Niklas' Algorithm Competition Template Library (NACTL)

Template

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

#ifdef COMPPROG_LOCAL
#include <fmt/format.h>
#define dbg(...) fmt::print(stderr, __VA_ARGS__)
#define debug_assert assert
#else
#define dbg(...)
#define debug_assert(...)
#endif

using namespace std;
using uint = unsigned;
using ll = long long;
using ull = unsigned long long;

template <typename E = uint> using Graph = vector<vector<E>>;
template <typename T>
using min_priority_queue = priority_queue<T, vector<T>, std::greater<T>>;

#define rep(a, b) for (int a = 0; a < (b); ++a)
#define urep(a, b) for (uint a = 0; a < (b); ++a)
#define all(a) (a).begin(), (a).end()
#define endl '\n'

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.precision(10);
}
```

Data Structures

UnionFind

```cpp
struct UnionFind {
    vector<uint> p, size;

    UnionFind(uint n) : p(n), size(n, 1) { iota(p.begin(), p.end(), 0); }

    auto raw_key(uint i) const {
        while (p[i] != i) i = p[i];
        return i;
    }
    void update_key(uint i, const uint k) {
        while (p[i] != k) i = exchange(p[i], k);
    }
    auto key(uint i) {
        const auto k = raw_key(i);
        update_key(i, k);
        return k;
    }
    void unite(uint i, uint j) {
```

```
        const auto ki = raw_key(i), kj = raw_key(j);
        if (ki == kj) return;
        const auto k = size[ki] >= size[kj] ? ki : kj;
        size[k] = size[ki] + size[kj];
        update_key(i, k);
        update_key(j, k);
    }
};
```

Range

```
struct Range {
    ull start, end;

    // static Range from_ends(ull start, ull end) { return {start, end}; }

    bool operator==(const Range &rhs) const {
        return (is_empty() && rhs.is_empty()) ||
               (start == rhs.start && end == rhs.end);
    }

    bool is_subset_of(Range other) const {
        return other.start <= start && end <= other.end;
    }
    Range intersect_with(Range other) const {
        return Range{
            .start = max(start, other.start),
            .end = min(end, other.end),
        };
    }
    bool is_empty() const { return start >= end; }
    ull size() const {
        if (is_empty()) return 0;
        else return end - start;
    }
    bool contains(ull point) const { return start <= point && point < end; }
    pair<Range, Range> split_at_mid() const {
        assert(!is_empty());
        auto m = start + (end - start) / 2;
        return {{start, m}, {m, end}};
    }
};
```

Segment Tree

```
template <typename T, typename Op, T neutral_element> struct SegmentTree {
    Op op;
    uint n;
    vector<T> tree;
    vector<T> lazy;

    SegmentTree(uint _n)
        : n(bit_ceil(_n))
        , tree(2 * n + 1, neutral_element)
        , lazy(2 * n + 1, neutral_element) {}

    bool is_leaf(uint v) const { return n <= v && v < 2 * n; }
    void recalc(uint v) { tree[v] = op(tree[left_child(v)], tree[right_child(v)]); }
```

```
    void recalc_all_parents(uint v) {
        v = parent(v);
        while (root <= v) recalc(v), v = parent(v);
    }
    static const uint root = 1;
    static uint parent(uint v) { return v / 2; }
    static uint left_child(uint v) { return 2 * v; }
    static uint right_child(uint v) { return 2 * v + 1; }

    void push_lazy(uint v) {
        if (lazy[v] == neutral_element) return;

        apply_lazy(left_child(v), lazy[v]);
        apply_lazy(right_child(v), lazy[v]);
        lazy[v] = neutral_element;
    }
    void apply_lazy(uint v, T value) {
        tree[v] = op(tree[v], value);
        if (!is_leaf(v)) lazy[v] = op(lazy[v], value);
    }

    void push_all_lazy(uint v) {
        for (uint bits_to_shift = bit_width(v) - 1; bits_to_shift > 0; --
bits_to_shift)
            push_lazy(v >> bits_to_shift);
    }

    void range_update(Range range, T value) {
        push_all_lazy(n + range.start - 1);
        push_all_lazy(n + range.end - 2);
        for (auto l = n + range.start - 1, r = n + range.end - 1; l < r;
             l = parent(l), r = parent(r)) {
            if (l & 1) apply_lazy(l++, value);
            if (r & 1) apply_lazy(--r, value);
        }
        auto first_touched = [](auto x) { return x >> countr_zero(x); };
        recalc_all_parents(first_touched(n + range.start - 1));
        recalc_all_parents(first_touched(n + range.end - 1) - 1);
    }
    T range_query(Range range) {
        auto agg = neutral_element;
        push_all_lazy(n + range.start - 1);
        push_all_lazy(n + range.end - 2);
        for (auto l = n + range.start - 1, r = n + range.end - 1; l < r;
             l = parent(l), r = parent(r)) {
            if (l & 1) agg = op(tree[l++], agg);
            if (r & 1) agg = op(agg, tree[--r]);
        }
        return agg;
    }
};
```

Graph

Flows

Dinitz

```
struct Edge {
    uint from, to;
    ll cap;
    ll flow = 0;
    Edge *rev = nullptr;
};

Graph<Edge *> g(n);

auto add_edge = [&](uint u, uint v, ll cap) {
    if (cap == 0) return;
    auto uv = new Edge{u, v, cap};
    auto vu = new Edge{v, u, 0};
    uv->rev = vu;
    vu->rev = uv;
    g[u].push_back(uv);
    g[v].push_back(vu);
};

ll total_flow = 0;
while (true) {
    // build L
    const auto inf = numeric_limits<uint>::max();
    vector<uint> dist(g.size(), inf);
    dist[source] = 0;
    queue<uint> q;
    q.push(source);
    while (!q.empty()) {
        auto u = q.front();
        q.pop();
        for (auto e : g[u]) {
            if (dist[e->to] == inf && e->flow < e->cap) {
                q.push(e->to);
                dist[e->to] = dist[u] + 1;
            }
        }
    }
    if (dist[target] == inf) break;

    // augment
    vector<uint> current_edge_index(g.size(), 0);

    auto dfs = [&](uint u, ll bottleneck_so_far, auto &rec) -> ll {
        if (u == target) return bottleneck_so_far;
        for (auto &i = current_edge_index[u]; i < g[u].size(); ++i) {
            auto e = g[u][i];
            if (e->flow == e->cap) continue;
            if (dist[e->to] != dist[u] + 1) continue;
            auto pushed = rec(e->to, min(bottleneck_so_far, e->cap - e->flow), rec);
            if (pushed == 0) continue;
            e->flow += pushed;
            e->rev->flow -= pushed;
            return pushed;
        }
        return 0;
    };
```

```
    while (true) {
        auto bottleneck = dfs(source, numeric_limits<ll>::max(), dfs);
        if (bottleneck == 0) break;
        total_flow += bottleneck;
    }
}
```

Path Decomposition

```
vector<uint> current_edge_index(n, 0);
vector<pair<ll, vector<uint>>> paths;

while (true) {
    vector<Edge *> path_edges;
    vector<uint> path;
    auto dfs = [&](uint u, ll bottleneck_so_far, auto &rec) -> ll {
        if (u == target) return bottleneck_so_far;
        for (auto &i = current_edge_index[u]; i < g[u].size(); ++i) {
            auto e = g[u][i];
            if (e->cap != 0 && e->flow > 0) {
                path_edges.push_back(e);
                path.push_back(u);
                return rec(e->to, min(bottleneck_so_far, e->flow), rec);
            }
        }
        return 0;
    };
    auto bn = dfs(source, numeric_limits<ll>::max(), dfs);
    if (bn == 0) break;
    for (auto e : path_edges) e->flow -= bn, e->rev->flow += bn;

    path.push_back(target);
    paths.push_back(make_pair(bn, move(path)));
}
```

Trees

Strings

Suffix Array

```
pair<vector<vector<uint>>, vector<uint>> build_suffix_array(const auto &s) {
    debug_assert(has_single_bit(s.size()));

    vector<uint> order(s.size());
    iota(all(order), 0);
    ranges::sort(order, [&](auto i, auto j) { return s[i] < s[j]; });

    auto equiv = vector(ceil(log2(s.size())) + 1, vector(s.size(), 0u));
    urep(i, s.size() - 1) equiv[0][order[i + 1]] =
        equiv[0][order[i]] + (s[order[i]] != s[order[i + 1]]);

    for (uint k = 0; (1 << k) < s.size(); ++k) {
        const auto &cur_equiv = equiv[k];

        vector<uint> count(s.size(), 0);
        urep(i, s.size()) count[cur_equiv[i]]++;
```

```
        partial_sum(all(count), count.begin());

        vector<uint> by_second(s.size());
        urep(i, s.size()) by_second[i] =
            ((((int)order[i] - (1 << k)) % s.size()) + s.size()) % s.size();
        ranges::reverse(by_second);
        for (auto i : by_second) order[--count[cur_equiv[i]]] = i;

        const auto parts = [&](auto i) {
            return pair(cur_equiv[i], cur_equiv[(i + (1 << k)) % s.size()]);
        };

        // ranges::sort(order,
        //              [&](auto i, auto j) { return parts(i) < parts(j); });

        auto &next_equiv = equiv[k + 1];
        urep(i, s.size() - 1) next_equiv[order[i + 1]] =
            next_equiv[order[i]] + (parts(order[i]) != parts(order[i + 1]));
    }

    return pair(move(equiv), move(order));
}

string s;

s += '$';
while (!has_single_bit(s.size())) s += '$';

const auto [_equivs, _suf] = build_suffix_array(s);
const auto &equivs = _equivs;
const auto &suf = _suf;

const auto lcp = [&](uint i, uint j) {
    debug_assert(i != j);

    uint len = 0;
    for (int k = equivs.size() - 1; k >= 0; --k) {
        if (equivs[k][i] == equivs[k][j]) {
            len += 1 << k;
            i += 1 << k;
            j += 1 << k;
        }
    }
    return len;
};
```

Math

Chinese Remainder Theorem

```
array<ll, 3> gcd_ext(ll a, ll b) {
    if (!a) return {b, 0, 1};
    auto [d, x1, y1] = gcd_ext(b % a, a);
    return {d, y1 - (b / a) * x1, x1};
}

pair<ll, ll> crt(ll a, ll m, ll b, ll n) {
```

```
    auto [g, x, y] = gcd_ext(m, n);
    ll l = m / g * n;
    ll ret = (a + ((x * (b - a) / g) % (l / m)) * m) % l;
    return {(b - a) % g ? -1 : (ret + l) % l, l};
}

pair<ll, ll> crt(span<ll const> a, span<ll const> m) {
    ll sol = a[0], l = m[0];
    for (uint i = 1; i < a.size(); i++) {
        tie(sol, l) = crt(sol, l, a[i], m[i]);
        if (sol == -1) break;
    }
    return {sol, l};
}
```

Primality and Factoring

```
bool miller_rabin(ull n) {
    assert(n > 1);
    constexpr static array<ll, 12> witnesses = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
31, 37};
    if (n < 38) return find(all(witnesses), n) != witnesses.end();

    auto d = n - 1;
    auto s = countr_zero(d);
    d >>= s;

    return ranges::all_of(witnesses, [=](auto witness) {
        ll x = mod_pow(witness, d, n);
        if (x == 1 || x == n - 1) return true;
        rep(_, s - 1) {
            x = (x * x) % n;
            if (x == n - 1) return true;
        }
        return false;
    });
}

ll rho(ll n, ll x0 = 2, ll c = 1) {
    if (n % 2 == 0) return 2;
    auto f = [=](ll x) { return (((x * x) % n) + c) % n; };
    ll x = x0;
    ll y = f(x0);
    ll g = gcd((((x - y) % n) + n) % n, n);
    while (g == 1) {
        x = f(x);
        y = f(f(y));
        g = gcd((((x - y) % n) + n) % n, n);
    }
    return g;
}

void factorize(auto n, auto &callback) {
    static mt19937_64 gen;
    static uniform_int_distribution<> distrib(2, 1000);

    assert(n > 1);
```

```
    if (miller_rabin(n)) {
        callback(n);
        return;
    }

    auto d = rho(n, 2, distrib(gen));
    while (d == n) d = rho(n, 2, distrib(gen));

    factorize(d, callback);
    factorize(n / d, callback);
}
```

Note: FFT repeats after four applications!

Fast Fourier Transform

```
using C = complex<long double>;
constexpr auto pi = std::numbers::pi_v<C::value_type>;
using Polynomial = vector<C>;
using PointForm = Polynomial;

void fft_iter(Polynomial &a, const span<const C> z) {
    const auto n = a.size();
    const auto log_n = bit_width(n) - 1;
    for (uint i = 0, j = 0; i < n - 1; ++i) {
        if (i < j) swap(a[i], a[j]);
        const uint x = countr_one(i) + 1;
        j ^= ((1 << x) - 1) << (log_n - x);
    }
    for (uint k = 1; 2 * k <= n; k *= 2) {
        for (uint i = 0; i < n; i += 2 * k) {
            urep(j, k) {
                const auto u = a[i + j], v = a[i + j + k];
                a[i + j] = u + v * z[j * n / (2 * k)];
                a[i + j + k] = u - v * z[j * n / (2 * k)];
            }
        }
    }
}

Polynomial multiply(Polynomial a, Polynomial b) {
    const auto na = a.size(), nb = b.size();
    const auto n = bit_ceil(na + nb - 1);
    a.resize(n, 0);
    b.resize(n, 0);

    PointForm z(n);
    urep(i, n) z[i] = polar(C::value_type(1), i * 2 * pi / n);

    fft_iter(a, z);
    fft_iter(b, z);

    PointForm c(n);
    urep(i, n) c[i] = a[i] * b[i] / C(n);
    fft_iter(c, z);

    reverse(c.begin() + 1, c.end());
    c.resize(na + nb - 1);
```

```
    return c;
}
```

Number Theoretic Transform

```
template <const ull MOD, const ull prim_root> struct NTT {
    constexpr static ull modmul(ull a, ull b) {
        return (a * b) % MOD;
        // ll ret = a * b - MOD * ull(1.L / MOD * a * b);
        // return ret + MOD * (ret < 0) - MOD * (ret >= (ll)MOD);
    }
    constexpr static ull modpow(ull b, ull e) {
        ull ans = 1;
        for (; e; b = modmul(b, b), e /= 2)
            if (e & 1) ans = modmul(ans, b);
        return ans;
    }
    constexpr static ull modinv(ull x) { return modpow(x, MOD - 2); }

    static_assert(modpow(prim_root, MOD - 1) == 1);

    using Polynomial = vector<ull>;
    using PointForm = Polynomial;

    static void ntt_iter(Polynomial &a) {
        const auto n = a.size();
        debug_assert(has_single_bit(n));
        const auto log_n = bit_width(n) - 1;
        for (uint i = 0, j = 0; i < n - 1; ++i) {
            if (i < j) swap(a[i], a[j]);
            const uint x = countr_one(i) + 1;
            j ^= ((1 << x) - 1) << (log_n - x);
        }
        for (uint k = 1; 2 * k <= n; k *= 2) {
            debug_assert((MOD - 1) % (2 * k) == 0);
            const auto zn = modpow(prim_root, (MOD - 1) / (2 * k));
            for (uint i = 0; i < n; i += 2 * k) {
                ull z = 1;
                urep(j, k) {
                    const auto u = a[i + j], v = a[i + j + k];
                    const auto vv = modmul(v, z);
                    a[i + j] = (u + vv) % MOD;
                    a[i + j + k] = (u + (MOD - vv)) % MOD;
                    z = modmul(z, zn);
                }
            }
        }
    }

    // Repeated application
    void ntt_q(Polynomial &a, ull q) {
        const auto factor = modpow(a.size(), ((q / 4) * 2));
        for (auto &x : a) x = modmul(x, factor);
        urep(_, q % 4) ntt_iter(a);
    }
};
```