noya library

Competitive Programming Library

isaunoya

https://github.com/isaunoya/libra

Table of Contents

Contents

Data Structure

cartesian_tree.hpp

```cpp
#include <cassert>
#include <vector>

namespace noya {

/// @brief Cartesian tree where each node is the minimum of its subtree.
Equal values prefer the leftmost.
struct min_cartesian_tree {
  int n, rt;
  std::vector<int> par, lef, rig;
  template <class T> void build(int n_, T *as) {
    assert(n_ >= 1);
    n = n_;
    rt = 0;
    par.assign(n, -1);
    lef.assign(n, -1);
    rig.assign(n, -1);
    int top = 0;
    std::vector<int> stack(n, 0);
    for (int u = 1; u < n; ++u) {
      if (as[stack[top]] > as[u]) {
        for (; top >= 1 && as[stack[top - 1]] > as[u]; --top) {
        }
        if (top == 0) {
          rt = par[lef[u] = stack[top]] = u;
        } else {
          par[lef[u] = stack[top]] = u;
          rig[par[u] = stack[top - 1]] = u;
        }
        stack[top] = u;
      } else {
        rig[par[u] = stack[top]] = u;
        stack[++top] = u;
      }
    }
  }
  template <class T> void build(const T &as) { build(as.size(),
as.data()); }
};

/// @brief Cartesian tree where each node is the maximum of its subtree.
Equal values prefer the leftmost.
struct max_cartesian_tree {
  int n, rt;
  std::vector<int> par, lef, rig;
  template <class T> void build(int n_, T *as) {
    assert(n_ >= 1);
    n = n_;
    rt = 0;
    par.assign(n, -1);
    lef.assign(n, -1);
    rig.assign(n, -1);
    int top = 0;
    std::vector<int> stack(n, 0);
    for (int u = 1; u < n; ++u) {
      if (as[stack[top]] < as[u]) {
        for (; top >= 1 && as[stack[top - 1]] < as[u]; --top) {
        }
        if (top == 0) {
          rt = par[lef[u] = stack[top]] = u;
        } else {
          par[lef[u] = stack[top]] = u;
          rig[par[u] = stack[top - 1]] = u;
        }
        stack[top] = u;
      } else {
        rig[par[u] = stack[top]] = u;
        stack[++top] = u;
      }
    }
  }
  template <class T> void build(const T &as) { build(as.size(),
as.data()); }
};

} // namespace noya
```

consecutive_segment.hpp

```cpp
#include <map>
#include <tuple>
#include <vector>

namespace noya {

/// @brief Interval assignment container maintaining consecutive segments
of equal values.
template <class T> struct consecutive_segment {
  std::map<int, T> mp;
  consecutive_segment() {}
  consecutive_segment(int N, int v = 0) { mp[N] = v; }
  void split(int x) {
    auto it = mp.lower_bound(x);
    mp[x] = it->second;
  }

  /// @brief Get the value at position x.
  T get(int x) const {
    return mp.upper_bound(x)->second;
  }

  /// @brief Assign [l, r) = v.
  /// @return The original segments as vector of (left, right, value).
  std::vector<std::tuple<int, int, T>> assign(int l, int r, T v) {
    split(l);
    split(r);
    auto it = mp.find(l);
    T t = mp[l];
    std::vector<std::tuple<int, int, T>> segments;
    while (it->first != r) {
      auto nx = next(it);
      segments.push_back({it->first, nx->first, nx->second});
      it = mp.erase(it);
    }
    mp[l] = t;
    mp[r] = v;
    return segments;
  }
};

} // namespace noya
```

double_ended_heap.hpp

```cpp
#include <algorithm>
#include <functional>
#include <queue>
#include <vector>

namespace noya {
/// @brief Priority queue supporting lazy deletion of arbitrary elements.
template <class T, class C> struct removable_heap {
  std::priority_queue<T, std::vector<T>, C> p, q;
  void push(T x) { p.emplace(x); }
  void pop(T x) { q.emplace(x); }
  bool empty() {
    while (!p.empty() && !q.empty() && p.top() == q.top()) {
      p.pop();
      q.pop();
    }
    return p.empty();
  }
  T top() {
    while (!p.empty() && !q.empty() && p.top() == q.top()) {
      p.pop();
      q.pop();
    }
    assert(!p.empty());
    return p.top();
  }
};

/// @brief Double-ended heap supporting push, lazy pop, get_min, and
get_max.
template <class T> struct double_ended_heap {
  removable_heap<T, std::greater<>> min_heap;
  removable_heap<T, std::less<>> max_heap;

  void push(T x) {
    min_heap.push(x);
    max_heap.push(x);
  }
  void pop(T x) {
    min_heap.pop(x);
    max_heap.pop(x);
  }

  T get_min() { return min_heap.top(); }
  T get_max() { return max_heap.top(); }
};
} // namespace noya
```

fastset.hpp

```cpp
#include <assert.h>
#include <vector>
// https://github.com/hos-lyric/libra/blob/master/data_structure/fast_set.
cpp
namespace noya {
struct fast_set {
  // max{ceil(log_64(n)), 1}
  int log64N, n;
  std::vector<unsigned long long> a[6];
  explicit fast_set(int n_ = 0) : n(n_) {
    assert(n >= 0);
    int m = n ? n : 1;
    for (int d = 0;; ++d) {
      m = (m + 63) >> 6;
      a[d].assign(m, 0);
      if (m == 1) {
        log64N = d + 1;
        break;
      }
    }
  }
  bool empty() const { return !a[log64N - 1][0]; }
  bool contains(int x) const { return (a[0][x >> 6] >> (x & 63)) & 1; }
  void insert(int x) {
    for (int d = 0; d < log64N; ++d) {
      const int q = x >> 6, r = x & 63;
      a[d][q] |= 1ULL << r;
      x = q;
    }
  }
  void erase(int x) {
    for (int d = 0; d < log64N; ++d) {
      const int q = x >> 6, r = x & 63;
      if ((a[d][q] &= ~(1ULL << r)))
        break;
      x = q;
    }
  }
  /// @brief Find max element <= x, or -1 if none.
  int prev(int x) const {
```

```
    if (x > n - 1)
      x = n - 1;
    for (int d = 0; d <= log64N; ++d) {
      if (x < 0)
        break;
      const int q = x >> 6, r = x & 63;
      const unsigned long long lower = a[d][q] << (63 - r);
      if (lower) {
        x -= __builtin_clzll(lower);
        for (int e = d; --e >= 0;)
          x = x << 6 | (63 - __builtin_clzll(a[e][x]));
        return x;
      }
      x = q - 1;
    }
    return -1;
  }
  /// @brief Find min element >= x, or n if none.
  int next(int x) const {
    if (x < 0)
      x = 0;
    for (int d = 0; d < log64N; ++d) {
      const int q = x >> 6, r = x & 63;
      if (static_cast<unsigned>(q) >= a[d].size())
        break;
      const unsigned long long upper = a[d][q] >> r;
      if (upper) {
        x += __builtin_ctzll(upper);
        for (int e = d; --e >= 0;)
          x = x << 6 | __builtin_ctzll(a[e][x]);
        return x;
      }
      x = q + 1;
    }
    return n;
  }
};

template <class T> struct painter {
  int n;
  fast_set s;
  std::vector<T> ts;
  painter() {}
  painter(int n_, const T &t) : n(n_), s(n + 1), ts(n + 2, t) {}
  template <class F> void paint(int a, int b, const T &t, F f) {
    assert(0 <= a);
    assert(a <= b);
    assert(b <= n);
    if (a == b)
      return;
    // auto it = this->lower_bound(a);
    int c = s.next(a);
    if (b < c) {
      f(a, b, ts[c]);
      s.insert(a);
      ts[a] = ts[c];
      s.insert(b);
      ts[b] = t;
    } else if (a < c) {
      const T ta = ts[c];
      int k = a;
      for (; c <= b; s.erase(c), c = s.next(c)) {
        f(k, c, ts[c]);
        k = c;
      }
      if (k < b) {
        f(k, b, ts[c]);
      }
      s.insert(a);
      ts[a] = ta;
      s.insert(b);
      ts[b] = t;
    } else {
      c = s.next(c + 1);
      int k = a;
      for (; c <= b; s.erase(c), c = s.next(c)) {
        f(k, c, ts[c]);
        k = c;
      }
      if (k < b) {
        f(k, b, ts[c]);
      }
      s.insert(b);
      ts[b] = t;
    }
  }
  void paint(int a, int b, const T &t) {
    paint(a, b, t, [&](int, int, const T &) -> void {});
  }
  T get(int k) const {
    assert(0 <= k);
    assert(k < n);
    return ts[s.next(k + 1)];
  }
};
} // namespace noya
```

hashmap.hpp

```
#include <chrono>
#include <unordered_map>

namespace noya {
/// @brief Randomized splitmix64 hash to prevent hash collision attacks.
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);
  }
};

/// @brief Unordered map with anti-hash randomized hash function.
template <typename K, typename V, typename Hash = splitmix64_hash>
using HashMap = std::unordered_map<K, V, Hash>;

} // namespace noya
```

persistent_segtree.hpp

```
#include <algorithm>
#include <vector>

namespace noya {
/// @brief Persistent segment tree supporting 2D rectangle sum and k-th
queries.
template <class T> struct persistent_segtree {
  std::vector<int> rt;
  std::vector<int> ls, rs;
  int N, M;
  std::vector<T> Xs, Ys;

  std::vector<T> sum;
  int cnt, C;

  persistent_segtree() {}

  persistent_segtree(std::vector<std::array<T, 3>> &points) {
    if (!points.empty())
      build(points);
  }

  void update(int pre, int &p, int l, int r, int x, T v) {
    p = cnt++;
    ls[p] = ls[pre];
    rs[p] = rs[pre];
    sum[p] = sum[pre] + v;
    if (l + 1 == r)
      return;
    int m = (l + r) / 2;
    if (x < m)
      update(ls[pre], ls[p], l, m, x, v);
    else
      update(rs[pre], rs[p], m, r, x, v);
  }

  int query_kth(int L, int R, int l, int r, T k) {
    if (l + 1 == r) {
      return l;
    }
    int m = (l + r) / 2;
    T val = sum[ls[R]] - sum[ls[L]];
    if (val >= k) {
      return query_kth(ls[L], ls[R], l, m, k);
    } else {
      return query_kth(rs[L], rs[R], m, r, k - val);
    }
  }

  T query_prefix(int p, int l, int r, int x) {
    if (!p)
      return 0;

    if (l + 1 == r)
      return sum[p];

    int m = (l + r) / 2;
    if (x < m)
      return query_prefix(ls[p], l, m, x);
    else
      return sum[ls[p]] + query_prefix(rs[p], m, r, x);
  }

  T query_single(int p, int l, int r, int x) {
    if (!p)
      return 0;

    if (l + 1 == r)
      return sum[p];

    int m = (l + r) / 2;
    if (x < m)
      return query_single(ls[p], l, m, x);
    else
      return query_single(rs[p], m, r, x);
  }

  T query(int L, int R, int ql, int qr, int l, int r) {
    if (ql <= l && r <= qr)
      return sum[R] - sum[L];

    int m = (l + r) / 2;
    T res = 0;
    if (ql < m)
      res += query(ls[L], ls[R], ql, qr, l, m);
    if (qr > m)
      res += query(rs[L], rs[R], ql, qr, m, r);

    return res;
  }

  /// @brief Query the prefix sum up to coordinate (r, u) in the original
space.
  T prod(T r, T u) {
    r = std::lower_bound(Xs.begin(), Xs.end(), r) - Xs.begin();
    u = std::lower_bound(Ys.begin(), Ys.end(), u) - Ys.begin();

    assert(0 <= r && r <= N);
    assert(0 <= u && u <= M);

    if (u > 0)
```

```
      return query_prefix(rt[r], 0, M, u - 1);
    else
      return 0;
  }

  /// @brief Query the sum of weights in rectangle [l, r) x [d, u).
  T prod(T l, T r, T d, T u) {
    l = std::lower_bound(Xs.begin(), Xs.end(), l) - Xs.begin();
    r = std::lower_bound(Xs.begin(), Xs.end(), r) - Xs.begin();

    d = std::lower_bound(Ys.begin(), Ys.end(), d) - Ys.begin();
    u = std::lower_bound(Ys.begin(), Ys.end(), u) - Ys.begin();

    assert(0 <= l && l <= r && r <= N);
    assert(0 <= d && d <= u && u <= M);

    if (l == r)
      return 0;
    else if (d == u)
      return 0;
    else
      return query(rt[l], rt[r], d, u, 0, M);
  }

  /// @brief Return the k-th smallest Y-coordinate among points with X in
[l, r).
  T kth(T l, T r, T k) {
    l = std::lower_bound(Xs.begin(), Xs.end(), l) - Xs.begin();
    r = std::lower_bound(Xs.begin(), Xs.end(), r) - Xs.begin();
    assert(0 <= l && l <= r && r <= N);

    if (sum[rt[r]] - sum[rt[l]] < k) {
      return std::numeric_limits<T>::max();
    }

    int res = query_kth(rt[l], rt[r], 0, M, k);
    return Ys[res];
  }

  /// @brief Build the persistent segment tree from weighted 2D points {x,
y, w}.
  void build(std::vector<std::array<T, 3>> points) {
    C = int(points.size()) * 30;
    cnt = 1;

    ls.assign(C, 0);
    rs.assign(C, 0);
    sum.assign(C, 0);

    Xs.clear();
    Ys.clear();

    for (auto &[x, y, w] : points) {
      Xs.push_back(x);
      Ys.push_back(y);
    }

    std::sort(Xs.begin(), Xs.end());
    std::sort(Ys.begin(), Ys.end());
    Xs.erase(std::unique(Xs.begin(), Xs.end()), Xs.end());
    Ys.erase(std::unique(Ys.begin(), Ys.end()), Ys.end());
    N = int(Xs.size());

    std::vector<std::vector<std::array<T, 2>>> add(N);
    for (auto &[x, y, w] : points) {
      x = std::lower_bound(Xs.begin(), Xs.end(), x) - Xs.begin();
      y = std::lower_bound(Ys.begin(), Ys.end(), y) - Ys.begin();
      add[x].push_back({y, w});
    }

    M = int(Ys.size());
    rt.assign(N + 1, 0);

    rt[0] = cnt++;
    for (int i = 0; i < N; i++) {
      rt[i + 1] = rt[i];
      for (auto &[y, w] : add[i]) {
        int new_rt = 0;
        update(rt[i + 1], new_rt, 0, M, y, w);
        rt[i + 1] = new_rt;
      }
    }

    std::vector<std::vector<std::array<T, 2>>>().swap(add);
  }
  /// @brief @return vector of (ls, rs, len, value) for all nodes.
  std::vector<std::array<int, 4>> all_nodes() {
    std::vector<std::array<int, 4>> nodes(cnt, std::array<int, 4>{});
    std::vector<bool> vis(cnt);
    vis[0] = true;
    for (int i = 0; i <= N; i++) {
      auto dfs = [&](auto &dfs, int p, int l, int r) -> void {
        if (vis[p])
          return;
        vis[p] = true;
        nodes[p] = {ls[p], rs[p], r - l, sum[p]};
        int m = (l + r) / 2;
        dfs(dfs, ls[p], l, m);
        dfs(dfs, rs[p], m, r);
      };
      dfs(dfs, rt[i], 0, M);
    }
    return nodes;
  }
};
} // namespace noya
```

point_add_range_sum.hpp

```
#include "atcoder/fenwicktree.hpp"
#include <cmath>
#include <vector>

namespace noya {
```

```
template <class T> struct block {
  int V, sqrtV;

  block() {}
  block(const int &_V) {
    if (_V > 0) {
      build(_V);
    }
  }

  std::vector<T> point, blo;
  void build(const int &_V) {
    V = _V;
    sqrtV = sqrt(V);
    point.assign(V, 0);
    blo.assign(V / sqrtV + 1, 0);
  }

  void add(int x, T v) {
    assert(0 <= x && x < V);
    int bel = x / sqrtV;
    blo[bel] += v;
    point[x] += v;
  }

  T query(int x) const {
    assert(0 <= x && x <= V);
    T res = 0;
    int bel = x / sqrtV;
    for (int i = 0; i < bel; i++)
      res += blo[i];
    int start = bel * sqrtV;
    int end = x;
    for (int i = start; i < end; i++)
      res += point[i];
    return res;
  }

  /// @brief Sum of [l, r).
  T prod(int l, int r) const {
    assert(0 <= l && l <= r && r <= V);
    return query(r) - query(l);
  }
};

template <class T> struct fenwick : atcoder::fenwick_tree<T> {
  using atcoder::fenwick_tree<T>::fenwick_tree;
  using atcoder::fenwick_tree<T>::add;
  T query(int x) { return this->sum(0, x); }
  T prod(int l, int r) { return this->sum(l, r); }
};

} // namespace noya
```

point_add_rec_sum.hpp

```
#include "noya/persistent_segtree.hpp"

namespace noya {
/// @brief Dynamic point-add rectangle-sum with semi-offline rebuilds.
template <class T> struct dynamic_point_add_rectangle_sum {
  persistent_segtree<T> seg;

  std::vector<std::array<T, 3>> weighted_points;
  std::vector<std::array<T, 3>> buckets;

  dynamic_point_add_rectangle_sum() {}
  dynamic_point_add_rectangle_sum(std::vector<std::array<T, 3>> &points) {
    buckets = points;
    build();
  };

  const int B = 6000;
  /// @brief Add a weighted point at (x, y) with weight w.
  void add_points(T x, T y, T w) {
    buckets.push_back({x, y, w});
    if (int(buckets.size()) >= B) {
      build();
    }
  }

  void build() {
    if (!buckets.empty()) {
      for (auto &[x, y, w] : buckets) {
        weighted_points.push_back({x, y, w});
      }
      buckets.clear();
    }
    seg.build(weighted_points);
  }

  /// @brief Query the sum of weights in rectangle [l, r) x [d, u).
  T query(T l, T r, T d, T u) {
    T ans = 0;
    ans = seg.prod(l, r, d, u);
    for (auto &[x, y, w] : buckets) {
      if (l <= x && x < r && d <= y && y < u) {
        ans += w;
      }
    }
    return ans;
  }
};
} // namespace noya
```

point_set_range_freq.hpp

```
#include <algorithm>
#include <vector>
namespace noya {
```

```
template <class T> struct point_set_range_frequency {
  point_set_range_frequency() {}

  const int B = 4000;
  int N;
  std::vector<T> A;
  point_set_range_frequency(std::vector<T> &_A) {
    A = _A;
    if (!A.empty())
      build();
  }

  std::vector<std::pair<int, T>> point_set;
  void set(int x, T v) {
    assert(0 <= x && x < N);
    for (auto &[p, q] : point_set) {
      if (p == x) {
        q = v;
        return;
      }
    }
    point_set.push_back({x, v});
    if (int(point_set.size()) > B) {
      build();
    }
  }

  std::vector<std::vector<int>> pos;
  std::vector<T> Xs;

  // frequency [l, r)
  /// @brief Count occurrences of x in [l, r).
  int query(int l, int r, T x) {
    assert(0 <= l && l <= r && r <= N);
    int ans = 0;
    if (std::binary_search(Xs.begin(), Xs.end(), x)) {
      int a = std::lower_bound(Xs.begin(), Xs.end(), x) - Xs.begin();
      ans += std::lower_bound(pos[a].begin(), pos[a].end(), r) -
             std::lower_bound(pos[a].begin(), pos[a].end(), l);
    }
    for (auto &[p, q] : point_set) {
      if (l <= p && p < r) {
        if (A[p] == x) {
          ans--;
        }
        if (q == x) {
          ans++;
        }
      }
    }
    return ans;
  }

  void build() {
    if (!point_set.empty()) {
      for (auto &[x, v] : point_set) {
        A[x] = v;
      }
      point_set.clear();
    }

    Xs.clear();
    for (auto &a : A)
      Xs.push_back(a);
    std::sort(Xs.begin(), Xs.end());
    Xs.erase(std::unique(Xs.begin(), Xs.end()), Xs.end());

    N = int(A.size());
    pos.assign(Xs.size(), {});
    for (int i = 0; i < N; i++) {
      auto a = A[i];
      a = std::lower_bound(Xs.begin(), Xs.end(), a) - Xs.begin();
      pos[a].push_back(i);
    }
  }
};
} // namespace noya
```

range_add_point_get.hpp

```
#include "noya/point_add_range_sum.hpp"

namespace noya {

template <class T, class C = noya::fenwick<T>> struct range_add_point_get {
  int N;
  C ST;
  range_add_point_get(int _N) : ST(_N), N(_N) {}

  /// @brief Add v to [l, r).
  void range_add(int l, int r, T v) {
    ST.add(l, v);
    if (r + 1 < N) {
      ST.add(r + 1, -v);
    }
  }

  T point_get(int x) { return ST.prod(0, x + 1); }
};

} // namespace noya
```

rec_add_point_get.hpp

```
#include "noya/persistent_segtree.hpp"
#include "noya/rectangle_sum.hpp"

namespace noya {

/// @brief Offline rectangle-add point-get: add weight w to all points in
[l, r) x [d, u).
template <class T, class C = noya::fenwick<T>>
std::vector<T>
static_rectangle_add_point_get(std::vector<std::array<int, 5>> areas,
                               std::vector<std::array<int, 2>> points) {
  std::vector<std::array<T, 3>> weighted_points;
  for (auto &[l, r, d, u, w] : areas) {
    weighted_points.push_back({l, d, w});
    weighted_points.push_back({l, u, -w});

    weighted_points.push_back({r, d, -w});
    weighted_points.push_back({r, u, w});
  }
  std::vector<std::array<T, 4>> queries;
  for (auto &[x, y] : points) {
    queries.push_back({0, x + 1, 0, y + 1});
  }
  return rectangle_sum<T, C>(weighted_points, queries);
}

/// @brief Dynamic rectangle-add point-get with semi-offline rebuilds.
template <class T> struct dynamic_rectangle_add_point_get {
  persistent_segtree<T> seg;

  std::vector<std::array<T, 3>> weighted_points;
  std::vector<std::array<T, 5>> buckets;

  dynamic_rectangle_add_point_get() {}

  dynamic_rectangle_add_point_get(std::vector<std::array<T, 5>> &areas) {
    buckets = areas;
    build();
  };

  const int B = 6000;
  /// @brief Add weight w to all points in rectangle [l, r) x [d, u).
  void add_rectangle(T l, T r, T d, T u, T w) {
    buckets.push_back({l, r, d, u, w});
    if (int(buckets.size()) >= B) {
      build();
    }
  }

  void build() {
    if (!buckets.empty()) {
      for (auto &[l, r, d, u, w] : buckets) {
        weighted_points.push_back({l, d, w});
        weighted_points.push_back({l, u, -w});
        weighted_points.push_back({r, d, -w});
        weighted_points.push_back({r, u, w});
      }
      buckets.clear();
    }
    seg.build(weighted_points);
  }

  /// @brief Query the accumulated weight at point (x, y).
  T query(int x, int y) {
    T ans = 0;
    ans = seg.prod(x + 1, y + 1);
    for (auto &[l, r, d, u, w] : buckets) {
      if (l <= x && x < r && d <= y && y < u) {
        ans += w;
      }
    }
    return ans;
  }
};

} // namespace noya
```

rectangle_sum.hpp

```
#include "noya/point_add_range_sum.hpp"

#include <algorithm>
#include <array>

namespace noya {

/// @brief Offline rectangle sum. Points (x, y, weight), queries [l, r) x
[d, u).
template <class T, class C = noya::fenwick<T>>
std::vector<T> rectangle_sum(std::vector<std::array<int, 3>> points,
                             std::vector<std::array<int, 4>> queries) {
  std::vector<int> Xs;
  std::vector<int> Ys;
  for (auto &[x, y, z] : points) {
    Xs.push_back(x);
    Ys.push_back(y);
  }
  for (auto &[l, r, d, u] : queries) {
    Xs.push_back(l);
    Xs.push_back(r);
    Ys.push_back(d);
    Ys.push_back(u);
  }

  std::sort(Xs.begin(), Xs.end());
  Xs.erase(std::unique(Xs.begin(), Xs.end()), Xs.end());

  std::sort(Ys.begin(), Ys.end());
  Ys.erase(std::unique(Ys.begin(), Ys.end()), Ys.end());

  for (auto &[x, y, z] : points) {
    x = std::lower_bound(Xs.begin(), Xs.end(), x) - Xs.begin();
    y = std::lower_bound(Ys.begin(), Ys.end(), y) - Ys.begin();
  }

  for (auto &[l, r, d, u] : queries) {
    l = std::lower_bound(Xs.begin(), Xs.end(), l) - Xs.begin();
    r = std::lower_bound(Xs.begin(), Xs.end(), r) - Xs.begin();
    d = std::lower_bound(Ys.begin(), Ys.end(), d) - Ys.begin();
    u = std::lower_bound(Ys.begin(), Ys.end(), u) - Ys.begin();
  }
```

```
  int X = int(Xs.size());
  int Y = int(Ys.size());
  C F(Y);
  std::vector<std::vector<std::array<int, 3>>> Q(X + 1);
  for (int i = 0; i < int(queries.size()); i++) {
    auto &[l, r, d, u] = queries[i];
    Q[r].push_back({i, d, u});
    Q[l].push_back({~i, d, u});
  }

  std::vector<std::vector<std::array<int, 2>>> A(X + 1);
  for (auto &[x, y, z] : points) {
    A[x].push_back({y, z});
  }

  std::vector<T> ans(queries.size());
  for (int x = 0; x < X; x++) {
    for (auto &[y, z] : A[x]) {
      F.add(y, z);
    }
    for (auto &[i, d, u] : Q[x + 1]) {
      if (i >= 0)
        ans[i] += F.prod(d, u);
      else
        ans[~i] -= F.prod(d, u);
    }
  }
  return ans;
}

} // namespace noya
```

sliding_window_aggregation.hpp

```
#include <algorithm>
#include <cassert>
#include <functional>
#include <vector>

namespace noya {
#if __cplusplus >= 201703L

/// @brief Sliding window aggregation (SWAG) supporting push_back,
pop_front, and monoid product queries.
template <class S, auto op, auto e> struct swag {
  static_assert(std::is_convertible_v<decltype(op), std::function<S(S,
S)>>,
                "op must work as S(S, S)");
  static_assert(std::is_convertible_v<decltype(e), std::function<S()>>,
                "e must work as S()");

#else

/// @brief Sliding window aggregation (SWAG) supporting push_back,
pop_front, and monoid product queries.
template <class S, S (*op)(S, S), S (*e)()> struct swag {

#endif
  int sz = 0;
  std::vector<S> dat;
  std::vector<S> sum_l;
  S sum_r;

  swag() {
    sum_l = {e()};
    sum_r = e();
  }

  int size() { return sz; }
  /// @brief Push element x to the back of the window.
  void push(S x) {
    ++sz;
    sum_r = op(sum_r, x);
    dat.push_back(x);
  }

  /// @brief Pop the front element from the window.
  void pop() {
    assert(0 < sz);
    --sz;
    sum_l.pop_back();
    if ((int)sum_l.size() == 0) {
      sum_l = {e()};
      sum_r = e();
      while ((int)dat.size() > 1) {
        sum_l.push_back(op(dat.back(), sum_l.back()));
        dat.pop_back();
      }
      dat.pop_back();
    }
  }

  S lprod() { return sum_l.back(); }
  S rprod() { return sum_r; }

  /// @brief Return the aggregate product of all elements in the window.
  S prod() { return op(sum_l.back(), sum_r); }
};
} // namespace noya
```

sparse_table.hpp

```
#include <cassert>
#include <vector>

namespace noya {
template <class T, auto op, T e = T()> struct sparse_table {
  static int highest_bit(unsigned x) {
    return x == 0 ? -1 : 31 - __builtin_clz(x);
  }
  int n = 0;
  std::vector<std::vector<T>> ST;

  sparse_table(const std::vector<T> &values = {}) {
    if (!values.empty())
      build(values);
  }

  void build(const std::vector<T> &values) {
    n = int(values.size());
    int levels = highest_bit(n) + 1;
    ST.resize(levels);

    for (int k = 0; k < levels; k++)
      ST[k].resize(n - (1 << k) + 1);

    if (n > 0)
      ST[0] = values;

    for (int k = 1; k < levels; k++)
      for (int i = 0; i <= n - (1 << k); i++)
        ST[k][i] = op(ST[k - 1][i], ST[k - 1][i + (1 << (k - 1))]);
  }
  /// @brief Query op over [l, r).
  T prod(int l, int r) const {
    assert(0 <= l && l <= r && r <= n);
    if (l == r) {
      return e;
    }
    int L = highest_bit(r - l);
    return op(ST[L][l], ST[L][r - (1 << L)]);
  }
};
} // namespace noya
```

tag_container.hpp

```
#include "noya/hashmap.hpp"
#include <algorithm>
#include <cassert>
#include <functional>
#include <map>
#include <vector>

namespace noya {
/// @brief HashMap with a global additive tag, supporting mergeable insert
and bulk offset.
template <class K, class V, auto op> struct tag_HashMap {
  HashMap<K, V> S;
  V tag = V();
  /// @brief Insert or merge a key-value pair (value is the true value
before tag).
  void insert(K key, V value) {
    auto it = S.find(key);
    if (it == S.end()) {
      S[key] = value - tag;
    } else {
      S[key] = op(it->second, value - tag);
    }
  }

  void erase(K key) { S.erase(key); }
  int size() const { return (int)S.size(); }

  /// @brief Merge another tag_HashMap into this one (small-to-large).
  void join(tag_HashMap<K, V, op> &other) {
    if (size() > other.size()) {
      for (auto &[key, value] : other.S) {
        insert(key, value + other.tag);
      }
    } else {
      S.swap(other.S);
      swap(tag, other.tag);
      for (auto &[key, value] : other.S) {
        insert(key, value + other.tag);
      }
    }
  }

  /// @brief Add x to the global tag (offsets all stored values).
  void add(V x) { tag += x; }

  // return the true value
  V operator[](const K &x) {
    if (S.find(x) == S.end()) {
      return V();
    } else {
      V value = S[x];
      return value + tag;
    }
  }
};

} // namespace noya
```

Graph

bipartite_matching.hpp

```
#include <algorithm>
#include <bitset>
#include <cassert>
#include <random>
#include <vector>
```

```cpp
namespace noya {
/// @brief Dense bipartite matching using bitset-accelerated augmenting
paths.
template <typename BS> struct BipartiteMatching_Dense {
  int N1, N2;
  std::vector<BS> &adj;
  std::vector<int> match_1, match_2;
  std::vector<int> que;
  std::vector<int> prev;
  BS vis;

  BipartiteMatching_Dense(std::vector<BS> &adj, int N1, int N2)
      : N1(N1), N2(N2), adj(adj), match_1(N1, -1), match_2(N2, -1) {
    for (int s = 0; s < N1; s++)
      bfs(s);
  }

  void bfs(int s) {
    if (match_1[s] != -1)
      return;
    que.resize(N1), prev.resize(N1);
    int l = 0, r = 0;
    vis.set(), prev[s] = -1;

    que[r++] = s;
    while (l < r) {
      int u = que[l++];
      BS cand = vis & adj[u];
      for (int v = cand._Find_first(); v < N2; v = cand._Find_next(v)) {
        vis[v] = 0;
        if (match_2[v] != -1) {
          que[r++] = match_2[v];
          prev[match_2[v]] = u;
          continue;
        }
        int a = u, b = v;
        while (a != -1) {
          int t = match_1[a];
          match_1[a] = b, match_2[b] = a, a = prev[a], b = t;
        }
        return;
      }
    }
    return;
  }

  /// @brief Return all matched pairs (left_vertex, right_vertex).
  std::vector<std::pair<int, int>> matching() {
    std::vector<std::pair<int, int>> res;
    for (int v = 0; v < N1; v++)
      if (match_1[v] != -1)
        res.emplace_back(v, match_1[v]);
    return res;
  }

  /// @brief Compute minimum vertex cover as (left_vertices,
right_vertices).
  std::pair<std::vector<int>, std::vector<int>> vertex_cover() {
    std::vector<int> que(N1);
    int l = 0, r = 0;
    vis.set();
    std::vector<bool> done(N1);
    for (int i = 0; i < N1; i++) {
      if (match_1[i] == -1)
        done[i] = 1, que[r++] = i;
    }
    while (l < r) {
      int a = que[l++];
      BS cand = adj[a] & vis;
      for (int b = cand._Find_first(); b < N2; b = cand._Find_next(b)) {
        vis[b] = 0;
        int to = match_2[b];
        assert(to != -1);
        if (!done[to])
          done[to] = 1, que[r++] = to;
      }
    }
    std::vector<int> left, right;
    for (int i = 0; i < N1; i++)
      if (!done[i])
        left.emplace_back(i);
    for (int i = 0; i < N2; i++)
      if (!vis[i])
        right.emplace_back(i);
    return {left, right};
  }
};

/// @brief Hopcroft-Karp maximum bipartite matching in O(E sqrt(V)).
struct HopcroftKarp {
  std::vector<int> g, l, r;
  int ans;
  HopcroftKarp(int n, int m, std::vector<std::pair<int, int>> &e)
      : g(e.size()), l(n, -1), r(m, -1), ans(0) {
    std::mt19937 rng(0);
    std::shuffle(e.begin(), e.end(), rng);
    std::vector<int> deg(n + 1), a, p, q(n);
    for (auto &[x, y] : e)
      deg[x]++;
    for (int i = 1; i <= n; i++)
      deg[i] += deg[i - 1];
    for (auto &[x, y] : e)
      g[--deg[x]] = y;
    for (bool match = true; match;) {
      a.assign(n, -1);
      p.assign(n, -1);
      int t = 0;
      match = false;
      for (int i = 0; i < n; i++)
        if (l[i] == -1)
          q[t++] = a[i] = p[i] = i;
      for (int i = 0; i < t; i++) {
        int x = q[i];
        if (~l[a[x]])
          continue;
        for (int j = deg[x]; j < deg[x + 1]; j++) {
          int y = g[j];
          if (r[y] == -1) {
            while (~y) {
              r[y] = x;
              std::swap(l[x], y);
              x = p[x];
            }
            match = true;
            ans++;
            break;
          }
          if (p[r[y]] == -1) {
            q[t++] = y = r[y];
            p[y] = x;
            a[y] = a[x];
          }
        }
      }
    }
  }
};

} // namespace noya
```

cycle.hpp

```cpp
#include "atcoder/dsu.hpp"
#include <numeric>
#include <vector>

namespace noya {
/// @brief Detect whether a directed graph contains a cycle.
bool cycle_detection_directed(const std::vector<std::vector<int>> &g) {
  int N = int(g.size());
  std::vector<int> deg(N);
  for (auto &gi : g) {
    for (auto &v : gi) {
      deg[v] += 1;
    }
  }
  std::vector<int> que;
  for (int i = 0; i < N; i++) {
    if (deg[i] == 0) {
      que.push_back(i);
    }
  }

  for (int i = 0; i < int(que.size()); i++) {
    int u = que[i];
    for (auto v : g[u]) {
      deg[v] -= 1;
      if (!deg[v]) {
        que.push_back(v);
      }
    }
  }
  return std::accumulate(deg.begin(), deg.end(), 0) > 0;
}

/// @brief Detect whether an undirected edge list contains a cycle.
bool cycle_detection_undirected(const std::vector<std::pair<int, int>>
&edge) {
  int N = 0;
  for (auto &[a, b] : edge) {
    N = std::max(N, a);
    N = std::max(N, b);
  }
  N++;
  atcoder::dsu f(N);
  for (auto &[a, b] : edge) {
    if (f.same(a, b)) {
      return true;
    }
    f.merge(a, b);
  }
  return false;
}
}
```

dsu_on_tree.hpp

```cpp
#include "noya/heavy_light_decomposition.hpp"

namespace noya {

/// @brief DSU on tree. add(u) adds vertex u, query(u, x) queries
contribution of x to u, del(u) removes vertex u.
void dsu_on_tree(std::vector<std::vector<int>> &g, auto &&add, auto
&&query,
                 auto &&del) {
  hld hld_(g);
  auto dfs = [&](auto &dfs, int u, bool heavy) -> void {
    for (auto v : g[u]) {
      if (v != hld_.fa[u] && v != hld_.son[u]) {
        dfs(dfs, v, false);
      }
    }
    if (hld_.son[u] != -1)
      dfs(dfs, hld_.son[u], true);

    for (auto v : g[u]) {
      if (v != hld_.fa[u] && v != hld_.son[u]) {
        auto [l, r] = hld_.subtree(v);
        for (int i = l; i < r; i++) {
          int x = hld_.tour_list[i];
          query(u, x);
        }
        for (int i = l; i < r; i++) {
          int x = hld_.tour_list[i];
          add(x);
        }
```

```
      }
    }

    query(u, u);
    add(u);

    if (!heavy) {
      auto [l, r] = hld_.subtree(u);
      for (int i = l; i < r; i++) {
        int x = hld_.tour_list[i];
        del(x);
      }
    }
  };
  dfs(dfs, 0, true);
}
} // namespace noya
```

heavy_light_decomposition.hpp

```
#include <algorithm>
#include <array>
#include <cassert>
#include <vector>

namespace noya {
/// @brief Heavy-light decomposition for path and subtree queries on trees.
struct hld {
  hld() {}
  std::vector<std::vector<int>> G;
  int n;
  std::vector<int> dfn, siz, son, top, d, fa;
  int idx;
  void dfs(int u, int p) {
    siz[u] = 1;
    for (auto v : G[u])
      if (v != p) {
        fa[v] = u;
        d[v] = d[u] + 1;
        dfs(v, u), siz[u] += siz[v];
        if (son[u] == -1 || siz[v] > siz[son[u]])
          son[u] = v;
      }
  }

  std::vector<int> tour_list;
  void dfs2(int u, int t) {
    top[u] = t, dfn[u] = idx++;
    tour_list.push_back(u);
    if (son[u] != -1)
      dfs2(son[u], t);
    for (auto v : G[u])
      if (top[v] == -1)
        dfs2(v, v);
  }

  /// @brief Return the k-th ancestor of node a, or -1 if k > depth(a).
  int get_kth_ancestor(int a, int k) const {
    if (k > d[a])
      return -1;

    int goal = d[a] - k;
    while (d[top[a]] > goal)
      a = fa[top[a]];

    int pos = dfn[a] - (d[a] - goal);
    return tour_list[pos];
  }

  /// @brief Return the k-th node (0-indexed) on the path from a to b, or
-1 if out of range.
  int get_kth_node_on_path(int a, int b, int k) const {
    int anc = lca(a, b);
    int first_half = d[a] - d[anc];
    int second_half = d[b] - d[anc];

    if (k < 0 || k > first_half + second_half)
      return -1;

    if (k < first_half)
      return get_kth_ancestor(a, k);
    else
      return get_kth_ancestor(b, first_half + second_half - k);
  }

  hld(const std::vector<std::vector<int>> &g = {}, const int &root = 0) {
    if (!g.empty())
      build(g, root);
  }

  void build(const std::vector<std::vector<int>> &g = {}, const int &root =
0) {
    n = g.size();
    G = g;
    siz.assign(n, 0);
    dfn.assign(n, -1);
    son.assign(n, -1);
    top.assign(n, -1);
    d.assign(n, 0);
    fa.assign(n, -1);
    d[root] = 1;
    dfs(root, -1);
    idx = 0;
    dfs2(root, root);
  }

  /// @brief Check if a is in the subtree of b.
  bool is_subtree(int a, int b) const {
    if (dfn[b] <= dfn[a] && dfn[a] < dfn[b] + siz[b]) {
      return true;
    } else {
      return false;
    }
  }

  /// @brief Return the lowest common ancestor of x and y.
  int lca(int x, int y) const {
    while (top[x] != top[y]) {
      if (d[top[x]] < d[top[y]])
        std::swap(x, y);
      x = fa[top[x]];
    }
    return d[x] < d[y] ? x : y;
  }

  /// @brief Decompose path x->y into chain segments. @return (dfn_l,
dfn_r, direction).
  std::vector<std::tuple<int, int, bool>> chain(int x, int y) const {
    assert(0 <= x && x < n);
    assert(0 <= y && y < n);
    std::vector<std::tuple<int, int, bool>> L, R;
    while (top[x] != top[y]) {
      assert(0 <= x && x < n);
      assert(0 <= y && y < n);
      if (d[top[x]] > d[top[y]]) {
        L.emplace_back(dfn[top[x]], dfn[x] + 1, false);
        x = fa[top[x]];
      } else {
        R.emplace_back(dfn[top[y]], dfn[y] + 1, true);
        y = fa[top[y]];
      }
    }
    if (dfn[y] < dfn[x])
      L.emplace_back(dfn[y], dfn[x] + 1, false);
    else
      R.emplace_back(dfn[x], dfn[y] + 1, true);
    reverse(R.begin(), R.end());
    L.insert(L.end(), R.begin(), R.end());
    return L;
  }

  /// @brief Return the DFN range [l, r) for the subtree of node a.
  std::array<int, 2> subtree(int a) const { return {dfn[a], dfn[a] +
siz[a]}; }

  /// @brief Return the LCA when the tree is re-rooted at c.
  int rooted_lca(int a, int b, int c) const {
    return lca(a, b) ^ lca(a, c) ^ lca(b, c);
  }

  /// @brief Compute the intersection of paths (a,b) and (c,d) as a pair of
endpoints.
  std::pair<int, int> intersection(int a, int b, int c, int d) const {
    int ab = lca(a, b), ac = lca(a, c), ad = lca(a, d);
    int bc = lca(b, c), bd = lca(b, d), cd = lca(c, d);
    int x = ab ^ ac ^ bc;
    int y = ab ^ ad ^ bd;
    if (x != y) {
      return {x, y};
    }
    int z = ac ^ ad ^ cd;
    if (x != z) {
      x = -1;
    }
    return {x, x};
  }

  std::pair<int, int> intersection(std::pair<int, int> a,
                                   std::pair<int, int> b) const {
    return intersection(a.first, a.second, b.first, b.second);
  }
};
} // namespace noya
```

hungarian.hpp

```
#include <cassert>
#include <limits>
#include <tuple>
#include <vector>

namespace noya {

/// @brief Hungarian algorithm. @return (cost, match, X-potential, Y-
potential).
template <typename T, bool MINIMIZE>
std::tuple<T, std::vector<int>, std::vector<T>, std::vector<T>>
hungarian(std::vector<std::vector<T>> &C) {
  int N = int(C.size());
  int M = int(C[0].size());
  assert(N <= M);
  std::vector<std::vector<T>> A(N + 1, std::vector<T>(M + 1));
  for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
      A[1 + i][1 + j] = (MINIMIZE ? 1 : -1) * C[i][j];
  ++N, ++M;

  std::vector<int> P(M), way(M);
  std::vector<T> X(N), Y(M);
  std::vector<T> minV;
  std::vector<bool> used;

  for (int i = 1; i < N; i++) {
    P[0] = i;
    minV.assign(M, std::numeric_limits<T>::max());
    used.assign(M, false);
    int j0 = 0;
    while (P[j0] != 0) {
      int i0 = P[j0], j1 = 0;
      used[j0] = true;
      T delta = std::numeric_limits<T>::max();
        for (int j = 1; j < M; j++) {
        if (used[j])
          continue;
        T curr = A[i0][j] - X[i0] - Y[j];
        if (curr < minV[j])
          minV[j] = curr, way[j] = j0;
        if (minV[j] < delta)
```

```
        delta = minV[j], j1 = j;
      }
      for (int j = 0; j < M; j++) {
        if (used[j])
          X[P[j]] += delta, Y[j] -= delta;
        else
          minV[j] -= delta;
      }
      j0 = j1;
    }
    do {
      P[j0] = P[way[j0]];
      j0 = way[j0];
    } while (j0 != 0);
  }
  T res = -Y[0];
  X.erase(X.begin());
  Y.erase(Y.begin());
  std::vector<int> match(N);
  for (int i = 0; i < M; i++)
    match[P[i]] = i;
  match.erase(match.begin());
  for (auto &&i : match)
    --i;
  if (!MINIMIZE)
    res = -res;
  return {res, match, X, Y};
}
} // namespace noya
```

lowest_common_ancestor.hpp

```
#include "noya/sparse_table.hpp"
#include <algorithm>

namespace noya {
/// @brief Sparse-table-based LCA with O(n log n) build and O(1) query.
struct fastlca {
  static std::pair<int, int> min_op(std::pair<int, int> a,
                                    std::pair<int, int> b) {
    return std::min(a, b);
  }
  int n;
  std::vector<int> dfn;
  std::vector<int> d;
  std::vector<int64_t> len;
  std::vector<int> siz;
  sparse_table<std::pair<int, int>, min_op> ST;
  bool weighted;

  std::vector<int> fa;

  template <class T>
  fastlca(const std::vector<T> &g = {}, const bool &_weighted = false,
          const int &root = 0) {
    weighted = _weighted;
    if (!g.empty())
      build(g, root);
  };

  template <class T>
  void build(const std::vector<std::vector<T>> &g, const int &root = 0) {
    n = int(g.size());
    std::vector<std::vector<int>> g2(n);

    for (int u = 0; u < n; u++) {
      for (auto &[v, w] : g[u]) {
        g2[u].push_back(v);
      }
    }
    build(g2, root);

    len.assign(n, 0);
    auto dfs = [&](auto &dfs, int u) -> void {
      for (auto &[v, w] : g[u]) {
        if (v == fa[u])
          continue;
        len[v] = len[u] + w;
        dfs(dfs, v);
      }
    };

    dfs(dfs, root);
  }

  void build(const std::vector<std::vector<int>> &g = {}, const int &root =
0) {
    n = int(g.size());
    d.assign(n, 0);
    dfn.assign(n, -1);
    siz.assign(n, 0);
    fa.assign(n, -1);

    int idx = 0;
    std::vector<std::pair<int, int>> a;
    a.reserve(n);

    auto dfs = [&](auto &dfs, int u, int p) -> void {
      fa[u] = p;
      siz[u] = 1;
      dfn[u] = idx++;
      if (p == -1)
        a.push_back({-1, -1});
      else
        a.push_back(std::make_pair(dfn[p], p));

      for (auto &v : g[u]) {
        if (v != p) {
          d[v] = d[u] + 1;
          dfs(dfs, v, u);
          siz[u] += siz[v];
        }
      }
    };

    dfs(dfs, root, -1);
    ST.build(a);
  }

  /// @brief Check if node a is in the subtree of node b.
  bool is_subtree(int a, int b) {
    if (dfn[b] <= dfn[a] && dfn[a] < dfn[b] + siz[b]) {
      return true;
    } else {
      return false;
    }
  }

  /// @brief Return the lowest common ancestor of nodes u and v.
  int lca(int u, int v) const {
    assert(0 <= u && u < n);
    assert(0 <= v && v < n);

    if (u == v) {
      return u;
    } else {
      int a = dfn[u];
      int b = dfn[v];
      if (a > b) {
        std::swap(a, b);
      }
      return ST.prod(a + 1, b + 1).second;
    }
  }

  /// @brief Return the LCA when the tree is re-rooted at c.
  int rooted_lca(int a, int b, int c) const {
    return lca(a, b) ^ lca(a, c) ^ lca(b, c);
  }

  /// @brief Return the (weighted or unweighted) distance between nodes a
and b.
  int64_t distance(int a, int b) const {
    int c = lca(a, b);
    if (!weighted) {
      return d[a] + d[b] - d[c] * 2;
    } else {
      return len[a] + len[b] - len[c] * 2;
    }
  }

  /// @brief Compute the intersection of paths (a,b) and (c,d) as a pair of
endpoints.
  std::pair<int, int> intersection(int a, int b, int c, int d) const {
    int ab = lca(a, b), ac = lca(a, c), ad = lca(a, d);
    int bc = lca(b, c), bd = lca(b, d), cd = lca(c, d);
    int x = ab ^ ac ^ bc;
    int y = ab ^ ad ^ bd;
    if (x != y) {
      return {x, y};
    }
    int z = ac ^ ad ^ cd;
    if (x != z) {
      x = -1;
    }
    return {x, x};
  }

  std::pair<int, int> intersection(std::pair<int, int> a,
                                   std::pair<int, int> b) const {
    return intersection(a.first, a.second, b.first, b.second);
  }

  /// @brief Return the list of vertices on the path from a to b.
  std::vector<int> path(int a, int b) {
    int c = lca(a, b);
    std::vector<int> ac;
    while (a != c) {
      ac.push_back(a);
      a = fa[a];
    }
    std::vector<int> bc;
    while (b != c) {
      bc.push_back(b);
      b = fa[b];
    }
    std::vector<int> res = std::move(ac);
    res.push_back(c);
    res.insert(res.end(), bc.rbegin(), bc.rend());
    return res;
  }
};
} // namespace noya
```

manhattan_mst.hpp

```
#include "noya/minimum_spanning_tree.hpp"

#include <algorithm>
#include <map>
#include <numeric>
#include <tuple>
#include <vector>

namespace noya {

/// @brief Compute candidate edges for Manhattan MST in O(n log n).
/// @return Sorted edges as (weight, vertex_i, vertex_j).
template <typename T>
std::vector<std::tuple<T, int, int>> manhattan_edges(std::vector<T> xs,
                                                     std::vector<T> ys) {
  const int n = xs.size();
  std::vector<int> idx(n);
  std::iota(idx.begin(), idx.end(), 0);
  std::vector<std::tuple<T, int, int>> ret;
  for (int s = 0; s < 2; s++) {
    for (int t = 0; t < 2; t++) {
      auto cmp = [&](int i, int j) { return xs[i] + ys[i] < xs[j] +
```

```
ys[j]; };
      std::sort(idx.begin(), idx.end(), cmp);
      std::map<T, int> sweep;
      for (int i : idx) {
        for (auto it = sweep.lower_bound(-ys[i]); it != sweep.end();
             it = sweep.erase(it)) {
          int j = it->second;
          if (xs[i] - xs[j] < ys[i] - ys[j])
            break;
          ret.emplace_back(std::abs(xs[i] - xs[j]) + std::abs(ys[i] -
ys[j]), i,
                           j);
        }
        sweep[-ys[i]] = i;
      }
      std::swap(xs, ys);
    }
    for (auto &x : xs)
      x = -x;
  }
  std::sort(ret.begin(), ret.end());
  return ret;
}

template <typename PointType>
auto manhattan_edges(const std::vector<PointType> &points) {
  assert(!points.empty());
  using CoordType = std::decay_t<decltype(std::get<0>(points[0]))>;
  std::vector<CoordType> xs, ys;
  for (const auto &point : points) {
    xs.push_back(std::get<0>(point));
    ys.push_back(std::get<1>(point));
  }
  return manhattan_edges(xs, ys);
}

} // namespace noya
```

minimum_spanning_tree.hpp

```
#include "atcoder/dsu.hpp"

#include <algorithm>
#include <numeric>
#include <vector>

namespace noya {

/// @brief Kruskal's MST. Edges are (w, u, v). @return (cost, edge
indices).
template <class T>
std::pair<int64_t, std::vector<int>>
minimum_spanning_tree(int N, const std::vector<T> &edges) {
  int64_t ans = 0;
  std::vector<int> idx;
  atcoder::dsu f(N);
  int m = int(edges.size());
  std::vector<int> p(m);
  std::iota(p.begin(), p.end(), 0);
  std::sort(p.begin(), p.end(),
            [&](int i, int j) { return edges[i] < edges[j]; });

  for (auto &i : p) {
    auto &[w, u, v] = edges[i];
    assert(0 <= u && u < N);
    assert(0 <= v && v < N);
    if (!f.same(u, v)) {
      ans += w;
      f.merge(u, v);
      idx.push_back(i);
    }
  }
  return make_pair(ans, idx);
}

/// @brief Kruskal's maximum spanning tree. Edges are (w, u, v). @return
(cost, edge indices).
template <class T>
std::pair<int64_t, std::vector<int>>
maximum_spanning_tree(int N, const std::vector<T> &edges) {
  int64_t ans = 0;
  std::vector<int> idx;
  atcoder::dsu f(N);
  int m = int(edges.size());
  std::vector<int> p(m);
  std::iota(p.begin(), p.end(), 0);
  std::sort(p.begin(), p.end(),
            [&](int i, int j) { return edges[i] > edges[j]; });

  for (auto &i : p) {
    auto &[w, u, v] = edges[i];
    assert(0 <= u && u < N);
    assert(0 <= v && v < N);
    if (!f.same(u, v)) {
      ans += w;
      f.merge(u, v);
      idx.push_back(i);
    }
  }
  return make_pair(ans, idx);
}

/// @brief Prim's MST for dense graphs. Edges are (w, u, v). @return (cost,
edge indices).
template <class T>
std::pair<int64_t, std::vector<int>> prim_dense(int N,
                                                const std::vector<T>
&edges) {
  std::vector<std::vector<int>> id(N, std::vector<int>(N, -1));
  const int M = int(edges.size());
  for (int i = 0; i < M; i++) {
    auto [w, u, v] = edges[i];
    assert(0 <= u && u < N);
    assert(0 <= v && v < N);
    id[u][v] = id[v][u] = i;
  }
  std::vector<int> idx;
  std::vector<int> vis(N, 0);
  std::vector<int> dis(N, -1);
  auto cmp = [&](int a, int b) -> int {
    if (a == -1)
      return b;
    if (b == -1)
      return a;
    auto [w1, u1, v1] = edges[a];
    auto [w2, u2, v2] = edges[b];
    return w1 < w2 ? a : b;
  };

  int64_t ans = 0;
  int k = 0;
  for (int t = 1; t < N; t++) {
    vis[k] = 1;
    int nx = -1;
    for (int i = 0; i < N; i++) {
      if (!vis[i]) {
        dis[i] = cmp(dis[i], id[k][i]);
        nx = cmp(nx, dis[i]);
      }
    }
    if (nx == -1)
      break;
    idx.push_back(nx);
    {
      auto [w, u, v] = edges[nx];
      ans += w;
      if (vis[u]) {
        k = v;
      } else {
        k = u;
      }
    }
  }
  return make_pair(ans, idx);
}

} // namespace noya
```

rooted_tree_isomorphism.hpp

```
#include <algorithm>
#include <map>
#include <vector>

namespace noya {

/// @brief Assign canonical hash labels to rooted subtrees for isomorphism
testing.
struct tree_isomorphism {
  std::map<std::vector<int>, int> mp;
  int cnt = 0;

  /// @brief Compute canonical labels for all nodes of a rooted tree.
  /// @return Vector mapping each node to its canonical subtree label.
  std::vector<int> solve(const std::vector<std::vector<int>> &g,
                         const int root = 0) {
    int N = int(g.size());
    std::vector<int> ans(N);
    auto dfs = [&](auto &dfs, int u, int p) -> int {
      std::vector<int> sons;
      for (auto v : g[u]) {
        if (v == p)
          continue;
        sons.push_back(dfs(dfs, v, u));
      }
      std::sort(sons.begin(), sons.end());
      if (!mp.count(sons))
        mp[sons] = cnt++;
      return ans[u] = mp[sons];
    };
    dfs(dfs, root, -1);
    return ans;
  }
};

} // namespace noya
```

shortest_path.hpp

```
#include <algorithm>
#include <cassert>
#include <limits>
#include <queue>
#include <vector>

namespace noya {
/// @brief BFS on unweighted graph. @return (distances, predecessors).
inline std::pair<std::vector<int>, std::vector<int>>
bfs_unweighted(std::vector<std::vector<int>> &g, int start) {
  int N = int(g.size());
  const int INF = std::numeric_limits<int>::max();
  std::vector<int> dis(N, INF);
  std::vector<int> pre(N, -1);
  dis[start] = 0;
  pre[start] = start;

  std::vector<int> que{start};
  for (int i = 0; i < int(que.size()); i++) {
    int u = que[i];
    for (auto v : g[u])
      if (dis[v] == INF) {
        dis[v] = dis[u] + 1;
        pre[v] = u;
        que.push_back(v);
      }
```

```
  }
  return {dis, pre};
}

/// @brief Dijkstra's algorithm. @return (distances, predecessors).
template <class T>
std::pair<std::vector<int64_t>, std::vector<int>>
dijkstra(std::vector<std::vector<T>> &g, int start) {
  int N = int(g.size());
  const int64_t INF = std::numeric_limits<int64_t>::max();
  std::vector<int64_t> dis(N, INF);
  std::vector<int> pre(N, -1);
  std::priority_queue<std::pair<int64_t, int>,
                      std::vector<std::pair<int64_t, int>>, std::greater<>>
      que;
  que.emplace(dis[start] = 0, start);
  pre[start] = start;
  while (!que.empty()) {
    auto [d, u] = que.top();
    que.pop();
    if (d > dis[u])
      continue;
    for (auto &[v, w] : g[u])
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        pre[v] = u;
        que.emplace(dis[v], v);
      }
  }
  return {dis, pre};
}

inline std::vector<int> find_path(std::vector<int> &pre, int s, int t) {
  std::vector<int> path;
  int cur = t;
  while (cur != s) {
    assert(cur >= 0);
    path.push_back(cur);
    cur = pre[cur];
  }
  path.push_back(s);
  std::reverse(path.begin(), path.end());
  return path;
}

template <class T>
std::vector<std::vector<int64_t>> floyd(std::vector<std::vector<T>> &g) {
  int N = int(g.size());
  const int64_t INF = std::numeric_limits<T>::max() / 2;
  std::vector<std::vector<int64_t>> f(N, std::vector<int64_t>(N, INF));
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      f[i][j] = g[i][j];
  for (int i = 0; i < N; i++)
    f[i][i] = 0;
  for (int k = 0; k < N; k++)
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        f[i][j] = std::min(f[i][j], f[i][k] + f[k][j]);
  return f;
}

} // namespace noya
```

tree_diameter.hpp

```
#include "noya/lowest_common_ancestor.hpp"
#include "noya/shortest_path.hpp"

#include <algorithm>
#include <array>
#include <tuple>
#include <vector>

namespace noya {

/// @brief Unweighted tree diameter via double BFS. @return (diameter, u,
v, eccentricity[]).
inline std::tuple<int, int, int, std::vector<int>>
tree_diam(std::vector<std::vector<int>> &g) {
  if (g.empty())
    return {-1, -1, -1, {}};
  auto d0 = bfs_unweighted(g, 0).first;
  int p = std::max_element(d0.begin(), d0.end()) - d0.begin();
  auto dp = bfs_unweighted(g, p).first;
  int q = std::max_element(dp.begin(), dp.end()) - dp.begin();
  auto dq = bfs_unweighted(g, q).first;
  int n = int(g.size());
  std::vector<int> ecc(n);
  for (int i = 0; i < n; i++)
    ecc[i] = std::max(dp[i], dq[i]);
  return {dp[q], p, q, ecc};
}

/// @brief Diameter monoid for segment tree. Merge two vertex sets and
track the farthest pair.
struct diameter_monoid {
  using S = std::pair<int64_t, std::array<int, 2>>;
  static constexpr S identity = {-1, {-1, -1}};

  fastlca *lca;
  explicit diameter_monoid(fastlca &l) : lca(&l) {}

  S make(int v) const { return {0, {v, v}}; }

  S merge(S a, S b) const {
    if (a == identity) return b;
    if (b == identity) return a;
    S c = std::max(a, b);
    for (auto x : a.second)
      for (auto y : b.second) {
        int64_t d = lca->distance(x, y);
        if (d > c.first)
          c = {d, {x, y}};
      }
    return c;
  }
};

} // namespace noya
```

String

lyndon_factor.hpp

```
#include <string>
#include <vector>

namespace noya {

/// @brief Compute the Lyndon factorization of a sequence.
/// @return Boundary indices of Lyndon factors (starting positions
including 0 and n).
template <class T> std::vector<int> lyndon(const std::vector<T> &s) {
  int n = int(s.size());
  std::vector<int> res{0};
  for (int i = 0; i < n;) {
    int j = i + 1, k = i;
    for (; j < n && s[k] <= s[j]; j += 1)
      k = s[k] < s[j] ? i : k + 1;
    while (i <= k)
      res.push_back(i += j - k);
  }
  return res;
}
inline std::vector<int> lyndon(const std::string &s) {
  std::vector<char> a(s.begin(), s.end());
  return lyndon(a);
}

} // namespace noya
```

manacher.hpp

```
#include <cassert>
#include <string>
#include <vector>

namespace noya {

/// @brief Compute palindrome radii for all centers (characters and gaps)
using Manacher's algorithm.
/// @return Array of length 2n-1 where entry i is the radius of the longest
palindrome centered at position i/2.
template <class T> std::vector<int> manacher(const std::vector<T> &s) {
  int n = int(s.size());
  if (n == 0)
    return {};
  std::vector<T> t{-2, -1};
  for (auto &a : s) {
    assert(a >= 0);
    t.push_back(a);
    t.push_back(-1);
  }
  int N = int(t.size());
  std::vector<int> p(N);
  int m = -1, r = -1;
  for (int i = 0; i < N; i++) {
    if (i <= r) {
      p[i] = std::min(p[m * 2 - i], r - i + 1);
    }
    while (i - p[i] >= 0 && i + p[i] < N && t[i + p[i]] == t[i - p[i]]) {
      ++p[i];
    }
    if (i + p[i] > r) {
      r = i + p[i] - 1;
      m = i;
    }
    --p[i];
  }
  std::vector<T>().swap(t);
  return std::vector<int>(p.begin() + 2, p.end() - 1);
}

inline std::vector<int> manacher(const std::string &s) {
  int n = int(s.size());
  std::vector<int> s2(n);
  for (int i = 0; i < n; i++) {
    s2[i] = s[i];
  }
  return manacher(s2);
}

} // namespace noya
```

minimal_string.hpp

```
#include <algorithm>
#include <string>
#include <vector>

namespace noya {

/// @brief Find the starting index of the lexicographically smallest
rotation.
template <class T> int minimal_string_index(const std::vector<T> &sec) {
```

```
  int k = 0, i = 0, j = 1, n = sec.size();
  while (k < n && i < n && j < n) {
    if (sec[(i + k) % n] == sec[(j + k) % n]) {
      k++;
    } else {
      if (sec[(i + k) % n] > sec[(j + k) % n])
        i = i + k + 1;
      else
        j = j + k + 1;
      if (i == j)
        i++;
      k = 0;
    }
  }
  return std::min(i, j);
}

/// @brief Return the lexicographically smallest rotation of the sequence.
template <class T> std::vector<T> minimal_string(const std::vector<T> &sec)
{
  if (sec.empty())
    return {};
  int start = minimal_string_index(sec);
  std::vector<T> ret(sec.size());
  std::rotate_copy(sec.begin(), sec.begin() + start, sec.end(),
ret.begin());
  return ret;
}

inline std::string minimal_string(const std::string &sec) {
  if (sec.empty())
    return "";
  int start = minimal_string_index(std::vector<char>(sec.begin(),
sec.end()));
  return sec.substr(start) + sec.substr(0, start);
}

} // namespace noya
```

online_z_algo.hpp

```
#include <string>
#include <vector>

namespace noya {
/// @brief Online Z-algorithm: compute Z-values incrementally as characters
are appended.
template <class S> struct online_z_algo {
  std::vector<std::vector<int>> memo;
  std::vector<int> z;
  std::vector<S> str;
  int p;
  online_z_algo() { p = 1; }

  /// @brief Append character c at position i and return indices whose Z-
values are now finalized.
  std::vector<int> add(int i, S c) {
    str.push_back(c);
    int len = str.size();
    z.push_back(-1);
    memo.resize(1 + i);
    std::vector<int> end;

    if (len == 1) {
      return end;
    }

    if (str[0] != c) {
      z[i] = 0;
      end.push_back(i);
    }

    auto del = [&](int j) {
      z[j] = i - j;
      memo[i].push_back(j);
      end.push_back(j);
    };

    while (p <= i) {
      if (z[p] != -1) {
        p += 1;
      } else if (str[i - p] != str[i]) {
        del(p);
        p += 1;
      } else {
        break;
      }
    }
    if (p < i) {
      for (auto j : memo[i - p]) {
        del(j + p);
      }
    }
    return end;
  }

  /// @brief Return the Z-value at position i (length of longest common
prefix with the string).
  int query(int i) {
    if (z[i] == -1) {
      return str.size() - i;
    }
    return z[i];
  }
};
} // namespace noya
```

palindromic_automaton.hpp

```
#include <array>
#include <cassert>
#include <vector>

namespace noya {

/// @brief Palindromic automaton (Eertree) for enumerating distinct
palindromic substrings.
template <int sigma = 26> struct palindromic_automaton {
  struct Node {
    std::array<int, sigma> next;
    int fail, len, cnt;

    Node(int fail, int len) : fail(fail), len(len), cnt(0) {
      std::fill(next.begin(), next.end(), 0);
    }
  };

  std::vector<Node> nodes;
  std::vector<int> s;
  int last = 0;

  palindromic_automaton() {
    nodes.push_back(Node(1, 0));
    nodes.push_back(Node(0, -1));
    s.push_back(-1);
    last = 0;
  }

  int get_fail(int x) {
    while (s[int(s.size()) - 1 - nodes[x].len - 1] != s.back()) {
      x = nodes[x].fail;
    }
    return x;
  }

  /// @brief Append character c (in [0, sigma)) and return the node id of
the new longest palindromic suffix.
  int extend(int c) {
    assert(0 <= c && c < sigma);
    s.push_back(c);
    int cur = get_fail(last);
    if (!nodes[cur].next[c]) {
      int now = int(nodes.size());
      int fail = nodes[get_fail(nodes[cur].fail)].next[c];
      nodes.push_back(Node(fail, nodes[cur].len + 2));
      nodes[cur].next[c] = now;
    }
    last = nodes[cur].next[c];
    nodes[last].cnt++;
    return last;
  }

  /// @brief Return the number of nodes in the automaton.
  int size() const { return int(nodes.size()); }
};

} // namespace noya
```

rolling_hash.hpp

```
#include "atcoder/modint.hpp"
#include "noya/rnd.hpp"

#include <string>
#include <vector>

namespace noya {
inline int64_t rolling_hash_base() {
  static int64_t base = internal::gen_values()();
  return base;
}

template <class T, class S> struct hash_array {
  int n;
  std::vector<T> H;
  std::vector<T> pw;
  const T base = T(rolling_hash_base());
  hash_array(const std::vector<S> &a) { build(a); }
  hash_array(const std::string &a) {
    std::vector<S> a0(a.begin(), a.end());
    build(a0);
  }
  hash_array() {}

  void build(const std::vector<S> &a) {
    n = int(a.size());
    H.resize(n + 1);
    for (int i = 0; i < n; i++) {
      H[i + 1] = H[i] * base + T(a[i]);
    }
    pw.resize(n + 1);
    pw[0] = 1;
    for (int i = 0; i < n; i++) {
      pw[i + 1] = pw[i] * base;
    }
  }
  void build(const std::string &a) {
    std::vector<S> a0(a.begin(), a.end());
    build(a0);
  }

  /// @brief Hash of [l, r).
  T prod(int l, int r) const {
    assert(0 <= l && l <= r && r <= n);
    T X = H[r] - H[l] * pw[r - l];
    return X;
  }
};

template <class A, class B, class S> struct double_hash_array {
  int n;

  hash_array<A, S> A_hash_array;
  hash_array<B, S> B_hash_array;
```

```
  double_hash_array(const std::vector<S> &a) { build(a); }
  double_hash_array() {}
  double_hash_array(const std::string &a) {
    std::vector<S> a0(a.begin(), a.end());
    build(a0);
  }

  void build(const std::vector<S> &a) {
    n = int(a.size());
    A_hash_array.build(a);
    B_hash_array.build(a);
  }
  void build(const std::string &a) {
    std::vector<S> a0(a.begin(), a.end());
    build(a0);
  }

  /// @brief Hash of [l, r).
  std::pair<A, B> prod(int l, int r) const {
    return std::make_pair(A_hash_array.prod(l, r), B_hash_array.prod(l,
r));
  }
};

using mint1 = atcoder::modint1000000007;
using mint2 = atcoder::modint998244353;
using mint3 = uint64_t;

template <class T> int lcp(const T &A, int i, const T &B, int j) {
  int n = A.n;
  int m = B.n;

  int l = 0;
  int r = std::min(n - i, m - j) + 1;

  while (r - l > 1) {
    int mid = (l + r) / 2;
    if (A.prod(i, i + mid) == B.prod(j, j + mid))
      l = mid;
    else
      r = mid;
  }
  return l;
}

template <class T> int lcs(const T &A, int i, const T &B, int j) {
  int l = 0;
  int r = std::min(i + 1, j + 1) + 1;

  while (r - l > 1) {
    int mid = (l + r) / 2;
    if (A.prod(i + 1 - mid, i + 1) == B.prod(j + 1 - mid, j + 1))
      l = mid;
    else
      r = mid;
  }
  return l;
}

template <class T>
/// @brief Check if A[l1, r1) == B[l2, r2).
bool same(const T &A, int l1, int r1, const T &B, int l2, int r2) {
  int n1 = r1 - l1;
  int n2 = r2 - l2;

  assert(0 <= l1 && l1 <= r1 && r1 <= A.n);
  assert(0 <= l2 && l2 <= r2 && r2 <= B.n);

  if (n1 != n2)
    return false;
  int lc = lcp(A, l1, B, l2);
  return lc == n1;
}

} // namespace noya
```

runs.hpp

```
#include "noya/rolling_hash.hpp"

#include <ranges>
#include <string>
#include <tuple>
#include <vector>

namespace noya {

/// @brief Compute all runs (maximal periodicities) in a sequence.
/// @return Vector of (period, left, right) where the run covers [left,
right).
template <class T>
std::vector<std::tuple<int, int, int>> runs(const std::vector<T> &s) {
  int n = int(s.size());
  noya::double_hash_array<mint1, mint2, T> H;
  H.build(s);
  std::vector<std::tuple<int, int, int>> runs;
  for (bool inv : {false, true}) {
    std::vector<int> lyn(n, n), stack;
    for (int i = 0; i < n; i += 1) {
      while (!stack.empty()) {
        int j = stack.back(), k = lcp(H, i, H, j);
        if (i + k < n && ((s[i + k] > s[j + k]) ^ inv))
          break;
        lyn[j] = i;
        stack.pop_back();
      }
      stack.push_back(i);
    }
    for (int i = 0; i < n; i += 1) {
      int j = lyn[i], t = j - i, l = i - lcs(H, i - 1, H, j - 1),
          r = j + lcp(H, i, H, j);
      if (r - l >= 2 * t)
        runs.emplace_back(t, l, r);
    }
  }
  std::ranges::sort(runs);
  runs.erase(std::ranges::unique(runs).begin(), runs.end());
  return runs;
}
std::vector<std::tuple<int, int, int>> runs(const std::string &s) {
  std::vector<char> a(s.begin(), s.end());
  return runs(a);
}

} // namespace noya
```

suffix_automaton.hpp

```
#include <algorithm>
#include <array>
#include <cassert>
#include <map>
#include <vector>

namespace noya {
/// @brief Suffix automaton (SAM) for online suffix structure construction.
template <int sigma = 26> struct suffix_automaton {
  struct Node {
    std::array<int, sigma> next;
    int link;
    int len;

    Node(int link, int len) : link(link), len(len) {
      std::fill(next.begin(), next.end(), -1);
    }
  };

  std::vector<Node> nodes;
  int last = 0;

  suffix_automaton() {
    nodes.push_back(Node(-1, 0));
    last = 0;
  }

  /// @brief Append character c (in [0, sigma)) and return the new node id.
  // https://maspypy.github.io/library/string/suffix_automaton.hpp
  int extend(int c) {
    assert(0 <= c && c < sigma);
    int new_node = int(nodes.size());
    nodes.push_back(Node(-1, nodes[last].len + 1));
    int p = last;
    while (p != -1 && nodes[p].next[c] == -1) {
      nodes[p].next[c] = new_node;
      p = nodes[p].link;
    }
    int q = (p == -1 ? 0 : nodes[p].next[c]);
    if (p == -1 || nodes[p].len + 1 == nodes[q].len) {
      nodes[new_node].link = q;
    } else {
      int new_q = int(nodes.size());
      nodes.push_back(Node(nodes[q].link, nodes[p].len + 1));
      nodes.back().next = nodes[q].next;
      nodes[q].link = new_q;
      nodes[new_node].link = new_q;
      while (p != -1 && nodes[p].next[c] == q) {
        nodes[p].next[c] = new_q;
        p = nodes[p].link;
      }
    }
    return last = new_node;
  }

  /// @brief Return the suffix-link tree as an adjacency list.
  std::vector<std::vector<int>> get_tree() const {
    int n = int(nodes.size());
    std::vector<std::vector<int>> g(n);
    for (int i = 1; i < n; i++) {
      g[nodes[i].link].push_back(i);
    }
    return g;
  }

  /// @brief Return the number of distinct substrings represented by node
i.
  long long count_substring_at(int i) const {
    if (i == 0) {
      return 0;
    } else {
      return nodes[i].len - nodes[nodes[i].link].len;
    }
  };
  /// @brief Return the total number of distinct non-empty substrings.
  long long count_substring() const {
    long long ans = 0;
    int n = int(nodes.size());
    for (int i = 1; i < n; i++) {
      ans += count_substring_at(i);
    }
    return ans;
  }
};

} // namespace noya
```

Math

binomial.hpp

```cpp
#include <vector>
#include <cassert>

namespace noya {
/// @brief Precomputed binomial coefficient table over a modular type T.
template <class T> struct binom {
  std::vector<T> fac, ifac;
  binom() {}
  binom(int n) { prepare(n); }

  /// @brief Precompute factorials and inverse factorials up to n.
  void prepare(int n) {
    if (fac.empty()) {
      fac = {1};
      ifac = {1};
    }

    for (int i = int(fac.size()); i <= n; i++) {
      fac.push_back(fac[i - 1] * i);
      ifac.push_back(fac[i].inv());
    }
  }

  /// @brief Return n! (mod p).
  T fact(int n) {
    if (n >= int(fac.size()))
      prepare(n * 2);
    return fac[n];
  }

  /// @brief Return (n!)^{-1} (mod p).
  T ifact(int n) {
    if (n >= int(ifac.size()))
      prepare(n * 2);
    return ifac[n];
  }

  /// @brief Return the binomial coefficient C(n, m).
  T C(int n, int m) {
    // assert(0 <= m && m <= n);
    if (!(0 <= m && m <= n))
      return 0;
    return fact(n) * ifact(m) * ifact(n - m);
  }
};
}
```

convolution.hpp

```cpp
#include "atcoder/convolution.hpp"

namespace noya {
inline int highest_bit(unsigned x) {
  return x == 0 ? -1 : 31 - __builtin_clz(x);
}
/// @brief Compute bitwise-AND convolution of two vectors of length 2^e.
template <class T>
std::vector<T> bitwise_and_convolution(std::vector<T> a, std::vector<T> b)
{
  int N = int(a.size());
  assert(int(b.size()) == N);
  int e = highest_bit(N);

  assert(N == (1 << e));

  auto convolution = [&](std::vector<T> &f) -> void {
    for (int j = 0; j < e; j++)
      for (int i = 0; i < N; i++)
        if (i >> j & 1)
          f[i ^ 1 << j] += f[i];
  };
  auto envolution = [&](std::vector<T> &f) -> void {
    for (int j = 0; j < e; j++)
      for (int i = 0; i < N; i++)
        if (i >> j & 1)
          f[i ^ 1 << j] -= f[i];
  };

  convolution(a), convolution(b);
  std::vector<T> c(N);
  for (int i = 0; i < N; i++)
    c[i] = a[i] * b[i];
  envolution(c);
  return c;
}
/// @brief Compute bitwise-OR convolution of two vectors of length 2^e.
template <class T>
std::vector<T> bitwise_or_convolution(std::vector<T> a, std::vector<T> b) {
  int N = int(a.size());
  assert(int(b.size()) == N);
  int e = highest_bit(N);

  assert(N == (1 << e));

  std::reverse(a.begin(), a.end());
  std::reverse(b.begin(), b.end());
  auto c = bitwise_and_convolution(a, b);
  std::reverse(c.begin(), c.end());
  return c;
}

/// @brief Compute bitwise-XOR convolution of two vectors of length 2^e.
template <class T>
std::vector<T> bitwise_xor_convolution(std::vector<T> a, std::vector<T> b)
{
  int N = int(a.size());
  assert(int(b.size()) == N);
  int e = highest_bit(N);

  assert(N == (1 << e));

  auto convolution = [&](std::vector<T> &f) -> void {
    for (int j = 0; j < e; j++)
      for (int i = 0; i < N; i++)
        if (~i >> j & 1)
          f[i ^ 1 << j] += f[i];
  };
  auto envolution = [&](std::vector<T> &f) -> void {
    for (int j = 0; j < e; j++)
      for (int i = 0; i < N; i++)
        if (~i >> j & 1)
          f[i ^ 1 << j] -= f[i];
  };

  convolution(a), convolution(b);
  std::vector<T> c(N);
  for (int i = 0; i < N; i++)
    c[i] = a[i] * b[i];
  envolution(c);
  return c;
}

/// @brief Compute subset convolution of two vectors of length 2^e.
template <class T>
std::vector<T> subset_convolution(std::vector<T> a, std::vector<T> b) {
  int N = int(a.size());
  assert(int(b.size()) == N);
  int e = highest_bit(N);

  assert(N == (1 << e));
  auto convolution = [&](std::vector<T> &f) -> void {
    for (int j = 0; j < e; j++)
      for (int i = 0; i < N; i++)
        if (~i >> j & 1)
          f[i ^ 1 << j] += f[i];
  };
  auto envolution = [&](std::vector<T> &f) -> void {
    for (int j = 0; j < e; j++)
      for (int i = 0; i < N; i++)
        if (~i >> j & 1)
          f[i ^ 1 << j] -= f[i];
  };

  std::vector<std::vector<T>> f(e + 1, std::vector<T>(N));
  std::vector<std::vector<T>> g(e + 1, std::vector<T>(N));
  std::vector<T> ans(N);

  for (int i = 0; i < N; i++) {
    f[__builtin_popcount(i)][i] = a[i];
    g[__builtin_popcount(i)][i] = b[i];
  }

  for (int i = 0; i < e; i++) {
    convolution(f[i]);
    convolution(g[i]);
  }

  std::vector<T> tmp(N);
  for (int i = 0; i <= e; i++) {
    for (int j = 0; j < N; j++)
      tmp[j] = 0;
    for (int j = 0; j <= i; j++)
      for (int k = 0; k < N; k++)
        tmp[k] += f[j][k] * g[i - j][k];
    envolution(tmp);
    for (int k = 0; k < N; k++)
      if (__builtin_popcount(k) == i)
        ans[k] = tmp[k];
  }
  return ans;
}
} // namespace noya
```

extended_gcd.hpp

```cpp
#include <algorithm>
#include <cassert>
#include <vector>

namespace noya {

/// @brief Compute gcd(a, b) and coefficients x, y such that a*x + b*y =
gcd.
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;
}

/// @brief Solve a*x + b*y = c for integers x, y; returns false if no
solution.
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;
}

/// @brief Chinese Remainder Theorem for two congruences; returns false if
no solution.
inline 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;
}

/// @brief Garner's algorithm for CRT with multiple moduli; stores result
in res.
template <typename T>
void crt_garner(const std::vector<int> &p, const std::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;
      std::swap(q, b);
      u -= t * v;
      std::swap(u, v);
    }
    assert(b == 1);
    if (u < 0)
      u += m;
    return u;
  };
  std::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];
  }
}

} // namespace noya
```

gcd_maintenance.hpp

```
#include <algorithm>
#include <vector>
namespace noya {

/// @brief For each starting position l, returns the positions where op-
product changes.
template <class T, auto op>
std::vector<std::vector<int>> get_suffix(const std::vector<T> &A) {
  int N = int(A.size());
  std::vector<std::vector<int>> position(N);
  std::vector<std::pair<T, int>> cur;
  for (int i = N - 1; i >= 0; i--) {
    T x = A[i];
    if (!cur.empty()) {
      std::vector<std::pair<T, int>> new_cur;
      std::reverse(cur.begin(), cur.end());
      T g = x;
      for (const auto &[value, pos] : cur) {
        g = op(g, value);
        if (new_cur.empty() || new_cur.back().first != g) {
          new_cur.push_back({g, pos});
        }
      }
      cur.swap(new_cur);
      std::reverse(cur.begin(), cur.end());
    }
    cur.push_back({x, i + 1});
    for (const auto &[value, pos] : cur) {
      position[i].push_back(pos);
    }
  }
  return position;
};

/// @brief For each starting position l, returns (position, op-product)
pairs.
template <class T, auto op>
std::vector<std::vector<std::pair<int, T>>>
get_suffix_values(const std::vector<T> &A) {
  int N = int(A.size());
  std::vector<std::vector<std::pair<int, T>>> values(N);
  std::vector<std::pair<int, T>> cur;
  for (int i = N - 1; i >= 0; i--) {
    T x = A[i];
    if (!cur.empty()) {
      std::vector<std::pair<int, T>> new_cur;
      std::reverse(cur.begin(), cur.end());
      T g = x;
      for (const auto &[pos, val] : cur) {
        g = op(val, g);
        if (new_cur.empty() || new_cur.back().second != g) {
          new_cur.push_back({pos, g});
        }
      }
      cur.swap(new_cur);
      std::reverse(cur.begin(), cur.end());
    }
    cur.push_back({i + 1, x});
    values[i] = cur;
  }
  return values;
};

} // namespace noya
```

modint.hpp

```
#include "atcoder/modint.hpp"

namespace noya {

/// @brief Alias for atcoder::modint998244353.
using mint998 = atcoder::modint998244353;
/// @brief Alias for atcoder::modint1000000007.
using mint107 = atcoder::modint1000000007;

/// @brief Alias for atcoder::static_modint with a custom modulus.
template<const int MOD>
using mint = atcoder::static_modint<MOD>;

} // namespace noya
```

DP / Optimization

convex_hull_trick.hpp

```
#include <cassert>
#include <limits>
#include <map>
#include <set>
#include <vector>

namespace noya {
template <class T> struct linear_cht_min {
  std::vector<std::pair<T, T>> stack;
  /// @brief Add line y = px + q. p must be decreasing.
  void add(T p, T q) {
    if (stack.size() >= 1) {
      assert(stack.back().first >= p);
    }
    if (stack.size() >= 1 && stack.back().first == p) {
      if (stack.back().second <= q) {
        return;
      } else {
        stack.pop_back();
      }
    }
    while ((int)stack.size() >= 2) {
      const auto [p2, q2] = stack.end()[-2];
      const auto [p1, q1] = stack.end()[-1];
      if ((__int128)(q1 - q2) * (p1 - p) < (__int128)(q - q1) * (p2 - p1))
{
        break;
      }
      stack.pop_back();
    }
    stack.emplace_back(p, q);
  }
  /// @brief Query minimum at x. x must be decreasing.
  T get(T x) {
    while ((int)stack.size() >= 2) {
      const auto [p2, q2] = stack.end()[-2];
      const auto [p1, q1] = stack.end()[-1];
```

```
      if (p2 * x + q2 > p1 * x + q1) {
        break;
      } else {
        stack.pop_back();
      }
    }
    if (stack.empty()) {
      return std::numeric_limits<T>::max();
    } else {
      auto [k, b] = stack.back();
      return k * x + b;
    }
  }
};

// https://maspypy.github.io/library/convex/cht.hpp
template <typename T> struct Line {
  mutable T k, m, p;
  bool operator<(const Line &o) const { return k < o.k; }
  bool operator<(T x) const { return p < x; }
};

template <typename T> T lc_inf() { return std::numeric_limits<T>::max(); }
template <> inline long double lc_inf<long double>() { return 1 / .0; }

template <typename T> T lc_div(T a, T b) {
  return a / b - ((a ^ b) < 0 and a % b);
}
template <> inline long double lc_div(long double a, long double b)
{ return a / b; }
template <> inline double lc_div(double a, double b) { return a / b; }
template <typename T, bool MINIMIZE = true>
struct line_container : std::multiset<Line<T>, std::less<>> {
  using super = std::multiset<Line<T>, std::less<>>;
  using super::begin, super::end, super::insert, super::erase;
  using super::empty, super::lower_bound;
  T inf = lc_inf<T>();
  bool insect(typename super::iterator x, typename super::iterator y) {
    if (y == end())
      return x->p = inf, false;
    if (x->k == y->k)
      x->p = (x->m > y->m ? inf : -inf);
    else
      x->p = lc_div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(T k, T m) {
    if (MINIMIZE) {
      k = -k, m = -m;
    }
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (insect(y, z))
      z = erase(z);
    if (x != begin() and insect(--x, y))
      insect(x, y = erase(y));
    while ((y = x) != begin() and (--x)->p >= y->p)
      insect(x, erase(y));
  }
  T query(T x) {
    assert(!empty());
    auto l = *lower_bound(x);
    T v = (l.k * x + l.m);
    return (MINIMIZE ? -v : v);
  }
};

template <typename T> using cht_min = line_container<T, true>;
template <typename T> using cht_max = line_container<T, false>;
template <typename T> struct cht_xy {
  static_assert(std::is_same_v<T, long long> ||
std::is_floating_point_v<T>);
  using ld = long double;
  cht_min<ld> cht_min_;
  cht_max<ld> cht_max_;
  T amax = std::numeric_limits<T>::min(), amin =
std::numeric_limits<T>::max();
  T bmax = std::numeric_limits<T>::min(), bmin =
std::numeric_limits<T>::max();
  int amax_idx = -1, amin_idx = -1;
  int bmax_idx = -1, bmin_idx = -1;
  bool empty = true;
  std::map<std::pair<T, T>, int> MP;

  void clear() {
    empty = true;
    cht_min_.clear();
    cht_max_.clear();
  }
  void add(T a, T b, int i = -1) {
    empty = false;
    cht_min_.add(b, a);
    cht_max_.add(b, a);
    std::pair<T, T> p = {a, b};
    MP[p] = i;

    if (amax < a)
      amax = a, amax_idx = i;
    if (amin > a)
      amin = a, amin_idx = i;
    if (bmax < b)
      bmax = b, bmax_idx = i;
    if (bmin > b)
      bmin = b, bmin_idx = i;
  }

  std::pair<T, int> get_max(T x, T y) {
    if (cht_min_.empty())
      return {std::numeric_limits<T>::min(), -1};

    if (x == 0) {
      if (bmax * y > bmin * y) {
        return {bmax * y, bmax_idx};
      }
      return {bmin * y, bmin_idx};
    }
    ld z = ld(y) / x;
    if (x > 0) {
      auto l = cht_max_.lower_bound(z);
      T a = l->m, b = l->k;
      std::pair<T, T> p = {a, b};
      int idx = MP[p];
      return {a * x + b * y, idx};
    }
    auto l = cht_min_.lower_bound(z);
    T a = -(l->m), b = -(l->k);
    std::pair<T, T> p = {a, b};
    int idx = MP[p];
    return {a * x + b * y, idx};
  }

  std::pair<T, int> get_min(T x, T y) {
    auto [f, i] = get_max(-x, -y);
    return {-f, i};
  }
};
} // namespace noya
```

knapsack.hpp

```
#include "noya/max_plus_convolution.hpp"

#include <algorithm>
#include <bitset>
#include <limits>
#include <vector>

namespace noya {

/// @brief 0/1 knapsack. items are (weight, value) pairs. O(NlogN + L^2).
template <class P>
std::vector<int64_t> knapsack(int L, const std::vector<P> &items) {
  std::vector<std::vector<int>> buckets(L + 1);
  for (auto &[w, v] : items) {
    assert(w >= 0);
    assert(v >= 0);
    if (w <= L) {
      buckets[w].push_back(v);
    }
  }

  std::vector<int64_t> dp(L + 1, std::numeric_limits<int64_t>::lowest());
  dp[0] = 0;

  for (int w = 1; w <= L; w++) {
    auto &bucket = buckets[w];
    if (bucket.empty()) {
      continue;
    }
    std::sort(bucket.begin(), bucket.end(), std::greater<>());
    const int m = std::min(int(bucket.size()), L / w);
    std::vector<int64_t> sum(m + 1);
    for (int i = 0; i < m; i++) {
      sum[i + 1] = sum[i] + bucket[i];
    }
    // remainder class enumeration
    for (int r = 0; r < w; r++) {
      const int n = int(L - r) / w + 1;
      std::vector<int64_t> v(n);
      for (int i = 0; i < n; i++) {
        v[i] = dp[i * w + r];
      }
      const std::vector<int64_t> &res = concave_maxplus_convolution(v,
sum);
      for (int i = 0; i < n; i++) {
        dp[i * w + r] = res[i];
      }
    }
  }
  return dp;
}

// https://qoj.ac/problem/7403
// O(n max{A})
template <class T> T max_subsetsum_leq(const T &C, const std::vector<int>
&A) {
  int N = int(A.size());
  int p = -1;
  T cur = 0;
  for (int i = 0; i < N; i++) {
    if (cur + A[i] > C) {
      p = i;
      break;
    } else {
      cur += A[i];
    }
  }
  if (p == -1) {
    return cur;
  }
  const int shift = *std::max_element(A.begin(), A.end());
  std::vector<int> dp0(2 * shift + 1);
  std::vector<int> dp1(2 * shift + 1);

  for (int i = 0; i <= shift; i++)
    dp0[i] = -1;

  dp0[cur - C + shift] = p;

  for (int i = p; i < N; i++) {
    dp1 = dp0;
    for (int j = 0; j <= shift; j++)
      dp1[j + A[i]] = std::max(dp1[j + A[i]], dp0[j]);

    for (int j = shift + A[i]; j >= shift + 1; j--)
      for (int k = dp1[j] - 1; k >= dp0[j]; k--)
        dp1[j - A[k]] = std::max(dp1[j - A[k]], k);
    dp0 = dp1;
  }

  for (int i = shift; i >= 0; i--)
    if (dp0[i] >= 0)
```

```
      return i - shift + C;
  return 0;
}

template <int N> std::bitset<N> bool_knapsack(const std::vector<int>
&items) {
  std::vector<int> freq(N);
  for (auto &item : items) {
    if (item < N) {
      freq[item] += 1;
    }
  }
  for (int i = 1; i < N; i++) {
    if (freq[i] >= 3) {
      int a = (freq[i] - 1) / 2;
      freq[i * 2] += a;
      freq[i] -= a * 2;
    }
  }
  std::bitset<N> res;
  res[0] = 1;
  for (int i = 1; i < N; i++) {
    for (int j = 0; j < freq[i]; j++) {
      res |= res << i;
    }
  }
  return res;
}

} // namespace noya
```

larsch.hpp

```
// https://noshi91.github.io/Library/algorithm/larsch.cpp.html
#include <functional>
#include <memory>
#include <vector>

namespace noya {

/// @brief LARSCH algorithm for online row-minima of totally monotone
matrices.
template <class T> class larsch {
  struct reduce_row;
  struct reduce_col;

  struct reduce_row {
    int n;
    std::function<T(int, int)> f;
    int cur_row;
    int state;
    std::unique_ptr<reduce_col> rec;

    reduce_row(int n_) : n(n_), f(), cur_row(0), state(0), rec() {
      const int m = n / 2;
      if (m != 0) {
        rec = std::make_unique<reduce_col>(m);
      }
    }

    void set_f(std::function<T(int, int)> f_) {
      f = f_;
      if (rec) {
        rec->set_f([&](int i, int j) -> T { return f(2 * i + 1, j); });
      }
    }

    int get_argmin() {
      const int cur_row_ = cur_row;
      cur_row += 1;
      if (cur_row_ % 2 == 0) {
        const int prev_argmin = state;
        const int next_argmin = [&]() {
          if (cur_row_ + 1 == n) {
            return n - 1;
          } else {
            return rec->get_argmin();
          }
        }();
        state = next_argmin;
        int ret = prev_argmin;
        for (int j = prev_argmin + 1; j <= next_argmin; j += 1) {
          if (f(cur_row_, ret) > f(cur_row_, j)) {
            ret = j;
          }
        }
        return ret;
      } else {
        if (f(cur_row_, state) <= f(cur_row_, cur_row_)) {
          return state;
        } else {
          return cur_row_;
        }
      }
    }
  };

  struct reduce_col {
    int n;
    std::function<T(int, int)> f;
    int cur_row;
    std::vector<int> cols;
    reduce_row rec;

    reduce_col(int n_) : n(n_), f(), cur_row(0), cols(), rec(n) {}

    void set_f(std::function<T(int, int)> f_) {
      f = f_;
      rec.set_f([&](int i, int j) -> T { return f(i, cols[j]); });
    }

    int get_argmin() {
      const int cur_row_ = cur_row;
      cur_row += 1;
      const auto cs = [&]() -> std::vector<int> {
        if (cur_row_ == 0) {
          return {0};
        } else {
          return {2 * cur_row_ - 1, 2 * cur_row_};
        }
      }();
      for (const int j : cs) {
        while ([&]() {
          const int size = cols.size();
          return size != cur_row_ && f(size - 1, cols.back()) > f(size - 1,
j);
        }()) {
          cols.pop_back();
        }
        if (cols.size() != n) {
          cols.push_back(j);
        }
      }
      return cols[rec.get_argmin()];
    }
  };

  std::unique_ptr<reduce_row> base;

public:
  larsch(int n, std::function<T(int, int)> f)
      : base(std::make_unique<reduce_row>(n)) {
    base->set_f(f);
  }

  /// @brief Return the column index of the minimum in the next row.
  int get_argmin() { return base->get_argmin(); }
};

} // namespace noya
```

longest_increasing_subsequence.hpp

```
#include <algorithm>
#include <vector>

namespace noya {

/// @brief Compute indices of a longest strictly increasing subsequence.
/// @return Indices into A forming a longest increasing subsequence.
template <class T>
std::vector<int> longest_increasing_subsequence(const std::vector<T> &A) {
  if (A.empty())
    return {};
  std::vector<T> B;
  std::vector<int> indice;
  int N = int(A.size());
  std::vector<int> pre(N, -1);
  for (int i = 0; i < N; i++) {
    T a = A[i];
    int j = std::lower_bound(B.begin(), B.end(), a) - B.begin();
    if (j == int(B.size())) {
      indice.push_back(i);
      B.push_back(a);
    } else {
      indice[j] = i;
      B[j] = a;
    }
    if (j > 0) {
      pre[i] = indice[j - 1];
    }
  }
  std::vector<int> ans;
  for (int cur = indice.back(); cur != -1; cur = pre[cur])
    ans.push_back(cur);

  std::reverse(ans.begin(), ans.end());
  return ans;
}

} // namespace noya
```

max_plus_convolution.hpp

```
#include "noya/smawk.hpp"

#include <vector>

namespace noya {

/// @brief Max-plus convolution of two concave sequences.
template <class T>
std::vector<T> two_concave_maxplus_convolution(const std::vector<T> &a,
                                               const std::vector<T> &b) {
  if (a.empty())
    return b;
  if (b.empty())
    return a;
  const int n = int(a.size());
  const int m = int(b.size());
  int p = 0, q = 0;
  std::vector<T> c(n + m - 1);
  c[0] = a[0] + b[0];
  for (int i = 1; i < n + m - 1; i++) {
    if (p + 1 == n) {
      q++;
    } else if (q + 1 == m) {
      p++;
    } else {
      if (a[p + 1] - a[p] > b[q + 1] - b[q]) {
        p++;
      } else {
        q++;
      }
    }
```

```
    c[i] = a[p] + b[q];
  }
  return c;
}

/// @brief Min-plus convolution of two concave sequences.
template <class T>
std::vector<T> two_concave_minplus_convolution(const std::vector<T> &a,
                                               const std::vector<T> &b) {
  const int n = int(a.size());
  const int m = int(b.size());
  std::vector<T> negative_a(n);
  std::vector<T> negative_b(m);
  for (int i = 0; i < n; i++) {
    negative_a[i] = -a[i];
  }
  for (int i = 0; i < m; i++) {
    negative_b[i] = -b[i];
  }
  auto negative_c = two_concave_maxplus_convolution(negative_a,
negative_b);
  std::vector<T> c(n + m - 1);
  for (int i = 0; i < n + m - 1; i++)
    c[i] = -negative_c[i];
  return c;
}

/// @brief Max-plus convolution where b is concave.
template <class T>
std::vector<T> concave_maxplus_convolution(const std::vector<T> &a,
                                           const std::vector<T> &b) {
  if (a.empty())
    return b;
  if (b.empty())
    return a;
  const int n = int(a.size());
  const int m = int(b.size());
  const auto get = [&](const int &i, const int &j) -> T {
    return a[j] + b[i - j];
  };
  const auto select = [&](const int &i, const int &j, const int &k) -> bool
{
    if (i < k)
      return false;
    if (i - j >= m)
      return true;
    return get(i, j) <= get(i, k);
  };
  const auto amax = smawk(n + m - 1, n, select);
  std::vector<T> c(n + m - 1);
  for (int i = 0; i < n + m - 1; i++)
    c[i] = get(i, amax[i]);
  return c;
}

/// @brief Min-plus convolution where b is concave.
template <class T>
std::vector<T> concave_minplus_convolution(const std::vector<T> &a,
                                           const std::vector<T> &b) {
  const int n = int(a.size());
  const int m = int(b.size());
  std::vector<T> negative_a(n);
  std::vector<T> negative_b(m);
  for (int i = 0; i < n; i++) {
    negative_a[i] = -a[i];
  }
  for (int i = 0; i < m; i++) {
    negative_b[i] = -b[i];
  }
  auto negative_c = concave_maxplus_convolution(negative_a, negative_b);
  std::vector<T> c(n + m - 1);
  for (int i = 0; i < n + m - 1; i++)
    c[i] = -negative_c[i];
  return c;
}

} // namespace noya
```

smawk.hpp

```
// https://noshi91.github.io/Library/algorithm/smawk.cpp.html
#include <functional>
#include <numeric>
#include <vector>

namespace noya {

/// @brief SMAWK algorithm: compute row minima of a totally monotone
matrix.
/// @return Vector where ans[i] is the column index of the minimum in row
i.
template <class Select>
std::vector<int> smawk(const int row_size, const int col_size,
                       const Select &select) {
  const std::function<std::vector<int>(const std::vector<int> &,
                                       const std::vector<int> &)>
      solve = [&](const std::vector<int> &row,
                  const std::vector<int> &col) -> std::vector<int> {
    const int n = int(row.size());
    if (n == 0)
      return {};
    std::vector<int> c2;
    for (const int i : col) {
      while (!c2.empty() && select(row[c2.size() - 1], c2.back(), i))
        c2.pop_back();
      if (c2.size() < n)
        c2.push_back(i);
    }
    std::vector<int> r2;
    for (int i = 1; i < n; i += 2)
      r2.push_back(row[i]);
    const std::vector<int> a2 = solve(r2, c2);
    std::vector<int> ans(n);
    for (int i = 0; i != a2.size(); i += 1)
      ans[i * 2 + 1] = a2[i];
    int j = 0;
    for (int i = 0; i < n; i += 2) {
      ans[i] = c2[j];
      const int end = i + 1 == n ? c2.back() : ans[i + 1];
      while (c2[j] != end) {
        j += 1;
        if (select(row[i], ans[i], c2[j]))
          ans[i] = c2[j];
      }
    }
    return ans;
  };
  std::vector<int> row(row_size);
  std::iota(row.begin(), row.end(), 0);
  std::vector<int> col(col_size);
  std::iota(col.begin(), col.end(), 0);
  return solve(row, col);
}

} // namespace noya
```

Utility

debug.hpp

```
#include "atcoder/modint.hpp"

#include <iostream>
#include <ostream>
#include <ranges>
#include <sstream>
#include <string>
#include <tuple>
#include <utility>

template <class T, size_t size = std::tuple_size<T>::value>
std::string to_debug(T, std::string s = "")
  requires(not std::ranges::range<T>);
std::string to_debug(auto x)
  requires requires(std::ostream &os) { os << x; }
{
  return static_cast<std::ostringstream>(std::ostringstream() << x).str();
}
template <typename T>
std::string to_debug(T x)
  requires std::is_base_of_v<atcoder::internal::modint_base, T>
{
  return std::to_string(x.val());
}

std::string to_debug(std::ranges::range auto x, std::string s = "")
  requires(not std::is_same_v<decltype(x), std::string>)
{
  for (auto xi : x) {
    s += ", " + to_debug(xi);
  }
  return "[" + s.substr(s.empty() ? 0 : 2) + "]";
}
template <class T, size_t size>
std::string to_debug(T x, std::string s)
  requires(not std::ranges::range<T>)
{
  [&]<size_t... I>(std::index_sequence<I...>) {
    ((s += ", " + to_debug(get<I>(x))), ...);
  }(std::make_index_sequence<size>());
  return "(" + s.substr(s.empty() ? 0 : 2) + ")";
}
#define debug(...)
\
  std::cerr << "[" #__VA_ARGS__ "] = " << to_debug(std::tuple(__VA_ARGS__))
\
            << "\n"
```

AtCoder Library

convolution.hpp

```
#include <algorithm>
#include <array>
#include <cassert>
#include <type_traits>
#include <vector>

#include "atcoder/internal_bit"
#include "atcoder/modint"

namespace atcoder {

namespace internal {

template <class mint,
          int g = internal::primitive_root<mint::mod()>,
          internal::is_static_modint_t<mint>* = nullptr>
struct fft_info {
    static constexpr int rank2 = countr_zero_constexpr(mint::mod() - 1);
    std::array<mint, rank2 + 1> root;   // root[i]^(2^i) == 1
    std::array<mint, rank2 + 1> iroot;  // root[i] * iroot[i] == 1

    std::array<mint, std::max(0, rank2 - 2 + 1)> rate2;
```

```
    std::array<mint, std::max(0, rank2 - 2 + 1)> irate2;

    std::array<mint, std::max(0, rank2 - 3 + 1)> rate3;
    std::array<mint, std::max(0, rank2 - 3 + 1)> irate3;

    fft_info() {
        root[rank2] = mint(g).pow((mint::mod() - 1) >> rank2);
        iroot[rank2] = root[rank2].inv();
        for (int i = rank2 - 1; i >= 0; i--) {
            root[i] = root[i + 1] * root[i + 1];
            iroot[i] = iroot[i + 1] * iroot[i + 1];
        }

        {
            mint prod = 1, iprod = 1;
            for (int i = 0; i <= rank2 - 2; i++) {
                rate2[i] = root[i + 2] * prod;
                irate2[i] = iroot[i + 2] * iprod;
                prod *= iroot[i + 2];
                iprod *= root[i + 2];
            }
        }
        {
            mint prod = 1, iprod = 1;
            for (int i = 0; i <= rank2 - 3; i++) {
                rate3[i] = root[i + 3] * prod;
                irate3[i] = iroot[i + 3] * iprod;
                prod *= iroot[i + 3];
                iprod *= root[i + 3];
            }
        }
    }
};

template <class mint, internal::is_static_modint_t<mint>* = nullptr>
void butterfly(std::vector<mint>& a) {
    int n = int(a.size());
    int h = internal::countr_zero((unsigned int)n);

    static const fft_info<mint> info;

    int len = 0;  // a[i, i+(n>>len), i+2*(n>>len), ..] is transformed
    while (len < h) {
        if (h - len == 1) {
            int p = 1 << (h - len - 1);
            mint rot = 1;
            for (int s = 0; s < (1 << len); s++) {
                int offset = s << (h - len);
                for (int i = 0; i < p; i++) {
                    auto l = a[i + offset];
                    auto r = a[i + offset + p] * rot;
                    a[i + offset] = l + r;
                    a[i + offset + p] = l - r;
                }
                if (s + 1 != (1 << len))
                    rot *= info.rate2[countr_zero(~(unsigned int)(s))];
            }
            len++;
        } else {
            // 4-base
            int p = 1 << (h - len - 2);
            mint rot = 1, imag = info.root[2];
            for (int s = 0; s < (1 << len); s++) {
                mint rot2 = rot * rot;
                mint rot3 = rot2 * rot;
                int offset = s << (h - len);
                for (int i = 0; i < p; i++) {
                    auto mod2 = 1ULL * mint::mod() * mint::mod();
                    auto a0 = 1ULL * a[i + offset].val();
                    auto a1 = 1ULL * a[i + offset + p].val() * rot.val();
                    auto a2 = 1ULL * a[i + offset + 2 * p].val() *
rot2.val();
                    auto a3 = 1ULL * a[i + offset + 3 * p].val() *
rot3.val();
                    auto a1na3imag =
                        1ULL * mint(a1 + mod2 - a3).val() * imag.val();
                    auto na2 = mod2 - a2;
                    a[i + offset] = a0 + a2 + a1 + a3;
                    a[i + offset + 1 * p] = a0 + a2 + (2 * mod2 - (a1 +
a3));
                    a[i + offset + 2 * p] = a0 + na2 + a1na3imag;
                    a[i + offset + 3 * p] = a0 + na2 + (mod2 - a1na3imag);
                }
                if (s + 1 != (1 << len))
                    rot *= info.rate3[countr_zero(~(unsigned int)(s))];
            }
            len += 2;
        }
    }
}

template <class mint, internal::is_static_modint_t<mint>* = nullptr>
void butterfly_inv(std::vector<mint>& a) {
    int n = int(a.size());
    int h = internal::countr_zero((unsigned int)n);

    static const fft_info<mint> info;

    int len = h;  // a[i, i+(n>>len), i+2*(n>>len), ..] is transformed
    while (len) {
        if (len == 1) {
            int p = 1 << (h - len);
            mint irot = 1;
            for (int s = 0; s < (1 << (len - 1)); s++) {
                int offset = s << (h - len + 1);
                for (int i = 0; i < p; i++) {
                    auto l = a[i + offset];
                    auto r = a[i + offset + p];
                    a[i + offset] = l + r;
                    a[i + offset + p] =
                        (unsigned long long)((unsigned int)(l.val() -
r.val()) + mint::mod()) *
                        irot.val();
                    ;
                }
                if (s + 1 != (1 << (len - 1)))
                    irot *= info.irate2[countr_zero(~(unsigned int)(s))];
            }
            len--;
        } else {
            // 4-base
            int p = 1 << (h - len);
            mint irot = 1, iimag = info.iroot[2];
            for (int s = 0; s < (1 << (len - 2)); s++) {
                mint irot2 = irot * irot;
                mint irot3 = irot2 * irot;
                int offset = s << (h - len + 2);
                for (int i = 0; i < p; i++) {
                    auto a0 = 1ULL * a[i + offset + 0 * p].val();
                    auto a1 = 1ULL * a[i + offset + 1 * p].val();
                    auto a2 = 1ULL * a[i + offset + 2 * p].val();
                    auto a3 = 1ULL * a[i + offset + 3 * p].val();

                    auto a2na3iimag =
                        1ULL *
                        mint((mint::mod() + a2 - a3) * iimag.val()).val();

                    a[i + offset] = a0 + a1 + a2 + a3;
                    a[i + offset + 1 * p] =
                        (a0 + (mint::mod() - a1) + a2na3iimag) *
irot.val();
                    a[i + offset + 2 * p] =
                        (a0 + a1 + (mint::mod() - a2) + (mint::mod() - a3))
*
                        irot2.val();
                    a[i + offset + 3 * p] =
                        (a0 + (mint::mod() - a1) + (mint::mod() -
a2na3iimag)) *
                        irot3.val();
                }
                if (s + 1 != (1 << (len - 2)))
                    irot *= info.irate3[countr_zero(~(unsigned int)(s))];
            }
            len -= 2;
        }
    }
}

template <class mint, internal::is_static_modint_t<mint>* = nullptr>
std::vector<mint> convolution_naive(const std::vector<mint>& a,
                                    const std::vector<mint>& b) {
    int n = int(a.size()), m = int(b.size());
    std::vector<mint> ans(n + m - 1);
    if (n < m) {
        for (int j = 0; j < m; j++) {
            for (int i = 0; i < n; i++) {
                ans[i + j] += a[i] * b[j];
            }
        }
    } else {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                ans[i + j] += a[i] * b[j];
            }
        }
    }
    return ans;
}

template <class mint, internal::is_static_modint_t<mint>* = nullptr>
std::vector<mint> convolution_fft(std::vector<mint> a, std::vector<mint> b)
{
    int n = int(a.size()), m = int(b.size());
    int z = (int)internal::bit_ceil((unsigned int)(n + m - 1));
    a.resize(z);
    internal::butterfly(a);
    b.resize(z);
    internal::butterfly(b);
    for (int i = 0; i < z; i++) {
        a[i] *= b[i];
    }
    internal::butterfly_inv(a);
    a.resize(n + m - 1);
    mint iz = mint(z).inv();
    for (int i = 0; i < n + m - 1; i++) a[i] *= iz;
    return a;
}

}  // namespace internal

template <class mint, internal::is_static_modint_t<mint>* = nullptr>
std::vector<mint> convolution(std::vector<mint>&& a, std::vector<mint>&& b)
{
    int n = int(a.size()), m = int(b.size());
    if (!n || !m) return {};

    int z = (int)internal::bit_ceil((unsigned int)(n + m - 1));
    assert((mint::mod() - 1) % z == 0);

    if (std::min(n, m) <= 60) return convolution_naive(std::move(a),
std::move(b));
    return internal::convolution_fft(std::move(a), std::move(b));
}
template <class mint, internal::is_static_modint_t<mint>* = nullptr>
std::vector<mint> convolution(const std::vector<mint>& a,
                              const std::vector<mint>& b) {
    int n = int(a.size()), m = int(b.size());
    if (!n || !m) return {};

    int z = (int)internal::bit_ceil((unsigned int)(n + m - 1));
    assert((mint::mod() - 1) % z == 0);

    if (std::min(n, m) <= 60) return convolution_naive(a, b);
    return internal::convolution_fft(a, b);
}

template <unsigned int mod = 998244353,
          class T,
          std::enable_if_t<internal::is_integral<T>::value>* = nullptr>
std::vector<T> convolution(const std::vector<T>& a, const std::vector<T>&
b) {
    int n = int(a.size()), m = int(b.size());
    if (!n || !m) return {};
```

```
    using mint = static_modint<mod>;

    int z = (int)internal::bit_ceil((unsigned int)(n + m - 1));
    assert((mint::mod() - 1) % z == 0);

    std::vector<mint> a2(n), b2(m);
    for (int i = 0; i < n; i++) {
        a2[i] = mint(a[i]);
    }
    for (int i = 0; i < m; i++) {
        b2[i] = mint(b[i]);
    }
    auto c2 = convolution(std::move(a2), std::move(b2));
    std::vector<T> c(n + m - 1);
    for (int i = 0; i < n + m - 1; i++) {
        c[i] = c2[i].val();
    }
    return c;
}

std::vector<long long> convolution_ll(const std::vector<long long>& a,
                                      const std::vector<long long>& b) {
    int n = int(a.size()), m = int(b.size());
    if (!n || !m) return {};

    static constexpr unsigned long long MOD1 = 754974721;  // 2^24
    static constexpr unsigned long long MOD2 = 167772161;  // 2^25
    static constexpr unsigned long long MOD3 = 469762049;  // 2^26
    static constexpr unsigned long long M2M3 = MOD2 * MOD3;
    static constexpr unsigned long long M1M3 = MOD1 * MOD3;
    static constexpr unsigned long long M1M2 = MOD1 * MOD2;
    static constexpr unsigned long long M1M2M3 = MOD1 * MOD2 * MOD3;

    static constexpr unsigned long long i1 =
        internal::inv_gcd(MOD2 * MOD3, MOD1).second;
    static constexpr unsigned long long i2 =
        internal::inv_gcd(MOD1 * MOD3, MOD2).second;
    static constexpr unsigned long long i3 =
        internal::inv_gcd(MOD1 * MOD2, MOD3).second;

    static constexpr int MAX_AB_BIT = 24;
    static_assert(MOD1 % (1ull << MAX_AB_BIT) == 1, "MOD1 isn't enough to
support an array length of 2^24.");
    static_assert(MOD2 % (1ull << MAX_AB_BIT) == 1, "MOD2 isn't enough to
support an array length of 2^24.");
    static_assert(MOD3 % (1ull << MAX_AB_BIT) == 1, "MOD3 isn't enough to
support an array length of 2^24.");
    assert(n + m - 1 <= (1 << MAX_AB_BIT));

    auto c1 = convolution<MOD1>(a, b);
    auto c2 = convolution<MOD2>(a, b);
    auto c3 = convolution<MOD3>(a, b);

    std::vector<long long> c(n + m - 1);
    for (int i = 0; i < n + m - 1; i++) {
        unsigned long long x = 0;
        x += (c1[i] * i1) % MOD1 * M2M3;
        x += (c2[i] * i2) % MOD2 * M1M3;
        x += (c3[i] * i3) % MOD3 * M1M2;
        // B = 2^63, -B <= x, r(real value) < B
        // (x, x - M, x - 2M, or x - 3M) = r (mod 2B)
        // r = c1[i] (mod MOD1)
        // focus on MOD1
        // r = x, x - M', x - 2M', x - 3M' (M' = M % 2^64) (mod 2B)
        // r = x,
        //     x - M' + (0 or 2B),
        //     x - 2M' + (0, 2B or 4B),
        //     x - 3M' + (0, 2B, 4B or 6B) (without mod!)
        // (r - x) = 0, (0)
        //           - M' + (0 or 2B), (1)
        //           -2M' + (0 or 2B or 4B), (2)
        //           -3M' + (0 or 2B or 4B or 6B) (3) (mod MOD1)
        // we checked that
        //   ((1) mod MOD1) mod 5 = 2
        //   ((2) mod MOD1) mod 5 = 3
        //   ((3) mod MOD1) mod 5 = 4
        long long diff =
            c1[i] - internal::safe_mod((long long)(x), (long long)(MOD1));
        if (diff < 0) diff += MOD1;
        static constexpr unsigned long long offset[5] = {
            0, 0, M1M2M3, 2 * M1M2M3, 3 * M1M2M3};
        x -= offset[diff % 5];
        c[i] = x;
    }

    return c;
}

}  // namespace atcoder
```

dsu.hpp

```
#include <algorithm>
#include <cassert>
#include <vector>

namespace atcoder {

// Implement (union by size) + (path compression)
// Reference:
// Zvi Galil and Giuseppe F. Italiano,
// Data structures and algorithms for disjoint set union problems
struct dsu {
  public:
    dsu() : _n(0) {}
    explicit dsu(int n) : _n(n), parent_or_size(n, -1) {}

    int merge(int a, int b) {
        assert(0 <= a && a < _n);
        assert(0 <= b && b < _n);
        int x = leader(a), y = leader(b);
        if (x == y) return x;
        if (-parent_or_size[x] < -parent_or_size[y]) std::swap(x, y);
        parent_or_size[x] += parent_or_size[y];
        parent_or_size[y] = x;
        return x;
    }

    bool same(int a, int b) {
        assert(0 <= a && a < _n);
        assert(0 <= b && b < _n);
        return leader(a) == leader(b);
    }

    int leader(int a) {
        assert(0 <= a && a < _n);
        return _leader(a);
    }

    int size(int a) {
        assert(0 <= a && a < _n);
        return -parent_or_size[leader(a)];
    }

    std::vector<std::vector<int>> groups() {
        std::vector<int> leader_buf(_n), group_size(_n);
        for (int i = 0; i < _n; i++) {
            leader_buf[i] = leader(i);
            group_size[leader_buf[i]]++;
        }
        std::vector<std::vector<int>> result(_n);
        for (int i = 0; i < _n; i++) {
            result[i].reserve(group_size[i]);
        }
        for (int i = 0; i < _n; i++) {
            result[leader_buf[i]].push_back(i);
        }
        result.erase(
            std::remove_if(result.begin(), result.end(),
                           [&](const std::vector<int>& v) { return
v.empty(); }),
            result.end());
        return result;
    }

  private:
    int _n;
    // root node: -1 * component size
    // otherwise: parent
    std::vector<int> parent_or_size;

    int _leader(int a) {
        if (parent_or_size[a] < 0) return a;
        return parent_or_size[a] = _leader(parent_or_size[a]);
    }
};

}  // namespace atcoder
```

fenwicktree.hpp

```
#include <cassert>
#include <vector>

#include "atcoder/internal_type_traits"

namespace atcoder {

// Reference: https://en.wikipedia.org/wiki/Fenwick_tree
template <class T> struct fenwick_tree {
    using U = internal::to_unsigned_t<T>;

  public:
    fenwick_tree() : _n(0) {}
    explicit fenwick_tree(int n) : _n(n), data(n) {}

    void add(int p, T x) {
        assert(0 <= p && p < _n);
        p++;
        while (p <= _n) {
            data[p - 1] += U(x);
            p += p & -p;
        }
    }

    T sum(int l, int r) {
        assert(0 <= l && l <= r && r <= _n);
        return sum(r) - sum(l);
    }

  private:
    int _n;
    std::vector<U> data;

    U sum(int r) {
        U s = 0;
        while (r > 0) {
            s += data[r - 1];
            r -= r & -r;
        }
        return s;
    }
};

}  // namespace atcoder
```

lazysegtree.hpp

```
#include <algorithm>
#include <cassert>
#include <functional>
#include <vector>

#include "atcoder/internal_bit"
```

```
namespace atcoder {

#if __cplusplus >= 201703L

template <class S,
          auto op,
          auto e,
          class F,
          auto mapping,
          auto composition,
          auto id>
struct lazy_segtree {
    static_assert(std::is_convertible_v<decltype(op), std::function<S(S,
S)>>,
                  "op must work as S(S, S)");
    static_assert(std::is_convertible_v<decltype(e), std::function<S()>>,
                  "e must work as S()");
    static_assert(
        std::is_convertible_v<decltype(mapping), std::function<S(F, S)>>,
        "mapping must work as S(F, S)");
    static_assert(
        std::is_convertible_v<decltype(composition), std::function<F(F,
F)>>,
        "composition must work as F(F, F)");
    static_assert(std::is_convertible_v<decltype(id), std::function<F()>>,
                  "id must work as F()");

#else

template <class S,
          S (*op)(S, S),
          S (*e)(),
          class F,
          S (*mapping)(F, S),
          F (*composition)(F, F),
          F (*id)()>
struct lazy_segtree {

#endif

  public:
    lazy_segtree() : lazy_segtree(0) {}
    explicit lazy_segtree(int n) : lazy_segtree(std::vector<S>(n, e())) {}
    explicit lazy_segtree(const std::vector<S>& v) : _n(int(v.size())) {
        size = (int)internal::bit_ceil((unsigned int)(_n));
        log = internal::countr_zero((unsigned int)size);
        d = std::vector<S>(2 * size, e());
        lz = std::vector<F>(size, id());
        for (int i = 0; i < _n; i++) d[size + i] = v[i];
        for (int i = size - 1; i >= 1; i--) {
            update(i);
        }
    }

    void set(int p, S x) {
        assert(0 <= p && p < _n);
        p += size;
        for (int i = log; i >= 1; i--) push(p >> i);
        d[p] = x;
        for (int i = 1; i <= log; i++) update(p >> i);
    }

    S get(int p) {
        assert(0 <= p && p < _n);
        p += size;
        for (int i = log; i >= 1; i--) push(p >> i);
        return d[p];
    }

    S prod(int l, int r) {
        assert(0 <= l && l <= r && r <= _n);
        if (l == r) return e();

        l += size;
        r += size;

        for (int i = log; i >= 1; i--) {
            if (((l >> i) << i) != l) push(l >> i);
            if (((r >> i) << i) != r) push((r - 1) >> i);
        }

        S sml = e(), smr = e();
        while (l < r) {
            if (l & 1) sml = op(sml, d[l++]);
            if (r & 1) smr = op(d[--r], smr);
            l >>= 1;
            r >>= 1;
        }

        return op(sml, smr);
    }

    S all_prod() { return d[1]; }

    void apply(int p, F f) {
        assert(0 <= p && p < _n);
        p += size;
        for (int i = log; i >= 1; i--) push(p >> i);
        d[p] = mapping(f, d[p]);
        for (int i = 1; i <= log; i++) update(p >> i);
    }
    void apply(int l, int r, F f) {
        assert(0 <= l && l <= r && r <= _n);
        if (l == r) return;

        l += size;
        r += size;

        for (int i = log; i >= 1; i--) {
            if (((l >> i) << i) != l) push(l >> i);
            if (((r >> i) << i) != r) push((r - 1) >> i);
        }

        {
            int l2 = l, r2 = r;
            while (l < r) {
                if (l & 1) all_apply(l++, f);
                if (r & 1) all_apply(--r, f);
                l >>= 1;
                r >>= 1;
            }
            l = l2;
            r = r2;
        }

        for (int i = 1; i <= log; i++) {
            if (((l >> i) << i) != l) update(l >> i);
            if (((r >> i) << i) != r) update((r - 1) >> i);
        }
    }

    template <bool (*g)(S)> int max_right(int l) {
        return max_right(l, [](S x) { return g(x); });
    }
    template <class G> int max_right(int l, G g) {
        assert(0 <= l && l <= _n);
        assert(g(e()));
        if (l == _n) return _n;
        l += size;
        for (int i = log; i >= 1; i--) push(l >> i);
        S sm = e();
        do {
            while (l % 2 == 0) l >>= 1;
            if (!g(op(sm, d[l]))) {
                while (l < size) {
                    push(l);
                    l = (2 * l);
                    if (g(op(sm, d[l]))) {
                        sm = op(sm, d[l]);
                        l++;
                    }
                }
                return l - size;
            }
            sm = op(sm, d[l]);
            l++;
        } while ((l & -l) != l);
        return _n;
    }

    template <bool (*g)(S)> int min_left(int r) {
        return min_left(r, [](S x) { return g(x); });
    }
    template <class G> int min_left(int r, G g) {
        assert(0 <= r && r <= _n);
        assert(g(e()));
        if (r == 0) return 0;
        r += size;
        for (int i = log; i >= 1; i--) push((r - 1) >> i);
        S sm = e();
        do {
            r--;
            while (r > 1 && (r % 2)) r >>= 1;
            if (!g(op(d[r], sm))) {
                while (r < size) {
                    push(r);
                    r = (2 * r + 1);
                    if (g(op(d[r], sm))) {
                        sm = op(d[r], sm);
                        r--;
                    }
                }
                return r + 1 - size;
            }
            sm = op(d[r], sm);
        } while ((r & -r) != r);
        return 0;
    }

  private:
    int _n, size, log;
    std::vector<S> d;
    std::vector<F> lz;

    void update(int k) { d[k] = op(d[2 * k], d[2 * k + 1]); }
    void all_apply(int k, F f) {
        d[k] = mapping(f, d[k]);
        if (k < size) lz[k] = composition(f, lz[k]);
    }
    void push(int k) {
        all_apply(2 * k, lz[k]);
        all_apply(2 * k + 1, lz[k]);
        lz[k] = id();
    }
};

}  // namespace atcoder
```

math.hpp

```
#include <algorithm>
#include <cassert>
#include <tuple>
#include <vector>

#include "atcoder/internal_math"

namespace atcoder {

long long pow_mod(long long x, long long n, int m) {
    assert(0 <= n && 1 <= m);
    if (m == 1) return 0;
    internal::barrett bt((unsigned int)(m));
    unsigned int r = 1, y = (unsigned int)(internal::safe_mod(x, m));
    while (n) {
        if (n & 1) r = bt.mul(r, y);
        y = bt.mul(y, y);
        n >>= 1;
    }
    return r;
}
```

```
long long inv_mod(long long x, long long m) {
    assert(1 <= m);
    auto z = internal::inv_gcd(x, m);
    assert(z.first == 1);
    return z.second;
}

// (rem, mod)
std::pair<long long, long long> crt(const std::vector<long long>& r,
                                    const std::vector<long long>& m) {
    assert(r.size() == m.size());
    int n = int(r.size());
    // Contracts: 0 <= r0 < m0
    long long r0 = 0, m0 = 1;
    for (int i = 0; i < n; i++) {
        assert(1 <= m[i]);
        long long r1 = internal::safe_mod(r[i], m[i]), m1 = m[i];
        if (m0 < m1) {
            std::swap(r0, r1);
            std::swap(m0, m1);
        }
        if (m0 % m1 == 0) {
            if (r0 % m1 != r1) return {0, 0};
            continue;
        }
        // assume: m0 > m1, lcm(m0, m1) >= 2 * max(m0, m1)

        // (r0, m0), (r1, m1) -> (r2, m2 = lcm(m0, m1));
        // r2 % m0 = r0
        // r2 % m1 = r1
        // -> (r0 + x*m0) % m1 = r1
        // -> x*u0*g = r1-r0 (mod u1*g) (u0*g = m0, u1*g = m1)
        // -> x = (r1 - r0) / g * inv(u0) (mod u1)

        // im = inv(u0) (mod u1) (0 <= im < u1)
        long long g, im;
        std::tie(g, im) = internal::inv_gcd(m0, m1);

        long long u1 = (m1 / g);
        // |r1 - r0| < (m0 + m1) <= lcm(m0, m1)
        if ((r1 - r0) % g) return {0, 0};

        // u1 * u1 <= m1 * m1 / g / g <= m0 * m1 / g = lcm(m0, m1)
        long long x = (r1 - r0) / g % u1 * im % u1;

        // |r0| + |m0 * x|
        // < m0 + m0 * (u1 - 1)
        // = m0 + m0 * m1 / g - m0
        // = lcm(m0, m1)
        r0 += x * m0;
        m0 *= u1;  // -> lcm(m0, m1)
        if (r0 < 0) r0 += m0;
    }
    return {r0, m0};
}

long long floor_sum(long long n, long long m, long long a, long long b) {
    assert(0 <= n && n < (1LL << 32));
    assert(1 <= m && m < (1LL << 32));
    unsigned long long ans = 0;
    if (a < 0) {
        unsigned long long a2 = internal::safe_mod(a, m);
        ans -= 1ULL * n * (n - 1) / 2 * ((a2 - a) / m);
        a = a2;
    }
    if (b < 0) {
        unsigned long long b2 = internal::safe_mod(b, m);
        ans -= 1ULL * n * ((b2 - b) / m);
        b = b2;
    }
    return ans + internal::floor_sum_unsigned(n, m, a, b);
}

}  // namespace atcoder
```

maxflow.hpp

```
#include <algorithm>
#include <cassert>
#include <limits>
#include <queue>
#include <vector>

#include "atcoder/internal_queue"

namespace atcoder {

template <class Cap> struct mf_graph {
  public:
    mf_graph() : _n(0) {}
    explicit mf_graph(int n) : _n(n), g(n) {}

    int add_edge(int from, int to, Cap cap) {
        assert(0 <= from && from < _n);
        assert(0 <= to && to < _n);
        assert(0 <= cap);
        int m = int(pos.size());
        pos.push_back({from, int(g[from].size())});
        int from_id = int(g[from].size());
        int to_id = int(g[to].size());
        if (from == to) to_id++;
        g[from].push_back(_edge{to, to_id, cap});
        g[to].push_back(_edge{from, from_id, 0});
        return m;
    }

    struct edge {
        int from, to;
        Cap cap, flow;
    };

    edge get_edge(int i) {
        int m = int(pos.size());
        assert(0 <= i && i < m);
        auto _e = g[pos[i].first][pos[i].second];
        auto _re = g[_e.to][_e.rev];
        return edge{pos[i].first, _e.to, _e.cap + _re.cap, _re.cap};
    }
    std::vector<edge> edges() {
        int m = int(pos.size());
        std::vector<edge> result;
        for (int i = 0; i < m; i++) {
            result.push_back(get_edge(i));
        }
        return result;
    }
    void change_edge(int i, Cap new_cap, Cap new_flow) {
        int m = int(pos.size());
        assert(0 <= i && i < m);
        assert(0 <= new_flow && new_flow <= new_cap);
        auto& _e = g[pos[i].first][pos[i].second];
        auto& _re = g[_e.to][_e.rev];
        _e.cap = new_cap - new_flow;
        _re.cap = new_flow;
    }

    Cap flow(int s, int t) {
        return flow(s, t, std::numeric_limits<Cap>::max());
    }
    Cap flow(int s, int t, Cap flow_limit) {
        assert(0 <= s && s < _n);
        assert(0 <= t && t < _n);
        assert(s != t);

        std::vector<int> level(_n), iter(_n);
        internal::simple_queue<int> que;

        auto bfs = [&]() {
            std::fill(level.begin(), level.end(), -1);
            level[s] = 0;
            que.clear();
            que.push(s);
            while (!que.empty()) {
                int v = que.front();
                que.pop();
                for (auto e : g[v]) {
                    if (e.cap == 0 || level[e.to] >= 0) continue;
                    level[e.to] = level[v] + 1;
                    if (e.to == t) return;
                    que.push(e.to);
                }
            }
        };
        auto dfs = [&](auto self, int v, Cap up) {
            if (v == s) return up;
            Cap res = 0;
            int level_v = level[v];
            for (int& i = iter[v]; i < int(g[v].size()); i++) {
                _edge& e = g[v][i];
                if (level_v <= level[e.to] || g[e.to][e.rev].cap == 0)
continue;
                Cap d =
                    self(self, e.to, std::min(up - res, g[e.to]
[e.rev].cap));
                if (d <= 0) continue;
                g[v][i].cap += d;
                g[e.to][e.rev].cap -= d;
                res += d;
                if (res == up) return res;
            }
            level[v] = _n;
            return res;
        };

        Cap flow = 0;
        while (flow < flow_limit) {
            bfs();
            if (level[t] == -1) break;
            std::fill(iter.begin(), iter.end(), 0);
            Cap f = dfs(dfs, t, flow_limit - flow);
            if (!f) break;
            flow += f;
        }
        return flow;
    }

    std::vector<bool> min_cut(int s) {
        std::vector<bool> visited(_n);
        internal::simple_queue<int> que;
        que.push(s);
        while (!que.empty()) {
            int p = que.front();
            que.pop();
            visited[p] = true;
            for (auto e : g[p]) {
                if (e.cap && !visited[e.to]) {
                    visited[e.to] = true;
                    que.push(e.to);
                }
            }
        }
        return visited;
    }

  private:
    int _n;
    struct _edge {
        int to, rev;
        Cap cap;
    };
    std::vector<std::pair<int, int>> pos;
    std::vector<std::vector<_edge>> g;
};

}  // namespace atcoder
```

mincostflow.hpp

```
#include <algorithm>
#include <cassert>
#include <limits>
#include <queue>
#include <vector>

#include "atcoder/internal_csr"
#include "atcoder/internal_queue"

namespace atcoder {

template <class Cap, class Cost> struct mcf_graph {
  public:
    mcf_graph() {}
    explicit mcf_graph(int n) : _n(n) {}

    int add_edge(int from, int to, Cap cap, Cost cost) {
        assert(0 <= from && from < _n);
        assert(0 <= to && to < _n);
        assert(0 <= cap);
        assert(0 <= cost);
        int m = int(_edges.size());
        _edges.push_back({from, to, cap, 0, cost});
        return m;
    }

    struct edge {
        int from, to;
        Cap cap, flow;
        Cost cost;
    };

    edge get_edge(int i) {
        int m = int(_edges.size());
        assert(0 <= i && i < m);
        return _edges[i];
    }
    std::vector<edge> edges() { return _edges; }

    std::pair<Cap, Cost> flow(int s, int t) {
        return flow(s, t, std::numeric_limits<Cap>::max());
    }
    std::pair<Cap, Cost> flow(int s, int t, Cap flow_limit) {
        return slope(s, t, flow_limit).back();
    }
    std::vector<std::pair<Cap, Cost>> slope(int s, int t) {
        return slope(s, t, std::numeric_limits<Cap>::max());
    }
    std::vector<std::pair<Cap, Cost>> slope(int s, int t, Cap flow_limit) {
        assert(0 <= s && s < _n);
        assert(0 <= t && t < _n);
        assert(s != t);

        int m = int(_edges.size());
        std::vector<int> edge_idx(m);

        auto g = [&]() {
            std::vector<int> degree(_n), redge_idx(m);
            std::vector<std::pair<int, _edge>> elist;
            elist.reserve(2 * m);
            for (int i = 0; i < m; i++) {
                auto e = _edges[i];
                edge_idx[i] = degree[e.from]++;
                redge_idx[i] = degree[e.to]++;
                elist.push_back({e.from, {e.to, -1, e.cap - e.flow,
e.cost}});
                elist.push_back({e.to, {e.from, -1, e.flow, -e.cost}});
            }
            auto _g = internal::csr<_edge>(_n, elist);
            for (int i = 0; i < m; i++) {
                auto e = _edges[i];
                edge_idx[i] += _g.start[e.from];
                redge_idx[i] += _g.start[e.to];
                _g.elist[edge_idx[i]].rev = redge_idx[i];
                _g.elist[redge_idx[i]].rev = edge_idx[i];
            }
            return _g;
        }();

        auto result = slope(g, s, t, flow_limit);

        for (int i = 0; i < m; i++) {
            auto e = g.elist[edge_idx[i]];
            _edges[i].flow = _edges[i].cap - e.cap;
        }

        return result;
    }

  private:
    int _n;
    std::vector<edge> _edges;

    // inside edge
    struct _edge {
        int to, rev;
        Cap cap;
        Cost cost;
    };

    std::vector<std::pair<Cap, Cost>> slope(internal::csr<_edge>& g,
                                            int s,
                                            int t,
                                            Cap flow_limit) {
        // variants (C = maxcost):
        // -(n-1)C <= dual[s] <= dual[i] <= dual[t] = 0
        // reduced cost (= e.cost + dual[e.from] - dual[e.to]) >= 0 for all
edge

        // dual_dist[i] = (dual[i], dist[i])
        std::vector<std::pair<Cost, Cost>> dual_dist(_n);
        std::vector<int> prev_e(_n);
        std::vector<bool> vis(_n);
        struct Q {
            Cost key;
            int to;
            bool operator<(Q r) const { return key > r.key; }
        };
        std::vector<int> que_min;
        std::vector<Q> que;
        auto dual_ref = [&]() {
            for (int i = 0; i < _n; i++) {
                dual_dist[i].second = std::numeric_limits<Cost>::max();
            }
            std::fill(vis.begin(), vis.end(), false);
            que_min.clear();
            que.clear();

            // que[0..heap_r) was heapified
            size_t heap_r = 0;

            dual_dist[s].second = 0;
            que_min.push_back(s);
            while (!que_min.empty() || !que.empty()) {
                int v;
                if (!que_min.empty()) {
                    v = que_min.back();
                    que_min.pop_back();
                } else {
                    while (heap_r < que.size()) {
                        heap_r++;
                        std::push_heap(que.begin(), que.begin() + heap_r);
                    }
                    v = que.front().to;
                    std::pop_heap(que.begin(), que.end());
                    que.pop_back();
                    heap_r--;
                }
                if (vis[v]) continue;
                vis[v] = true;
                if (v == t) break;
                // dist[v] = shortest(s, v) + dual[s] - dual[v]
                // dist[v] >= 0 (all reduced cost are positive)
                // dist[v] <= (n-1)C
                Cost dual_v = dual_dist[v].first, dist_v =
dual_dist[v].second;
                for (int i = g.start[v]; i < g.start[v + 1]; i++) {
                    auto e = g.elist[i];
                    if (!e.cap) continue;
                    // |-dual[e.to] + dual[v]| <= (n-1)C
                    // cost <= C - -(n-1)C + 0 = nC
                    Cost cost = e.cost - dual_dist[e.to].first + dual_v;
                    if (dual_dist[e.to].second - dist_v > cost) {
                        Cost dist_to = dist_v + cost;
                        dual_dist[e.to].second = dist_to;
                        prev_e[e.to] = e.rev;
                        if (dist_to == dist_v) {
                            que_min.push_back(e.to);
                        } else {
                            que.push_back(Q{dist_to, e.to});
                        }
                    }
                }
            }
            if (!vis[t]) {
                return false;
            }

            for (int v = 0; v < _n; v++) {
                if (!vis[v]) continue;
                // dual[v] = dual[v] - dist[t] + dist[v]
                //         = dual[v] - (shortest(s, t) + dual[s] - dual[t])
+
                //         (shortest(s, v) + dual[s] - dual[v]) = -
shortest(s,
                //         t) + dual[t] + shortest(s, v) = shortest(s, v) -
                //         shortest(s, t) >= 0 - (n-1)C
                dual_dist[v].first -= dual_dist[t].second -
dual_dist[v].second;
            }
            return true;
        };
        Cap flow = 0;
        Cost cost = 0, prev_cost_per_flow = -1;
        std::vector<std::pair<Cap, Cost>> result = {{Cap(0), Cost(0)}};
        while (flow < flow_limit) {
            if (!dual_ref()) break;
            Cap c = flow_limit - flow;
            for (int v = t; v != s; v = g.elist[prev_e[v]].to) {
                c = std::min(c, g.elist[g.elist[prev_e[v]].rev].cap);
            }
            for (int v = t; v != s; v = g.elist[prev_e[v]].to) {
                auto& e = g.elist[prev_e[v]];
                e.cap += c;
                g.elist[e.rev].cap -= c;
            }
            Cost d = -dual_dist[s].first;
            flow += c;
            cost += c * d;
            if (prev_cost_per_flow == d) {
                result.pop_back();
            }
            result.push_back({flow, cost});
            prev_cost_per_flow = d;
        }
        return result;
    }
};

}  // namespace atcoder
```

modint.hpp

```
#include <cassert>
#include <numeric>
#include <type_traits>

#ifdef _MSC_VER
```

```
#include <intrin.h>
#endif

#include "atcoder/internal_math"
#include "atcoder/internal_type_traits"

namespace atcoder {

namespace internal {

struct modint_base {};
struct static_modint_base : modint_base {};

template <class T> using is_modint = std::is_base_of<modint_base, T>;
template <class T> using is_modint_t =
std::enable_if_t<is_modint<T>::value>;

}  // namespace internal

template <int m, std::enable_if_t<(1 <= m)>* = nullptr>
struct static_modint : internal::static_modint_base {
    using mint = static_modint;

  public:
    static constexpr int mod() { return m; }
    static mint raw(int v) {
        mint x;
        x._v = v;
        return x;
    }

    static_modint() : _v(0) {}
    template <class T, internal::is_signed_int_t<T>* = nullptr>
    static_modint(T v) {
        long long x = (long long)(v % (long long)(umod()));
        if (x < 0) x += umod();
        _v = (unsigned int)(x);
    }
    template <class T, internal::is_unsigned_int_t<T>* = nullptr>
    static_modint(T v) {
        _v = (unsigned int)(v % umod());
    }

    int val() const { return _v; }

    mint& operator++() {
        _v++;
        if (_v == umod()) _v = 0;
        return *this;
    }
    mint& operator--() {
        if (_v == 0) _v = umod();
        _v--;
        return *this;
    }
    mint operator++(int) {
        mint result = *this;
        ++*this;
        return result;
    }
    mint operator--(int) {
        mint result = *this;
        --*this;
        return result;
    }

    mint& operator+=(const mint& rhs) {
        _v += rhs._v;
        if (_v >= umod()) _v -= umod();
        return *this;
    }
    mint& operator-=(const mint& rhs) {
        _v -= rhs._v;
        if (_v >= umod()) _v += umod();
        return *this;
    }
    mint& operator*=(const mint& rhs) {
        unsigned long long z = _v;
        z *= rhs._v;
        _v = (unsigned int)(z % umod());
        return *this;
    }
    mint& operator/=(const mint& rhs) { return *this = *this * rhs.inv(); }

    mint operator+() const { return *this; }
    mint operator-() const { return mint() - *this; }

    mint pow(long long n) const {
        assert(0 <= n);
        mint x = *this, r = 1;
        while (n) {
            if (n & 1) r *= x;
            x *= x;
            n >>= 1;
        }
        return r;
    }
    mint inv() const {
        if (prime) {
            assert(_v);
            return pow(umod() - 2);
        } else {
            auto eg = internal::inv_gcd(_v, m);
            assert(eg.first == 1);
            return eg.second;
        }
    }

    friend mint operator+(const mint& lhs, const mint& rhs) {
        return mint(lhs) += rhs;
    }
    friend mint operator-(const mint& lhs, const mint& rhs) {
        return mint(lhs) -= rhs;
    }
    friend mint operator*(const mint& lhs, const mint& rhs) {
        return mint(lhs) *= rhs;
    }
    friend mint operator/(const mint& lhs, const mint& rhs) {
        return mint(lhs) /= rhs;
    }
    friend bool operator==(const mint& lhs, const mint& rhs) {
        return lhs._v == rhs._v;
    }
    friend bool operator!=(const mint& lhs, const mint& rhs) {
        return lhs._v != rhs._v;
    }

  private:
    unsigned int _v;
    static constexpr unsigned int umod() { return m; }
    static constexpr bool prime = internal::is_prime<m>;
};

template <int id> struct dynamic_modint : internal::modint_base {
    using mint = dynamic_modint;

  public:
    static int mod() { return (int)(bt.umod()); }
    static void set_mod(int m) {
        assert(1 <= m);
        bt = internal::barrett(m);
    }
    static mint raw(int v) {
        mint x;
        x._v = v;
        return x;
    }

    dynamic_modint() : _v(0) {}
    template <class T, internal::is_signed_int_t<T>* = nullptr>
    dynamic_modint(T v) {
        long long x = (long long)(v % (long long)(mod()));
        if (x < 0) x += mod();
        _v = (unsigned int)(x);
    }
    template <class T, internal::is_unsigned_int_t<T>* = nullptr>
    dynamic_modint(T v) {
        _v = (unsigned int)(v % mod());
    }

    int val() const { return _v; }

    mint& operator++() {
        _v++;
        if (_v == umod()) _v = 0;
        return *this;
    }
    mint& operator--() {
        if (_v == 0) _v = umod();
        _v--;
        return *this;
    }
    mint operator++(int) {
        mint result = *this;
        ++*this;
        return result;
    }
    mint operator--(int) {
        mint result = *this;
        --*this;
        return result;
    }

    mint& operator+=(const mint& rhs) {
        _v += rhs._v;
        if (_v >= umod()) _v -= umod();
        return *this;
    }
    mint& operator-=(const mint& rhs) {
        _v += mod() - rhs._v;
        if (_v >= umod()) _v -= umod();
        return *this;
    }
    mint& operator*=(const mint& rhs) {
        _v = bt.mul(_v, rhs._v);
        return *this;
    }
    mint& operator/=(const mint& rhs) { return *this = *this * rhs.inv(); }

    mint operator+() const { return *this; }
    mint operator-() const { return mint() - *this; }

    mint pow(long long n) const {
        assert(0 <= n);
        mint x = *this, r = 1;
        while (n) {
            if (n & 1) r *= x;
            x *= x;
            n >>= 1;
        }
        return r;
    }
    mint inv() const {
        auto eg = internal::inv_gcd(_v, mod());
        assert(eg.first == 1);
        return eg.second;
    }

    friend mint operator+(const mint& lhs, const mint& rhs) {
        return mint(lhs) += rhs;
    }
    friend mint operator-(const mint& lhs, const mint& rhs) {
        return mint(lhs) -= rhs;
    }
    friend mint operator*(const mint& lhs, const mint& rhs) {
        return mint(lhs) *= rhs;
    }
    friend mint operator/(const mint& lhs, const mint& rhs) {
        return mint(lhs) /= rhs;
    }
    friend bool operator==(const mint& lhs, const mint& rhs) {
        return lhs._v == rhs._v;
    }
    friend bool operator!=(const mint& lhs, const mint& rhs) {
```

```
        return lhs._v != rhs._v;
    }

  private:
    unsigned int _v;
    static internal::barrett bt;
    static unsigned int umod() { return bt.umod(); }
};
template <int id> internal::barrett dynamic_modint<id>::bt(998244353);

using modint998244353 = static_modint<998244353>;
using modint1000000007 = static_modint<1000000007>;
using modint = dynamic_modint<-1>;

namespace internal {

template <class T>
using is_static_modint = std::is_base_of<internal::static_modint_base, T>;

template <class T>
using is_static_modint_t = std::enable_if_t<is_static_modint<T>::value>;

template <class> struct is_dynamic_modint : public std::false_type {};
template <int id>
struct is_dynamic_modint<dynamic_modint<id>> : public std::true_type {};

template <class T>
using is_dynamic_modint_t = std::enable_if_t<is_dynamic_modint<T>::value>;

}  // namespace internal

}  // namespace atcoder
```

scc.hpp

```
#include <algorithm>
#include <cassert>
#include <vector>

#include "atcoder/internal_scc"

namespace atcoder {

struct scc_graph {
  public:
    scc_graph() : internal(0) {}
    explicit scc_graph(int n) : internal(n) {}

    void add_edge(int from, int to) {
        int n = internal.num_vertices();
        assert(0 <= from && from < n);
        assert(0 <= to && to < n);
        internal.add_edge(from, to);
    }

    std::vector<std::vector<int>> scc() { return internal.scc(); }

  private:
    internal::scc_graph internal;
};

}  // namespace atcoder
```

segtree.hpp

```
#include <algorithm>
#include <cassert>
#include <functional>
#include <vector>

#include "atcoder/internal_bit"

namespace atcoder {

#if __cplusplus >= 201703L

template <class S, auto op, auto e> struct segtree {
    static_assert(std::is_convertible_v<decltype(op), std::function<S(S,
S)>>,
                  "op must work as S(S, S)");
    static_assert(std::is_convertible_v<decltype(e), std::function<S()>>,
                  "e must work as S()");

#else

template <class S, S (*op)(S, S), S (*e)()> struct segtree {

#endif

  public:
    segtree() : segtree(0) {}
    explicit segtree(int n) : segtree(std::vector<S>(n, e())) {}
    explicit segtree(const std::vector<S>& v) : _n(int(v.size())) {
        size = (int)internal::bit_ceil((unsigned int)(_n));
        log = internal::countr_zero((unsigned int)size);
        d = std::vector<S>(2 * size, e());
        for (int i = 0; i < _n; i++) d[size + i] = v[i];
        for (int i = size - 1; i >= 1; i--) {
            update(i);
        }
    }

    void set(int p, S x) {
        assert(0 <= p && p < _n);
        p += size;
        d[p] = x;
        for (int i = 1; i <= log; i++) update(p >> i);
    }

    S get(int p) const {
        assert(0 <= p && p < _n);
        return d[p + size];
    }

    S prod(int l, int r) const {
        assert(0 <= l && l <= r && r <= _n);
        S sml = e(), smr = e();
        l += size;
        r += size;

        while (l < r) {
            if (l & 1) sml = op(sml, d[l++]);
            if (r & 1) smr = op(d[--r], smr);
            l >>= 1;
            r >>= 1;
        }
        return op(sml, smr);
    }

    S all_prod() const { return d[1]; }

    template <bool (*f)(S)> int max_right(int l) const {
        return max_right(l, [](S x) { return f(x); });
    }
    template <class F> int max_right(int l, F f) const {
        assert(0 <= l && l <= _n);
        assert(f(e()));
        if (l == _n) return _n;
        l += size;
        S sm = e();
        do {
            while (l % 2 == 0) l >>= 1;
            if (!f(op(sm, d[l]))) {
                while (l < size) {
                    l = (2 * l);
                    if (f(op(sm, d[l]))) {
                        sm = op(sm, d[l]);
                        l++;
                    }
                }
                return l - size;
            }
            sm = op(sm, d[l]);
            l++;
        } while ((l & -l) != l);
        return _n;
    }

    template <bool (*f)(S)> int min_left(int r) const {
        return min_left(r, [](S x) { return f(x); });
    }
    template <class F> int min_left(int r, F f) const {
        assert(0 <= r && r <= _n);
        assert(f(e()));
        if (r == 0) return 0;
        r += size;
        S sm = e();
        do {
            r--;
            while (r > 1 && (r % 2)) r >>= 1;
            if (!f(op(d[r], sm))) {
                while (r < size) {
                    r = (2 * r + 1);
                    if (f(op(d[r], sm))) {
                        sm = op(d[r], sm);
                        r--;
                    }
                }
                return r + 1 - size;
            }
            sm = op(d[r], sm);
        } while ((r & -r) != r);
        return 0;
    }

  private:
    int _n, size, log;
    std::vector<S> d;

    void update(int k) { d[k] = op(d[2 * k], d[2 * k + 1]); }
};

}  // namespace atcoder
```

string.hpp

```
#include <algorithm>
#include <cassert>
#include <numeric>
#include <string>
#include <vector>

namespace atcoder {

namespace internal {

std::vector<int> sa_naive(const std::vector<int>& s) {
    int n = int(s.size());
    std::vector<int> sa(n);
    std::iota(sa.begin(), sa.end(), 0);
    std::sort(sa.begin(), sa.end(), [&](int l, int r) {
        if (l == r) return false;
        while (l < n && r < n) {
            if (s[l] != s[r]) return s[l] < s[r];
            l++;
            r++;
        }
        return l == n;
    });
    return sa;
}

std::vector<int> sa_doubling(const std::vector<int>& s) {
    int n = int(s.size());
    std::vector<int> sa(n), rnk = s, tmp(n);
```

```
    std::iota(sa.begin(), sa.end(), 0);
    for (int k = 1; k < n; k *= 2) {
        auto cmp = [&](int x, int y) {
            if (rnk[x] != rnk[y]) return rnk[x] < rnk[y];
            int rx = x + k < n ? rnk[x + k] : -1;
            int ry = y + k < n ? rnk[y + k] : -1;
            return rx < ry;
        };
        std::sort(sa.begin(), sa.end(), cmp);
        tmp[sa[0]] = 0;
        for (int i = 1; i < n; i++) {
            tmp[sa[i]] = tmp[sa[i - 1]] + (cmp(sa[i - 1], sa[i]) ? 1 : 0);
        }
        std::swap(tmp, rnk);
    }
    return sa;
}

// SA-IS, linear-time suffix array construction
// Reference:
// G. Nong, S. Zhang, and W. H. Chan,
// Two Efficient Algorithms for Linear Time Suffix Array Construction
template <int THRESHOLD_NAIVE = 10, int THRESHOLD_DOUBLING = 40>
std::vector<int> sa_is(const std::vector<int>& s, int upper) {
    int n = int(s.size());
    if (n == 0) return {};
    if (n == 1) return {0};
    if (n == 2) {
        if (s[0] < s[1]) {
            return {0, 1};
        } else {
            return {1, 0};
        }
    }
    if (n < THRESHOLD_NAIVE) {
        return sa_naive(s);
    }
    if (n < THRESHOLD_DOUBLING) {
        return sa_doubling(s);
    }

    std::vector<int> sa(n);
    std::vector<bool> ls(n);
    for (int i = n - 2; i >= 0; i--) {
        ls[i] = (s[i] == s[i + 1]) ? ls[i + 1] : (s[i] < s[i + 1]);
    }
    std::vector<int> sum_l(upper + 1), sum_s(upper + 1);
    for (int i = 0; i < n; i++) {
        if (!ls[i]) {
            sum_s[s[i]]++;
        } else {
            sum_l[s[i] + 1]++;
        }
    }
    for (int i = 0; i <= upper; i++) {
        sum_s[i] += sum_l[i];
        if (i < upper) sum_l[i + 1] += sum_s[i];
    }

    auto induce = [&](const std::vector<int>& lms) {
        std::fill(sa.begin(), sa.end(), -1);
        std::vector<int> buf(upper + 1);
        std::copy(sum_s.begin(), sum_s.end(), buf.begin());
        for (auto d : lms) {
            if (d == n) continue;
            sa[buf[s[d]]++] = d;
        }
        std::copy(sum_l.begin(), sum_l.end(), buf.begin());
        sa[buf[s[n - 1]]++] = n - 1;
        for (int i = 0; i < n; i++) {
            int v = sa[i];
            if (v >= 1 && !ls[v - 1]) {
                sa[buf[s[v - 1]]++] = v - 1;
            }
        }
        std::copy(sum_l.begin(), sum_l.end(), buf.begin());
        for (int i = n - 1; i >= 0; i--) {
            int v = sa[i];
            if (v >= 1 && ls[v - 1]) {
                sa[--buf[s[v - 1] + 1]] = v - 1;
            }
        }
    };

    std::vector<int> lms_map(n + 1, -1);
    int m = 0;
    for (int i = 1; i < n; i++) {
        if (!ls[i - 1] && ls[i]) {
            lms_map[i] = m++;
        }
    }
    std::vector<int> lms;
    lms.reserve(m);
    for (int i = 1; i < n; i++) {
        if (!ls[i - 1] && ls[i]) {
            lms.push_back(i);
        }
    }

    induce(lms);

    if (m) {
        std::vector<int> sorted_lms;
        sorted_lms.reserve(m);
        for (int v : sa) {
            if (lms_map[v] != -1) sorted_lms.push_back(v);
        }
        std::vector<int> rec_s(m);
        int rec_upper = 0;
        rec_s[lms_map[sorted_lms[0]]] = 0;
        for (int i = 1; i < m; i++) {
            int l = sorted_lms[i - 1], r = sorted_lms[i];
            int end_l = (lms_map[l] + 1 < m) ? lms[lms_map[l] + 1] : n;
            int end_r = (lms_map[r] + 1 < m) ? lms[lms_map[r] + 1] : n;
            bool same = true;
            if (end_l - l != end_r - r) {
                same = false;
            } else {
                while (l < end_l) {
                    if (s[l] != s[r]) {
                        break;
                    }
                    l++;
                    r++;
                }
                if (l == n || s[l] != s[r]) same = false;
            }
            if (!same) rec_upper++;
            rec_s[lms_map[sorted_lms[i]]] = rec_upper;
        }

        auto rec_sa =
            sa_is<THRESHOLD_NAIVE, THRESHOLD_DOUBLING>(rec_s, rec_upper);

        for (int i = 0; i < m; i++) {
            sorted_lms[i] = lms[rec_sa[i]];
        }
        induce(sorted_lms);
    }
    return sa;
}

}  // namespace internal

std::vector<int> suffix_array(const std::vector<int>& s, int upper) {
    assert(0 <= upper);
    for (int d : s) {
        assert(0 <= d && d <= upper);
    }
    auto sa = internal::sa_is(s, upper);
    return sa;
}

template <class T> std::vector<int> suffix_array(const std::vector<T>& s) {
    int n = int(s.size());
    std::vector<int> idx(n);
    iota(idx.begin(), idx.end(), 0);
    sort(idx.begin(), idx.end(), [&](int l, int r) { return s[l] <
s[r]; });
    std::vector<int> s2(n);
    int now = 0;
    for (int i = 0; i < n; i++) {
        if (i && s[idx[i - 1]] != s[idx[i]]) now++;
        s2[idx[i]] = now;
    }
    return internal::sa_is(s2, now);
}

std::vector<int> suffix_array(const std::string& s) {
    int n = int(s.size());
    std::vector<int> s2(n);
    for (int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return internal::sa_is(s2, 255);
}

// Reference:
// T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park,
// Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its
// Applications
template <class T>
std::vector<int> lcp_array(const std::vector<T>& s,
                           const std::vector<int>& sa) {
    assert(s.size() == sa.size());
    int n = int(s.size());
    assert(n >= 1);
    std::vector<int> rnk(n);
    for (int i = 0; i < n; i++) {
        assert(0 <= sa[i] && sa[i] < n);
        rnk[sa[i]] = i;
    }
    std::vector<int> lcp(n - 1);
    int h = 0;
    for (int i = 0; i < n; i++) {
        if (h > 0) h--;
        if (rnk[i] == 0) continue;
        int j = sa[rnk[i] - 1];
        for (; j + h < n && i + h < n; h++) {
            if (s[j + h] != s[i + h]) break;
        }
        lcp[rnk[i] - 1] = h;
    }
    return lcp;
}

std::vector<int> lcp_array(const std::string& s, const std::vector<int>&
sa) {
    int n = int(s.size());
    std::vector<int> s2(n);
    for (int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return lcp_array(s2, sa);
}

// Reference:
// D. Gusfield,
// Algorithms on Strings, Trees, and Sequences: Computer Science and
// Computational Biology
template <class T> std::vector<int> z_algorithm(const std::vector<T>& s) {
    int n = int(s.size());
    if (n == 0) return {};
    std::vector<int> z(n);
    z[0] = 0;
    for (int i = 1, j = 0; i < n; i++) {
        int& k = z[i];
        k = (j + z[j] <= i) ? 0 : std::min(j + z[j] - i, z[i - j]);
        while (i + k < n && s[k] == s[i + k]) k++;
        if (j + z[j] < i + z[i]) j = i;
    }
    z[0] = n;
    return z;
}
```

```
std::vector<int> z_algorithm(const std::string& s) {
    int n = int(s.size());
    std::vector<int> s2(n);
    for (int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return z_algorithm(s2);
}

}  // namespace atcoder
```

twosat.hpp

```
#include <cassert>
#include <vector>

#include "atcoder/internal_scc"

namespace atcoder {

// Reference:
// B. Aspvall, M. Plass, and R. Tarjan,
// A Linear-Time Algorithm for Testing the Truth of Certain Quantified
Boolean
// Formulas
struct two_sat {
  public:
    two_sat() : _n(0), scc(0) {}
    explicit two_sat(int n) : _n(n), _answer(n), scc(2 * n) {}

    void add_clause(int i, bool f, int j, bool g) {
        assert(0 <= i && i < _n);
        assert(0 <= j && j < _n);
        scc.add_edge(2 * i + (f ? 0 : 1), 2 * j + (g ? 1 : 0));
        scc.add_edge(2 * j + (g ? 0 : 1), 2 * i + (f ? 1 : 0));
    }
    bool satisfiable() {
        auto id = scc.scc_ids().second;
        for (int i = 0; i < _n; i++) {
            if (id[2 * i] == id[2 * i + 1]) return false;
            _answer[i] = id[2 * i] < id[2 * i + 1];
        }
        return true;
    }
    std::vector<bool> answer() { return _answer; }

  private:
    int _n;
    std::vector<bool> _answer;
    internal::scc_graph scc;
};

}  // namespace atcoder
```