Contents

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 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";
    };
};
```

3 floor sum + 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);
}
```

4 primality 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();
        }
    }
}
```

5 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;
}
};
```

6 link cut

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

struct LinkCut {
    struct Node {
        int next[2], prev, sz, flip;
        Node() : next{-1, -1}, prev(-1), sz(1), flip(0) { ; }
    };

    std::vector<Node> data;

    LinkCut(int n = 0) : data(n) { ; }

    int get_sz(int i) { return i == -1 ? 0 : data[i].sz; }
    void set_prev(int i, int p) { i != -1 ? data[i].prev = p : 0; }
    void flip(int i) { i != -1 ? data[i].flip ^= 1 : 0; }
    void pull(int i) { data[i].sz = get_sz(data[i].next[0]) + 1 + get_sz(data[i].next[1]); }
    void push(int i) {
        data[i].flip ? (flip(data[i].next[0]), flip(data[i].next[1]),
                        std::swap(data[i].next[0], data[i].next[1]), data[i].flip = false)
                     : 0;
    }
```

```
    int ch_num(int i) {
        return data[i].prev == -1
                   ? -1
                   : (data[data[i].prev].next[0] == i ? 0 : (data[data[i].prev].next[1] == i ? 1 : -1));
    }

    void rotate(int i) {
        int j = data[i].prev, k = data[j].prev, ni = ch_num(i), nj = ch_num(j);
        data[j].next[ni] = data[i].next[!ni], set_prev(data[i].next[!ni], j);
        data[i].next[!ni] = j, data[i].prev = k, data[j].prev = i, pull(j), pull(i);
        if (nj != -1) data[k].next[nj] = i;
    }

    void splay(int i) {
        push(i);
        while (ch_num(i) != -1) {
            int j = data[i].prev, k = data[j].prev;
            if (ch_num(j) == -1) {
                push(j), push(i), rotate(i);
            } else {
                push(k), push(j), push(i), rotate(ch_num(i) == ch_num(j) ? j : i), rotate(i);
            }
        }
    }

    void expose(int i) {
        splay(i), data[i].next[1] = -1, pull(i);
        while (data[i].prev != -1) {
            splay(data[i].prev), data[data[i].prev].next[1] = i, pull(data[i].prev), splay(i);
        }
    }

    void ch_root(int i) { expose(i), flip(i); }
    void link(int i, int j) { ch_root(j), data[j].prev = i; }
    void cut(int i, int j) { ch_root(i), expose(i), splay(j), data[j].prev = -1; }
    int get_dist(int i, int j) {
        return i == j ? 0 : (ch_root(i), expose(j), (data[i].prev != -1 ? data[j].sz - 1 : -1));
    }
};
```

7 simplex

```
// src: https://github.com/dacin21/dacin21_codebook/blob/master/numerical/simplex_lp.cpp

// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
// OUTPUT: value of the optimal solution (inf if unbounded
//         above, -inf if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

// 2018-05: added steepest edge pricing, speed up factor of ~5

template <typename DOUBLE>
struct Simplex_Steep {
    using VD = vector<DOUBLE>;
    using VVD = vector<VD>;
    using VI = vector<int>;
    const DOUBLE EPS = 1e-12;
    int m, n;
    VI B, N;
    VVD D;

    int iteration_cnt = 0;

    Simplex_Steep(const VVD& A, const VD& b, const VD& c) : m(b.size()), n(c.size()), B(m), N(n + 1), D(m + 2, VI
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++) D[i][j] = A[i][j];
        for (int i = 0; i < m; i++) {
            B[i] = n + i;
            D[i][n] = -1;
            D[i][n + 1] = b[i];
        }
        for (int j = 0; j < n; j++) {
            N[j] = j;
            D[m][j] = -c[j];
        }
        N[n] = -1;
        D[m + 1][n] = 1;
    }

    void Pivot(int r, int s) {
        for (int i = 0; i < m + 2; i++)
            if (i != r)
                for (int j = 0; j < n + 2; j++)
                    if (j != s)
                        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
        for (int j = 0; j < n + 2; j++)
            if (j != s) D[r][j] /= D[r][s];
        for (int i = 0; i < m + 2; i++)
            if (i != r) D[i][s] /= -D[r][s];
        D[r][s] = 1.0 / D[r][s];
        swap(B[r], N[s]);
    }

    bool Simplex(int phase) {
        int x = phase == 1 ? m + 1 : m;
        while (true) {
            ++iteration_cnt;
            int s = -1;
            DOUBLE c_val = -1;
            for (int j = 0; j <= n; j++) {
                if (phase == 2 && N[j] == -1) continue;
                DOUBLE norm_sq = 0;
                for (int k = 0; k <= m; ++k) {
                    norm_sq += D[k][j] * D[k][j];
                }
                if (norm_sq < EPS) norm_sq = EPS;  // stop division by 0
                DOUBLE c_val_j = D[x][j] / sqrtl(norm_sq);
                if (s == -1 || c_val_j < c_val || (c_val == c_val_j && N[j] < N[s])) {
                    s = j;
                    c_val = c_val_j;
                }
            }
            if (D[x][s] >= -EPS) return true;
            int r = -1;
            for (int i = 0; i < m; i++) {
```

```
                if (D[i][s] <= EPS) continue;
                if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
                    (D[i][n + 1] / D[i][s] == D[r][n + 1] / D[r][s] && B[i] < B[r])) r = i;
            }
            if (r == -1) return false;
            Pivot(r, s);
        }
    }

    DOUBLE solve(VD& x) {
        int r = 0;
        for (int i = 1; i < m; i++)
            if (D[i][n + 1] < D[r][n + 1]) r = i;
        if (D[r][n + 1] <= -EPS) {
            Pivot(r, n);
            if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric_limits<DOUBLE>::infinity();
            for (int i = 0; i < m; i++)
                if (B[i] == -1) {
                    int s = -1;
                    for (int j = 0; j <= n; j++)
                        if (s == -1 || D[i][j] < D[i][s] || (D[i][j] == D[i][s] && N[j] < N[s])) s = j;
                    Pivot(i, s);
                }
        }
        if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
        x = VD(n);
        for (int i = 0; i < m; i++)
            if (B[i] < n) x[B[i]] = D[i][n + 1];
        // cerr << "Steepest edge iterations: " << iteration_cnt << "\n";
        return D[m][n + 1];
    }
};
```

8 suffix array

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

std::vector<int> sa_is(const std::vector<int>& input) {
    if (input.size() <= 1) {
        return std::vector<int>(input.size(), 0);
    }

    int n = input.size();
    int mx = *std::max_element(input.begin(), input.end()) + 1;

    std::cerr << "n: " << n << "   mx: " << mx << "\n";

    // assert(mx <= n);
    assert(input.back() < *std::min_element(input.begin(), input.end() - 1));

    std::vector<bool> tp(n + 1);
    tp[n] = true;
    for (int i = n - 2; i >= 0; i--) {
        tp[i] = input[i] == input[i + 1] ? tp[i + 1] : (input[i] < input[i + 1]);
    }

    std::vector<int> suf(n + 1, -1);
    std::vector<int> bck(mx + 1);

    for (int val : input) bck[val] += 1;
    std::exclusive_scan(bck.begin(), bck.end(), bck.begin(), 1);

    std::vector<int> ptr(mx + 1);

    std::copy(bck.begin(), bck.end(), ptr.begin());
    suf[0] = n;
    for (int i = 1; i < n; i++) {
        if (!tp[i - 1] && tp[i]) {
            suf[--ptr[input[i] + 1]] = i;
        }
    }

    auto induced_sort = [&] {
        std::copy(bck.begin(), bck.end(), ptr.begin());
        for (int i = 0; i <= n; i++) {
            int p = suf[i];
            if (p > 0 && !tp[p - 1]) {
                suf[ptr[input[p - 1]]++] = p - 1;
            }
        }

        std::copy(bck.begin(), bck.end(), ptr.begin());
        for (int i = n; i >= 0; i--) {
            int p = suf[i];
            if (p > 0 && tp[p - 1]) {
                suf[--ptr[input[p - 1] + 1]] = p - 1;
            }
        }
    };

    induced_sort();

    int m = 0;
    std::vector<int> lms_pos(n + 1, -1);
    for (int i = 1; i <= n; i++) {
        if (!tp[i - 1] && tp[i]) {
            lms_pos[i] = m++;
        }
    }

    std::vector<int> input2(m);
    for (int i = 0, cnt = 0, last = -1; i <= n; i++) {
        int p = suf[i];
        if (p > 0 && !tp[p - 1] && tp[p]) {
            if (last != -1) {
                if (last == n) {
                    cnt++;
                } else {
                    for (int j = 0;; j++) {
                        if (p + j == n || last + j == n || input[p + j] != input[last + j]) {
                            cnt++;
                            break;
                        }
                        if (j != 0 && !tp[p + j - 1] && tp[p + j]) {
                            break;
                        }
                    }
                }
            }
            last = p;
            input2[lms_pos[p]] = cnt;
```

```
        }
    }

    std::vector<int> suf2;
    if (*std::max_element(input2.begin(), input2.end()) == m - 1) {
        suf2.assign(m, -1);
        for (int i = 0; i < m; i++) {
            suf2[input2[i]] = i;
        }
    } else {
        suf2 = sa_is(input2);
    }

    std::vector<int> ind;
    ind.swap(lms_pos), ind.clear();
    for (int i = 1; i <= n; i++) {
        if (!tp[i - 1] && tp[i]) {
            ind.push_back(i);
        }
    }
    ind.push_back(n);

    std::fill(suf.begin(), suf.end(), -1);

    std::copy(bck.begin(), bck.end(), ptr.begin());
    suf[0] = n;
    for (int i = m - 1; i > 0; i--) {
        int p = ind[suf2[i]];
        suf[--ptr[input[p] + 1]] = p;
    }

    induced_sort();

    suf.erase(suf.begin());

    return suf;
}

std::vector<int> suffix_array(std::vector<int> input) {
    if (input.size() <= 1) {
        return std::vector<int>(input.size(), 0);
    }

    int n = input.size();
    int mx = *std::max_element(input.begin(), input.end()) + 1;
    if (mx > n) {
        std::vector<int> fuck(mx, -1);
        for (int i = 0; i < n; i++) {
            fuck[input[i]] = 0;
        }
        int mx2 = 0;
        for (int i = 0; i < mx; i++) {
            if (fuck[i] != -1) {
                fuck[i] = mx2++;
            }
        }
        for (int i = 0; i < n; i++) {
            input[i] = fuck[input[i]];
        }
        mx = mx2;
    }
    if (std::all_of(input.begin(), input.end() - 1, [&](int val) { return val > input.back(); })) {
        return sa_is(input);
    } else {
        for (auto& i : input) {
            i++;
        }
        input.push_back(0);
        auto suf = sa_is(input);
        suf.erase(suf.begin());
        return suf;
    }
}
```

9 dom tree

```
// src: https://judge.yosupo.jp/submission/233128

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

struct dominator_tree {
    int n, t;
    vector<basic_string<int>> g, rg, bucket;
    vector<int> arr, par, rev, sdom, dom, dsu, label;

    dominator_tree(int n)
        : g(n),
          rg(n),
          bucket(n),
          arr(n, -1),
          par(n),
          rev(n),
          sdom(n),
          dom(n),
          dsu(n),
          label(n),
          n(n),
          t(0) {}

    void add_edge(int u, int v) { g[u] += v; }

    void dfs(int u) {
        arr[u] = t;
        rev[t] = u;
        label[t] = sdom[t] = dsu[t] = t;
        t++;
        for (int w : g[u]) {
            if (arr[w] == -1) {
                dfs(w);
                par[arr[w]] = arr[u];
            }
            rg[arr[w]] += arr[u];
        }
    }

    int find(int u, int x = 0) {
        if (u == dsu[u]) return x ? -1 : u;
        int v = find(dsu[u], x + 1);
        if (v < 0) return u;
```

```
        if (sdom[label[dsu[u]]] < sdom[label[u]]) label[u] = label[dsu[u]];
        dsu[u] = v;
        return x ? v : label[u];
    }

    vector<int> run(int root) {
        dfs(root);
        iota(dom.begin(), dom.end(), 0);
        for (int i = t - 1; i >= 0; i--) {
            for (int w : rg[i]) sdom[i] = min(sdom[i], sdom[find(w)]);
            if (i) bucket[sdom[i]] += i;
            for (int w : bucket[i]) {
                int v = find(w);
                if (sdom[v] == sdom[w])
                    dom[w] = sdom[w];
                else
                    dom[w] = v;
            }
            if (i > 1) dsu[i] = par[i];
        }
        for (int i = 1; i < t; i++) {
            if (dom[i] != sdom[i]) dom[i] = dom[dom[i]];
        }
        vector<int> outside_dom(n);
        iota(begin(outside_dom), end(outside_dom), 0);
        for (int i = 0; i < t; i++) outside_dom[rev[i]] = rev[dom[i]];
        return outside_dom;
    }
};

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    int n, m, r;
    cin >> n >> m >> r;
    dominator_tree d(n);
    while (m--) {
        int x, y;
        cin >> x >> y;
        d.add_edge(x, y);
    }
    auto v = d.run(r);
    for (int i = 0; i < n; i++) {
        if (i != r && v[i] == i)
            cout << "-1 ";
        else
            cout << v[i] << ' ';
    }
    return 0;
}
```

10 poly

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

using u32 = uint32_t;
using u64 = uint64_t;

template <u32 mod>
struct MintT {
private:
 u32 m_val;

 static u32 add(u32 a, u32 b) { return a + b - mod * (a + b >= mod); }
 static u32 mul(u32 a, u32 b) { return u64(a) * b % mod; }

public:
 MintT() : m_val(0) { ; }
 MintT(int64_t x) : m_val((x % mod + mod) % mod) { ; }

 static MintT from_u32_unchecked(u32 val) {
     MintT m;
     m.m_val = val;
     return m;
 }

 MintT& operator+=(const MintT& other) { return m_val = add(m_val, other.m_val), *this; }
 MintT& operator-=(const MintT& other) { return m_val = add(m_val, mod - other.m_val), *this; }
 MintT& operator*=(const MintT& other) { return m_val = mul(m_val, other.m_val), *this; }

 MintT operator-() const { return MintT() - *this; }
 friend MintT operator+(MintT a, const MintT& b) { return MintT(a += b); }
 friend MintT operator-(MintT a, const MintT& b) { return MintT(a -= b); }
 friend MintT operator*(MintT a, const MintT& b) { return MintT(a *= b); }

 MintT power(u64 exp) const {
     MintT r = 1, b = *this;
     for (; exp; exp >>= 1) {
         if (exp & 1) r *= b;
         b *= b;
     }
     return r;
 }

 MintT inverse() const {
     assert(m_val != 0);
     return power(mod - 2);
 }
 static std::vector<MintT> bulk_inverse(const std::vector<MintT>& vec) {
     std::vector<MintT> res(vec.size(), 1);
     MintT val = 1;
     for (int i = 0; i < vec.size(); i++) {
         res[i] *= val, val *= vec[i];
     }
     val = val.inverse();
     for (int i = 0; i < vec.size(); i++) {
         res.rbegin()[i] *= val, val *= vec.rbegin()[i];
     }
     return res;
 }
 u32 get_value() const { return m_val; }

 friend bool operator!=(const MintT& a, const MintT& b) { return a.m_val != b.m_val; }
 friend bool operator==(const MintT& a, const MintT& b) { return a.m_val == b.m_val; }

 friend std::istream& operator>>(std::istream& in, MintT& x) {
     int64_t val;
     in >> val;
     x = MintT(val);
     return in;
 }
```

```
    friend std::ostream& operator<<(std::ostream& out, const MintT& x) {
        return out << x.get_value();
    }
};

template <u32 mod>
class NTT {
   public:
    using mint = MintT<mod>;

   private:
    mint pr_root;
    std::vector<mint> wd, wrd, w_rt, wr_rt;

    static u32 find_pr_root() {
        std::vector<u32> vec;
        u32 val = mod - 1;
        for (u64 i = 2; i * i <= val; i++) {
            if (val % i == 0) {
                vec.push_back(i);
                do {
                    val /= i;
                } while (val % i == 0);
            }
        }
        if (val != 1) {
            vec.push_back(val);
        }
        for (u32 i = 2; i < mod; i++) {
            if (std::all_of(vec.begin(), vec.end(),
                            [&](u32 q) { return mint(i).power((mod - 1) / q) != 1; })) {
                return i;
            }
        }
        assert(false && "pr_root not found");
    }

   public:
    NTT() : pr_root(find_pr_root()) {
        int lg = __builtin_ctz(mod - 1);
        wd.assign(lg, 0), wrd.assign(lg, 0);
        w_rt.assign(lg - 1, 0), wr_rt.assign(lg - 1, 0);
        for (int k = 0; k + 2 <= lg; k++) {
            mint a = pr_root.power(mod - 1 >> k + 2);
            mint b = pr_root.power((mod - 1 >> k + 2) * ((2 << k) - 2));
            w_rt[k] = a, wr_rt[k] = a.inverse();
            wd[k] = a * b.inverse(), wrd[k] = a.inverse() * b;
        }
    }

   private:
    template <bool inverse>
    static void butterfly_x2(mint& a, mint& b, mint w) {
        mint x = a, y = b;
        if (!inverse) {
            y *= w, a = x + y, b = x - y;
        } else {
            a = x + y, b = (x - y) * w;
        }
    }

   public:
    template <bool inverse, bool right_part = false>
    void transform(int lg, mint* data) const {
        for (int k = inverse ? 0 : lg - 1; inverse ? k < lg : k >= 0; inverse ? k++ : k--) {
            mint wi = right_part ? (inverse ? wr_rt : w_rt)[lg - k - 1] : mint(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 *= (inverse ? wrd : wd)[__builtin_ctz(~i >> k + 1)];
            }
        }
        if (inverse) {
            mint inv = mint(mod + 1 >> 1).power(lg);
            for (int i = 0; i < (1 << lg); i++) {
                data[i] *= inv;
            }
        }
    }

    void expand_ntt(int lg, mint* data) const {
        std::copy(data, data + (1 << lg), data + (1 << lg));
        transform<true>(lg, data + (1 << lg));
        transform<false, true>(lg, data + (1 << lg));
    }

    void extract_cum(int lg, mint* data, bool odd = false) const {
        const mint inv2 = mint(mod + 1 >> 1);
        if (!odd) {
            for (int i = 0; i < (1 << lg); i++) {
                data[i] = (data[2 * i] + data[2 * i + 1]) * inv2;
            }
        } else {
            mint wi = 1 * inv2;
            for (int i = 0; i < (1 << lg); i++) {
                data[i] = (data[2 * i] - data[2 * i + 1]) * wi;
                wi *= wrd[__builtin_ctz(~i)];
            }
        }
    }

    void convolve_cyclic(int lg, mint* a, mint* b) const {
        transform<false>(lg, a);
        transform<false>(lg, b);
        for (int i = 0; i < (1 << lg); i++) {
            a[i] *= b[i];
        }
        transform<true>(lg, a);
    }

    std::vector<mint> convolve(std::vector<mint> a, std::vector<mint> b) const {
        if (a.empty() || b.empty()) {
            return {};
        }
        int n = a.size(), m = b.size(), lg = (n == 1 && m == 1) ? 0 : 32 - __builtin_clz(n + m - 2);
        if (a.size() * b.size() < int64_t(1 << lg) * lg * 2) {
            std::vector<mint> c(n + m - 1);
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    c[i + j] += a[i] * b[j];
                }
```

```
            }
            return c;
        }
        if (lg > 0 && n + m - 1 == (1 << lg - 1) + 1) {
            mint p = a.back() * b.back();
            a.reserve((1 << lg - 1) + 1);
            a.resize(1 << lg - 1), b.resize(1 << lg - 1);
            convolve_cyclic(lg - 1, a.data(), b.data());
            a[0] -= p, a.push_back(p);
            return a;
        }
        a.resize(1 << lg), b.resize(1 << lg);
        convolve_cyclic(lg, a.data(), b.data());
        a.resize(n + m - 1);
        return a;
    }
};

namespace polynomial {
template <u32 mod>
struct Poly : public std::vector<MintT<mod>> {
   public:
    using base = std::vector<MintT<mod>>;
    using base::base, base::size, base::resize;
    using mint = MintT<mod>;

   private:
    static const NTT<mod> ntt;

   public:
    mint coeff(size_t ind) const { return ind < this->size() ? this->operator[](ind) : mint(); }

    int64_t deg() const {
        for (int64_t i = size() - 1; i >= 0; i--) {
            if (this->operator[](i) != 0) {
                return i;
            }
        }
        return -1;
    }

    friend std::ostream& operator<<(std::ostream& out, const Poly& p) {
        out << "{";
        for (int i = 0; i < p.size(); i++) {
            if (i != 0) {
                out << ", ";
            }
            out << p[i];
        }
        out << "}";
        return out;
    }

    void remove_zeros() {
        while (size() && this->back() == 0) {
            this->pop_back();
        }
    }

    friend Poly operator*(const Poly& a, const Poly& b) {
        int64_t n = a.deg(), m = b.deg();
        if (n == -1 || m == -1) {
            return {};
        }
        auto p = ntt.convolve(std::vector<mint>(a.begin(), a.begin() + n + 1),
                              std::vector<mint>(b.begin(), b.begin() + m + 1));
        Poly c(p.begin(), p.end());
        c.remove_zeros();
        return c;
    }
    Poly& operator*=(const Poly& other) { return *this = *this * other; }

    Poly operator-() const {
        Poly a = *this;
        for (int i = 0; i < a.size(); i++) {
            a[i] = -a[i];
        }
        a.remove_zeros();
        return a;
    }

    Poly& operator+=(const Poly& b) {
        resize(std::max(size(), b.size()));
        for (int i = 0; i < b.size(); i++) {
            this->operator[](i) += b[i];
        }
        remove_zeros();
        return *this;
    }
    Poly& operator-=(const Poly& b) {
        resize(std::max(size(), b.size()));
        for (int i = 0; i < b.size(); i++) {
            this->operator[](i) -= b[i];
        }
        remove_zeros();
        return *this;
    }
    friend Poly operator+(Poly a, Poly b) { return a += b; }
    friend Poly operator-(Poly a, Poly b) { return a -= b; }

    // sub  x = ax
    Poly sub_ax(mint a) const {
        mint p = 1;
        Poly res = *this;
        for (int i = 0; i < size(); i++, p *= a) {
            res[i] *= p;
        }
        return res;
    }

    Poly div_xk(size_t k) const { return Poly(this->begin() + std::min(size(), k), this->end()); }

    Poly mul_xk(size_t k) const {
        Poly a = *this;
        a.insert(a.begin(), k, 0);
        return a;
    }

    Poly mod_xk(size_t k) const { return Poly(this->begin(), this->begin() + std::min(size(), k)); }

    Poly inv_series(int n) const {
        Poly a = *this;
```

```
        a.resize(n);
        Poly b = {a.coeff(0).inverse()};
        for (int k = 0; (1 << k) < n; k++) {
            int m = 1 << k;
            Poly c = a.mod_xk(2 * m);
            b.resize(4 * m);
            c.resize(4 * m);
            ntt.template transform<false>(k + 2, b.data());
            ntt.template transform<false>(k + 2, c.data());
            for (int i = 0; i < (4 * m); i++) {
                b[i] *= 2 - b[i] * c[i];
            }
            ntt.template transform<true>(k + 2, b.data());
            b.resize(2 * m);
        }
        b.resize(n);
        return b;
    }

    Poly div(Poly b, Poly b_inv = {}) const {
        Poly a = *this;
        a.remove_zeros(), b.remove_zeros();
        assert(b.size());
        if (a.size() < b.size()) {
            return {{}, {}};
        }
        std::reverse(a.begin(), a.end()), std::reverse(b.begin(), b.end());
        size_t d = a.size() - b.size() + 1;

        if (b_inv.size() < d) {
            b_inv = b.inv_series(d);
        }

        Poly q = (a.mod_xk(d) * b_inv.mod_xk(d)).mod_xk(d);
        q.resize(d);
        std::reverse(q.begin(), q.end());
        return q;
    }
    std::pair<Poly, Poly> divmod(Poly b, const Poly& b_inv = {}) const {
        Poly q = this->div(b, b_inv);
        Poly r = *this - q * b;
        assert(r.size() < b.size());
        r.remove_zeros();
        return {q, r};
    }
    friend Poly operator/(Poly a, Poly b) { return a.div(b); }
    friend Poly operator%(const Poly& a, const Poly& b) { return a.divmod(b).second; }

    Poly power(u64 exp) const {
        if (exp == 0) {
            return Poly{1};
        } else if (exp & 1) {
            return power(exp - 1) * *this;
        } else {
            Poly a = power(exp >> 1);
            return a * a;
        }
    }
    Poly power_mod(u64 exp, const Poly& md, std::shared_ptr<Poly> md_inv = nullptr) const {
        if (exp == 0) {
            return Poly{1};
        }
        if (md_inv == nullptr || md_inv->size() < md.size()) {
            md_inv = std::make_shared<Poly>(Poly(md.rbegin(), md.rend()).inv_series(md.size()));
        }
        if (exp & 1) {
            return (power_mod(exp - 1, md, md_inv) * *this).divmod(md, *md_inv).second;
        } else {
            Poly a = power_mod(exp >> 1, md, md_inv);
            return (a * a).divmod(md, *md_inv).second;
        }
    }

    mint dot(const Poly& b) const {
        mint res = 0;
        for (size_t i = 0; i < std::min(size(), b.size()); i++) {
            res += this->operator[](i) * b[i];
        }
        return res;
    }

    Poly deriv() const {
        Poly res = *this;
        for (int i = 0; i < size(); i++) {
            res[i] *= i;
        }
        if (res.size()) {
            res.erase(res.begin());
        }
        res.remove_zeros();
        return res;
    }

    Poly integ() const {
        Poly res = *this;
        res.remove_zeros();
        mint val = 1;
        for (int i = 0; i < res.size(); i++) {
            res[i] *= val, val *= (i + 1);
        }
        val = val.inverse();
        for (int i = (int)res.size() - 1; i >= 0; i--) {
            res[i] *= val, val *= (i + 1);
        }
        res.insert(res.begin(), 0);
        res.remove_zeros();
        return res;
    }

    Poly ln(int n) const {
        if (n <= 1) {
            return Poly(n);
        }
        return (mod_xk(n).deriv() * inv_series(n - 1)).mod_xk(n - 1).integ();
    }

    Poly exp(int n) const {
        assert(coeff(0) == 0);
        Poly b = {1};
        for (int k = 0; (1 << k) < n; k++) {
            int m = 1 << k;
            // b = (b * (Poly{1} - b.ln(2 * m) + this->mod_xk(2 * m))).mod_xk(2 * m);
```

```
            Poly b2 = b;
            Poly c = Poly{1} - b.ln(2 * m) + this->mod_xk(2 * m);
            b2.resize(2 * m), c.resize(2 * m), b.resize(2 * m);
            ntt.convolve_cyclic(k + 1, b2.data(), c.data());
            for (int i = m; i < 2 * m; i++) {
                b[i] = b2[i];
            }
        }
        b.resize(n);
        return b;
    }

    std::vector<mint> evaluate(const std::vector<mint>& pts) {
        if (pts.empty()) {
            return {};
        }
        int sz = 1;
        while (sz < pts.size()) sz *= 2;

        std::vector<Poly> data(2 * sz);
        for (int i = 0; i < pts.size(); i++) {
            data[sz + i] = Poly({-pts[i], 1});
        }
        for (int i = pts.size(); i < sz; i++) {
            data[sz + i] = Poly({1});
        }
        for (int i = sz - 1; i > 0; i--) {
            data[i] = data[2 * i] * data[2 * i + 1];
        }

        data[1] = *this % data[1];
        for (int i = 2; i < 2 * sz; i++) {
            data[i] = data[i >> 1] % data[i];
        }
        std::vector<mint> res(pts.size());
        for (int i = 0; i < pts.size(); i++) {
            res[i] = data[sz + i].coeff(0);
        }
        return res;
    }
};
template <u32 mod>
const NTT<mod> Poly<mod>::ntt;

template <u32 mod>
Poly<mod> interpolate(const std::vector<MintT<mod>>& pts, const std::vector<MintT<mod>>& vals) {
    assert(pts.size() == vals.size());
    if (pts.empty()) {
        return Poly<mod>{};
    }

    int sz = 1;
    while (sz < pts.size()) sz *= 2;

    std::vector<Poly<mod>> data(2 * sz);
    for (int i = 0; i < pts.size(); i++) {
        data[sz + i] = Poly<mod>({-pts[i], 1});
    }
    for (int i = pts.size(); i < sz; i++) {
        data[sz + i] = Poly<mod>({1});
    }
    for (int i = sz - 1; i > 0; i--) {
        data[i] = data[2 * i] * data[2 * i + 1];
    }

    std::vector<MintT<mod>> d = data[1].deriv().evaluate(pts);
    d = MintT<mod>::bulk_inverse(d);

    auto rec = [&](auto rec, int i) -> Poly<mod> {
        if (i >= sz) {
            if (i - sz < vals.size()) {
                return Poly<mod>{vals[i - sz] * d[i - sz]};
            } else {
                return Poly<mod>{};
            }
        }
        Poly<mod> a = rec(rec, 2 * i);
        Poly<mod> b = rec(rec, 2 * i + 1);
        return a * data[2 * i + 1] + b * data[2 * i];
    };
    return rec(rec, 1);
}

// https://arxiv.org/abs/2008.08822
template <u32 mod>
MintT<mod> bostan_mori(u64 k, Poly<mod> p, Poly<mod> q) {
    assert(q.coeff(0) != 0);

    using mint = MintT<mod>;
    using poly = Poly<mod>;

    q.remove_zeros(), p.remove_zeros();
    int64_t n = q.deg();
    int lg = 1;
    while ((1 << lg) <= 2 * n) {
        lg++;
    }

    static const NTT<mod> ntt;

    q.resize(1 << lg);
    p.resize(1 << lg);
    poly r(1 << lg), t(1 << lg);

    if (n < k) {
        ntt.template transform<false>(lg, q.data());
        ntt.template transform<false>(lg, p.data());

        while (n < k) {
            for (int i = 0; i < (1 << lg); i += 2) {
                mint a = p[i], b = p[i + 1], c = q[i], d = q[i + 1];
                p[i] = a * d, p[i + 1] = b * c, q[i >> 1] = c * d;
            }
            ntt.extract_cum(lg - 1, p.data(), k & 1);
            k >>= 1;
            if (n < k) {
                ntt.expand_ntt(lg - 1, q.data());
                ntt.expand_ntt(lg - 1, p.data());
            }
        }

        ntt.template transform<true>(lg - 1, q.data());
```

```
        ntt.template transform<true>(lg - 1, p.data());

        p.resize(k + 1), q.resize(k + 1);
        p.remove_zeros(), q.remove_zeros();
    }

    return (p * q.inv_series(k + 1)).coeff(k);
}

template <u32 mod>
MintT<mod> kth_linear(u64 k, const Poly<mod>& gen, const Poly<mod>& ch) {
    // Poly<mod> r = Poly<mod>({0, 1}).power_mod(k, Poly<mod>(ch.rbegin(), ch.rend()));
    // return gen.dot(r);

    int64_t d = ch.deg();
    return bostan_mori(k, (gen * ch).mod_xk(d), ch);
}
};  // namespace polynomial

constexpr u32 mod = 998'244'353;
using mint = MintT<mod>;
using poly = polynomial::Poly<mod>;
```

11 multipoint

```
// https://judge.yosupo.jp/submission/243168

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

size_t ntt_sum_size = 0;

using u32 = uint32_t;
using u64 = uint64_t;

struct WTF {
    static constexpr u32 mod = 998'244'353;
    static constexpr u32 pr_root = 3;
    static constexpr int LG = 32;

    u32 wd[LG], wrd[LG];
    u32 w_rt[LG], wr_rt[LG];

    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 u64(a) * 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;
    }

    WTF() {
        int lg = __builtin_ctz(mod - 1) + 1;
        for (int i = 0; i < std::min(lg, LG); i++) {
            u32 wi = power(pr_root, mod - 1 >> i + 2);
            u32 rm = power(pr_root, (mod - 1 >> i + 1) * ((1 << i) - 1));
            u32 w_dlt = mul(wi, power(rm, mod - 2));
            w_rt[i] = wi, wr_rt[i] = power(wi, mod - 2);
            wd[i] = w_dlt, wrd[i] = power(w_dlt, mod - 2);
        }
    }

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

    template <bool inverse = false, bool right_part = false>
    void transform(int lg, u32* data) {
        ntt_sum_size += 1 << lg;
        for (int k = !inverse ? lg - 1 : 0; !inverse ? k >= 0 : k < lg; !inverse ? k-- : k++) {
            u32 wi = right_part ? (inverse ? wr_rt : w_rt)[lg - 1 - k] : 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 ? wd : wrd)[__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);
            }
        }
    }

    std::vector<u32> inv_fps(std::vector<u32> vec) {
        assert(vec.size() && vec[0] != 0);
        int k = 0;
        std::vector<u32> inv = {power(vec[0], mod - 2)};
        std::vector<u32> tmp1, tmp2, tmp3;
        while ((1 << k) < vec.size()) {
            int n = 1 << k;
            vec.resize(std::max<int>(vec.size(), 2 * n));

            tmp1.assign(2 * n, 0);
            std::copy(inv.begin(), inv.begin() + n, tmp1.begin());
            transform<false>(k + 1, tmp1.data());

            tmp2.assign(2 * n, 0);
            std::copy(vec.begin(), vec.begin() + 2 * n, tmp2.begin());
            transform<false>(k + 1, tmp2.data());

            // const u32 fix = power(mod + 1 >> 1, k + 1);
            for (int i = 0; i < 2 * n; i++) {
                tmp2[i] = mul(tmp1[i], tmp2[i]);
            }
            transform<true>(k + 1, tmp2.data());
            for (int i = 0; i < n; i++) {
```

```
            tmp2[i] = 0;
        }
        transform<false>(k + 1, tmp2.data());
        for (int i = 0; i < 2 * n; i++) {
            tmp1[i] = mul(tmp1[i], add(1, mod - tmp2[i]));
        }
        transform<true>(k + 1, tmp1.data());
        inv.resize(2 * n);
        std::copy(tmp1.begin() + n, tmp1.begin() + 2 * n, inv.begin() + n);
        k++;
    }

    inv.resize(vec.size());
    return inv;
}

std::vector<u32> evaluate(std::vector<u32> poly, std::vector<u32> points) {
    int res_sz = points.size();
    int n = std::max(poly.size(), points.size());
    int lg = std::__lg(std::max<int>(n - 1, 1)) + 1;  // * doesn't work for lg = 0
    poly.resize(1 << lg), points.resize(1 << lg);
    std::vector<std::vector<u32>> data(lg + 1, std::vector<u32>(1 << lg + 1));
    for (int i = 0; i < (1 << lg); i++) {
        data[0][2 * i] = add(0, mod + 1 - points[i]);
        data[0][2 * i + 1] = add(0, mod - 1 - points[i]);
    }
    for (int k = 0; k < lg; k++) {
        for (int i = 0; i < (1 << lg + 1); i += 1 << k + 2) {
            for (int j = 0; j < (1 << k + 1); j++) {
                data[k + 1][i + j] = mul(data[k][i + j], data[k][i + (1 << k + 1) + j]);
            }
            if (k + 1 != lg) {
                std::copy(data[k + 1].begin() + i, data[k + 1].begin() + i + (1 << k + 1),
                          data[k + 1].begin() + i + (1 << k + 1));
                transform<true>(k + 1, data[k + 1].data() + i + (1 << k + 1));
                add_to(data[k + 1][i + (1 << k + 1)], mod - 2);
                transform<false, true>(k + 1, data[k + 1].data() + i + (1 << k + 1));
            } else {
                transform<true>(k + 1, data[k + 1].data() + i);
                add_to(data[k + 1][i], mod - 1);
                add_to(data[k + 1][i + (1 << k + 1)], 1);
            }
        }
    }

    std::vector<u32> dt = std::move(data[lg]);

    std::reverse(dt.begin(), dt.begin() + (1 << lg) + 1);
    dt.resize(1 << lg);

    dt = inv_fps(dt);
    std::reverse(dt.begin(), dt.end());

    dt.resize(1 << lg + 1);
    transform<false>(lg + 1, dt.data());

    poly.resize(1 << lg + 1);
    std::rotate(poly.begin(), poly.begin() + (1 << lg + 1) - 1, poly.end());
    transform<false>(lg + 1, poly.data());
    for (int i = 0; i < (1 << lg + 1); i++) {
        dt[i] = mul(dt[i], poly[i]);
        }

        for (int k = lg - 1; k >= 0; k--) {
            for (int i = 0; i < (1 << lg + 1); i += (1 << k + 2)) {
                transform<true, true>(k + 1, dt.data() + i + (1 << k + 1));
                transform<false>(k + 1, dt.data() + i + (1 << k + 1));
                for (int j = 0; j < (1 << k + 1); j++) {
                    u32 val = add(dt[i + j], mod - dt[i + (1 << k + 1) + j]);
                    dt[i + (1 << k + 1) + j] = mul(val, data[k][i + j]);
                    dt[i + j] = mul(val, data[k][i + (1 << k + 1) + j]);
                }
            }
        }

        std::vector<u32> ans(1 << lg);
        u32 fix = power(mod + 1 >> 1, lg + 1);
        for (int i = 0; i < (1 << lg); i++) {
            ans[i] = add(dt[2 * i], mod - dt[2 * i + 1]);
            ans[i] = mul(ans[i], fix);
        }
        ans.resize(res_sz);
        return ans;
    }
};
```

12 ******** 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);
 }
};
```

13 cached compile

```
import os
import subprocess


def cached_compile(path):
    if not os.path.exists(path) or os.path.getmtime(path) < os.path.getmtime(f"{path}.cpp"):
```

```
        print(f"compiling {path}...", end=" ", flush=True)
        ret = subprocess.run(f"g++ {path}.cpp -o {path} -std=c++20 -O2", shell=True).returncode
        if ret != 0:
            print("fuck")
            exit(1)
        print("compiled", flush=True)
```

14 suff automaton

```
#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;
}
```

15 sum kth powers

The first seven examples of Faulhaber's formula are

$$\sum_{k=1}^{n} k^0 = \frac{1}{1}\left(n\right)$$
$$\sum_{k=1}^{n} k^1 = \frac{1}{2}\left(n^2 + \tfrac{2}{2}n\right)$$
$$\sum_{k=1}^{n} k^2 = \frac{1}{3}\left(n^3 + \tfrac{3}{2}n^2 + \tfrac{3}{6}n\right)$$
$$\sum_{k=1}^{n} k^3 = \frac{1}{4}\left(n^4 + \tfrac{4}{2}n^3 + \tfrac{6}{6}n^2 + 0n\right)$$
$$\sum_{k=1}^{n} k^4 = \frac{1}{5}\left(n^5 + \tfrac{5}{2}n^4 + \tfrac{10}{6}n^3 + 0n^2 - \tfrac{5}{30}n\right)$$
$$\sum_{k=1}^{n} k^5 = \frac{1}{6}\left(n^6 + \tfrac{6}{2}n^5 + \tfrac{15}{6}n^4 + 0n^3 - \tfrac{15}{30}n^2 + 0n\right)$$
$$\sum_{k=1}^{n} k^6 = \frac{1}{7}\left(n^7 + \tfrac{7}{2}n^6 + \tfrac{21}{6}n^5 + 0n^4 - \tfrac{35}{30}n^3 + 0n^2 + \tfrac{7}{42}n\right).$$

The coefficients of Faulhaber's formula in its general form involve the Bernoulli numbers B_j. The Bernoulli numbers begin

$$\begin{array}{llll} B_0 = 1 & B_1 = \frac{1}{2} & B_2 = \frac{1}{6} & B_3 = 0 \\ B_4 = -\frac{1}{30} & B_5 = 0 & B_6 = \frac{1}{42} & B_7 = 0, \end{array}$$

where here we use the convention that $B_1 = +\frac{1}{2}$. The Bernoulli numbers have various definitions (see Bernoulli number § Definitions), such as that they are the coefficients of the exponential generating function

$$\frac{t}{1-\mathrm{e}^{-t}} = \frac{t}{2}\left(\coth\frac{t}{2} + 1\right) = \sum_{k=0}^{\infty} B_k \frac{t^k}{k!}.$$

Then Faulhaber's formula is that

$$\sum_{k=1}^{n} k^p = \frac{1}{p+1}\sum_{r=0}^{p}\binom{p+1}{r} B_r n^{p+1-r}.$$