d) {
                x = bl_jump[x];
            } else {
                x = bl_prv[x];
            }
        }
        return x;
    };
    auto augment = [&](int u0, int v0) {
        int u = dsu.get(u0), v = dsu.get(v0);
        std::list<std::pair<int, int>> list;
        for (int x = dsu.get(u0); prv[x][0] != -1; x = dsu.get(prv[x][0])) {
            list.push_front({prv[x][0], prv[x][1]});
        }
        list.push_front({-1, prv[u][0] == -1 ? u : dsu.get(list.front().first)});
        list.push_back({u0, v0});
        for (int x = dsu.get(v0); prv[x][0] != -1; x = dsu.get(prv[x][0])) {
            list.push_back({prv[x][1], prv[x][0]});
```

```
    }
    list.push_back({prv[v][0] == -1 ? v : dsu.get(list.back().second), -1});

    for (auto it = std::next(list.begin()); it != list.end();) {
        auto [a1, a2] = *std::prev(it);
        auto [b1, b2] = *it;
        if (a2 == b1) {
            it++;
            continue;
        }
        bool f1 = (a1 == -1 || vec[a1] == a2);
        bool f2 = (b2 == -1 || vec[b1] == b2);
        assert(f1 != f2);
        bool rev = false;
        if (f2) {
            rev = true;
            std::swap(a1, b2);
            std::swap(a2, b1);
        }
        decltype(list) list2;
        int c = jump(b1, bl_lvl[a2]);
        auto &mv = !bl_dir[c] ? fwd : bwd;
        for (int x = c; x != a2; x = mv[x][0]) {
            list2.push_front({mv[x][1], mv[x][2]});
        }
        if (!rev) {
            it = list.insert(it, list2.begin(), list2.end());
        } else {
            for (auto &[a, b] : list2) {
                std::swap(a, b);
            }
            it = list.insert(it, list2.rbegin(), list2.rend());
        }
    }
    list.pop_front(), list.pop_back();
    assert(list.size() % 2 == 1);
    for (auto it = list.begin();; it = std::next(it, 2)) {
        vec[it->first] = it->second;
        vec[it->second] = it->first;
        if (std::next(it) == list.end()) {
            break;
        }
    }
};

auto process_edge = [&](int u0, int v0) {
    int u = dsu.get(u0);
    int v = dsu.get(v0);
    if (u == v || depth[v] == 1 || dsu2.get(u) == n || dsu2.get(v) == n) {
        return;
    }
    if (depth[v] == -1) {
        int w = vec[v];
        dsu2.prv[v] = u;
        dsu2.prv[w] = u;
        depth[v] = 1;
        depth[w] = 0;
        prv[v] = {u0, v0};
        prv[w] = {v0, w};
        deque.push_back(w);
    } else {
        int l = find_lca(u, v);
        if (l != -1) {
            contract_blossom(u0, v0, l);
        } else {
            // augment(u0, v0);
            aug_vec.push_back({u0, v0});
            for (int x : std::array<int, 2>{u, v}) {
                for (; x != -1; x = (prv[x][0] == -1 ? -1 : dsu.get(prv[x][0]))) {
                    dsu2.prv[x] = n;
                }
            }
        }
    }
};

while (true) {
    dsu.clear(n), dsu2.clear(n + 1);
    prv.assign(n, {-1, -1});
    fwd.assign(n, {-1, -1, -1}), bwd.assign(n, {-1, -1, -1});
    bl_dir.assign(n, -1), bl_prv.assign(n, -1), bl_lvl.assign(n, -1);
    depth.assign(n, -1);
    deque.clear();
    aug_vec.clear();
    bl_ord.assign(n, -1);
    bl_depth.assign(n, -1);
    bl_jump.assign(n, -1);
    std::iota(bl_ord.begin(), bl_ord.end(), 0);
    std::iota(bl_prv.begin(), bl_prv.end(), 0);
    total_bl = 0;

    for (int i = 0; i < n; i++) {
        if (vec[i] == -1) {
            depth[i] = 0;
            bl_lvl[i] = total_bl;
            deque.push_back(i);
        }
    }
    while (deque.size()) {
        int u0 = deque.front();
        deque.pop_front();
        for (int v0 : gr[u0]) {
            process_edge(u0, v0);
        }
    }

    if (aug_vec.size()) {
        init_jump();
        std::cerr << aug_vec.size() << " ";
        for (auto [u0, v0] : aug_vec) {
            augment(u0, v0);
        }

    } else {
        break;
    }
}

std::vector<std::pair<int, int>> res;
for (int i = 0; i < n; i++) {
    if (vec[i] > i) {
        res.push_back({i, vec[i]});
```

```
        }
    }
    return res;
}
```

2 suff

```
#include <bits/stdc++.h>

struct Node {
    int32_t go[26];
    int32_t suf_link = -1;
    int32_t parent = -1;
    int32_t count = 1;

    int32_t min_len = 1'000'000'000, max_len = -1'000'000'000;

    Node() {
        for (int32_t i = 0; i < 26; i++)
            go[i] = -1;
    }

    void copy_go(Node& node) {
        for (int32_t i = 0; i < 26; i++)
            go[i] = node.go[i];
    }
};

int32_t next = 1;

int32_t extend(int32_t last, int32_t ch, Node* nodes) {
    int32_t b = next++;
    nodes[b].parent = last;
    for (int32_t a = last; a > -1; a = nodes[a].suf_link) {
        if (nodes[a].go[ch] == -1) {
            nodes[a].go[ch] = b;
            continue;
        }

        int32_t c = nodes[a].go[ch];
        if (nodes[c].parent == a) {
            nodes[b].suf_link = c;
            return b;
        }

        int32_t clone = next++;
        nodes[clone].copy_go(nodes[c]);
        nodes[clone].parent = a;

        nodes[clone].suf_link = nodes[c].suf_link;
        nodes[c].suf_link = clone;
        nodes[b].suf_link = clone;

        nodes[clone].count = 0;

        for (; a > -1; a = nodes[a].suf_link)
            if (nodes[a].go[ch] == c)
                nodes[a].go[ch] = clone;
            else
                break;

        return b;
    }
    nodes[b].suf_link = 0;
    return b;
}
```

3 svg

```
#include <stdint.h>

#include <fstream>
#include <sstream>

template <typename T>
struct Vector {
    T x, y;

    Vector() : x(0), y(0) { ; }

    Vector(T x, T y) : x(x), y(y) { ; }

    // template <typename T2>
    // operator Vector<T2>() const {
    //     return Vector<T2>(x, y);
    // }

    Vector operator+(const Vector& other) const {
        return Vector(x + other.x, y + other.y);
    }

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

    Vector operator-(const Vector& other) const {
        return Vector(x - other.x, y - other.y);
    }

    T operator*(const Vector& other) const {
        return x * other.x + y * other.y;
    }

    T operator%(const Vector& other) const {
        return x * other.y - y * other.x;
    }

    Vector<double> operator*(const double& val) const {
        return Vector(x * val, y * val);
    }

    bool operator==(const Vector& other) const {
        return x == other.x && y == other.y;
    }
};

struct Svg {
    std::stringstream sout;

    static constexpr double scale = 500;
    static constexpr double shift = 50;
```

```
    Svg() {
        clear();
    }

    void clear() {
        sout = std::stringstream();
        sout.precision(5);
        sout << std::fixed;
        sout << R"meow(<svg width=\"1000px\" height=\"1000px\" style=\"background-color:lightgreen\"
        xmlns=\"http://www.w3.org/2000/svg\">\n)meow";
    }

    void print() {
        std::string s = sout.str();
        s += "</svg>\n";

        std::ofstream fout("meow.svg");
        fout << s << "\n";
        fout.flush();
        fout.close();
    }

    void line(Vector<double> pt1, Vector<double> pt2, std::string color, double width = 1) {
        sout << "<line ";
        sout << "x1=\"" << (float)(pt1.x * scale + shift) << "\" ";
        sout << "y1=\"" << (float)((scale - pt1.y * scale) + shift) << "\" ";
        sout << "x2=\"" << (float)(pt2.x * scale + shift) << "\" ";
        sout << "y2=\"" << (float)((scale - pt2.y * scale) + shift) << "\" ";
        sout << " stroke=\"" << color << "\"";
        sout << " stroke-width=\"" << float(width) << "\"";
        sout << "/>\n";
    }

    void circle(Vector<double> pt, double r, std::string color, double width = 1) {
        sout << "<circle ";
        sout << "cx=\"" << float(pt.x * scale + shift) << "\" ";
        sout << "cy=\"" << float((scale - pt.y * scale) + shift) << "\" ";
        sout << "r=\"" << float(r) << "\" ";
        sout << " stroke=\"" << color << "\"";
        sout << " stroke-width=\"" << float(width) << "\"";
        sout << "/>\n";
    };
};
```

4 ******** ntt

```
#include <immintrin.h>

#include <algorithm>
#include <array>
#include <cassert>
#include <cstdint>
#include <cstring>
#include <vector>

#pragma GCC target("avx2,bmi")

using u32 = uint32_t;
using u64 = uint64_t;

struct Montgomery {
    u32 mod;    // mod
    u32 mod2;   // 2 * mod
    u32 n_inv;  // n_inv * mod == -1 (mod 2^32)
    u32 r;      // 2^32 % mod
    u32 r2;     // (2^32)^2 % mod

    Montgomery() = default;
    Montgomery(u32 mod) : mod(mod) {
        assert(mod % 2 == 1);
        assert(mod < (1 << 30));
        mod2 = 2 * mod;
        n_inv = 1;
        for (int i = 0; i < 5; i++) {
            n_inv *= 2 + n_inv * mod;
        }
        r = (u64(1) << 32) % mod;
        r2 = u64(r) * r % mod;
    }

    u32 shrink(u32 val) const {
        return std::min(val, val - mod);
    }
    u32 shrink2(u32 val) const {
        return std::min(val, val - mod2);
    }

    template <bool strict = true>
    u32 reduce(u64 val) const {
        u32 res = val + u32(val) * n_inv * u64(mod) >> 32;
        if constexpr (strict)
            res = shrink(res);
        return res;
    }

    template <bool strict = true>
    u32 mul(u32 a, u32 b) const {
        return reduce<strict>(u64(a) * b);
    }

    template <bool input_in_space = false, bool output_in_space = false>
    u32 power(u32 b, u32 e) const {
        if (!input_in_space)
            b = mul<false>(b, r2);
        u32 r = output_in_space ? this->r : 1;
        for (; e > 0; e >>= 1) {
            if (e & 1)
                r = mul<false>(r, b);
            b = mul<false>(b, b);
        }
        return shrink(r);
    }
};

using i256 = __m256i;
using u32x8 = u32 __attribute__((vector_size(32)));
using u64x4 = u64 __attribute__((vector_size(32)));

u32x8 load_u32x8(const u32* ptr) {
    return (u32x8)_mm256_load_si256((const i256*)ptr);
```

```
}
void store_u32x8(u32* ptr, u32x8 vec) {
    _mm256_store_si256((i256*)ptr, (i256)vec);
}

struct Montgomery_simd {
    u32x8 mod;    // mod
    u32x8 mod2;   // 2 * mod
    u32x8 n_inv;  // n_inv * mod == -1 (mod 2^32)
    u32x8 r;      // 2^32 % mod
    u32x8 r2;     // (2^32)^2 % mod

    Montgomery_simd() = default;
    Montgomery_simd(u32 mod) {
        Montgomery mt(mod);
        this->mod = (u32x8)_mm256_set1_epi32(mt.mod);
        this->mod2 = (u32x8)_mm256_set1_epi32(mt.mod2);
        this->n_inv = (u32x8)_mm256_set1_epi32(mt.n_inv);
        this->r = (u32x8)_mm256_set1_epi32(mt.r);
        this->r2 = (u32x8)_mm256_set1_epi32(mt.r2);
    }

    u32x8 shrink(u32x8 vec) const {
        return (u32x8)_mm256_min_epu32((i256)vec, _mm256_sub_epi32((i256)vec, (i256)mod));
    }
    u32x8 shrink2(u32x8 vec) const {
        return (u32x8)_mm256_min_epu32((i256)vec, _mm256_sub_epi32((i256)vec, (i256)mod2));
    }
    u32x8 shrink_n(u32x8 vec) const {
        return (u32x8)_mm256_min_epu32((i256)vec, _mm256_add_epi32((i256)vec, (i256)mod));
    }
    u32x8 shrink2_n(u32x8 vec) const {
        return (u32x8)_mm256_min_epu32((i256)vec, _mm256_add_epi32((i256)vec, (i256)mod2));
    }

    template <bool strict = true>
    u32x8 reduce(u64x4 x0246, u64x4 x1357) const {
        u64x4 x0246_ninv = (u64x4)_mm256_mul_epu32((i256)x0246, (i256)n_inv);
        u64x4 x1357_ninv = (u64x4)_mm256_mul_epu32((i256)x1357, (i256)n_inv);
        u64x4 x0246_res = (u64x4)_mm256_add_epi64((i256)x0246, _mm256_mul_epu32((i256)x0246_ninv, (i256)mod));
        u64x4 x1357_res = (u64x4)_mm256_add_epi64((i256)x1357, _mm256_mul_epu32((i256)x1357_ninv, (i256)mod));
        u32x8 res = (u32x8)_mm256_or_si256(_mm256_bsrli_epi128((i256)x0246_res, 4), (i256)x1357_res);
        if (strict)
            res = shrink(res);
        return res;
    }

    template <bool strict = true, bool b_use_only_even = false>
    u32x8 mul_u32x8(u32x8 a, u32x8 b) const {
        u32x8 a_sh = (u32x8)_mm256_bsrli_epi128((i256)a, 4);
        u32x8 b_sh = b_use_only_even ? b : (u32x8)_mm256_bsrli_epi128((i256)b, 4);
        u64x4 x0246 = (u64x4)_mm256_mul_epu32((i256)a, (i256)b);
        u64x4 x1357 = (u64x4)_mm256_mul_epu32((i256)a_sh, (i256)b_sh);
        return reduce<strict>(x0246, x1357);
    }

    template <bool strict = true>
    u64x4 mul_u64x4(u64x4 a, u64x4 b) const {
        u64x4 pr = (u64x4)_mm256_mul_epu32((i256)a, (i256)b);
        u64x4 pr2 = (u64x4)_mm256_mul_epu32(_mm256_mul_epu32((i256)pr, (i256)n_inv), (i256)mod);
        u64x4 res = (u64x4)_mm256_bsrli_epi128(_mm256_add_epi64((i256)pr, (i256)pr2), 4);
        if (strict)
            res = (u64x4)shrink((u32x8)res);
        return res;
    }
};

class NTT {
   public:
    u32 mod, pr_root;

   private:
    static constexpr int LG = 32;  // more than enough for u32

    Montgomery mt;
    Montgomery_simd mts;

    u32 w[4], wr[4];
    u32 wd[LG], wrd[LG];

    u64x4 wt_init, wrt_init;
    u64x4 wd_x4[LG], wrd_x4[LG];
    u64x4 wl_init;
    u64x4 wld_x4[LG];

    static u32 find_pr_root(u32 mod, const Montgomery& mt) {
        std::vector<u32> factors;
        u32 n = mod - 1;
        for (u32 i = 2; u64(i) * i <= n; i++) {
            if (n % i == 0) {
                factors.push_back(i);
                do {
                    n /= i;
                } while (n % i == 0);
            }
        }
        if (n > 1) {
            factors.push_back(n);
        }
        for (u32 i = 2; i < mod; i++) {
            if (std::all_of(factors.begin(), factors.end(),
                            [&](u32 f) { return mt.power<false, false>(i, (mod - 1) / f) != 1; })) {
                return i;
            }
        }
        assert(false && "primitive root not found");
    }

   public:
    NTT() = default;
    NTT(u32 mod) : mod(mod), mt(mod), mts(mod) {
        const Montgomery mt = this->mt;
        const Montgomery_simd mts = this->mts;

        pr_root = find_pr_root(mod, mt);

        int lg = __builtin_ctz(mod - 1);
        assert(lg <= LG);

        memset(w, 0, sizeof(w));
        memset(wr, 0, sizeof(wr));
        memset(wd_x4, 0, sizeof(wd_x4));
```

```
    memset(wrd_x4, 0, sizeof(wrd_x4));
    memset(wld_x4, 0, sizeof(wld_x4));

    std::vector<u32> vec(lg + 1), vecr(lg + 1);
    vec[lg] = mt.power<false, true>(pr_root, mod - 1 >> lg);
    vecr[lg] = mt.power<true, true>(vec[lg], mod - 2);
    for (int i = lg - 1; i >= 0; i--) {
        vec[i] = mt.mul<true>(vec[i + 1], vec[i + 1]);
        vecr[i] = mt.mul<true>(vecr[i + 1], vecr[i + 1]);
    }

    w[0] = wr[0] = mt.r;
    if (lg >= 2) {
        w[1] = vec[2], wr[1] = vecr[2];
        if (lg >= 3) {
            w[2] = vec[3], wr[2] = vecr[3];
            w[3] = mt.mul<true>(w[1], w[2]);
            wr[3] = mt.mul<true>(wr[1], wr[2]);
        }
    }
    wt_init = (u64x4)_mm256_setr_epi64x(w[0], w[0], w[0], w[1]);
    wrt_init = (u64x4)_mm256_setr_epi64x(wr[0], wr[0], wr[0], wr[1]);

    wl_init = (u64x4)_mm256_setr_epi64x(w[0], w[1], w[2], w[3]);

    u32 prf = mt.r, prf_r = mt.r;
    for (int i = 0; i < lg - 2; i++) {
        u32 f = mt.mul<true>(prf, vec[i + 3]), fr = mt.mul<true>(prf_r, vecr[i + 3]);
        prf = mt.mul<true>(prf, vecr[i + 3]), prf_r = mt.mul<true>(prf_r, vec[i + 3]);
        u32 f2 = mt.mul<true>(f, f), f2r = mt.mul<true>(fr, fr);

        wd_x4[i] = (u64x4)_mm256_setr_epi64x(f2, f, f2, f);
        wrd_x4[i] = (u64x4)_mm256_setr_epi64x(f2r, fr, f2r, fr);
    }

    prf = mt.r;
    for (int i = 0; i < lg - 3; i++) {
        u32 f = mt.mul<true>(prf, vec[i + 4]);
        prf = mt.mul<true>(prf, vecr[i + 4]);
        wld_x4[i] = (u64x4)_mm256_set1_epi64x(f);
    }
}

private:
 static constexpr int L0 = 3;
 int get_low_lg(int lg) const {
     return lg % 2 == L0 % 2 ? L0 : L0 + 1;
 }

 //    public:
 //     bool lg_available(int lg) {
 //         return L0 <= lg && lg <= __builtin_ctz(mod - 1) + get_low_lg(lg);
 //     }

private:
 template <bool transposed, bool trivial = false>
 static void butterfly_x2(u32* ptr_a, u32* ptr_b, u32x8 w, const Montgomery_simd& mts) {
     u32x8 a = load_u32x8(ptr_a), b = load_u32x8(ptr_b);
     u32x8 a2, b2;
     if (!transposed) {
         a = mts.shrink2(a), b = trivial ? mts.shrink2(b) : mts.mul_u32x8<false, true>(b, w);
         a2 = a + b, b2 = a + mts.mod2 - b;
     } else {
         a2 = mts.shrink2(a + b), b2 = trivial ? mts.shrink2_n(a - b)
                                               : mts.mul_u32x8<false, true>(a + mts.mod2 - b, w);
     }
     store_u32x8(ptr_a, a2), store_u32x8(ptr_b, b2);
 }

 template <bool transposed, bool trivial = false>
 static void butterfly_x4(u32* ptr_a, u32* ptr_b, u32* ptr_c, u32* ptr_d,
                          u32x8 w1, u32x8 w2, u32x8 w3, const Montgomery_simd& mts) {
     u32x8 a = load_u32x8(ptr_a), b = load_u32x8(ptr_b), c = load_u32x8(ptr_c), d = load_u32x8(ptr_d);
     if (!transposed) {
         butterfly_x2<false, trivial>((u32*)&a, (u32*)&c, w1, mts);
         butterfly_x2<false, trivial>((u32*)&b, (u32*)&d, w1, mts);
         butterfly_x2<false, trivial>((u32*)&a, (u32*)&b, w2, mts);
         butterfly_x2<false, false>((u32*)&c, (u32*)&d, w3, mts);
     } else {
         butterfly_x2<true, trivial>((u32*)&a, (u32*)&b, w2, mts);
         butterfly_x2<true, false>((u32*)&c, (u32*)&d, w3, mts);
         butterfly_x2<true, trivial>((u32*)&a, (u32*)&c, w1, mts);
         butterfly_x2<true, trivial>((u32*)&b, (u32*)&d, w1, mts);
     }
     store_u32x8(ptr_a, a), store_u32x8(ptr_b, b), store_u32x8(ptr_c, c), store_u32x8(ptr_d, d);
 }

 template <bool inverse, bool trivial = false>
 void transform_aux(int k, int i, u32* data, u64x4& wi, const Montgomery_simd& mts) const {
     u32x8 w1 = (u32x8)_mm256_shuffle_epi32((i256)wi, 0b00'00'00'00);
     // only even indices will be used
     u32x8 w2 = (u32x8)_mm256_permute4x64_epi64((i256)wi, 0b01'01'01'01);
     u32x8 w3 = (u32x8)_mm256_permute4x64_epi64((i256)wi, 0b11'11'11'11);
     for (int j = 0; j < (1 << k); j += 8) {
         butterfly_x4<inverse, trivial>(data + i + (1 << k) * 0 + j, data + i + (1 << k) * 1 + j,
                                        data + i + (1 << k) * 2 + j, data + i + (1 << k) * 3 + j,
                                        w1, w2, w3, mts);
     }
     wi = mts.mul_u64x4<true>(wi, (inverse ? wrd_x4 : wd_x4)[__builtin_ctz(~i >> k + 2)]);
 }

public:
 // input in [0, 4 * mod)
 // output in [0, 4 * mod)
 // data must be 32-byte aligned
 void transform_forward(int lg, u32* data) const {
     const Montgomery_simd mts = this->mts;
     const int L = get_low_lg(lg);

     // for (int k = lg - 2; k >= L; k -= 2) {
     //     u64x4 wi = wt_init;
     //     transform_aux<false, true>(k, 0, data, wi, mts);
     //     for (int i = (1 << k + 2); i < (1 << lg); i += (1 << k + 2)) {
     //         transform_aux<false>(k, i, data, wi, mts);
     //     }
     // }

     if (L < lg) {
         const int lc = (lg - L) / 2;
         u64x4 wi_data[LG / 2];
         std::fill(wi_data, wi_data + lc, wt_init);
```

```
        for (int k = lg - 2; k >= L; k -= 2) {
            transform_aux<false, true>(k, 0, data, wi_data[k - L >> 1], mts);
        }
        for (int i = 1; i < (1 << lc * 2 - 2); i++) {
            int s = __builtin_ctz(i) >> 1;
            for (int k = s; k >= 0; k--) {
                transform_aux<false>(2 * k + L, i * (1 << L + 2), data, wi_data[k], mts);
            }
        }
    }
}

// input in [0, 2 * mod)
// output in [0, mod)
// data must be 32-byte aligned
template <bool mul_by_sc = false>
void transform_inverse(int lg, u32* data, /* as normal number */ u32 sc = u32()) const {
    const Montgomery_simd mts = this->mts;
    const int L = get_low_lg(lg);

    // for (int k = L; k + 2 <= lg; k += 2) {
    //     u64x4 wi = wrt_init;
    //     transform_aux<true, true>(k, 0, data, wi, mts);
    //     for (int i = (1 << k + 2); i < (1 << lg); i += (1 << k + 2)) {
    //         transform_aux<true>(k, i, data, wi, mts);
    //     }
    // }

    if (L < lg) {
        const int lc = (lg - L) / 2;
        u64x4 wi_data[LG / 2];
        std::fill(wi_data, wi_data + lc, wrt_init);

        for (int i = 0; i < (1 << lc * 2 - 2); i++) {
            int s = __builtin_ctz(~i) >> 1;
            if (i + 1 == (1 << 2 * s)) {
                s--;
            }
            for (int k = 0; k <= s; k++) {
                transform_aux<true>(2 * k + L,
                                    (i + 1 - (1 << 2 * k)) * (1 << L + 2), data, wi_data[k], mts);
            }
            if (i + 1 == (1 << 2 * (s + 1))) {
                s++;
                transform_aux<true, true>(2 * s + L,
                                          (i + 1 - (1 << 2 * s)) * (1 << L + 2), data, wi_data[s], mts);
            }
        }
    }

    const Montgomery mt = this->mt;
    u32 f = mt.power<false, true>(mod + 1 >> 1, lg - L);
    if constexpr (mul_by_sc)
        f = mt.mul<true>(f, mt.mul<false>(mt.r2, sc));
    u32x8 f_x8 = (u32x8)_mm256_set1_epi32(f);
    for (int i = 0; i < (1 << lg); i += 8) {
        store_u32x8(data + i, mts.mul_u32x8<true, true>(load_u32x8(data + i), f_x8));
    }
}

private:
```

```
// input in [0, 4 * mod)
// output in [0, 2 * mod)
// multiplies mod (x^2^L - w)
template <int L, int K, bool remove_montgomery_reduction_factor = true>
/* !!! O3 is crucial here !!! */ __attribute__((optimize("O3"))) static void
aux_mul_mod_x2L(const u32* a, const u32* b, u32* c,
                const std::array<u32x8, K>& ar_w, const Montgomery_simd& mts) {
    static_assert(L >= 3);
    // static_assert(L == L0 || L == L0 + 1);

    constexpr int n = 1 << L;
    alignas(64) u32 aux_a[K][n];
    alignas(64) u64 aux_b[K][n * 2];
    for (int k = 0; k < K; k++) {
        for (int i = 0; i < n; i += 8) {
            u32x8 ai = load_u32x8(a + n * k + i);
            if constexpr (remove_montgomery_reduction_factor) {
                ai = mts.mul_u32x8<true, true>(ai, mts.r2);
            } else {
                ai = mts.shrink(mts.shrink2(ai));
            }
            store_u32x8(aux_a[k] + i, ai);

            u32x8 bi = load_u32x8(b + n * k + i);
            u32x8 bi_0 = mts.shrink(mts.shrink2(bi));
            u32x8 bi_w = mts.mul_u32x8<true, true>(bi, ar_w[k]);

            store_u32x8((u32*)(aux_b[k] + i + 0),
                        (u32x8)_mm256_permutevar8x32_epi32((i256)bi_w, _mm256_setr_epi64x(0, 1, 2, 3)));
            store_u32x8((u32*)(aux_b[k] + i + 4),
                        (u32x8)_mm256_permutevar8x32_epi32((i256)bi_w, _mm256_setr_epi64x(4, 5, 6, 7)));
            store_u32x8((u32*)(aux_b[k] + n + i + 0),
                        (u32x8)_mm256_permutevar8x32_epi32((i256)bi_0, _mm256_setr_epi64x(0, 1, 2, 3)));
            store_u32x8((u32*)(aux_b[k] + n + i + 4),
                        (u32x8)_mm256_permutevar8x32_epi32((i256)bi_0, _mm256_setr_epi64x(4, 5, 6, 7)));
        }
    }

    u64x4 aux_ans[K][n / 4];
    memset(aux_ans, 0, sizeof(aux_ans));
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < K; k++) {
            u64x4 ai = (u64x4)_mm256_set1_epi32(aux_a[k][i]);
            for (int j = 0; j < n; j += 4) {
                u64x4 bi = (u64x4)_mm256_loadu_si256((i256*)(aux_b[k] + n - i + j));
                aux_ans[k][j / 4] += /* 64-bit addition */
                    (u64x4)_mm256_mul_epu32((i256)ai, (i256)bi);
            }
        }
        if (i >= 8 && (i & 7) == 7) {
            for (int k = 0; k < K; k++) {
                for (int j = 0; j < n; j += 4) {
                    aux_ans[k][j / 4] = (u64x4)mts.shrink2((u32x8)aux_ans[k][j / 4]);
                }
            }
        }
    }

    for (int k = 0; k < K; k++) {
        for (int i = 0; i < n; i += 8) {
            u64x4 c0 = aux_ans[k][i / 4], c1 = aux_ans[k][i / 4 + 1];
```

```
            u32x8 res = (u32x8)_mm256_permutevar8x32_epi32(
                (i256)mts.reduce<false>(c0, c1), _mm256_setr_epi32(0, 2, 4, 6, 1, 3, 5, 7));
            store_u32x8(c + k * n + i, mts.shrink2(res));
        }
    }
}

template <int L, bool remove_montgomery_reduction_factor = true>
void aux_mul_mod_full(int lg, const u32* a, const u32* b, u32* c) const {
    constexpr int sz = 1 << L;
    const Montgomery_simd mts = this->mts;
    int cnt = 1 << lg - L;
    if (cnt == 1) {
        aux_mul_mod_x2L<L, 1, remove_montgomery_reduction_factor>(a, b, c, {mts.r}, mts);
        return;
    }
    if (cnt <= 8) {
        for (int i = 0; i < cnt; i += 2) {
            u32x8 wi = (u32x8)_mm256_set1_epi32(w[i / 2]);
            aux_mul_mod_x2L<L, 2, remove_montgomery_reduction_factor>(
                a + i * sz, b + i * sz, c + i * sz, {wi, (mts.mod - wi)}, mts);
        }
        return;
    }
    u64x4 wi = wl_init;
    for (int i = 0; i < cnt; i += 8) {
        u32x8 w_ar[4] = {
            (u32x8)_mm256_permute4x64_epi64((i256)wi, 0b00'00'00'00),
            (u32x8)_mm256_permute4x64_epi64((i256)wi, 0b01'01'01'01),
            (u32x8)_mm256_permute4x64_epi64((i256)wi, 0b10'10'10'10),
            (u32x8)_mm256_permute4x64_epi64((i256)wi, 0b11'11'11'11),
        };
        if constexpr (L == L0) {
            for (int j = 0; j < 8; j += 4) {
                aux_mul_mod_x2L<L, 4, remove_montgomery_reduction_factor>(
                    a + (i + j) * sz, b + (i + j) * sz, c + (i + j) * sz,
                    {w_ar[j / 2], mts.mod - w_ar[j / 2], w_ar[j / 2 + 1],
                     mts.mod - w_ar[j / 2 + 1]},
                    mts);
            }
        } else {
            for (int j = 0; j < 8; j += 2) {
                aux_mul_mod_x2L<L, 2, remove_montgomery_reduction_factor>(
                    a + (i + j) * sz, b + (i + j) * sz, c + (i + j) * sz,
                    {w_ar[j / 2], mts.mod - w_ar[j / 2]}, mts);
            }
        }
        wi = mts.mul_u64x4<true>(wi, wld_x4[__builtin_ctz(~i >> 3)]);
    }
}

public:
 template <bool remove_montgomery_reduction_factor = true>
 void aux_dot_mod(int lg, const u32* a, const u32* b, u32* c) const {
    int L = get_low_lg(lg);
    if (L == L0) {
        aux_mul_mod_full<L0, remove_montgomery_reduction_factor>(lg, a, b, c);
    } else {
        aux_mul_mod_full<L0 + 1, remove_montgomery_reduction_factor>(lg, a, b, c);
    }
 }
    // lg must be greater than or equal to 3
    // a, b must be 32-byte aligned
    void convolve_cyclic(int lg, u32* a, u32* b) const {
        transform_forward(lg, a);
        transform_forward(lg, b);
        aux_dot_mod(lg, a, b, a);
        transform_inverse(lg, a);
    }
};
```

5 mod linear

```
/*
    forged from https://mangooste.ru/lib/algo/mod-of-linear
*/

/*
 ! WARNING: careful with overflow. Don't forget to specify large enough type T.
 * Returns sum_{x = 0}^{n - 1} floor((kx + b) / m).
 * Require: k >= 0, b >= 0, m > 0, n >= 0.
 */
template <typename T>
T floor_sum(T k, T b, T m, T n) {
    if (k == 0) {
        return (b / m) * n;
    }
    if (k >= m || b >= m) {
        return n * (n - 1) / 2 * (k / m) + n * (b / m) + floor_sum(k % m, b % m, m, n);
    }
    T ymax = (k * (n - 1) + b) / m;
    return n * ymax - floor_sum(m, m + k - b - 1, k, ymax);
}

/*
 ! WARNING: careful with overflow. Don't forget to specify large enough to fit floor_sum type T.
 * Returns sum_{x = 0}^{n - 1} (kx + b) % m.
 * Require: m > 0, n >= 0.
 */
template <typename T>
T mod_sum(T k, T b, T m, T n) {
    k = (k % m + m) % m;
    b = (b % m + m) % m;
    return n * (n - 1) / 2 * k + n * b - m * floor_sum(k, b, m, n);
}

// ---------

/*
 * Returns min_{x=0}^{n - 1} (kx + b) mod m
 * Require: n, m > 0, 0 <= b, k < m
 */
template <typename T>
T min_of_mod_of_linear(T n, T m, T k, T b, T step_cost = 1, T overflow_cost = 0);

/*
 * Returns max_{x=0}^{n - 1} (kx + b) mod m
 * Require: n, m > 0, 0 <= b, k < m
 */
template <typename T>
```

```
T max_of_mod_of_linear(T n, T m, T k, T b, T step_cost = 1, T overflow_cost = 0);

template <typename T>
T max_of_mod_of_linear(T n, T m, T k, T b, T step_cost, T overflow_cost) {
    if (k == 0) {
        return b;
    }
    if (b < m - k) {
        T steps = (m - b - 1) / k;
        T cost = step_cost * steps;
        if (cost >= n) {
            return k * ((n - 1) / step_cost) + b;
        }
        n -= cost;
        b += steps * k;
    }
    return m - 1 - min_of_mod_of_linear(
        n, k, m % k, m - 1 - b, (m / k) * step_cost + overflow_cost, step_cost);
}

template <typename T>
T min_of_mod_of_linear(T n, T m, T k, T b, T step_cost, T overflow_cost) {
    if (k == 0) {
        return b;
    }
    if (b >= k) {
        T steps = (m - b + k - 1) / k;
        T cost = step_cost * steps + overflow_cost;
        if (cost >= n) {
            return b;
        }
        n -= cost;
        b += steps * k - m;
    }
    return k - 1 - max_of_mod_of_linear(
        n, k, m % k, k - 1 - b, (m / k) * step_cost + overflow_cost, step_cost);
}
```

6 ntt

```
#include <bits/stdc++.h>

using u32 = uint32_t;

constexpr u32 mod = 998244353;
constexpr u32 pr_root = 3;

u32 add(u32 a, u32 b) {
    return a + b - mod * (a + b >= mod);
}
void add_to(u32 &a, u32 b) {
    a = add(a, b);
}
u32 mul(u32 a, u32 b) {
    return a * 1ULL * b % mod;
}

u32 power(u32 b, u32 e) {
    u32 r = 1;
    for (; e > 0; e >>= 1) {
        if (e & 1) {
            r = mul(r, b);
        }
        b = mul(b, b);
    }
    return r;
}

int bit_ceil(int n) {
    int k = 1;
    while (k < n)
        k *= 2;
    return k;
}

int64_t total = 0;

struct NTT {
    std::vector<u32> wd, wrd;

    NTT() {
        int lg = __builtin_ctz(mod - 1);

        wd.assign(lg + 1, 0), wrd.assign(lg + 1, 0);
        for (int i = 0; i + 2 <= lg; i++) {
            u32 a = power(pr_root, (mod - 1 >> i + 2));
            u32 b = power(pr_root, (mod - 1 >> i + 2) * ((2 << i) - 2));

            u32 f = mul(a, power(b, mod - 2));

            wd[i] = f;
            wrd[i] = power(f, mod - 2);
        }
    }

    template <bool transposed>
    void butterfly_x2(u32 &a, u32 &b, u32 w) const {
        if (!transposed) {
            u32 a2 = a, b2 = mul(b, w);
            a = add(a2, b2), b = add(a2, mod - b2);
        } else {
            u32 a2 = add(a, b), b2 = mul(add(a, mod - b), w);
            a = a2, b = b2;
        }
    }

    template <bool inverse>
    void transform(int lg, u32 *data) const {
        total += 1 << lg;
        // std::cerr << "transform  " << inverse << "  " << lg << "\n";
        for (int k = inverse ? 0 : lg - 1; inverse ? k < lg : k >= 0; inverse ? k++ : k--) {
            u32 wi = 1;
            for (int i = 0; i < (1 << lg); i += (1 << k + 1)) {
                for (int j = 0; j < (1 << k); j++) {
                    butterfly_x2<inverse>(data[i + j], data[i + (1 << k) + j], wi);
                }
                wi = mul(wi, (inverse ? wrd : wd)[__builtin_ctz(~i >> k + 1)]);
            }
        }
        if (inverse) {
            u32 f = power(mod + 1 >> 1, lg);
```

```
        for (int i = 0; i < (1 << lg); i++) {
            data[i] = mul(data[i], f);
        }
    }
}

void dot(int n, const u32 *a, const u32 *b, u32 *c) const {
    for (int i = 0; i < n; i++) {
        c[i] = mul(a[i], b[i]);
    }
}

std::vector<u32> poly_mul(int n, std::vector<u32> a, std::vector<u32> b) {
    int lg = __builtin_ctz(bit_ceil(std::max<int>(1, (int)a.size() + (int)b.size() - 1)));
    a.resize(1 << lg), b.resize(1 << lg);
    transform<false>(lg, a.data());
    transform<false>(lg, b.data());
    dot(1 << lg, a.data(), b.data(), a.data());
    transform<true>(lg, a.data());

    a.resize(n);
    return a;
}

std::vector<u32> inv(int n, std::vector<u32> vec) {
    assert(vec.size() && vec[0] != 0);
    std::vector<u32> res = {power(vec[0], mod - 2)};

    std::vector<u32> tmp;
    for (int k = 0; (1 << k) < n; k++) {
        int m = 1 << k;

        res.resize(4 * m);
        tmp.assign(4 * m, 0);
        std::copy(vec.begin(), vec.begin() + std::min<int>(vec.size(), 2 * m), tmp.begin());

        transform<false>(k + 2, res.data());
        transform<false>(k + 2, tmp.data());
        for (int i = 0; i < (1 << k + 2); i++) {
            res[i] = mul(res[i], add(2, mod - mul(res[i], tmp[i])));
        }
        transform<true>(k + 2, res.data());
        res.resize(1 << k + 1);
    }

    res.resize(n);
    return res;
}

std::vector<u32> deriv(std::vector<u32> vec) {
    assert(vec.size() > 0);
    for (int i = 1; i < vec.size(); i++) {
        vec[i - 1] = mul(vec[i], i);
    }
    vec.pop_back();
    return vec;
}

std::vector<u32> integ(std::vector<u32> vec) {
    static std::vector<u32> inv = {0};
    while (inv.size() <= vec.size()) {
        inv.push_back(power(inv.size(), mod - 2));
    }

    vec.push_back(0);
    for (int i = vec.size() - 1; i >= 1; i--) {
        vec[i] = mul(vec[i - 1], inv[i]);
    }
    vec[0] = 0;
    return vec;
}

std::vector<u32> ln(int n, std::vector<u32> vec) {
    assert(vec[0] == 1);
    return integ(poly_mul(n - 1, deriv(vec), inv(n - 1, vec)));
}

std::vector<u32> sub(const std::vector<u32> &a, const std::vector<u32> &b) {
    std::vector<u32> res(std::max(a.size(), b.size()));
    for (int i = 0; i < res.size(); i++) {
        res[i] = add(((i < a.size()) ? a[i] : 0), mod - ((i < b.size()) ? b[i] : 0));
    }
    return res;
}

std::vector<u32> exp(int n, std::vector<u32> vec) {
    std::vector<u32> res = {1};
    for (int k = 0; (1 << k) < n; k++) {
        int m = 1 << k;
        std::vector<u32> tmp(vec.begin(), vec.begin() + std::min<int>(vec.size(), 1 << k + 1));
        // res = sub(res, poly_mul(2 * m, res, sub(ln(2 * m, res), tmp)));
        // res = sub(res, poly_mul(2 * m, res, sub(ln(2 * m, res), tmp)));
        res = poly_mul(2 * m, res, sub({1}, sub(ln(2 * m, res), tmp)));
    }
    return res;
}
} ntt;

std::vector<u32> inverse = {0};
std::vector<u32> factorial = {1}, inv_factorial = {1};

void expand(int n) {
    while (inverse.size() <= n) {
        int k = inverse.size();
        inverse.push_back(power(k, mod - 2));
        factorial.push_back(mul(factorial.back(), k));
        inv_factorial.push_back(mul(inv_factorial.back(), inverse[k]));
    }
}

int C(int n, int k) {
    return mul(factorial[n], mul(inv_factorial[k], inv_factorial[n - k]));
}
```

7 simd binsearch int32

```
#pragma GCC target("avx2,bmi,bmi2")
#include <immintrin.h>
#include <stdint.h>

#include <algorithm>
```

```
#include <cassert>
#include <numeric>

using i256 = __m256i;

int lower_bound_epi32(i256 vec, i256 val) {
    i256 cmp = _mm256_cmpgt_epi32(val, vec);
    uint32_t mask = _mm256_movemask_ps((__m256)cmp);
    return __builtin_ctz(~mask);
}

int lower_bound_epi32(i256 vec0, i256 vec1, i256 val) {
    i256 cmp0 = _mm256_cmpgt_epi32(val, vec0), cmp1 = _mm256_cmpgt_epi32(val, vec1);
    uint32_t mask = _mm256_movemask_ps((__m256)cmp0) | _mm256_movemask_ps((__m256)cmp1) << 8;
    return __builtin_ctz(~mask);
}
int div_up(int a, int b) {
    return (a - 1) / b + 1;
}

struct Meow {
    static constexpr int K = 16;

    int n;
    int lg;
    const int* data;
    const int** meow;

    Meow() = default;

    // data should be padded with at least 64 bytes
    [[gnu::noinline]] Meow(int n, const int* data) : n(n), data(data) {
        if (n <= K) {
            lg = 0;
            return;
        }
        int m = div_up(n + 1, K + 1);
        for (lg = 0; m > 1; m = div_up(m, K + 1), lg++) {
            ;
        }
        meow = new const int*[lg];
        m = div_up(n + 1, K + 1);
        for (int64_t i = lg - 1, f = 1; i >= 0; i--) {
            int b_cnt = div_up(m, K + 1);
            f *= (K + 1);
            int* ptr = (int*)_mm_malloc(4 * K * b_cnt, 64);

            for (int j = 0; j < b_cnt; j++) {
                for (int t = 0; t < K; t++) {
                    int64_t ind = j * f * (K + 1) + f * t + f - 1;
                    if (ind >= n) {
                        ptr[j * K + t] = std::numeric_limits<int>::max();
                    } else {
                        ptr[j * K + t] = data[ind];
                    }
                }
            }
            meow[i] = ptr;
            m = b_cnt;
        }
        assert(m == 1);
    }

    // [[gnu::noinline]]
    int lower_bound(int val) {
        int ind = 0;
        i256 vec_val = _mm256_set1_epi32(val);
        for (int i = 0; i < lg; i++) {
            i256 vec0 = _mm256_load_si256((__m256i*)(meow[i] + ind * K));
            i256 vec1 = _mm256_load_si256((__m256i*)(meow[i] + ind * K + 8));
            int dt = lower_bound_epi32(vec0, vec1, vec_val);
            ind = ind * (K + 1) + dt;
        }
        {
            // assert(0 <= ind && ind * (K + 1) <= n);
            i256 vec0 = _mm256_loadu_si256((__m256i*)(data + ind * (K + 1)));
            i256 vec1 = _mm256_loadu_si256((__m256i*)(data + ind * (K + 1) + 8));
            int dt = lower_bound_epi32(vec0, vec1, vec_val);
            ind = ind * (K + 1) + dt;

            ind = std::min(ind, n);
        }
        return ind;
    }
};
```

8 Prime test + Rho factorization

```
#include <bits/stdc++.h>

using u64 = uint64_t;
using u128 = __uint128_t;

u64 add(u64 a, u64 b, u64 mod) {
    return a + b - mod * (a + b >= mod);
}
u64 mul(u64 a, u64 b, u64 mod) {
    return u128(a) * b % mod;
}
u64 power(u64 b, u64 e, u64 mod) {
    u64 r = 1;
    for (; e > 0; e >>= 1) {
        if (e & 1) {
            r = mul(r, b, mod);
        }
        b = mul(b, b, mod);
    }
    return r;
}

u64 is_prime(u64 val) {
    if (val % 2 == 0) {
        return val == 2;
    }
    int lg = __builtin_ctzll(val - 1);
    for (u64 a : std::array{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 39, 41}) {
        u64 pw = power(a, val - 1 >> lg, val);
        for (int i = 0; i < lg && pw != 1; i++) {
            u64 pw2 = mul(pw, pw, val);
            if (i + 1 == lg && pw2 != 1) {
                return false;
```

```
            }
            if (pw2 == 1 && pw != val - 1) {
                return false;
            }
            pw = pw2;
        }
    }
    return true;
}

u64 find_divisor(u64 val) {
    for (int i = 2; i <= 1000; i++) {
        if (val % i == 0) {
            if (val == i) {
                return 0;
            }
            return i;
        }
    }
    if (val <= 1e6 || is_prime(val)) {
        return 0;
    }
    auto f = [&](u64 x) {
        return add(mul(x, x, val), 3, val);
    };
    static std::mt19937_64 rnd;
    u64 a = rnd() % val;
    u64 b = a;
    std::vector<int64_t> vec;
    for (int64_t it = 0;; it++) {
        a = f(a);
        b = f(f(b));
        u64 diff = std::max(a, b) - std::min(a, b);
        vec.push_back(diff);
        if (vec.size() >= 200) {
            u64 prod = 1;
            for (u64 i : vec) {
                if (prod != 0) {
                    prod = mul(prod, i, val);
                }
            }
            if (std::gcd(prod, val) != 1) {
                for (u64 i : vec) {
                    if (std::gcd(i, val) != 1) {
                        return std::gcd(i, val);
                    }
                }
                assert(false);
            }
            vec.clear();
        }
    }
}
```

9 Flow

```
#include <bits/stdc++.h>

constexpr int64_t inf = 1e18;

struct Flow {
    struct Edge {
        int to, id;
        int64_t fl, cp;
    };

    int n, S, T;
    int64_t flow;
    std::vector<std::vector<Edge>> gr;
    std::vector<int> dist, used;

    Flow(int n, int S, int T) : n(n), S(S), T(T), flow(0) {
        gr.assign(n, {});
    }

    void add_edge(int u, int v, int64_t cp) {
        int a = gr[u].size(), b = gr[v].size();
        gr[u].push_back({v, b, 0, cp});
        gr[v].push_back({u, a, cp, cp});
    }

    void bfs(int s) {
        std::deque<int> deque;
        dist.assign(n, n);
        deque.push_back(s);
        dist[s] = 0;
        while (!deque.empty()) {
            int v = deque.front();
            deque.pop_front();
            for (auto e : gr[v]) {
                if (e.fl != e.cp) {
                    if (dist[e.to] == n) {
                        dist[e.to] = dist[v] + 1;
                        deque.push_back(e.to);
                    }
                }
            }
        }
    }

    int64_t dfs(int v, int64_t min) {
        if (v == T || min == 0) {
            return min;
        }
        for (int &i = used[v]; i < gr[v].size(); i++) {
            auto &e = gr[v][i];
            if (e.fl != e.cp && dist[e.to] == dist[v] + 1) {
                if (int64_t dt = dfs(e.to, std::min(min, e.cp - e.fl)); dt) {
                    e.fl += dt;
                    gr[e.to][e.id].fl -= dt;
                    return dt;
                }
            }
        }
        return 0;
    }

    int64_t find_max_flow() {
        while (bfs(S), dist[T] != n) {
            used.assign(n, 0);
            while (int64_t dlt = dfs(S, inf)) {
```

```
                flow += dlt;
            }
        }
        return flow;
    }
};
```

10 simd merge + compress

```
#include <bits/stdc++.h>
#include <immintrin.h>

__m256i compress_epi32(__m256i val, __m256i mask) {
    uint32_t i32 = _mm256_movemask_epi8(mask);
    constexpr uint32_t identity = 0x76543210;
    i32 = _pext_u32(identity, i32);
    constexpr uint64_t expand = 0x0F0F0F0F0F0F0F0Full;
    uint64_t i64 = _pdep_u64(i32, expand);
    __m128i vec = _mm_cvtsi64_si128((int64_t)i64);
    __m256i perm = _mm256_cvtepu8_epi32(vec);

    return _mm256_permutevar8x32_epi32(val, perm);
}

inline void merge_epi32(__m256i& a, __m256i& b) {
    {
        static const __m256i xr = _mm256_set_epi32(-1, -1, -1, -1, 0, 0, 0, 0);
        const __m256i rev_perm = _mm256_set_epi32(0, 1, 2, 3, 4, 5, 6, 7);
        b = _mm256_permutevar8x32_epi32(b, rev_perm);
        __m256i vcmp = _mm256_cmpgt_epi32(a, b);
        vcmp = _mm256_xor_si256(vcmp, xr);
        __m256i tmp_a = _mm256_blendv_epi8(a, b, vcmp);
        __m256i tmp_b = _mm256_blendv_epi8(b, a, vcmp);
        a = tmp_a;
        b = tmp_b;
        b = _mm256_permute2x128_si256(b, a, 0b0000'0001);
    }
    {
        static const __m256i xr = _mm256_set_epi32(-1, -1, 0, 0, -1, -1, 0, 0);
        __m256i vcmp = _mm256_cmpgt_epi32(a, b);
        vcmp = _mm256_xor_si256(vcmp, xr);
        __m256i tmp_a = _mm256_blendv_epi8(a, b, vcmp);
        __m256i tmp_b = _mm256_blendv_epi8(b, a, vcmp);
        a = tmp_a;
        b = tmp_b;
        b = _mm256_shuffle_epi32(b, _MM_SHUFFLE(1, 0, 3, 2));
    }
    {
        static const __m256i xr = _mm256_set_epi32(-1, 0, -1, 0, -1, 0, -1, 0);
        __m256i vcmp = _mm256_cmpgt_epi32(a, b);
        vcmp = _mm256_xor_si256(vcmp, xr);
        __m256i tmp_a = _mm256_blendv_epi8(a, b, vcmp);
        __m256i tmp_b = _mm256_blendv_epi8(b, a, vcmp);
        a = tmp_a;
        b = tmp_b;
        b = _mm256_shuffle_epi32(b, _MM_SHUFFLE(2, 3, 0, 1));
    }
    {
        // static const __m256i xr = _mm256_set_epi32(-1, 0, -1, 0, -1, 0, -1, 0);
        __m256i vcmp = _mm256_cmpgt_epi32(a, b);
        // vcmp = _mm256_xor_si256(vcmp, xr);
        __m256i tmp_a = _mm256_blendv_epi8(a, b, vcmp);
        __m256i tmp_b = _mm256_blendv_epi8(b, a, vcmp);
        a = tmp_a;
        b = tmp_b;
    }
    {
        __m256i tmp_a = _mm256_unpacklo_epi32(a, b);
        __m256i tmp_b = _mm256_unpackhi_epi32(a, b);
        // a = tmp_a;
        // b = tmp_b;
        a = _mm256_permute2x128_si256(tmp_a, tmp_b, 0b0010'0000);
        b = _mm256_permute2x128_si256(tmp_a, tmp_b, 0b0011'0001);
    }
}
```

11 tree bitset

```
#include <iostream>
#pragma GCC optimize("O3")
#pragma GCC target("avx2,lzcnt,bmi,bmi2")
#include <bits/stdc++.h>

template <typename u_tp = uint64_t>
class TreeBitset {
   private:
    static constexpr size_t B = sizeof(u_tp) * 8;

    std::vector<u_tp> my_data;
    std::vector<u_tp*> data;
    size_t n, lg;

   public:
    TreeBitset(size_t n = 0) {
        assign(n);
    }

    void assign(size_t n) {
        this->n = n;
        size_t m = n + 2;
        std::vector<size_t> vec;
        while (m > 1) {
            m = (m - 1) / B + 1;
            vec.push_back(m);
        }
        std::reverse(vec.begin(), vec.end());

        lg = vec.size();
        data.resize(vec.size());
        size_t sum = std::accumulate(vec.begin(), vec.end(), size_t(0));
        my_data.assign(sum, 0);
        for (size_t i = 0, s = 0; i < lg; s += vec[i], i++) {
            data[i] = my_data.data() + s;
        }

        for (size_t i = 0, k = lg; k--; i /= B) {
            data[k][i / B] |= u_tp(1) << i % B;
        }
        for (size_t i = n + 1, k = lg; k--; i /= B) {
            data[k][i / B] |= u_tp(1) << i % B;
```

```
    }
}

size_t size() const {
    return n;
}

void clear() {
    my_data.assign(my_data.size(), 0);
    for (size_t i = 0, k = lg; k--; i /= B) {
        data[k][i / B] |= u_tp(1) << i % B;
    }
    for (size_t i = n + 1, k = lg; k--; i /= B) {
        data[k][i / B] |= u_tp(1) << i % B;
    }
}

// i must be in [0, n)
bool insert(size_t i) {
    i++;
    if ((data[lg - 1][i / B] >> i % B) & 1) {
        return false;
    }
    for (size_t k = lg; k--; i /= B) {
        data[k][i / B] |= u_tp(1) << i % B;
    }
    return true;
}

// i must be in [0, n)
bool erase(size_t i) {
    i++;
    if (!((data[lg - 1][i / B] >> i % B) & 1)) {
        return false;
    }
    data[lg - 1][i / B] ^= u_tp(1) << i % B;
    i /= B;
    for (size_t k = lg - 1; k > 0 && !data[k][i]; k--, i /= B) {
        data[k - 1][i / B] ^= u_tp(1) << i % B;
    }
    return true;
}

// i must be in [0, n)
bool contains(size_t i) const {
    i++;
    return (data[lg - 1][i / B] >> i % B) & 1;
}

// i must be in [0, n]
// smallest element greater than or equal to i, n if doesn't exist
size_t find_next(size_t i) const {
    i++;
    size_t k = lg - 1;

    for (; !u_tp(data[k][i / B] >> i % B); k--) {
        i = i / B + 1;
    }

    for (; k < lg; k++) {
        u_tp mask = u_tp(data[k][i / B] >> i % B) << i % B;
        size_t ind = std::countr_zero(mask);
        i = (i / B * B + ind) * B;
    }
    i /= B;
    return i - 1;
}

// i must be in [0, n)
// largest element less than or equal to i, n if doesn't exist
size_t find_prev(size_t i) const {
    i++;
    size_t k = lg - 1;
    for (; !u_tp(data[k][i / B] << (B - i % B - 1)); k--) {
        i = i / B - 1;
    }

    for (; k < lg; k++) {
        u_tp mask = u_tp(data[k][i / B] << (B - i % B - 1)) >> (B - i % B - 1);
        assert(mask);
        size_t ind = B - 1 - std::countl_zero(mask);
        i = (i / B * B + ind) * B + (B - 1);
    }
    i /= B;
    if (i == 0) {
        return n;
    }
    return i - 1;
}
};
```

Извлечение квадратного корня по простому модулю

Алгоритм:

`input:` $3 \leqslant p$ — простое, $1 \leqslant a < p$

`output:` $1 \leqslant x < p$ такой, что $x^2 = a$ или -1, если $\nexists x$

1. Если $a^{\frac{p-1}{2}} \neq 1$ вернуть $x = -1$
2. Пока корень не найден:
3. $\quad i$:= `random` $[1..p-1]$
4. $\quad T(x) := (x+i)^{\frac{p-1}{2}} - 1 \mod (x^2 - a) = b \cdot x + c$ (многочлены от x)
5. $\quad$ Если $b \neq 0$, то вернуть $x = c \cdot b^{-1} = c \cdot b^{p-2}$