Niklas' Algorithm Competition Template Library (NACTL)

Contents

0.a. 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);
}
```

1. Data Structures

1.a. Union Find

```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);
    }
};
```

1.b. Range

```cpp
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}};
    }
};
```

1.c. Sparse Table

```
template <typename T, auto op, T neutral_element> struct
SparseTable {
    // op needs to
    // - be associative
    // - be idempotent
    // - have neutral element neutral_element

    uint n;
    vector<T> aggs;

    uint idx(uint i, uint row) const { return n * row +
i; }

    SparseTable(span<T const> vals) : n(vals.size()),
aggs(n * (bit_width(n) + 1)) {

        ranges::copy(vals, aggs.begin());

        for (uint row = 1, agg_size = 2; agg_size < n; +
+row, agg_size <<= 1) {
            for (uint i = 0; i + (agg_size / 2) < n; ++i)
{
                aggs[idx(i, row)] =
                    op(aggs[idx(i, row - 1)], aggs[idx(i +
(agg_size / 2), row - 1)]);
            }
        }
    }

    // zero-based
    T range_query(Range range) const {
        if (range.is_empty()) return neutral_element;
        auto agg_index = bit_width(range.size()) - 1;
        auto agg_size = 1u << agg_index;
        return op(aggs[idx(range.start, agg_index)],
aggs[idx(range.end - agg_size, agg_index)]);
    }
};
```

1.d. 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;
    }
};
```

1.e. Treap

```
struct Treap {
    struct Lazy {
        bool reversed = false;

        bool operator==(const Lazy &) const = default;
    };
    struct Agg {
        ull sum;
        ll max;
    };

    static inline mt19937 gen;
    static inline uniform_int_distribution<uint> dist;

    struct Node {
        uint value;
        uint priority;

        Node *left = nullptr, *right = nullptr;
        Agg agg;
        Lazy lazy{};
        uint size = 1;

        Node(uint _value) : value(_value),
priority(dist(gen)), agg({value, value}) {}

        void push_lazy() {
            if (lazy.reversed) {
                swap(left, right);
                if (left) left->lazy.reversed ^= 1;
                if (right) right->lazy.reversed ^= 1;
            }
            lazy = Lazy{};
        }
        void update_agg() {
            debug_assert(lazy == Lazy{});

            agg.sum = value;
            agg.max = value;
            size = 1;
            if (left) {
                agg.sum += left->agg.sum;
                agg.max = max(agg.max, left->agg.max);
                size += left->size;
            }
            if (right) {
                agg.sum += right->agg.sum;
                agg.max = max(agg.max, right->agg.max);
                size += right->size;
            }
        }
        const Node *get_kth(uint k) {
            debug_assert(1 <= k);
            debug_assert(k <= size);
            push_lazy();
            if (k <= sz(left)) return left->get_kth(k);
            else if (k == 1 + sz(left)) return this;
            else return right->get_kth(k - 1 - sz(left));
        }
        void update_kth(uint k, uint new_value) {
            debug_assert(1 <= k);
            debug_assert(k <= size);
            push_lazy();
            if (k <= sz(left)) left->update_kth(k,
new_value);
            else if (k == 1 + sz(left)) value = new_value;
            else right->update_kth(k - 1 - sz(left),
new_value);
            update_agg();
        }
    };

    static Node *merge(Node *a, Node *b) {
        if (!a) return b;
        if (!b) return a;

        if (a->priority < b->priority) {
            a->push_lazy();
            a->right = merge(a->right, b);
            a->update_agg();
            return a;
        } else {
            b->push_lazy();
            b->left = merge(a, b->left);
            b->update_agg();
            return b;
        }
    }

    static uint sz(Node *v) { return v ? v->size : 0; }
    static pair<Node *, Node *> split_size(Node *v, uint
left_size) {
        if (!v) return {nullptr, nullptr};

        v->push_lazy();

        if (1 + sz(v->left) <= left_size) {
            auto [rl, rr] = split_size(v->right, left_size
- 1 - sz(v->left));
            v->right = rl;
            v->update_agg();
            return {v, rr};
        } else {
            auto [ll, lr] = split_size(v->left,
left_size);
            v->left = lr;
            v->update_agg();
            return {ll, v};
        }
    }

    Node *root = nullptr;

    Treap(span<uint> initial_vals) {
        for (auto v : initial_vals) root = merge(root, new
Node(v));
    }

    static tuple<Node *, Node *, Node *>
extract_inner_range(Node *v, Range range) {
        auto [lm, r] = split_size(v, range.end - 1);
        auto [l, m] = split_size(lm, range.start - 1);
        return {l, m, r};
    }

    void insert_after(uint pos, uint value) { root =
insert_all_after(root, pos, new Node(value)); }
    static Node *insert_all_after(Node *v, uint pos, Node
*vals) {
        auto [l, r] = split_size(v, pos);
        return merge(merge(l, vals), r);
    }
    void remove(uint pos) {
        auto [l, single, r] = extract_inner_range(root,
{pos, pos + 1});
        debug_assert(sz(single) == 1);
        dbg("Removed node with value {}\n", single-
>value);
        root = merge(l, r);
    }
    const Node *at(uint pos) { return root-
>get_kth(pos); }
    void update(uint pos, uint value) { root-
>update_kth(pos, value); }

    Agg range_query(Range range) {
        auto [l, m, r] = extract_inner_range(root, range);
        Agg result = m ? m->agg : Agg{};
        root = merge(l, merge(m, r));
        return result;
    }

    void move_range(Range range, uint to_pos) {
        auto [l, m, r] = extract_inner_range(root, range);
        root = merge(l, r);
        root = insert_all_after(root, to_pos - 1, m);
    }
    void reverse_in_range(Range range) {
        auto [l, m, r] = extract_inner_range(root, range);
        m->lazy.reversed ^= 1;
        root = merge(l, merge(m, r));
    }
    template <typename F> static void visit_in_order(Node
*v, F &f) {
        if (!v) return;
        v->push_lazy();
        visit_in_order(v->left, f);
        f(*v);
        visit_in_order(v->right, f);
    }
    void debug_print_all() {
#ifdef COMPPROG_LOCAL
        auto visit = [](Node &v) { dbg("{} ", v.value); };
        visit_in_order(root, visit);
#endif
    }
};
```

2. Graph

2.a. Topological Order

```
vector<uint> topo;
urep(u, n) if (!in_deg[u]) topo.push_back(u);
urep(i, n) for (auto u : g[topo[i]]) if (!--in_deg[u])
topo.push_back(u);
```

2.b. Bridges

```
vector<uint> st;
vector<vector<uint>> components;
vector<int> depth_of(n, -1);
vector<pair<uint, uint>> bridges;

auto dfs = [&](uint v, uint parent, uint depth, auto &rec)
-> uint {
    auto old_size = size(st);
    st.push_back(v);

    uint up = 0;
    depth_of[v] = depth;
    for (auto neigh : g[v]) {
        if (depth_of[neigh] == -1) up += rec(neigh, v,
depth + 1, rec);
        up += (depth_of[neigh] < depth);
        up -= (depth_of[neigh] > depth);
    }

    if (up == 1) {
        bridges.emplace_back(parent, v);
    }

    if (depth == 0 || up == 1) {
        components.emplace_back(st.begin() + old_size,
st.end());
        st.resize(old_size);
    }
    return up;
};
dfs(0, 0, 0, dfs);
```

2.c. Cut-Vertices

```
// returns earliest discover time
// reachable via 1 edge from my subtree
int dfs(int v) {
    int low = disc[v] = time++;
    int kids = 0;
    for (int v2 : adj[v]) {
        int st_pos = size(st);
        if (disc[v2] == -1) { // tree edge
            ++kids;
            int low2 = dfs(v2, v);
            low = min(low, low2);
            if (low2 >= disc[v]) { // exiting comp
                comps.emplace_back(begin(st) + st_pos,
end(st));
                st.resize(st_pos);

                if (p[v] != -1 && kids > 1) {
                    // v is cut-vertex
                }
            }
        }
        if (disc[v2] < disc[v]) // from below
            st.push_back({v, v2});
        low = min(low, disc[v2]);
    }
    return low;
}
```

2.d. SCC

```
uint scc_count = 0;
vector<vector<uint>> scss;
vector<uint> scc_of(g.size(), inf);
vector<uint> disc(g.size(), inf);
vector<uint> some_node_in_scc;

auto dfs = [&, time = 0u, st = vector<uint>(), in_stack =
vector(g.size(), false)](
               uint v, auto &rec) mutable -> uint {
    uint old_size = st.size();
    st.push_back(v);
    in_stack[v] = true;
    uint low = disc[v] = time++;
    for (auto v2 : g[v]) {
        if (disc[v2] == inf) low = min(low, rec(v2, rec));
        else if (in_stack[v2]) low = min(low, disc[v2]);
    }
    if (low == disc[v]) {
        for (uint i = old_size; i < st.size(); ++i) {
            scc_of[st[i]] = scc_count;
            in_stack[st[i]] = false;
        }
        some_node_in_scc.push_back(v);
        scc_count++;

        sccs.emplace_back(st.begin() + old_size,
st.end());
        st.resize(old_size);
    }
    return low;
};
urep(v, g.size()) if (disc[v] == inf) dfs(v, dfs);
```

3. Flows

3.a. 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;
    }
}
```

3.b. 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)));
}
```

3.c. Min Cost Flow

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

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

ll total_flow = 0;

vector<ll> johnson_offset(g.size(), inf);
{
    // Bellman Ford to get johnson offsets
    auto &dist = johnson_offset;
    dist[source] = 0;
    queue<uint> q;
    q.push(source);
    vector<bool> in_q(g.size());
    in_q[source] = true;

    while (!q.empty()) {
        auto u = q.front();
        q.pop();
        in_q[u] = false;
        for (auto e : g[u]) {
            if (e->flow == e->cap) continue;
            if (dist[e->from] == inf) continue;
            if (dist[e->from] + e->cost >= dist[e->to])
continue;
            dist[e->to] = dist[e->from] + e->cost;
            if (in_q[e->to]) continue;
            q.push(e->to);
            in_q[e->to] = true;
        }
    }
}
while (true) {
    // try to find shortest path according to cost
    vector<ll> dist(g.size(), inf);
    dist[source] = 0;
    vector<Edge *> incoming(g.size());
    priority_queue<pair<ll, uint>, vector<pair<ll, uint>>,
greater<>> q;
    q.push({0, source});
    vector<bool> done(g.size());
    while (!q.empty()) {
        auto [_, u] = q.top();
        q.pop();
        done[u] = true;
        for (auto e : g[u]) {
            if (done[e->to]) continue;
            if (e->flow == e->cap) continue;
            if (dist[e->from] == inf) continue;
            const auto john_cost = e->cost +
johnson_offset[e->from] - johnson_offset[e->to];
            if (dist[e->from] + john_cost >= dist[e->to])
continue;
            dist[e->to] = dist[e->from] + john_cost;
            incoming[e->to] = e;
            q.push({dist[e->to], e->to});
        }
    }
    if (dist[target] == inf) break;

    urep(v, g.size()) johnson_offset[v] += dist[v] -
dist[target];

    // find bottleneck
    ll bottleneck = inf;
    for (auto v = target; v != source; v = incoming[v]-
>from)
        bottleneck = min(bottleneck, incoming[v]->cap -
incoming[v]->flow);
    total_flow += bottleneck;
    // augment
    for (auto v = target; v != source; v = incoming[v]-
>from) {
        incoming[v]->flow += bottleneck;
        incoming[v]->rev->flow -= bottleneck;
    }
}

ll total_cost = 0;
for (auto &row : g)
    for (auto e : row) total_cost += e->flow * e->cost;
total_cost /= 2;
```

4. Trees

4.a. Binary lifting

```
auto lift_by = vector(bit_width(g.size()),
vector(g.size(), root));
urep(v, n) lift_by[0][v] = parent[v];
lift_by[0][root] = root;
urep(prev_height, lift_by.size() - 1)
    urep(v, n) lift_by[prev_height + 1][v] =
lift_by[prev_height][lift_by[prev_height][v]];

auto lca = [&](uint u, uint v) {
    if (depth[u] > depth[v]) swap(u, v);
    const auto depth_diff = depth[v] - depth[u];
    urep(i, lift_by.size()) if (depth_diff & (1 << i)) u =
lift_by[i][u];
    debug_assert(depth[u] == depth[v]);
    if (u == v) return u;
    for (int i = lift_by.size() - 1; i >= 0; --i)
        if (lift_by[i][u] != lift_by[i][v]) u = lift_by[i]
[u], v = lift_by[i][v];
    return parent[u];
};
```

4.b. Heavy Light Decomposition

```
const auto inf = numeric_limits<uint>::max();
vector<uint> size(n, 0), parent(n, inf), depth(n, inf);
vector<uint> heavy_child(n, inf);
auto dfs1 = [&](auto v, auto &rec) -> void {
    size[v] = 1;
    for (auto child : g[v])
        if (child != parent[v]) {
            parent[child] = v;
            depth[child] = depth[v] + 1;
            rec(child, rec);
            size[v] += size[child];
            if (heavy_child[v] == inf ||
size[heavy_child[v]] < size[child]) heavy_child[v] =
child;
        }
};
const uint root = 0;
depth[root] = 0;
parent[root] = root;
dfs1(root, dfs1);

// heavy light
vector<uint> head(n), pos(n);
vector<uint> initial_values(n);
auto dfs2 = [&, next_index = 1u](uint v, uint cur_head,
auto &rec) mutable -> void {
    head[v] = cur_head;
    pos[v] = next_index++;
    initial_values[pos[v] - 1] = value_for_node[v];
    if (heavy_child[v] != inf) rec(heavy_child[v],
cur_head, rec);
    for (auto child : g[v])
        if (child != parent[v] && child != heavy_child[v])
rec(child, child, rec);
};
dfs2(root, root, dfs2);

SegmentTree seg(initial_values);

uint u = a, v = b;
SegmentTree::Node ans = SegmentTree::neutral_element;
for (; head[u] != head[v]; v = parent[head[v]]) {
    if (depth[head[u]] > depth[head[v]]) swap(u, v);
    ans = SegmentTree::op(ans,
seg.range_query({pos[head[v]], pos[v] + 1}));
}
if (depth[u] > depth[v]) swap(u, v);
ans = SegmentTree::op(ans, seg.range_query({pos[u], pos[v]
+ 1}));
```

4.c. Centroids

```
uint find_centroid(Graph<> &g, const uint n, const uint
root, span<Data const> data) {
    uint centroid = root;

check:
    for (uint i = 0; i < g[centroid].size();) {
        const auto child = g[centroid][i];
        if (data[child].deleted) {
            swap_remove(g[centroid], i);
            continue;
        }
        ++i;

        if (child != data[centroid].parent && 2 *
data[child].sz > n) {
            centroid = child;
            goto check;
        }
    }

    return centroid;
}
```

Note: Slides mention centroid tree for radius queries.

5. Strings

5.a. P and Z functions

```
auto prefix(string_view x) {
    // P[i] is the length of the longest proper prefix of
x that is a suffix of
    // x[..i]
    vector<unsigned> P(x.length());
    P[0] = 0;
    for (unsigned i = 1; i < x.length(); i++) {
        auto j = P[i - 1];
        while (j > 0 && x[i] != x[j]) j = P[j - 1];
        if (x[i] == x[j]) j++;
        P[i] = j;
    }
    return P;
}

auto string_z(string_view x) {
    // Z[i] is the length of the longest common prefix of
t and t[i..]
    vector<unsigned> Z(x.length(), 0);
    unsigned l = 0, r = 0;
    for (unsigned i = 1; i < x.length(); i++) {
        if (i <= r) Z[i] = min(r - i + 1, Z[i - l]);
        while (i + Z[i] < x.length() && x[Z[i]] == x[i +
Z[i]]) Z[i]++;
        if (i + Z[i] - 1 > r) {
            l = i;
            r = i + Z[i] - 1;
        }
    }
    return Z;
}
```

5.b. Rolling hashes

```
constexpr ull base = 28;
constexpr ull base_inv = 535714297;
constexpr ull mod = 1000000021;
constexpr array<ull, 32> a_powers = ([]() {
    array<ull, 32> res;
    res[0] = base;
    urep(i, 31) res[i + 1] = (res[i] * res[i]) % mod;
    return res;
})();
auto to_num = [](char c) { return 1ull + c - 'a'; };

auto pow_a = [&a_powers](auto exp) {
    ull res = 1;
    urep(i, 32) {
        if (exp & 1) res = (res * a_powers[i]) % mod;
        exp >>= 1;
    }
    return res;
};

string s;
ull hash = 0;
rep(i, len) hash = (((hash * base) % mod) + to_num(s[len -
1 - i])) % mod;
```

5.c. 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;
};
```

5.d. Aho-Corasick

```
struct AhoCorasick {
    constexpr static uint A = 26;

    static uint to_num(char c) { return c - 'a'; }

    struct Node {
        array<uint, A> next = {};
        uint matches_here = 0;
    };

    constexpr static uint root = 0;
    vector<Node> trie;

    AhoCorasick() : trie(1) {}

    void add_string(string_view s) {
        uint v = root;
        for (auto c : s) {
            c = to_num(c);
            if (!trie[v].next[c]) {
                trie[v].next[c] = trie.size();
                trie.emplace_back();
            }
            v = trie[v].next[c];
        }
        ++trie[v].matches_here;
    }
    void finalize() {
        queue<array<uint, 4>> q;
        q.push({0, 0, 0, 0});
        while (!q.empty()) {
            auto [v, p, plink, pch] = q.front();
            q.pop();

            const auto link = p == 0 ? 0 :
trie[plink].next[pch];
            urep(c, A) {
                if (trie[v].next[c])
q.push({trie[v].next[c], v, link, c});
                else trie[v].next[c] = trie[link].next[c];
            }
                trie[v].matches_here +=
trie[link].matches_here;
        }
    }
    uint count_matches(string_view text) {
        uint v = root;
        uint count = trie[v].matches_here;
        for (auto c : text) {
            c = to_num(c);
            v = trie[v].next[c];
            count += trie[v].matches_here;
        }
        return count;
    }
};
```

6. Math

6.a. 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};
}
```

6.b. 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);
}
```

6.c. 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;
}
```

6.d. 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
    static 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);
    }
};
```