Team Reference Document

tourist

2024 年 10 月 17 日

目录

1 data

1.1 bstnode.cpp

```
class node {
 public:
  int id;
  node* l;
  node* r;
  node* p;
  bool rev;
  int sz;
  // declare extra variables:


  node(int _id) {
    id = _id;
    l = r = p = nullptr;
    rev = false;
    sz = 1;
    // init extra variables:

  }

  // push everything else:
  void push_stuff() {

  }

  void unsafe_reverse() {
    push_stuff(); // !! edu 112
    rev ^= 1;
    swap(l, r);
    pull();
  }

  // apply changes:
  void unsafe_apply() {

  }

  void push() {
    if (rev) {
      if (l != nullptr) {
        l->unsafe_reverse();
      }
      if (r != nullptr) {
        r->unsafe_reverse();
      }
      rev = 0;
    }
    push_stuff();
  }

  void pull() {
    sz = 1;
    // now init from self:

    if (l != nullptr) {
      l->p = this;
      sz += l->sz;
      // now pull from l:

    }
    if (r != nullptr) {
      r->p = this;
      sz += r->sz;
```

```
      // now pull from r:

    }
  }
};

void debug_node(node* v, string pref = "") {
  #ifdef LOCAL
    if (v != nullptr) {
      debug_node(v->r, pref + "␣");
      cerr << pref << "-" << "␣" << v->id << '\n';
      debug_node(v->l, pref + "␣");
    } else {
      cerr << pref << "-" << "␣" << "nullptr" << '\n';
    }
  #endif
}
```

1.2 disjointsparsetable.cpp

```
template <typename T, typename F>
class DisjointSparseTable {
 public:
  int n;
  vector<vector<T>> mat;
  F func;

  DisjointSparseTable(const vector<T>& a, const F& f) : n(int(a.size())),
      func(f) {
    mat.push_back(a);
    for (int p = 1; (1 << p) < n; p++) {
      mat.emplace_back(n);
      for (int mid = 1 << p; mid < n; mid += 1 << (p + 1)) {
        mat[p][mid - 1] = a[mid - 1];
        for (int j = mid - 2; j >= mid - (1 << p); j--) {
          mat[p][j] = func(a[j], mat[p][j + 1]);
        }
        mat[p][mid] = a[mid];
        for (int j = mid + 1; j < min(n, mid + (1 << p)); j++) {
          mat[p][j] = func(mat[p][j - 1], a[j]);
        }
      }
    }
  }

  T Query(int l, int r) const {
    assert(0 <= l && l < r && r <= n);
    if (r - l == 1) {
      return mat[0][l];
    }
    int p = bit_width(unsigned(l ^ (r - 1))) - 1;
    return func(mat[p][l], mat[p][r - 1]);
  }
};
```

1.3 dsu.cpp

```
class dsu {
 public:
  vector<int> p;
  int n;

  dsu(int _n) : n(_n) {
    p.resize(n);
    iota(p.begin(), p.end(), 0);
  }

  inline int get(int x) {
    return (x == p[x] ? x : (p[x] = get(p[x])));
  }

  inline bool unite(int x, int y) {
    x = get(x);
    y = get(y);
    if (x != y) {
      p[x] = y;
      return true;
    }
    return false;
  }
};
```

1.4 fenwick.cpp

```
template <typename T>
class FenwickTree {
```

```
 public:
  vector<T> fenw;
  int n;
  int pw;

  FenwickTree() : n(0) {}
  FenwickTree(int n_) : n(n_) {
    fenw.resize(n);
    pw = bit_floor(unsigned(n));
  }

  void Modify(int x, T v) {
    assert(0 <= x && x < n);
    while (x < n) {
      fenw[x] += v;
      x |= x + 1;
    }
  }

  T Query(int x) {
    assert(0 <= x && x <= n);
    T v{};
    while (x > 0) {
      v += fenw[x - 1];
      x &= x - 1;
    }
    return v;
  }

  // Returns the length of the longest prefix with sum <= c
  int MaxPrefix(T c) {
    T v{};
    int at = 0;
    for (int len = pw; len > 0; len >>= 1) {
      if (at + len <= n) {
        auto nv = v;
        nv += fenw[at + len - 1];
        if (!(c < nv)) {
          v = nv;
          at += len;
        }
      }
    }
    assert(0 <= at && at <= n);
    return at;
  }
};
```

1.5 fenwick2d.cpp

```
template <typename T>
class FenwickTree2D {
 public:
  vector<vector<T>> fenw;
  int n, m;

  FenwickTree() : n(0), m(0) {}
  FenwickTree2D(int n_, int m_) : n(n_), m(m_) {
    fenw.resize(n);
    for (int i = 0; i < n; i++) {
      fenw[i].resize(m);
    }
  }

  void Modify(int i, int j, T v) {
    assert(0 <= i && i < n && 0 <= j && j < m);
    int x = i;
    while (x < n) {
      int y = j;
      while (y < m) {
        fenw[x][y] += v;
        y |= y + 1;
      }
      x |= x + 1;
    }
  }

  T Query(int i, int j) {
    assert(0 <= i && i <= n && 0 <= j && j <= m);
    T v{};
    int x = i;
    while (x > 0) {
      int y = j;
      while (y > 0) {
        v += fenw[x - 1][y - 1];
        y &= y - 1;
```

```
      }
      x &= x - 1;
    }
    return v;
  }
};
```

1.6 fenwicknode.cpp

```
struct FenwickTreeNode {
  ${0}... a = ...;

  inline void operator += (FenwickTreeNode &other) {
    a = ...(a, other.a);
  }

  inline bool operator < (FenwickTreeNode &other) {
    return a < other.a;
  }
};
```

1.7 hashmap.cpp

```
// #include<bits/extc++.h>
#include <ext/pb_ds/assoc_container.hpp>

struct splitmix64_hash {
        static uint64_t splitmix64(uint64_t x) {
                // http://xorshift.di.unimi.it/splitmix64.c
                x += 0x9e3779b97f4a7c15;
                x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
                x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
                return x ^ (x >> 31);
        }

        size_t operator()(uint64_t x) const {
                static const uint64_t FIXED_RANDOM = std::chrono::
                    steady_clock::now().time_since_epoch().count();
                return splitmix64(x + FIXED_RANDOM);
        }
};

template <typename K, typename V, typename Hash = splitmix64_hash>
using HashMap = __gnu_pbds::gp_hash_table<K, V, Hash>;

template <typename K, typename Hash = splitmix64_hash>
using HashSet = HashMap<K, __gnu_pbds::null_type, Hash>;
```

1.8 linkcut.cpp

```
template <bool rooted>
class link_cut_tree {
 public:
  int n;
  vector<node*> nodes;

  link_cut_tree(int _n) : n(_n) {
    nodes.resize(n);
    for (int i = 0; i < n; i++) {
      nodes[i] = new node(i);
    }
  }

  int add_node() {
    int id = (int) nodes.size();
    nodes.push_back(new node(id));
    return id;
  }

  void expose(node* v) {
    node* r = nullptr;
    node* u = v;
    while (u != nullptr) {
      splay(u);
      u->push();
      u->r = r;
      u->pull();
      r = u;
      u = u->p;
    }
    splay(v);
    assert(v->p == nullptr);
  }

  int get_root(int i) {
    node* v = nodes[i];
```

```
    expose(v);
    return get_leftmost(v)->id;
  }

  bool link(int i, int j) { // for rooted: (x, parent[x])
    if (i == j) {
      return false;
    }
    node* v = nodes[i];
    node* u = nodes[j];
    if (rooted) {
      splay(v);
      if (v->p != nullptr || v->l != nullptr) {
        return false; // not a root
      }
    } else {
      make_root(i);
    }
    expose(u);
    if (v->p != nullptr) {
      return false;
    }
    v->p = u;
    return true;
  }

  bool cut(int i, int j) { // for rooted: (x, parent[x])
    if (i == j) {
      return false;
    }
    node* v = nodes[i];
    node* u = nodes[j];
    expose(u);
    splay(v);
    if (v->p != u) {
      if (rooted) {
        return false;
      }
      swap(u, v);
      expose(u);
      splay(v);
      if (v->p != u) {
        return false;
      }
    }
    v->p = nullptr;
    return true;
  }

  bool cut(int i) { // only for rooted
    assert(rooted);
    node* v = nodes[i];
    expose(v);
    v->push();
    if (v->l == nullptr) {
      return false; // already a root
    }
    v->l->p = nullptr;
    v->l = nullptr;
    v->pull();
    return true;
  }

  bool connected(int i, int j) {
    if (i == j) {
      return true;
    }
    node* v = nodes[i];
    node* u = nodes[j];
    expose(v);
    expose(u);
    return v->p != nullptr;
  }

  int lca(int i, int j) {
    if (i == j) {
      return i;
    }
    node* v = nodes[i];
    node* u = nodes[j];
    expose(v);
    expose(u);
    if (v->p == nullptr) {
      return -1;
    }
    splay(v);
```

```
    if (v->p == nullptr) {
      return v->id;
    }
    return v->p->id;
  }

  bool is_ancestor(int i, int j) {
    if (i == j) {
      return true;
    }
    node* v = nodes[i];
    node* u = nodes[j];
    expose(u);
    splay(v);
    return v->p == nullptr && u->p != nullptr;
  }

  void make_root(int i) {
    assert(!rooted);
    node* v = nodes[i];
    expose(v);
    reverse(v);
  }

  node* get_path_from_root(int i) {
    node* v = nodes[i];
    expose(v);
    return v;
  }

  template <typename... T>
  void apply(int i, T... args) {
    node* v = nodes[i];
    splay_tree::apply(v, args...);
  }
};
```

1.9 pbds.cpp

```
#include <ext/pb_ds/assoc_container.hpp>

template <typename K, typename V, typename Comp = std::less<K>>
using ordered_map = __gnu_pbds::tree<
        K, V, Comp,
        __gnu_pbds::rb_tree_tag,
        __gnu_pbds::tree_order_statistics_node_update
>;

template <typename K, typename Comp = std::less<K>>
using ordered_set = ordered_map<K, __gnu_pbds::null_type, Comp>;

// Supports
//  auto iterator = ordered_set().find_by_order(idx); // (0-indexed)
//  int num_strictly_smaller = ordered_set().order_of_key(key);
```

1.10 queue.cpp

```
template <typename T, typename F>
class Queue {
 public:
  vector<T> pref;
  vector<pair<T, T>> suf;
  F func;

  Queue(const F& f) : func(f) {}

  bool Empty() { return pref.empty() && suf.empty(); }
  int Size() { return int(pref.size()) + int(suf.size()); }
  void Clear() { pref.clear(); suf.clear(); }

  void Push(T t) {
    if (suf.empty()) {
      suf.emplace_back(t, t);
    } else {
      suf.emplace_back(t, func(suf.back().second, t));
    }
  }

  void Pop() {
    if (!pref.empty()) {
      pref.pop_back();
      return;
    }
    assert(!suf.empty());
    if (suf.size() > 1) {
      pref.resize(suf.size() - 1);
```

```
      pref[0] = suf.back().first;
      for (int i = 1; i < int(pref.size()); i++) {
        pref[i] = func(suf[int(suf.size()) - 1 - i].first, pref[i - 1]);
      }
    }
    suf.clear();
  }

  T Get() {
    assert(!Empty());
    if (pref.empty()) {
      return suf.back().second;
    }
    if (suf.empty()) {
      return pref.back();
    }
    return func(pref.back(), suf.back().second);
  }
};

template <typename T, typename F>
Queue<T, F> MakeQueue(const F& f) {
  return Queue<T, F>(f);
}
```

1.11 segtree.cpp

```
class segtree {
 public:
  struct node {
    // don't forget to set default value (used for leaves)
    // not necessarily neutral element!
    ... a = ...;

    void apply(int l, int r, ... v) {
      ...
    }
  };

  node unite(const node &a, const node &b) const {
    node res;
    ...
    return res;
  }

  inline void push(int x, int l, int r) {
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    // push from x into (x + 1) and z
    ...
/*
    if (tree[x].add != 0) {
      tree[x + 1].apply(l, y, tree[x].add);
      tree[z].apply(y + 1, r, tree[x].add);
      tree[x].add = 0;
    }
*/
  }

  inline void pull(int x, int z) {
    tree[x] = unite(tree[x + 1], tree[z]);
  }

  int n;
  vector<node> tree;

  void build(int x, int l, int r) {
    if (l == r) {
      return;
    }
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    build(x + 1, l, y);
    build(z, y + 1, r);
    pull(x, z);
  }

  template <typename M>
  void build(int x, int l, int r, const vector<M> &v) {
    if (l == r) {
      tree[x].apply(l, r, v[l]);
      return;
    }
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    build(x + 1, l, y, v);
```

```
    build(z, y + 1, r, v);
    pull(x, z);
  }

  node get(int x, int l, int r, int ll, int rr) {
    if (ll <= l && r <= rr) {
      return tree[x];
    }
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    push(x, l, r);
    node res{};
    if (rr <= y) {
      res = get(x + 1, l, y, ll, rr);
    } else {
      if (ll > y) {
        res = get(z, y + 1, r, ll, rr);
      } else {
        res = unite(get(x + 1, l, y, ll, rr), get(z, y + 1, r, ll, rr));
      }
    }
    pull(x, z);
    return res;
  }

  template <typename... M>
  void modify(int x, int l, int r, int ll, int rr, const M&... v) {
    if (ll <= l && r <= rr) {
      tree[x].apply(l, r, v...);
      return;
    }
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    push(x, l, r);
    if (ll <= y) {
      modify(x + 1, l, y, ll, rr, v...);
    }
    if (rr > y) {
      modify(z, y + 1, r, ll, rr, v...);
    }
    pull(x, z);
  }

  int find_first_knowingly(int x, int l, int r, const function<bool(const
      node&)> &f) {
    if (l == r) {
      return l;
    }
    push(x, l, r);
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    int res;
    if (f(tree[x + 1])) {
      res = find_first_knowingly(x + 1, l, y, f);
    } else {
      res = find_first_knowingly(z, y + 1, r, f);
    }
    pull(x, z);
    return res;
  }

  int find_first(int x, int l, int r, int ll, int rr, const function<bool(
      const node&)> &f) {
    if (ll <= l && r <= rr) {
      if (!f(tree[x])) {
        return -1;
      }
      return find_first_knowingly(x, l, r, f);
    }
    push(x, l, r);
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    int res = -1;
    if (ll <= y) {
      res = find_first(x + 1, l, y, ll, rr, f);
    }
    if (rr > y && res == -1) {
      res = find_first(z, y + 1, r, ll, rr, f);
    }
    pull(x, z);
    return res;
  }

  int find_last_knowingly(int x, int l, int r, const function<bool(const
      node&)> &f) {
    if (l == r) {
```

```
      return l;
    }
    push(x, l, r);
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    int res;
    if (f(tree[z])) {
      res = find_last_knowingly(z, y + 1, r, f);
    } else {
      res = find_last_knowingly(x + 1, l, y, f);
    }
    pull(x, z);
    return res;
  }

  int find_last(int x, int l, int r, int ll, int rr, const function<bool(
      const node&)> &f) {
    if (ll <= l && r <= rr) {
      if (!f(tree[x])) {
        return -1;
      }
      return find_last_knowingly(x, l, r, f);
    }
    push(x, l, r);
    int y = (l + r) >> 1;
    int z = x + ((y - l + 1) << 1);
    int res = -1;
    if (rr > y) {
      res = find_last(z, y + 1, r, ll, rr, f);
    }
    if (ll <= y && res == -1) {
      res = find_last(x + 1, l, y, ll, rr, f);
    }
    pull(x, z);
    return res;
  }

  segtree(int _n) : n(_n) {
    assert(n > 0);
    tree.resize(2 * n - 1);
    build(0, 0, n - 1);
  }

  template <typename M>
  segtree(const vector<M> &v) {
    n = v.size();
    assert(n > 0);
    tree.resize(2 * n - 1);
    build(0, 0, n - 1, v);
  }

  node get(int ll, int rr) {
    assert(0 <= ll && ll <= rr && rr <= n - 1);
    return get(0, 0, n - 1, ll, rr);
  }

  node get(int p) {
    assert(0 <= p && p <= n - 1);
    return get(0, 0, n - 1, p, p);
  }

  template <typename... M>
  void modify(int ll, int rr, const M&... v) {
    assert(0 <= ll && ll <= rr && rr <= n - 1);
    modify(0, 0, n - 1, ll, rr, v...);
  }

  // find_first and find_last call all FALSE elements
  // to the left (right) of the sought position exactly once

  int find_first(int ll, int rr, const function<bool(const node&)> &f) {
    assert(0 <= ll && ll <= rr && rr <= n - 1);
    return find_first(0, 0, n - 1, ll, rr, f);
  }

  int find_last(int ll, int rr, const function<bool(const node&)> &f) {
    assert(0 <= ll && ll <= rr && rr <= n - 1);
    return find_last(0, 0, n - 1, ll, rr, f);
  }
};
```

1.12 sparsetable.cpp

```
template <typename T, typename F>
class SparseTable {
 public:
```

```
  int n;
  vector<vector<T>> mat;
  F func;

  SparseTable(const vector<T>& a, const F& f) : func(f) {
    n = static_cast<int>(a.size());
    int max_log = 32 - __builtin_clz(n);
    mat.resize(max_log);
    mat[0] = a;
    for (int j = 1; j < max_log; j++) {
      mat[j].resize(n - (1 << j) + 1);
      for (int i = 0; i <= n - (1 << j); i++) {
        mat[j][i] = func(mat[j - 1][i], mat[j - 1][i + (1 << (j - 1))]);
      }
    }
  }

  T get(int from, int to) const {
    assert(0 <= from && from <= to && to <= n - 1);
    int lg = 32 - __builtin_clz(to - from + 1) - 1;
    return func(mat[lg][from], mat[lg][to - (1 << lg) + 1]);
  }
};
```

1.13 splay.cpp

```
namespace splay_tree {

bool is_bst_root(node* v) {
  if (v == nullptr) {
    return false;
  }
  return (v->p == nullptr || (v->p->l != v && v->p->r != v));
}

void rotate(node* v) {
  node* u = v->p;
  assert(u != nullptr);
  u->push();
  v->push();
  v->p = u->p;
  if (v->p != nullptr) {
    if (v->p->l == u) {
      v->p->l = v;
    }
    if (v->p->r == u) {
      v->p->r = v;
    }
  }
  if (v == u->l) {
    u->l = v->r;
    v->r = u;
  } else {
    u->r = v->l;
    v->l = u;
  }
  u->pull();
  v->pull();
}

void splay(node* v) {
  if (v == nullptr) {
    return;
  }
  while (!is_bst_root(v)) {
    node* u = v->p;
    if (!is_bst_root(u)) {
      if ((u->l == v) ^ (u->p->l == u)) {
        rotate(v);
      } else {
        rotate(u);
      }
    }
    rotate(v);
  }
}

pair<node*, int> find(node* v, const function<int(node*)> &go_to) {
  // go_to returns: 0 -- found; -1 -- go left; 1 -- go right
  // find returns the last vertex on the descent and its go_to
  if (v == nullptr) {
    return {nullptr, 0};
  }
  splay(v);
  int dir;
  while (true) {
```

```
    v->push();
    dir = go_to(v);
    if (dir == 0) {
      break;
    }
    node* u = (dir == -1 ? v->l : v->r);
    if (u == nullptr) {
      break;
    }
    v = u;
  }
  splay(v);
  return {v, dir};
}

node* get_leftmost(node* v) {
  return find(v, [&](node*) { return -1; }).first;
}

node* get_rightmost(node* v) {
  return find(v, [&](node*) { return 1; }).first;
}

node* get_kth(node* v, int k) { // 0-indexed
  pair<node*, int> p = find(v, [&](node* u) {
    if (u->l != nullptr) {
      if (u->l->sz > k) {
        return -1;
      }
      k -= u->l->sz;
    }
    if (k == 0) {
      return 0;
    }
    k--;
    return 1;
  });
  return (p.second == 0 ? p.first : nullptr);
}

int get_position(node* v) { // 0-indexed
  splay(v);
  return (v->l != nullptr ? v->l->sz : 0);
}

}

node* get_bst_root(node* v) {
  splay(v);
  return v;
}

pair<node*, node*> split(node* v, const function<bool(node*)> &is_right) {
  if (v == nullptr) {
    return {nullptr, nullptr};
  }
  pair<node*, int> p = find(v, [&](node* u) { return is_right(u) ? -1 : 1;
      });
  v = p.first;
  v->push();
  if (p.second == -1) {
    node* u = v->l;
    if (u == nullptr) {
      return {nullptr, v};
    }
    v->l = nullptr;
    u->p = v->p;
    u = get_rightmost(u);
    v->p = u;
    v->pull();
    return {u, v};
  } else {
    node* u = v->r;
    if (u == nullptr) {
      return {v, nullptr};
    }
    v->r = nullptr;
    v->pull();
    return {v, u};
  }
}

pair<node*, node*> split_leftmost_k(node* v, int k) {
  return split(v, [&](node* u) {
    int left_and_me = (u->l != nullptr ? u->l->sz : 0) + 1;
    if (k >= left_and_me) {
      k -= left_and_me;
      return false;
```

```
    }
    return true;
  });
}

node* merge(node* v, node* u) {
  if (v == nullptr) {
    return u;
  }
  if (u == nullptr) {
    return v;
  }
  v = get_rightmost(v);
  assert(v->r == nullptr);
  splay(u);
  v->push();
  v->r = u;
  v->pull();
  return v;
}

int count_left(node* v, const function<bool(node*)> &is_right) {
  if (v == nullptr) {
    return 0;
  }
  pair<node*, int> p = find(v, [&](node* u) { return is_right(u) ? -1 : 1;
      });
  node* u = p.first;
  return (u->l != nullptr ? u->l->sz : 0) + (p.second == 1);
}

node* add(node* r, node* v, const function<bool(node*)> &go_left) {
  pair<node*, node*> p = split(r, go_left);
  return merge(p.first, merge(v, p.second));
}

node* remove(node* v) { // returns the new root
  splay(v);
  v->push();
  node* x = v->l;
  node* y = v->r;
  v->l = v->r = nullptr;
  node* z = merge(x, y);
  if (z != nullptr) {
    z->p = v->p;
  }
  v->p = nullptr;
  v->push();
  v->pull(); // now v might be reusable...
  return z;
}

node* next(node* v) {
  splay(v);
  v->push();
  if (v->r == nullptr) {
    return nullptr;
  }
  v = v->r;
  while (v->l != nullptr) {
    v->push();
    v = v->l;
  }
  splay(v);
  return v;
}

node* prev(node* v) {
  splay(v);
  v->push();
  if (v->l == nullptr) {
    return nullptr;
  }
  v = v->l;
  while (v->r != nullptr) {
    v->push();
    v = v->r;
  }
  splay(v);
  return v;
}

int get_size(node* v) {
  splay(v);
  return (v != nullptr ? v->sz : 0);
}
```

```
template<typename... T>
void do_apply(node* v, T... args) {
  splay(v);
  v->unsafe_apply(args...);
}

void reverse(node* v) {
  splay(v);
  v->unsafe_reverse();
}

}  // namespace splay_tree

using namespace splay_tree;
```

1.14 treap.cpp

```
namespace treap {

pair<node*, int> find(node* v, const function<int(node*)> &go_to) {
  // go_to returns: 0 -- found; -1 -- go left; 1 -- go right
  // find returns the last vertex on the descent and its go_to
  if (v == nullptr) {
    return {nullptr, 0};
  }
  int dir;
  while (true) {
    v->push();
    dir = go_to(v);
    if (dir == 0) {
      break;
    }
    node* u = (dir == -1 ? v->l : v->r);
    if (u == nullptr) {
      break;
    }
    v = u;
  }
  return {v, dir};
}

node* get_leftmost(node* v) {
  return find(v, [&](node*) { return -1; }).first;
}

node* get_rightmost(node* v) {
  return find(v, [&](node*) { return 1; }).first;
}

node* get_kth(node* v, int k) { // 0-indexed
  pair<node*, int> p = find(v, [&](node* u) {
    if (u->l != nullptr) {
      if (u->l->sz > k) {
        return -1;
      }
      k -= u->l->sz;
    }
    if (k == 0) {
      return 0;
    }
    k--;
    return 1;
  });
  return (p.second == 0 ? p.first : nullptr);
}

int get_position(node* v) { // 0-indexed
  int k = (v->l != nullptr ? v->l->sz : 0);
  while (v->p != nullptr) {
    if (v == v->p->r) {
      k++;
      if (v->p->l != nullptr) {
        k += v->p->l->sz;
      }
    }
    v = v->p;
  }
  return k;
}

node* get_bst_root(node* v) {
  while (v->p != nullptr) {
    v = v->p;
  }
  return v;
```

```
}

pair<node*, node*> split(node* v, const function<bool(node*)> &is_right) {
  if (v == nullptr) {
    return {nullptr, nullptr};
  }
  v->push();
  if (is_right(v)) {
    pair<node*, node*> p = split(v->l, is_right);
    if (p.first != nullptr) {
      p.first->p = nullptr;
    }
    v->l = p.second;
    v->pull();
    return {p.first, v};
  } else {
    pair<node*, node*> p = split(v->r, is_right);
    v->r = p.first;
    if (p.second != nullptr) {
      p.second->p = nullptr;
    }
    v->pull();
    return {v, p.second};
  }
}

pair<node*, node*> split_leftmost_k(node* v, int k) {
  return split(v, [&](node* u) {
    int left_and_me = (u->l != nullptr ? u->l->sz : 0) + 1;
    if (k >= left_and_me) {
      k -= left_and_me;
      return false;
    }
    return true;
  });
}

node* merge(node* v, node* u) {
  if (v == nullptr) {
    return u;
  }
  if (u == nullptr) {
    return v;
  }
  if (v->P > u->P) {
//    if (rng() % (v->sz + u->sz) < (unsigned int) v->sz) {
    v->push();
    v->r = merge(v->r, u);
    v->pull();
    return v;
  } else {
    u->push();
    u->l = merge(v, u->l);
    u->pull();
    return u;
  }
}

int count_left(node* v, const function<bool(node*)> &is_right) {
  if (v == nullptr) {
    return 0;
  }
  v->push();
  if (is_right(v)) {
    return count_left(v->l, is_right);
  }
  return (v->l != nullptr ? v->l->sz : 0) + 1 + count_left(v->r, is_right);
}

node* add(node* r, node* v, const function<bool(node*)> &go_left) {
  pair<node*, node*> p = split(r, go_left);
  return merge(p.first, merge(v, p.second));
}

node* remove(node* v) { // returns the new root
  v->push();
  node* x = v->l;
  node* y = v->r;
  node* p = v->p;
  v->l = v->r = v->p = nullptr;
  v->push();
  v->pull(); // now v might be reusable...
  node* z = merge(x, y);
  if (p == nullptr) {
    if (z != nullptr) {
      z->p = nullptr;
```

```
    }
    return z;
  }
  if (p->l == v) {
    p->l = z;
  }
  if (p->r == v) {
    p->r = z;
  }
  while (true) {
    p->push();
    p->pull();
    if (p->p == nullptr) {
      break;
    }
    p = p->p;
  }
  return p;
}

node* next(node* v) {
  if (v->r == nullptr) {
    while (v->p != nullptr && v->p->r == v) {
      v = v->p;
    }
    return v->p;
  }
  v->push();
  v = v->r;
  while (v->l != nullptr) {
    v->push();
    v = v->l;
  }
  return v;
}

node* prev(node* v) {
  if (v->l == nullptr) {
    while (v->p != nullptr && v->p->l == v) {
      v = v->p;
    }
    return v->p;
  }
  v->push();
  v = v->l;
  while (v->r != nullptr) {
    v->push();
    v = v->r;
  }
  return v;
}

int get_size(node* v) {
  return (v != nullptr ? v->sz : 0);
}

template<typename... T>
void apply(node* v, T... args) {
  v->unsafe_apply(args...);
}

void reverse(node* v) {
  v->unsafe_reverse();
}

}  // namespace treap

using namespace treap;
```

2 flows

2.1 blossom.cpp

```
template <typename T>
vector<int> find_max_unweighted_matching(const undigraph<T>& g) {
  vector<int> mate(g.n, -1);
  vector<int> label(g.n);
  vector<int> parent(g.n);
  vector<int> orig(g.n);
  queue<int> q;
  vector<int> aux(g.n, -1);
  int aux_time = -1;
  auto lca = [&](int x, int y) {
    aux_time++;
    while (true) {
```

```
      if (x != -1) {
        if (aux[x] == aux_time) {
          return x;
        }
        aux[x] = aux_time;
        if (mate[x] == -1) {
          x = -1;
        } else {
          x = orig[parent[mate[x]]];
        }
      }
      swap(x, y);
    }
  };
  auto blossom = [&](int v, int w, int a) {
    while (orig[v] != a) {
      parent[v] = w;
      w = mate[v];
      if (label[w] == 1) {
        label[w] = 0;
        q.push(w);
      }
      orig[v] = orig[w] = a;
      v = parent[w];
    }
  };
  auto augment = [&](int v) {
    while (v != -1) {
      int pv = parent[v];
      int nv = mate[pv];
      mate[v] = pv;
      mate[pv] = v;
      v = nv;
    }
  };
  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    while (!q.empty()) {
      q.pop();
    }
    q.push(root);
    label[root] = 0;
    while (!q.empty()) {
      int v = q.front();
      q.pop();
      for (int id : g.g[v]) {
        auto &e = g.edges[id];
        int x = e.from ^ e.to ^ v;
        if (label[x] == -1) {
          label[x] = 1;
          parent[x] = v;
          if (mate[x] == -1) {
            augment(x);
            return true;
          }
          label[mate[x]] = 0;
          q.push(mate[x]);
          continue;
        }
        if (label[x] == 0 && orig[v] != orig[x]) {
          int a = lca(orig[v], orig[x]);
          blossom(x, v, a);
          blossom(v, x, a);
        }
      }
    }
    return false;
  };
  auto greedy = [&]() {
    vector<int> order(g.n);
    iota(order.begin(), order.end(), 0);
    shuffle(order.begin(), order.end(), mt19937(787788));
    for (int i : order) {
      if (mate[i] == -1) {
        for (int id : g.g[i]) {
          auto &e = g.edges[id];
          int to = e.from ^ e.to ^ i;
          if (i != to && mate[to] == -1) {
            mate[i] = to;
            mate[to] = i;
            break;
          }
        }
      }
    }
```

```
  };
  greedy();
  for (int i = 0; i < g.n; i++) {
    if (mate[i] == -1) {
      bfs(i);
    }
  }
  return mate;
}
```

2.2 dinic-edge-ids.cpp

```
template <typename T>
class flow_graph {
 public:
  static constexpr T eps = (T) 1e-9;

  struct edge {
    int from;
    int to;
    T c;
    T f;
  };

  vector<vector<int>> g;
  vector<edge> edges;
  int n;
  int st, fin;
  T flow;

  vector<int> ptr;
  vector<int> d;
  vector<int> q;

  flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
    assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
    g.resize(n);
    ptr.resize(n);
    d.resize(n);
    q.resize(n);
    flow = 0;
  }

  void clear_flow() {
    for (const edge &e : edges) {
      e.f = 0;
    }
    flow = 0;
  }

  void add(int from, int to, T forward_cap, T backward_cap) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    g[from].push_back((int) edges.size());
    edges.push_back({from, to, forward_cap, 0});
    g[to].push_back((int) edges.size());
    edges.push_back({to, from, backward_cap, 0});
  }

  bool expath() {
    fill(d.begin(), d.end(), -1);
    q[0] = fin;
    d[fin] = 0;
    int beg = 0, end = 1;
    while (beg < end) {
      int i = q[beg++];
      for (int id : g[i]) {
        const edge &e = edges[id];
        const edge &back = edges[id ^ 1];
        if (back.c - back.f > eps && d[e.to] == -1) {
          d[e.to] = d[i] + 1;
          if (e.to == st) {
            return true;
          }
          q[end++] = e.to;
        }
      }
    }
    return false;
  }

  T dfs(int v, T w) {
    if (v == fin) {
      return w;
    }
    int &j = ptr[v];
    while (j >= 0) {
```

```
      int id = g[v][j];
      const edge &e = edges[id];
      if (e.c - e.f > eps && d[e.to] == d[v] - 1) {
        T t = dfs(e.to, min(e.c - e.f, w));
        if (t > eps) {
          edges[id].f += t;
          edges[id ^ 1].f -= t;
          return t;
        }
      }
      j--;
    }
    return 0;
  }

  T max_flow() {
    while (expath()) {
      for (int i = 0; i < n; i++) {
        ptr[i] = (int) g[i].size() - 1;
      }
      T big_add = 0;
      while (true) {
        T add = dfs(st, numeric_limits<T>::max());
        if (add <= eps) {
          break;
        }
        big_add += add;
      }
      if (big_add <= eps) {
        break;
      }
      flow += big_add;
    }
    return flow;
  }

  vector<bool> min_cut() {
    max_flow();
    vector<bool> ret(n);
    for (int i = 0; i < n; i++) {
      ret[i] = (d[i] != -1);
    }
    return ret;
  }

  }

  // Maximum flow / minimum cut, Dinic's algorithm
  // Usage:
  // 1) flow_graph<T> g(n, start, finish); [T == int / long long / double]
  // 2) g.add(from, to, forward_cap, backward_cap);
  // 3) cout << g.max_flow() << endl;
  // 4) vector<bool> cut = g.min_cut();
  //    for (auto &e : g.edges)
  //      if (cut[e.from] != cut[e.to]) ; // edge e = (e.from -> e.to) is
      cut
};
```

2.3 dinic-old.cpp

```
template <typename T>
class flow_graph {
 public:
  static constexpr T eps = (T) 1e-9;

  struct edge {
    int to;
    T c;
    T f;
    int rev;
  };

  vector<vector<edge>> g;
  vector<int> ptr;
  vector<int> d;
  vector<int> q;
  int n;
  int st, fin;
  T flow;

  flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
    assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
    g.resize(n);
    ptr.resize(n);
    d.resize(n);
    q.resize(n);
    flow = 0;
  }
```

```
29
30    void clear_flow() {
31      for (int i = 0; i < n; i++) {
32        for (edge &e : g[i]) {
33          e.f = 0;
34        }
35      }
36      flow = 0;
37    }
38
39    void add(int from, int to, T forward_cap, T backward_cap) {
40      assert(0 <= from && from < n && 0 <= to && to < n);
41      int from_size = g[from].size();
42      int to_size = g[to].size();
43      g[from].push_back({to, forward_cap, 0, to_size});
44      g[to].push_back({from, backward_cap, 0, from_size});
45    }
46
47    bool expath() {
48      fill(d.begin(), d.end(), -1);
49      q[0] = fin;
50      d[fin] = 0;
51      int beg = 0, end = 1;
52      while (beg < end) {
53        int i = q[beg++];
54        for (const edge &e : g[i]) {
55          const edge &back = g[e.to][e.rev];
56          if (back.c - back.f > eps && d[e.to] == -1) {
57            d[e.to] = d[i] + 1;
58            if (e.to == st) {
59              return true;
60            }
61            q[end++] = e.to;
62          }
63        }
64      }
65      return false;
66    }
67
68    T dfs(int v, T w) {
69      if (v == fin) {
70        return w;
71      }
72      int &j = ptr[v];
73      while (j >= 0) {
74        const edge &e = g[v][j];
75        if (e.c - e.f > eps && d[e.to] == d[v] - 1) {
76          T t = dfs(e.to, min(e.c - e.f, w));
77          if (t > eps) {
78            g[v][j].f += t;
79            g[e.to][e.rev].f -= t;
80            return t;
81          }
82        }
83        j--;
84      }
85      return 0;
86    }
87
88    T max_flow() {
89      while (expath()) {
90        for (int i = 0; i < n; i++) {
91          ptr[i] = (int) g[i].size() - 1;
92        }
93        T big_add = 0;
94        while (true) {
95          T add = dfs(st, numeric_limits<T>::max());
96          if (add <= eps) {
97            break;
98          }
99          big_add += add;
100       }
101       if (big_add <= eps) {
102         break;
103       }
104       flow += big_add;
105     }
106     return flow;
107   }
108
109   vector<bool> min_cut() {
110     max_flow();
111     vector<bool> ret(n);
112     for (int i = 0; i < n; i++) {
113       ret[i] = (d[i] != -1);
114     }
```

```
    return ret;
  }
};
```

2.4 dinic.cpp

```
template <typename T>
class dinic {
 public:
  flow_graph<T> &g;

  vector<int> ptr;
  vector<int> d;
  vector<int> q;

  dinic(flow_graph<T> &_g) : g(_g) {
    ptr.resize(g.n);
    d.resize(g.n);
    q.resize(g.n);
  }

  bool expath() {
    fill(d.begin(), d.end(), -1);
    q[0] = g.fin;
    d[g.fin] = 0;
    int beg = 0, end = 1;
    while (beg < end) {
      int i = q[beg++];
      for (int id : g.g[i]) {
        const auto &e = g.edges[id];
        const auto &back = g.edges[id ^ 1];
        if (back.c - back.f > g.eps && d[e.to] == -1) {
          d[e.to] = d[i] + 1;
          if (e.to == g.st) {
            return true;
          }
          q[end++] = e.to;
        }
      }
    }
    return false;
  }

  T dfs(int v, T w) {
    if (v == g.fin) {
      return w;
    }
    int &j = ptr[v];
    while (j >= 0) {
      int id = g.g[v][j];
      const auto &e = g.edges[id];
      if (e.c - e.f > g.eps && d[e.to] == d[v] - 1) {
        T t = dfs(e.to, min(e.c - e.f, w));
        if (t > g.eps) {
          g.edges[id].f += t;
          g.edges[id ^ 1].f -= t;
          return t;
        }
      }
      j--;
    }
    return 0;
  }

  T max_flow() {
    while (expath()) {
      for (int i = 0; i < g.n; i++) {
        ptr[i] = (int) g.g[i].size() - 1;
      }
      T big_add = 0;
      while (true) {
        T add = dfs(g.st, numeric_limits<T>::max());
        if (add <= g.eps) {
          break;
        }
        big_add += add;
      }
      if (big_add <= g.eps) {
        break;
      }
      g.flow += big_add;
    }
    return g.flow;
  }

  vector<bool> min_cut() {
```

```
    max_flow();
    vector<bool> ret(g.n);
    for (int i = 0; i < g.n; i++) {
      ret[i] = (d[i] != -1);
    }
    return ret;
  }
};
```

2.5 fastflow-other.cpp

```
// https://pastebin.com/exQM152L

// Doesn't walk through the whole path during augment at the cost of bigger
    constant
// Not recommended to use with double

template <typename T>
class flow_graph {
 public:
  static constexpr T eps = (T) 1e-9;

  struct edge {
    int to;
    T c;
    T f;
    int rev;
  };

  vector<vector<edge>> g;
  vector<int> ptr;
  vector<int> d;
  vector<int> q;
  vector<int> cnt_on_layer;
  vector<int> prev_edge;
  vector<T> to_push;
  vector<T> pushed;
  vector<int> smallest;
  bool can_reach_sink;

  int n;
  int st, fin;
  T flow;

  flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
    assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
    g.resize(n);
    ptr.resize(n);
    d.resize(n);
    q.resize(n);
    cnt_on_layer.resize(n + 1);
    prev_edge.resize(n);
    to_push.resize(n);
    pushed.resize(n);
    smallest.resize(n);
    flow = 0;
  }

  void clear_flow() {
    for (int i = 0; i < n; i++) {
      for (edge &e : g[i]) {
        e.f = 0;
      }
    }
    flow = 0;
  }

  void add(int from, int to, T forward_cap, T backward_cap) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int from_size = g[from].size();
    int to_size = g[to].size();
    g[from].push_back({to, forward_cap, 0, to_size});
    g[to].push_back({from, backward_cap, 0, from_size});
  }

  bool expath() {
    fill(d.begin(), d.end(), n);
    q[0] = fin;
    d[fin] = 0;
    fill(cnt_on_layer.begin(), cnt_on_layer.end(), 0);
    cnt_on_layer[n] = n - 1;
    cnt_on_layer[0] = 1;
    int beg = 0, end = 1;
    while (beg < end) {
      int i = q[beg++];
      for (const edge &e : g[i]) {
```

```
        const edge &back = g[e.to][e.rev];
        if (back.c - back.f > eps && d[e.to] == n) {
          cnt_on_layer[d[e.to]]--;
          d[e.to] = d[i] + 1;
          cnt_on_layer[d[e.to]]++;
          q[end++] = e.to;
        }
      }
    }
    return (d[st] != n);
  }

  void rollback(int &v) {
    edge &e = g[v][prev_edge[v]];
    if (pushed[v]) {
      edge &back = g[e.to][e.rev];
      back.f += pushed[v];
      e.f -= pushed[v];
      pushed[e.to] += pushed[v];
      to_push[e.to] -= pushed[v];
      pushed[v] = 0;
    }
    v = e.to;
  }

  void augment(int &v) {
    pushed[v] += to_push[v];
    to_push[v] = 0;
    int new_v = smallest[v];
    while (v != new_v) {
      rollback(v);
    }
  }

  void retreat(int &v) {
    int new_dist = n - 1;
    for (const edge &e : g[v]) {
      if (e.c - e.f > eps && d[e.to] < new_dist) {
        new_dist = d[e.to];
      }
    }
    cnt_on_layer[d[v]]--;
    if (cnt_on_layer[d[v]] == 0) {
      if (new_dist + 1 > d[v]) {
        can_reach_sink = false;
      }
    }
    d[v] = new_dist + 1;
    cnt_on_layer[d[v]]++;
    if (v != st) {
      rollback(v);
    }
  }

  T max_flow() {
    can_reach_sink = true;
    for (int i = 0; i < n; i++) {
      ptr[i] = (int) g[i].size() - 1;
    }
    if (expath()) {
      int v = st;
      to_push[v] = numeric_limits<T>::max();
      smallest[v] = v;
      while (d[st] < n) {
        while (ptr[v] >= 0) {
          const edge &e = g[v][ptr[v]];
          if (e.c - e.f > eps && d[e.to] == d[v] - 1) {
            prev_edge[e.to] = e.rev;
            to_push[e.to] = to_push[v];
            smallest[e.to] = smallest[v];
            if (e.c - e.f < to_push[e.to]) {
              to_push[e.to] = e.c - e.f;
              smallest[e.to] = v;
            }
            v = e.to;
            if (v == fin) {
              augment(v);
            }
            break;
          }
          ptr[v]--;
        }
        if (ptr[v] < 0) {
          ptr[v] = (int) g[v].size() - 1;
          retreat(v);
          if (!can_reach_sink) {
```

```
            break;
          }
        }
      }
      while (v != st) {
        rollback(v);
      }
      flow += pushed[st];
      pushed[st] = 0;
    }
    return flow;
  }

  vector<bool> min_cut() {
    max_flow();
    assert(!expath());
    vector<bool> ret(n);
    for (int i = 0; i < n; i++) {
      ret[i] = (d[i] != n);
    }
    return ret;
  }
};
```

2.6 fastflow.cpp

```
// https://pastebin.com/exQM152L

template <typename T>
class flow_graph {
 public:
  static constexpr T eps = (T) 1e-9;

  struct edge {
    int to;
    T c;
    T f;
    int rev;
  };

  vector<vector<edge>> g;
  vector<int> ptr;
  vector<int> d;
  vector<int> q;
  vector<int> cnt_on_layer;
  vector<int> prev_edge;
  bool can_reach_sink;

  int n;
  int st, fin;
  T flow;

  flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
    assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
    g.resize(n);
    ptr.resize(n);
    d.resize(n);
    q.resize(n);
    cnt_on_layer.resize(n + 1);
    prev_edge.resize(n);
    flow = 0;
  }

  void clear_flow() {
    for (int i = 0; i < n; i++) {
      for (edge &e : g[i]) {
        e.f = 0;
      }
    }
    flow = 0;
  }

  void add(int from, int to, T forward_cap, T backward_cap) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int from_size = g[from].size();
    int to_size = g[to].size();
    g[from].push_back({to, forward_cap, 0, to_size});
    g[to].push_back({from, backward_cap, 0, from_size});
  }

  bool expath() {
    fill(d.begin(), d.end(), n);
    q[0] = fin;
    d[fin] = 0;
    fill(cnt_on_layer.begin(), cnt_on_layer.end(), 0);
    cnt_on_layer[n] = n - 1;
```

```
    cnt_on_layer[0] = 1;
    int beg = 0, end = 1;
    while (beg < end) {
      int i = q[beg++];
      for (const edge &e : g[i]) {
        const edge &back = g[e.to][e.rev];
        if (back.c - back.f > eps && d[e.to] == n) {
          cnt_on_layer[d[e.to]]--;
          d[e.to] = d[i] + 1;
          cnt_on_layer[d[e.to]]++;
          q[end++] = e.to;
        }
      }
    }
    return (d[st] != n);
  }

  T augment(int &v) {
    T cur = numeric_limits<T>::max();
    int i = fin;
    while (i != st) {
      const edge &e = g[i][prev_edge[i]];
      const edge &back = g[e.to][e.rev];
      cur = min(cur, back.c - back.f);
      i = e.to;
    }
    i = fin;
    while (i != st) {
      edge &e = g[i][prev_edge[i]];
      edge &back = g[e.to][e.rev];
      back.f += cur;
      e.f -= cur;
      i = e.to;
      if (back.c - back.f <= eps) {
        v = i;
      }
    }
    return cur;
  }

  int retreat(int v) {
    int new_dist = n - 1;
    for (const edge &e : g[v]) {
      if (e.c - e.f > eps && d[e.to] < new_dist) {
        new_dist = d[e.to];
      }
    }
    cnt_on_layer[d[v]]--;
    if (cnt_on_layer[d[v]] == 0) {
      if (new_dist + 1 > d[v]) {
        can_reach_sink = false;
      }
    }
    d[v] = new_dist + 1;
    cnt_on_layer[d[v]]++;
    if (v != st) {
      v = g[v][prev_edge[v]].to;
    }
    return v;
  }

  T max_flow() {
    can_reach_sink = true;
    for (int i = 0; i < n; i++) {
      ptr[i] = (int) g[i].size() - 1;
    }
    if (expath()) {
      int v = st;
      while (d[st] < n) {
        while (ptr[v] >= 0) {
          const edge &e = g[v][ptr[v]];
          if (e.c - e.f > eps && d[e.to] == d[v] - 1) {
            prev_edge[e.to] = e.rev;
            v = e.to;
            if (v == fin) {
              flow += augment(v);
            }
            break;
          }
          ptr[v]--;
        }
        if (ptr[v] < 0) {
          ptr[v] = (int) g[v].size() - 1;
          v = retreat(v);
          if (!can_reach_sink) {
            break;
```

```
          }
        }
      }
    }
    return flow;
  }

  vector<bool> min_cut() {
    max_flow();
    assert(!expath());
    vector<bool> ret(n);
    for (int i = 0; i < n; i++) {
      ret[i] = (d[i] != n);
    }
    return ret;
  }
};
```

2.7 flow-decomposition.cpp

```
template <typename T>
class flow_decomposition {
 public:
  const flow_graph<T> &g;

  vector<vector<int>> paths;
  vector<T> path_flows;
  vector<vector<int>> cycles;
  vector<T> cycle_flows;

  flow_decomposition(const flow_graph<T> &_g) : g(_g) {
  }

  void decompose() {
    vector<T> fs(g.edges.size());
    for (int i = 0; i < (int) g.edges.size(); i++) {
      fs[i] = g.edges[i].f;
    }
    paths.clear();
    path_flows.clear();
    cycles.clear();
    cycle_flows.clear();
    vector<int> ptr(g.n);
    for (int i = 0; i < g.n; i++) {
      ptr[i] = (int) g.g[i].size() - 1;
    }
    vector<int> was(g.n, -1);
    int start = g.st;
    for (int iter = 0; ; iter++) {
      bool found_start = false;
      while (true) {
        if (ptr[start] >= 0) {
          int id = g.g[start][ptr[start]];
          if (fs[id] > g.eps) {
            found_start = true;
            break;
          }
          ptr[start]--;
          continue;
        }
        start = (start + 1) % g.n;
        if (start == g.st) {
          break;
        }
      }
      if (!found_start) {
        break;
      }
      vector<int> path;
      bool is_cycle = false;
      int v = start;
      while (true) {
        if (v == g.fin) {
          break;
        }
        if (was[v] == iter) {
          bool found = false;
          for (int i = 0; i < (int) path.size(); i++) {
            int id = path[i];
            auto &e = g.edges[id];
            if (e.from == v) {
              path.erase(path.begin(), path.begin() + i);
              found = true;
              break;
            }
          }
```

```
          assert(found);
          is_cycle = true;
          break;
        }
        was[v] = iter;
        bool found = false;
        while (ptr[v] >= 0) {
          int id = g.g[v][ptr[v]];
          if (fs[id] > g.eps) {
            path.push_back(id);
            v = g.edges[id].to;
            found = true;
            break;
          }
          ptr[v]--;
        }
        assert(found);
      }
      T path_flow = numeric_limits<T>::max();
      for (int id : path) {
        path_flow = min(path_flow, fs[id]);
      }
      for (int id : path) {
        fs[id] -= path_flow;
        fs[id ^ 1] += path_flow;
      }
      if (is_cycle) {
        cycles.push_back(path);
        cycle_flows.push_back(path_flow);
      } else {
        paths.push_back(path);
        path_flows.push_back(path_flow);
      }
    }
    for (const T& f : fs) {
      assert(-g.eps <= f && f <= g.eps);
    }
  }
};
```

2.8 flow-graph.cpp

```
template <typename T>
class flow_graph {
 public:
  static constexpr T eps = (T) 1e-9;

  struct edge {
    int from;
    int to;
    T c;
    T f;
  };

  vector<vector<int>> g;
  vector<edge> edges;
  int n;
  int st;
  int fin;
  T flow;

  flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
    assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
    g.resize(n);
    flow = 0;
  }

  void clear_flow() {
    for (const edge &e : edges) {
      e.f = 0;
    }
    flow = 0;
  }

  int add(int from, int to, T forward_cap, T backward_cap) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = (int) edges.size();
    g[from].push_back(id);
    edges.push_back({from, to, forward_cap, 0});
    g[to].push_back(id + 1);
    edges.push_back({to, from, backward_cap, 0});
    return id;
  }
};
```

2.9 gomory-hu-old.cpp

```cpp
template <typename T>
forest<T> gomory_hu(const undigraph<T> &g) {
  int n = g.n;
  if (n == 1) {
    return forest<T>(n);
  }
  flow_graph<T> fg(n, 0, 1);
  for (auto &e : g.edges) {
    fg.add(e.from, e.to, e.cost, e.cost);
  }
  vector<vector<int>> dist(n, vector<int>(n, numeric_limits<T>::max()));
  function<void(vector<int>)> dfs = [&g, &n, &fg, &dist, &dfs](vector<int>
      group) {
    int sz = group.size();
    if (sz == 1) {
      return;
    }
    fg.clear_flow();
    fg.st = group[0];
    fg.fin = group[1];
    T flow = fg.max_flow();
    vector<bool> cut = fg.min_cut();
    for (int i = 0; i < n; i++) {
      for (int j = i + 1; j < n; j++) {
        if (cut[i] != cut[j]) {
          dist[i][j] = min(dist[i][j], flow);
        }
      }
    }
    vector<int> new_groups[2];
    for (int v : group) {
      new_groups[(int) cut[v]].push_back(v);
    }
    for (int id = 0; id < 2; id++) {
      dfs(new_groups[id]);
    }
  };
  vector<int> group(n);
  iota(group.begin(), group.end(), 0);
  dfs(group);
  undigraph<T> mg(n);
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      mg.add(i, j, -dist[i][j]);
    }
  }
  T foo;
  vector<int> ids = mst(mg, foo);
  forest<T> ret(n);
  for (int id : ids) {
    auto &e = mg.edges[id];
    ret.add(e.from, e.to, -e.cost);
  }
  return ret;
  // don't be lazy next time!
  // implement a proper gomory-hu tree
}
```

2.10 gomory-hu.cpp

```cpp
template <typename T>
forest<T> gomory_hu(const undigraph<T>& g) {
  int n = g.n;
  flow_graph<T> fg(n, 0, 1);
  for (auto& e : g.edges) {
    fg.add(e.from, e.to, e.cost, e.cost);
  }
  forest<T> ret(n);
  vector<int> pr(n, 0);
  for (int i = 1; i < n; i++) {
    fg.clear_flow();
    fg.st = i;
    fg.fin = pr[i];
    T flow = fg.max_flow();
    vector<bool> cut = fg.min_cut();
    for (int j = i + 1; j < n; j++) {
      if (cut[j] == cut[i] && pr[j] == pr[i]) {
        pr[j] = i;
      }
    }
    ret.add(i, pr[i], flow);
  }
  return ret;
  // can be optimized by compressing components
}
```

2.11 hungarian-arrays.cpp

```
template <typename T>
class hungarian {
 public:
  static const int MAX_N = ... + 1;

  int n;
  int m;
  T a[MAX_N][MAX_N];
  T u[MAX_N];
  T v[MAX_N];
  int pa[MAX_N];
  int pb[MAX_N];
  int way[MAX_N];
  T minv[MAX_N];
  bool used[MAX_N];
  T inf;

  hungarian(int _n, int _m) : n(_n), m(_m) {
    assert(n <= m);
    T zero = T{};
    fill(u, u + n + 1, zero);
    fill(v, v + m + 1, zero);
    fill(pa, pa + n + 1, -1);
    fill(pb, pb + m + 1, -1);
    inf = numeric_limits<T>::max();
  }

  inline void add_row(int i) {
    fill(minv, minv + m + 1, inf);
    fill(used, used + m + 1, false);
    pb[m] = i;
    pa[i] = m;
    int j0 = m;
    do {
      used[j0] = true;
      int i0 = pb[j0];
      T delta = inf;
      int j1 = -1;
      for (int j = 0; j < m; j++) {
        if (!used[j]) {
          T cur = a[i0][j] - u[i0] - v[j];
          if (cur < minv[j]) {
            minv[j] = cur;
            way[j] = j0;
          }
          if (minv[j] < delta) {
            delta = minv[j];
            j1 = j;
          }
        }
      }
      for (int j = 0; j <= m; j++) {
        if (used[j]) {
          u[pb[j]] += delta;
          v[j] -= delta;
        } else {
          minv[j] -= delta;
        }
      }
      j0 = j1;
    } while (pb[j0] != -1);
    do {
      int j1 = way[j0];
      pb[j0] = pb[j1];
      pa[pb[j0]] = j0;
      j0 = j1;
    } while (j0 != m);
  }

  inline T current_score() {
    return -v[m];
  }

  inline T solve() {
    for (int i = 0; i < n; i++) {
      add_row(i);
    }
    return current_score();
  }
};
```

2.12 hungarian.cpp

```
template <typename T>
class hungarian {
```

```
 public:
  int n;
  int m;
  vector<vector<T>> a;
  vector<T> u;
  vector<T> v;
  vector<int> pa;
  vector<int> pb;
  vector<int> way;
  vector<T> minv;
  vector<bool> used;
  T inf;

  hungarian(int _n, int _m) : n(_n), m(_m) {
    assert(n <= m);
    a = vector<vector<T>>(n, vector<T>(m));
    u = vector<T>(n + 1);
    v = vector<T>(m + 1);
    pa = vector<int>(n + 1, -1);
    pb = vector<int>(m + 1, -1);
    way = vector<int>(m, -1);
    minv = vector<T>(m);
    used = vector<bool>(m + 1);
    inf = numeric_limits<T>::max();
  }

  inline void add_row(int i) {
    fill(minv.begin(), minv.end(), inf);
    fill(used.begin(), used.end(), false);
    pb[m] = i;
    pa[i] = m;
    int j0 = m;
    do {
      used[j0] = true;
      int i0 = pb[j0];
      T delta = inf;
      int j1 = -1;
      for (int j = 0; j < m; j++) {
        if (!used[j]) {
          T cur = a[i0][j] - u[i0] - v[j];
          if (cur < minv[j]) {
            minv[j] = cur;
            way[j] = j0;
          }
          if (minv[j] < delta) {
            delta = minv[j];
            j1 = j;
          }
        }
      }
      for (int j = 0; j <= m; j++) {
        if (used[j]) {
          u[pb[j]] += delta;
          v[j] -= delta;
        } else {
          minv[j] -= delta;
        }
      }
      j0 = j1;
    } while (pb[j0] != -1);
    do {
      int j1 = way[j0];
      pb[j0] = pb[j1];
      pa[pb[j0]] = j0;
      j0 = j1;
    } while (j0 != m);
  }

  inline T current_score() {
    return -v[m];
  }

  inline T solve() {
    for (int i = 0; i < n; i++) {
      add_row(i);
    }
    return current_score();
  }
};
```

2.13 matching.cpp

```
class matching {
 public:
  vector<vector<int>> g;
  vector<int> pa;
```

```
  vector<int> pb;
  vector<int> was;
  int n, m;
  int res;
  int iter;

  matching(int _n, int _m) : n(_n), m(_m) {
    assert(0 <= n && 0 <= m);
    pa = vector<int>(n, -1);
    pb = vector<int>(m, -1);
    was = vector<int>(n, 0);
    g.resize(n);
    res = 0;
    iter = 0;
  }

  void add(int from, int to) {
    assert(0 <= from && from < n && 0 <= to && to < m);
    g[from].push_back(to);
  }

  bool dfs(int v) {
    was[v] = iter;
    for (int u : g[v]) {
      if (pb[u] == -1) {
        pa[v] = u;
        pb[u] = v;
        return true;
      }
    }
    for (int u : g[v]) {
      if (was[pb[u]] != iter && dfs(pb[u])) {
        pa[v] = u;
        pb[u] = v;
        return true;
      }
    }
    return false;
  }

  int solve() {
    while (true) {
      iter++;
      int add = 0;
      for (int i = 0; i < n; i++) {
        if (pa[i] == -1 && dfs(i)) {
          add++;
        }
      }
      if (add == 0) {
        break;
      }
      res += add;
    }
    return res;
  }

  int run_one(int v) {
    if (pa[v] != -1) {
      return 0;
    }
    iter++;
    return (int) dfs(v);
  }
};
```

2.14 mcmf-slow.cpp

```
template <typename T, typename C>
class mcmf {
 public:
  static constexpr T eps = (T) 1e-9;

  struct edge {
    int from;
    int to;
    T c;
    T f;
    C cost;
  };

  vector<vector<int>> g;
  vector<edge> edges;
  vector<C> d;
  vector<int> q;
  vector<bool> in_queue;
```

```
  vector<int> pe;
  int n;
  int st, fin;
  T flow;
  C cost;

  mcmf(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
    assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
    g.resize(n);
    d.resize(n);
    in_queue.resize(n);
    pe.resize(n);
    flow = 0;
    cost = 0;
  }

  void clear_flow() {
    for (const edge &e : edges) {
      e.f = 0;
    }
    flow = 0;
  }

  void add(int from, int to, T forward_cap, T backward_cap, C cost) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    g[from].push_back((int) edges.size());
    edges.push_back({from, to, forward_cap, 0, cost});
    g[to].push_back((int) edges.size());
    edges.push_back({to, from, backward_cap, 0, -cost});
  }

  bool expath() {
    fill(d.begin(), d.end(), numeric_limits<C>::max());
    q.clear();
    q.push_back(st);
    d[st] = 0;
    in_queue[st] = true;
    int beg = 0;
    bool found = false;
    while (beg < (int) q.size()) {
      int i = q[beg++];
      if (i == fin) {
        found = true;
      }
      in_queue[i] = false;
      for (int id : g[i]) {
        const edge &e = edges[id];
        if (e.c - e.f > eps && d[i] + e.cost < d[e.to]) {
          d[e.to] = d[i] + e.cost;
          pe[e.to] = id;
          if (!in_queue[e.to]) {
            q.push_back(e.to);
            in_queue[e.to] = true;
          }
        }
      }
    }
    if (found) {
      T push = numeric_limits<T>::max();
      int v = fin;
      while (v != st) {
        const edge &e = edges[pe[v]];
        push = min(push, e.c - e.f);
        v = e.from;
      }
      v = fin;
      while (v != st) {
        edge &e = edges[pe[v]];
        e.f += push;
        edge &back = edges[pe[v] ^ 1];
        back.f -= push;
        v = e.from;
      }
      flow += push;
      cost += push * d[fin];
    }
    return found;
  }

  pair<T, C> max_flow_min_cost() {
    while (expath()) {}
    return {flow, cost};
  }
};
```

2.15 mcmf.cpp

```cpp
#include <bits/extc++.h>

template <typename T, typename C>
class MCMF {
 public:
  static constexpr T eps = (T) 1e-9;

  struct edge {
    int from;
    int to;
    T c;
    T f;
    C cost;
  };

  int n;
  vector<vector<int>> g;
  vector<edge> edges;
  vector<C> d;
  vector<C> pot;
  __gnu_pbds::priority_queue<pair<C, int>> q;
  vector<typename decltype(q)::point_iterator> its;
  vector<int> pe;
  const C INF_C = numeric_limits<C>::max() / 2;

  explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0), its(n), pe(n) {}

  int add(int from, int to, T forward_cap, T backward_cap, C edge_cost) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    assert(forward_cap >= 0 && backward_cap >= 0);
    int id = static_cast<int>(edges.size());
    g[from].push_back(id);
    edges.push_back({from, to, forward_cap, 0, edge_cost});
    g[to].push_back(id + 1);
    edges.push_back({to, from, backward_cap, 0, -edge_cost});
    return id;
  }

  void expath(int st) {
    fill(d.begin(), d.end(), INF_C);
    q.clear();
    fill(its.begin(), its.end(), q.end());
    its[st] = q.push({pot[st], st});
    d[st] = 0;
    while (!q.empty()) {
      int i = q.top().second;
      q.pop();
      its[i] = q.end();
      for (int id : g[i]) {
        const edge &e = edges[id];
        int j = e.to;
        if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
          d[j] = d[i] + e.cost;
          pe[j] = id;
          if (its[j] == q.end()) {
            its[j] = q.push({pot[j] - d[j], j});
          } else {
            q.modify(its[j], {pot[j] - d[j], j});
          }
        }
      }
    }
    swap(d, pot);
  }

  pair<T, C> max_flow_min_cost(int st, int fin) {
    T flow = 0;
    C cost = 0;
    bool ok = true;
    for (auto& e : edges) {
      if (e.c - e.f > eps && e.cost + pot[e.from] - pot[e.to] < 0) {
        ok = false;
        break;
      }
    }
    if (ok) {
      expath(st);
    } else {
      vector<int> deg(n, 0);
      for (int i = 0; i < n; i++) {
        for (int eid : g[i]) {
          auto& e = edges[eid];
          if (e.c - e.f > eps) {
            deg[e.to] += 1;
          }
```

```
    }
  }
  vector<int> que;
  for (int i = 0; i < n; i++) {
    if (deg[i] == 0) {
      que.push_back(i);
    }
  }
  for (int b = 0; b < (int) que.size(); b++) {
    for (int eid : g[que[b]]) {
      auto& e = edges[eid];
      if (e.c - e.f > eps) {
        deg[e.to] -= 1;
        if (deg[e.to] == 0) {
          que.push_back(e.to);
        }
      }
    }
  }
  fill(pot.begin(), pot.end(), INF_C);
  pot[st] = 0;
  if (static_cast<int>(que.size()) == n) {
    for (int v : que) {
      if (pot[v] < INF_C) {
        for (int eid : g[v]) {
          auto& e = edges[eid];
          if (e.c - e.f > eps) {
            if (pot[v] + e.cost < pot[e.to]) {
              pot[e.to] = pot[v] + e.cost;
              pe[e.to] = eid;
            }
          }
        }
      }
    }
  } else {
    que.assign(1, st);
    vector<bool> in_queue(n, false);
    in_queue[st] = true;
    for (int b = 0; b < (int) que.size(); b++) {
      int i = que[b];
      in_queue[i] = false;
      for (int id : g[i]) {
        const edge &e = edges[id];
        if (e.c - e.f > eps && pot[i] + e.cost < pot[e.to]) {
          pot[e.to] = pot[i] + e.cost;
          pe[e.to] = id;
          if (!in_queue[e.to]) {
            que.push_back(e.to);
            in_queue[e.to] = true;
          }
        }
      }
    }
  }
  while (pot[fin] < INF_C) {
    T push = numeric_limits<T>::max();
    int v = fin;
    while (v != st) {
      const edge &e = edges[pe[v]];
      push = min(push, e.c - e.f);
      v = e.from;
    }
    v = fin;
    while (v != st) {
      edge &e = edges[pe[v]];
      e.f += push;
      edge &back = edges[pe[v] ^ 1];
      back.f -= push;
      v = e.from;
    }
    flow += push;
    cost += push * pot[fin];
    expath(st);
  }
  return {flow, cost};
 }
};
```

2.16 mincut.cpp

```
template <typename T>
pair<T, vector<bool>> MinCut(vector<vector<T>> g) {
  int n = static_cast<int>(g.size());
  for (int i = 0; i < n; i++) {
```

```
    assert(static_cast<int>(g[i].size()) == n);
  }
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      assert(g[i][j] == g[j][i]);
    }
  }
  vector<vector<bool>> v(n, vector<bool>(n));
  for (int i = 0; i < n; i++) {
    v[i][i] = true;
  }
  vector<T> w(n);
  vector<bool> exists(n, true);
  vector<bool> in_a(n);
  T best_cost = numeric_limits<T>::max();
  vector<bool> best_cut;
  for (int ph = 0; ph < n - 1; ph++) {
    fill(in_a.begin(), in_a.end(), false);
    fill(w.begin(), w.end(), T(0));
    int prev = -1;
    for (int it = 0; it < n - ph; it++) {
      int sel = -1;
      for (int i = 0; i < n; i++) {
        if (exists[i] && !in_a[i] && (sel == -1 || w[i] > w[sel])) {
          sel = i;
        }
      }
      if (it == n - ph - 1) {
        if (w[sel] < best_cost) {
          best_cost = w[sel];
          best_cut = v[sel];
        }
        for (int i = 0; i < n; i++) {
          v[prev][i] = v[prev][i] | v[sel][i];
          g[prev][i] += g[sel][i];
          g[i][prev] += g[i][sel];
        }
        exists[sel] = false;
        break;
      }
      in_a[sel] = true;
      for (int i = 0; i < n; i++) {
        w[i] += g[sel][i];
      }
      prev = sel;
    }
  }
  return make_pair(best_cost, best_cut);
}
```

3 graph

3.1 bicone.cpp

```
template <typename T>
vector<int> find_bicone(dfs_undigraph<T> &g, int &cnt) {
  g.dfs_all();
  vector<int> vertex_comp(g.n);
  cnt = 0;
  for (int i : g.order) {
    if (g.pv[i] == -1 || g.min_depth[i] == g.depth[i]) {
      vertex_comp[i] = cnt++;
    } else {
      vertex_comp[i] = vertex_comp[g.pv[i]];
    }
  }
  return vertex_comp;
}
```

3.2 biconv.cpp

```
template <typename T>
vector<int> find_biconv(dfs_undigraph<T> &g, int &cnt) {
  g.dfs_all();
  vector<int> vertex_comp(g.n);
  cnt = 0;
  for (int i : g.order) {
    if (g.pv[i] == -1) {
      vertex_comp[i] = -1;
      continue;
    }
    if (g.min_depth[i] >= g.depth[g.pv[i]]) {
      vertex_comp[i] = cnt++;
    } else {
      vertex_comp[i] = vertex_comp[g.pv[i]];
```

```
    }
  }
  vector<int> edge_comp(g.edges.size(), -1);
  for (int id = 0; id < (int) g.edges.size(); id++) {
    int x = g.edges[id].from;
    int y = g.edges[id].to;
    int z = (g.depth[x] > g.depth[y] ? x : y);
    edge_comp[id] = vertex_comp[z];
  }
  return edge_comp;
}
```

3.3 bridges.cpp

```
template <typename T>
vector<bool> find_bridges(dfs_undigraph<T> &g) {
  g.dfs_all();
  vector<bool> bridge(g.edges.size(), false);
  for (int i = 0; i < g.n; i++) {
    if (g.pv[i] != -1 && g.min_depth[i] == g.depth[i]) {
      bridge[g.pe[i]] = true;
    }
  }
  return bridge;
}
```

3.4 cutpoints.cpp

```
template <typename T>
vector<bool> find_cutpoints(dfs_undigraph<T> &g) {
  g.dfs_all();
  vector<bool> cutpoint(g.n, false);
  for (int i = 0; i < g.n; i++) {
    if (g.pv[i] != -1 && g.min_depth[i] >= g.depth[g.pv[i]]) {
      cutpoint[g.pv[i]] = true;
    }
  }
  vector<int> children(g.n, 0);
  for (int i = 0; i < g.n; i++) {
    if (g.pv[i] != -1) {
      children[g.pv[i]]++;
    }
  }
  for (int i = 0; i < g.n; i++) {
    if (g.pv[i] == -1 && children[i] < 2) {
      cutpoint[i] = false;
    }
  }
  return cutpoint;
}
```

3.5 cycles.cpp

```
template <typename T>
vector<vector<int>> find_cycles(const graph<T> &g, int bound_cnt = 1 << 30,
    int bound_size = 1 << 30) {
  vector<int> was(g.n, -1);
  vector<int> st;
  vector<vector<int>> cycles;
  int total_size = 0;
  function<void(int, int)> dfs = [&](int v, int pe) {
    if ((int) cycles.size() >= bound_cnt || total_size >= bound_size) {
      return;
    }
    was[v] = (int) st.size();
    for (int id : g.g[v]) {
      if (id == pe) {
        continue;
      }
      auto &e = g.edges[id];
      int to = e.from ^ e.to ^ v;
      if (was[to] >= 0) {
        vector<int> cycle(1, id);
        for (int j = was[to]; j < (int) st.size(); j++) {
          cycle.push_back(st[j]);
        }
        cycles.push_back(cycle);
        total_size += (int) cycle.size();
        if ((int) cycles.size() >= bound_cnt || total_size >= bound_size) {
          was[v] = -2;
          return;
        }
        continue;
      }
      if (was[to] == -1) {
        st.push_back(id);
```

```
        dfs(to, id);
        st.pop_back();
      }
    }
    was[v] = -2;
  };
  for (int i = 0; i < g.n; i++) {
    if (was[i] == -1) {
      dfs(i, -1);
    }
  }
  return cycles;
  // cycles are given by edge ids, all cycles are simple
  // breaks after getting bound_cnt cycles or total_size >= bound_size
  // digraph: finds at least one cycle in every connected component (if not broken)
  // undigraph: finds cycle basis
}

template <typename T>
vector<int> edges_to_vertices(const graph<T> &g, const vector<int> &edge_cycle) {
  int sz = (int) edge_cycle.size();
  vector<int> vertex_cycle;
  if (sz <= 2) {
    vertex_cycle.push_back(g.edges[edge_cycle[0]].from);
    if (sz == 2) {
      vertex_cycle.push_back(g.edges[edge_cycle[0]].to);
    }
  } else {
    for (int i = 0; i < sz; i++) {
      int j = (i + 1) % sz;
      auto &e = g.edges[edge_cycle[i]];
      auto &other = g.edges[edge_cycle[j]];
      if (other.from == e.from || other.to == e.from) {
        vertex_cycle.push_back(e.to);
      } else {
        vertex_cycle.push_back(e.from);
      }
    }
  }
  return vertex_cycle;
  // only for simple cycles!
}
```

3.6 dfs-digraph-useless.cpp

```
template <typename T>
class dfs_digraph : public digraph<T> {
 public:
  using digraph<T>::edges;
  using digraph<T>::g;
  using digraph<T>::n;

  vector<int> pv;
  vector<int> pe;
  vector<int> order;
  vector<int> pos;
  vector<int> end;
  vector<int> sz;
  vector<int> root;
  vector<int> depth;
  vector<T> dist;

  dfs_digraph(int _n) : digraph<T>(_n) {
  }

  void clear() {
    pv.clear();
    pe.clear();
    order.clear();
    pos.clear();
    end.clear();
    sz.clear();
    root.clear();
    depth.clear();
    dist.clear();
  }

  void init() {
    pv = vector<int>(n, -1);
    pe = vector<int>(n, -1);
    order.clear();
    pos = vector<int>(n, -1);
    end = vector<int>(n, -1);
    sz = vector<int>(n, 0);
```

```
    root = vector<int>(n, -1);
    depth = vector<int>(n, -1);
    dist = vector<T>(n);
  }

 private:
  void do_dfs(int v) {
    pos[v] = (int) order.size();
    order.push_back(v);
    sz[v] = 1;
    for (int id : g[v]) {
      if (id == pe[v]) {
        continue;
      }
      auto &e = edges[id];
      int to = e.from ^ e.to ^ v;
      // well, this is controversial...
      if (depth[to] != -1) {
        continue;
      }
      depth[to] = depth[v] + 1;
      dist[to] = dist[v] + e.cost;
      pv[to] = v;
      pe[to] = id;
      root[to] = (root[v] != -1 ? root[v] : to);
      do_dfs(to);
      sz[v] += sz[to];
    }
    end[v] = (int) order.size() - 1;
  }

  void do_dfs_from(int v) {
    depth[v] = 0;
    dist[v] = T{};
    root[v] = v;
    pv[v] = pe[v] = -1;
    do_dfs(v);
  }

 public:
  int dfs_one_unsafe(int v) {
    // run init() before this
    // then run this with the required v's
    do_dfs_from(v);
    return v;
  }

  int dfs(int v) {
    init();
    do_dfs_from(v);
//    assert((int) order.size() == n);
    return v;
  }

  void dfs_many(const vector<int> &roots) {
    init();
    for (int v : roots) {
      if (depth[v] == -1) {
        do_dfs_from(v);
      }
    }
//    assert((int) order.size() == n);
  }

  vector<int> dfs_all() {
    init();
    vector<int> roots;
    for (int v = 0; v < n; v++) {
      if (depth[v] == -1) {
        roots.push_back(v);
        do_dfs_from(v);
      }
    }
    assert((int) order.size() == n);
    return roots;
  }
};
```

3.7 dfs-forest.cpp

```
template <typename T>
class dfs_forest : public forest<T> {
 public:
  using forest<T>::edges;
  using forest<T>::g;
  using forest<T>::n;
```

```
7
8   vector<int> pv;
9   vector<int> pe;
10  vector<int> order;
11  vector<int> pos;
12  vector<int> end;
13  vector<int> sz;
14  vector<int> root;
15  vector<int> depth;
16  vector<T> dist;
17
18  dfs_forest(int _n) : forest<T>(_n) {
19  }
20
21  void init() {
22    pv = vector<int>(n, -1);
23    pe = vector<int>(n, -1);
24    order.clear();
25    pos = vector<int>(n, -1);
26    end = vector<int>(n, -1);
27    sz = vector<int>(n, 0);
28    root = vector<int>(n, -1);
29    depth = vector<int>(n, -1);
30    dist = vector<T>(n);
31  }
32
33  void clear() {
34    pv.clear();
35    pe.clear();
36    order.clear();
37    pos.clear();
38    end.clear();
39    sz.clear();
40    root.clear();
41    depth.clear();
42    dist.clear();
43  }
44
45 private:
46  void do_dfs(int v) {
47    pos[v] = (int) order.size();
48    order.push_back(v);
49    sz[v] = 1;
50    for (int id : g[v]) {
51      if (id == pe[v]) {
52        continue;
53      }
54      auto &e = edges[id];
55      int to = e.from ^ e.to ^ v;
56      depth[to] = depth[v] + 1;
57      dist[to] = dist[v] + e.cost;
58      pv[to] = v;
59      pe[to] = id;
60      root[to] = (root[v] != -1 ? root[v] : to);
61      do_dfs(to);
62      sz[v] += sz[to];
63    }
64    end[v] = (int) order.size() - 1;
65  }
66
67  void do_dfs_from(int v) {
68    depth[v] = 0;
69    dist[v] = T{};
70    root[v] = v;
71    pv[v] = pe[v] = -1;
72    do_dfs(v);
73  }
74
75 public:
76  void dfs(int v, bool clear_order = true) {
77    if (pv.empty()) {
78      init();
79    } else {
80      if (clear_order) {
81        order.clear();
82      }
83    }
84    do_dfs_from(v);
85  }
86
87  void dfs_all() {
88    init();
89    for (int v = 0; v < n; v++) {
90      if (depth[v] == -1) {
91        do_dfs_from(v);
92      }
```

```
    }
    assert((int) order.size() == n);
  }
};
```

3.8 dfs-undigraph.cpp

```
template <typename T>
class dfs_undigraph : public undigraph<T> {
 public:
  using undigraph<T>::edges;
  using undigraph<T>::g;
  using undigraph<T>::n;

  vector<int> pv;
  vector<int> pe;
  vector<int> order;
  vector<int> pos;
  vector<int> end;
  vector<int> sz;
  vector<int> root;
  vector<int> depth;
  vector<int> min_depth;
  vector<T> dist;
  vector<int> was;
  int attempt;

  dfs_undigraph(int _n) : undigraph<T>(_n) {
  }

  void init() {
    pv = vector<int>(n, -1);
    pe = vector<int>(n, -1);
    order.clear();
    pos = vector<int>(n, -1);
    end = vector<int>(n, -1);
    sz = vector<int>(n, 0);
    root = vector<int>(n, -1);
    depth = vector<int>(n, -1);
    min_depth = vector<int>(n, -1);
    dist = vector<T>(n);
    was = vector<int>(n, -1);
    attempt = 0;
  }

  void clear() {
    pv.clear();
    pe.clear();
    order.clear();
    pos.clear();
    end.clear();
    sz.clear();
    root.clear();
    depth.clear();
    min_depth.clear();
    dist.clear();
    was.clear();
  }

 private:
  void do_dfs(int v) {
    was[v] = attempt;
    pos[v] = (int) order.size();
    order.push_back(v);
    sz[v] = 1;
    min_depth[v] = depth[v];
    for (int id : g[v]) {
      if (id == pe[v]) {
        continue;
      }
      auto &e = edges[id];
      int to = e.from ^ e.to ^ v;
      if (was[to] == attempt) {
        min_depth[v] = min(min_depth[v], depth[to]);
        continue;
      }
      depth[to] = depth[v] + 1;
      dist[to] = dist[v] + e.cost;
      pv[to] = v;
      pe[to] = id;
      root[to] = (root[v] != -1 ? root[v] : to);
      do_dfs(to);
      sz[v] += sz[to];
      min_depth[v] = min(min_depth[v], min_depth[to]);
    }
    end[v] = (int) order.size() - 1;
```

```
  }

  void do_dfs_from(int v) {
    ++attempt;
    depth[v] = 0;
    dist[v] = T{};
    root[v] = v;
    pv[v] = pe[v] = -1;
    do_dfs(v);
  }

 public:
  void dfs(int v, bool clear_order = true) {
    if (pv.empty()) {
      init();
    } else {
      if (clear_order) {
        order.clear();
      }
    }
    do_dfs_from(v);
  }

  void dfs_all() {
    init();
    for (int v = 0; v < n; v++) {
      if (depth[v] == -1) {
        do_dfs_from(v);
      }
    }
    assert((int) order.size() == n);
  }
};
```

3.9 digraph.cpp

```
template <typename T>
class digraph : public graph<T> {
 public:
  using graph<T>::edges;
  using graph<T>::g;
  using graph<T>::n;

  digraph(int _n) : graph<T>(_n) {
  }

  int add(int from, int to, T cost = 1) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = (int) edges.size();
    g[from].push_back(id);
    edges.push_back({from, to, cost});
    return id;
  }

  digraph<T> reverse() const {
    digraph<T> rev(n);
    for (auto &e : edges) {
      rev.add(e.to, e.from, e.cost);
    }
    return rev;
  }
};
```

3.10 dijkstra-set.cpp

```
template <typename T>
vector<T> dijkstra(const graph<T> &g, int start) {
  assert(0 <= start && start < g.n);
  vector<T> dist(g.n, numeric_limits<T>::max());
  dist[start] = 0;
  set<pair<T, int>> s;
  s.emplace(dist[start], start);
  while (!s.empty()) {
    int i = s.begin()->second;
    s.erase(s.begin());
    for (int id : g.g[i]) {
      auto &e = g.edges[id];
      int to = e.from ^ e.to ^ i;
      if (dist[i] + e.cost < dist[to]) {
        s.erase({dist[to], to});
        dist[to] = dist[i] + e.cost;
        s.emplace(dist[to], to);
      }
    }
  }
  return dist;
```

```
  // returns numeric_limits<T>::max() if there's no path
}
```

3.11 dijkstra.cpp

```
template <typename T>
vector<T> dijkstra(const graph<T> &g, int start) {
  assert(0 <= start && start < g.n);
  vector<T> dist(g.n, numeric_limits<T>::max());
  priority_queue<pair<T, int>, vector<pair<T, int>>, greater<pair<T, int>>>
      s;
  dist[start] = 0;
  s.emplace(dist[start], start);
  while (!s.empty()) {
    T expected = s.top().first;
    int i = s.top().second;
    s.pop();
    if (dist[i] != expected) {
      continue;
    }
    for (int id : g.g[i]) {
      auto &e = g.edges[id];
      int to = e.from ^ e.to ^ i;
      if (dist[i] + e.cost < dist[to]) {
        dist[to] = dist[i] + e.cost;
        s.emplace(dist[to], to);
      }
    }
  }
  return dist;
  // returns numeric_limits<T>::max() if there's no path
}
```

3.12 dominators.cpp

```
template <typename T>
vector<int> find_dominators(const digraph<T> &g, int root) {
  int n = g.n;
  vector<int> pos(n, -1);
  vector<int> order;
  vector<int> parent(n, -1);
  function<void(int)> dfs = [&g, &pos, &order, &parent, &dfs](int v) {
    pos[v] = (int) order.size();
    order.push_back(v);
    for (int id : g.g[v]) {
      auto &e = g.edges[id];
      int u = e.to;
      if (pos[u] == -1) {
        parent[u] = v;
        dfs(u);
      }
    }
  };
  dfs(root);
  vector<int> p(n), best(n);
  iota(p.begin(), p.end(), 0);
  iota(best.begin(), best.end(), 0);
  vector<int> sdom = pos;
  function<int(int)> find_best = [&p, &best, &sdom, &find_best](int x) {
    if (p[x] != x) {
      int u = find_best(p[x]);
      if (sdom[u] < sdom[best[x]]) {
        best[x] = u;
      }
      p[x] = p[p[x]];
    }
    if (sdom[best[p[x]]] < sdom[best[x]]) {
      best[x] = best[p[x]];
    }
    return best[x];
  };
  digraph<int> g_rev = g.reverse();
  vector<int> idom(n, -1);
  vector<int> link(n, 0);
  vector<vector<int>> bucket(n);
  for (int it = (int) order.size() - 1; it >= 0; it--) {
    int w = order[it];
    for (int id : g_rev.g[w]) {
      auto &e = g_rev.edges[id];
      int u = e.to;
      if (pos[u] != -1) {
        sdom[w] = min(sdom[w], sdom[find_best(u)]);
      }
    }
    idom[w] = order[sdom[w]];
    for (int u : bucket[w]) {
```

```
      link[u] = find_best(u);
    }
    for (int id : g.g[w]) {
      auto &e = g.edges[id];
      int u = e.to;
      if (parent[u] == w) {
        p[u] = w;
      }
    }
    bucket[order[sdom[w]]].push_back(w);
  }
  for (int it = 1; it < (int) order.size(); it++) {
    int w = order[it];
    idom[w] = idom[link[w]];
  }
  return idom;
  // idom[i] -- immediate dominator for vertex i
}
```

3.13 eulerian.cpp

```
template <typename T>
vector<int> find_eulerian_path(const graph<T> &g, int &root) {
  // in_deg and out_deg are fake for undigraph!
  vector<int> in_deg(g.n, 0);
  vector<int> out_deg(g.n, 0);
  int cnt_edges = 0;
  for (int id = 0; id < (int) g.edges.size(); id++) {
    cnt_edges++;
    auto &e = g.edges[id];
    out_deg[e.from]++;
    in_deg[e.to]++;
  }
  root = -1;
  int odd = 0;
  for (int i = 0; i < g.n; i++) {
    if ((in_deg[i] + out_deg[i]) % 2 == 1) {
      odd++;
      if (root == -1 || out_deg[i] - in_deg[i] > out_deg[root] - in_deg[root
          ]) {
        root = i;
      }
    }
  }
  if (odd > 2) {
    root = -1;
    return vector<int>();
  }
  if (root == -1) {
    root = 0;
    while (root < g.n && in_deg[root] + out_deg[root] == 0) {
      root++;
    }
    if (root == g.n) {
      // an empty path
      root = 0;
      return vector<int>();
    }
  }
  vector<bool> used(g.edges.size(), false);
  vector<int> ptr(g.n, 0);
  vector<int> balance(g.n, 0);
  vector<int> res(cnt_edges);
  int stack_ptr = 0;
  int write_ptr = cnt_edges;
  int v = root;
  while (true) {
    bool found = false;
    while (ptr[v] < (int) g.g[v].size()) {
      int id = g.g[v][ptr[v]++];
      if (used[id]) {
        continue;
      }
      used[id] = true;
      res[stack_ptr++] = id;
      auto &e = g.edges[id];
      balance[v]++;
      v ^= e.from ^ e.to;
      balance[v]--;
      found = true;
      break;
    }
    if (!found) {
      if (stack_ptr == 0) {
        break;
      }
```

```
      int id = res[--stack_ptr];
      res[--write_ptr] = id;
      auto &e = g.edges[id];
      v ^= e.from ^ e.to;
    }
  }
  int disbalance = 0;
  for (int i = 0; i < g.n; i++) {
    disbalance += abs(balance[i]);
  }
  if (write_ptr != 0 || disbalance > 2) {
    root = -1;
    return vector<int>();
  }
  return res;
  // returns edge ids in the path (or the cycle if it exists)
  // root == -1 if there is no path
  // (or res.empty(), but this is also true when there are no edges)
}
```

3.14 forest.cpp

```
template <typename T>
class forest : public graph<T> {
 public:
  using graph<T>::edges;
  using graph<T>::g;
  using graph<T>::n;

  forest(int _n) : graph<T>(_n) {
  }

  int add(int from, int to, T cost = 1) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = (int) edges.size();
    assert(id < n - 1);
    g[from].push_back(id);
    g[to].push_back(id);
    edges.push_back({from, to, cost});
    return id;
  }
};
```

3.15 graph.cpp

```
template <typename T>
class graph {
 public:
  struct edge {
    int from;
    int to;
    T cost;
  };

  vector<edge> edges;
  vector<vector<int>> g;
  int n;

  graph(int _n) : n(_n) {
    g.resize(n);
  }

  virtual int add(int from, int to, T cost) = 0;
};
```

3.16 hld-forest.cpp

```
template <typename T>
class hld_forest : public dfs_forest<T> {
 public:
  using dfs_forest<T>::edges;
  using dfs_forest<T>::g;
  using dfs_forest<T>::n;
  using dfs_forest<T>::pv;
  using dfs_forest<T>::sz;
  using dfs_forest<T>::root;
  using dfs_forest<T>::pos;
  using dfs_forest<T>::end;
  using dfs_forest<T>::order;
  using dfs_forest<T>::depth;
  using dfs_forest<T>::dfs;
  using dfs_forest<T>::dfs_all;

  vector<int> head;
  vector<int> visited;

  hld_forest(int _n) : dfs_forest<T>(_n) {
```

```
    visited.resize(n);
  }

  void build_hld(const vector<int> &vs) {
    for (int tries = 0; tries < 2; tries++) {
      if (vs.empty()) {
        dfs_all();
      } else {
        order.clear();
        for (int v : vs) {
          dfs(v, false);
        }
        assert((int) order.size() == n);
      }
      if (tries == 1) {
        break;
      }
      for (int i = 0; i < n; i++) {
        if (g[i].empty()) {
          continue;
        }
        int best = -1, bid = 0;
        for (int j = 0; j < (int) g[i].size(); j++) {
          int id = g[i][j];
          int v = edges[id].from ^ edges[id].to ^ i;
          if (pv[v] != i) {
            continue;
          }
          if (sz[v] > best) {
            best = sz[v];
            bid = j;
          }
        }
        swap(g[i][0], g[i][bid]);
      }
    }
    head.resize(n);
    for (int i = 0; i < n; i++) {
      head[i] = i;
    }
    for (int i = 0; i < n - 1; i++) {
      int x = order[i];
      int y = order[i + 1];
      if (pv[y] == x) {
        head[y] = head[x];
      }
    }
  }

  void build_hld(int v) {
    build_hld(vector<int>(1, v));
  }

  void build_hld_all() {
    build_hld(vector<int>());
  }

  bool apply_on_path(int x, int y, bool with_lca, function<void(int,int,bool
      )> f) {
    // f(x, y, up): up -- whether this part of the path goes up
    assert(!head.empty());
    int z = lca(x, y);
    if (z == -1) {
      return false;
    }
    {
      int v = x;
      while (v != z) {
        if (depth[head[v]] <= depth[z]) {
          f(pos[z] + 1, pos[v], true);
          break;
        }
        f(pos[head[v]], pos[v], true);
        v = pv[head[v]];
      }
    }
    if (with_lca) {
      f(pos[z], pos[z], false);
    }
    {
      int v = y;
      int cnt_visited = 0;
      while (v != z) {
        if (depth[head[v]] <= depth[z]) {
          f(pos[z] + 1, pos[v], false);
          break;
```

```
        }
        visited[cnt_visited++] = v;
        v = pv[head[v]];
      }
      for (int at = cnt_visited - 1; at >= 0; at--) {
        v = visited[at];
        f(pos[head[v]], pos[v], false);
      }
    }
    return true;
  }

  inline bool anc(int x, int y) {
    return (pos[x] <= pos[y] && end[y] <= end[x]);
  }

  inline int go_up(int x, int up) {
    int target = depth[x] - up;
    if (target < 0) {
      return -1;
    }
    while (depth[head[x]] > target) {
      x = pv[head[x]];
    }
    return order[pos[x] - depth[x] + target];
  }

  inline int lca(int x, int y) {
    if (root[x] != root[y]) {
      return -1;
    }
    while (head[x] != head[y]) {
      if (depth[head[x]] > depth[head[y]]) {
        x = pv[head[x]];
      } else {
        y = pv[head[y]];
      }
    }
    return depth[x] < depth[y] ? x : y;
  }
};
```

3.17 hld-forest-old.cpp

```
template <typename T>
class hld_forest_old : public lca_forest<T> {
 public:
  using lca_forest<T>::edges;
  using lca_forest<T>::g;
  using lca_forest<T>::n;
  using lca_forest<T>::pv;
  using lca_forest<T>::sz;
  using lca_forest<T>::pos;
  using lca_forest<T>::order;
  using lca_forest<T>::depth;
  using lca_forest<T>::dfs;
  using lca_forest<T>::dfs_all;
  using lca_forest<T>::lca;
  using lca_forest<T>::build_lca;

  vector<int> head;
  vector<int> visited;

  hld_forest_old(int _n) : lca_forest<T>(_n) {
    visited.resize(n);
  }

  void build_hld(const vector<int> &vs) {
    for (int tries = 0; tries < 2; tries++) {
      if (vs.empty()) {
        dfs_all();
      } else {
        order.clear();
        for (int v : vs) {
          dfs(v, false);
        }
        assert((int) order.size() == n);
      }
      if (tries == 1) {
        break;
      }
      for (int i = 0; i < n; i++) {
        if (g[i].empty()) {
          continue;
        }
        int best = -1, bid = 0;
```

```
        for (int j = 0; j < (int) g[i].size(); j++) {
          int id = g[i][j];
          int v = edges[id].from ^ edges[id].to ^ i;
          if (pv[v] != i) {
            continue;
          }
          if (sz[v] > best) {
            best = sz[v];
            bid = j;
          }
        }
        swap(g[i][0], g[i][bid]);
      }
    }
    build_lca();
    head.resize(n);
    for (int i = 0; i < n; i++) {
      head[i] = i;
    }
    for (int i = 0; i < n - 1; i++) {
      int x = order[i];
      int y = order[i + 1];
      if (pv[y] == x) {
        head[y] = head[x];
      }
    }
  }

  void build_hld(int v) {
    build_hld(vector<int>(1, v));
  }

  void build_hld_all() {
    build_hld(vector<int>());
  }

  bool apply_on_path(int x, int y, bool with_lca, function<void(int,int,bool
      )> f) {
    // f(x, y, up): up -- whether this part of the path goes up
    assert(!head.empty());
    int z = lca(x, y);
    if (z == -1) {
      return false;
    }
    {
      int v = x;
      while (v != z) {
        if (depth[head[v]] <= depth[z]) {
          f(pos[z] + 1, pos[v], true);
          break;
        }
        f(pos[head[v]], pos[v], true);
        v = pv[head[v]];
      }
    }
    if (with_lca) {
      f(pos[z], pos[z], false);
    }
    {
      int v = y;
      int cnt_visited = 0;
      while (v != z) {
        if (depth[head[v]] <= depth[z]) {
          f(pos[z] + 1, pos[v], false);
          break;
        }
        visited[cnt_visited++] = v;
        v = pv[head[v]];
      }
      for (int at = cnt_visited - 1; at >= 0; at--) {
        v = visited[at];
        f(pos[head[v]], pos[v], false);
      }
    }
    return true;
  }
};
```

3.18 lca-forest.cpp

```
template <typename T>
class lca_forest : public dfs_forest<T> {
 public:
  using dfs_forest<T>::edges;
  using dfs_forest<T>::g;
  using dfs_forest<T>::n;
```

```
  using dfs_forest<T>::pv;
  using dfs_forest<T>::pos;
  using dfs_forest<T>::end;
  using dfs_forest<T>::depth;

  int h;
  vector<vector<int>> pr;

  lca_forest(int _n) : dfs_forest<T>(_n) {
  }

  inline void build_lca() {
    assert(!pv.empty());
    int max_depth = 0;
    for (int i = 0; i < n; i++) {
      max_depth = max(max_depth, depth[i]);
    }
    h = 1;
    while ((1 << h) <= max_depth) {
      h++;
    }
    pr.resize(n);
    for (int i = 0; i < n; i++) {
      pr[i].resize(h);
      pr[i][0] = pv[i];
    }
    for (int j = 1; j < h; j++) {
      for (int i = 0; i < n; i++) {
        pr[i][j] = (pr[i][j - 1] == -1 ? -1 : pr[pr[i][j - 1]][j - 1]);
      }
    }
  }

  inline bool anc(int x, int y) {
    return (pos[x] <= pos[y] && end[y] <= end[x]);
  }

  inline int go_up(int x, int up) {
    assert(!pr.empty());
    up = min(up, (1 << h) - 1);
    for (int j = h - 1; j >= 0; j--) {
      if (up & (1 << j)) {
        x = pr[x][j];
        if (x == -1) {
          break;
        }
      }
    }
    return x;
  }

  inline int lca(int x, int y) {
    assert(!pr.empty());
    if (anc(x, y)) {
      return x;
    }
    if (anc(y, x)) {
      return y;
    }
    for (int j = h - 1; j >= 0; j--) {
      if (pr[x][j] != -1 && !anc(pr[x][j], y)) {
        x = pr[x][j];
      }
    }
    return pr[x][0];
  }
};
```

3.19 mst.cpp

```
template <typename T>
vector<int> find_mst(const undigraph<T> &g, T &ans) {
  vector<int> order(g.edges.size());
  iota(order.begin(), order.end(), 0);
  sort(order.begin(), order.end(), [&g](int a, int b) {
    return g.edges[a].cost < g.edges[b].cost;
  });
  dsu d(g.n);
  vector<int> ans_list;
  ans = 0;
  for (int id : order) {
    auto &e = g.edges[id];
    if (d.get(e.from) != d.get(e.to)) {
      d.unite(e.from, e.to);
      ans_list.push_back(id);
      ans += e.cost;
```

```
    }
  }
  return ans_list;
  // returns edge ids of minimum "spanning" forest
}
```

3.20 scc.cpp

```
template <typename T>
vector<int> find_scc(const digraph<T> &g, int &cnt) {
  digraph<T> g_rev = g.reverse();
  vector<int> order;
  vector<bool> was(g.n, false);
  function<void(int)> dfs1 = [&](int v) {
    was[v] = true;
    for (int id : g.g[v]) {
      auto &e = g.edges[id];
      int to = e.to;
      if (!was[to]) {
        dfs1(to);
      }
    }
    order.push_back(v);
  };
  for (int i = 0; i < g.n; i++) {
    if (!was[i]) {
      dfs1(i);
    }
  }
  vector<int> c(g.n, -1);
  function<void(int)> dfs2 = [&](int v) {
    for (int id : g_rev.g[v]) {
      auto &e = g_rev.edges[id];
      int to = e.to;
      if (c[to] == -1) {
        c[to] = c[v];
        dfs2(to);
      }
    }
  };
  cnt = 0;
  for (int id = g.n - 1; id >= 0; id--) {
    int i = order[id];
    if (c[i] != -1) {
      continue;
    }
    c[i] = cnt++;
    dfs2(i);
  }
  return c;
  // c[i] <= c[j] for every edge i -> j
}
```

3.21 topsort.cpp

```
template <typename T>
vector<int> find_topsort(const digraph<T> &g) {
  vector<int> deg(g.n, 0);
  for (int id = 0; id < (int) g.edges.size(); id++) {
    deg[g.edges[id].to]++;
  }
  vector<int> x;
  for (int i = 0; i < g.n; i++) {
    if (deg[i] == 0) {
      x.push_back(i);
    }
  }
  for (int ptr = 0; ptr < (int) x.size(); ptr++) {
    int i = x[ptr];
    for (int id : g.g[i]) {
      auto &e = g.edges[id];
      int to = e.to;
      if (--deg[to] == 0) {
        x.push_back(to);
      }
    }
  }
  if ((int) x.size() != g.n) {
    return vector<int>();
  }
  return x;
}
```

3.22 tree-dp.cpp

```
struct Data {
```

```
  ${0}... a = ...;
};
auto Unite = [&](const Data& a, const Data& b) -> Data {
  return ...;
};
auto AddVertex = [&](const Data& a, int v) -> Data {
  return ...;
};
auto MoveUp = [&](const Data& a, int v, int eid) -> Data {
  auto& e = g.edges[eid];
  return ...;
};
g.dfs(0);
vector<Data> down(g.n);
vector<Data> up(g.n);
vector<Data> dp(g.n);
{
  for (int it = g.n - 1; it >= 0; it--) {
    int i = g.order[it];
    for (int eid : g.g[i]) {
      auto& e = g.edges[eid];
      int to = e.from ^ e.to ^ i;
      if (to == g.pv[i]) {
        continue;
      }
      down[i] = Unite(down[i], MoveUp(down[to], i, eid));
    }
    down[i] = AddVertex(down[i], i);
  }
  for (int it = 0; it < g.n; it++) {
    int i = g.order[it];
    vector<int> children;
    vector<Data> vals;
    for (int eid : g.g[i]) {
      auto& e = g.edges[eid];
      int to = e.from ^ e.to ^ i;
      if (to == g.pv[i]) {
        continue;
      }
      children.push_back(to);
      vals.push_back(MoveUp(down[to], i, eid));
    }
    vector<Data> suf(vals.size() + 1);
    for (int j = int(vals.size()) - 1; j >= 0; j--) {
      suf[j] = Unite(vals[j], suf[j + 1]);
    }
    Data pref;
    if (g.pv[i] != -1) {
      pref = MoveUp(up[i], i, g.pe[i]);
    }
    for (int j = 0; j < int(vals.size()); j++) {
      up[children[j]] = AddVertex(Unite(pref, suf[j + 1]), i);
      pref = Unite(pref, vals[j]);
    }
    dp[i] = AddVertex(pref, i);
  }
}
```

3.23 twosat.cpp

```
class twosat {
 public:
  digraph<int> g;
  int n;

  twosat(int _n) : g(digraph<int>(2 * _n)), n(_n) {
  }

  // (v[x] == value_x)
  inline void add(int x, int value_x) {
    assert(0 <= x && x < n);
    assert(0 <= value_x && value_x <= 1);
    g.add(2 * x + (value_x ^ 1), 2 * x + value_x);
  }

  // (v[x] == value_x || v[y] == value_y)
  inline void add(int x, int value_x, int y, int value_y) {
    assert(0 <= x && x < n && 0 <= y && y < n);
    assert(0 <= value_x && value_x <= 1 && 0 <= value_y && value_y <= 1);
    g.add(2 * x + (value_x ^ 1), 2 * y + value_y);
    g.add(2 * y + (value_y ^ 1), 2 * x + value_x);
  }

  inline vector<int> solve() {
    int cnt;
    vector<int> c = find_scc(g, cnt);
```

```
    vector<int> res(n);
    for (int i = 0; i < n; i++) {
      if (c[2 * i] == c[2 * i + 1]) {
        return vector<int>();
      }
      res[i] = (c[2 * i] < c[2 * i + 1]);
    }
    return res;
  }
};
```

3.24 undigraph.cpp

```
template <typename T>
class undigraph : public graph<T> {
 public:
  using graph<T>::edges;
  using graph<T>::g;
  using graph<T>::n;

  undigraph(int _n) : graph<T>(_n) {
  }

  int add(int from, int to, T cost = 1) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = (int) edges.size();
    g[from].push_back(id);
    g[to].push_back(id);
    edges.push_back({from, to, cost});
    return id;
  }
};
```

4 misc

4.1 debug.cpp

```
template <typename A, typename B>
string to_string(pair<A, B> p);

template <typename A, typename B, typename C>
string to_string(tuple<A, B, C> p);

template <typename A, typename B, typename C, typename D>
string to_string(tuple<A, B, C, D> p);

string to_string(const string& s) {
  return '"' + s + '"';
}

string to_string(const char* s) {
  return to_string((string) s);
}

string to_string(bool b) {
  return (b ? "true" : "false");
}

string to_string(vector<bool> v) {
  bool first = true;
  string res = "{";
  for (int i = 0; i < static_cast<int>(v.size()); i++) {
    if (!first) {
      res += ",␣";
    }
    first = false;
    res += to_string(v[i]);
  }
  res += "}";
  return res;
}

template <size_t N>
string to_string(bitset<N> v) {
  string res = "";
  for (size_t i = 0; i < N; i++) {
    res += static_cast<char>('0' + v[i]);
  }
  return res;
}

template <typename A>
string to_string(A v) {
  bool first = true;
  string res = "{";
```

```
  for (const auto &x : v) {
    if (!first) {
      res += ",␣";
    }
    first = false;
    res += to_string(x);
  }
  res += "}";
  return res;
}

template <typename A, typename B>
string to_string(pair<A, B> p) {
  return "(" + to_string(p.first) + ",␣" + to_string(p.second) + ")";
}

template <typename A, typename B, typename C>
string to_string(tuple<A, B, C> p) {
  return "(" + to_string(get<0>(p)) + ",␣" + to_string(get<1>(p)) + ",␣" +
      to_string(get<2>(p)) + ")";
}

template <typename A, typename B, typename C, typename D>
string to_string(tuple<A, B, C, D> p) {
  return "(" + to_string(get<0>(p)) + ",␣" + to_string(get<1>(p)) + ",␣" +
      to_string(get<2>(p)) + ",␣" + to_string(get<3>(p)) + ")";
}

void debug_out() { cerr << endl; }

template <typename Head, typename... Tail>
void debug_out(Head H, Tail... T) {
  cerr << "␣" << to_string(H);
  debug_out(T...);
}

#ifdef LOCAL
#define debug(...) cerr << "[" << #__VA_ARGS__ << "]:", debug_out(
    __VA_ARGS__)
#else
#define debug(...) 42
#endif
```

4.2 fastinput.cpp

```
static struct FastInput {
  static constexpr int BUF_SIZE = 1 << 20;
  char buf[BUF_SIZE];
  size_t chars_read = 0;
  size_t buf_pos = 0;
  FILE *in = stdin;
  char cur = 0;

  inline char get_char() {
    if (buf_pos >= chars_read) {
      chars_read = fread(buf, 1, BUF_SIZE, in);
      buf_pos = 0;
      buf[0] = (chars_read == 0 ? -1 : buf[0]);
    }
    return cur = buf[buf_pos++];
  }

  inline void tie(int) {}

  inline explicit operator bool() {
    return cur != -1;
  }

  inline static bool is_blank(char c) {
    return c <= '␣';
  }

  inline bool skip_blanks() {
    while (is_blank(cur) && cur != -1) {
      get_char();
    }
    return cur != -1;
  }

  inline FastInput& operator>>(char& c) {
    skip_blanks();
    c = cur;
    return *this;
  }

  inline FastInput& operator>>(string& s) {
    if (skip_blanks()) {
```

```
      s.clear();
      do {
        s += cur;
      } while (!is_blank(get_char()));
    }
    return *this;
  }

  template <typename T>
  inline FastInput& read_integer(T& n) {
    // unsafe, doesn't check that characters are actually digits
    n = 0;
    if (skip_blanks()) {
      int sign = +1;
      if (cur == '-') {
        sign = -1;
        get_char();
      }
      do {
        n += n + (n << 3) + cur - '0';
      } while (!is_blank(get_char()));
      n *= sign;
    }
    return *this;
  }

  template <typename T>
  inline typename enable_if<is_integral<T>::value, FastInput&>::type
      operator>>(T& n) {
    return read_integer(n);
  }

  #if !defined(_WIN32) || defined(_WIN64)
  inline FastInput& operator>>(__int128& n) {
    return read_integer(n);
  }
  #endif

  template <typename T>
  inline typename enable_if<is_floating_point<T>::value, FastInput&>::type
      operator>>(T& n) {
    // not sure if really fast, for compatibility only
    n = 0;
    if (skip_blanks()) {
      string s;
      (*this) >> s;
      sscanf(s.c_str(), "%lf", &n);
    }
    return *this;
  }
} fast_input;

#define cin fast_input
```

4.3 fastoutput.cpp

```
static struct FastOutput {
  static constexpr int BUF_SIZE = 1 << 20;
  char buf[BUF_SIZE];
  size_t buf_pos = 0;
  static constexpr int TMP_SIZE = 1 << 20;
  char tmp[TMP_SIZE];
  FILE *out = stdout;

  inline void put_char(char c) {
    buf[buf_pos++] = c;
    if (buf_pos == BUF_SIZE) {
      fwrite(buf, 1, buf_pos, out);
      buf_pos = 0;
    }
  }

  ~FastOutput() {
    fwrite(buf, 1, buf_pos, out);
  }

  inline FastOutput& operator<<(char c) {
    put_char(c);
    return *this;
  }

  inline FastOutput& operator<<(const char* s) {
    while (*s) {
      put_char(*s++);
    }
    return *this;
```

```
  }

  inline FastOutput& operator<<(const string& s) {
    for (int i = 0; i < (int) s.size(); i++) {
      put_char(s[i]);
    }
    return *this;
  }

  template <typename T>
  inline char* integer_to_string(T n) {
    // beware of TMP_SIZE
    char* p = tmp + TMP_SIZE - 1;
    if (n == 0) {
      *--p = '0';
    } else {
      bool is_negative = false;
      if (n < 0) {
        is_negative = true;
        n = -n;
      }
      while (n > 0) {
        *--p = (char) ('0' + n % 10);
        n /= 10;
      }
      if (is_negative) {
        *--p = '-';
      }
    }
    return p;
  }

  template <typename T>
  inline typename enable_if<is_integral<T>::value, char*>::type stringify(T
      n) {
    return integer_to_string(n);
  }

  #if !defined(_WIN32) || defined(_WIN64)
  inline char* stringify(__int128 n) {
    return integer_to_string(n);
  }
  #endif

  template <typename T>
  inline typename enable_if<is_floating_point<T>::value, char*>::type
      stringify(T n) {
    sprintf(tmp, "%.17f", n);
    return tmp;
  }

  template <typename T>
  inline FastOutput& operator<<(const T& n) {
    auto p = stringify(n);
    for (; *p != 0; p++) {
      put_char(*p);
    }
    return *this;
  }
} fast_output;

#define cout fast_output
```

4.4 lis.cpp

```
template<typename T>
int lis(const vector<T>& a) {
  vector<T> u;
  for (const T& x : a) {
    auto it = lower_bound(u.begin(), u.end(), x);
    if (it == u.end()) {
      u.push_back(x);
    } else {
      *it = x;
    }
  }
  return (int) u.size();
}
```

4.5 pragma.cpp

```
#pragma GCC optimize("O3,unroll-loops")
// #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
```

4.6 radix.cpp

```cpp
namespace radix {

vector<int> p(65537);

template<typename T>
void SortShift(vector<T>& a, vector<T>& new_a, int shift) {
  assert(a.size() == new_a.size());
  int n = static_cast<int>(a.size());
  fill(p.begin(), p.end(), 0);
  for (int i = 0; i < n; i++) p[1 + ((a[i] >> shift) & 0xffff)]++;
  for (int i = 1; i <= 65536; i++) p[i] += p[i - 1];
  for (int i = 0; i < n; i++) new_a[p[(a[i] >> shift) & 0xffff]++] = a[i];
}

void Sort(vector<int32_t>& a) {
  constexpr int32_t flip = static_cast<int32_t>(1) << 31;
  for (auto& aa : a) aa ^= flip;
  vector<int32_t> b(a.size());
  SortShift(a, b, 0);
  SortShift(b, a, 16);
  for (auto& aa : a) aa ^= flip;
}

void Sort(vector<uint32_t>& a) {
  vector<uint32_t> b(a.size());
  SortShift(a, b, 0);
  SortShift(b, a, 16);
}

void Sort(vector<int64_t>& a) {
  constexpr int64_t flip = static_cast<int64_t>(1) << 63;
  for (auto& aa : a) aa ^= flip;
  vector<int64_t> b(a.size());
  SortShift(a, b, 0);
  SortShift(b, a, 16);
  SortShift(a, b, 32);
  SortShift(b, a, 48);
  for (auto& aa : a) aa ^= flip;
}

void Sort(vector<uint64_t>& a) {
  vector<uint64_t> b(a.size());
  SortShift(a, b, 0);
  SortShift(b, a, 16);
  SortShift(a, b, 32);
  SortShift(b, a, 48);
}

}  // namespace radix
```

4.7 rng.cpp

```cpp
mt19937_64 rng((unsigned int) chrono::steady_clock::now().time_since_epoch()
    .count());
```

5 numeric

5.1 bm.cpp

```cpp
template <typename T>
vector<T> BM(vector<T> a) {
  vector<T> p = {1};
  vector<T> q = {1};
  int l = 0;
  for (int r = 1; r <= (int) a.size(); r++) {
    T delta = 0;
    for (int j = 0; j <= l; j++) {
      delta += a[r - 1 - j] * p[j];
    }
    q.insert(q.begin(), 0);
    if (delta != 0) {
      vector<T> t = p;
      if (q.size() > t.size()) {
        t.resize(q.size());
      }
      for (int i = 0; i < (int) q.size(); i++) {
        t[i] -= delta * q[i];
      }
      if (2 * l <= r - 1) {
        q = p;
        T od = 1 / delta;
        for (T& x : q) {
          x *= od;
        }
        l = r - l;
```

```
      }
      swap(p, t);
    }
  }
  assert((int) p.size() == l + 1);
//  assert(l * 2 + 30 < (int) a.size());
  reverse(p.begin(), p.end());
  return p;
}
```

5.2 extgcd.cpp

```
template<typename T>
T extgcd(T a, T b, T &x, T &y) {
  if (a == 0) {
    x = 0;
    y = 1;
    return b;
  }
  T p = b / a;
  T g = extgcd(b - p * a, a, y, x);
  x -= p * y;
  return g;
}

template<typename T>
bool diophantine(T a, T b, T c, T &x, T &y, T &g) {
  if (a == 0 && b == 0) {
    if (c == 0) {
      x = y = g = 0;
      return true;
    }
    return false;
  }
  if (a == 0) {
    if (c % b == 0) {
      x = 0;
      y = c / b;
      g = abs(b);
      return true;
    }
    return false;
  }
  if (b == 0) {
    if (c % a == 0) {
      x = c / a;
      y = 0;
      g = abs(a);
      return true;
    }
    return false;
  }
  g = extgcd(a, b, x, y);
  if (c % g != 0) {
    return false;
  }
  T dx = c / a;
  c -= dx * a;
  T dy = c / b;
  c -= dy * b;
  x = dx + (T) ((__int128) x * (c / g) % b);
  y = dy + (T) ((__int128) y * (c / g) % a);
  g = abs(g);
  return true;
  // |x|, |y| <= max(|a|, |b|, |c|) [tested]
}

bool crt(long long k1, long long m1, long long k2, long long m2, long long &
    k, long long &m) {
  k1 %= m1;
  if (k1 < 0) k1 += m1;
  k2 %= m2;
  if (k2 < 0) k2 += m2;
  long long x, y, g;
  if (!diophantine(m1, -m2, k2 - k1, x, y, g)) {
    return false;
  }
  long long dx = m2 / g;
  long long delta = x / dx - (x % dx < 0);
  k = m1 * (x - dx * delta) + k1;
  m = m1 / g * m2;
  assert(0 <= k && k < m);
  return true;
}

// for distinct prime modulos
```

```cpp
template <typename T>
void crt_garner(const vector<int>& p, const vector<int>& a, T& res) {
  assert(p.size() == a.size());
  auto inverse = [&](int q, int m) {
    q %= m;
    if (q < 0) q += m;
    int b = m, u = 0, v = 1;
    while (q) {
      int t = b / q;
      b -= t * q; swap(q, b);
      u -= t * v; swap(u, v);
    }
    assert(b == 1);
    if (u < 0) u += m;
    return u;
  };
  vector<int> x(p.size());
  for (int i = 0; i < (int) p.size(); i++) {
    assert(0 <= a[i] && a[i] < p[i]);
    x[i] = a[i];
    for (int j = 0; j < i; j++) {
      x[i] = (int) ((long long) (x[i] - x[j]) * inverse(p[j], p[i]) % p[i]);
      if (x[i] < 0) x[i] += p[i];
    }
  }
  res = 0;
  for (int i = (int) p.size() - 1; i >= 0; i--) {
    res = res * p[i] + x[i];
  }
}
```

5.3 factorizer.cpp

```cpp
namespace factorizer {

template <typename T>
struct FactorizerVarMod { static T value; };
template <typename T>
T FactorizerVarMod<T>::value;

template <typename T>
bool IsPrime(T n, const vector<T>& bases) {
  if (n < 2) {
    return false;
  }
  vector<T> small_primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
  for (const T& x : small_primes) {
    if (n % x == 0) {
      return n == x;
    }
  }
  if (n < 31 * 31) {
    return true;
  }
  int s = 0;
  T d = n - 1;
  while ((d & 1) == 0) {
    d >>= 1;
    s++;
  }
  FactorizerVarMod<T>::value = n;
  for (const T& a : bases) {
    if (a % n == 0) {
      continue;
    }
    Modular<FactorizerVarMod<T>> cur = a;
    cur = power(cur, d);
    if (cur == 1) {
      continue;
    }
    bool witness = true;
    for (int r = 0; r < s; r++) {
      if (cur == n - 1) {
        witness = false;
        break;
      }
      cur *= cur;
    }
    if (witness) {
      return false;
    }
  }
  return true;
}

bool IsPrime(int64_t n) {
```

```
  return IsPrime(n, {2, 325, 9375, 28178, 450775, 9780504, 1795265022});
}

bool IsPrime(int32_t n) {
  return IsPrime(n, {2, 7, 61});
}

// but if you really need uint64_t version...
/*
bool IsPrime(uint64_t n) {
  if (n < 2) {
    return false;
  }
  vector<uint32_t> small_primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
  for (uint32_t x : small_primes) {
    if (n == x) {
      return true;
    }
    if (n % x == 0) {
      return false;
    }
  }
  if (n < 31 * 31) {
    return true;
  }
  uint32_t s = __builtin_ctzll(n - 1);
  uint64_t d = (n - 1) >> s;
  function<bool(uint64_t)> witness = [&n, &s, &d](uint64_t a) {
    uint64_t cur = 1, p = d;
    while (p > 0) {
      if (p & 1) {
        cur = (__uint128_t) cur * a % n;
      }
      a = (__uint128_t) a * a % n;
      p >>= 1;
    }
    if (cur == 1) {
      return false;
    }
    for (uint32_t r = 0; r < s; r++) {
      if (cur == n - 1) {
        return false;
      }
      cur = (__uint128_t) cur * cur % n;
    }
    return true;
  };
  vector<uint64_t> bases_64bit = {2, 325, 9375, 28178, 450775, 9780504,
      1795265022};
  for (uint64_t a : bases_64bit) {
    if (a % n == 0) {
      return true;
    }
    if (witness(a)) {
      return false;
    }
  }
  return true;
}
*/

vector<int> least = {0, 1};
vector<int> primes;
int precalculated = 1;

void RunLinearSieve(int n) {
  n = max(n, 1);
  least.assign(n + 1, 0);
  primes.clear();
  for (int i = 2; i <= n; i++) {
    if (least[i] == 0) {
      least[i] = i;
      primes.push_back(i);
    }
    for (int x : primes) {
      if (x > least[i] || i * x > n) {
        break;
      }
      least[i * x] = x;
    }
  }
  precalculated = n;
}

void RunSlowSieve(int n) {
  n = max(n, 1);
```

```
  least.assign(n + 1, 0);
  for (int i = 2; i * i <= n; i++) {
    if (least[i] == 0) {
      for (int j = i * i; j <= n; j += i) {
        if (least[j] == 0) {
          least[j] = i;
        }
      }
    }
  }
  primes.clear();
  for (int i = 2; i <= n; i++) {
    if (least[i] == 0) {
      least[i] = i;
      primes.push_back(i);
    }
  }
  precalculated = n;
}

void RunSieve(int n) {
  RunLinearSieve(n);
}

template <typename T>
vector<pair<T, int>> MergeFactors(const vector<pair<T, int>>& a, const
    vector<pair<T, int>>& b) {
  vector<pair<T, int>> c;
  int i = 0;
  int j = 0;
  while (i < (int) a.size() || j < (int) b.size()) {
    if (i < (int) a.size() && j < (int) b.size() && a[i].first == b[j].first
        ) {
      c.emplace_back(a[i].first, a[i].second + b[j].second);
      ++i;
      ++j;
      continue;
    }
    if (j == (int) b.size() || (i < (int) a.size() && a[i].first < b[j].
        first)) {
      c.push_back(a[i++]);
    } else {
      c.push_back(b[j++]);
    }
  }
  return c;
}

template <typename T>
vector<pair<T, int>> RhoC(const T& n, const T& c) {
  if (n <= 1) {
    return {};
  }
  if ((n & 1) == 0) {
    return MergeFactors({{2, 1}}, RhoC(n / 2, c));
  }
  if (IsPrime(n)) {
    return {{n, 1}};
  }
  FactorizerVarMod<T>::value = n;
  Modular<FactorizerVarMod<T>> x = 2;
  Modular<FactorizerVarMod<T>> saved = 2;
  T power = 1;
  T lam = 1;
  while (true) {
    x = x * x + c;
    T g = __gcd((x - saved)(), n);
    if (g != 1) {
      return MergeFactors(RhoC(g, c + 1), RhoC(n / g, c + 1));
    }
    if (power == lam) {
      saved = x;
      power <<= 1;
      lam = 0;
    }
    lam++;
  }
  return {};
}

template <typename T>
vector<pair<T, int>> Rho(const T& n) {
  return RhoC(n, static_cast<T>(1));
}

template <typename T>
```

```
vector<pair<T, int>> Factorize(T x) {
  if (x <= 1) {
    return {};
  }
  if (x <= precalculated) {
    vector<pair<T, int>> ret;
    while (x > 1) {
      if (!ret.empty() && ret.back().first == least[x]) {
        ret.back().second++;
      } else {
        ret.emplace_back(least[x], 1);
      }
      x /= least[x];
    }
    return ret;
  }
  if (x <= static_cast<int64_t>(precalculated) * precalculated) {
    vector<pair<T, int>> ret;
    if (!IsPrime(x)) {
      for (T i : primes) {
        T t = x / i;
        if (i > t) {
          break;
        }
        if (x == t * i) {
          int cnt = 0;
          while (x % i == 0) {
            x /= i;
            cnt++;
          }
          ret.emplace_back(i, cnt);
          if (IsPrime(x)) {
            break;
          }
        }
      }
    }
    if (x > 1) {
      ret.emplace_back(x, 1);
    }
    return ret;
  }
  return Rho(x);
}

template <typename T>
vector<T> BuildDivisorsFromFactors(const vector<pair<T, int>>& factors) {
  vector<T> divisors = {1};
  for (auto& p : factors) {
    int sz = (int) divisors.size();
    for (int i = 0; i < sz; i++) {
      T cur = divisors[i];
      for (int j = 0; j < p.second; j++) {
        cur *= p.first;
        divisors.push_back(cur);
      }
    }
  }
  sort(divisors.begin(), divisors.end());
  return divisors;
}

}  // namespace factorizer
```

5.4 fft.cpp

```
// make it understandable one day...
namespace fft {

typedef double dbl;

struct num {
  dbl x, y;
  num() { x = y = 0; }
  num(dbl x_, dbl y_) : x(x_), y(y_) {}
};

inline num operator+(num a, num b) { return num(a.x + b.x, a.y + b.y); }
inline num operator-(num a, num b) { return num(a.x - b.x, a.y - b.y); }
inline num operator*(num a, num b) { return num(a.x * b.x - a.y * b.y, a.x *
     b.y + a.y * b.x); }
inline num conj(num a) { return num(a.x, -a.y); }

int base = 1;
vector<num> roots = {{0, 0}, {1, 0}};
vector<int> rev = {0, 1};
```

```
20
21  const dbl PI = static_cast<dbl>(acosl(-1.0));
22
23  void ensure_base(int nbase) {
24    if (nbase <= base) {
25      return;
26    }
27    rev.resize(1 << nbase);
28    for (int i = 0; i < (1 << nbase); i++) {
29      rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
30    }
31    roots.resize(1 << nbase);
32    while (base < nbase) {
33      dbl angle = 2 * PI / (1 << (base + 1));
34  //      num z(cos(angle), sin(angle));
35      for (int i = 1 << (base - 1); i < (1 << base); i++) {
36        roots[i << 1] = roots[i];
37  //        roots[(i << 1) + 1] = roots[i] * z;
38        dbl angle_i = angle * (2 * i + 1 - (1 << base));
39        roots[(i << 1) + 1] = num(cos(angle_i), sin(angle_i));
40      }
41      base++;
42    }
43  }
44
45  void fft(vector<num>& a, int n = -1) {
46    if (n == -1) {
47      n = (int) a.size();
48    }
49    assert((n & (n - 1)) == 0);
50    int zeros = __builtin_ctz(n);
51    ensure_base(zeros);
52    int shift = base - zeros;
53    for (int i = 0; i < n; i++) {
54      if (i < (rev[i] >> shift)) {
55        swap(a[i], a[rev[i] >> shift]);
56      }
57    }
58    for (int k = 1; k < n; k <<= 1) {
59      for (int i = 0; i < n; i += 2 * k) {
60        for (int j = 0; j < k; j++) {
61          num z = a[i + j + k] * roots[j + k];
62          a[i + j + k] = a[i + j] - z;
63          a[i + j] = a[i + j] + z;
64        }
65      }
66    }
67  }
68
69  vector<num> fa, fb;
70
71  vector<int64_t> square(const vector<int>& a) {
72    if (a.empty()) {
73      return {};
74    }
75    int need = (int) a.size() + (int) a.size() - 1;
76    int nbase = 1;
77    while ((1 << nbase) < need) nbase++;
78    ensure_base(nbase);
79    int sz = 1 << nbase;
80    if ((sz >> 1) > (int) fa.size()) {
81      fa.resize(sz >> 1);
82    }
83    for (int i = 0; i < (sz >> 1); i++) {
84      int x = (2 * i < (int) a.size() ? a[2 * i] : 0);
85      int y = (2 * i + 1 < (int) a.size() ? a[2 * i + 1] : 0);
86      fa[i] = num(x, y);
87    }
88    fft(fa, sz >> 1);
89    num r(1.0 / (sz >> 1), 0.0);
90    for (int i = 0; i <= (sz >> 2); i++) {
91      int j = ((sz >> 1) - i) & ((sz >> 1) - 1);
92      num fe = (fa[i] + conj(fa[j])) * num(0.5, 0);
93      num fo = (fa[i] - conj(fa[j])) * num(0, -0.5);
94      num aux = fe * fe + fo * fo * roots[(sz >> 1) + i] * roots[(sz >> 1) + i
            ];
95      num tmp = fe * fo;
96      fa[i] = r * (conj(aux) + num(0, 2) * conj(tmp));
97      fa[j] = r * (aux + num(0, 2) * tmp);
98    }
99    fft(fa, sz >> 1);
100   vector<int64_t> res(need);
101   for (int i = 0; i < need; i++) {
102     res[i] = llround(i % 2 == 0 ? fa[i >> 1].x : fa[i >> 1].y);
103   }
104   return res;
```

```
}

vector<int64_t> multiply(const vector<int>& a, const vector<int>& b) {
  if (a.empty() || b.empty()) {
    return {};
  }
  if (a == b) {
    return square(a);
  }
  int need = (int) a.size() + (int) b.size() - 1;
  int nbase = 1;
  while ((1 << nbase) < need) nbase++;
  ensure_base(nbase);
  int sz = 1 << nbase;
  if (sz > (int) fa.size()) {
    fa.resize(sz);
  }
  for (int i = 0; i < sz; i++) {
    int x = (i < (int) a.size() ? a[i] : 0);
    int y = (i < (int) b.size() ? b[i] : 0);
    fa[i] = num(x, y);
  }
  fft(fa, sz);
  num r(0, -0.25 / (sz >> 1));
  for (int i = 0; i <= (sz >> 1); i++) {
    int j = (sz - i) & (sz - 1);
    num z = (fa[j] * fa[j] - conj(fa[i] * fa[i])) * r;
    fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[j])) * r;
    fa[i] = z;
  }
  for (int i = 0; i < (sz >> 1); i++) {
    num A0 = (fa[i] + fa[i + (sz >> 1)]) * num(0.5, 0);
    num A1 = (fa[i] - fa[i + (sz >> 1)]) * num(0.5, 0) * roots[(sz >> 1) + i
        ];
    fa[i] = A0 + A1 * num(0, 1);
  }
  fft(fa, sz >> 1);
  vector<int64_t> res(need);
  for (int i = 0; i < need; i++) {
    res[i] = llround(i % 2 == 0 ? fa[i >> 1].x : fa[i >> 1].y);
  }
  return res;
}


vector<int> multiply_mod(const vector<int>& a, const vector<int>& b, int m)
    {
  if (a.empty() || b.empty()) {
    return {};
  }
  int eq = (a.size() == b.size() && a == b);
  int need = (int) a.size() + (int) b.size() - 1;
  int nbase = 0;
  while ((1 << nbase) < need) nbase++;
  ensure_base(nbase);
  int sz = 1 << nbase;
  if (sz > (int) fa.size()) {
    fa.resize(sz);
  }
  for (int i = 0; i < (int) a.size(); i++) {
    int x = (a[i] % m + m) % m;
    fa[i] = num(x & ((1 << 15) - 1), x >> 15);
  }
  fill(fa.begin() + a.size(), fa.begin() + sz, num {0, 0});
  fft(fa, sz);
  if (sz > (int) fb.size()) {
    fb.resize(sz);
  }
  if (eq) {
    copy(fa.begin(), fa.begin() + sz, fb.begin());
  } else {
    for (int i = 0; i < (int) b.size(); i++) {
      int x = (b[i] % m + m) % m;
      fb[i] = num(x & ((1 << 15) - 1), x >> 15);
    }
    fill(fb.begin() + b.size(), fb.begin() + sz, num {0, 0});
    fft(fb, sz);
  }
  dbl ratio = 0.25 / sz;
  num r2(0, -1);
  num r3(ratio, 0);
  num r4(0, -ratio);
  num r5(0, 1);
  for (int i = 0; i <= (sz >> 1); i++) {
    int j = (sz - i) & (sz - 1);
    num a1 = (fa[i] + conj(fa[j]));
    num a2 = (fa[i] - conj(fa[j])) * r2;
```

```
    num b1 = (fb[i] + conj(fb[j])) * r3;
    num b2 = (fb[i] - conj(fb[j])) * r4;
    if (i != j) {
      num c1 = (fa[j] + conj(fa[i]));
      num c2 = (fa[j] - conj(fa[i])) * r2;
      num d1 = (fb[j] + conj(fb[i])) * r3;
      num d2 = (fb[j] - conj(fb[i])) * r4;
      fa[i] = c1 * d1 + c2 * d2 * r5;
      fb[i] = c1 * d2 + c2 * d1;
    }
    fa[j] = a1 * b1 + a2 * b2 * r5;
    fb[j] = a1 * b2 + a2 * b1;
  }
  fft(fa, sz);
  fft(fb, sz);
  vector<int> res(need);
  for (int i = 0; i < need; i++) {
    int64_t aa = llround(fa[i].x);
    int64_t bb = llround(fb[i].x);
    int64_t cc = llround(fa[i].y);
    res[i] = static_cast<int>((aa + ((bb % m) << 15) + ((cc % m) << 30)) % m
        );
  }
  return res;
}

}  // namespace fft

template <typename T>
typename enable_if<is_same<typename Modular<T>::Type, int>::value, vector<
    Modular<T>>>::type operator*(
    const vector<Modular<T>>& a,
    const vector<Modular<T>>& b) {
  if (a.empty() || b.empty()) {
    return {};
  }
  if (min(a.size(), b.size()) < 150) {
    vector<Modular<T>> c(a.size() + b.size() - 1, 0);
    for (int i = 0; i < (int) a.size(); i++) {
      for (int j = 0; j < (int) b.size(); j++) {
        c[i + j] += a[i] * b[j];
      }
    }
    return c;
  }
  vector<int> a_mul(a.size());
  for (int i = 0; i < (int) a.size(); i++) {
    a_mul[i] = static_cast<int>(a[i]);
  }
  vector<int> b_mul(b.size());
  for (int i = 0; i < (int) b.size(); i++) {
    b_mul[i] = static_cast<int>(b[i]);
  }
  vector<int> c_mul = fft::multiply_mod(a_mul, b_mul, T::value);
  vector<Modular<T>> c(c_mul.size());
  for (int i = 0; i < (int) c.size(); i++) {
    c[i] = c_mul[i];
  }
  return c;
}

template <typename T>
typename enable_if<is_same<typename Modular<T>::Type, int>::value, vector<
    Modular<T>>>::type& operator*=(
    vector<Modular<T>>& a,
    const vector<Modular<T>>& b) {
  return a = a * b;
}
```

5.5 fwht.cpp

```
namespace fwht {

template<typename T>
void hadamard(vector<T> &a) {
  int n = a.size();
  for (int k = 1; k < n; k <<= 1) {
    for (int i = 0; i < n; i += 2 * k) {
      for (int j = 0; j < k; j++) {
        T x = a[i + j];
        T y = a[i + j + k];
        a[i + j] = x + y;
        a[i + j + k] = x - y;
      }
    }
  }
```

```
}

template<typename T>
vector<T> multiply(vector<T> a, vector<T> b) {
  int eq = (a == b);
  int n = 1;
  while (n < (int) max(a.size(), b.size())) {
    n <<= 1;
  }
  a.resize(n);
  b.resize(n);
  hadamard(a);
  if (eq) b = a; else hadamard(b);
  for (int i = 0; i < n; i++) {
    a[i] *= b[i];
  }
  hadamard(a);
  T q = 1 / static_cast<T>(n);
  for (int i = 0; i < n; i++) {
    a[i] *= q;
  }
  return a;
}

}  // namespace fwht
```

5.6 gauss.cpp

```
const double eps = 1e-9;

bool IsZero(double v) {
  return abs(v) < 1e-9;
}

enum GAUSS_MODE {
  DEGREE, ABS
};

template <typename T>
void GaussianElimination(vector<vector<T>>& a, int limit, GAUSS_MODE mode =
    DEGREE) {
  if (a.empty() || a[0].empty()) {
    return;
  }
  int h = static_cast<int>(a.size());
  int w = static_cast<int>(a[0].size());
  for (int i = 0; i < h; i++) {
    assert(w == static_cast<int>(a[i].size()));
  }
  assert(limit <= w);
  vector<int> deg(h);
  for (int i = 0; i < h; i++) {
    for (int j = 0; j < w; j++) {
      deg[i] += !IsZero(a[i][j]);
    }
  }
  int r = 0;
  for (int c = 0; c < limit; c++) {
    int id = -1;
    for (int i = r; i < h; i++) {
      if (!IsZero(a[i][c]) && (id == -1 || (mode == DEGREE && deg[i] < deg[
          id]) || (mode == ABS && abs(a[id][c]) < abs(a[i][c])))) {
        id = i;
      }
    }
    if (id == -1) {
      continue;
    }
    if (id > r) {
      swap(a[r], a[id]);
      swap(deg[r], deg[id]);
      for (int j = c; j < w; j++) {
        a[id][j] = -a[id][j];
      }
    }
    vector<int> nonzero;
    for (int j = c; j < w; j++) {
      if (!IsZero(a[r][j])) {
        nonzero.push_back(j);
      }
    }
    T inv_a = 1 / a[r][c];
    for (int i = r + 1; i < h; i++) {
      if (IsZero(a[i][c])) {
        continue;
      }
```

```
      T coeff = -a[i][c] * inv_a;
      for (int j : nonzero) {
        if (!IsZero(a[i][j])) deg[i]--;
        a[i][j] += coeff * a[r][j];
        if (!IsZero(a[i][j])) deg[i]++;
      }
    }
    ++r;
  }
  for (r = h - 1; r >= 0; r--) {
    for (int c = 0; c < limit; c++) {
      if (!IsZero(a[r][c])) {
        T inv_a = 1 / a[r][c];
        for (int i = r - 1; i >= 0; i--) {
          if (IsZero(a[i][c])) {
            continue;
          }
          T coeff = -a[i][c] * inv_a;
          for (int j = c; j < w; j++) {
            a[i][j] += coeff * a[r][j];
          }
        }
        break;
      }
    }
  }
}

template <typename T>
T Determinant(vector<vector<T>>/*&*/ a) {
  if (a.empty()) {
    return T{1};
  }
  assert(a.size() == a[0].size());
  GaussianElimination(a, static_cast<int>(a[0].size()));
  T d{1};
  for (int i = 0; i < a.h; i++) {
    d *= a[i][i];
  }
  return d;
}

template <typename T>
int Rank(vector<vector<T>>/*&*/ a) {
  if (a.empty()) {
    return 0;
  }
  GaussianElimination(a, static_cast<int>(a[0].size()));
  int rank = 0;
  for (int i = 0; i < static_cast<int>(a.size()); i++) {
    for (int j = 0; j < static_cast<int>(a[i].size()); j++) {
      if (!IsZero(a[i][j])) {
        ++rank;
        break;
      }
    }
  }
  return rank;
}

template <typename T>
vector<T> SolveLinearSystem(vector<vector<T>>/*&*/ a, const vector<T>& b,
    int w) {
  int h = static_cast<int>(a.size());
  assert(h == static_cast<int>(b.size()));
  if (h > 0) {
    assert(w == static_cast<int>(a[0].size()));
  }
  for (int i = 0; i < h; i++) {
    a[i].push_back(b[i]);
  }
  GaussianElimination(a, w);
  vector<T> x(w, 0);
  for (int i = 0; i < h; i++) {
    for (int j = 0; j < w; j++) {
      if (!IsZero(a[i][j])) {
        x[j] = a[i][w] / a[i][j];
        break;
      }
    }
  }
  return x;
}

template <typename T>
vector<vector<T>> Inverse(vector<vector<T>>/*&*/ a) {
```

```
  if (a.empty()) {
    return a;
  }
  int h = static_cast<int>(a.size());
  for (int i = 0; i < h; i++) {
    assert(h == static_cast<int>(a[i].size()));
  }
  for (int i = 0; i < h; i++) {
    a[i].resize(2 * h);
    a[i][i + h] = 1;
  }
  GaussianElimination(a, h);
  for (int i = 0; i < h; i++) {
    if (IsZero(a[i][i])) {
      return {{}};
    }
  }
  vector<vector<T>> b(h);
  for (int i = 0; i < h; i++) {
    b[i] = vector<T>(a[i].begin() + h, a[i].end());
    T coeff = 1 / a[i][i];
    for (int j = 0; j < h; j++) {
      b[i][j] *= coeff;
    }
  }
  return b;
}
```

5.7 matrix.cpp

```
template <typename T, size_t N, size_t M, size_t K>
array<array<T, K>, N> operator*(const array<array<T, M>, N>& a, const array<
    array<T, K>, M>& b) {
  array<array<T, K>, N> c;
  for (size_t i = 0; i < N; i++) {
    for (size_t j = 0; j < K; j++) {
      c[i][j] = 0;
      for (size_t k = 0; k < M; k++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
  return c;
}

template <typename T>
vector<vector<T>> operator*(const vector<vector<T>>& a, const vector<vector<
    T>>& b) {
  if (a.empty() || b.empty()) {
    return {{}};
  }
  vector<vector<T>> c(a.size(), vector<T>(b[0].size()));
  for (int i = 0; i < static_cast<int>(c.size()); i++) {
    for (int j = 0; j < static_cast<int>(c[0].size()); j++) {
      c[i][j] = 0;
      for (int k = 0; k < static_cast<int>(b.size()); k++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
  return c;
}

template <typename T>
vector<vector<T>>& operator*=(vector<vector<T>>& a, const vector<vector<T>>&
     b) {
  return a = a * b;
}

template <typename T, typename U>
vector<vector<T>> power(const vector<vector<T>>& a, const U& b) {
  assert(b >= 0);
  vector<U> binary;
  U bb = b;
  while (bb > 0) {
    binary.push_back(bb & 1);
    bb >>= 1;
  }
  vector<vector<T>> res(a.size(), vector<T>(a.size()));
  for (int i = 0; i < static_cast<int>(a.size()); i++) {
    res[i][i] = 1;
  }
  for (int j = (int) binary.size() - 1; j >= 0; j--) {
    res *= res;
    if (binary[j] == 1) {
      res *= a;
```

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

5.8 mint.cpp

```
template <typename T>
T inverse(T a, T m) {
  T u = 0, v = 1;
  while (a != 0) {
    T t = m / a;
    m -= t * a; swap(a, m);
    u -= t * v; swap(u, v);
  }
  assert(m == 1);
  return u;
}

template <typename T>
class Modular {
 public:
  using Type = typename decay<decltype(T::value)>::type;

  constexpr Modular() : value() {}
  template <typename U>
  Modular(const U& x) {
    value = normalize(x);
  }

  template <typename U>
  static Type normalize(const U& x) {
    Type v;
    if (-mod() <= x && x < mod()) v = static_cast<Type>(x);
    else v = static_cast<Type>(x % mod());
    if (v < 0) v += mod();
    return v;
  }

  const Type& operator()() const { return value; }
  template <typename U>
  explicit operator U() const { return static_cast<U>(value); }
  constexpr static Type mod() { return T::value; }

  Modular& operator+=(const Modular& other) { if ((value += other.value) >= mod()) value -= mod(); return *this; }
  Modular& operator-=(const Modular& other) { if ((value -= other.value) < 0) value += mod(); return *this; }
  template <typename U> Modular& operator+=(const U& other) { return *this += Modular(other); }
  template <typename U> Modular& operator-=(const U& other) { return *this -= Modular(other); }
  Modular& operator++() { return *this += 1; }
  Modular& operator--() { return *this -= 1; }
  Modular operator++(int) { Modular result(*this); *this += 1; return result; }
  Modular operator--(int) { Modular result(*this); *this -= 1; return result; }
  Modular operator-() const { return Modular(-value); }

  template <typename U = T>
  typename enable_if<is_same<typename Modular<U>::Type, int>::value, Modular>::type& operator*=(const Modular& rhs) {
    value = normalize(static_cast<int64_t>(value) * static_cast<int64_t>(rhs.value));
    return *this;
  }
  template <typename U = T>
  typename enable_if<is_same<typename Modular<U>::Type, long long>::value, Modular>::type& operator*=(const Modular& rhs) {
    long long q = static_cast<long long>(static_cast<long double>(value) * rhs.value / mod());
    value = normalize(value * rhs.value - q * mod());
    return *this;
  }
  template <typename U = T>
  typename enable_if<!is_integral<typename Modular<U>::Type>::value, Modular>::type& operator*=(const Modular& rhs) {
    value = normalize(value * rhs.value);
    return *this;
  }

  Modular& operator/=(const Modular& other) { return *this *= Modular(inverse(other.value, mod())); }

  friend const Type& abs(const Modular& x) { return x.value; }
```

```
68
69    template <typename U>
70    friend bool operator==(const Modular<U>& lhs, const Modular<U>& rhs);
71
72    template <typename U>
73    friend bool operator<(const Modular<U>& lhs, const Modular<U>& rhs);
74
75    template <typename V, typename U>
76    friend V& operator>>(V& stream, Modular<U>& number);
77
78   private:
79    Type value;
80  };
81
82  template <typename T> bool operator==(const Modular<T>& lhs, const Modular<T
        >& rhs) { return lhs.value == rhs.value; }
83  template <typename T, typename U> bool operator==(const Modular<T>& lhs, U
        rhs) { return lhs == Modular<T>(rhs); }
84  template <typename T, typename U> bool operator==(U lhs, const Modular<T>&
        rhs) { return Modular<T>(lhs) == rhs; }
85
86  template <typename T> bool operator!=(const Modular<T>& lhs, const Modular<T
        >& rhs) { return !(lhs == rhs); }
87  template <typename T, typename U> bool operator!=(const Modular<T>& lhs, U
        rhs) { return !(lhs == rhs); }
88  template <typename T, typename U> bool operator!=(U lhs, const Modular<T>&
        rhs) { return !(lhs == rhs); }
89
90  template <typename T> bool operator<(const Modular<T>& lhs, const Modular<T
        >& rhs) { return lhs.value < rhs.value; }
91
92  template <typename T> Modular<T> operator+(const Modular<T>& lhs, const
        Modular<T>& rhs) { return Modular<T>(lhs) += rhs; }
93  template <typename T, typename U> Modular<T> operator+(const Modular<T>& lhs
        , U rhs) { return Modular<T>(lhs) += rhs; }
94  template <typename T, typename U> Modular<T> operator+(U lhs, const Modular<
        T>& rhs) { return Modular<T>(lhs) += rhs; }
95
96  template <typename T> Modular<T> operator-(const Modular<T>& lhs, const
        Modular<T>& rhs) { return Modular<T>(lhs) -= rhs; }
97  template <typename T, typename U> Modular<T> operator-(const Modular<T>& lhs
        , U rhs) { return Modular<T>(lhs) -= rhs; }
98  template <typename T, typename U> Modular<T> operator-(U lhs, const Modular<
        T>& rhs) { return Modular<T>(lhs) -= rhs; }
99
100 template <typename T> Modular<T> operator*(const Modular<T>& lhs, const
        Modular<T>& rhs) { return Modular<T>(lhs) *= rhs; }
101 template <typename T, typename U> Modular<T> operator*(const Modular<T>& lhs
        , U rhs) { return Modular<T>(lhs) *= rhs; }
102 template <typename T, typename U> Modular<T> operator*(U lhs, const Modular<
        T>& rhs) { return Modular<T>(lhs) *= rhs; }
103
104 template <typename T> Modular<T> operator/(const Modular<T>& lhs, const
        Modular<T>& rhs) { return Modular<T>(lhs) /= rhs; }
105 template <typename T, typename U> Modular<T> operator/(const Modular<T>& lhs
        , U rhs) { return Modular<T>(lhs) /= rhs; }
106 template <typename T, typename U> Modular<T> operator/(U lhs, const Modular<
        T>& rhs) { return Modular<T>(lhs) /= rhs; }
107
108 template<typename T, typename U>
109 Modular<T> power(const Modular<T>& a, const U& b) {
110   assert(b >= 0);
111   Modular<T> x = a, res = 1;
112   U p = b;
113   while (p > 0) {
114     if (p & 1) res *= x;
115     x *= x;
116     p >>= 1;
117   }
118   return res;
119 }
120
121 template <typename T>
122 bool IsZero(const Modular<T>& number) {
123   return number() == 0;
124 }
125
126 template <typename T>
127 string to_string(const Modular<T>& number) {
128   return to_string(number());
129 }
130
131 // U == std::ostream? but done this way because of fastoutput
132 template <typename U, typename T>
133 U& operator<<(U& stream, const Modular<T>& number) {
134   return stream << number();
```

```
}

// U == std::istream? but done this way because of fastinput
template <typename U, typename T>
U& operator>>(U& stream, Modular<T>& number) {
  typename common_type<typename Modular<T>::Type, long long>::type x;
  stream >> x;
  number.value = Modular<T>::normalize(x);
  return stream;
}

// using ModType = int;

// struct VarMod { static ModType value; };
// ModType VarMod::value;
// ModType& md = VarMod::value;
// using Mint = Modular<VarMod>;

constexpr int md = ${0};
using Mint = Modular<std::integral_constant<decay<decltype(md)>::type, md>>;

// vector<Mint> fact(1, 1);
// vector<Mint> inv_fact(1, 1);

// Mint C(int n, int k) {
//   if (k < 0 || k > n) {
//     return 0;
//   }
//   while ((int) fact.size() < n + 1) {
//     fact.push_back(fact.back() * (int) fact.size());
//     inv_fact.push_back(1 / fact.back());
//   }
//   return fact[n] * inv_fact[k] * inv_fact[n - k];
// }
```

5.9 ntt.cpp

```
template <typename T>
class NTT {
 public:
  using Type = typename decay<decltype(T::value)>::type;

  static Type md;
  static Modular<T> root;
  static int base;
  static int max_base;
  static vector<Modular<T>> roots;
  static vector<int> rev;

  static void clear() {
    root = 0;
    base = 0;
    max_base = 0;
    roots.clear();
    rev.clear();
  }

  static void init() {
    md = T::value;
    assert(md >= 3 && md % 2 == 1);
    auto tmp = md - 1;
    max_base = 0;
    while (tmp % 2 == 0) {
      tmp /= 2;
      max_base++;
    }
    root = 2;
    while (power(root, (md - 1) >> 1) == 1) {
      root++;
    }
    assert(power(root, md - 1) == 1);
    root = power(root, (md - 1) >> max_base);
    base = 1;
    rev = {0, 1};
    roots = {0, 1};
  }

  static void ensure_base(int nbase) {
    if (md != T::value) {
      clear();
    }
    if (roots.empty()) {
      init();
    }
    if (nbase <= base) {
      return;
```

```
    }
    assert(nbase <= max_base);
    rev.resize(1 << nbase);
    for (int i = 0; i < (1 << nbase); i++) {
      rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
    }
    roots.resize(1 << nbase);
    while (base < nbase) {
      Modular<T> z = power(root, 1 << (max_base - 1 - base));
      for (int i = 1 << (base - 1); i < (1 << base); i++) {
        roots[i << 1] = roots[i];
        roots[(i << 1) + 1] = roots[i] * z;
      }
      base++;
    }
  }

  static void fft(vector<Modular<T>> &a) {
    int n = (int) a.size();
    assert((n & (n - 1)) == 0);
    int zeros = __builtin_ctz(n);
    ensure_base(zeros);
    int shift = base - zeros;
    for (int i = 0; i < n; i++) {
      if (i < (rev[i] >> shift)) {
        swap(a[i], a[rev[i] >> shift]);
      }
    }
    for (int k = 1; k < n; k <<= 1) {
      for (int i = 0; i < n; i += 2 * k) {
        for (int j = 0; j < k; j++) {
          Modular<T> x = a[i + j];
          Modular<T> y = a[i + j + k] * roots[j + k];
          a[i + j] = x + y;
          a[i + j + k] = x - y;
        }
      }
    }
  }

  static vector<Modular<T>> multiply(vector<Modular<T>> a, vector<Modular<T
      >> b) {
    if (a.empty() || b.empty()) {
      return {};
    }
    int eq = (a == b);
    int need = (int) a.size() + (int) b.size() - 1;
    int nbase = 0;
    while ((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    a.resize(sz);
    b.resize(sz);
    fft(a);
    if (eq) b = a; else fft(b);
    Modular<T> inv_sz = 1 / static_cast<Modular<T>>(sz);
    for (int i = 0; i < sz; i++) {
      a[i] *= b[i] * inv_sz;
    }
    reverse(a.begin() + 1, a.end());
    fft(a);
    a.resize(need);
    return a;
  }
};

template <typename T> typename NTT<T>::Type NTT<T>::md;
template <typename T> Modular<T> NTT<T>::root;
template <typename T> int NTT<T>::base;
template <typename T> int NTT<T>::max_base;
template <typename T> vector<Modular<T>> NTT<T>::roots;
template <typename T> vector<int> NTT<T>::rev;

template <typename T>
vector<Modular<T>> inverse(const vector<Modular<T>>& a) {
  assert(!a.empty());
  int n = (int) a.size();
  vector<Modular<T>> b = {1 / a[0]};
  while ((int) b.size() < n) {
    vector<Modular<T>> x(a.begin(), a.begin() + min(a.size(), b.size() << 1)
        );
    x.resize(b.size() << 1);
    b.resize(b.size() << 1);
    vector<Modular<T>> c = b;
    NTT<T>::fft(c);
    NTT<T>::fft(x);
```

```
    Modular<T> inv = 1 / static_cast<Modular<T>>((int) x.size());
    for (int i = 0; i < (int) x.size(); i++) {
      x[i] *= c[i] * inv;
    }
    reverse(x.begin() + 1, x.end());
    NTT<T>::fft(x);
    rotate(x.begin(), x.begin() + (x.size() >> 1), x.end());
    fill(x.begin() + (x.size() >> 1), x.end(), 0);
    NTT<T>::fft(x);
    for (int i = 0; i < (int) x.size(); i++) {
      x[i] *= c[i] * inv;
    }
    reverse(x.begin() + 1, x.end());
    NTT<T>::fft(x);
    for (int i = 0; i < ((int) x.size() >> 1); i++) {
      b[i + ((int) x.size() >> 1)] = -x[i];
    }
  }
  b.resize(n);
  return b;
}

template <typename T>
vector<Modular<T>> inverse_old(vector<Modular<T>> a) {
  assert(!a.empty());
  int n = (int) a.size();
  if (n == 1) {
    return {1 / a[0]};
  }
  int m = (n + 1) >> 1;
  vector<Modular<T>> b = inverse(vector<Modular<T>>(a.begin(), a.begin() + m
      ));
  int need = n << 1;
  int nbase = 0;
  while ((1 << nbase) < need) {
    ++nbase;
  }
  NTT<T>::ensure_base(nbase);
  int size = 1 << nbase;
  a.resize(size);
  b.resize(size);
  NTT<T>::fft(a);
  NTT<T>::fft(b);
  Modular<T> inv = 1 / static_cast<Modular<T>>(size);
  for (int i = 0; i < size; ++i) {
    a[i] = (2 - a[i] * b[i]) * b[i] * inv;
  }
  reverse(a.begin() + 1, a.end());
  NTT<T>::fft(a);
  a.resize(n);
  return a;
}

template <typename T>
vector<Modular<T>> operator*(const vector<Modular<T>>& a, const vector<
    Modular<T>>& b) {
  if (a.empty() || b.empty()) {
    return {};
  }
  if (min(a.size(), b.size()) < 150) {
    vector<Modular<T>> c(a.size() + b.size() - 1, 0);
    for (int i = 0; i < (int) a.size(); i++) {
      for (int j = 0; j < (int) b.size(); j++) {
        c[i + j] += a[i] * b[j];
      }
    }
    return c;
  }
  return NTT<T>::multiply(a, b);
}

template <typename T>
vector<Modular<T>>& operator*=(vector<Modular<T>>& a, const vector<Modular<T
    >>& b) {
  return a = a * b;
}
```

5.10 poly.cpp

```
template <typename T>
vector<T>& operator+=(vector<T>& a, const vector<T>& b) {
  if (a.size() < b.size()) {
    a.resize(b.size());
  }
  for (int i = 0; i < (int) b.size(); i++) {
    a[i] += b[i];
```

```
  }
  return a;
}

template <typename T>
vector<T> operator+(const vector<T>& a, const vector<T>& b) {
  vector<T> c = a;
  return c += b;
}

template <typename T>
vector<T>& operator-=(vector<T>& a, const vector<T>& b) {
  if (a.size() < b.size()) {
    a.resize(b.size());
  }
  for (int i = 0; i < (int) b.size(); i++) {
    a[i] -= b[i];
  }
  return a;
}

template <typename T>
vector<T> operator-(const vector<T>& a, const vector<T>& b) {
  vector<T> c = a;
  return c -= b;
}

template <typename T>
vector<T> operator-(const vector<T>& a) {
  vector<T> c = a;
  for (int i = 0; i < (int) c.size(); i++) {
    c[i] = -c[i];
  }
  return c;
}

template <typename T>
vector<T> operator*(const vector<T>& a, const vector<T>& b) {
  if (a.empty() || b.empty()) {
    return {};
  }
  vector<T> c(a.size() + b.size() - 1, 0);
  for (int i = 0; i < (int) a.size(); i++) {
    for (int j = 0; j < (int) b.size(); j++) {
      c[i + j] += a[i] * b[j];
    }
  }
  return c;
}

template <typename T>
vector<T>& operator*=(vector<T>& a, const vector<T>& b) {
  return a = a * b;
}

template <typename T>
vector<T> inverse(const vector<T>& a) {
  assert(!a.empty());
  int n = (int) a.size();
  vector<T> b = {1 / a[0]};
  while ((int) b.size() < n) {
    vector<T> a_cut(a.begin(), a.begin() + min(a.size(), b.size() << 1));
    vector<T> x = b * b * a_cut;
    b.resize(b.size() << 1);
    for (int i = (int) b.size() >> 1; i < (int) min(x.size(), b.size()); i
        ++) {
      b[i] = -x[i];
    }
  }
  b.resize(n);
  return b;
}

template <typename T>
vector<T>& operator/=(vector<T>& a, const vector<T>& b) {
  int n = (int) a.size();
  int m = (int) b.size();
  if (n < m) {
    a.clear();
  } else {
    vector<T> d = b;
    reverse(a.begin(), a.end());
    reverse(d.begin(), d.end());
    d.resize(n - m + 1);
    a *= inverse(d);
    a.erase(a.begin() + n - m + 1, a.end());
```

```
    reverse(a.begin(), a.end());
  }
  return a;
}

template <typename T>
vector<T> operator/(const vector<T>& a, const vector<T>& b) {
  vector<T> c = a;
  return c /= b;
}

template <typename T>
vector<T>& operator%=(vector<T>& a, const vector<T>& b) {
  int n = (int) a.size();
  int m = (int) b.size();
  if (n >= m) {
    vector<T> c = (a / b) * b;
    a.resize(m - 1);
    for (int i = 0; i < m - 1; i++) {
      a[i] -= c[i];
    }
  }
  return a;
}

template <typename T>
vector<T> operator%(const vector<T>& a, const vector<T>& b) {
  vector<T> c = a;
  return c %= b;
}

template <typename T, typename U>
vector<T> power(const vector<T>& a, const U& b, const vector<T>& c) {
  assert(b >= 0);
  vector<U> binary;
  U bb = b;
  while (bb > 0) {
    binary.push_back(bb & 1);
    bb >>= 1;
  }
  vector<T> res = vector<T>{1} % c;
  for (int j = (int) binary.size() - 1; j >= 0; j--) {
    res = res * res % c;
    if (binary[j] == 1) {
      res = res * a % c;
    }
  }
  return res;
}

template <typename T>
vector<T> derivative(const vector<T>& a) {
  vector<T> c = a;
  for (int i = 0; i < (int) c.size(); i++) {
    c[i] *= i;
  }
  if (!c.empty()) {
    c.erase(c.begin());
  }
  return c;
}

template <typename T>
vector<T> primitive(const vector<T>& a) {
  vector<T> c = a;
  c.insert(c.begin(), 0);
  for (int i = 1; i < (int) c.size(); i++) {
    c[i] /= i;
  }
  return c;
}

template <typename T>
vector<T> logarithm(const vector<T>& a) {
  assert(!a.empty() && a[0] == 1);
  vector<T> u = primitive(derivative(a) * inverse(a));
  u.resize(a.size());
  return u;
}

template <typename T>
vector<T> exponent(const vector<T>& a) {
  assert(!a.empty() && a[0] == 0);
  int n = (int) a.size();
  vector<T> b = {1};
  while ((int) b.size() < n) {
```

```
    vector<T> x(a.begin(), a.begin() + min(a.size(), b.size() << 1));
    x[0] += 1;
    vector<T> old_b = b;
    b.resize(b.size() << 1);
    x -= logarithm(b);
    x *= old_b;
    for (int i = (int) b.size() >> 1; i < (int) min(x.size(), b.size()); i
        ++) {
      b[i] = x[i];
    }
  }
  b.resize(n);
  return b;
}

template <typename T>
vector<T> sqrt(const vector<T>& a) {
  assert(!a.empty() && a[0] == 1);
  int n = (int) a.size();
  vector<T> b = {1};
  while ((int) b.size() < n) {
    vector<T> x(a.begin(), a.begin() + min(a.size(), b.size() << 1));
    b.resize(b.size() << 1);
    x *= inverse(b);
    T inv2 = 1 / static_cast<T>(2);
    for (int i = (int) b.size() >> 1; i < (int) min(x.size(), b.size()); i
        ++) {
      b[i] = x[i] * inv2;
    }
  }
  b.resize(n);
  return b;
}

template <typename T>
vector<T> multiply(const vector<vector<T>>& a) {
  if (a.empty()) {
    return {0};
  }
  function<vector<T>(int, int)> mult = [&](int l, int r) {
    if (l == r) {
      return a[l];
    }
    int y = (l + r) >> 1;
    return mult(l, y) * mult(y + 1, r);
  };
  return mult(0, (int) a.size() - 1);
}

template <typename T>
T evaluate(const vector<T>& a, const T& x) {
  T res = 0;
  for (int i = (int) a.size() - 1; i >= 0; i--) {
    res = res * x + a[i];
  }
  return res;
}

template <typename T>
vector<T> evaluate(const vector<T>& a, const vector<T>& x) {
  if (x.empty()) {
    return {};
  }
  if (a.empty()) {
    return vector<T>(x.size(), 0);
  }
  int n = (int) x.size();
  vector<vector<T>> st((n << 1) - 1);
  function<void(int, int, int)> build = [&](int v, int l, int r) {
    if (l == r) {
      st[v] = vector<T>{-x[l], 1};
    } else {
      int y = (l + r) >> 1;
      int z = v + ((y - l + 1) << 1);
      build(v + 1, l, y);
      build(z, y + 1, r);
      st[v] = st[v + 1] * st[z];
    }
  };
  build(0, 0, n - 1);
  vector<T> res(n);
  function<void(int, int, int, vector<T>)> eval = [&](int v, int l, int r,
      vector<T> f) {
    f %= st[v];
    if ((int) f.size() < 150) {
      for (int i = l; i <= r; i++) {
```

```
        res[i] = evaluate(f, x[i]);
      }
      return;
    }
    if (l == r) {
      res[l] = f[0];
    } else {
      int y = (l + r) >> 1;
      int z = v + ((y - l + 1) << 1);
      eval(v + 1, l, y, f);
      eval(z, y + 1, r, f);
    }
  };
  eval(0, 0, n - 1, a);
  return res;
}

template <typename T>
vector<T> interpolate(const vector<T>& x, const vector<T>& y) {
  if (x.empty()) {
    return {};
  }
  assert(x.size() == y.size());
  int n = (int) x.size();
  vector<vector<T>> st((n << 1) - 1);
  function<void(int, int, int)> build = [&](int v, int l, int r) {
    if (l == r) {
      st[v] = vector<T>{-x[l], 1};
    } else {
      int w = (l + r) >> 1;
      int z = v + ((w - l + 1) << 1);
      build(v + 1, l, w);
      build(z, w + 1, r);
      st[v] = st[v + 1] * st[z];
    }
  };
  build(0, 0, n - 1);
  vector<T> m = st[0];
  vector<T> dm = derivative(m);
  vector<T> val(n);
  function<void(int, int, int, vector<T>)> eval = [&](int v, int l, int r,
      vector<T> f) {
    f %= st[v];
    if ((int) f.size() < 150) {
      for (int i = l; i <= r; i++) {
        val[i] = evaluate(f, x[i]);
      }
      return;
    }
    if (l == r) {
      val[l] = f[0];
    } else {
      int w = (l + r) >> 1;
      int z = v + ((w - l + 1) << 1);
      eval(v + 1, l, w, f);
      eval(z, w + 1, r, f);
    }
  };
  eval(0, 0, n - 1, dm);
  for (int i = 0; i < n; i++) {
    val[i] = y[i] / val[i];
  }
  function<vector<T>(int, int, int)> calc = [&](int v, int l, int r) {
    if (l == r) {
      return vector<T>{val[l]};
    }
    int w = (l + r) >> 1;
    int z = v + ((w - l + 1) << 1);
    return calc(v + 1, l, w) * st[z] + calc(z, w + 1, r) * st[v + 1];
  };
  return calc(0, 0, n - 1);
}

// f[i] = 1^i + 2^i + ... + up^i
template <typename T>
vector<T> faulhaber(const T& up, int n) {
  vector<T> ex(n + 1);
  T e = 1;
  for (int i = 0; i <= n; i++) {
    ex[i] = e;
    e /= i + 1;
  }
  vector<T> den = ex;
  den.erase(den.begin());
  for (auto& d : den) {
    d = -d;
```

```
  }
  vector<T> num(n);
  T p = 1;
  for (int i = 0; i < n; i++) {
    p *= up + 1;
    num[i] = ex[i + 1] * (1 - p);
  }
  vector<T> res = num * inverse(den);
  res.resize(n);
  T f = 1;
  for (int i = 0; i < n; i++) {
    res[i] *= f;
    f *= i + 1;
  }
  return res;
}

// (x + 1) * (x + 2) * ... * (x + n)
// (can be optimized with precomputed inverses)
template <typename T>
vector<T> sequence(int n) {
  if (n == 0) {
    return {1};
  }
  if (n % 2 == 1) {
    return sequence<T>(n - 1) * vector<T>{n, 1};
  }
  vector<T> c = sequence<T>(n / 2);
  vector<T> a = c;
  reverse(a.begin(), a.end());
  T f = 1;
  for (int i = n / 2 - 1; i >= 0; i--) {
    f *= n / 2 - i;
    a[i] *= f;
  }
  vector<T> b(n / 2 + 1);
  b[0] = 1;
  for (int i = 1; i <= n / 2; i++) {
    b[i] = b[i - 1] * (n / 2) / i;
  }
  vector<T> h = a * b;
  h.resize(n / 2 + 1);
  reverse(h.begin(), h.end());
  f = 1;
  for (int i = 1; i <= n / 2; i++) {
    f /= i;
    h[i] *= f;
  }
  vector<T> res = c * h;
  return res;
}

template <typename T>
class OnlineProduct {
 public:
  const vector<T> a;
  vector<T> b;
  vector<T> c;

  OnlineProduct(const vector<T>& a_) : a(a_) {}

  T add(const T& val) {
    int i = (int) b.size();
    b.push_back(val);
    if ((int) c.size() <= i) {
      c.resize(i + 1);
    }
    c[i] += a[0] * b[i];
    int z = 1;
    while ((i & (z - 1)) == z - 1 && (int) a.size() > z) {
      vector<T> a_mul(a.begin() + z, a.begin() + min(z << 1, (int) a.size())
          );
      vector<T> b_mul(b.end() - z, b.end());
      vector<T> c_mul = a_mul * b_mul;
      if ((int) c.size() <= i + (int) c_mul.size()) {
        c.resize(i + c_mul.size() + 1);
      }
      for (int j = 0; j < (int) c_mul.size(); j++) {
        c[i + 1 + j] += c_mul[j];
      }
      z <<= 1;
    }
    return c[i];
  }
};
```

5.11 primitive.cpp

```
template <typename T>
struct PrimitiveVarMod { static T value; };
template <typename T>
T PrimitiveVarMod<T>::value;

template <typename T, class F>
T GetPrimitiveRoot(const T& modulo, const F& factorize) {
  if (modulo <= 0) {
    return -1;
  }
  if (modulo == 1 || modulo == 2 || modulo == 4) {
    return modulo - 1;
  }
  vector<pair<T, int>> modulo_factors = factorize(modulo);
  if (modulo_factors[0].first == 2 && (modulo_factors[0].second != 1 ||
      modulo_factors.size() != 2)) {
    return -1;
  }
  if (modulo_factors[0].first != 2 && modulo_factors.size() != 1) {
    return -1;
  }
  set<T> phi_factors;
  T phi = modulo;
  for (auto& d : modulo_factors) {
    phi = phi / d.first * (d.first - 1);
    if (d.second > 1) {
      phi_factors.insert(d.first);
    }
    for (auto& e : factorize(d.first - 1)) {
      phi_factors.insert(e.first);
    }
  }
  PrimitiveVarMod<T>::value = modulo;
  Modular<PrimitiveVarMod<T>> gen = 2;
  while (gen != 0) {
    if (power(gen, phi) != 1) {
      continue;
    }
    bool ok = true;
    for (auto& p : phi_factors) {
       if (power(gen, phi / p) == 1) {
         ok = false;
         break;
       }
     }
    if (ok) {
      return gen();
    }
    gen++;
  }
  assert(false);
  return -1;
}

template <typename T>
T GetPrimitiveRoot(const T& modulo) {
  return GetPrimitiveRoot(modulo, factorizer::Factorize<T>);
}
```

5.12 simplex.cpp

```
typedef long double ld;

const ld eps = 1e-8;

vector<ld> simplex(vector<vector<ld>> a) {
  int n = (int) a.size() - 1;
  int m = (int) a[0].size() - 1;
  vector<int> left(n + 1);
  vector<int> up(m + 1);
  iota(left.begin(), left.end(), m);
  iota(up.begin(), up.end(), 0);
  auto pivot = [&](int x, int y) {
    swap(left[x], up[y]);
    ld k = a[x][y];
    a[x][y] = 1;
    vector<int> pos;
    for (int j = 0; j <= m; j++) {
      a[x][j] /= k;
      if (fabs(a[x][j]) > eps) {
        pos.push_back(j);
      }
    }
    for (int i = 0; i <= n; i++) {
      if (fabs(a[i][y]) < eps || i == x) {
```

```
        continue;
      }
      k = a[i][y];
      a[i][y] = 0;
      for (int j : pos) {
        a[i][j] -= k * a[x][j];
      }
    }
  };
  while (1) {
    int x = -1;
    for (int i = 1; i <= n; i++) {
      if (a[i][0] < -eps && (x == -1 || a[i][0] < a[x][0])) {
        x = i;
      }
    }
    if (x == -1) {
      break;
    }
    int y = -1;
    for (int j = 1; j <= m; j++) {
      if (a[x][j] < -eps && (y == -1 || a[x][j] < a[x][y])) {
        y = j;
      }
    }
    if (y == -1) {
      return vector<ld>(); // infeasible
    }
    pivot(x, y);
  }
  while (1) {
    int y = -1;
    for (int j = 1; j <= m; j++) {
      if (a[0][j] > eps && (y == -1 || a[0][j] > a[0][y])) {
        y = j;
      }
    }
    if (y == -1) {
      break;
    }
    int x = -1;
    for (int i = 1; i <= n; i++) {
      if (a[i][y] > eps && (x == -1 || a[i][0] / a[i][y] < a[x][0] / a[x][y
          ])) {
        x = i;
      }
    }
    if (x == -1) {
      return vector<ld>(); // unbounded
    }
    pivot(x, y);
  }
  vector<ld> ans(m + 1);
  for (int i = 1; i <= n; i++) {
    if (left[i] <= m) {
      ans[left[i]] = a[i][0];
    }
  }
  ans[0] = -a[0][0];
  return ans;
}
```

5.13 sparsematrix.cpp

```
const double eps = 1e-9;

bool IsZero(double v) {
  return abs(v) < 1e-9;
}

template <typename T>
class SparseMatrix {
 public:
  int h;
  int w;
  vector<map<int, T>> rows;
  vector<map<int, T>> cols;

  SparseMatrix(int h_, int w_) : h(h_), w(w_) {
    rows.resize(h);
    cols.resize(w);
  }

  void set(int i, int j, const T& value) {
    if (IsZero(value)) {
      rows[i].erase(j);
```

```
      cols[j].erase(i);
    } else {
      rows[i][j] = value;
      cols[j][i] = value;
    }
  }

  void modify(int i, int j, const T& value) {
    if (IsZero(value)) {
      return;
    }
    auto it = rows[i].find(j);
    if (it == rows[i].end()) {
      rows[i][j] = value;
      cols[j][i] = value;
    } else {
      it->second += value;
      if (IsZero(it->second)) {
        rows[i].erase(it);
        cols[j].erase(i);
      } else {
        cols[j][i] = it->second;
      }
    }
  }

  T get(int i, int j) {
    auto it = rows[i].find(j);
    if (it == rows[i].end()) {
      return T{};
    }
    return it->second;
  }

  void transpose() {
    swap(h, w);
    swap(rows, cols);
  }
};

template <typename T>
void GaussianElimination(SparseMatrix<T>& a, int limit) {
  assert(limit <= a.w);
  int r = 0;
  for (int c = 0; c < limit; c++) {
    int mn = a.w + 1;
    int id = -1;
    for (auto& p : a.cols[c]) {
      int i = p.first;
      if (i >= r) {
        int sz = static_cast<int>(a.rows[i].size());
        if (sz < mn) {
          mn = sz;
          id = i;
        }
      }
    }
    if (id == -1) {
      continue;
    }
    if (id > r) {
      set<int> s;
      for (auto& p : a.rows[r]) {
        s.insert(p.first);
      }
      for (auto& p : a.rows[id]) {
        s.insert(p.first);
      }
      for (int j : s) {
        T tmp = a.get(r, j);
        a.set(r, j, a.get(id, j));
        a.set(id, j, -tmp);
      }
    }
    T inv_a = 1 / a.get(r, c);
    vector<int> touched_rows;
    for (auto& p : a.cols[c]) {
      int i = p.first;
      if (i > r) {
        touched_rows.push_back(i);
        T coeff = -p.second * inv_a;
        for (auto& q : a.rows[r]) {
          if (q.first != c) {
            a.modify(i, q.first, coeff * q.second);
          }
        }
```

```
      }
    }
    for (int i : touched_rows) {
      a.set(i, c, 0);
    }
    ++r;
  }
}

template <typename T>
T Determinant(SparseMatrix<T>/*&*/ a) {
  assert(a.h == a.w);
  GaussianElimination(a, a.w);
  T d{1};
  for (int i = 0; i < a.h; i++) {
    d *= a.get(i, i);
  }
  return d;
}

template <typename T>
int Rank(SparseMatrix<T>/*&*/ a) {
  GaussianElimination(a, a.w);
  int rank = 0;
  for (int i = 0; i < a.h; i++) {
    if (!a.rows[i].empty()) {
      ++rank;
    }
  }
  return rank;
}

template <typename T>
vector<T> SolveLinearSystem(SparseMatrix<T>/*&*/ a, const vector<T>& b) {
  assert(a.h == static_cast<int>(b.size()));
  ++a.w;
  a.cols.emplace_back();
  for (int i = 0; i < a.h; i++) {
    a.set(i, a.w - 1, b[i]);
  }
  GaussianElimination(a, a.w - 1);
  vector<T> x(a.h, 0);
  for (int r = a.h - 1; r >= 0; r--) {
    int c = a.rows[r].begin()->first;
    if (c == a.w - 1) {
      return {};
    }
    x[c] = a.get(r, a.w - 1) / a.get(r, c);
    vector<int> touched_rows;
    for (auto& q : a.cols[c]) {
      int i = q.first;
      if (i < r) {
        touched_rows.push_back(i);
        a.modify(i, a.w - 1, -x[c] * q.second);
      }
    }
    for (int i : touched_rows) {
      a.set(i, c, 0);
    }
  }
  return x;
}
```

6 segtree

6.1 dynamic-fenwick.cpp

```
template <typename T>
class DynamicFenwickTree {
 public:
  HashMap<int, T> fenw;
  int n;
  int pw;

  DynamicFenwickTree() : n(0) {}
  DynamicFenwickTree(int n_) : n(n_) {
    pw = bit_floor(unsigned(n));
  }

  void Modify(int x, T v) {
    assert(0 <= x && x < n);
    while (x < n) {
      fenw[x] += v;
      x |= x + 1;
    }
```

```
  }

  T Query(int x) {
    assert(0 <= x && x <= n);
    T v{};
    while (x > 0) {
      auto it = fenw.find(x - 1);
      if (it != fenw.end()) {
        v += it->second;
      }
      x &= x - 1;
    }
    return v;
  }

  // Returns the length of the longest prefix with sum <= c
  int MaxPrefix(T c) {
    T v{};
    int at = 0;
    for (int len = pw; len > 0; len >>= 1) {
      if (at + len <= n) {
        auto nv = v;
        auto it = fenw.find(at + len - 1);
        if (it != fenw.end()) {
          nv += it->second;
        }
        if (!(c < nv)) {
          v = nv;
          at += len;
        }
      }
    }
    assert(0 <= at && at <= n);
    return at;
  }
};
```

6.2 dynamic-lazy.cpp

```
template <typename Info, typename Tag, typename Index = int, bool Persistent
    = false, bool Commutative = false>
class DynamicLazySegmentTree {
 public:
  struct Node {
    array<int, 2> c;
    Info d;
    Tag t;
  };

  Index n;
  vector<Node> nodes;

  DynamicLazySegmentTree(Index n_ = 0, int q = -1) : n(n_) {
    if (q >= 0) {
      nodes.reserve(2 + q * (Commutative ? 2 : 4) * bit_width(
          make_unsigned_t<Index>(2 * n - 1)));
    }
    nodes.resize(2);
    nodes[1] = {{-1, -1}, Info::GetDefault(0, n), Tag()};
  }

  DynamicLazySegmentTree(const vector<Info>& init, int q = -1) : n(Index(
      init.size())) {
    if (q >= 0) {
      nodes.reserve(2 * n + q * (Commutative ? 2 : 4) * bit_width(
          make_unsigned_t<Index>(2 * n - 1)));
    }
    nodes.resize(2 * n);
    auto Build = [&](auto&& self, int id, Index nl, Index nr) -> void {
      if (nr - nl == 1) {
        nodes[id] = {{-1, -1}, init[nl], Tag()};
        return;
      }
      Index mid = (nl + nr) >> 1;
      array<int, 2> c = {id + 1, id + 2 * (mid - nl)};
      self(self, c[0], nl, mid);
      self(self, c[1], mid, nr);
      nodes[id] = {c, nodes[c[0]].d.Unite(nodes[c[1]].d), Tag()};
    };
    Build(Build, 1, 0, n);
  }

  void RefreshNode(int& id, Index nl, Index nr) {
    if (id == -1) {
      nodes.push_back({{-1, -1}, Info::GetDefault(nl, nr), Tag()});
      id = int(nodes.size()) - 1;
```

```
    } else {
      if (Persistent) {
        nodes.push_back(nodes[id]);
        id = int(nodes.size()) - 1;
      }
    }
  }

  int ModifyImpl(Index l, Index r, const Tag& t, int id, Index nl, Index nr,
       Tag above) {
    RefreshNode(id, nl, nr);
    if (!above.Empty()) {
      above.ApplyTo(nodes[id].d, nl, nr);
      above.ApplyTo(nodes[id].t);
    }
    if (nr <= l || nl >= r) {
      return id;
    }
    if (l <= nl && nr <= r) {
      t.ApplyTo(nodes[id].d, nl, nr);
      t.ApplyTo(nodes[id].t);
      return id;
    }
    Index mid = (nl + nr) >> 1;
    if ((!Commutative && !nodes[id].t.Empty()) || l < mid) {
      int got = ModifyImpl(l, r, t, nodes[id].c[0], nl, mid, Commutative ?
          Tag() : nodes[id].t);
      nodes[id].c[0] = got;
    }
    if ((!Commutative && !nodes[id].t.Empty()) || r > mid) {
      int got = ModifyImpl(l, r, t, nodes[id].c[1], mid, nr, Commutative ?
          Tag() : nodes[id].t);
      nodes[id].c[1] = got;
    }
    auto lft = nodes[id].c[0] == -1 ? Info::GetDefault(nl, mid) : nodes[
        nodes[id].c[0]].d;
    auto rgt = nodes[id].c[1] == -1 ? Info::GetDefault(mid, nr) : nodes[
        nodes[id].c[1]].d;
    nodes[id].d = lft.Unite(rgt);
    if (Commutative) {
      nodes[id].t.ApplyTo(nodes[id].d, nl, nr);
    } else {
      nodes[id].t = Tag();
    }
    return id;
  }

  int Modify(Index l, Index r, const Tag& t, int root = Persistent ? -1 : 1)
       {
    assert(0 <= l && l <= r && r <= n && root >= 1);
    return l == r ? root : ModifyImpl(l, r, t, root, 0, n, Tag());
  }

  int SetImpl(Index p, const Info& v, int id, Index nl, Index nr, Tag above)
       {
    RefreshNode(id, nl, nr);
    if (!above.Empty()) {
      above.ApplyTo(nodes[id].d, nl, nr);
      above.ApplyTo(nodes[id].t);
    }
    if (p < nl || p >= nr) {
      return id;
    }
    if (nr - nl == 1) {
      nodes[id].d = v;
    } else {
      Index mid = (nl + nr) >> 1;
      int got = SetImpl(p, v, nodes[id].c[0], nl, mid, nodes[id].t);
      nodes[id].c[0] = got;
      got = SetImpl(p, v, nodes[id].c[1], mid, nr, nodes[id].t);
      nodes[id].c[1] = got;
      auto lft = nodes[nodes[id].c[0]].d;
      auto rgt = nodes[nodes[id].c[1]].d;
      nodes[id].d = lft.Unite(rgt);
      nodes[id].t = Tag();
    }
    return id;
  }

  int Set(Index p, const Info& v, int root = Persistent ? -1 : 1) {
    assert(0 <= p && p < n && root >= 1);
    return SetImpl(p, v, root, 0, n, Tag());
  }

  Info QueryImpl(Index l, Index r, int id, Index nl, Index nr, Tag t) {
    if (id == -1) {
```

```
      nl = max(l, nl);
      nr = min(r, nr);
      Info res = Info::GetDefault(nl, nr);
      if (!t.Empty()) {
        t.ApplyTo(res, nl, nr);
      }
      return res;
    }
    if (l <= nl && nr <= r) {
      Info res = nodes[id].d;
      if (!t.Empty()) {
        t.ApplyTo(res, nl, nr);
      }
      return res;
    }
    auto nt = nodes[id].t;
    if (!t.Empty()) {
      t.ApplyTo(nt);
    }
    Index mid = (nl + nr) >> 1;
    auto lft = l < mid ? QueryImpl(l, r, nodes[id].c[0], nl, mid, nt) : Info
        ();
    auto rgt = r > mid ? QueryImpl(l, r, nodes[id].c[1], mid, nr, nt) : Info
        ();
    return lft.Unite(rgt);
  }

  Info Query(Index l, Index r, int root = Persistent ? -1 : 1) {
    assert(0 <= l && l <= r && r <= n && root >= 1);
    return l == r ? Info() : QueryImpl(l, r, root, 0, n, Tag());
  }

  Info Get(Index p, int root = Persistent ? -1 : 1) {
    assert(0 <= p && p < n && root >= 1);
    int id = root;
    Index nl = 0;
    Index nr = n;
    Tag t;
    while (nr - nl > 1 && id != -1) {
      auto nt = nodes[id].t;
      if (!t.Empty()) {
        t.ApplyTo(nt);
      }
      t = nt;
      Index mid = (nl + nr) >> 1;
      if (p < mid) {
        id = nodes[id].c[0];
        nr = mid;
      } else {
        id = nodes[id].c[1];
        nl = mid;
      }
    }
    Info res = id == -1 ? Info::GetDefault(p, p + 1) : nodes[id].d;
    if (!t.Empty()) {
      t.ApplyTo(res, p, p + 1);
    }
    return res;
  }

  template<int N, typename F>
  Index MaxRight(array<int, N> roots, Index l, F f) {
    assert(0 <= l && l <= n);
    if (l == n) {
      return n;
    }
    Index res = n;
    array<Info, N> sums;
    for (int i = 0; i < N; i++) {
      sums[i] = Info();
    }
    array<Info, N> new_sums;
    auto Dfs = [&](auto&& self, array<int, N> v, Index nl, Index nr, array<
        Tag, N> tags) -> void {
      if (res != n) {
        return;
      }
      array<int, N> to;
      if (nl < l) {
        Index mid = (nl + nr) >> 1;
        for (int i = 0; i < N; i++) {
          auto nt = v[i] == -1 ? Tag() : nodes[v[i]].t;
          tags[i].ApplyTo(nt);
          tags[i] = nt;
        }
        if (l < mid) {
```

```
      for (int i = 0; i < N; i++) {
        to[i] = v[i] == -1 ? -1 : nodes[v[i]].c[0];
      }
      self(self, to, nl, mid, tags);
      if (res != n) {
        return;
      }
    }
    for (int i = 0; i < N; i++) {
      to[i] = v[i] == -1 ? -1 : nodes[v[i]].c[1];
    }
    self(self, to, mid, nr, tags);
    return;
  }
  for (int i = 0; i < N; i++) {
    auto d = v[i] == -1 ? Info::GetDefault(nl, nr) : nodes[v[i]].d;
    tags[i].ApplyTo(d, nl, nr);
    new_sums[i] = sums[i].Unite(d);
  }
  if (f(new_sums)) {
    sums = new_sums;
    return;
  }
  while (nr - nl > 1) {
    Index mid = (nl + nr) >> 1;
    for (int i = 0; i < N; i++) {
      auto nt = v[i] == -1 ? Tag() : nodes[v[i]].t;
      tags[i].ApplyTo(nt);
      tags[i] = nt;
    }
    for (int i = 0; i < N; i++) {
      auto d = v[i] == -1 || nodes[v[i]].c[0] == -1 ? Info::GetDefault(
          nl, mid) : nodes[nodes[v[i]].c[0]].d;
      tags[i].ApplyTo(d, nl, mid);
      new_sums[i] = sums[i].Unite(d);
    }
    if (f(new_sums)) {
      sums = new_sums;
      nl = mid;
      for (int i = 0; i < N; i++) {
        v[i] = v[i] == -1 ? -1 : nodes[v[i]].c[1];
      }
    } else {
      nr = mid;
      for (int i = 0; i < N; i++) {
        v[i] = v[i] == -1 ? -1 : nodes[v[i]].c[0];
      }
    }
  }
  res = nl;
};
array<Tag, N> tags;
for (int i = 0; i < N; i++) {
  tags[i] = Tag();
}
Dfs(Dfs, roots, 0, n, tags);
return res;
}

template<int N, typename F>
Index MinLeft(array<int, N> roots, Index r, F f) {
  assert(0 <= r && r <= n);
  if (r == 0) {
    return 0;
  }
  Index res = 0;
  array<Info, N> sums;
  for (int i = 0; i < N; i++) {
    sums[i] = Info();
  }
  array<Info, N> new_sums;
  auto Dfs = [&](auto&& self, array<int, N> v, Index nl, Index nr, array<
      Tag, N> tags) -> void {
    if (res != 0) {
      return;
    }
    array<int, N> to;
    if (nr > r) {
      Index mid = (nl + nr) >> 1;
      for (int i = 0; i < N; i++) {
        auto nt = v[i] == -1 ? Tag() : nodes[v[i]].t;
        tags[i].ApplyTo(nt);
        tags[i] = nt;
      }
      if (r > mid) {
        for (int i = 0; i < N; i++) {
```

```
        to[i] = v[i] == -1 ? -1 : nodes[v[i]].c[1];
      }
      self(self, to, mid, nr, tags);
      if (res != 0) {
        return;
      }
    }
    for (int i = 0; i < N; i++) {
      to[i] = v[i] == -1 ? -1 : nodes[v[i]].c[0];
    }
    self(self, to, nl, mid, tags);
    return;
  }
  for (int i = 0; i < N; i++) {
    auto d = v[i] == -1 ? Info::GetDefault(nl, nr) : nodes[v[i]].d;
    tags[i].ApplyTo(d, nl, nr);
    new_sums[i] = d.Unite(sums[i]);
  }
  if (f(new_sums)) {
    sums = new_sums;
    return;
  }
  while (nr - nl > 1) {
    Index mid = (nl + nr) >> 1;
    for (int i = 0; i < N; i++) {
      auto nt = v[i] == -1 ? Tag() : nodes[v[i]].t;
      tags[i].ApplyTo(nt);
      tags[i] = nt;
    }
    for (int i = 0; i < N; i++) {
      auto d = v[i] == -1 || nodes[v[i]].c[1] == -1 ? Info::GetDefault(
          mid, nr) : nodes[nodes[v[i]].c[1]].d;
      tags[i].ApplyTo(d, mid, nr);
      new_sums[i] = d.Unite(sums[i]);
    }
    if (f(new_sums)) {
      sums = new_sums;
      nr = mid;
      for (int i = 0; i < N; i++) {
        v[i] = v[i] == -1 ? -1 : nodes[v[i]].c[0];
      }
    } else {
      nl = mid;
      for (int i = 0; i < N; i++) {
        v[i] = v[i] == -1 ? -1 : nodes[v[i]].c[1];
      }
    }
  }
  res = nr;
 };
 array<Tag, N> tags;
 for (int i = 0; i < N; i++) {
  tags[i] = Tag();
 }
 Dfs(Dfs, roots, 0, n, tags);
 return res;
 }
};
```

6.3 dynamic-simple.cpp

```
template <typename Info, typename Index = int, bool Persistent = false>
class DynamicSimpleSegmentTree {
 public:
  struct Node {
    array<int, 2> c;
    Info d;
  };

  Index n;
  vector<Node> nodes;

  DynamicSimpleSegmentTree() : DynamicSimpleSegmentTree(0, -1) {}
  DynamicSimpleSegmentTree(Index n_) : DynamicSimpleSegmentTree(n_, -1) {}
  DynamicSimpleSegmentTree(const vector<Info>& a) : DynamicSimpleSegmentTree
      (a, -1) {}

  DynamicSimpleSegmentTree(Index n_, int q) : n(n_) {
    if (q >= 0) {
      nodes.reserve(2 + q * bit_width(make_unsigned_t<Index>(2 * n - 1)));
    }
    nodes.resize(2);
    nodes[1] = {{-1, -1}, Info::GetDefault(0, n)};
  }

  DynamicSimpleSegmentTree(const vector<Info>& a, int q) : n(int(a.size()))
```

```
    {
  if (q >= 0) {
    nodes.reserve(2 * n + q * bit_width(make_unsigned_t<Index>(2 * n - 1))
        );
  }
  nodes.resize(2 * n);
  auto Build = [&](auto&& self, int id, int nl, int nr) -> void {
    if (nr - nl == 1) {
      nodes[id] = {{-1, -1}, a[nl]};
      return;
    }
    int mid = (nl + nr) >> 1;
    array<int, 2> c = {id + 1, id + 2 * (mid - nl)};
    self(self, c[0], nl, mid);
    self(self, c[1], mid, nr);
    nodes[id] = {c, nodes[c[0]].d.Unite(nodes[c[1]].d)};
  };
  Build(Build, 1, 0, n);
}

int SetImpl(int root, Index p, const Info& v, Index nl, Index nr) {
  int me;
  if (root == -1) {
    me = int(nodes.size());
    nodes.push_back({{-1, -1}, Info::GetDefault(nl, nr)});
  } else {
    if (Persistent) {
      me = int(nodes.size());
      nodes.push_back(nodes[root]);
    } else {
      me = root;
    }
  }
  if (nr - nl == 1) {
    nodes[me].d = v;
  } else {
    Index mid = (nl + nr) >> 1;
    if (p < mid) {
      int got = SetImpl(nodes[me].c[0], p, v, nl, mid);
      nodes[me].c[0] = got;
    } else {
      int got = SetImpl(nodes[me].c[1], p, v, mid, nr);
      nodes[me].c[1] = got;
    }
    auto lft = nodes[me].c[0] == -1 ? Info::GetDefault(nl, mid) : nodes[
        nodes[me].c[0]].d;
    auto rgt = nodes[me].c[1] == -1 ? Info::GetDefault(mid, nr) : nodes[
        nodes[me].c[1]].d;
    nodes[me].d = lft.Unite(rgt);
  }
  return me;
}

int Set(Index p, const Info& v, int root = Persistent ? -1 : 1) {
  assert(0 <= p && p < n && root >= 1);
  return SetImpl(root, p, v, 0, n);
}

Info QueryImpl(int root, Index l, Index r, Index nl, Index nr) {
  if (root == -1) {
    return Info::GetDefault(max(l, nl), min(r, nr));
  }
  if (l <= nl && nr <= r) {
    return nodes[root].d;
  }
  Index mid = (nl + nr) >> 1;
  auto lft = l < mid ? QueryImpl(nodes[root].c[0], l, r, nl, mid) : Info()
      ;
  auto rgt = r > mid ? QueryImpl(nodes[root].c[1], l, r, mid, nr) : Info()
      ;
  return lft.Unite(rgt);
}

Info Query(Index l, Index r, int root = Persistent ? -1 : 1) {
  assert(0 <= l && l <= r && r <= n && root >= 1);
  if (l == r) {
    return Info();
  }
  return QueryImpl(root, l, r, 0, n);
}

Info Get(Index p, int root = Persistent ? -1 : 1) {
  assert(0 <= p && p < n && root >= 1);
  Index nl = 0;
  Index nr = n;
  while (nr - nl > 1 && root != -1) {
```

```
    Index mid = (nl + nr) >> 1;
    if (p < mid) {
      root = nodes[root].c[0];
      nr = mid;
    } else {
      root = nodes[root].c[1];
      nl = mid;
    }
  }
  return root == -1 ? Info::GetDefault(p, p + 1) : nodes[root].d;
}

template<int N, typename F>
Index MaxRight(array<int, N> roots, Index l, F f) {
  assert(0 <= l && l <= n);
  if (l == n) {
    return n;
  }
  Index res = n;
  array<Info, N> sums;
  for (int i = 0; i < N; i++) {
    sums[i] = Info();
  }
  array<Info, N> new_sums;
  auto Dfs = [&](auto&& self, array<int, N> v, Index nl, Index nr) -> void {
    if (res != n) {
      return;
    }
    array<int, N> to;
    if (nl < l) {
      Index mid = (nl + nr) >> 1;
      if (l < mid) {
        for (int i = 0; i < N; i++) {
          to[i] = v[i] == -1 ? -1 : nodes[v[i]].c[0];
        }
        self(self, to, nl, mid);
        if (res != n) {
          return;
        }
      }
      for (int i = 0; i < N; i++) {
        to[i] = v[i] == -1 ? -1 : nodes[v[i]].c[1];
      }
      self(self, to, mid, nr);
      return;
    }
    for (int i = 0; i < N; i++) {
      new_sums[i] = sums[i].Unite(v[i] == -1 ? Info::GetDefault(nl, nr) :
          nodes[v[i]].d);
    }
    if (f(new_sums)) {
      sums = new_sums;
      return;
    }
    while (nr - nl > 1) {
      Index mid = (nl + nr) >> 1;
      for (int i = 0; i < N; i++) {
        new_sums[i] = sums[i].Unite(v[i] == -1 || nodes[v[i]].c[0] == -1 ?
            Info::GetDefault(nl, mid) : nodes[nodes[v[i]].c[0]].d);
      }
      if (f(new_sums)) {
        sums = new_sums;
        nl = mid;
        for (int i = 0; i < N; i++) {
          v[i] = v[i] == -1 ? -1 : nodes[v[i]].c[1];
        }
      } else {
        nr = mid;
        for (int i = 0; i < N; i++) {
          v[i] = v[i] == -1 ? -1 : nodes[v[i]].c[0];
        }
      }
    }
    res = nl;
  };
  Dfs(Dfs, roots, 0, n);
  return res;
}

template<int N, typename F>
Index MinLeft(array<int, N> roots, Index r, F f) {
  assert(0 <= r && r <= n);
  if (r == 0) {
    return 0;
  }
```

```
    Index res = 0;
    array<Info, N> sums;
    for (int i = 0; i < N; i++) {
      sums[i] = Info();
    }
    array<Info, N> new_sums;
    auto Dfs = [&](auto&& self, array<int, N> v, Index nl, Index nr) -> void {
      if (res != 0) {
        return;
      }
      array<int, N> to;
      if (nr > r) {
        Index mid = (nl + nr) >> 1;
        if (r > mid) {
          for (int i = 0; i < N; i++) {
            to[i] = v[i] == -1 ? -1 : nodes[v[i]].c[1];
          }
          self(self, to, mid, nr);
          if (res != 0) {
            return;
          }
        }
        for (int i = 0; i < N; i++) {
          to[i] = v[i] == -1 ? -1 : nodes[v[i]].c[0];
        }
        self(self, to, nl, mid);
        return;
      }
      for (int i = 0; i < N; i++) {
        new_sums[i] = (v[i] == -1 ? Info::GetDefault(nl, nr) : nodes[v[i]].d).Unite(sums[i]);
      }
      if (f(new_sums)) {
        sums = new_sums;
        return;
      }
      while (nr - nl > 1) {
        Index mid = (nl + nr) >> 1;
        for (int i = 0; i < N; i++) {
          new_sums[i] = (v[i] == -1 || nodes[v[i]].c[1] == -1 ? Info::GetDefault(mid, nr) : nodes[nodes[v[i]].c[1]].d).Unite(sums[i]);
        }
        if (f(new_sums)) {
          sums = new_sums;
          nr = mid;
          for (int i = 0; i < N; i++) {
            v[i] = v[i] == -1 ? -1 : nodes[v[i]].c[0];
          }
        } else {
          nl = mid;
          for (int i = 0; i < N; i++) {
            v[i] = v[i] == -1 ? -1 : nodes[v[i]].c[1];
          }
        }
      }
      res = nr;
    };
    Dfs(Dfs, roots, 0, n);
    return res;
  }
};
```

6.4 info.cpp

```
struct Info {
  ${0}... a = ...;

  Info Unite(const Info& b) const {
    Info res;
    ...
    return res;
  }

  static Info GetDefault([[maybe_unused]] int l, [[maybe_unused]] int r) {
    return Info();
  }
};
```

6.5 layout.cpp

```
namespace seg_tree {

// Floor of log_2(a); index of highest 1-bit
inline int floor_log_2(int a) {
```

```
  return a ? bit_width(unsigned(a)) - 1 : -1;
}

struct point {
  int a;
  point() : a(0) {}
  explicit point(int a_) : a(a_) { assert(a >= -1); }

  explicit operator bool () { return bool(a); }

  // This is useful so you can directly do array indices
  /* implicit */ operator int() const { return a; }

  point c(bool z) const {
    return point((a << 1) | z);
  }

  point operator [] (bool z) const {
    return c(z);
  }

  point p() const {
    return point(a >> 1);
  }

  friend std::ostream& operator << (std::ostream& o, const point& p) {
      return o << int(p); }

  template <typename F> void for_each(F f) const {
    for (int v = a; v > 0; v >>= 1) {
      f(point(v));
    }
  }

  template <typename F> void for_parents_down(F f) const {
    // strictly greater than 0
    for (int L = floor_log_2(a); L > 0; L--) {
      f(point(a >> L));
    }
  }

  template <typename F> void for_parents_up(F f) const {
    for (int v = a >> 1; v > 0; v >>= 1) {
      f(point(v));
    }
  }

  point& operator ++ () { ++a; return *this; }
  point operator ++ (int) { return point(a++); }
  point& operator -- () { --a; return *this; }
  point operator -- (int) { return point(a--); }
};

struct range {
  int a, b;
  range() : a(1), b(1) {}
  range(int a_, int b_) : a(a_), b(b_) {
    assert(1 <= a && a <= b && b <= 2 * a);
  }
  explicit range(std::array<int, 2> r) : range(r[0], r[1]) {}

  explicit operator std::array<int, 2>() const {
    return {a, b};
  }

  const int& operator[] (bool z) const {
    return z ? b : a;
  }

  friend std::ostream& operator << (std::ostream& o, const range& r) {
      return o << "[" << r.a << ".." << r.b << ")"; }

  // Iterate over the range from outside-in.
  //   Calls f(point a)
  template <typename F> void for_each(F f) const {
    for (int x = a, y = b; x < y; x >>= 1, y >>= 1) {
      if (x & 1) f(point(x++));
      if (y & 1) f(point(--y));
    }
  }

  // Iterate over the range from outside-in.
  //   Calls f(point a, bool is_right)
  template <typename F> void for_each_with_side(F f) const {
    for (int x = a, y = b; x < y; x >>= 1, y >>= 1) {
      if (x & 1) f(point(x++), false);
```

```
      if (y & 1) f(point(--y), true);
    }
  }

  // Iterate over the range from left to right.
  //    Calls f(point)
  template <typename F> void for_each_l_to_r(F f) const {
    int anc_depth = floor_log_2((a - 1) ^ b);
    int anc_msk = (1 << anc_depth) - 1;
    for (int v = (-a) & anc_msk; v; v &= v - 1) {
      int i = countr_zero(unsigned(v));
      f(point(((a - 1) >> i) + 1));
    }
    for (int v = b & anc_msk; v; ) {
      int i = floor_log_2(v);
      f(point((b >> i) - 1));
      v ^= (1 << i);
    }
  }

  // Iterate over the range from right to left.
  //    Calls f(point)
  template <typename F> void for_each_r_to_l(F f) const {
    int anc_depth = floor_log_2((a - 1) ^ b);
    int anc_msk = (1 << anc_depth) - 1;
    for (int v = b & anc_msk; v; v &= v - 1) {
      int i = countr_zero(unsigned(v));
      f(point((b >> i) - 1));
    }
    for (int v = (-a) & anc_msk; v; ) {
      int i = floor_log_2(v);
      f(point(((a - 1) >> i) + 1));
      v ^= (1 << i);
    }
  }

  template <typename F> void for_parents_down(F f) const {
    int x = a, y = b;
    if ((x ^ y) > x) { x <<= 1, std::swap(x, y); }
    int dx = countr_zero(unsigned(x));
    int dy = countr_zero(unsigned(y));
    int anc_depth = floor_log_2((x - 1) ^ y);
    for (int i = floor_log_2(x); i > dx; i--) {
      f(point(x >> i));
    }
    for (int i = anc_depth; i > dy; i--) {
      f(point(y >> i));
    }
  }

  template <typename F> void for_parents_up(F f) const {
    int x = a, y = b;
    if ((x ^ y) > x) { x <<= 1, std::swap(x, y); }
    int dx = countr_zero(unsigned(x));
    int dy = countr_zero(unsigned(y));
    int anc_depth = floor_log_2((x - 1) ^ y);
    for (int i = dx + 1; i <= anc_depth; i++) {
      f(point(x >> i));
    }
    for (int v = y >> (dy + 1); v; v >>= 1) {
      f(point(v));
    }
  }
};

struct in_order_layout {
  // Alias them in for convenience
  using point = seg_tree::point;
  using range = seg_tree::range;

  int n, s;
  in_order_layout() : n(0), s(0) {}
  in_order_layout(int n_) : n(n_), s(n ? bit_ceil(unsigned(n)) : 0) {}

  point get_point(int a) const {
    assert(0 <= a && a < n);
    a += s;
    return point(a >= 2 * n ? a - n : a);
  }

  range get_range(int a, int b) const {
    assert(0 <= a && a <= b && b <= n);
    if (n == 0) return range();
    a += s, b += s;
    return range((a >= 2 * n ? 2 * (a - n) : a), (b >= 2 * n ? 2 * (b - n) :
        b));
```

```
  }

  range get_range(std::array<int, 2> p) const {
    return get_range(p[0], p[1]);
  }

  int get_leaf_index(point pt) const {
    int a = int(pt);
    assert(n <= a && a < 2 * n);
    return (a < s ? a + n : a) - s;
  }

  std::array<int, 2> get_node_bounds(point pt) const {
    int a = int(pt);
    assert(1 <= a && a < 2 * n);
    int l = countl_zero(unsigned(a)) - countl_zero(unsigned(2 * n - 1));
    int x = a << l, y = (a + 1) << l;
    assert(s <= x && x < y && y <= 2 * s);
    return {(x >= 2 * n ? (x >> 1) + n : x) - s, (y >= 2 * n ? (y >> 1) + n
        : y) - s};
  }

  int get_node_split(point pt) const {
    int a = int(pt);
    assert(1 <= a && a < n);
    int l = countl_zero(unsigned(2 * a + 1)) - countl_zero(unsigned(2 * n -
        1));
    int x = (2 * a + 1) << l;
    assert(s <= x && x < 2 * s);
    return (x >= 2 * n ? (x >> 1) + n : x) - s;
  }

  int get_node_size(point pt) const {
    auto bounds = get_node_bounds(pt);
    return bounds[1] - bounds[0];
  }
};

struct circular_layout {
  // Alias them in for convenience
  using point = seg_tree::point;
  using range = seg_tree::range;

  int n;
  circular_layout() : n(0) {}
  circular_layout(int n_) : n(n_) {}

  point get_point(int a) const {
    assert(0 <= a && a < n);
    return point(n + a);
  }

  range get_range(int a, int b) const {
    assert(0 <= a && a <= b && b <= n);
    if (n == 0) return range();
    return range(n + a, n + b);
  }

  range get_range(std::array<int, 2> p) const {
    return get_range(p[0], p[1]);
  }

  int get_leaf_index(point pt) const {
    int a = int(pt);
    assert(n <= a && a < 2 * n);
    return a - n;
  }

  // Returns {x,y} so that 0 <= x < n and 1 <= y <= n
  // If the point is non-wrapping, then 0 <= x < y <= n
  std::array<int, 2> get_node_bounds(point pt) const {
    int a = int(pt);
    assert(1 <= a && a < 2 * n);
    int l = countl_zero(unsigned(a)) - countl_zero(unsigned(2 * n - 1));
    int s = bit_ceil(unsigned(n));
    int x = a << l, y = (a + 1) << l;
    assert(s <= x && x < y && y <= 2 * s);
    return {(x >= 2 * n ? x >> 1 : x) - n, (y > 2 * n ? y >> 1 : y) - n};
  }

  // Returns the split point of the node, such that 1 <= s <= n.
  int get_node_split(point pt) const {
    int a = int(pt);
    assert(1 <= a && a < n);
    return get_node_bounds(pt.c(0))[1];
  }
```

```
258
259   int get_node_size(point pt) const {
260     auto bounds = get_node_bounds(pt);
261     int r = bounds[1] - bounds[0];
262     return r > 0 ? r : r + n;
263   }
264 };
265
266 } // namespace seg_tree
```

6.6 lazy.cpp

```
1  template <typename Info, typename Tag>
2  class LazySegmentTree {
3   public:
4    int n;
5    vector<Info> infos;
6    vector<Tag> tags;
7    seg_tree::in_order_layout layout;
8
9    void Apply(seg_tree::point a, const Tag& t) {
10     auto [l, r] = layout.get_node_bounds(a);
11     if (!t.ApplyTo(infos[a], l, r)) {
12       assert(a < n);
13       DowndateNode(a);
14       Apply(a.c(0), t);
15       Apply(a.c(1), t);
16       UpdateNode(a);
17       return;
18     }
19     if (a < n) {
20       t.ApplyTo(tags[a]);
21     }
22   }
23
24   void DowndateNode(seg_tree::point a) {
25     if (!tags[a].Empty()) {
26       Apply(a.c(0), tags[a]);
27       Apply(a.c(1), tags[a]);
28       tags[a] = Tag();
29     }
30   }
31
32   void UpdateNode(seg_tree::point a) {
33     infos[a] = infos[a.c(0)].Unite(infos[a.c(1)]);
34   }
35
36   LazySegmentTree() : LazySegmentTree(0) {}
37   LazySegmentTree(int n_) : LazySegmentTree(vector<Info>(n_)) {}
38   LazySegmentTree(const vector<Info>& a) : n(int(a.size())) {
39     infos.resize(2 * n);
40     tags.resize(n);
41     layout = seg_tree::in_order_layout(n);
42     for (int i = 0; i < n; i++) {
43       infos[layout.get_point(i)] = a[i];
44     }
45     for (int i = n - 1; i >= 1; i--) {
46       UpdateNode(seg_tree::point(i));
47     }
48   }
49
50   void Modify(int l, int r, const Tag& t) {
51     auto rng = layout.get_range(l, r);
52     rng.for_parents_down([&](seg_tree::point a) {
53       DowndateNode(a);
54     });
55     rng.for_each([&](seg_tree::point a) {
56       Apply(a, t);
57     });
58     rng.for_parents_up([&](seg_tree::point a) {
59       UpdateNode(a);
60     });
61   }
62
63   void Set(int p, const Info& v) {
64     auto pt = layout.get_point(p);
65     pt.for_parents_down([&](seg_tree::point a) {
66       DowndateNode(a);
67     });
68     infos[pt] = v;
69     pt.for_parents_up([&](seg_tree::point a) {
70       UpdateNode(a);
71     });
72   }
73
74   Info Query(int l, int r) {
```

```
    auto rng = layout.get_range(l, r);
    rng.for_parents_down([&](seg_tree::point a) {
      DowndateNode(a);
    });
    Info res;
    rng.for_each_l_to_r([&](seg_tree::point a) {
      res = res.Unite(infos[a]);
    });
    return res;
  }

  Info Get(int p) {
    auto pt = layout.get_point(p);
    pt.for_parents_down([&](seg_tree::point a) {
      DowndateNode(a);
    });
    return infos[pt];
  }

  template<typename F>
  int MaxRight(int l, F f) {
    auto rng = layout.get_range(l, n);
    rng.for_parents_down([&](seg_tree::point a) {
      DowndateNode(a);
    });
    int res = n;
    Info sum;
    rng.for_each_l_to_r([&](seg_tree::point a) {
      if (res != n) {
        return;
      }
      auto new_sum = sum.Unite(infos[a]);
      if (f(new_sum)) {
        sum = new_sum;
        return;
      }
      while (a < n) {
        DowndateNode(a);
        new_sum = sum.Unite(infos[a.c(0)]);
        if (f(new_sum)) {
          sum = new_sum;
          a = a.c(1);
        } else {
          a = a.c(0);
        }
      }
      res = layout.get_node_bounds(a)[0];
    });
    return res;
  }

  template<typename F>
  int MinLeft(int r, F f) {
    auto rng = layout.get_range(0, r);
    rng.for_parents_down([&](seg_tree::point a) {
      DowndateNode(a);
    });
    int res = 0;
    Info sum;
    rng.for_each_r_to_l([&](seg_tree::point a) {
      if (res != 0) {
        return;
      }
      auto new_sum = infos[a].Unite(sum);
      if (f(new_sum)) {
        sum = new_sum;
        return;
      }
      while (a < n) {
        DowndateNode(a);
        new_sum = infos[a.c(1)].Unite(sum);
        if (f(new_sum)) {
          sum = new_sum;
          a = a.c(0);
        } else {
          a = a.c(1);
        }
      }
      res = layout.get_node_bounds(a)[1];
    });
    return res;
  }
};
```

6.7 simple.cpp

```cpp
template <typename Info>
class SimpleSegmentTree {
 public:
  int n;
  vector<Info> infos;
  seg_tree::in_order_layout layout;

  void UpdateNode(seg_tree::point a) {
    infos[a] = infos[a.c(0)].Unite(infos[a.c(1)]);
  }

  SimpleSegmentTree(int n_) : SimpleSegmentTree(vector<Info>(n_)) {}

  SimpleSegmentTree(const vector<Info>& a) : n(int(a.size())) {
    assert(n > 0);
    infos.resize(2 * n);
    layout = seg_tree::in_order_layout(n);
    for (int i = 0; i < n; i++) {
      infos[layout.get_point(i)] = a[i];
    }
    for (int i = n - 1; i >= 1; i--) {
      infos[i] = infos[2 * i].Unite(infos[2 * i + 1]);
    }
  }

  void Set(int p, const Info& v) {
    auto pt = layout.get_point(p);
    infos[pt] = v;
    pt.for_parents_up([&](seg_tree::point a) {
      UpdateNode(a);
    });
  }

  Info Query(int l, int r) {
    auto rng = layout.get_range(l, r);
    Info res;
    rng.for_each_l_to_r([&](seg_tree::point a) {
      res = res.Unite(infos[a]);
    });
    return res;
  }

  Info Get(int p) {
    auto pt = layout.get_point(p);
    return infos[pt];
  }

  template<typename F>
  int MaxRight(int l, F f) {
    auto rng = layout.get_range(l, n);
    int res = n;
    Info sum;
    rng.for_each_l_to_r([&](seg_tree::point a) {
      if (res != n) {
        return;
      }
      auto new_sum = sum.Unite(infos[a]);
      if (f(new_sum)) {
        sum = new_sum;
        return;
      }
      while (a < n) {
        new_sum = sum.Unite(infos[a.c(0)]);
        if (f(new_sum)) {
          sum = new_sum;
          a = a.c(1);
        } else {
          a = a.c(0);
        }
      }
      res = layout.get_node_bounds(a)[0];
    });
    return res;
  }

  template<typename F>
  int MinLeft(int r, F f) {
    auto rng = layout.get_range(0, r);
    int res = 0;
    Info sum;
    rng.for_each_r_to_l([&](seg_tree::point a) {
      if (res != 0) {
        return;
      }
      auto new_sum = infos[a].Unite(sum);
      if (f(new_sum)) {
```

```
        sum = new_sum;
        return;
      }
      while (a < n) {
        new_sum = infos[a.c(1)].Unite(sum);
        if (f(new_sum)) {
          sum = new_sum;
          a = a.c(0);
        } else {
          a = a.c(1);
        }
      }
      res = layout.get_node_bounds(a)[1];
    });
    return res;
  }
};
```

6.8 tag.cpp

```
struct Tag {
  ${0}... add = ...;

  bool ApplyTo(Info& a, [[maybe_unused]] int l, [[maybe_unused]] int r)
      const {
    ...
    return true;
  }

  void ApplyTo(Tag& t) const {
    ...
    t.add += add;
  }

  bool Empty() const {
    ...
    return add == 0;
  }
};
```

7 string

7.1 duval.cpp

```
template <typename T>
int duval(int n, const T &s) {
  assert(n >= 1);
  int i = 0, ans = 0;
  while (i < n) {
    ans = i;
    int j = i + 1, k = i;
    while (j < n + n && !(s[j % n] < s[k % n])) {
      if (s[k % n] < s[j % n]) {
        k = i;
      } else {
        k++;
      }
      j++;
    }
    while (i <= k) {
      i += j - k;
    }
  }
  return ans;
  // returns 0-indexed position of the least cyclic shift
}

template <typename T>
int duval(const T &s) {
  return duval((int) s.size(), s);
}
```

7.2 duval-prefixes.cpp

```
template <typename T>
vector<int> duval_prefixes(int n, const T &s) {
  vector<int> z = z_function(n, s);
  vector<int> ans(n, 0);
  int i = 0, pos = 0;
  while (i < n) {
    int j = i, k = i;
    while (j < n) {
      j++;
```

```
      if (j > pos) {
        if (z[k] <= pos - k && s[z[k]] < s[k + z[k]]) {
          int shift = (pos - i) / (j - k) * (j - k);
          ans[pos] = ans[pos - shift] + shift;
        } else {
          ans[pos] = i;
        }
        pos++;
      }
      if (s[k] < s[j]) k = i; else
      if (!(s[j] < s[k])) k++; else
      else break;
    }
    while (i <= k) {
      i += j - k;
    }
  }
  return ans;
  // returns 0-indexed positions of the least cyclic shifts of all prefixes
}

template <typename T>
vector<int> duval_prefixes(const T &s) {
  return duval_prefixes((int) s.size(), s);
}
```

7.3 hash61.cpp

```
struct hash61 {
  static const uint64_t md = (1LL << 61) - 1;
  static uint64_t step;
  static vector<uint64_t> pw;

  uint64_t addmod(uint64_t a, uint64_t b) const {
    a += b;
    if (a >= md) a -= md;
    return a;
  }

  uint64_t submod(uint64_t a, uint64_t b) const {
    a += md - b;
    if (a >= md) a -= md;
    return a;
  }

  uint64_t mulmod(uint64_t a, uint64_t b) const {
    uint64_t l1 = (uint32_t) a, h1 = a >> 32, l2 = (uint32_t) b, h2 = b >>
        32;
    uint64_t l = l1 * l2, m = l1 * h2 + l2 * h1, h = h1 * h2;
    uint64_t ret = (l & md) + (l >> 61) + (h << 3) + (m >> 29) + (m << 35 >>
         3) + 1;
    ret = (ret & md) + (ret >> 61);
    ret = (ret & md) + (ret >> 61);
    return ret - 1;
  }

  void ensure_pw(int sz) {
    int cur = (int) pw.size();
    if (cur < sz) {
      pw.resize(sz);
      for (int i = cur; i < sz; i++) {
        pw[i] = mulmod(pw[i - 1], step);
      }
    }
  }

  vector<uint64_t> pref;
  int n;

  template<typename T>
  hash61(const T& s) {
    n = (int) s.size();
    ensure_pw(n + 1);
    pref.resize(n + 1);
    pref[0] = 1;
    for (int i = 0; i < n; i++) {
      pref[i + 1] = addmod(mulmod(pref[i], step), s[i]);
    }
  }

  inline uint64_t operator()(const int from, const int to) const {
    assert(0 <= from && from <= to && to <= n - 1);
    return submod(pref[to + 1], mulmod(pref[from], pw[to - from + 1]));
  }
};

```

```
uint64_t hash61::step = (md >> 2) + rng() % (md >> 1);
vector<uint64_t> hash61::pw = vector<uint64_t>(1, 1);
```

7.4 kmp.cpp

```
template <typename T>
vector<int> kmp_table(int n, const T &s) {
  vector<int> p(n, 0);
  int k = 0;
  for (int i = 1; i < n; i++) {
    while (k > 0 && !(s[i] == s[k])) {
      k = p[k - 1];
    }
    if (s[i] == s[k]) {
      k++;
    }
    p[i] = k;
  }
  return p;
}

template <typename T>
vector<int> kmp_table(const T &s) {
  return kmp_table((int) s.size(), s);
}

template <typename T>
vector<int> kmp_search(int n, const T &s, int m, const T &w, const vector<
    int> &p) {
  assert(n >= 1 && (int) p.size() == n);
  vector<int> res;
  int k = 0;
  for (int i = 0; i < m; i++) {
    while (k > 0 && (k == n || !(w[i] == s[k]))) {
      k = p[k - 1];
    }
    if (w[i] == s[k]) {
      k++;
    }
    if (k == n) {
      res.push_back(i - n + 1);
    }
  }
  return res;
  // returns 0-indexed positions of occurrences of s in w
}

template <typename T>
vector<int> kmp_search(const T &s, const T &w, const vector<int> &p) {
  return kmp_search((int) s.size(), s, (int) w.size(), w, p);
}
```

7.5 manacher.cpp

```
template <typename T>
vector<int> manacher(int n, const T &s) {
  if (n == 0) {
    return vector<int>();
  }
  vector<int> res(2 * n - 1, 0);
  int l = -1, r = -1;
  for (int z = 0; z < 2 * n - 1; z++) {
    int i = (z + 1) >> 1;
    int j = z >> 1;
    int p = (i >= r ? 0 : min(r - i, res[2 * (l + r) - z]));
    while (j + p + 1 < n && i - p - 1 >= 0) {
      if (!(s[j + p + 1] == s[i - p - 1])) {
        break;
      }
      p++;
    }
    if (j + p > r) {
      l = i - p;
      r = j + p;
    }
    res[z] = p;
  }
  return res;
  // res[2 * i] = odd radius in position i
  // res[2 * i + 1] = even radius between positions i and i + 1
  // s = "abaa" -> res = {0, 0, 1, 0, 0, 1, 0}
  // in other words, for every z from 0 to 2 * n - 2:
  // calculate i = (z + 1) >> 1 and j = z >> 1
  // now there is a palindrome from i - res[z] to j + res[z]
  // (watch out for i > j and res[z] = 0)
}
```

```

template <typename T>
vector<int> manacher(const T &s) {
  return manacher((int) s.size(), s);
}
```

7.6 suffix-array.cpp

```
template <typename T>
vector<int> suffix_array(int n, const T &s, int char_bound) {
  vector<int> a(n);
  if (n == 0) {
    return a;
  }
  if (char_bound != -1) {
    vector<int> aux(char_bound, 0);
    for (int i = 0; i < n; i++) {
      aux[s[i]]++;
    }
    int sum = 0;
    for (int i = 0; i < char_bound; i++) {
      int add = aux[i];
      aux[i] = sum;
      sum += add;
    }
    for (int i = 0; i < n; i++) {
      a[aux[s[i]]++] = i;
    }
  } else {
    iota(a.begin(), a.end(), 0);
    sort(a.begin(), a.end(), [&s](int i, int j) { return s[i] < s[j]; });
  }
  vector<int> sorted_by_second(n);
  vector<int> ptr_group(n);
  vector<int> new_group(n);
  vector<int> group(n);
  group[a[0]] = 0;
  for (int i = 1; i < n; i++) {
    group[a[i]] = group[a[i - 1]] + (!(s[a[i]] == s[a[i - 1]]));
  }
  int cnt = group[a[n - 1]] + 1;
  int step = 1;
  while (cnt < n) {
    int at = 0;
    for (int i = n - step; i < n; i++) {
      sorted_by_second[at++] = i;
    }
    for (int i = 0; i < n; i++) {
      if (a[i] - step >= 0) {
        sorted_by_second[at++] = a[i] - step;
      }
    }
    for (int i = n - 1; i >= 0; i--) {
      ptr_group[group[a[i]]] = i;
    }
    for (int i = 0; i < n; i++) {
      int x = sorted_by_second[i];
      a[ptr_group[group[x]]++] = x;
    }
    new_group[a[0]] = 0;
    for (int i = 1; i < n; i++) {
      if (group[a[i]] != group[a[i - 1]]) {
        new_group[a[i]] = new_group[a[i - 1]] + 1;
      } else {
        int pre = (a[i - 1] + step >= n ? -1 : group[a[i - 1] + step]);
        int cur = (a[i] + step >= n ? -1 : group[a[i] + step]);
        new_group[a[i]] = new_group[a[i - 1]] + (pre != cur);
      }
    }
    swap(group, new_group);
    cnt = group[a[n - 1]] + 1;
    step <<= 1;
  }
  return a;
}

template <typename T>
vector<int> suffix_array(const T &s, int char_bound) {
  return suffix_array((int) s.size(), s, char_bound);
}

template <typename T>
vector<int> build_lcp(int n, const T &s, const vector<int> &sa) {
  assert((int) sa.size() == n);
  vector<int> pos(n);
  for (int i = 0; i < n; i++) {
```

```
    pos[sa[i]] = i;
  }
  vector<int> lcp(max(n - 1, 0));
  int k = 0;
  for (int i = 0; i < n; i++) {
    k = max(k - 1, 0);
    if (pos[i] == n - 1) {
      k = 0;
    } else {
      int j = sa[pos[i] + 1];
      while (i + k < n && j + k < n && s[i + k] == s[j + k]) {
        k++;
      }
      lcp[pos[i]] = k;
    }
  }
  return lcp;
}

template <typename T>
vector<int> build_lcp(const T &s, const vector<int> &sa) {
  return build_lcp((int) s.size(), s, sa);
}
```

7.7 z.cpp

```
template <typename T>
vector<int> z_function(int n, const T &s) {
  vector<int> z(n, n);
  int l = 0, r = 0;
  for (int i = 1; i < n; i++) {
    z[i] = (i > r ? 0 : min(r - i + 1, z[i - l]));
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      z[i]++;
    }
    if (i + z[i] - 1 > r) {
      l = i;
      r = i + z[i] - 1;
    }
  }
  return z;
}

template <typename T>
vector<int> z_function(const T &s) {
  return z_function((int) s.size(), s);
}
```

8 template

8.1 hc.cpp

```
/**
 *    author:  tourist
 *    created: $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:$CURRENT_MINUTE:$CURRENT_SECOND
**/
#include <bits/stdc++.h>

using namespace std;

#ifdef LOCAL
#include "algo/debug.h"
#else
#define debug(...) 42
#endif

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  int tt;
  cin >> tt;
  for (int qq = 1; qq <= tt; qq++) {
    cout << "Case␣#" << qq << ":␣";
    ${0}
  }
  return 0;
}
```

8.2 multithreaded.cpp

```
/**
 *    author:  tourist
 *    created: $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:$CURRENT_MINUTE:$CURRENT_SECOND
```

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

using namespace std;

class Solution {
  public:
  int k;
  string s, w;

  void readData() {

  }

  void solve(stringstream& out) {

  }
};

const int maxThreads = 8;
const int numTests = 1000;

stringstream out[numTests];
mutex mu;
int cur, tt;
thread threads[maxThreads];

void solutionRunner() {
  while (true) {
    Solution s;
    int id;
    mu.lock();
    if (cur >= tt) {
      mu.unlock();
      return;
    }
    id = cur;
    cur++;
    s.readData();
    mu.unlock();
    s.solve(out[id]);
  }
}

using namespace std::chrono;

long long now() {
  milliseconds ms = duration_cast<milliseconds>(system_clock::now().
      time_since_epoch());
  return ms.count();
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  long long start = now();
  cin >> tt;
  cur = 0;
  for (int i = 0; i < maxThreads; i++) {
    threads[i] = thread(solutionRunner);
  }
  for (int i = 0; i < maxThreads; i++) {
    threads[i].join();
  }
  for (int i = 0; i < tt; i++) {
    cout << "Case␣#" << i + 1 << ":␣" << '\n';
    cout << out[i].str();
  }
  cerr << "time␣=␣" << now() - start << "␣ms" << endl;
  return 0;
}
```

8.3 multithreaded2.cpp

```
/**
 *    author:  tourist
 *    created: $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:
     $CURRENT_MINUTE:$CURRENT_SECOND
**/
#undef _GLIBCXX_DEBUG

#include <bits/stdc++.h>

using namespace std;

#ifdef LOCAL
```

```
#include "algo/debug.h"
#else
#define debug(...) 42
#endif

mutex mut;
int qq = 0;
int tt;

class Solution {
 public:
  int test_id;

  explicit Solution(int test_id_) : test_id(test_id_) {}

  ${0}

  void ReadData() {

  }

  void Solve(stringstream& out) {

    mut.lock();
    debug(++qq, tt, test_id, clock());
    mut.unlock();
  }
};

int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  cin >> tt;
  vector<Solution> solutions;
  solutions.reserve(tt);
  for (int test_id = 0; test_id < tt; test_id++) {
    solutions.emplace_back(test_id);
    solutions.back().ReadData();
  }
  debug("input␣read,␣kicking␣off");
  vector<stringstream> outs(tt);
  vector<thread> threads;
  threads.reserve(tt);
  for (int test_id = 0; test_id < tt; test_id++) {
    threads.emplace_back(&Solution::Solve, &solutions[test_id], ref(outs[
        test_id]));
  }
  for (int test_id = 0; test_id < tt; test_id++) {
    threads[test_id].join();
  }
  for (int i = 0; i < tt; i++) {
    cout << "Case␣#" << i + 1 << ":␣" << outs[i].str();
  }
  return 0;
}
```

8.4 q1.cpp

```
/**
 *    author:  tourist
 *    created: $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:
     $CURRENT_MINUTE:$CURRENT_SECOND
**/
#include <bits/stdc++.h>

using namespace std;

#ifdef LOCAL
#include "algo/debug.h"
#else
#define debug(...) 42
#endif

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  ${0}
  return 0;
}
```

8.5 qt.cpp

```
/**
 *    author:  tourist
 *    created: $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:
     $CURRENT_MINUTE:$CURRENT_SECOND
```

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

using namespace std;

#ifdef LOCAL
#include "algo/debug.h"
#else
#define debug(...) 42
#endif

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  int tt;
  cin >> tt;
  while (tt--) {
    ${0}
  }
  return 0;
}
```