```cpp
/*
 * Returns a longest increasing subsequence in O(n lg n).
 */
template <typename Seq>
vector<typename Seq::value_type>
increasing_subsequence(const Seq& seq, bool strict = true) {
    auto len = distance(begin(seq), end(seq));
    vector<int> best, pred(len);
    auto cmp = [&](int i, int j) { return seq[i] < seq[j]; };
    for (int i = 0; i < len; ++i) {
        auto find = strict
            ? lower_bound(best.begin(), best.end(), i, cmp)
            : upper_bound(best.begin(), best.end(), i, cmp);
        pred[i] = find == best.begin() ? -1 : *prev(find);
        if (find == best.end()) {
            best.push_back(i);
        } else {
            *find = i;
        }
    }
    // Can return best.size() here if just need length.
    vector<typename Seq::value_type> res(best.size());
    int curr = best.empty() ? -1 : best.back();
    for (int i = best.size(); i > 0; curr = pred[curr])
        res[--i] = seq[curr];
    return res;
}
```

```cpp
/*
 * Binary search finds the smallest integer x in [0,n) for which f(x)
 * is true, else n if f(x) is never true. O(lg n).
 *
 * Ternary search finds the x in [l,r] that maximizes f(x) assuming
 * that f is increasing then decreasing. O(lg n). The search
 * interval is (l,r) in the floating-point case.
 *
 * l := left
 * r := right
 * m := mid
 * s := step size
 */
constexpr double eps = 1e-9;

template <typename F>
int binary_search_lr(int n, F f) {
    int l = 0, r = n-1;
    while (l < r) {
        int m = (l + r) / 2;
        f(m) ? r = m : l = m+1;
    }
    return l;
}

template <typename F>
int binary_search_bit(int n, F f) {
    int l = -1, s = 1;
    while (s < n)
        s <<= 1;
    for (; s > 0; s >>= 1)
        if (l + s < n && !f(l + s))
            l += s;
    return l + 1;
}

template <typename F>
int ternary_search(int l, int r, F f) {
    while (l < r) {
        int m = (l + r) / 2;
        f(m) > f(m+1) ? r = m : l = m+1;
    }
    return l;
}

template <typename F>
double ternary_search(double l, double r, F f) {
    while (r - l > eps) {
        auto third = (r - l) / 3;
        auto m1 = l + third;
        auto m2 = r - third;
        f(m1) > f(m2) ? r = m2 : l = m1;
    }
    return (l + r) / 2;
}
```

```cpp
/*
 * Binary indexed tree.
 */
template <typename T>
struct bit {
    int n;
    vector<T> v;

    bit(int n): n(n+1), v(n+1) {}

    void update(int i, T t) {
        for (++i; i < n; i += i & -i) {
            v[i] += t;
        }
    }

    T query(int i) {
        T sum = 0;
        for (++i; i > 0; i -= i & -i)
            sum += v[i];
        return sum;
    }

    T query(int l, int r) {
        return l <= r ? query(r) - query(l-1) : 0;
    }

    int lower_bound(T x) {
        int i = 0, step = 1;
        while (step < n)
            step <<= 1;
        for (; step > 0; step >>= 1)
            if (i + step < n && v[i + step] < x)
                i += step, x -= v[i];
        return i;
    }
};
```

```cpp
/*
 * 2-dimensional binary indexed tree.
 */
template <typename T>
struct bit_2d {
    int n, m;
    vector<vector<T>> v;

    bit_2d(int n, int m): n(n+1), m(m+1), v(n+1, vector<T>(m+1)) {}

    void update(int x, int y, T t) {
        for (int i = x + 1; i < n; i += i & -i) {
            for (int j = y + 1; j < m; j += j & -j) {
                v[i][j] += t;
            }
        }
    }

    T query(int x, int y) {
        T sum = 0;
        for (int i = x + 1; i > 0; i -= i & -i)
            for (int j = y + 1; j > 0; j -= j & -j)
                sum += v[i][j];
        return sum;
    }

    T query(int x1, int y1, int x2, int y2) {
        if (x2 < x1 || y2 < y1)
            return 0;
        return query(x2, y2) - query(x2, y1-1)
                             - query(x1-1, y2) + query(x1-1, y1-1);
    }
};
```

```cpp
/*
 * Persistent binary indexed tree.
 * Queries with timestamp i will compute the result
 *   as if only the first i updates have been applied.
 */
template <typename T>
struct bit_persistent {
    using History = vector<pair<int,T>>;
    vector<History> v;
    int ts;

    bit_persistent(int n)
        : v(n+1, History({make_pair(0, T(0))}))
        , ts(0) {}

    void update(int i, T t) {
        for (++ts, ++i; i < v.size(); i += i & -i) {
            v[i].emplace_back(ts, v[i].back().second + t);
        }
    }

    T query(int i, int time) {
        T sum = 0;
        auto key = make_pair(time, numeric_limits<T>::max());
        for (++i; i > 0; i -= i & -i)
            sum += prev(upper_bound(v[i].begin(), v[i].end(), key))->second;
        return sum;
    }

    T query(int l, int r, int time) {
        return l <= r ? query(r, time) - query(l-1, time) : 0;
    }
};
```

```cpp
/*
 * Binary indexed tree, supporting range updates and point queries.
 */
template <typename T>
struct bit_range {
    vector<T> v;

    bit_range(int n): v(n+1) {}

    void update(int i, T t) {
        for (++i; i < v.size(); i += i & -i) {
            v[i] += t;
        }
    }

    void update(int a, int b, T t) {
        update(a, t);
        update(b+1, -t);
    }

    T query(int i) {
        T sum = 0;
        for (++i; i > 0; i -= i & -i)
            sum += v[i];
        return sum;
    }
};
```

```cpp
/*
 * Segment tree supporting an arbitrary monoid.
 */
template <typename Monoid>
struct seg_tree {
    using T = typename Monoid::T;
    Monoid m;
    int s;
    vector<T> v;

    seg_tree(int n, Monoid m = Monoid()): m(m) {
        for (s = 1; s < n; )
            s <<= 1;
        v.resize(2*s, T(m.id));
    }

    template <typename It>
    void set_leaves(It begin, It end) {
        copy(begin, end, v.begin() + s);
        for (int i = s - 1; i >= 0; --i) {
            v[i] = m.op(v[2*i], v[2*i+1]);
        }
    }

    void update(int i, T t) {
        i += s;
        v[i] = t;
        for (i /= 2; i > 0; i /= 2) {
            v[i] = m.op(v[2*i], v[2*i+1]);
        }
    }

    T query(int i, int j) {
        i += s, j += s;
        T l = m.id, r = m.id;
        for (; i <= j; i /= 2, j /= 2) {
            if (i % 2 == 1) l = m.op(l, v[i++]);
            if (j % 2 == 0) r = m.op(v[j--], r);
        }
        return m.op(l, r);
    }
};

struct monoid {
    using T = int;
    static constexpr T id = numeric_limits<T>::min();
    static T op(T a, T b) { return max(a, b); }
};
```

```cpp
/*
 * Segment tree supporting delta updates.
 */
template <typename M>
struct seg_tree_delta {
    M m;
    int s;
    vector<typename M::State> v;

    seg_tree_delta(int n, M m = M()): m(m) {
        for (s = 1; s < n; )
            s <<= 1;
        v.resize(2*s);
    }

    void update(int i, typename M::Update u) {
        for (i += s; i > 0; i /= 2) {
            m.update(v[i], u);
        }
    }

    typename M::Res query(int i, int j, typename M::Query q) {
        i += s, j += s;
        auto l = m.id, r = m.id;
        for (; i <= j; i /= 2, j /= 2) {
            if (i % 2 == 1) l = m.compose(l, m.query(v[i++], q));
            if (j % 2 == 0) r = m.compose(m.query(v[j--], q), r);
        }
        return m.compose(l, r);
    }
};

struct M {
    using State = unordered_map<int,int>;
    using Update = int;
    using Query = int;
    using Res = int;
    static constexpr Res id = 0;

    static void update(State& state, Update val) {
        ++state[val];
    }

    static Res query(State& state, Query val) {
        return state[val];
    }

    static Res compose(Res a, Res b) {
        return a + b;
    }
};
```

```cpp
/*
 * Lazy segment tree supporting an arbitrary monoid.
 */
template <typename Monoid>
struct seg_tree_lazy {
    using T = typename Monoid::T;
    using Update = typename Monoid::update;
    Monoid m;
    int s, h;
    vector<T> v;
    vector<Update> lazy;

    seg_tree_lazy(int n, Monoid m = Monoid()): m(m) {
        for (s = 1, h = 1; s < n; )
            s <<= 1, ++h;
        v.resize(2*s, T(m.id));
        lazy.resize(s);
    }

    template <typename InputIt>
    void set_leaves(InputIt begin, InputIt end) {
        copy(begin, end, v.begin() + s);
        for (int i = s - 1; i >= 0; --i) {
            v[i] = m.op(v[2*i], v[2*i+1]);
        }
    }

    void apply(int i, int d, const Update& u) {
        v[i] = u.apply(v[i], d);
        if (i < s) {
            lazy[i] = u.compose(lazy[i]);
        }
    }

    void push(int i) {
        for (int d = h; d > 0; --d) {
            int l = i >> d;
            if (lazy[l]) {
                apply(2*l,   d-1, lazy[l]);
                apply(2*l+1, d-1, lazy[l]);
                lazy[l] = Update();
            }
        }
    }

    void pull(int i) {
        for (int d = 1; d <= h; ++d) {
            int l = i >> d;
            T combined = m.op(v[2*l], v[2*l+1]);
            v[l] = lazy[l].apply(combined, d);
        }
    }

    void update(int i, int j, const Update& u) {
        i += s, j += s;
        push(i), push(j); // Needed if updates are not commutative.
        for (int l = i, r = j, d = 0; l <= r; l /= 2, r /= 2, ++d) {
            if (l % 2 == 1) apply(l++, d, u);
            if (r % 2 == 0) apply(r--, d, u);
        }
        pull(i), pull(j);
    }

    T query(int i, int j) {
        i += s, j += s;
        push(i), push(j);
        T l = m.id, r = m.id;
        for (; i <= j; i /= 2, j /= 2) {
            if (i % 2 == 1) l = m.op(l, v[i++]);
            if (j % 2 == 0) r = m.op(v[j--], r);
```

```
        }
        return m.op(l, r);
    }
};

struct monoid {
    using T = int;
    static constexpr T id = 0;
    static T op(T a, T b) { return a + b; }
    static T exp(T x, int e) { return x * e; }
    struct update;
};

struct monoid::update {
    enum kind { kNoop, kSet, kOperate };
    kind k;
    T x;

    update(kind k = kNoop, T x = id): k(k), x(x) {}

    explicit operator bool() const { return k != kNoop; }

    T apply(T n, int d) const {
        switch (k) {
            case kNoop:    return n;
            case kSet:     return exp(x, 1 << d);
            case kOperate: return op(n, exp(x, 1 << d));
        }
    }

    update compose(const update& other) const {
        if (other.k == kNoop || k == kSet) {
            return *this;
        } else if (k == kNoop) {
            return other;
        } else {
            assert(k == kOperate);
            return update(other.k, op(other.x, x));
        }
    }
};
```

```cpp
/*
 * Union-find for integers 0 to n-1.
 * p := pointer
 * s := size
 * r := rank (approximate tree depth)
 * c := count (number of disjoint sets)
 */
struct union_find {
    vector<int> p, s, r;
    int c;

    union_find(int n): p(n), s(n, 1), r(n), c(n) {
        for (int i = 0; i < n; ++i) {
            p[i] = i;
        }
    }

    int rep(int i) {
        return p[i] == i ? i : p[i] = rep(p[i]);
    }

    void merge(int a, int b) {
        a = rep(a), b = rep(b);
        if (a == b)
            return;
        if (r[a] > r[b])
            swap(a, b);
        p[a] = b;
        s[b] += s[a];
        if (r[a] == r[b])
            ++r[b];
        --c;
    }
};
```

```cpp
/*
 * Bipartite matching using Kuhn's algorithm.
 * Runs in O(VE), but often faster.
 * Nodes on both sides are 0-indexed.
 * a := number of left nodes
 * b := number of right nodes
 */
struct graph {
    int a, b;
    vector<vector<int>> adj;
    vector<int> visit, match;

    graph(int a, int b)
        : a(a), b(b), adj(a+b), visit(a+b, -1), match(a+b, -1) {}

    void edge(int i, int j) {
        adj[i].push_back(a+j);
        adj[a+j].push_back(i);
    }

    bool augment(int n, int run) {
        if (visit[n] == run)
            return false;
        visit[n] = run;
        for (int c : adj[n]) {
            if (match[c] == -1 || augment(match[c], run)) {
                match[n] = c, match[c] = n;
                return true;
            }
        }
        return false;
    }

    int matching() {
        int ret = 0;
        for (int i = 0; i < a; ++i)
            ret += augment(i, i);
        return ret;
    }
};
```

```cpp
/*
 * A graph data structure with shortest path algorithms.
 * Supports directed, weighted, and duplicate edges.
 * n := number of nodes
 * x := current node
 * c := child (neighbor) of current node
 * w := edge weight
 *
 * Dijkstra: ~ O(E log V)
 * - returns distances from a source node s to every other
 * - negative edges not supported
 *
 * Bellman-Ford: ~ O(VE)
 * - returns distances from a source node s to every other
 * - supports negative edges
 *
 * Floyd-Warshall: O(V^3)
 * - returns distances between every pair of nodes
 * - supports negative edges
 *
 * Johnson's Algorithm: ~ O(V^2 log V + VE)
 * - returns distances between every pair of nodes
 * - better than Floyd-Warshall for sparse graphs with negative edges
 *
 * Mst:
 * - returns the weight of the minimum spanning tree
 */
template <typename T>
struct graph {
    int n;
    vector<vector<pair<int,T>>> adj;
    static constexpr T infty = numeric_limits<T>::max();

    graph(int n): n(n), adj(n) {}

    void arc (int i, int j, T w) { adj[i].emplace_back(j, w); }
    void edge(int i, int j, T w) { arc(i, j, w), arc(j, i, w); }

    pair<vector<T>, vector<int>>
    dijkstra(int s) const;

    pair<vector<T>, vector<int>>
    bellman_ford(int s, bool* negative_cycle) const;

    pair<vector<vector<T>>, vector<vector<int>>>
    floyd_warshall() const;

    pair<vector<vector<T>>, vector<vector<int>>>
    johnsons_algorithm(bool* negative_cycle) const;

    T mst() const;
};

template <typename T>
pair<vector<T>, vector<int>>
graph<T>::dijkstra(int s) const {
    vector<T> dist(n, T(infty));
    vector<int> prev(n, -1);
    vector<bool> done(n);
    priority_queue<pair<T,int>,
                   vector<pair<T,int>>,
                   greater<pair<T,int>>> q;
    q.emplace(0, s);
    dist[s] = 0;

    while (!q.empty()) {
        int x = q.top().second;
        q.pop();
        if (done[x])
            continue;
```

```
        done[x] = true;
        for (auto& p : adj[x]) {
            int c; T w;
            tie(c, w) = p;
            if (!done[c] && dist[c] > dist[x] + w) {
                dist[c] = dist[x] + w;
                prev[c] = x;
                q.emplace(dist[c], c);
            }
        }
    }

    return make_pair(dist, prev);
}

template <typename T>
pair<vector<T>, vector<int>>
graph<T>::bellman_ford(int s, bool* negative_cycle) const {
    *negative_cycle = false;
    vector<T> dist(n, T(infty));
    vector<int> prev(n, -1);
    dist[s] = 0;

    for (int i = 0; i < n; ++i)
        for (int x = 0; x < n; ++x)
            for (auto& p : adj[x])
                if (dist[x] < infty)
                    if (dist[p.first] > dist[x] + p.second) {
                        dist[p.first] = dist[x] + p.second;
                        prev[p.first] = x;
                        if (i == n - 1) {
                            *negative_cycle = true;
                        }
                    }

    return make_pair(dist, prev);
}

template <typename T>
pair<vector<vector<T>>, vector<vector<int>>>
graph<T>::floyd_warshall() const {
    vector<vector<T>> dist(n, vector<T>(n, T(infty)));
    vector<vector<int>> next(n, vector<int>(n, -1));

    for (int i = 0; i < n; ++i) {
        dist[i][i] = 0;
        for (auto& p : adj[i]) {
            dist[i][p.first] = min(dist[i][p.first], p.second);
            next[i][p.first] = p.first;
        }
    }

    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (dist[i][k] < infty && dist[k][j] < infty)
                    if (dist[i][j] > dist[i][k] + dist[k][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                        next[i][j] = next[i][k];
                    }

    return make_pair(dist, next);
}

template <typename T>
pair<vector<vector<T>>, vector<vector<int>>>
graph<T>::johnsons_algorithm(bool* negative_cycle) const {
    *negative_cycle = false;
    vector<vector<T>> dist(n);
    vector<vector<int>> prev(n);
```

```
    graph<T> alt = *this;
    alt.n += 1;
    alt.adj.resize(alt.n);
    for (int i = 0; i < n; ++i)
        alt.arc(n, i, 0);
    auto q_dist = alt.bellman_ford(n, negative_cycle).first;
    if (*negative_cycle)
        return make_pair(dist, prev);

    alt.n -= 1;
    alt.adj.resize(alt.n);
    for (int i = 0; i < n; ++i)
        for (auto& p : alt.adj[i])
            p.second += q_dist[i] - q_dist[p.first];

    for (int s = 0; s < n; ++s)
        tie(dist[s], prev[s]) = alt.dijkstra(s);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (dist[i][j] < infty)
                dist[i][j] -= q_dist[i] - q_dist[j];

    return make_pair(dist, prev);
}

template <typename T>
T graph<T>::mst() const {
    vector<bool> done(n);
    priority_queue<pair<T,int>,
                   vector<pair<T,int>>,
                   greater<pair<T,int>>> q;
    q.emplace(0, 0);
    T ret = 0;

    while (!q.empty()) {
        T w; int c;
        tie(w, c) = q.top();
        q.pop();
        if (!done[c]) {
            done[c] = true;
            ret += w;
            for (auto& p : adj[c]) {
                q.emplace(p.second, p.first);
            }
        }
    }

    return ret;
}
```

```cpp
/*
 * A graph data structure and various algorithms.
 * All traversals are non-recursive.
 * n := number of nodes
 * counter := auxiliary counter for traversal isolation
 * visit := auxiliary array for tracking visited nodes
 * next := auxiliary array for tracking next DFS child to visit
 * p := parent node
 * x := current node
 * nbr := neighbor of the current node
 * cc := connected components
 * scc := strongly connected components
 * lca := lowest common ancestor
 */
void noop(int p, int x) {}

struct graph {
    int n;
    vector<vector<int>> adj;
    mutable int counter;
    mutable vector<int> visit, next;

    graph(int n): n(n), adj(n), counter(0), visit(n), next(n) {}

    void arc (int i, int j) { adj[i].emplace_back(j); }
    void edge(int i, int j) { arc(i, j), arc(j, i); }

    template <typename Pre, typename Post>
    void dfs(int s, Pre pre, Post post) const;

    template <typename Pre, typename Post>
    void dfs(Pre pre, Post post) const;

    template <typename F>
    void bfs(int s, F f) const;

    template <typename F>
    void bfs(F f) const;

    template <typename F>
    void euler_tour(int s, F f) const;

    int cc() const;

    vector<int> scc() const;

    pair<bool, vector<bool>> bipartite() const;

    vector<int> articulation_points() const;

    graph dfs_tree(int root) const;

    vector<pair<int,int>> push_pop_order(int root) const;

    function<int(int,int)> lca(int root) const;
};

template <typename Pre, typename Post>
void graph::dfs(int s, Pre pre, Post post) const {
    ++counter;
    if (visit[s] == counter)
        return;
    stack<pair<int,int>> q;
    q.emplace(-1, s);
    pre(-1, s);
    visit[s] = counter;
    next[s] = 0;
    while (!q.empty()) {
        int p, x;
        tie(p, x) = q.top();
```

```
        if (next[x] == adj[x].size()) {
            q.pop();
            post(p, x);
        } else while (next[x] < adj[x].size()) {
            int nbr = adj[x][next[x]++];
            if (visit[nbr] != counter) {
                q.emplace(x, nbr);
                pre(x, nbr);
                visit[nbr] = counter;
                next[nbr] = 0;
                break;
            }
        }
    }
}

template <typename Pre, typename Post>
void graph::dfs(Pre pre, Post post) const {
    for (int i = 0; i < n; ++i) {
        if (i) --counter;
        dfs(i, pre, post);
    }
}

template <typename F>
void graph::bfs(int s, F f) const {
    ++counter;
    if (visit[s] == counter)
        return;
    queue<pair<int,int>> q;
    q.emplace(-1, s);
    visit[s] = counter;
    while (!q.empty()) {
        int p, x;
        tie(p, x) = q.front();
        q.pop();
        f(p, x);
        for (auto nbr : adj[x]) {
            if (visit[nbr] != counter) {
                q.emplace(x, nbr);
                visit[nbr] = counter;
            }
        }
    }
}

template <typename F>
void graph::bfs(F f) const {
    for (int i = 0; i < n; ++i) {
        if (i) --counter;
        bfs(i, f);
    }
}

template <typename F>
void graph::euler_tour(int s, F f) const {
    auto pre  = [&](int p, int x) { if (p != -1) f(p); };
    auto post = [&](int p, int x) { f(x); };
    dfs(s, pre, post);
}

int graph::cc() const {
    int count = 0;
    bfs([&](int p, int x) { if (p == -1) ++count; });
    return count;
}

graph graph::dfs_tree(int root) const {
    graph g(n);
    dfs(root, noop, [&](int p, int x) { if (p != -1) g.arc(p, x); });
```

```
    return g;
}

pair<bool, vector<bool>> graph::bipartite() const {
    vector<bool> color(n);
    bfs([&](int p, int x) {
        color[x] = p == -1 ? false : !color[p];
    });
    bool ok = true;
    for (int i = 0; i < n; ++i)
        for (auto nbr : adj[i])
            ok &= color[i] != color[nbr];
    return make_pair(ok, color);
}

vector<int> graph::articulation_points() const {
    // To find bridges, must also check for cliques of size 2.
    vector<int> parent(n), dist(n), low(n);
    vector<int> points;

    auto pre = [&](int p, int x) {
        parent[x] = p;
        dist[x] = p == -1 ? 1 : dist[p] + 1;
    };

    auto post = [&](int p, int x) {
        if (p == -1) {
            // Root vertex is an articulation point if it has
            // more than one DFS child.
            int deg = 0;
            for (auto nbr : adj[x])
                if (parent[nbr] == x)
                    ++deg;
            if (deg > 1) {
                points.push_back(x);
            }
        } else {
            // Non-root vertex x is an articulation point if some
            // DFS child nbr exists with low[nbr] >= dist[x].
            low[x] = dist[x];
            for (auto nbr : adj[x]) {
                if (nbr != p)
                    low[x] = min(low[x], dist[nbr]);
                if (parent[nbr] == x) {
                    low[x] = min(low[x], low[nbr]);
                    if (low[nbr] >= dist[x]) {
                        points.push_back(x);
                    }
                }
            }
        }
    };

    dfs(pre, post);
    sort(points.begin(), points.end());
    points.erase(unique(points.begin(), points.end()), points.end());
    return points;
}

vector<int> graph::scc() const {
    vector<int> q(n);
    int idx = n;
    dfs(noop, [&](int p, int x) { q[--idx] = x; });
    graph transpose(n);
    for (int i = 0; i < n; ++i)
        for (auto nbr : adj[i])
            transpose.arc(nbr, i);
    vector<int> res(n, -1);
    for (int i = 0; i < n; ++i) {
        if (i) --transpose.counter; // Keep marked nodes marked.
```

```
        transpose.bfs(q[i], [&](int p, int x) { res[x] = q[i]; });
    }
    return res;
}

vector<pair<int,int>> graph::push_pop_order(int root) const {
    vector<pair<int,int>> order(n);
    int i = 0;
    auto pre  = [&](int p, int x) { order[x].first  = i++; };
    auto post = [&](int p, int x) { order[x].second = i++; };
    dfs(root, pre, post);
    return order;
}

function<int(int,int)> graph::lca(int root) const {
    int log = 0;
    while (1 << log < n)
        ++log;

    // dp[k][i] := (2^k)th parent of node i, or -1 if none.
    vector<vector<int>> dp(log + 1, vector<int>(n, -1));
    bfs(root, [&](int p, int x) { dp[0][x] = p; });
    for (int k = 1; k <= log; ++k)
        for (int i = 0; i < n; ++i)
            if (dp[k-1][i] != -1)
                dp[k][i] = dp[k-1][dp[k-1][i]];

    vector<int> dist(n);
    bfs(root, [&](int p, int x) {
        dist[x] = p == -1 ? 0 : dist[p] + 1;
    });

    return [=](int a, int b) {
        if (dist[a] < dist[b])
            swap(a, b);
        for (int k = log; k >= 0; --k)
            if (dp[k][a] != -1 && dist[dp[k][a]] >= dist[b])
                a = dp[k][a];
        for (int k = log; k >= 0; --k)
            if (dp[k][a] != dp[k][b])
                a = dp[k][a], b = dp[k][b];
        return a == b ? a : dp[0][a];
    };
}
```

```
constexpr ll mod = 1000000000 + 7;

constexpr size_t max_fact = 1000000;
ll fact[max_fact + 1];
void precomp_fact() {
    fact[0] = fact[1] = 1;
    for (ll i = 2; i <= max_fact; ++i) {
        fact[i] = fact[i-1] * i % mod;
    }
}

constexpr size_t max_ncrf = 1000;
double ncrf[max_ncrf + 1][max_ncrf + 1];
void precomp_ncrf() {
    for (size_t i = 0; i <= max_ncrf; ++i) {
        ncrf[i][0] = ncrf[i][i] = 1.;
        for (size_t j = 1; j < i; ++j)
            ncrf[i][j] = ncrf[i-1][j-1] + ncrf[i-1][j];
    }
}

ll pow(ll b, ll e) {
    if (e == 0) return 1;
    ll rec = pow(b * b % mod, e / 2);
    return e % 2 ? rec * b % mod : rec;
}

ll inverse(ll n) {
    return pow(n, mod - 2);
}

// For n on the order of 1e6.
ll ncr(ll n, ll r) {
    return n < r ? 0 : fact[n] * inverse(fact[r] * fact[n - r] % mod) % mod;
}

// For very large n and relatively small r.
ll ncrx(ll n, ll r) {
    if (n < r || n >= mod)
        return 0;
    ll num = 1, denom = 1;
    for (ll i = 0; i < r; ++i) {
        denom = denom * (i + 1) % mod;
        num = num * (n - i) % mod;
    }
    return num * inverse(denom) % mod;
}
```

```
ll gcd(ll x, ll y) { return y ? gcd(y, x % y) : abs(x); }

struct frac {
    ll a, b;

    frac(ll aa = 1, ll bb = 1): a(aa), b(bb) {
        if (b < 0)
            a = -a, b = -b;
        ll d = gcd(a, b);
        a /= d, b /= d;
    }

    frac operator+(frac o) {
        return frac(a * o.b + o.a * b, b * o.b);
    }

    frac operator-(frac o) {
        return frac(a * o.b - o.a * b, b * o.b);
    }

    frac operator*(frac o) {
        return frac(a * o.a, b * o.b);
    }

    friend ostream& operator<<(ostream& os, frac f) {
        return f.b == 1
            ? os << f.a
            : os << f.a << '/' << f.b;
    }
};
```

```cpp
template <typename T>
struct matrix {
    int n;
    vector<T> v;

    matrix<T>(int n, T diag = 0, T fill = 0)
        : n(n), v(n * n, fill)
    {
        for (int i = 0; i < n; i++)
            (*this)(i, i) = diag;
    }

    matrix<T>(int n, initializer_list<T> l): n(n), v(l) { v.resize(n * n); }

    T&       operator()(int i, int j)       { return v[n * i + j]; }
    const T& operator()(int i, int j) const { return v[n * i + j]; }

    matrix<T> operator+(const matrix<T>& b) const {
        matrix<T> ret = b;
        for (int i = 0; i < n * n; ++i)
            ret.v[i] += v[i];
        return ret;
    }

    matrix<T> operator*(const matrix<T>& b) const {
        matrix<T> ret(n);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                for (int k = 0; k < n; ++k)
                    ret(i, j) += (*this)(i, k) * b(k, j);
        return ret;
    }

    matrix<T> pow(ll e) const {
        if (e == 0) return matrix<T>(n, 1);
        matrix<T> rec = (*this * *this).pow(e / 2);
        return e % 2 ? rec * *this : rec;
    }

    bool operator==(const matrix<T>& b) const { return v == b.v; }
    bool operator!=(const matrix<T>& b) const { return v != b.v; }
};

template <typename T>
ostream& operator<<(ostream& os, const matrix<T>& m) {
    for (int i = 0; i < m.n; ++i) {
        for (int j = 0; j < m.n; ++j) {
            if (j) cout << " ";
            cout << m.v[m.n * i + j];
        }
        cout << endl;
    }
    return os;
}
```

```cpp
int gcd(int x, int y) { return y ? gcd(y, x % y) : abs(x); }

int lcm(int x, int y) { return x && y ? abs(x) / gcd(x, y) * abs(y) : 0; }

// Finds a and b such that a*x + b*y == gcd(x,y).
pair<int,int> bezout(int x, int y) {
    if (y == 0) return pair<int,int>(1, 0);
    pair<int,int> u = bezout(y, x % y);
    return pair<int,int>(u.second, u.first - (x/y) * u.second);
}

vector<bool> sieve(int n) {
    if (n == 0)
        return vector<bool>(1);
    vector<bool> prime(n+1, true);
    prime[0] = prime[1] = false;
    for (int i = 2; i <= n; ++i)
        if (prime[i])
            for (int j = i + i; j <= n; j += i)
                prime[j] = false;
    return prime;
}

// Returns all primes up to and including n.
vector<int> primes(int n) {
    auto s = sieve(n);
    vector<int> p;
    for (int i = 0; i <= n; ++i)
        if (s[i])
            p.emplace_back(i);
    return p;
}

// Note: can precompute primes if needed.
vector<pair<int,int>> prime_factors(int n) {
    vector<pair<int,int>> factors;
    for (int i = 2; i*i <= n; ++i) {
        if (n % i == 0) {
            int pow = 0;
            while (n % i == 0)
                ++pow, n /= i;
            factors.emplace_back(i, pow);
        }
    }
    if (n > 1)
        factors.emplace_back(n, 1);
    return factors;
}

vector<int> divisors(int n) {
    vector<int> small, large;
    for (int i = 1; i*i <= n; ++i) {
        if (n % i == 0) {
            small.emplace_back(i);
            if (i*i != n) {
                large.emplace_back(n / i);
            }
        }
    }
    copy(large.rbegin(), large.rend(), back_inserter(small));
    return small;
}
```

```cpp
/*
 * Knuth-Morris-Pratt algorithm.
 * O(n) precomputation and O(m) query, where n is the size of the
 * needle to find, and m is the size of the haystack.
 * Assumes n > 0.
 */
template <typename Container>
struct kmp {
    const Container needle;
    int n;
    vector<int> succ;

    kmp(const Container needle)
        : needle(needle)
        , n(distance(begin(needle), end(needle)))
        , succ(n + 1)
    {
        succ[0] = -1;
        succ[1] = 0;
        for (int cur = 0, i = 2; i < n; ) {
            if (needle[i-1] == needle[cur]) succ[i++] = ++cur;
            else if (cur) cur = succ[cur];
            else succ[i++] = 0;
        }
    }

    vector<int> find(const Container& haystack) const {
        vector<int> res;
        int len = distance(begin(haystack), end(haystack));
        for (int m = 0, i = 0; m + i < len; ) {
            if (needle[i] == haystack[m + i]) {
                if (i == n - 1)
                    res.emplace_back(m);
                ++i;
            } else if (succ[i] != -1) {
                m = m + i - succ[i];
                i = succ[i];
            } else {
                ++m;
                i = 0;
            }
        }
        return res;
    }

    bool in(const Container& haystack) const {
        return !find(haystack).empty();
    }
};
```

```cpp
/*
 * Suffix array for an arbitrary sequence of values.
 * Gives the lexicographic ordering of all suffixes of the input sequence,
 *   enabling quick operations such as longest common prefix (LCP).
 * O(n lg n lg n) construction, but reducible to O(n lg n) with counting sort.
 * O(lg n) to find the longest common prefix of two suffixes.
 * O(|W| lg n) to search for a substring W.
 *
 * v := the sequence of values (usually a string)
 * log := lg(n) (ceil)
 * dp[k][i] := sorted position of substring i among (2^k)-length substrings.
 * order[i] := the start index of the ith lexicographically lowest suffix
 */
template <typename Sequence>
struct suffix_array {
    const Sequence v;
    int n, log;
    vector<vector<int>> dp;
    vector<int> order;

    suffix_array(const Sequence v)
        : v(v), n(distance(begin(v), end(v))), log(0), order(n)
    {
        while (1 << log < n)
            ++log;
        dp.resize(log + 1, vector<int>(n));
        for (int i = 0; i < n; ++i)
            order[i] = i;

        for (int pow = 0; pow <= log; ++pow) {
            auto cmp = [&](int i, int j) {
                if (pow == 0)
                    return v[i] < v[j];
                if (dp[pow-1][i] != dp[pow-1][j])
                    return dp[pow-1][i] < dp[pow-1][j];
                auto step = 1 << (pow-1);
                if (i + step < n && j + step < n)
                    return dp[pow-1][i+step] < dp[pow-1][j+step];
                return i > j;
            };
            sort(order.begin(), order.end(), cmp);
            for (int i = 0; i < n; ++i) {
                auto diff = i == 0 || cmp(order[i-1], order[i]);
                dp[pow][order[i]] = diff ? i : dp[pow][order[i-1]];
            }
        }
    }

    int lcp(int i, int j) {
        int len = 0;
        for (int pow = log; pow >= 0 && i < n && j < n; --pow)
            if (dp[pow][i] == dp[pow][j])
                len += 1 << pow, i += 1 << pow, j += 1 << pow;
        return len;
    }

    int lower_bound(const Sequence& key) {
        int i = -1;
        for (int step = 1 << log; step > 0; step >>= 1)
            if (i + step < n)
                if (lexicographical_compare(
                        begin(v) + order[i+step], end(v),
                        begin(key), end(key)))
                    i += step;
        return i + 1;
    }

    bool has_substr(const Sequence& key) {
        int lb = lower_bound(key);
        return distance(begin(key), end(key)) <= n - order[lb]
```

```
            && equal(begin(key), end(key), begin(v) + order[lb]);
    }
};
```

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

using ll = long long;

#define FU(i, a, b) for (int i = (a); i < (b); ++i)
#define fu(i, b) FU(i, 0, b)
#define FD(i, a, b) for (int i = (int) (b) - 1; i >= (a); --i)
#define fd(i, b) FD(i, 0, b)
#define all(x) (x).begin(), (x).end()

#define TRACE(x) x
#define WATCH(x) TRACE(cout << #x" = " << x << endl)
#define WATCHR(b, e) TRACE( \
    for (auto it = b; it != e; ++it) \
        { if (it != b) cout << " "; cout << *it; } \
    cout << endl)
#define WATCHC(v) TRACE({cout << #v" = "; WATCHR(v.begin(), v.end());})

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0); cout.tie(0);
    cout << fixed << setprecision(6);
    return 0;
}
```