1. unindexed

1.1. BitTricks

```
#ifndef _LIB_BIT_TRICKS
#define _LIB_BIT_TRICKS
#include <bits/stdc++.h>

namespace lib {
long long next_power_of_two(long long n) {
  if (n <= 0) return 1;
  return 1LL << (sizeof(long long) * 8 - 1 - __builtin_clzll(n) +
                 ((n & (n - 1LL)) != 0));
}
} // namespace lib

#endif
```

1.2. Combinatorics

```
#ifndef _LIB_COMBINATORICS
#define _LIB_COMBINATORICS
#include <bits/stdc++.h>
#include "BitTricks.cpp"

namespace lib {
using namespace std;
template<typename T>
struct Combinatorics {
    static vector<T> fat;
    static vector<T> inv;
    static vector<T> ifat;

    static T factorial(int i) {
        ensure_fat(next_power_of_two(i));
        return fat[i];
    }

    static T inverse(int i) {
        ensure_inv(next_power_of_two(i));
        return inv[i];
    }

    static T ifactorial(int i) {
        ensure_ifat(next_power_of_two(i));
        return ifat[i];
    }

    static T nCr(int n, int K) {
        if(K > n) return 0;
        ensure_fat(next_power_of_two(n));
        ensure_ifat(next_power_of_two(n));
        return fat[n] * ifat[n-K] * ifat[K];
    }

    static T arrangement(int n, int K) {
        return nCr(n, K) * factorial(n);
    }

    static T nCr_rep(int n, int K) {
        return interpolate(n - 1, K);
    }

    static T interpolate(int a, int b) {
        return nCr(a+b, b);
    }

    static void ensure_fat(int i) {
        int o = fat.size();
        if(i < o) return;
        fat.resize(i+1);
        for(int j = o; j <= i; j++) fat[j] = fat[j-1]*j;
    }

    static void ensure_inv(int i) {
        int o = inv.size();
        if(i < o) return;
        inv.resize(i+1);
        for(int j = o; j <= i; j++) inv[j] = -(inv[T::mod%j] * (T::mod/j));
    }

    static void ensure_ifat(int i) {
        int o = ifat.size();
        if(i < o) return;
        ifat.resize(i+1);
        ensure_inv(i);
        for(int j = o; j <= i; j++) ifat[j] = ifat[j-1]*inv[j];
    }
};

template<typename T>
vector<T> Combinatorics<T>::fat = vector<T>(1, T(1));
template<typename T>
vector<T> Combinatorics<T>::inv = vector<T>(2, T(1));
template<typename T>
vector<T> Combinatorics<T>::ifat = vector<T>(1, T(1));
} // namespace lib

#endif
```

1.3. Complex

```
#ifndef _LIB_COMPLEX
#define _LIB_COMPLEX
#include <bits/stdc++.h>

namespace lib {
using namespace std;
template <typename T> struct Complex {
  T re, im;
  Complex(T a = T(), T b = T()) : re(a), im(b) {}
  T real() const { return re; }
  T imag() const { return im; }
  explicit operator T() const { return re; }
  template<typename G>
  operator Complex<G>() const { return Complex<G>(re, im); }
  Complex conj() const { return Complex(re, -im); }
  void operator+=(const Complex<T> &rhs) { re += rhs.re, im += rhs.im; }
  void operator-=(const Complex<T> &rhs) { re -= rhs.re, im -= rhs.im; }
  void operator*=(const Complex<T> &rhs) {
    tie(re, im) =
        make_pair(re * rhs.re - im * rhs.im, re * rhs.im + im * rhs.re);
  }
  Complex<T> operator+(const Complex<T> &rhs) {
    Complex<T> res = *this;
    res += rhs;
    return res;
  }
```

```
  Complex<T> operator-(const Complex<T> &rhs) {
    Complex<T> res = *this;
    res -= rhs;
    return res;
  }
  Complex<T> operator*(const Complex<T> &rhs) {
    Complex<T> res = *this;
    res *= rhs;
    return res;
  }
  Complex<T> operator-() const {
    return {-re, -im};
  }
  void operator/=(const T x) { re /= x, im /= x; }
};
} // namespace lib

#endif
```

1.4. DFT

```
#ifndef _LIB_DFT
#define _LIB_DFT
#include <bits/stdc++.h>
#include "BitTricks.cpp"

namespace lib {
using namespace std;
namespace linalg {
template <typename Ring, typename Provider>
struct DFT {
  static vector<int> rev;
  static vector<Ring> fa;

  // function used to precompute rev for fixed size fft (n is a power of two)
  static void dft_rev(int n) {
    Provider()(n);
    int lbn = __builtin_ctz(n);
    if ((int)rev.size() < (1 << lbn))
      rev.resize(1 << lbn);
    int h = -1;
    for (int i = 1; i < n; i++) {
      if ((i & (i - 1)) == 0)
        h++;
      rev[i] = rev[i ^ (1 << h)] | (1 << (lbn - h - 1));
    }
  }

  static void dft_iter(Ring *p, int n) {
    Provider w;
    for (int L = 2; L <= n; L <<= 1) {
      for (int i = 0; i < n; i += L) {
        for (int j = 0; j < L / 2; j++) {
          Ring z = p[i + j + L / 2] * w[j + L / 2];
          p[i + j + L / 2] = p[i + j] - z;
          p[i + j] += z;
        }
      }
    }
  }

  static void swap(vector<Ring> &buf) { std::swap(fa, buf); }
  static void _dft(Ring *p, int n) {
    dft_rev(n);
    for (int i = 0; i < n; i++)
      if (i < rev[i])
        std::swap(p[i], p[rev[i]]);
    dft_iter(p, n);
  }
  static void _idft(Ring *p, int n) {
    _dft(p, n);
    reverse(p + 1, p + n);
    Ring inv = Provider().inverse(n);
    for (int i = 0; i < n; i++)
      p[i] *= inv;
  }

  static void dft(int n) { _dft(fa.data(), n); }

  static void idft(int n) { _idft(fa.data(), n); }

  static void dft(vector<Ring> &v, int n) {
    swap(v);
    dft(n);
    swap(v);
  }
  static void idft(vector<Ring> &v, int n) {
    swap(v);
    idft(n);
    swap(v);
  }

  static int ensure(int a, int b = 0) {
    int n = a+b;
    n = next_power_of_two(n);
    if ((int)fa.size() < n)
      fa.resize(n);
    return n;
  }

  static void clear(int n) { fill(fa.begin(), fa.begin() + n, 0); }

  template<typename Iterator>
  static void fill(Iterator begin, Iterator end) {
    int n = ensure(distance(begin, end));
    int i = 0;
    for(auto it = begin; it != end; ++it) {
      fa[i++] = *it;
    }
    for(;i < n; i++) fa[i] = Ring();
  }
};

template<typename DF, typename U>
static vector<U> retrieve(int n) {
  assert(n <= DF::fa.size());
  vector<U> res(n);
  for(int i = 0; i < n; i++) res[i] = (U)DF::fa[i];
  return res;
}

template<typename Ring, typename Provider>
vector<int> DFT<Ring, Provider>::rev = vector<int>();

template<typename Ring, typename Provider>
vector<Ring> DFT<Ring, Provider>::fa = vector<Ring>();
}
} // namespace lib

```

```cpp
#endif
```

1.5. DSU

```cpp
#ifndef _LIB_DSU
#define _LIB_DSU
#include <bits/stdc++.h>

namespace lib {
using namespace std;

struct DSU {
  vector<int> p, ptime, sz;
  int tempo = 0;
  int merges = 0;
  pair<int, int> last_merge_ = {-1, -1};

  DSU(int n = 0) : p(n), ptime(n, 1e9), sz(n, 1) { iota(p.begin(), p.end(), 0); }

  int make_node() {
    int i = p.size();
    p.emplace_back(i);
    ptime.emplace_back(0);
    sz.emplace_back(1);
    return 1;
  }

  int get(int i, int at) const {
    return p[i] == i ? i : (at >= ptime[i] ? get(p[i], at) : i);
  }

  int operator[](int i) const { return get(i, tempo); }

  int merge(int u, int v) {
    u = (*this)[u], v = (*this)[v];
    if (u == v)
      return 0;
    if (sz[u] < sz[v])
      swap(u, v);
    p[v] = u;
    ptime[v] = ++tempo;
    sz[u] += sz[v];
    last_merge_ = {v, u};
    merges++;
    return 1;
  }
  pair<int, int> last_merge() const {
    return last_merge_;
  }

  int n_comps() const { return (int)p.size() - merges; }
};

struct CompressedDSU {
  vector<int> p;
  CompressedDSU(int n = 0) : p(n) { iota(p.begin(), p.end(), 0); }
  int get(int i) {
    return p[i] == i ? i : p[i] = get(p[i]);
  }
  int operator[](int i) { return get(i); }
  int& parent(int i) { return p[i]; }
};

struct FastDSU {
  vector<int> p, sz;
  int merges = 0;
  pair<int, int> last_merge_ = {-1, -1};
  FastDSU(int n = 0) : p(n), sz(n, 1) { iota(p.begin(), p.end(), 0); }

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

  int merge(int u, int v) {
    u = get(u), v = get(v);
    if(u == v) return 0;
    if(sz[u] < sz[v])
      swap(u, v);
    p[v] = u;
    sz[u] += sz[v];
    merges++;
    last_merge_ = {v, u};
    return 1;
  }
  pair<int, int> last_merge() const {
    return last_merge_;
  }
  int n_comps() const { return (int)p.size() - merges; }
};
} // namespace lib

#endif
```

1.6. Epsilon

```cpp
#ifndef _LIB_EPSILON
#define _LIB_EPSILON
#include <bits/stdc++.h>

namespace lib {
using namespace std;

template <typename T = double> struct Epsilon {
  T eps;
  constexpr Epsilon(T eps = 1e-9) : eps(eps) {}

  template <typename G,
            typename enable_if<is_floating_point<G>::value>::type * = nullptr>
  int operator()(G a, G b = 0) const {
    return a + eps < b ? -1 : (b + eps < a ? 1 : 0);
  }

  template <typename G,
            typename enable_if<!is_floating_point<G>::value>::type * = nullptr>
  int operator()(G a, G b = 0) const {
    return a < b ? -1 : (a > b ? 1 : 0);
  }

  template <typename G,
            typename enable_if<is_floating_point<G>::value>::type * = nullptr>
  bool null(G a) const {
    return (*this)(a) == 0;
  }
```

```

  template <typename G,
            typename enable_if<!is_floating_point<G>::value>::type * =
    nullptr>
  bool null(G a) const {
    return a == 0;
  }
};
} // namespace lib

#endif
```

1.7. Euclid

```
#ifndef _LIB_EUCLID
#define _LIB_EUCLID
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace math {
namespace {
constexpr static size_t ULL_SIZE = sizeof(unsigned long long);

template <typename T>
using IsManageableInt =
    typename conditional<is_integral<T>::value && sizeof(T) <= ULL_SIZE,
                         true_type, false_type>::type;
} // namespace

template <typename T, typename US = T> struct Euclid {
  template <typename U, typename V> static V safe_mod(U x, V m) {
    x %= m;
    if (x < 0)
      x += m;
    return x;
  }

  template <typename U,
            typename enable_if<IsManageableInt<U>::value>::type * = nullptr>
  static U safe_mult(U a, U b, U m) {
    a = safe_mod(a, m), b = safe_mod(b, m);

    if (!b) return 0;
    int hi = 63 - __builtin_clzll((unsigned long long)b);
    U res = 0;
    for (int i = hi; i >= 0; i--) {
      res = safe_mod(res * 2, m);
      if ((b >> i) & 1)
        res = safe_mod(res + a, m);
    }
    return res;
  }

  template <typename U,
            typename enable_if<!IsManageableInt<U>::value>::type * = nullptr>
  static U safe_mult(U a, U b, U m) {
    return a * b % m;
  }

  static T euclid_(T a, T b, T &x, T &y) {
    if (a == 0) {
      x = 0, y = 1;
      return b;
    }
    T x1, y1;
    T g = euclid_(b % a, a, x1, y1);
    x = y1 - b / a * x1;
    y = x1;
    return g;
  }

  static T euclid(T a, T b, T &x, T &y) {
    T g = euclid_(a, b, x, y);
    if (g < 0)
      g = -g, x = -x, y = -y;
    return g;
  }

  static pair<T, T> crt(T a, T b, T m1, T m2) {
    if (m1 < m2)
      swap(m1, m2), swap(a, b);
    T xx, yy;
    T g = euclid(m1, m2, xx, yy);
    if (safe_mod(a, g) != safe_mod(b, g))
      return {0, 0};

    T mod = m1 / g * m2;

    T x = safe_mod<T>(xx, mod);
    US s = safe_mult<T>(x, (b - a) / g, m2 / g) * m1 % mod;
    T res = safe_mod<US, US>((US)a + s, mod);

    return {safe_mod<T>(res, mod), mod};
  }

  static pair<T, T> crt(const vector<pair<T, T>> &equations) {
    pair<T, T> acc = {0, 1};
    for (const pair<T, T> &e : equations) {
      acc = crt(acc.first, e.first, acc.second, e.second);
      if (!acc.second)
        return {0, 0};
    }
    return acc;
  }

  static bool diophantine_solution(T a, T b, T c, T& x0, T& y0, T& g) {
    g = euclid(a, b, x0, y0);
    if (c % g)
      return false;
    x0 *= c/g;
    y0 *= c/g;
    return true;
  }

  // Give solutions for diophantine in the form [x = x.first * k + x.second].
  static bool diophantine_solutions(T a, T b, T c, pair<T, T>& x, pair<T,
    T>& y) {
    T g;
    if(!diophantine_solution(a, b, c, x.second, y.second, g))
      return false;
    x.first = b / g;
    y.first = -a / g;
    return true;
  }

  // Give parameterized solution (in terms of k) to:
  // a_1 * k + b_1 = ... = a_n * k + b_n, i.e, an equation for where those
  // functions meet.
```

```cpp
  static bool linear_equality_system(const vector<pair<T, T>>& v, pair<T,
      T>& res) {
    assert(!v.empty());
    res = v[0];
    for(int i = 1; i < v.size(); i++) {
      pair<T, T> x, y;
      if (!diophantine_solutions(res.first, -v[i].first, v[i].second -
    res.second, x, y))
        return false;
      auto num = res.first * x.first;
      if (num < 0) num = -num;
      res = {
        num,
        safe_mod(res.second + safe_mult(res.first, x.second, num), num),
      };
    }
    return true;
  }
};

using LongCRT = Euclid<long long, unsigned long long>;
} // namespace math
} // namespace lib

#endif
```

1.8. FFT

```cpp
#ifndef _LIB_FFT
#define _LIB_FFT
#include "DFT.cpp"
#include "Complex.cpp"
#include "geometry/Trigonometry.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace linalg {

template<typename T>
struct ComplexRootProvider {
  typedef Complex<T> cd;
  typedef Complex<long double> cld;
  static vector<cd> w;
  static vector<cld> wl;

  static cld root(long double ang) {
    return cld(geo::trig::cos(ang), geo::trig::sin(ang));
  }

  cd operator()(int n, int k) {
    long double ang = 2.0l * geo::trig::PI / (n / k);
    return root(ang);
  }
  void operator()(int n) {
    n = max(n, 2);
    int k = max((int)w.size(), 2);
    if ((int)w.size() < n)
      w.resize(n), wl.resize(n);
    else
      return;
    w[0] = w[1] = cd(1.0, 0.0);
    wl[0] = wl[1] = cld(1.0, 0.0);
    for (; k < n; k *= 2) {
      long double ang = 2.0l * geo::trig::PI / (2*k);
      cld step = root(ang);
      for(int i = k; i < 2*k; i++)
        w[i] = wl[i] = (i&1) ? wl[i/2] * step : wl[i/2];
    }
  }
  cd operator[](int i) {
    return w[i];
  }
  cd inverse(int n) {
    return cd(1.0 / n, 0.0);
  }
};

template<typename T>
vector<Complex<T>> ComplexRootProvider<T>::w = vector<Complex<T>>();
template<typename T>
vector<Complex<long double>> ComplexRootProvider<T>::wl =
    vector<Complex<long double>>();

template<typename T = double>
struct FFT : public DFT<Complex<T>, ComplexRootProvider<T>> {
  using Parent = DFT<Complex<T>, ComplexRootProvider<T>>;
  using Parent::fa;

  template <typename U>
  static void _convolve(const vector<U> &a, const vector<U> &b) {
    typedef Complex<T> cd;
    int n = Parent::ensure(a.size(), b.size());
    for (size_t i = 0; i < (size_t)n; i++)
      fa[i] = cd(i < a.size() ? (T)a[i] : T(),
                 i < b.size() ? (T)b[i] : T());
    Parent::dft(n);
    for (int i = 0; i < n; i++)
      fa[i] *= fa[i];
    Parent::idft(n);
    for (int i = 0; i < n; i++)
      fa[i] = cd(fa[i].imag() / 2, T());
  }

  template<typename U>
  static vector<U> convolve(const vector<U>& a, const vector<U>& b) {
    int sz = (int)a.size() + b.size() - 1;
    _convolve(a, b);
    return retrieve<Parent, U>(sz);
  }

  template<typename U>
  static vector<U> convolve_rounded(const vector<U>& a, const vector<U>& b) {
    int sz = (int)a.size() + b.size() - 1;
    _convolve(a, b);
    vector<U> res(sz);
    for(int i = 0; i < sz; i++) res[i] = (U)(long long)(fa[i].real() + 0.5);
    return res;
  }

  // TODO: use separate static buffers for this function
  template <typename M>
  static vector<M> convolve_mod(const vector<M> &a, const vector<M> &b) {
    typedef typename M::type_int type_int;
    typedef typename M::large_int large_int;
    typedef Complex<T> cd;
    typedef vector<cd> vcd;

    static_assert(sizeof(M::mods) / sizeof(type_int) == 1,
```

```
                    "cant multiply with multiple mods");
    type_int base = sqrtl(M::mods[0]) + 0.5;
    M base_m = base;
    int sza = a.size();
    int szb = b.size();
    int sz = sza+szb-1;
    int n = next_power_of_two(sz);
    Parent::dft_rev(n);

    // establish buffers
    vcd fa(n), fb(n), C1(n), C2(n);

    for (int i = 0; i < n; i++)
      fa[i] = i < sza ? cd((type_int)a[i] / base, (type_int)a[i] % base) : cd();
    for (int i = 0; i < n; i++)
      fb[i] = i < szb ? cd((type_int)b[i] / base, (type_int)b[i] % base) : cd();
    Parent::dft(fa, n);
    Parent::dft(fb, n);

    for (int i = 0; i < n; i++) {
      int j = i ? n - i : 0;
      cd a1 = (fa[i] + fa[j].conj()) * cd(0.5, 0.0);
      cd a2 = (fa[i] - fa[j].conj()) * cd(0.0, -0.5);
      cd b1 = (fb[i] + fb[j].conj()) * cd(0.5, 0.0);
      cd b2 = (fb[i] - fb[j].conj()) * cd(0.0, -0.5);
      cd c11 = a1 * b1, c12 = a1 * b2;
      cd c21 = a2 * b1, c22 = a2 * b2;
      C1[j] = c11 + c12 * cd(0.0, 1.0);
      C2[j] = c21 + c22 * cd(0.0, 1.0);
    }
    Parent::idft(C1, n), Parent::idft(C2, n);

    vector<M> res(sz);
    for (int i = 0; i < sz; i++) {
      int j = i ? n - i : 0;
      M x = large_int(C1[j].real() + 0.5);
      M y1 = large_int(C1[j].imag() + 0.5);
      M y2 = large_int(C2[j].real() + 0.5);
      M z = large_int(C2[j].imag() + 0.5);
      res[i] = x * base_m * base_m + (y1 + y2) * base_m + z;
    }

    return res;
  }
};
} // namespace linalg

namespace math {
struct FastMultiplication {
  template<typename T>
  using Transform = linalg::FFT<T>;
  template <typename Field, typename U = double>
  vector<Field> operator()(const vector<Field> &a,
                           const vector<Field> &b) const {
    return linalg::FFT<U>::convolve_rounded(a, b);
  }
};

struct FFTMultiplication {
  template<typename T>
  using Transform = linalg::FFT<T>;
  template <typename Field, typename U = double>
  vector<Field> operator()(const vector<Field> &a,
                           const vector<Field> &b) const {
    return linalg::FFT<U>::convolve(a, b);
  }
};

struct SafeMultiplication {
  template<typename T>
  using Transform = linalg::FFT<T>;
  template <typename Field, typename U = double>
  vector<Field> operator()(const vector<Field> &a,
                           const vector<Field> &b) const {
    return linalg::FFT<U>::convolve_mod(a, b);
  };
};
} // namespace math
} // namespace lib

#endif
```

1.9. FHT

```
#ifndef _LIB_FHT
#define _LIB_FHT
#include <bits/stdc++.h>
#include "BitTricks.cpp"
#include "NTT.cpp"
#include "polynomial/Transform.cpp"

namespace lib {
using namespace std;
namespace linalg {
template <typename Ring>
struct FHT {
  using Provider = MintRootProvider<Ring>;
  using T = Ring;
  using U = make_unsigned_t<typename Ring::type_int>;
  using U64 = make_unsigned_t<typename Ring::large_int>;
  static vector<Ring> fa;
  static const int MAX_LG_N = 30;
  static vector<Ring> g, ig;

  static void precompute() {
    if(!g.empty()) return;
    Provider();
    g.resize(MAX_LG_N);
    ig.resize(MAX_LG_N);
    for(int i = 0; i < MAX_LG_N; i++) {
      Ring w = Provider::g ^ (((Ring::mod-1) >> (i + 2)) * 3);
      w = -w;
      Ring iw = w.inverse();
      g[i] = w;
      ig[i] = iw;
    }
  }

  static inline U& v(Ring& p) {
    return (U&)p.data();
  }

  static inline U v(const Ring& p) {
    return (U)p.data();
  }

  static void dft_iter(Ring *p, int n) {
```

```cpp
    // decimation-in-time
    // natural to reverse ordering
    for (int B = n >> 1; B; B >>= 1) {
      Ring w = 1;
      for (int i = 0, twiddle = 0; i < n; i += B * 2) {
        for (int j = i; j < i + B; j++) {
          Ring z = p[j + B] * w;
          p[j + B] = p[j] - z;
          p[j] += z;
        }
        w *= g[__builtin_ctz(++twiddle)];
      }
    }
  }

  static void idft_iter(Ring *p, int n) {
    // decimation-in-frequency
    // reverse to natural ordering
    for (int B = 1; B < n; B <<= 1) {
      Ring w = 1;
      for (int i = 0, twiddle = 0; i < n; i += B * 2) {
        for (int j = i; j < i + B; j++) {
          Ring z = (p[j] - p[j + B]) * w;
          p[j] += p[j + B];
          p[j + B] = z;
        }
        w *= ig[__builtin_ctz(++twiddle)];
      }
    }
  }

  static void swap(vector<Ring> &buf) { std::swap(fa, buf); }
  static void _dft(Ring *p, int n) {
    precompute();
    dft_iter(p, n);
  }
  static void _idft(Ring *p, int n) {
    precompute();
    idft_iter(p, n);
    Ring inv = Provider().inverse(n);
    for (int i = 0; i < n; i++)
      p[i] *= inv;
  }

  static void dft(int n) { _dft(fa.data(), n); }

  static void idft(int n) { _idft(fa.data(), n); }

  static void dft(vector<Ring> &v, int n) {
    swap(v);
    dft(n);
    swap(v);
  }
  static void idft(vector<Ring> &v, int n) {
    swap(v);
    idft(n);
    swap(v);
  }

  static int ensure(int a, int b = 0) {
    int n = a+b;
    n = next_power_of_two(n);
    if ((int)fa.size() < n)
      fa.resize(n);
    return n;
  }

  static void clear(int n) { fill(fa.begin(), fa.begin() + n, 0); }

  template<typename Iterator>
  static void fill(Iterator begin, Iterator end) {
    int n = ensure(distance(begin, end));
    int i = 0;
    for(auto it = begin; it != end; ++it) {
      fa[i++] = *it;
    }
    for(;i < n; i++) fa[i] = Ring();
  }

  static void _convolve(const vector<T> &a) {
    int n = ensure(a.size(), a.size());
    for (size_t i = 0; i < (size_t)n; i++)
      fa[i] = i < a.size() ? a[i] : T();
    dft(n);
    for (int i = 0; i < n; i++)
      fa[i] *= fa[i];
    idft(n);
  }

  static void _convolve(const vector<T> &a, const vector<T> &b) {
    if(std::addressof(a) == std::addressof(b))
      return _convolve(a);
    int n = ensure(a.size(), b.size());
    for (size_t i = 0; i < (size_t)n; i++)
      fa[i] = i < a.size() ? a[i] : T();
    dft(n);
    // TODO: have a buffer for this
    auto fb = retrieve<FHT<T>, T>(n);
    for(size_t i = 0; i < (size_t)n; i++)
      fa[i] = i < b.size() ? b[i] : T();
    dft(n);
    for (int i = 0; i < n; i++)
      fa[i] *= fb[i];
    idft(n);
  }

  static vector<T> convolve(const vector<T>& a, const vector<T>& b) {
    int sz = (int)a.size() + b.size() - 1;
    _convolve(a, b);
    return retrieve<FHT<T>, T>(sz);
  }

  static VectorN<T> transform(vector<T> a, int n) {
    a.resize(n);
    dft(a, n);
    return a;
  }

  static vector<T> itransform(vector<T> a, int n) {
    int sz = a.size();
    idft(a, sz);
    a.resize(min(n, sz));
    return a;
  }
};

template<typename Ring>
vector<Ring> FHT<Ring>::fa = vector<Ring>();
template<typename Ring>
vector<Ring> FHT<Ring>::g = vector<Ring>();
```

```cpp
template<typename Ring>
vector<Ring> FHT<Ring>::ig = vector<Ring>();
}

using FHTMultiplication = TransformMultiplication<linalg::FHT>;
} // namespace lib

#endif
```

1.10. FastMap

```cpp
#ifndef _LIB_FAST_MAP
#define _LIB_FAST_MAP
#include <bits/stdc++.h>

// Pretty much copied from:
//
    https://nyaannyaan.github.io/library/data-structure/hash-map-variable-length.hpp
namespace lib {
using namespace std;

template <typename Key, typename Val = Key>
struct FastMap {
  using u32 = uint32_t;
  using u64 = uint64_t;

  u32 cap, s;
  vector<Key> keys;
  vector<Val> vals;
  vector<bool> flag;
  u64 r;
  u32 shift;
  Val DefaultValue;

  static u64 rng() {
    u64 m = chrono::duration_cast<chrono::nanoseconds>(
                chrono::high_resolution_clock::now().time_since_epoch())
                .count();
    m ^= m >> 16;
    m ^= m << 32;
    return m;
  }

  void reallocate() {
    cap <<= 1;
    vector<Key> k(cap);
    vector<Val> v(cap);
    vector<bool> f(cap);
    u32 sh = shift - 1;
    for (int i = 0; i < (int)flag.size(); i++) {
      if (flag[i]) {
        u32 hash = (u64(keys[i]) * r) >> sh;
        while (f[hash]) hash = (hash + 1) & (cap - 1);
        k[hash] = keys[i];
        v[hash] = vals[i];
        f[hash] = 1;
      }
    }
    keys.swap(k);
    vals.swap(v);
    flag.swap(f);
    --shift;
  }

  explicit FastMap()
      : cap(8),
        s(0),
        keys(cap),
        vals(cap),
        flag(cap),
        r(rng()),
        shift(64 - __lg(cap)),
        DefaultValue(Val()) {}

  Val& operator[](const Key& i) {
    u32 hash = (u64(i) * r) >> shift;
    while (true) {
      if (!flag[hash]) {
        if (s + s / 4 >= cap) {
          reallocate();
          return (*this)[i];
        }
        keys[hash] = i;
        flag[hash] = 1;
        ++s;
        return vals[hash] = DefaultValue;
      }
      if (keys[hash] == i) return vals[hash];
      hash = (hash + 1) & (cap - 1);
    }
  }

  // exist -> return pointer of Val
  // not exist -> return nullptr
  const Val* find(const Key& i) const {
    u32 hash = (u64(i) * r) >> shift;
    while (true) {
      if (!flag[hash]) return nullptr;
      if (keys[hash] == i) return &(vals[hash]);
      hash = (hash + 1) & (cap - 1);
    }
  }

  // return vector< pair<const Key&, val& > >
  vector<pair<Key, Val>> enumerate() const {
    vector<pair<Key, Val>> ret;
    for (u32 i = 0; i < cap; ++i)
      if (flag[i]) ret.emplace_back(keys[i], vals[i]);
    return ret;
  }

  int size() const { return s; }

  // set default_value
  void set_default(const Val& val) { DefaultValue = val; }
};
} // namespace lib

#endif
```

1.11. Fenwick

```cpp
#ifndef _LIB_FENWICK
#define _LIB_FENWICK
#include <bits/stdc++.h>

namespace lib {
using namespace std;
```

```
template<typename T>
struct Fenwick {
  vector<int> t;
  Fenwick(int n) : t(n+1) {}
  int size() const { return t.size() - 1; }
  void add(int i, T x) {
    for(i++; i < t.size(); i += (i&-i))
      t[i] += x;
  }
  T get(int i) const {
    T res = 0;
    for(i++; i > 0; i -= (i&-i))
      res += t[i];
    return res;
  }
  T get(int i, int j) const {
    return get(j) - get(i - 1);
  }
  T from(int i) const {
    return get(i, size() - 1);
  }
};
} // namespace lib
#endif
```

1.12. Graph

```
#ifndef _LIB_GRAPH
#define _LIB_GRAPH
#include "Traits.cpp"
#include "utils/Wrappers.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace graph {
template <typename V = void, typename E = void, bool Directed = false>
struct GraphImpl {
  typedef GraphImpl<V, E> self_type;
  typedef vector<vector<int>> adj_list;
  typedef Edge<E> edge_type;
  typedef VertexWrapper<V> vertex_type;

  const static bool directed = Directed;

  vector<edge_type> edges;
  adj_list adj;

  vector<vertex_type> vertices;

  class iterator {
  public:
    typedef iterator self_type;
    typedef edge_type value_type;
    typedef edge_type &reference;
    typedef edge_type *pointer;
    typedef std::forward_iterator_tag iterator_category;
    typedef int difference_type;
    iterator(vector<int> *adj, vector<edge_type> *edges, int ptr = 0)
        : adj_(adj), edges_(edges), ptr_(ptr) {}
    self_type operator++() {
      ptr_++;
      return *this;
    }
    self_type operator++(int junk) {
      self_type i = *this;
      ptr_++;
      return i;
    }
    reference operator*() { return (*edges_)[(*adj_)[ptr_]]; }
    pointer operator->() { return &(*edges_)[(*adj_)[ptr_]]; }
    bool operator==(const self_type &rhs) const {
      return adj_ == rhs.adj_ && ptr_ == rhs.ptr_;
    }
    bool operator!=(const self_type &rhs) const { return !(*this == rhs); }

  private:
    vector<int> *adj_;
    vector<edge_type> *edges_;
    int ptr_;
  };

  class const_iterator {
  public:
    typedef const_iterator self_type;
    typedef edge_type value_type;
    typedef edge_type &reference;
    typedef edge_type *pointer;
    typedef std::forward_iterator_tag iterator_category;
    typedef int difference_type;
    const_iterator(vector<int> *adj, vector<edge_type> *edges, int ptr = 0)
        : adj_(adj), edges_(edges), ptr_(ptr) {}
    self_type operator++() {
      ptr_++;
      return *this;
    }
    self_type operator++(int junk) {
      self_type i = *this;
      ptr_++;
      return i;
    }
    const value_type &operator*() { return (*edges_)[(*adj_)[ptr_]]; }
    const value_type *operator->() { return &(*edges_)[(*adj_)[ptr_]]; }
    bool operator==(const self_type &rhs) const {
      return adj_ == rhs.adj_ && ptr_ == rhs.ptr_;
    }
    bool operator!=(const self_type &rhs) const { return !(*this == rhs); }

  private:
    vector<int> *adj_;
    vector<edge_type> *edges_;
    int ptr_;
  };

  struct iterable {
    vector<int> *adj_;
    vector<edge_type> *edges_;

    iterable(vector<int> *adj, vector<edge_type> *edges)
        : adj_(adj), edges_(edges) {}

    inline iterator begin() { return iterator(adj_, edges_); }
    inline iterator end() { return iterator(adj_, edges_, adj_->size()); }

    inline const_iterator cbegin() const {
      return const_iterator(adj_, edges_);
    }
    inline const_iterator cend() const {
      return const_iterator(adj_, edges_, adj_->size());
```

```
    }

    inline const_iterator begin() const { return cbegin(); }
    inline const_iterator end() const { return cend(); }

    inline edge_type &operator[](int i) { return (*edges_)[(*adj_)[i]]; }
    inline const edge_type &operator[](int i) const {
      return (*edges_)[(*adj_)[i]];
    }

    inline int index(int i) const { return (*adj_)[i]; }
    inline int size() const { return adj_->size(); }
  };

  GraphImpl() {}

  template <typename S = V,
            typename enable_if<is_void<S>::value>::type * = nullptr>
  GraphImpl(size_t n) : adj(n) {}

  template <typename S = V,
            typename enable_if<!is_void<S>::value>::type * = nullptr>
  GraphImpl(size_t n) : adj(n), vertices(n) {}

  inline iterable n_edges(int i) { return iterable(&adj[i], &edges); }
  inline const iterable n_edges(int i) const {
    return iterable(const_cast<vector<int> *>(&adj[i]),
                    const_cast<vector<edge_type> *>(&edges));
  }
  inline int degree(int i) const { return adj[i].size(); }

  inline int size() const { return adj.size(); }
  inline int edge_size() const { return edges.size(); }
  inline edge_type &edge(int i) { return edges[i]; }
  inline edge_type edge(int i) const { return edges[i]; }

  inline vector<edge_type> all_edges() const { return edges; }

  template <typename S = V,
            typename enable_if<!is_void<S>::value>::type * = nullptr>
  inline S &vertex(int i) {
    return vertices[i];
  }

  template <typename S = V,
            typename enable_if<!is_void<S>::value>::type * = nullptr>
  inline V vertex(int i) const {
    return vertices[i];
  }

  template <typename S = V,
            typename enable_if<is_void<S>::value>::type * = nullptr>
  inline void add_vertex() {
    adj.emplace_back();
  }

  template <typename S = V,
            typename enable_if<!is_void<S>::value>::type * = nullptr>
  inline S &add_vertex() {
    adj.emplace_back();
    return vertices.emplace_back().data;
  }

  template <typename S = E,
            typename enable_if<is_void<S>::value>::type * = nullptr>
  inline void add_edge_(int u, int v) {
    adj[u].push_back(edges.size());
    edges.push_back({u, v});
  }

  template <typename S = E,
            typename enable_if<!is_void<S>::value>::type * = nullptr>
  inline S &add_edge_(int u, int v) {
    adj[u].push_back(edges.size());
    edges.push_back({u, v});
    return edges.back().data;
  }

  void add_2edge(int u, int v) {
    add_edge_(u, v);
    add_edge_(v, u);
  }

  template <typename S = E,
            typename enable_if<!is_void<S>::value>::type * = nullptr>
  inline void add_2edge(int u, int v, const S &data) {
    add_edge_(u, v) = data;
    add_edge_(v, u) = data;
  }

  template <typename S = E,
            typename enable_if<is_void<S>::value && Directed>::type * =
    nullptr>
  inline void add_edge(int u, int v) {
    adj[u].push_back(edges.size());
    edges.push_back({u, v});
  }

  template <typename S = E,
            typename enable_if<!is_void<S>::value && Directed>::type * =
    nullptr>
  inline S &add_edge(int u, int v) {
    adj[u].push_back(edges.size());
    edges.push_back({u, v});
    return edges.back().data;
  }
};

template<typename V = void, typename E = void>
using Graph = GraphImpl<V, E, false>;

template<typename V = void, typename E = void>
using DirectedGraph = GraphImpl<V, E, true>;

template <typename V = void, typename E = void>
struct RootedForest : public DirectedGraph<V, E> {
  typedef RootedForest<V, E> self_type;
  using typename DirectedGraph<V, E>::adj_list;
  using typename DirectedGraph<V, E>::edge_type;
  using DirectedGraph<V, E>::DirectedGraph;
  using DirectedGraph<V, E>::adj;
  using DirectedGraph<V, E>::edge;
  vector<int> p, pe;

  void build_parents() {
    if ((int)p.size() == this->size())
      return;

    int n = this->size();
    stack<int> st;
```

```
    vector<bool> vis(n);
    p.assign(n, -1), pe.assign(n, -1);
    for (int i = 0; i < n; i++) {
      if (!vis[i]) {
        st.push(i);
        vis[i] = true;
        while (!st.empty()) {
          int u = st.top();
          st.pop();

          for (int k : adj[u]) {
            int v = edge(k).to;
            vis[v] = true;
            st.push(v), pe[v] = k, p[v] = u;
          }
        }
      }
    }
  }

  inline int parent(int i) const {
    const_cast<self_type *>(this)->build_parents();
    return p[i];
  }

  inline bool is_root(int i) const { return parent(i) != -1; }

  inline edge_type &parent_edge(int i) {
    build_parents();
    return edge(pe[i]);
  }
  inline edge_type &parent_edge(int i) const {
    const_cast<self_type *>(this)->build_parents();
    return edge(pe[i]);
  }

  vector<int> roots() const {
    vector<int> res;
    const_cast<self_type *>(this)->build_parents();
    int n = this->size();

    for (int i = 0; i < n; i++)
      if (p[i] == -1)
        res.push_back(i);
    return res;
  }
};

template <typename V = void, typename E = void>
struct RootedTree : public RootedForest<V, E> {
  using typename RootedForest<V, E>::adj_list;
  int root;

  RootedTree(int n, int root) : RootedForest<V, E>(n) {
    assert(n > 0);
    assert(root < n);
    this->root = root;
  }

  RootedTree(const adj_list &adj, int root) : RootedForest<V, E>(adj) {
    assert(adj.size() > 0);
    assert(root < adj.size());
    this->root = root;
  }
};

namespace builders {
namespace {
template <typename F, typename G>
void dfs_rooted_forest(F &forest, const G &graph, int u, vector<bool> &vis) {
  vis[u] = true;
  for (const auto &ed : graph.n_edges(u)) {
    int v = ed.to;
    if (!vis[v]) {
      forest.add_edge(u, v);
      dfs_rooted_forest(forest, graph, v, vis);
    }
  }
}
} // namespace

template <typename A, typename B>
RootedForest<A, B> make_rooted_forest(const Graph<A, B> &graph,
                                      const vector<int> &roots) {
  RootedForest<A, B> res(graph.size());
  vector<bool> vis(graph.size());
  for (int i : roots)
    if (!vis[i])
      dfs_rooted_forest(res, graph, i, vis);
  for (int i = 0; i < graph.size(); i++)
    if (!vis[i])
      dfs_rooted_forest(res, graph, i, vis);
  return res;
}
} // namespace builders
} // namespace graph
} // namespace lib

#endif
```

1.13. HLD

```
#ifndef _LIB_HLD
#define _LIB_HLD
#include "Graph.cpp"
#include "Segtree.cpp"
#include "Traits.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace graph {
namespace {
void empty_lifter(int a, int b, bool inv) {}
} // namespace

template <typename G> struct HLD {
  G graph;
  vector<int> in, out, rin;
  vector<int> L, sz, ch;
  int tempo;

  HLD(const G &g)
      : graph(g), in(g.size()), out(g.size()), rin(g.size()), L(g.size()),
        sz(g.size()), ch(g.size()) {
    build();
  }

  inline int size() const { return graph.size(); }
```

```

  void dfs0(int u) {
    sz[u] = 1;
    for (auto &k : graph.adj[u]) {
      int v = graph.edge(k).to;
      L[v] = L[u] + 1;
      dfs0(v);
      if (sz[v] > sz[graph.edge(graph.adj[u][0]).to])
        swap(k, graph.adj[u][0]);
      sz[u] += sz[v];
    }
  }

  void dfs1(int u) {
    in[u] = tempo++;
    rin[in[u]] = u;

    if (graph.adj[u].size() > 0) {
      int v = graph.edge(graph.adj[u][0]).to;
      ch[v] = ch[u];
      dfs1(v);
      for (size_t i = 1; i < graph.adj[u].size(); i++) {
        v = graph.edge(graph.adj[u][i]).to;
        ch[v] = v;
        dfs1(v);
      }
    }
    out[u] = tempo;
  }

  void build() {
    vector<int> roots = graph.roots();
    for (int i : roots)
      dfs0(i);
    tempo = 0;
    for (int i : roots)
      dfs1(i);
  }

  template <typename Lifter>
  inline void operate_on_subtree(int u, Lifter &lifter) {
    lifter(in[u], out[u] - 1, false);
  }

  template <typename T, typename QueryIssuer>
  inline T query_on_subtree(int u, const QueryIssuer &issuer) {
    return issuer(in[u], out[u] - 1);
  }

  template <typename Lifter>
  inline void operate_on_subtree_edges(int u, Lifter &lifter) {
    if (in[u] + 2 <= out[u])
      lifter(in[u] + 1, out[u] - 1, false);
  }

  template <typename T, typename QueryIssuer>
  inline void query_on_subtree_edges(int u, const QueryIssuer &issuer) {
    assert(in[u] + 2 <= out[u]);
    return issuer(in[u] + 1, out[u] - 1);
  }

  template <bool is_edge, typename Lifter>
  int _query_path(int u, int v, Lifter &lifter) {
    int inv = 0;
    for (; ch[u] != ch[v]; u = graph.parent(ch[u])) {
      if (L[ch[u]] < L[ch[v]])
        swap(u, v), inv ^= 1;
      lifter(in[ch[u]], in[u], (bool)inv);
    }
    if (L[u] > L[v])
      swap(u, v), inv ^= 1;
    inv ^= 1;
    if (is_edge && in[u] + 1 <= in[v])
      lifter(in[u] + 1, in[v], (bool)inv);
    else if (!is_edge)
      lifter(in[u], in[v], (bool)inv);
    return u;
  }

  template <typename Lifter>
  inline int operate_on_path(int u, int v, Lifter &lifter) {
    return _query_path<false>(u, v, lifter);
  }

  template <typename Lifter>
  inline int operate_on_path_edges(int u, int v, Lifter &lifter) {
    return _query_path<true>(u, v, lifter);
  }

  template <typename Op> inline void operate_on_vertex(int u, Op &op) {
    op(in[u]);
  }

  template <typename T, typename QueryIssuer>
  inline T query_on_vertex(int u, const QueryIssuer &issuer) {
    return issuer(in[u]);
  }

  inline int lca(int u, int v) {
    return _query_path<false>(u, v, empty_lifter);
  }

  inline int dist(int u, int v) {
    int uv = lca(u, v);
    return L[u] + L[v] - 2 * L[uv];
  }
};

template <typename G> HLD<G> make_hld(const G &graph) { return
    HLD<G>(graph); }
} // namespace graph
} // namespace lib

#endif
```

1.14. Karatsuba

```
#ifndef _LIB_KARATSUBA
#define _LIB_KARATSUBA
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace math {
struct Karatsuba {
  template <typename Field>
  vector<Field> multiply(const vector<Field> &a, const vector<Field> &b)
    const {
    if (b.size() == 0)
```

```
      return {};
    if (b.size() == 1) {
      vector<Field> res = a;
      for (Field &res : a)
        res *= b[0];
    }

    int shift = a.size() / 2;
    vector<Field> a0 = a;
    vector<Field> b0 = b;
    a0.resize(min(shift, a.size()));
    b0.resize(min(shift, b.size()));
  }

  template <typename Field>
  vector<Field> operator()(const vector<Field> &a,
                           const vector<Field> &b) const {
    if (a.size() >= b.size())
      return multiply(a, b);
    else
      return multiply(b, a);
  }
};
} // namespace math
} // namespace lib

#endif
```

1.15. Lagrange

```
#ifndef _LIB_LAGRANGE
#define _LIB_LAGRANGE
#include <bits/stdc++.h>
#include "Combinatorics.cpp"

namespace lib {
using namespace std;
namespace linalg {
template <typename Field> struct PrefixLagrange {
  vector<Field> pref, suf;
  PrefixLagrange() {}

  void ensure(int n) {
    int o = pref.size();
    if (n <= o)
      return;
    pref.resize(n), suf.resize(n);
  }

  template <typename T> Field eval(const vector<Field> &v, T x) {
    using C = Combinatorics<Field>;
    assert(!v.empty());
    int d = (int)v.size() - 1;
    if (x <= d)
      return v[x];

    ensure(d + 1);

    Field a = x;
    pref[0] = suf[d] = 1;
    for (T i = 0; i < d; i++)
      pref[i + 1] = pref[i] * a, a -= 1;
    for (T i = d; i; i--)
      suf[i - 1] = suf[i] * a, a += 1;

    Field ans = 0;
    for (int i = 0; i <= d; i++) {
      Field l = pref[i] * suf[i] * C::ifactorial(i) * C::ifactorial(d-i) *
    v[i];
      if ((d + i) & 1)
        l = -l;
      ans += l;
    }
    return ans;
  }
};

template<typename T, typename U>
T lagrange_iota(const vector<T>& f, U n) {
  static PrefixLagrange<T> lag;
  return lag.eval(f, n);
}

template<typename T, typename U>
T lagrange_iota_sum(const vector<T>& f, U n) {
  int m = f.size();
  vector<T> g(m + 1);
  for(int i = 1; i <= m; i++)
      g[i] = g[i-1] + f[i-1];
  return lagrange_iota(g, n);
}
} // namespace linalg
} // namespace lib

#endif
```

1.16. LinearProgram

```
#ifndef _LIB_LINEAR_PROGRAM
#define _LIB_LINEAR_PROGRAM
#include "Simplex.cpp"
#include "Symbolic.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
template <typename T = double> struct LinearProgram {
  struct ConstraintVisitor : StackVisitor<T> {
    const vector<Variable<T>> &vars;
    const vector<Constraint<T>> &consts;

    vector<vector<T>> A;
    vector<T> b;
    T mult;

    ConstraintVisitor(const vector<Variable<T>> &vars,
                      const vector<Constraint<T>> &consts)
      : vars(vars), consts(consts), mult(1) {
      A = vector<vector<T>>();
      b = vector<T>();
    }

    void populate() {
      for (int i = 0; i < consts.size(); i++) {
        const auto &constraint = consts[i];
        if (constraint.op == ConstraintOperation::less_eq)
          visit_constraint(constraint, 1);
        else if (constraint.op == ConstraintOperation::greater_eq)
```

```
        visit_constraint(constraint, -1);
      else if (constraint.op == ConstraintOperation::equals)
        visit_constraint(constraint, 1), visit_constraint(constraint, -1);
    }

    // for(int i = 0; i < b.size(); i++) {
    //     for(int j = 0; j < vars.size(); j++) {
    //         cout << A[i][j] << " ";
    //     }
    //     cout << b[i] << endl;
    // }
  }

  void visit_constraint(const Constraint<T> &constraint, T constraint_mult) {
    A.emplace_back(vars.size());
    b.emplace_back();
    mult *= constraint_mult;
    this->visit(constraint.lhs);
    mult = -mult;
    this->visit(constraint.rhs);
    mult = -mult;
    mult *= constraint_mult;
  }

  int index(const Variable<T> &v) const {
    return lower_bound(vars.begin(), vars.end(), v) - vars.begin();
  }

  virtual void visit_variable(const Expression<T> &e) override {
    A.back()[index(e->var)] += this->top() * e->coef * mult;
  }
  virtual void visit_literal(const Expression<T> &e) override {
    b.back() -= this->top() * e->coef * mult;
  }
};

struct ObjectiveVisitor : StackVisitor<T> {
  const vector<Variable<T>> &vars;
  const Expression<T> &obj;

  vector<T> c;
  T mult;

  ObjectiveVisitor(const vector<Variable<T>> &vars, const Expression<T> &obj,
                   T mult)
      : vars(vars), obj(obj), mult(mult) {
    c = vector<T>(vars.size());
  }

  void populate() {
    this->visit(obj);
    // cout << "---" << endl;
    // for(int i = 0; i < vars.size(); i++)
    //     cout << c[i] << " ";
    // cout << endl;
  }

  int index(const Variable<T> &v) const {
    return lower_bound(vars.begin(), vars.end(), v) - vars.begin();
  }

  virtual void visit_variable(const Expression<T> &e) override {
    c[index(e->var)] += this->top() * e->coef * mult;
    }
  };

  vector<Constraint<T>> constraints;
  void add_constraint(Constraint<T> constraint) {
    constraints.push_back(constraint);
  }
  set<Variable<T>> get_variables(const Expression<T> &obj) const {
    auto visitor = make_unique<VariableVisitor<T>>();
    for (const auto &c : constraints) {
      visitor->visit(c.lhs);
      visitor->visit(c.rhs);
    }
    visitor->visit(obj);
    return visitor->seen;
  }
  map<Variable<T>, T> _solve(const Expression<T> &obj, T obj_mult = 1) {
    const auto &variables = get_variables(obj);
    vector<Variable<T>> vs(variables.begin(), variables.end());
    auto visitor = make_unique<ConstraintVisitor>(vs, constraints);
    visitor->populate();
    auto objVisitor = make_unique<ObjectiveVisitor>(vs, obj, obj_mult);
    objVisitor->populate();

    LPSolver<T> solver(visitor->A, visitor->b, objVisitor->c);
    vector<T> ans;
    solver.Solve(ans);
    if (ans.size() < vs.size())
      return {};

    map<Variable<T>, T> res;
    for (int i = 0; i < vs.size(); i++)
      res[vs[i]] = ans[i];
    return res;
  }

  map<Variable<T>, T> maximize(const Expression<T> &obj) { return _solve(obj); }

  map<Variable<T>, T> minimize(const Expression<T> &obj) {
    return _solve(obj, -1);
  }
};
} // namespace lib

#endif
```

1.17. LinearRecurrence

```
#ifndef _LIB_LINEAR_RECURRENCE
#define _LIB_LINEAR_RECURRENCE
#include "PolynomialRing.cpp"
#include "Traits.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace linalg {
namespace {
using traits::HasRandomIterator;
using traits::IsRandomIterator;
} // namespace

template <typename P> struct BMSolver {
```

```
  typedef BMSolver<P> type;
  typedef typename P::field field_type;
  typedef P poly_type;

  vector<field_type> base;
  vector<field_type> T;

  template <
      typename Iterator,
      typename enable_if<IsRandomIterator<Iterator>::value>::type * =
    nullptr>
  void solve(Iterator begin, Iterator end) {
    auto get = [begin](int i) { return *(begin + i); };

    int n = distance(begin, end);

    vector<field_type> C = {1}, B = {1};
    field_type b = 1;
    int L = 0;

    for (int i = 0, x = 1; i < n; i++, x++) {
      // evaluate new element
      field_type d = 0;
      for (size_t j = 0; j < C.size(); j++)
        d += get(i - j) * C[j];
      if (d == 0)
        continue;
      if (2 * L <= i) {
        auto tmp = C;
        if (C.size() < B.size() + x)
          C.resize(B.size() + x);
        field_type coef = d / b;
        for (size_t j = 0; j < B.size(); j++)
          C[j + x] -= coef * B[j];
        L = i + 1 - L;
        B = tmp;
        b = d;
        x = 0;
      } else {
        if (C.size() < B.size() + x)
          C.resize(B.size() + x);
        field_type coef = d / b;
        for (size_t j = 0; j < B.size(); j++)
          C[j + x] -= coef * B[j];
      }
    }

    T = vector<field_type>((int)C.size() - 1);
    for (size_t i = 0; i < T.size(); i++)
      T[i] = -C[i + 1];
    base = vector<field_type>(begin, end);
  }

  template <
      typename Container,
      typename enable_if<HasRandomIterator<Container>::value>::type * =
    nullptr>
  void solve(const Container &container) {
    solve(container.begin(), container.end());
  }

  void solve(const initializer_list<field_type> &l) {
    solve(l.begin(), l.end());
  }

  bool solved() const { return T.size() > 0 && base.size() >= T.size(); }

  void ensure(int nsz) const {
    auto *self = const_cast<type *>(this);
    for (int j = base.size(); j < nsz; j++) {
      field_type acc = 0;
      for (int i = 0; i < (int)T.size(); i++)
        acc += base[j - i - 1] * T[i];
      self->base.push_back(acc);
    }
  }

  poly_type mod_function() const {
    poly_type res;
    int m = T.size();
    res[m] = 1;
    for (int i = 0; i < m; i++)
      res[i] = -T[m - i - 1];
    return res;
  }

  vector<field_type> compute(long long K, int n) {
    assert(n > 0);
    assert(solved());
    vector<field_type> res;
    int N = T.size();
    int cons = min(n, N);

    if (K < (int)base.size()) {
      for (int j = 0; j < n && K + j < (int)base.size(); j++)
        res.push_back({base[K + j]});

      while ((int)res.size() < cons) {
        field_type acc = 0;
        int sz = res.size();
        int mid = min(sz, N);
        for (int i = 0; i < mid; i++)
          acc += res[sz - i - 1] * T[i];
        sz = base.size();
        for (int i = mid; i < N; i++)
          acc += base[sz - 1 - (i - mid)] * T[i];
        res.push_back(acc);
      }
    } else {
      ensure(cons + N - 1);

      poly_type x = poly_type::kth(K, mod_function());

      for (int j = 0; j < cons; j++) {
        field_type acc = 0;
        for (int i = 0; i < N; i++)
          acc += x[i] * base[i + j];
        res.push_back(acc);
      }
    }

    for (int j = res.size(); j < n; j++) {
      field_type acc = 0;
      for (int i = 0; i < N; i++)
        acc += res[j - i - 1] * T[i];
      res.push_back(acc);
    }
    return res;
  }
```

```cpp
  field_type compute(long long K) { return compute(K, 1)[0]; }
};

template<typename Poly>
struct LinearRecurrence {
  typedef LinearRecurrence<Poly> type;
  typedef typename Poly::field field_type;
  typedef Poly poly_type;

  poly_type P, Q;

  LinearRecurrence(const vector<field_type>& base, vector<field_type> T) {
    assert(base.size() == T.size());
    assert(T.back() != field_type());
    for(auto& x : T) x = -x;
    T.insert(T.begin(), field_type(1));
    Q = poly_type(T);
    P = poly_type(base) % T.size() * Q % ((int)T.size() - 1);
  }

  template<typename I>
  field_type compute(I N) {
    auto P1 = P;
    auto Q1 = Q;
    while(N) {
      auto Q2 = Q1;
      for(int i = 1; i < Q2.size(); i += 2) Q2[i] = -Q2[i];
      auto U = P1 * Q2;
      P1 = poly_type();
      for(int i = N % 2, j = 0; j < Q.degree(); j++, i += 2)
        P1[j] = U[i];
      auto A = Q1 * Q2;
      Q1 = poly_type();
      for(int i = 0, j = 0; j <= Q.degree(); j++, i += 2)
        Q1[j] = A[i];
      N /= 2;
      if(N < P.size()) break;
    }
    return (P1 * Q1.inverse())[N];
  }
};
} // namespace linalg
} // namespace lib

#endif
```

1.18. LongMultiplication

```cpp
#ifndef _LIB_LONG_MULTIPLICATION
#define _LIB_LONG_MULTIPLICATION
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace math {
struct NaiveMultiplication {
  template<typename T>
  using Transform = void;

  template <typename Field>
  vector<Field> operator()(const vector<Field> &a,
                           const vector<Field> &b) const {
    vector<Field> res(a.size() + b.size());
    for (size_t i = 0; i < a.size(); i++) {
      for (size_t j = 0; j < b.size(); j++) {
        res[i + j] += a[i] * b[j];
      }
    }
    return res;
  }
};

template <typename Mult, typename Field>
vector<Field> shift_conv(const vector<Field> &a, vector<Field> b) {
  if (b.empty())
    return {};
  reverse(b.begin(), b.end());
  int n = a.size();
  int m = b.size();

  auto res = Mult()(a, b);
  return vector<Field>(res.begin() + m - 1, res.end());
}
} // namespace math
} // namespace lib

#endif
```

1.19. Math

```cpp
#ifndef _LIB_MATH
#define _LIB_MATH
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace math {

/// caide keep
template <typename Type> struct DefaultPowerOp {
  Type operator()() const { return Type(1); }
  Type operator()(const Type &a) const { return a; }
  void operator()(Type &x, const Type &a, long long cur) const {
    x *= x;
    if (cur & 1)
      x *= a;
  }
};

template <typename Type, typename Op>
Type generic_power(const Type &a, long long n, Op op) {
  if (n == 0)
    return op();
  Type res = op(a);
  int hi = 63 - __builtin_clzll(n);
  for (int i = hi - 1; ~i; i--) {
    op(res, a, n >> i);
  }
  return res;
}

template <typename Type> Type generic_power(const Type &a, long long n) {
  return generic_power(a, n, DefaultPowerOp<Type>());
}
} // namespace math
} // namespace lib

#endif
```

1.20. Matrix

```cpp
#ifndef _LIB_MATRIX
#define _LIB_MATRIX
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace linalg {

template <typename T>
struct MultCombiner {
  inline constexpr static T default_value = 0;
  void operator()(T& x, const T& a, const T& b) {
    x += a * b;
  }
};

template <typename T, T def = numeric_limits<T>::max(), typename Cmp = less<T>>
struct OptCombiner {
  inline constexpr static T default_value = def;
  void operator()(T& x, const T& a, const T& b) {
    x = Cmp()(a, b) ? a : b;
  }
};

template <typename T, T def = numeric_limits<T>::max(), typename Cmp = less<T>>
struct OptSumCombiner {
  inline constexpr static T default_value = def;
  void operator()(T& x, const T& a, const T& b) {
    auto sum = a + b;
    x = Cmp()(x, sum) ? x : sum;
  }
};

template <typename T, typename Cmp = less<T>>
struct SafeOptSumCombiner {
  inline constexpr static T default_value = numeric_limits<T>::max();
  void operator()(T& x, const T& a, const T& b) {
    if (a == default_value || b == default_value) return;
    T sum;
    if(!__builtin_add_overflow(a, b, &sum))
      x = Cmp()(x, sum) ? x : sum;
  }
};

template <typename T, typename Combiner = MultCombiner<T>> struct Matrix {
  inline constexpr static T default_value = Combiner::default_value;
  typedef long long large_int;
  typedef initializer_list<initializer_list<T>> nested_list;
  vector<T> g;
  int n, m;

  Matrix() {}
  Matrix(int n, int m) : g(n * m), n(n), m(m) {}
  Matrix(const nested_list &l) : Matrix(l.size(), l.begin()->size()) {
    auto it1 = l.begin();
    for (int i = 0; i < n; i++, ++it1) {
      assert((int)it1->size() == m);
      auto it2 = it1->begin();
      for (int j = 0; j < m; j++, ++it2) {
        (*this)(i, j) = *it2;
      }
    }
  }

  inline int rows() const { return n; }
  inline int cols() const { return m; }
  inline int size() const { return n * m; }
  inline bool is_square() const { return n == m; }
  T operator()(const int i, const int j) const { return g[i * m + j]; }
  T &operator()(const int i, const int j) { return g[i * m + j]; }

  Matrix t() const {
    Matrix res(m, n);
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        res(i, j) = (*this)(j, i);
      }
    }
    return res;
  }

  Matrix &operator+=(const Matrix &rhs) {
    assert(n == rhs.n && m == rhs.m);
    int sz = size();
    for (int i = 0; i < sz; i++)
      g[i] += rhs.g[i];
    return *this;
  }
  Matrix &operator-=(const Matrix &rhs) {
    assert(n == rhs.n && m == rhs.m);
    int sz = size();
    for (int i = 0; i < sz; i++)
      g[i] -= rhs.g[i];
    return *this;
  }
  Matrix &operator*=(const Matrix &rhs) {
    assert(n == rhs.n && m == rhs.m);
    int sz = size();
    for (int i = 0; i < sz; i++)
      g[i] *= rhs.g[i];
    return *this;
  }
  Matrix operator-() const {
    Matrix res = *this;
    for (T &t : g)
      t = -t;
    return res;
  }

  friend Matrix operator+(const Matrix &lhs, const Matrix &rhs) {
    Matrix res = lhs;
    return res += rhs;
  }
  friend Matrix<T> operator-(const Matrix &lhs, const Matrix &rhs) {
    Matrix res = lhs;
    return res -= rhs;
  }
  friend Matrix operator*(const Matrix &lhs, const Matrix &rhs) {
    Matrix res = lhs;
    return res *= rhs;
  }
  friend Matrix operator%(const Matrix &lhs, const Matrix &rhs) {
    assert(lhs.m == rhs.n);
    auto res = Matrix::same(lhs.n, rhs.m, Combiner::default_value);
    Combiner combiner;
    for (int i = 0; i < lhs.n; i++) {
```

```
      for (int k = 0; k < lhs.m; k++) {
        for (int j = 0; j < rhs.m; j++) {
          combiner(res(i, j), lhs(i, k), rhs(k, j));
        }
      }
    }
    return res;
  }

  static Matrix id(int n) {
    Matrix res(n, n);
    for (int i = 0; i < n; i++)
      res(i, i) = 1;
    return res;
  }

  static Matrix ones(int n, int m) {
    return same(n, m, 1);
  }

  static Matrix same(int n, int m, T x) {
    Matrix res(n, m);
    res.fill(x);
    return res;
  }

  static Matrix _power(const Matrix &a, large_int p) {
    if (p == 1)
      return a;
    Matrix res = power(a, p >> 1);
    res = res % res;
    if (p & 1)
      res = res % a;
    return res;
  }

  static Matrix power(const Matrix &a, large_int p) {
    assert(p >= 0);
    if (p == 0) {
      assert(a.is_square());
      return Matrix::id(a.n);
    } else if (p == 1)
      return a;
    else
      return _power(a, p);
  }

  friend Matrix operator^(const Matrix &lhs, const large_int rhs) {
    return Matrix::power(lhs, rhs);
  }

  inline void fill(T x) {
    for (T &t : g)
      t = x;
  }

  friend bool operator==(const Matrix &lhs, const Matrix &rhs) {
    assert(lhs.n == rhs.n && lhs.m == rhs.m);
    int sz = size();
    for (int i = 0; i < sz; i++)
      if (lhs.g[i] != rhs.g[i])
        return false;
    return true;
  }
  friend bool operator!=(const Matrix &lhs, const Matrix &rhs) {
    return !(lhs == rhs);
  }

  friend istream &operator>>(istream &input, Matrix &var) {
    for (T &t : var.g)
      input >> t;
    return input;
  }
  friend ostream &operator<<(ostream &output, Matrix &var) {
    for (int i = 0; i < var.n; i++) {
      if (i == 0)
        output << "[";
      else
        output << " ";
      for (int j = 0; j < var.m; j++) {
        if (j)
          output << " ";
        output << var(i, j);
      }
      output << "\n";
    }
    return output << "]";
  }
};
} // namespace linalg
} // namespace lib

#endif
```

1.21. Maxflow

```
#ifndef _LIB_MAX_FLOW
#define _LIB_MAX_FLOW
#include "Graph.cpp"
#include <bits/stdc++.h>
// TODO: L-R flow

namespace lib {
using namespace std;
namespace flow {
template <typename T, typename E> struct Edge {
  T cap;
  bool original;
  E label;
};
template <typename T> struct Edge<T, void> {
  T cap;
  bool original;
};

template <typename T, typename E = void> struct Maxflow {
  typedef Maxflow<T, E> type;
  typedef Edge<T, E> flow_edge_type;
  typedef lib::graph::DirectedGraph<void, flow_edge_type> graph;
  using edge_type = typename graph::edge_type;

  graph g;
  int source, sink;
  vector<bool> visited;
  vector<int> dist;
  vector<size_t> used;

  explicit Maxflow(int n) : g(n), source(n - 2), sink(n - 1) { assert(n >=
    2); }
```

```
  void setup(int a, int b) { source = a, sink = b; }
  void add_fake_edge(int u, int v, T weight) {
    g.add_edge(u, v) = {weight, false};
    g.add_edge(v, u) = {0, false};
  }
  template <typename S = E,
            typename enable_if<is_void<S>::value>::type * = nullptr>
  void add_edge(int u, int v, T weight = 1) {
    g.add_edge(u, v) = {weight, true};
    g.add_edge(v, u) = {0, true};
  }
  template <typename S = E,
            typename enable_if<!is_void<S>::value>::type * = nullptr>
  void add_edge(int u, int v, T weight = 1, S data = S()) {
    g.add_edge(u, v) = {weight, true, data};
    g.add_edge(v, u) = {0, true, S()};
  }
  inline int size() const { return g.size(); }
  inline int edge_size() const { return g.edge_size(); }
  edge_type reverse(int i) const { return g.edge(i ^ 1); }
  edge_type edge(int i) const { return g.edge(i); }
  flow_edge_type &flow_edge(int i) { return g.edge(i).data; }
  flow_edge_type &reverse_flow_edge(int i) { return g.edge(i ^ 1).data; }

  bool layered_bfs() {
    int n = size();
    dist.assign(n, -1);
    dist[source] = 0;
    vector<int> q;
    q.reserve(n);
    q.push_back(source);

    for (size_t i = 0; i < q.size(); i++) {
      int u = q[i];
      if (u == sink)
        break;
      for (const auto &e : g.n_edges(u)) {
        if (dist[e.to] == -1 && e.data.cap > 0) {
          dist[e.to] = dist[u] + 1;
          q.push_back(e.to);
        }
      }
    }

    return dist[sink] != -1;
  }

  T augmenting_path(const int u, const T bottle) {
    if (!bottle)
      return 0;
    if (u == sink)
      return bottle;
    for (size_t &i = used[u]; i < g.adj[u].size(); i++) {
      int x = g.adj[u][i];
      auto &e = g.edge(x);
      if (dist[e.to] != dist[u] + 1)
        continue;
      T cf = augmenting_path(e.to, min(bottle, e.data.cap));
      e.data.cap -= cf;
      g.edge(x ^ 1).data.cap += cf;
      if (cf)
        return cf;
    }
    return 0;
  }

  T blocking_flow() {
    if (!layered_bfs())
      return 0;
    used.assign(size(), 0);
    T aug, flow = 0;
    while ((aug = augmenting_path(source, numeric_limits<T>::max())))
      flow += aug;
    return flow;
  }

  T maxflow() {
    T aug, flow = 0;
    while ((aug = blocking_flow()))
      flow += aug;
    return flow;
  }

  vector<bool> mincut() const {
    int n = size();
    vector<bool> vis(n);
    vector<int> q;
    q.reserve(n);
    q.push_back(source);
    vis[source] = true;
    for (size_t i = 0; i < q.size(); i++) {
      int u = q[i];
      for (const auto &e : g.n_edges(u)) {
        if (e.data.cap > 0 && !vis[e.to]) {
          q.push_back(e.to);
          vis[e.to] = true;
        }
      }
    }
    return vis;
  }
};
} // namespace flow
} // namespace lib

#endif
```

1.22. ModularInteger

```
#ifndef _LIB_MODULAR_INTEGER
#define _LIB_MODULAR_INTEGER
#include "NumberTheory.cpp"
#include <bits/stdc++.h>

#if __cplusplus < 201300
#error required(c++14)
#endif

namespace lib {
using namespace std;
namespace {
template <typename T, T... Mods> struct ModularIntegerBase {
  typedef ModularIntegerBase<T, Mods...> type;

  T x[sizeof...(Mods)];
  friend ostream &operator<<(ostream &output, const type &var) {
    output << "(";
    for (int i = 0; i < sizeof...(Mods); i++) {
      if (i)
```

```
        output << ", ";
      output << var.x[i];
    }
    return output << ")";
  }
};

template <typename T, T Mod> struct ModularIntegerBase<T, Mod> {
  typedef ModularIntegerBase<T, Mod> type;
  constexpr static T mod = Mod;

  T x[1];

  T& data() { return this->x[0]; }
  T data() const { return this->x[0]; }
  explicit operator int() const { return this->x[0]; }
  explicit operator int64_t() const { return this->x[0]; }
  explicit operator double() const { return this->x[0]; }
  explicit operator long double() const { return this->x[0]; }
  friend ostream &operator<<(ostream &output, const type &var) {
    return output << var.x[0];
  }
};

template<typename T, typename U, T... Mods>
struct InversesTable {
  constexpr static size_t n_mods = sizeof...(Mods);
  constexpr static T mods[sizeof...(Mods)] = {Mods...};
  constexpr static int n_inverses = 1e6 + 10;

  T v[n_inverses][n_mods];
  T max_x;

  InversesTable() : v(), max_x(n_inverses) {
    for(int j = 0; j < sizeof...(Mods); j++)
      v[1][j] = 1, max_x = min(max_x, mods[j]);
    for(int i = 2; i < max_x; i++) {
      for(int j = 0; j < sizeof...(Mods); j++) {
        v[i][j] = mods[j] - (T)((U)(mods[j] / i) * v[mods[j] % i][j] % mods[j]);
      }
    }
  }
};

// Make available for linkage.
template <typename T, class U, T... Mods>
constexpr T InversesTable<T, U, Mods...>::mods[];

template <typename T, class Enable, T... Mods>
struct ModularIntegerImpl : ModularIntegerBase<T, Mods...> {
  typedef ModularIntegerImpl<T, Enable, Mods...> type;
  typedef T type_int;
  typedef uint64_t large_int;
  constexpr static size_t n_mods = sizeof...(Mods);
  constexpr static T mods[sizeof...(Mods)] = {Mods...};
  using ModularIntegerBase<T, Mods...>::x;
  using Inverses = InversesTable<T, large_int, Mods...>;

  struct Less {
    bool operator()(const type &lhs, const type &rhs) const {
      for (size_t i = 0; i < sizeof...(Mods); i++)
        if (lhs.x[i] != rhs.x[i])
          return lhs.x[i] < rhs.x[i];
      return false;
    };
  };
  typedef Less less;


  constexpr ModularIntegerImpl() {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      x[i] = T();
  }
  constexpr ModularIntegerImpl(large_int y) {
    for (size_t i = 0; i < sizeof...(Mods); i++) {
      x[i] = y % mods[i];
      if (x[i] < 0)
        x[i] += mods[i];
    }
  }
  static type with_remainders(T y[sizeof...(Mods)]) {
    type res;
    for (size_t i = 0; i < sizeof...(Mods); i++)
      res.x[i] = y[i];
    res.normalize();
    return res;
  }

  inline void normalize() {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if ((x[i] %= mods[i]) < 0)
        x[i] += mods[i];
  }

  inline T operator[](int i) const { return x[i]; }

  inline T multiply(T a, T b, T mod) const { return (large_int)a * b % mod; }

  inline T inv(T a, T mod) const { return static_cast<T>(nt::inverse(a, mod)); }

  inline T invi(T a, int i) const {
    const static Inverses inverses = Inverses();
    if(a < inverses.max_x)
      return inverses.v[a][i];
    return inv(a, mods[i]);
  }

  type inverse() const {
    T res[sizeof...(Mods)];
    for (size_t i = 0; i < sizeof...(Mods); i++)
      res[i] = invi(x[i], i);
    return type::with_remainders(res);
  }

  template <typename U> T power_(T a, U p, T mod) {
    if (mod == 1)
      return T();
    if (p < 0) {
      if (a == 0)
        throw domain_error("0^p with negative p is invalid");
      p = -p;
      a = inv(a, mod);
    }
    if (p == 0)
      return T(1);
    if (p == 1)
      return a;
    T res = 1;
```

```
    while (p > 0) {
      if (p & 1)
        res = multiply(res, a, mod);
      p >>= 1;
      a = multiply(a, a, mod);
    }
    return res;
  }

  inline type &operator+=(const type &rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if ((x[i] += rhs.x[i]) >= mods[i])
        x[i] -= mods[i];
    return *this;
  }
  inline type &operator-=(const type &rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if ((x[i] -= rhs.x[i]) < 0)
        x[i] += mods[i];
    return *this;
  }
  inline type &operator*=(const type &rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      x[i] = multiply(x[i], rhs.x[i], mods[i]);
    return *this;
  }
  inline type &operator/=(const type &rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      x[i] = multiply(x[i], invi(rhs.x[i], i), mods[i]);
    return *this;
  }

  inline type &operator+=(T rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if ((x[i] += rhs) >= mods[i])
        x[i] -= mods[i];
    return *this;
  }

  type &operator-=(T rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if ((x[i] -= rhs) < 0)
        x[i] += mods[i];
    return *this;
  }

  type &operator*=(T rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      x[i] = multiply(x[i], rhs, mods[i]);
    return *this;
  }

  type &operator/=(T rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      x[i] = multiply(invi(rhs, i), x[i], mods[i]);
    return *this;
  }

  type &operator^=(large_int p) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      x[i] = power_(x[i], p, mods[i]);
    return *this;
  }

  type &operator++() {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if ((++x[i]) >= mods[i])
        x[i] -= mods[i];
    return *this;
  }
  type &operator--() {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if ((--x[i]) < 0)
        x[i] += mods[i];
    return *this;
  }
  type operator++(int unused) {
    type res = *this;
    ++(*this);
    return res;
  }
  type operator--(int unused) {
    type res = *this;
    --(*this);
    return res;
  }

  friend type operator+(const type &lhs, const type &rhs) {
    type res = lhs;
    return res += rhs;
  }
  friend type operator-(const type &lhs, const type &rhs) {
    type res = lhs;
    return res -= rhs;
  }
  friend type operator*(const type &lhs, const type &rhs) {
    type res = lhs;
    return res *= rhs;
  }
  friend type operator/(const type &lhs, const type &rhs) {
    type res = lhs;
    return res /= rhs;
  }

  friend type operator+(const type &lhs, T rhs) {
    type res = lhs;
    return res += rhs;
  }

  friend type operator-(const type &lhs, T rhs) {
    type res = lhs;
    return res -= rhs;
  }

  friend type operator*(const type &lhs, T rhs) {
    type res = lhs;
    return res *= rhs;
  }

  friend type operator/(const type &lhs, T rhs) {
    type res = lhs;
    return res /= rhs;
  }

  friend type operator^(const type &lhs, large_int rhs) {
    type res = lhs;
    return res ^= rhs;
  }

  friend type power(const type &lhs, large_int rhs) { return lhs ^ rhs; }
```

```cpp

  type operator-() const {
    type res = *this;
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if (res.x[i])
        res.x[i] = mods[i] - res.x[i];
    return res;
  }

  friend bool operator==(const type &lhs, const type &rhs) {
    for (size_t i = 0; i < sizeof...(Mods); i++)
      if (lhs.x[i] != rhs.x[i])
        return false;
    return true;
  }
  friend bool operator!=(const type &lhs, const type &rhs) {
    return !(lhs == rhs);
  }

  friend istream &operator>>(istream &input, type &var) {
    T y;
    cin >> y;
    var = y;
    return input;
  }
};
} // namespace

// Explicitly make constexpr available for linkage.
template <typename T, class Enable, T... Mods>
constexpr T ModularIntegerImpl<T, Enable, Mods...>::mods[];

template <typename T, T... Mods>
using ModularInteger =
    ModularIntegerImpl<T, typename enable_if<is_integral<T>::value>::type,
                       Mods...>;

template <int32_t... Mods> using Mint32 = ModularInteger<int32_t, Mods...>;

template <int64_t... Mods> using Mint64 = ModularInteger<int64_t, Mods...>;

using MintP = Mint32<(int32_t)1e9+7>;
using MintNTT = Mint32<998244353>;
} // namespace lib

#endif
```

1.23. MontgomeryInteger

```cpp
#ifndef _LIB_MONTGOMERY_INTEGER
#define _LIB_MONTGOMERY_INTEGER
#include <bits/stdc++.h>

#if __cplusplus < 201300
#error required(c++14)
#endif

namespace lib {
using namespace std;
namespace {
template <typename U, U Mod>
struct MontgomeryIntegerImpl {
  using S = make_signed_t<U>;
  using T = make_unsigned_t<U>;
  typedef MontgomeryIntegerImpl<U, Mod> type;
  constexpr static T mod = (T)Mod;

  T x;
  typedef U type_int;
  typedef uint64_t large_int;

  constexpr static T get_r() {
    T ret = Mod;
    for(int i = 0; i < 4; i++)
      ret *= 2 - mod * ret;
    return ret;
  }

  constexpr static T r = get_r();
  constexpr static T n2 = -large_int(mod) % mod;
  static_assert(r * mod == 1, "assert(r * mod == 1)");
  static_assert(mod < (1 << 30), "assert(mod < 2^30)");
  static_assert(mod % 2 == 1, "assert(mod % 2 == 1)");

  constexpr MontgomeryIntegerImpl() : x(0) {}
  constexpr MontgomeryIntegerImpl(large_int y)
    : x(reduce(large_int(y % mod + mod) * n2)) {}

  constexpr inline static T reduce(large_int y) {
    return (y + large_int(T(y) * T(-r)) * mod) >> 32;
  }

  constexpr inline type &operator+=(const type &rhs) {
    if(S(x += rhs.x - 2 * mod) < 0) x += 2 * mod;
    return *this;
  }
  constexpr inline type &operator-=(const type &rhs) {
    if(S(x -= rhs.x) < 0) x += 2 * mod;
    return *this;
  }
  constexpr inline type &operator*=(const type &rhs) {
    x = reduce(large_int(x) * rhs.x);
    return *this;
  }
  constexpr inline type &operator/=(const type &rhs) {
    return *this *= rhs.inverse();
  }

  constexpr inline type inverse() const {
    return (*this).power(large_int(mod - 2));
  }

  constexpr type &operator^=(large_int p) {
    return *this = power(p);
  }

  constexpr type &operator++() {
    return *this += type(1);
  }
  constexpr type &operator--() {
    return *this -= type(1);
  }
  constexpr type operator++(int unused) {
    type res = *this;
    ++(*this);
    return res;
  }
  constexpr type operator--(int unused) {
    type res = *this;
```

```
    --(*this);
    return res;
  }

  friend constexpr type operator+(const type &lhs, const type &rhs) {
    type res = lhs;
    return res += rhs;
  }
  friend constexpr type operator-(const type &lhs, const type &rhs) {
    type res = lhs;
    return res -= rhs;
  }
  friend constexpr type operator*(const type &lhs, const type &rhs) {
    type res = lhs;
    return res *= rhs;
  }
  friend constexpr type operator/(const type &lhs, const type &rhs) {
    type res = lhs;
    return res /= rhs;
  }

  friend constexpr type operator^(const type &lhs, large_int rhs) {
    type res = lhs;
    return res ^= rhs;
  }

  friend constexpr type power(const type &lhs, large_int rhs) {
    return lhs.power(rhs);
  }

  constexpr type operator-() const {
    return type() - *this;
  }

  constexpr type power(large_int rhs) const {
    type ret(1), mul(*this);
    while(rhs > 0) {
      if(rhs&1) ret *= mul;
      mul *= mul;
      rhs /= 2;
    }
    return ret;
  }

  constexpr T get() const {
    T ret = reduce(x);
    return ret >= mod ? ret - mod : ret;
  }

  friend bool operator==(const type &lhs, const type &rhs) {
    return lhs.get() == rhs.get();
  }
  friend bool operator!=(const type &lhs, const type &rhs) {
    return !(lhs == rhs);
  }

  explicit operator int() const { return get();; }
  explicit operator int64_t() const { return get(); }
  explicit operator long long() const { return get(); }
  explicit operator double() const { return get(); }
  explicit operator long double() const { return get(); }
  friend ostream &operator<<(ostream &output, const type &var) {
    return output << var.get();
  }

  friend istream &operator>>(istream &input, type &var) {
    T y;
    cin >> y;
    var = type(y);
    return input;
  }
};
} // namespace


template <typename T, T... Mods>
using MontgomeryInteger = MontgomeryIntegerImpl<T, Mods...>;

template <int32_t... Mods> using Mont32 = MontgomeryInteger<int32_t,
    Mods...>;

using MontP = Mont32<(int32_t)1e9+7>;
using MontNTT = Mont32<998244353>;
} // namespace lib

#endif
```

1.24. NTT

```
#ifndef _LIB_NTT
#define _LIB_NTT
#include <bits/stdc++.h>
#include "DFT.cpp"
#include "NumberTheory.cpp"
#include "VectorN.cpp"

namespace lib {
using namespace std;
namespace linalg {
template<typename T>
struct MintRootProvider {
  static size_t max_sz;
  static T g;
  static vector<T> w;

  MintRootProvider() {
    if(g == 0) {
      auto acc = T::mod-1;
      while(acc % 2 == 0) acc /= 2, max_sz++;

      auto factors = nt::factors(T::mod - 1);
      for(g = 2; (typename T::type_int)g < T::mod; g++) {
        bool ok = true;
        for(auto f : factors) {
          if(power(g, (T::mod-1)/f) == 1) {
            ok = false;
            break;
          }
        }
        if(ok) break;
      }
      assert(g != 0);
    }
  }

  pair<T, T> roots(int num, int den) {
    auto p = g ^ ((long long)(T::mod - 1) / den * num);
    return {p, p.inverse()};
  }
```

```cpp

  T operator()(int n, int k) {
    return power(g, (T::mod-1)/(n/k));
  }
  void operator()(int n) {
    n = max(n, 2);
    int k = max((int)w.size(), 2);
    assert(n <= (1LL << max_sz));
    if ((int)w.size() < n)
      w.resize(n);
    else
      return;
    w[0] = w[1] = 1;
    for (; k < n; k *= 2) {
      T step = power(g, (T::mod-1)/(2*k));
      for(int i = k; i < 2*k; i++)
        w[i] = (i&1) ? w[i/2] * step : w[i/2];
    }
  }
  T operator[](int i) {
    return w[i];
  }

  T inverse(int n) {
    return T(1) / n;
  }
};

template<typename T>
size_t MintRootProvider<T>::max_sz = 1;
template<typename T>
T MintRootProvider<T>::g = T();
template<typename T>
vector<T> MintRootProvider<T>::w = vector<T>();

template<typename T>
struct NTT : public DFT<T, MintRootProvider<T>> {
  using Parent = DFT<T, MintRootProvider<T>>;
  using Parent::fa;
  using Parent::dft;
  using Parent::idft;

  static void _convolve(const vector<T> &a) {
    int n = Parent::ensure(a.size(), a.size());
    for (size_t i = 0; i < (size_t)n; i++)
      fa[i] = i < a.size() ? a[i] : T();
    Parent::dft(n);
    for (int i = 0; i < n; i++)
      fa[i] *= fa[i];
    Parent::idft(n);
  }

  static void _convolve(const vector<T> &a, const vector<T> &b) {
    if(std::addressof(a) == std::addressof(b))
      return _convolve(a);
    int n = Parent::ensure(a.size(), b.size());
    for (size_t i = 0; i < (size_t)n; i++)
      fa[i] = i < a.size() ? a[i] : T();
    Parent::dft(n);
    // TODO: have a buffer for this
    auto fb = retrieve<Parent, T>(n);
    for(size_t i = 0; i < (size_t)n; i++)
      fa[i] = i < b.size() ? b[i] : T();
    Parent::dft(n);
    for (int i = 0; i < n; i++)
      fa[i] *= fb[i];
    Parent::idft(n);
  }

  static vector<T> convolve(const vector<T>& a, const vector<T>& b) {
    int sz = (int)a.size() + b.size() - 1;
    _convolve(a, b);
    return retrieve<Parent, T>(sz);
  }

  static VectorN<T> transform(vector<T> a, int n) {
    a.resize(n);
    Parent::dft(a, n);
    return a;
  }

  static vector<T> itransform(vector<T> a, int n) {
    int sz = a.size();
    Parent::idft(a, sz);
    a.resize(min(n, sz));
    return a;
  }
};
}

struct NTTMultiplication {
  template<typename T>
  using Transform = linalg::NTT<T>;

  template <typename Field>
  vector<Field> operator()(const vector<Field> &a,
                           const vector<Field> &b) const {
    return linalg::NTT<Field>::convolve(a, b);
  };

  template<typename Field>
  inline VectorN<Field> transform(int n, const vector<Field>& p) const {
    int np = next_power_of_two(n);
    return linalg::NTT<Field>::transform(p, np);
  }

  template<typename Field>
  inline vector<Field> itransform(int n, const vector<Field>& p) const {
    return linalg::NTT<Field>::itransform(p, n);
  }

  template <typename Field, typename Functor, typename ...Ts>
  inline vector<Field> on_transform(
    int n,
    Functor& f,
    const vector<Ts>&... vs) const {
    int np = next_power_of_two(n);
    return linalg::NTT<Field>::itransform(
      f(n, linalg::NTT<Field>::transform(vs, np)...), n);
  }
};
} // namespace lib

#endif
```

1.25. NumberTheory

```cpp
#ifndef _LIB_NUMBER_THEORY
#define _LIB_NUMBER_THEORY
```

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

namespace lib {
using namespace std;
namespace nt {
int64_t inverse(int64_t a, int64_t b) {
  long long b0 = b, t, q;
  long long x0 = 0, x1 = 1;
  if (b == 1)
    return 1;
  while (a > 1) {
    q = a / b;
    t = b, b = a % b, a = t;
    t = x0, x0 = x1 - q * x0, x1 = t;
  }
  if (x1 < 0)
    x1 += b0;
  return x1;
}
template<typename T, typename U>
T powmod (T a, U b, U p) {
    int res = 1;
    while (b)
        if (b & 1)
            res = (int) (res * 1ll * a % p),  --b;
        else
            a = (int) (a * 1ll * a % p),  b >>= 1;
    return res;
}
template<typename T>
vector<T> factors(T n) {
  vector<T> f;
  for(T i = 2; i*i <= n; i++) {
    if(n % i == 0) f.push_back(i);
    while(n % i == 0) n /= i;
  }
  if(n > 1) f.push_back(n);
  return f;
}
} // namespace nt
} // namespace lib

#endif
```

1.26. OfflineRMQ

```cpp
#ifndef _LIB_OFFLINE_RMQ
#define _LIB_OFFLINE_RMQ
#include <bits/stdc++.h>
#include "DSU.cpp"

namespace lib {
using namespace std;

// O(n + qlogn)
template<typename T, typename U = T>
vector<T> offline_rmq(const vector<T>& v, const vector<pair<U, U>>& qrs) {
  int n = v.size();
  vector<vector<pair<U, int>>> cont(n);
  for(int i = 0; i < (int)qrs.size(); i++) {
    auto p = qrs[i];
    cont[p.second].push_back({p.first, i});
  }
  vector<T> ans(qrs.size());

  CompressedDSU dsu(n);
  vector<U> s;
  for(int i = 0; i < n; i++) {
    while(!s.empty() && v[s.back()] > v[i]) {
      dsu.parent(s.back()) = i;
      s.pop_back();
    }
    s.push_back(i);
    for(auto p : cont[i]) {
      ans[p.second] = v[dsu[p.first]];
    }
  }
  return ans;
}
} // namespace lib

#endif
```

1.27. PolynomialRing

```cpp
#ifndef _LIB_POLYNOMIAL_RING
#define _LIB_POLYNOMIAL_RING
#include "Epsilon.cpp"
#include "Math.cpp"
#include "ModularInteger.cpp"
#include "Traits.cpp"
#include "LongMultiplication.cpp"
#include "VectorN.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace math {
namespace poly {

namespace {
/// keep caide
using traits::IsInputIterator;
/// keep caide
using traits::HasInputIterator;
} // namespace

namespace detail {
  template<class>
  struct sfinae_true : std::true_type{};

  template<class T, class Field, class Func>
  static auto test_transform(int)
      -> sfinae_true<decltype(
    std::declval<T>().template on_transform<Field>(std::declval<int>(),
     std::declval<Func&>()))>;

  template<class, class Field, class Func>
  static auto test_transform(long) -> std::false_type;
} // detail::

template<class T, class Field, class Func =
     std::function<VectorN<Field>(int)>>
struct has_transform : decltype(detail::test_transform<T, Field, Func>(0)){};

template <typename P> struct DefaultPowerOp {
  int mod;
  DefaultPowerOp(int mod) : mod(mod) {}
```

```
  P operator()() const { return P(1); }
  P operator()(const P &a) const { return a % mod; }
  void operator()(P &x, const P &a, long long cur) const {
    (x *= x) %= mod;
    if (cur & 1)
      (x *= a) %= mod;
  }
};

template <typename P> struct ModPowerOp {
  const P &mod;
  ModPowerOp(const P &p) : mod(p) {}
  P operator()() const { return P(1); }
  P operator()(const P &a) { return a % mod; }
  void operator()(P &x, const P &a, long long cur) const {
    (x *= x) %= mod;
    if (cur & 1)
      (x *= a) %= mod;
  }
};

template <typename P> struct ModShiftPowerOp {
  const P &mod;
  ModShiftPowerOp(const P &p) : mod(p) {}
  P operator()() const { return P(1); }
  P operator()(const P &a) { return a % mod; }
  void operator()(P &x, const P &a, long long cur) const {
    // if(cur < mod.degree())
    // x = P::kth(cur);
    if (cur & 1)
      (x *= (x << 1)) %= mod;
    else
      (x *= x) %= mod;
  }
};

struct DefaultDivmod;
struct NaiveDivmod;

template <typename Field, typename Mult, typename Divmod = DefaultDivmod>
struct Polynomial {
  constexpr static int Magic = 64;
  constexpr static bool NaiveMod = is_same<Divmod, NaiveDivmod>::value;
  constexpr static bool HasTransform = has_transform<Mult, Field>::value;
  using Transform =  typename Mult::template Transform<Field>;

  typedef Polynomial<Field, Mult, Divmod> type;
  typedef Field field;
  vector<Field> p;

  Polynomial() : p(0) {}
  explicit Polynomial(Field x) : p(1, x) {}

  template <
      typename Iterator,
      typename enable_if<IsInputIterator<Iterator>::value>::type * = nullptr>
  Polynomial(Iterator begin, Iterator end) : p(distance(begin, end)) {
    int i = 0;
    for (auto it = begin; it != end; ++it, ++i)
      p[i] = *it;
    normalize();
  }

  template <
      typename Container,
      typename enable_if<HasInputIterator<Container>::value>::type * =
    nullptr>
  Polynomial(const Container &container)
      : Polynomial(container.begin(), container.end()) {}

  Polynomial(const initializer_list<Field> &v)
      : Polynomial(v.begin(), v.end()) {}

  static type from_root(const Field &root) { return Polynomial({-root, 1}); }

  void normalize() const {
    type *self = const_cast<type *>(this);
    int sz = self->p.size();
    while (sz > 0 && Epsilon<>().null(self->p[sz - 1]))
      sz--;
    if (sz != (int)self->p.size())
      self->p.resize(sz);
  }

  inline int size() const { return p.size(); }
  inline int degree() const { return max((int)p.size() - 1, 0); }
  bool null() const {
    for (Field x : p)
      if (!Epsilon<>().null(x))
        return false;
    return true;
  }

  const vector<Field>& data() const {
    return p;
  }

  Field eval(Field x) const {
    Field pw = 1;
    Field res = 0;
    for(Field c : p) {
      res += pw * c;
      pw *= x;
    }
    return res;
  }

  inline Field operator[](const int i) const {
    if (i >= size())
      return 0;
    return p[i];
  }
  inline Field &operator[](const int i) {
    if (i >= size())
      p.resize(i + 1);
    return p[i];
  }

  Field operator()(const Field &x) const {
    if (null())
      return Field();
    Field acc = p.back();
    for (int i = (int)size() - 2; i >= 0; i--) {
      acc *= x;
      acc += p[i];
    }
    return acc;
  }

  type substr(int i, int sz) const {
```

```
    int j = min(sz + i, size());
    i = min(i, size());
    if(i >= j) return type();
    return type(begin(p)+i, begin(p)+j);
  }

  type &operator+=(const type &rhs) {
    if (rhs.size() > size())
      p.resize(rhs.size());
    int sz = size();
    for (int i = 0; i < sz; i++)
      p[i] += rhs[i];
    normalize();
    return *this;
  }

  type &operator-=(const type &rhs) {
    if (rhs.size() > size())
      p.resize(rhs.size());
    int sz = size();
    for (int i = 0; i < sz; i++)
      p[i] -= rhs[i];
    normalize();
    return *this;
  }

  static vector<Field> multiply(const vector<Field>& a, const vector<Field>& b) {
    if(min(a.size(), b.size()) < Magic)
      return NaiveMultiplication()(a, b);
    return Mult()(a, b);
  }

  type &operator*=(const type &rhs) {
    p = multiply(p, rhs.p);
    normalize();
    return *this;
  }

  type &operator*=(const Field &rhs) {
    int sz = size();
    for (int i = 0; i < sz; i++)
      p[i] *= rhs;
    normalize();
    return *this;
  }

  type &operator/=(const Field &rhs) {
    int sz = size();
    for (int i = 0; i < sz; i++)
      p[i] /= rhs;
    normalize();
    return *this;
  }

  type &operator<<=(const int rhs) {
    if (rhs < 0)
      return *this >>= rhs;
    if (rhs == 0)
      return *this;
    int sz = size();
    p.resize(sz + rhs);
    for (int i = sz - 1; i >= 0; i--)
      p[i + rhs] = p[i];
    fill_n(p.begin(), rhs, 0);
    return *this;
  }

  type &operator>>=(const int rhs) {
    if (rhs < 0)
      return *this <<= rhs;
    if (rhs == 0)
      return *this;
    int sz = size();
    if (rhs >= sz) {
      p.clear();
      return *this;
    }
    for (int i = rhs; i < sz; i++)
      p[i - rhs] = p[i];
    p.resize(sz - rhs);
    return *this;
  }

  type &operator%=(const int rhs) {
    if (rhs < size())
      p.resize(rhs);
    normalize();
    return *this;
  }

  type &operator/=(const type &rhs) { return *this = *this / rhs; }

  type operator%=(const type &rhs) { return *this = *this % rhs; }

  type operator+(const type &rhs) const {
    type res = *this;
    return res += rhs;
  }

  type operator-(const type &rhs) const {
    type res = *this;
    return res -= rhs;
  }

  type operator*(const type &rhs) const { return type(multiply(p, rhs.p)); }

  type operator*(const Field &rhs) const {
    type res = *this;
    return res *= rhs;
  }

  type operator/(const Field &rhs) const {
    type res = *this;
    return res /= rhs;
  }

  type operator<<(const int rhs) const {
    type res = *this;
    return res <<= rhs;
  }

  type operator>>(const int rhs) const {
    type res = *this;
    return res >>= rhs;
  }

  type operator%(const int rhs) const {
    return Polynomial(p.begin(), p.begin() + min(rhs, size()));
  }
```

```

  type operator/(const type &rhs) const {
    return type::divmod(*this, rhs).first;
  }

  type operator%(const type &rhs) const {
    return type::divmod(*this, rhs).second;
  }

  bool operator==(const type &rhs) const {
    normalize();
    rhs.normalize();
    return p == rhs.p;
  }

  template <// Used in SFINAE.
            typename U = Field,
            enable_if_t<has_transform<Mult, U>::value>* = nullptr>
  inline VectorN<U> transform(int n) {
    return Mult().template transform<U>(n, p);
  }

  template <// Used in SFINAE.
            typename U = Field,
            enable_if_t<has_transform<Mult, U>::value>* = nullptr>
  inline static type itransform(int n, const vector<U>& v) {
    return Mult().template itransform<U>(n, v);
  }

  template <typename Functor,
            // Used in SFINAE.
            typename U = Field,
            enable_if_t<has_transform<Mult, U>::value>* = nullptr,
            typename ...Ts>
  inline static type on_transform(
    int n,
    Functor f,
    const Ts&... vs) {
    if(n < Magic)
      return f(n, vs...);
    return Mult().template on_transform<U>(n, f, vs.p...);
  }

  template <typename Functor,
            // Used in SFINAE.
            typename U = Field,
            enable_if_t<!has_transform<Mult, U>::value>* = nullptr,
            typename ...Ts>
  inline static type on_transform(
    int n,
    Functor f,
    const Ts&... vs) {
    return f(n, vs...);
  }

  template <
    // Used in SFINAE.
    typename U = Field,
    enable_if_t<has_transform<Mult, U>::value>* = nullptr>
  type inverse(int m) const {
    if(null()) return *this;
    type r = {Field(1) / p[0]};
    r.p.reserve(m);
    for(int i = 1; i < m; i *= 2) {
      int n = 2 * i;
      vector<U> f = (*this % n).p; f.resize(n);
      vector<U> g = r.p; g.resize(n);
      Transform::dft(f, n);
      Transform::dft(g, n);
      for(int j = 0; j < n; j++) f[j] *= g[j];
      Transform::idft(f, n);
      for(int j = 0; j < i; j++) f[j] = 0;
      Transform::dft(f, n);
      for(int j = 0; j < n; j++) f[j] *= g[j];
      Transform::idft(f, n);
      for(int j = i; j < min(n, m); j++)
        r[j] = -f[j];
    }
    return r;
  }

  type inverse_slow(int m) const {
    if(null()) return *this;
    type b = {Field(1) / p[0]};
    b.p.reserve(2 * m);
    for(int i = 1; i < m; i *= 2) {
      int n = min(2 * i, m);
      auto bb = b * b % n;
      b += b;
      b -= *this % n * bb;
      b %= n;
    }
    return b % m;
  }

  template <
    // Used in SFINAE.
    typename U = Field,
    enable_if_t<!has_transform<Mult, U>::value>* = nullptr>
  type inverse(int m) const {
    return inverse_slow(m);
  }

  type inverse() const {
    return inverse(size());
  }

  type reciprocal() const {
    normalize();
    return type(p.rbegin(), p.rend());
  }

  type integral() const {
    int sz = size();
    if (sz == 0)
      return {};
    type res = *this;
    for (int i = sz; i; i--) {
      res[i] = res[i - 1] / i;
    }
    res[0] = 0;
    res.normalize();
    return res;
  }

  type derivative() const {
    int sz = size();
    if (sz == 0)
      return {};
    type res = *this;
```

```
    for (int i = 0; i + 1 < sz; i++) {
      res[i] = res[i + 1] * (i + 1);
    }
    res.p.back() = 0;
    res.normalize();
    return res;
  }

  type mulx(field x) const { // component-wise multiplication with x^k
    field cur = 1;
    type res(*this);
    for(auto& c : res.p)
      c *= cur, cur *= x;
    return res;
  }
  type mulx_sq(field x) const { // component-wise multiplication with x^{k^2}
    field cur = x;
    field total = 1;
    field xx = x * x;
    type res(*this);
    for(auto& c : res.p)
      c *= total, total *= cur, cur *= xx;
    return res;
  }
  static pair<type, type> divmod(const type &a, const type &b) {
    if (NaiveMod || min(a.size(), b.size()) < Magic)
      return naive_divmod(a, b);
    a.normalize();
    b.normalize();
    int m = a.size();
    int n = b.size();
    if (m < n)
      return {Polynomial(), a};
    int sz = m - n + 1;
    type ar = a.reciprocal() % sz;
    type br = b.reciprocal() % sz;
    type q = (ar * br.inverse(sz) % sz).reciprocal();
    type r = a - b * q;

    return {q, r % (n-1)};
  }

  static pair<type, type> naive_divmod(const type &a, const type &b) {
    type res = a;
    int a_deg = a.degree();
    int b_deg = b.degree();
    Field normalizer = Field(1) / b[b_deg];
    for (int i = 0; i < a_deg - b_deg + 1; i++) {
      Field coef = (res[a_deg - i] *= normalizer);
      if (coef != 0) {
        for (int j = 1; j <= b_deg; j++) {
          res[a_deg - i - j] += -b[b_deg - j] * coef;
        }
      }
    }
    return {res >> b_deg, res % b_deg};
  }
  vector<Field> czt_even(Field z, int n) const { // P(1), P(z^2), P(z^4),
    ..., P(z^2(n-1))
    int m = degree();
    if(null()) {
      return vector<Field>(n);
    }
    vector<Field> vv(m + n);
    Field zi = Field(1) / z;
    Field zz = zi * zi;
    Field cur = zi;
    Field total = 1;
    for(int i = 0; i <= max(n - 1, m); i++) {
      if(i <= m) {vv[m - i] = total;}
      if(i < n) {vv[m + i] = total;}
      total *= cur;
      cur *= zz;
    }
    type w = (mulx_sq(z) * vv).substr(m, n).mulx_sq(z);
    vector<Field> res(n);
    for(int i = 0; i < n; i++) {
      res[i] = w[i];
    }
    return res;
  }
  vector<Field> czt(Field z, int n) const {
    auto even = czt_even(z, (n+1)/2);
    auto odd = mulx(z).czt_even(z, n/2);
    vector<Field> ans(n);
    for(int i = 0; i < n/2; i++) {
      ans[2*i] = even[i];
      ans[2*i+1] = odd[i];
    }
    if(n&1) {
      ans.back() = even.back();
    }
    return ans;
  }
  friend type kmul(const vector<type>& polys, int l, int r) {
    if(l == r) return polys[l];
    int mid = (l+r)/2;
    return kmul(polys, l, mid) * kmul(polys, mid+1, r);
  }
  friend type kmul(const vector<type>& polys) {
    if(polys.empty()) return type();
    return kmul(polys, 0, (int)polys.size() - 1);
  }
  static type power(const type &a, long long n, const int mod) {
    return math::generic_power<type>(a, n, DefaultPowerOp<type>(mod));
  }

  static type power(const type &a, long long n, const type &mod) {
    return math::generic_power<type>(a, n, ModPowerOp<type>(mod));
  }

  static type kth(int K) { return type(1) << K; }

  static type kth(long long K, const type &mod) {
    return math::generic_power<type>(type(1) << 1, K,
                                    ModShiftPowerOp<type>(mod));
  }

  friend ostream &operator<<(ostream &output, const type &var) {
    output << "[";
    int sz = var.size();
    for (int i = sz - 1; i >= 0; i--) {
      output << var[i];
      if (i)
        output << " ";
    }
    return output << "]";
  }
};
} // namespace poly
```

```
/// keep caide
using poly::Polynomial;
} // namespace math
} // namespace lib

#endif
```

1.28. PowerSeries

```
#ifndef _LIB_POWER_SERIES
#define _LIB_POWER_SERIES
#include "BitTricks.cpp"
#include "PolynomialRing.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace series {

template <typename P> P ln(const P &p, int n);

template <typename P> P inverse(P p, int n) {
  return p.inverse(n);
}

template <typename P> P ln(const P &p, int n) {
  return (p.derivative() * inverse(p, n) % n).integral() % n;
}

// \sum ln(1 + x^K), where K are elements of v.
template<typename P, typename I>
P ln_1px(const vector<I>& v, int n) {
  using Field = typename P::field;
  vector<I> h(n);
  vector<Field> res(n);
  for(auto x : v) if(x < n) h[x]++;
  res[0] = h[0];
  for(int i = 1; i < n; i++) {
    if(!h[i]) continue;
    for(int j = 0, k = i; k < n; k += i, j++) {
      Field c = Field(1) / Field(j + 1);
      if(j&1) c = -c;
      res[k] += c * h[i];
    }
  }
  return P(res);
}

template<typename P> pair<P, P> exp2(P p, int n) {
  assert(p[0] == 0);
  P f{1}, g{1};
  for(int i = 1; i <= n; i*=2) {
    g = g * 2 - (g*g%i*f)%i;
    P q = (p % i).derivative();
    q += g * (f.derivative() - f * q) % (2 * i - 1);
    f += f * (p % (2 * i) - q.integral()) % (2 * i);
  }
  return {f % n, g % n};
}

// p[0] must be null
template <typename P> P exp(P p, int n) {
  return exp2(p, n).first;
}

template <typename P> P power(const P &p, long long k, int n) {
  int m = p.size();
  for(int i = 0; i < m; i++) {
    if(p[i] == 0) continue;
    if(i > 0 && k > n / i) return {};
    auto rev = typename P::field(1) / p[i];
    auto D = (p * rev) >> i;
    int sz = n - i * k;
    D = exp(ln(D, sz) * k, sz) * (p[i] ^ k);
    if(i == 0) return D % n;
    long long S = k * i;
    D <<= S;
    return D % n;
  }
  return {};
}
} // namespace series
} // namespace lib

#endif
```

1.29. Random

```
#ifndef _LIB_RANDOM
#define _LIB_RANDOM
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace rng {
struct Generator {
  mt19937 rng;
  Generator() {
    seed_seq seq {
      (uint64_t) chrono::duration_cast<chrono::nanoseconds>(
          chrono::high_resolution_clock::now().time_since_epoch())
          .count(),
#if __cplusplus > 201300
          (uint64_t)make_unique<char>().get(),
#else
          (uint64_t)unique_ptr<char>(new char).get(),
#endif
          (uint64_t)__builtin_ia32_rdtsc()
    };
    rng = mt19937(seq);
  }
  Generator(seed_seq &seq) : rng(seq) {}

  template <typename T,
            typename enable_if<is_integral<T>::value>::type * = nullptr>
  inline T uniform_int(T L, T R) {
    return uniform_int_distribution<T>(L, R)(rng);
  }

  template <typename T> inline T uniform_int(T N) {
    return uniform_int(T(), N - 1);
  }

  template <typename T> inline T uniform_real(T N) {
    return uniform_real(0.0, static_cast<double>(N));
  }

  template <typename T> inline T uniform_real(T L, T R) {
```

```
    return uniform_real_distribution<double>(static_cast<double>(L),
                                             static_cast<double>(R))(rng);
  }

  inline double uniform_real() { return uniform_real(0.0, 1.0); }
};

static Generator gen = Generator();
} // namespace rng
static rng::Generator &rng_gen = rng::gen;
} // namespace lib

#endif
```

1.30. RangeDSU

```
#ifndef _LIB_RANGE_DSU
#define _LIB_RANGE_DSU
#include <bits/stdc++.h>
#include "SegtreeFast.cpp"
#include "DSU.cpp"

namespace lib {
using namespace std;
struct RangeDSU {
  struct NodeImpl {
    int low, high;
    int low_inv, high_inv;
    friend NodeImpl operator+(const NodeImpl& a, const NodeImpl& b) {
      NodeImpl res = a;
      if(b.low < res.low) res.low = b.low, res.low_inv = b.low_inv;
      if(b.high > res.high) res.high = b.high, res.high_inv = b.high_inv;
      return res;
    }
  };
  using Node = seg::Active<NodeImpl>;

  seg::SegtreeFast<Node, seg::CombineFolder<Node>> sg;
  FastDSU dsu;
  vector<vector<int>> inv;

  RangeDSU(int n) : sg(seg::make_builder(n)), dsu(n), inv(n) {
    // TODO: optimize
    for(int i = 0; i < n; i++) {
      sg.update_element(i, seg::SetUpdater<NodeImpl>(node_impl(i)));
      inv[i].push_back(i);
    }
  }

  NodeImpl node_impl(int i) {
    int u = dsu[i];
    return NodeImpl{u, u, i, i};
  }

  void activate(int i) {
    sg.update_element(i, seg::ActiveUpdater<Node>(true));
  }

  void deactivate(int i) {
    sg.update_element(i, seg::ActiveUpdater<Node>(false));
  }

  int operator[](int i) {
    return dsu[i];
  }

  bool merge(int u, int v) {
    if(!dsu.merge(u, v)) return false;
    tie(u, v) = dsu.last_merge();
    for(int x : inv[u]) {
      inv[v].push_back(x);
      sg.update_element(x, seg::SetUpdater<NodeImpl>(node_impl(x)));
    }
    return true;
  }

  int merge_range(int i, int j, int x) {
    x = dsu[x];
    Node res = sg.query<Node>(i, j, seg::CombineFolder<Node>());
    if(!res.is_active()) return -1;
    if(res.low != x) {
      merge(res.low, x);
      return res.low_inv;
    }
    if(res.high != x) {
      merge(res.high, x);
      return res.high_inv;
    }
    return -1;
  }

  void merge_all_range(int i, int j, int x) {
    while(merge_range(i, j, x) != -1);
  }

  pair<int, int> last_merge() const { return dsu.last_merge(); }
  vector<int> last_move() const { return inv[last_merge().first]; }
};
} // namespace lib

#endif
```

1.31. RollingHash

```
#ifndef _LIB_ROLLING_HASH
#define _LIB_ROLLING_HASH
#include "ModularInteger.cpp"
#include "Random.cpp"
#include "Traits.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace hashing {
namespace {
using traits::HasBidirectionalIterator;
using traits::HasInputIterator;
using traits::IsBidirectionalIterator;
using traits::IsInputIterator;
using traits::IsRandomIterator;
} // namespace

const static int DEFAULT_MAX_POWERS = 1e6 + 5;
const static int GOOD_MOD1 = (int)1e9 + 7;
const static int GOOD_MOD2 = (int)1e9 + 9;

template <typename T, T... Mods> struct BaseProvider {
  typedef BaseProvider<T, Mods...> type;
```

```
  typedef ModularInteger<T, Mods...> mint_type;

  mint_type b;
  vector<mint_type> powers;
  int max_powers = 0;

  BaseProvider(T bases[sizeof...(Mods)]) : powers(1, 1) {
    b = mint_type::with_remainders(bases);
  }
  BaseProvider() : powers(1, 1) {
    T bases[sizeof...(Mods)];
    for (size_t i = 0; i < sizeof...(Mods); i++)
      bases[i] = rng::gen.uniform_int(mint_type::mods[i]);
    b = mint_type::with_remainders(bases);
  }

  void set_max_powers(int x) { max_powers = x; }

  inline operator mint_type() const { return b; }
  inline T operator()(int i) { return b[i]; }

  void ensure(int p) const {
    type *self = const_cast<type *>(this);
    int cur = powers.size();
    if (p > cur)
      self->powers.resize(max(2 * cur, p));
    else
      return;
    int nsz = powers.size();
    for (int i = cur; i < nsz; i++)
      self->powers[i] = powers[i - 1] * b;
  }

  mint_type power(int p) const {
    if (p >= max_powers)
      return b ^ p;
    ensure(p + 1);
    return powers[p];
  }
  T power(int p, int i) const { return power(p)[i]; }
};

template <typename T, T... Mods> struct RollingHash {
  typedef RollingHash<T, Mods...> type;
  typedef ModularInteger<T, Mods...> mint_type;
  typedef BaseProvider<T, Mods...> base_type;

  vector<mint_type> hs;

  struct Hash {
    mint_type x;
    int n;

    struct Less {
      typename mint_type::less mint_less;
      bool operator()(const Hash &lhs, const Hash &rhs) const {
        if (lhs.n == rhs.n)
          return mint_less(lhs.x, rhs.x);
        return lhs.n < rhs.n;
      }
    };
    typedef Less less;

    Hash() : n(0) {}
    explicit Hash(mint_type y) : x(y), n(1) {}
    Hash(mint_type y, int n) : x(y), n(n) { assert(n >= 0); }

    explicit operator mint_type() const { return x; }

    friend bool operator==(const Hash &lhs, const Hash &rhs) {
      return tie(lhs.n, lhs.x) == tie(rhs.n, rhs.x);
    }
    friend bool operator!=(const Hash &lhs, const Hash &rhs) {
      return !(lhs == rhs);
    }
    friend ostream &operator<<(ostream &output, const Hash &var) {
      return output << var.x << "{" << var.n << "}";
    }
  };

  struct Cat {
    shared_ptr<base_type> base;
    Cat(const shared_ptr<base_type> &base) : base(base) {}

    template <
        typename Iterator,
        typename enable_if<IsInputIterator<Iterator>::value>::type * =
    nullptr>
    Hash operator()(Iterator begin, Iterator end) {
      Hash res;
      for (auto it = begin; it != end; ++it) {
        res.n += it->n;
        res.x *= base->power(it->n);
        res.x += it->x;
      }
      return res;
    }

    Hash operator()(const initializer_list<Hash> &hashes) {
      return (*this)(hashes.begin(), hashes.end());
    }

    template <class... Args> Hash operator()(Args... args) {
      return (*this)({args...});
    }

    template <class... Args>
    pair<Hash, Hash> cat(const pair<Args, Args> &... args) {
      initializer_list<Hash> fwd_list = {args.first...};
      initializer_list<Hash> bwd_list = {args.second...};
      return {cat(fwd_list.begin(), fwd_list.end()),
              cat(bwd_list.rbegin(), bwd_list.rend())};
    }
  };

  Cat cat;

  RollingHash(const shared_ptr<base_type> &base) : hs(1), cat(base) {}

  template <
      typename Container,
      typename enable_if<HasInputIterator<Container>::value>::type * =
    nullptr>
  RollingHash(const Container &container, const shared_ptr<base_type> &base)
      : hs(1), cat(base) {
    (*this) += container;
  }

  template <
      typename Iterator,
```

```
      typename enable_if<IsInputIterator<Iterator>::value>::type * = nullptr>
  RollingHash(Iterator begin, Iterator end, const shared_ptr<base_type>
    &base)
      : hs(1), cat(base) {
    append(begin, end);
  }

  inline int size() const { return (int)hs.size() - 1; }

  template <
      typename Iterator,
      typename enable_if<IsRandomIterator<Iterator>::value>::type * =
    nullptr>
  void append(Iterator begin, Iterator end) {
    int i = hs.size();
    hs.resize(hs.size() + distance(begin, end));
    for (auto it = begin; it != end; ++it, ++i)
      hs[i] = hs[i - 1] * (*cat.base) + mint_type(*it);
  }

  template <
      typename Iterator,
      typename enable_if<!IsRandomIterator<Iterator>::value>::type * =
    nullptr>
  void append(Iterator begin, Iterator end) {
    for (auto it = begin; it != end; ++it)
      (*this) += *it;
  }

  template <typename U> void append(U rhs) { (*this) += rhs; }

  template <typename U,
            typename enable_if<is_integral<U>::value>::type * = nullptr>
  RollingHash &operator+=(U rhs) {
    hs.push_back(mint_type(rhs) + hs.back() * (*cat.base));
    return *this;
  }

  template <
      typename Container,
      typename enable_if<HasInputIterator<Container>::value>::type * =
    nullptr>
  RollingHash &operator+=(const Container &rhs) {
    append(rhs.begin(), rhs.end());
    return *this;
  }

  inline void pop() {
    assert(size() > 0);
    hs.pop_back();
  }

  Hash prefix(int n) const {
    n = min(n, size());
    return Hash(hs[n], n);
  }

  Hash operator()(int i, int j) const {
    return Hash(hs[j + 1] - hs[i] * (cat.base->power(j - i + 1)), j - i + 1);
  }

  Hash suffix(int n) const {
    int sz = size();
    n = min(n, sz);
    return (*this)(sz - n, sz - 1);
  }

  pair<Hash, Hash> border(int n) const { return {prefix(n), suffix(n)}; }

  Hash substr(int i) const {
    i = min(i, size());
    return (*this)(i, size() - 1);
  }

  Hash substr(int i, int j) const { return (*this)(i, j); }

  Hash all() const { return Hash(hs.back(), size()); }

  friend int lcp(const type &lhs, const type &rhs) {
    int l = 0, r = min(lhs.size(), rhs.size());
    while (l < r) {
      int mid = (l + r) / 2;
      if (lhs.hs[mid + 1] != rhs.hs[mid + 1])
        r = mid;
      else
        l = mid + 1;
    }
    return l;
  }

  friend bool operator<(const type &lhs, const type &rhs) {
    int l = lcp(lhs, rhs);
    if (l == min(lhs.size(), rhs.size()))
      return lhs.size() < rhs.size();
    return lhs(l, l) < rhs(l, l);
  }
};

template <typename T, T... Mods> struct BidirectionalRollingHash {
  typedef RollingHash<T, Mods...> type;
  using Hash = typename type::Hash;
  using base_type = typename type::base_type;
  using Cat = typename type::Cat;

  type fwd, bwd;
  typename type::Cat cat;

  template <typename Container,
            typename
    enable_if<HasBidirectionalIterator<Container>::value>::type
                * = nullptr>
  BidirectionalRollingHash(const Container &container,
                           const shared_ptr<base_type> &base)
      : BidirectionalRollingHash<T, Mods...>(container.begin(),
    container.end(),
                                             base) {}

  template <typename Iterator,
            typename
    enable_if<IsBidirectionalIterator<Iterator>::value>::type
                * = nullptr>
  BidirectionalRollingHash(Iterator begin, Iterator end,
                           const shared_ptr<base_type> &base)
      : fwd(begin, end, base),
        bwd(make_reverse_iterator(end), make_reverse_iterator(begin), base),
        cat(base) {}

  inline Hash forward(int i, int j) const { return fwd(i, j); }

  inline Hash backward(int i, int j) const {
```

```
    int n = fwd.size();
    return bwd(n - j - 1, n - i - 1);
  }

  inline pair<Hash, Hash> operator()(int i, int j) const {
    return {forward(i, j), backward(i, j)};
  }
};

template <typename R> struct HashProvider {
  typedef R Roll;
  typedef typename R::base_type base_type;
  typedef typename R::Hash Hash;

  typename R::Cat cat;
  HashProvider() : cat(make_shared<base_type>()) {}
  explicit HashProvider(base_type base) : cat(make_shared<base_type>(base))
    {}

  template <class... Args> R operator()(Args... args) {
    return R(args..., cat.base);
  }
};

template <typename R> struct PowerHashProvider : HashProvider<R> {
  using typename HashProvider<R>::base_type;
  using HashProvider<R>::cat;

  PowerHashProvider() : PowerHashProvider<R>(base_type()) {}
  PowerHashProvider(base_type base) : HashProvider<R>(base) {
    cat.base->set_max_powers(DEFAULT_MAX_POWERS);
  }
};

template <int32_t... Mods> using Roll32 = RollingHash<int32_t, Mods...>;

template <int64_t... Mods> using Roll64 = RollingHash<int64_t, Mods...>;

template <int32_t... Mods>
using Biroll32 = BidirectionalRollingHash<int32_t, Mods...>;

template <int64_t... Mods>
using Biroll64 = BidirectionalRollingHash<int64_t, Mods...>;

using DefaultProvider = PowerHashProvider<Roll32<GOOD_MOD1, GOOD_MOD2>>;
using BiDefaultProvider = PowerHashProvider<Biroll32<GOOD_MOD1, GOOD_MOD2>>;
} // namespace hashing
} // namespace lib

#endif
```

1.32. Segtree

```
#ifndef _LIB_SEGTREE
#define _LIB_SEGTREE
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace seg {
struct LeafBuilder {
  template <typename Node> void operator()(Node &no, int i) const {}
  inline pair<int, int> range() const { return {0, 0}; }
  bool should_build() const { return true; }
};

struct EmptyLeafBuilder : LeafBuilder {
  int n;
  explicit EmptyLeafBuilder(int n) : n(n) {}
  inline pair<int, int> range() const { return {0, n - 1}; }
  bool should_build() const { return true; }
};

struct ImplicitBuilder : LeafBuilder {
  int L, R;
  explicit ImplicitBuilder(int L, int R) : L(L), R(R) {}
  inline pair<int, int> range() const { return {L, R}; }
  bool should_build() const { return false; }
};

// TODO: NOT IMPLEMENTED
template <typename DefaultNode>
struct ImplicitWithDefaultBuilder : LeafBuilder {
  int L, R;
  DefaultNode default_node;
  explicit ImplicitWithDefaultBuilder(int L, int R, DefaultNode def)
      : L(L), R(R), default_node(def) {}

  template <typename Node> inline void operator()(Node &no, int i) const {
    no = default_node;
  }

  inline pair<int, int> range() const { return {L, R}; }
  bool should_build() const { return false; }
};

template <typename RandomIterator> struct RangeLeafBuilder : LeafBuilder {
  RandomIterator begin, end;
  explicit RangeLeafBuilder(RandomIterator begin, RandomIterator end)
      : begin(begin), end(end) {}

  template <typename Node> inline void operator()(Node &no, int i) const {
    no = *(begin + i);
  }

  inline pair<int, int> range() const { return {0, end - begin - 1}; }
};

template <typename F> struct LambdaLeafBuilder : LeafBuilder {
  F f;
  pair<int, int> rng;
  explicit LambdaLeafBuilder(F f, pair<int, int> range)
      : f(f), rng(range) {}

  template <typename Node> inline void operator()(Node &no, int i) const {
    no = f(i);
  }

  inline pair<int, int> range() const { return rng; }
};

EmptyLeafBuilder make_builder(int n) { return EmptyLeafBuilder(n); }

template <typename RandomIterator>
RangeLeafBuilder<RandomIterator> make_builder(RandomIterator begin,
                                              RandomIterator end) {
  return RangeLeafBuilder<RandomIterator>(begin, end);
}

```

```
template <typename T>
RangeLeafBuilder<typename vector<T>::const_iterator>
make_builder(const vector<T> &v) {
  return RangeLeafBuilder<typename vector<T>::const_iterator>(v.begin(),
                                                              v.end());
}

template<typename T>
LambdaLeafBuilder<std::function<T(int)>>
make_builder(std::function<T(int)> f, pair<int, int> range) {
  return LambdaLeafBuilder<std::function<T(int)>>(f, range);
}

template <typename T> struct CombineFolder {
  inline T operator()() const { return T(); }

  template <typename Node> inline T operator()(const Node &no) const {
    return T(no);
  }

  inline T operator()(const T &a, const T &b) const { return a + b; }
};

template <typename T> struct EmptyFolder : CombineFolder<T> {
  using CombineFolder<T>::operator();

  template <typename Node> inline T operator()(const Node &no) const {
    return T();
  }
  inline T operator()(const T &a, const T &b) const { return T(); }
};

template <typename T> struct SumFolder : CombineFolder<T> {};

template <typename T> struct ProductFolder : CombineFolder<T> {
  using CombineFolder<T>::operator();
  inline T operator()() const { return T(1); }
  inline T operator()(const T &a, const T &b) const { return a * b; }
};

template <typename T> struct MaxFolder : CombineFolder<T> {
  using CombineFolder<T>::operator();
  inline T operator()() const { return numeric_limits<T>::min(); }
  inline T operator()(const T &a, const T &b) const { return max(a, b); }
};

template <typename T> struct MinFolder : CombineFolder<T> {
  using CombineFolder<T>::operator();
  inline T operator()() const { return numeric_limits<T>::max(); }
  inline T operator()(const T &a, const T &b) const { return min(a, b); }
};

template <typename T> struct SingleValueUpdater {
  T value;
  explicit SingleValueUpdater(T val) : value(val) {}
};

template <typename T> struct SetUpdater : SingleValueUpdater<T> {
  using SingleValueUpdater<T>::SingleValueUpdater;

  template <typename Node> inline void operator()(Node &no) const {
    no = this->value;
  }
};

template <typename T> struct AddUpdater : SingleValueUpdater<T> {
  using SingleValueUpdater<T>::SingleValueUpdater;

  template <typename Node> inline void operator()(Node &no) const {
    no += this->value;
  }
};

template <typename T> struct MultUpdater : SingleValueUpdater<T> {
  using SingleValueUpdater<T>::SingleValueUpdater;

  template <typename Node> inline void operator()(Node &no) const {
    no *= this->value;
  }
};

struct EmptyPushdown {
  template<typename Node>
  inline bool dirty(const Node& no) const { return false; }

  template<typename Node>
  inline void operator()(Node& no, int l, int r,
                         Node* ln, Node* rn) const {}
};

template<typename Node>
struct Active : public Node {
  bool active_ = false;
  Active& operator=(const Node& no) {
    Node::operator=(no);
    return *this;
  }
  bool is_active() const { return active_; }
  Active& activate() {
    active_ = true;
    return *this;
  }
  Active& deactivate() {
    active_ = false;
    return *this;
  }
  void toggle() {
    active_ = !active_;
  }
  friend Active<Node> operator+(const Active<Node>& a, const Active<Node>&
    b) {
    if(!a.active_) return b;
    else if(!b.active_) return a;
    Active<Node> res;
    res = Node(a) + Node(b);
    return res.activate();
  }
};

template <typename T>
struct ActiveUpdater {
  bool flag;

  ActiveUpdater(bool f) : flag(f) {}

  template <typename Node> inline void operator()(Node &no) const {
    no.active_ = flag;
  }
};
}  // namespace seg
```

```
}  // namespace lib

#endif
```

1.33. SegtreeBeats

```
#ifndef _LIB_SEGTREE_BEATS
#define _LIB_SEGTREE_BEATS
#include "Segtree.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace seg {
struct DefaultBreakCond {
  template <typename Node>
  inline bool operator()(const Node &no, int l, int r, int i, int j) const {
    return i > r || j < l;
  }
};

struct DefaultTagCond {
  template <typename Node>
  inline bool operator()(const Node &no, int l, int r, int i, int j) const {
    return i <= l && r <= j;
  }
};

template <typename T> struct SearchResult {
  bool found;
  int pos;
  T value;

  static SearchResult<T> not_found(T acc = T()) { return {false, 0, acc}; }
};

struct PrefixSearch;
struct SuffixSearch;

template <typename Direction> using IsSuffix = is_same<Direction,
    SuffixSearch>;

template <typename Node> struct InMemoryNodeManager {
  typedef int vnode;
  vector<Node> t;

  template <typename Builder> void initialize(const Builder &builder) {
    int L, R;
    tie(L, R) = builder.range();
    t = vector<Node>(4 * (R - L + 1));
  }

  inline bool has(vnode no) { return true; }
  inline vnode root() { return 1; }
  inline vnode new_root(vnode no) { return no; }
  inline vnode left(vnode no) { return no << 1; }
  inline vnode right(vnode no) { return no << 1 | 1; }
  inline Node &ref(vnode no) { return t[no]; }
  inline Node *ptr(vnode no) { return &t[no]; }
  inline Node value(vnode no) { return t[no]; }

  inline vnode persist(vnode no) { return no; }
  inline void ensure_left(vnode no) {}
  inline void ensure_right(vnode no) {}
};

template <
    typename Node, typename NodeManager, typename CombinerFn =
    EmptyFolder<int>,
    typename PushdownFn = EmptyPushdown, typename BreakCond =
    DefaultBreakCond,
    typename TagCond = DefaultTagCond>
struct SegtreeImpl {
  typedef typename NodeManager::vnode vnode;
  constexpr static bool has_lazy = !is_same<PushdownFn,
    EmptyPushdown>::value;
  constexpr static bool is_implicit =
      !is_same<NodeManager, InMemoryNodeManager<Node>>::value;

  CombinerFn combiner_fn;
  PushdownFn pushdown_fn;
  BreakCond break_cond;
  TagCond tag_cond;
  NodeManager manager;

  int L, R;

  template <typename Builder> explicit SegtreeImpl(const Builder &builder) {
    tie(L, R) = builder.range();
    assert(L <= R);
    manager.initialize(builder);
    if (builder.should_build())
      build(builder);
  }

  inline vnode root() { return manager.root(); }
  inline int split(int l, int r) { return l + (r - l) / 2; }

  template <typename Builder>
  vnode build(const Builder &builder, vnode no, int l, int r) {
    no = manager.persist(no);
    if (l == r) {
      builder(manager.ref(no), l);
    } else {
      int mid = split(l, r);
      build(builder, manager.left(no), l, mid);
      build(builder, manager.right(no), mid + 1, r);
      manager.ref(no) = combiner_fn(manager.value(manager.left(no)),
                                    manager.value(manager.right(no)));
    }
    return no;
  }

  template <typename Builder> vnode build(const Builder &builder) {
    return manager.new_root(build(builder, root(), L, R));
  }

  inline int size() const { return R - L + 1; }

  void push(vnode no, int l, int r) {
    if(!has_lazy) return;
    if (!pushdown_fn.dirty(manager.ref(no)))
      return;
    if(l == r) {
      pushdown_fn(manager.ref(no), l, r, nullptr, nullptr);
      return;
    }
    manager.ensure_left(no);
    manager.ensure_right(no);
```

```
    vnode lno = manager.persist(manager.left(no));
    vnode rno = manager.persist(manager.right(no));
    pushdown_fn(manager.ref(no), l, r, manager.ptr(lno), manager.ptr(rno));
  }

  template <typename T, typename Folder>
  T query(vnode no, int l, int r, int i, int j, const Folder &folder) {
    if (!manager.has(no))
      return folder();
    if (j < l || i > r)
      return folder();
    push(no, l, r);
    if (i <= l && r <= j)
      return folder(manager.ref(no));
    int mid = split(l, r);
    return folder(query<T>(manager.left(no), l, mid, i, j, folder),
                  query<T>(manager.right(no), mid + 1, r, i, j, folder));
  }

  template <typename T, typename Folder>
  inline T query(vnode root, int i, int j, const Folder &folder) {
    return query<T>(root, L, R, i, j, folder);
  }

  template <typename T, typename Folder>
  inline T query(int i, int j, const Folder &folder) {
    return query<T>(root(), i, j, folder);
  }

  template <typename Updater>
  vnode update(vnode no, int l, int r, int i, int j, const Updater &updater) {
    push(no, l, r);
    if (break_cond(manager.ref(no), l, r, i, j)) {
      return no;
    }
    no = manager.persist(no);
    if (tag_cond(manager.ref(no), l, r, i, j)) {
      updater(manager.ref(no));
      push(no, l, r);
      return no;
    }
    int mid = split(l, r);
    update(manager.left(no), l, mid, i, j, updater);
    update(manager.right(no), mid + 1, r, i, j, updater);
    manager.ref(no) = combiner_fn(manager.value(manager.left(no)),
                                  manager.value(manager.right(no)));
    return no;
  }

  template <typename Updater>
  inline vnode update(vnode root, int i, int j, const Updater &updater) {
    return manager.new_root(update(root, L, R, i, j, updater));
  }

  template <typename Updater>
  inline vnode update(int i, int j, const Updater &updater) {
    return update(root(), i, j, updater);
  }

  template <typename Beater, typename U = NodeManager,
            typename enable_if<
                is_same<U, InMemoryNodeManager<Node>>::value>::type * = nullptr>
  void beat(vnode no, int l, int r, int i, int j, const Beater &beater) {
    push(no, l, r);
    if (break_cond(manager.ref(no), l, r, i, j) ||
        beater.stop(manager.ref(no), l, r, i, j)) {
      return;
    }
    if (tag_cond(manager.ref(no), l, r, i, j) &&
        beater.tag(manager.ref(no), l, r, i, j)) {
      beater(manager.ref(no));
      push(no, l, r);
      return;
    }
    int mid = split(l, r);
    beat(manager.left(no), l, mid, i, j, beater);
    beat(manager.right(no), mid + 1, r, i, j, beater);
    manager.ref(no) = combiner_fn(manager.value(manager.left(no)),
                                  manager.value(manager.right(no)));
  }

  template <typename Beater>
  inline void beat(int i, int j, const Beater &beater) {
    beat(root(), L, R, i, j, beater);
  }

  template <typename T, typename Direction, typename Folder, typename Checker>
  SearchResult<T> bsearch_first(vnode no, int l, int r, int i, int j,
                                const Folder &folder, const Checker &checker,
                                T acc) {
    if (manager.has(no))
      push(no, l, r);
    if (j < l || i > r)
      return SearchResult<T>::not_found(folder());
    if (!manager.has(no)) {
      auto value = folder(acc, folder());
      if (checker(value))
        return {true, IsSuffix<Direction>::value ? r : l, value};
      else
        return SearchResult<T>::not_found(folder());
    }
    int mid = split(l, r);
    if (i <= l && r <= j) {
      auto b_value = folder(acc, manager.value(no));
      if (!checker(b_value))
        return SearchResult<T>::not_found(manager.value(no));
      if (l == r)
        return {true, l, b_value};
    }
    if (!IsSuffix<Direction>::value) {
      auto res_left = bsearch_first<T, Direction>(manager.left(no), l, mid, i,
                                                  j, folder, checker, acc);
      if (res_left.found)
        return res_left;
      return bsearch_first<T, Direction>(manager.right(no), mid + 1, r, i, j,
                                         folder, checker,
                                         folder(acc, res_left.value));
    } else {
      auto res_right = bsearch_first<T, Direction>(
          manager.right(no), mid + 1, r, i, j, folder, checker, acc);
      if (res_right.found)
        return res_right;
      return bsearch_first<T, Direction>(manager.left(no), l, mid, i, j, folder,
                                         checker, folder(acc, res_right.value));
```

```
    }
  }

  template <typename T, typename Direction, typename Folder, typename
    Checker>
  inline SearchResult<T> bsearch_first(vnode root, int i, int j,
                                       const Folder &folder,
                                       const Checker &checker) {
    auto res = bsearch_first<T, Direction>(root, L, R, i, j, folder, checker,
                                           folder());
    if (!res.found)
      res.pos = IsSuffix<Direction>::value ? i - 1 : j + 1;
    return res;
  }

  template <typename T, typename Direction, typename Folder, typename
    Checker>
  inline SearchResult<T> bsearch_first(int i, int j, const Folder &folder,
                                       const Checker &checker) {
    return bsearch_first<T, Direction>(root(), i, j, folder, checker);
  }

  template <typename T, typename Direction, typename Folder, typename
    Checker>
  inline SearchResult<T> bsearch_last(vnode root, int i, int j,
                                      const Folder &folder,
                                      const Checker &checker) {
    auto res = bsearch_first<T, Direction>(
        root, i, j, folder, [&checker](T x) { return !checker(x); });
    if (!IsSuffix<Direction>::value) {
      if (res.pos == i)
        res.found = false;
      res.pos--;
    } else {
      if (res.pos == j)
        res.found = false;
      res.pos++;
    }
    return res;
  }

  template <typename T, typename Direction, typename Folder, typename
    Checker>
  inline SearchResult<T> bsearch_last(int i, int j, const Folder &folder,
                                      const Checker &checker) {
    return bsearch_last<T, Direction>(root(), i, j, folder, checker);
  }
};

template <typename Node, typename CombinerFn = EmptyFolder<int>,
          typename PushdownFn = EmptyPushdown,
          typename BreakCond = DefaultBreakCond,
          typename TagCond = DefaultTagCond>
struct SegtreeBeats : SegtreeImpl<Node, InMemoryNodeManager<Node>,
    CombinerFn,
                                  PushdownFn, BreakCond, TagCond> {

  template <typename Builder>
  explicit SegtreeBeats(const Builder &builder)
      : SegtreeImpl<Node, InMemoryNodeManager<Node>, CombinerFn, PushdownFn,
                    BreakCond, TagCond>(builder) {}
};

template <typename Node> using Explicit = InMemoryNodeManager<Node>;
} // namespace seg
} // namespace lib

#endif
```

1.34. SegtreeFast

```
#ifndef _LIB_SEGTREE_FAST
#define _LIB_SEGTREE_FAST
#include "Segtree.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace seg {
template <typename Node, typename CombinerFn> struct SegtreeFastBase {
  const static int MULTIPLIER = 2;

  CombinerFn combiner_fn;

  vector<Node> t;
  int L, n;

  SegtreeFastBase() {}
  template <typename Builder> explicit SegtreeFastBase(const Builder
    &builder) {
    pair<int, int> range = builder.range();
    L = range.first;
    n = range.second - range.first + 1;
    assert(n > 0);
    t = vector<Node>(n * MULTIPLIER);
    build(builder);
  }

  template <typename Builder> void build(const Builder &builder) {
    for (int i = n; i < 2 * n; i++)
      builder(t[i], L + i - n);
    for (int i = n - 1; i > 0; i--)
      t[i] = combiner_fn(t[i << 1], t[i << 1 | 1]);
  }

  template <typename Rebuilder> void rebuild(const Rebuilder &rebuilder) {
    for (int i = n; i < 2 * n; i++)
      rebuilder(t[i]);
    for (int i = n - 1; i > 0; i--)
      rebuilder(t[i], t[i << 1], t[i << 1 | 1]);
  }
};

template <typename Node, typename CombinerFn>
struct SegtreeFast : SegtreeFastBase<Node, CombinerFn> {
  typedef SegtreeFastBase<Node, CombinerFn> Base;
  using Base::combiner_fn;
  using Base::L;
  using Base::n;
  using Base::SegtreeFastBase;
  using Base::t;

  template <typename Updater>
  void update_element(int i, const Updater &updater) {
    i -= L;
    assert(i >= 0);
    for (updater(t[i += n]); i /= 2;)
      t[i] = combiner_fn(t[i << 1], t[i << 1 | 1]);
  }
```

```cpp

  template <typename T, typename Folder>
  T query(int i, int j, const Folder &folder) {
    // input is [i, j]
    i -= L, j -= L;
    assert(i >= 0 && j >= 0);
    i += n, j += n;
    if (i == j)
      return folder(t[i]);
    T resl = folder(t[i]), resr = folder(t[j]);

    // now it is [i, j)
    i++;
    while (i < j) {
      if (i & 1)
        resl = folder(resl, folder(t[i++]));
      if (j & 1)
        resr = folder(folder(t[--j]), resr);
      i /= 2, j /= 2;
    }

    return folder(resl, resr);
  }
};

template <typename Node>
struct SegtreeFastSplash : SegtreeFastBase<Node, EmptyFolder<Node>> {
  typedef SegtreeFastBase<Node, EmptyFolder<Node>> Base;
  using Base::L;
  using Base::n;
  using Base::SegtreeFastBase;
  using Base::t;

  template <typename T, typename Folder>
  T query_element(int i, const Folder &folder) {
    i -= L;
    assert(i >= 0);
    T res = folder(t[i += n]);
    while (i /= 2) {
      res = folder(folder(t[i]), res);
    }
    return res;
  }

  template <typename Updater>
  void splash(int i, int j, const Updater &updater) {
    // input is [i, j]
    i -= L, j -= L;
    assert(i >= 0 && j >= 0);
    // now it is [i, j)
    i += n, j += n + 1;

    while (i < j) {
      if (i & 1)
        updater(t[i++]);
      if (j & 1)
        updater(t[--j]);
      i /= 2, j /= 2;
    }
  }
};

} // namespace seg
} // namespace lib

#endif
```

1.35. SegtreeHLD

```cpp
#ifndef _LIB_RANGE_HLD
#define _LIB_RANGE_HLD
#include "HLD.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace graph {
namespace range {
template <typename Builder, typename H> struct BuilderWrapper {
  H *hld;
  Builder builder;

  explicit BuilderWrapper(H *hld, const Builder &builder)
      : hld(hld), builder(builder) {}

  template <typename Node> void operator()(Node &no, int i) const {
    builder(no, hld->rin[i]);
  }

  pair<int, int> range() const { return {0, hld->size() - 1}; }
};

template <typename Builder, typename H>
struct RebuilderWrapper : BuilderWrapper<Builder, H> {
  using BuilderWrapper<Builder, H>::BuilderWrapper;
  using BuilderWrapper<Builder, H>::builder;
  template <typename Node>
  void operator()(Node &no, const Node &left, const Node &right) const {
    builder(no, left, right);
  }
};

template <typename S, typename T, typename Folder> struct QueryIssuer {
  S &seg;
  const Folder &folder;
  QueryIssuer(S &seg, const Folder &folder) : seg(seg), folder(folder) {}
  T operator()(int i) const { seg.template query_element<T>(i, folder); }
  T operator()(int i, int j) const {
    return seg.template query<T>(i, j, folder);
  }
};

template <typename S, typename T, typename Folder> struct QueryLifter {
  QueryIssuer<S, T, Folder> issuer;
  T res;
  QueryLifter(S &seg, const Folder &folder)
      : issuer(seg, folder), res(folder()) {}
  void operator()(int i, int j, bool) {
    res = issuer.folder(res, issuer(i, j));
  }
  T result() const { return res; }
};

template <typename S, typename T, typename Folder>
struct OrderedQueryLifter : QueryLifter<S, T, Folder> {
  using QueryLifter<S, T, Folder>::issuer;
  T resl, resr;

  OrderedQueryLifter(S &seg, const Folder &folder)
```

```
      : QueryLifter<S, T, Folder>(seg, folder), resl(folder()),
    resr(folder()) {
  }

  void operator()(int i, int j, bool right) {
    if (right)
      resr = issuer.folder(issuer(i, j), resr);
    else
      resl = issuer.folder(resl, issuer(i, j));
  }
  T result() const { return issuer.folder(resl, resr); }
};

template <typename S, typename Updater> struct UpdateIssuer {
  S &seg;
  const Updater &updater;
  UpdateIssuer(S &seg, const Updater &updater) : seg(seg), updater(updater) {}
  void operator()(int i, int j) { seg.update(i, j, updater); }
  void operator()(int i, int j, bool) { (*this)(i, j); }
};

template <typename S, typename Updater> struct SplashIssuer {
  S &seg;
  const Updater &updater;
  SplashIssuer(S &seg, const Updater &updater) : seg(seg), updater(updater) {}
  void operator()(int i, int j) { seg.splash(i, j, updater); }
  void operator()(int i, int j, bool) { (*this)(i, j); }
};

template <typename S, typename Beater> struct BeatIssuer {
  S &seg;
  const Beater &beater;
  BeatIssuer(S &seg, const Beater &beater) : seg(seg), beater(beater) {}
  void operator()(int i, int j) { seg.beat(i, j, beater); }
  void operator()(int i, int j, bool) { (*this)(i, j); }
};
} // namespace range

template <typename S, typename G> struct RangeHLD : HLD<G> {
  typedef seg::EmptyLeafBuilder empty_builder;

  template <typename Builder>
  using builder_wrapper = range::BuilderWrapper<Builder, HLD<G>>;
  template <typename Rebuilder>
  using rebuilder_wrapper = range::RebuilderWrapper<Rebuilder, HLD<G>>;

  S seg;

  explicit RangeHLD(const G &graph)
      : HLD<G>(graph),
        seg(builder_wrapper<empty_builder>(this,
    empty_builder(this->size()))) {
  }

  template <typename Builder>
  RangeHLD(const G &graph, const Builder &builder)
      : HLD<G>(graph), seg(builder_wrapper<Builder>(this, builder)) {}

  template <typename Builder> void build(const Builder &builder) {
    seg.build(builder_wrapper<Builder>(builder));
  }

  template <typename Rebuilder> void rebuild(const Rebuilder &rebuilder) {
    seg.rebuild(rebuilder_wrapper<Rebuilder>(rebuilder));
  }

  template <typename T, typename Folder>
  inline T query_subtree(int u, const Folder &folder) {
    return this->template query_on_subtree<T>(
        u, range::QueryIssuer<S, T, Folder>(seg, folder));
  }

  template <typename T, typename Folder>
  inline T query_subtree_edges(int u, const Folder &folder) {
    return this->template query_on_subtree_edges(
        u, range::QueryIssuer<S, T, Folder>(seg, folder));
  }

  template <typename T, typename Folder>
  inline T query_vertex(int u, const Folder &folder) {
    return this->template query_on_vertex(
        u, range::QueryIssuer<S, T, Folder>(seg, folder));
  }

  template <typename T, typename Folder>
  T query_path(int u, int v, const Folder &folder) {
    auto lifter = range::OrderedQueryLifter<S, T, Folder>(seg, folder);
    this->template operate_on_path(u, v, lifter);
    return lifter.result();
  }

  template <typename T, typename Folder>
  T query_path_edges(int u, int v, const Folder &folder) {
    auto lifter = range::OrderedQueryLifter<S, T, Folder>(seg, folder);
    this->template operate_on_path_edges(u, v, lifter);
    return lifter.result();
  }

  template <typename Updater>
  inline void update_subtree(int u, const Updater &updater) {
    auto issuer = range::UpdateIssuer<S, Updater>(seg, updater);
    this->template operate_on_subtree(u, issuer);
  }

  template <typename Updater>
  inline void update_subtree_edges(int u, const Updater &updater) {
    auto issuer = range::UpdateIssuer<S, Updater>(seg, updater);
    this->template operate_on_subtree_edges(u, issuer);
  }

  template <typename Updater>
  inline void update_path(int u, int v, const Updater &updater) {
    auto issuer = range::UpdateIssuer<S, Updater>(seg, updater);
    this->template operate_on_path(u, v, issuer);
  }

  template <typename Updater>
  inline void update_path_edges(int u, int v, const Updater &updater) {
    auto issuer = range::UpdateIssuer<S, Updater>(seg, updater);
    this->template operate_on_path_edges(u, v, issuer);
  }

  template <typename Beater>
  inline void beat_subtree(int u, const Beater &beater) {
    auto issuer = range::BeatIssuer<S, Beater>(seg, beater);
    this->template operate_on_subtree(u, issuer);
  }

```

```
  template <typename Beater>
  inline void beat_subtree_edges(int u, const Beater &beater) {
    auto issuer = range::BeatIssuer<S, Beater>(seg, beater);
    this->template operate_on_subtree_edges(u, issuer);
  }

  template <typename Beater>
  inline void beat_path(int u, int v, const Beater &beater) {
    auto issuer = range::BeatIssuer<S, Beater>(seg, beater);
    this->template operate_on_path(u, v, issuer);
  }

  template <typename Beater>
  inline void beat_path_edges(int u, int v, const Beater &beater) {
    auto issuer = range::BeatIssuer<S, Beater>(seg, beater);
    this->template operate_on_path_edges(u, v, issuer);
  }

  // TODO: FIX THOSE
  template <typename Updater>
  inline void update_element(int idx, const Updater &updater) {
    seg.update_element(idx, updater);
  }

  template <typename Updater>
  inline void splash(int i, int j, const Updater &updater) {
    seg.splash(i, j, updater);
  }
};

template <typename S, typename G>
RangeHLD<S, G> make_range_hld(const G &graph) {
  return RangeHLD<S, G>(graph);
}

template <typename S, typename G, typename Builder>
RangeHLD<S, G> make_range_hld(const G &graph, const Builder &builder) {
  return RangeHLD<S, G>(graph, builder);
}

} // namespace graph
} // namespace lib

#endif
```

1.36. SegtreeImplicit

```
#ifndef _LIB_SEGTREE_IMPLICIT
#define _LIB_SEGTREE_IMPLICIT
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace seg {

template <typename Node> struct ImplicitNodeManager {
  struct NodeWrapper {
    Node no;
    NodeWrapper *left = nullptr;
    NodeWrapper *right = nullptr;
  };

  struct VirtualNode {
    NodeWrapper *cur = nullptr, **edge = nullptr;
  };

  typedef VirtualNode vnode;

  vnode r = {new NodeWrapper()};

  template <typename Builder> void initialize(const Builder &builder) {}

  inline bool has(vnode no) const { return no.cur; }
  inline vnode root() { return r; }
  inline vnode new_root(vnode no) { return r = no; }
  inline vnode left(vnode no) { return {no.cur->left, &(no.cur->left)}; }
  inline vnode right(vnode no) { return {no.cur->right, &(no.cur->right)}; }
  inline Node &ref(vnode no) { return no.cur->no; }
  inline Node *ptr(vnode no) { return &(no.cur->no); }
  inline Node value(vnode no) { return no.cur->no; }

  inline vnode persist(vnode no) {
    if (no.cur)
      return no;
    vnode res = no;
    res.cur = *res.edge = new NodeWrapper();
    return res;
  }
  inline void ensure_left(vnode no) {
    if (!no.cur->left)
      no.cur->left = new NodeWrapper();
  }
  inline void ensure_right(vnode no) {
    if (!no.cur->right)
      no.cur->right = new NodeWrapper();
  }
};

template <typename Node> using Implicit = ImplicitNodeManager<Node>;
} // namespace seg
} // namespace lib

#endif
```

1.37. SegtreeLazy

```
#ifndef _LIB_SEGTREE_LAZY
#define _LIB_SEGTREE_LAZY
#include "SegtreeBeats.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace seg {
template <typename Node, typename CombinerFn, typename PushdownFn,
          typename NodeManager = Explicit<Node>>
struct SegtreeLazy : SegtreeImpl<Node, NodeManager, CombinerFn, PushdownFn> {
  typedef SegtreeImpl<Node, NodeManager, CombinerFn, PushdownFn> Base;
  using Base::SegtreeImpl;
  using typename Base::vnode;
};
} // namespace seg
} // namespace lib

#endif
```

1.38. SegtreeNormal

```
#ifndef _LIB_SEGTREE_NORMAL
#define _LIB_SEGTREE_NORMAL
#include "SegtreeBeats.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace seg {
template <typename Node, typename CombinerFn,
          typename NodeManager = Explicit<Node>>
struct SegtreeNormal : SegtreeImpl<Node, NodeManager, CombinerFn> {
  typedef SegtreeImpl<Node, NodeManager, CombinerFn> Base;
  using Base::combiner_fn;
  using Base::L;
  using Base::manager;
  using Base::R;
  using Base::SegtreeImpl;
  using Base::split;
  using typename Base::vnode;

  template <typename Updater>
  vnode update_element(vnode no, int l, int r, int idx,
                       const Updater &updater) {
    no = manager.persist(no);
    if (l == r)
      updater(manager.ref(no));
    else {
      int mid = split(l, r);
      if (idx <= mid)
        update_element(manager.left(no), l, mid, idx, updater);
      else
        update_element(manager.right(no), mid + 1, r, idx, updater);
      auto left_no = manager.left(no);
      auto right_no = manager.right(no);
      auto left_value =
          manager.has(left_no) ? manager.value(left_no) : combiner_fn();
      auto right_value =
          manager.has(right_no) ? manager.value(right_no) : combiner_fn();
      manager.ref(no) = combiner_fn(left_value, right_value);
    }
    return no;
  }

  template <typename Updater>
  inline vnode update_element(vnode root, int idx, const Updater &updater) {
    return manager.new_root(update_element(root, L, R, idx, updater));
  }

  template <typename Updater>
  inline vnode update_element(int idx, const Updater &updater) {
    return update_element(this->root(), idx, updater);
  }
};
} // namespace seg
} // namespace lib

#endif
```

1.39. SegtreePersistent

```
#ifndef _LIB_SEGTREE_PERSISTENT
#define _LIB_SEGTREE_PERSISTENT
#include "SegtreeImplicit.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace seg {

template <typename Node>
struct PersistentNodeManager : ImplicitNodeManager<Node> {
  using typename ImplicitNodeManager<Node>::vnode;
  using typename ImplicitNodeManager<Node>::NodeWrapper;

  inline vnode persist(vnode no) {
    vnode res = no;
    res.cur = no.cur ? new NodeWrapper(*no.cur) : new NodeWrapper();
    if (res.edge)
      *res.edge = res.cur;
    return res;
  }
};

template <typename Node> using Persistent = PersistentNodeManager<Node>;
} // namespace seg
} // namespace lib

#endif
```

1.40. SegtreeSplash

```
#ifndef _LIB_SEGTREE_SPLASH
#define _LIB_SEGTREE_SPLASH
#include "SegtreeBeats.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace seg {
template <typename Node, typename NodeManager = Explicit<Node>>
struct SegtreeSplash : SegtreeBeats<Node, NodeManager, EmptyFolder<void>> {
  typedef SegtreeBeats<Node, NodeManager, EmptyFolder<void>> Base;
  using Base::L;
  using Base::manager;
  using Base::R;
  using Base::SegtreeBeats;
  using Base::split;
  using typename Base::vnode;

  template <typename T, typename Folder>
  T query_element(vnode no, int l, int r, int idx, const Folder &folder) {
    if (!manager.has(no))
      return folder();
    T res = folder(manager.ref(no));
    if (l != r) {
      int mid = split(l, r);
      if (idx <= mid)
        res = folder(res,
                     query_element<T>(manager.left(no), l, mid, idx,
      folder));
      else
        res = folder(
            res, query_element<T>(manager.right(no), mid + 1, r, idx,
      folder));
    }
    return res;
  }
```

```

  template <typename T, typename Folder>
  inline T query_element(vnode root, int idx, const Folder &folder) {
    return query_element<T>(root, L, R, idx, folder);
  }

  template <typename T, typename Folder>
  inline T query_element(int idx, const Folder &folder) {
    return query_element<T>(this->root(), idx, folder);
  }

  template <typename Updater>
  vnode splash(vnode no, int l, int r, int i, int j, const Updater &updater)
    {
    no = manager.persist(no);
    if (tag_cond(manager.ref(no), l, r, i, j)) {
      updater(manager.ref(no));
      return no;
    }
    int mid = split(l, r);
    if (j <= mid) {
      manager.ensure_left(no);
      splash(manager.left(no), l, mid, i, j, updater);
    } else if (i > mid) {
      manager.ensure_right(no);
      splash(manager.right(no), mid + 1, r, i, j, updater);
    } else {
      manager.ensure_left(no), manager.ensure_right(no);
      splash(manager.left(no), l, mid, i, j, updater);
      splash(manager.right(no), mid + 1, r, i, j, updater);
    }
    return no;
  }

  template <typename Updater>
  inline vnode splash(vnode root, int i, int j, const Updater &updater) {
    return manager.new_root(splash(root, L, R, i, j, updater));
  }

  template <typename Updater>
  inline vnode splash(int i, int j, const Updater &updater) {
    return splash(this->root(), i, j, updater);
  }
};
} // namespace seg
} // namespace lib

#endif
```

1.41. Simplex

```
#ifndef _LIB_SIMPLEX
#define _LIB_SIMPLEX
#include <bits/stdc++.h>

namespace lib {
using namespace std;
template <typename DOUBLE> struct LPSolver {
  typedef vector<DOUBLE> VD;
  typedef vector<VD> VVD;
  typedef vector<int> VI;

  constexpr static DOUBLE EPS = 1e-9;

  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c)
      : m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
    for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
        D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) {
      B[i] = n + i;
      D[i][n] = -1;
      D[i][n + 1] = b[i];
    }
    for (int j = 0; j < n; j++) {
      N[j] = j;
      D[m][j] = -c[j];
    }
    N[n] = -1;
    D[m + 1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m + 2; i++)
      if (i != r)
        for (int j = 0; j < n + 2; j++)
          if (j != s)
            D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n + 2; j++)
      if (j != s)
        D[r][j] /= D[r][s];
    for (int i = 0; i < m + 2; i++)
      if (i != r)
        D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1)
          continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] <
  N[s])
          s = j;
      }
      if (D[x][s] > -EPS)
        return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS)
          continue;
        if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
            (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] <
  B[r])
          r = i;
      }
      if (r == -1)
        return false;
      Pivot(r, s);
    }
  }
```

```cpp

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++)
      if (D[i][n + 1] < D[r][n + 1])
        r = i;
    if (D[r][n + 1] < -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
        return -numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++)
        if (B[i] == -1) {
          int s = -1;
          for (int j = 0; j <= n; j++)
            if (s == -1 || D[i][j] < D[i][s] ||
                D[i][j] == D[i][s] && N[j] < N[s])
              s = j;
          Pivot(i, s);
        }
    }
    if (!Simplex(2))
      return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++)
      if (B[i] < n)
        x[B[i]] = D[i][n + 1];
    return D[m][n + 1];
  }
};
} // namespace lib

#endif
```

1.42. Subset

```cpp
#ifndef _LIB_SUBSET
#define _LIB_SUBSET
#include <bits/stdc++.h>

namespace lib {
using namespace std;
// Source: https://github.com/NyaanNyaan/library/tree/master/set-function

template <typename T>
void superset_zeta_transform(vector<T>& f) {
  int n = f.size();
  assert((n & (n - 1)) == 0);
  for (int i = 1; i < n; i <<= 1) {
    for (int j = 0; j < n; j++) {
      if ((j & i) == 0) {
        f[j] += f[j | i];
      }
    }
  }
}

template <typename T>
void superset_mobius_transform(vector<T>& f) {
  int n = f.size();
  assert((n & (n - 1)) == 0);
  for (int i = 1; i < n; i <<= 1) {
    for (int j = 0; j < n; j++) {
      if ((j & i) == 0) {
        f[j] -= f[j | i];
      }
    }
  }
}

template <typename T>
void subset_zeta_transform(vector<T>& f) {
  int n = f.size();
  assert((n & (n - 1)) == 0);
  for (int i = 1; i < n; i <<= 1) {
    for (int j = 0; j < n; j++) {
      if ((j & i) == 0) {
        f[j | i] += f[j];
      }
    }
  }
}

template <typename T>
void subset_mobius_transform(vector<T>& f) {
  int n = f.size();
  assert((n & (n - 1)) == 0);
  for (int i = 1; i < n; i <<= 1) {
    for (int j = 0; j < n; j++) {
      if ((j & i) == 0) {
        f[j | i] -= f[j];
      }
    }
  }
}

template <typename T>
vector<T> or_convolution(vector<T> a, vector<T> b) {
  assert(a.size() == b.size());
  subset_zeta_transform(a);
  subset_zeta_transform(b);
  for (int i = 0; i < (int)a.size(); i++) a[i] *= b[i];
  subset_mobius_transform(a);
  return a;
}

template <typename T>
vector<T> and_convolution(vector<T> a, vector<T> b) {
  assert(a.size() == b.size());
  superset_zeta_transform(a);
  superset_zeta_transform(b);
  for (int i = 0; i < (int)a.size(); i++) a[i] *= b[i];
  superset_mobius_transform(a);
  return a;
}

template<typename T>
vector<vector<T>> ranked_zeta_transform(const vector<T>& f) {
  int N = f.size();
  assert((N & (N-1)) == 0);
  int R = __builtin_ctz(N);
  vector<vector<T>> F(R + 1, vector<T>(N));
  for(int i = 0; i < N; i++)
    F[__builtin_popcount(i)][i] = f[i];
  for(int i = 0; i <= R; i++)
    subset_zeta_transform(F[i]);
  return F;
}

template<typename T>
```

```cpp
vector<T> subset_convolution(const vector<T>& a, const vector<T>& b, int offset = 0) {
  int N = a.size();
  assert(N == b.size());
  assert((N & (N-1)) == 0);
  int R = __builtin_ctz(N);

  auto A = ranked_zeta_transform(a), B = ranked_zeta_transform(b);
  auto C = vector<vector<T>>(R + 1, vector<T>(N));

  for(int m = 0; m < N; m++) {
    for(int i = 0; i <= R; i++) {
      for(int j = offset; j <= i; j++) {
        C[i][m] += A[j][m] * B[i + offset - j][m];
      }
    }
  }

  for(int i = 0; i <= R; i++)
    subset_mobius_transform(C[i]);
  vector<T> res(N);
  for(int i = 0; i < N; i++)
    res[i] = C[__builtin_popcount(i)][i];
  return res;
}
} // namespace lib

#endif
```

1.43. SuffixArray

```cpp
#ifndef _LIB_SUFFIX_ARRAY
#define _LIB_SUFFIX_ARRAY
#include <bits/stdc++.h>

namespace lib {
using namespace std;
template<typename C>
struct SuffixArray {
  int n, block;
  vector<C> s;
  vector<int> sa, rnk, tmp, aux, lcp_;
  vector<vector<int>> T;

  void init(int h) {
    h = max(h, n);
    sa = vector<int>(h+3), rnk = vector<int>(h+3),
      tmp = vector<int>(h+3), aux = vector<int>(h+3),
      lcp_ = vector<int>(h+3);
    T = vector<vector<int>>(n + 3, vector<int>(__lg(n) + 1));
  }

  SuffixArray(vector<C> s_) : s(s_), n(s_.size()) { build(); }
  SuffixArray(string s_) {
    s = vector<C>(s_.size());
    n = s_.size();
    for(int i = 0; i < n; i++) s[i] = s_[i];
    build();
  }

  bool suffix_cmp(int i, int j) {
    if (rnk[i] != rnk[j]) return rnk[i] < rnk[j];
    i += block, j += block;
    if (i >= n) i -= n;
    if (j >= n) j -= n;
    return rnk[i] < rnk[j];
  }
  void suffix_sort(int h) {
    for (int i = 0; i < n; i++) {
      aux[i] = sa[i] - block;
      if (aux[i] < 0) aux[i] += n;
    }
    for (int i = 0; i < h; i++) tmp[i] = 0;
    for (int i = 0; i < n; i++) tmp[rnk[aux[i]]]++;
    for (int i = 0; i < h - 1; i++) tmp[i + 1] += tmp[i];
    for (int i = n - 1; i >= 0; i--) sa[--tmp[rnk[aux[i]]]] = aux[i];
    tmp[0] = 0;
    for (int i = 0; i < n - 1; i++) tmp[i + 1] = tmp[i] + suffix_cmp(sa[i], sa[i + 1]);
    for (int i = 0; i < n; i++) rnk[sa[i]] = tmp[i];
  }
  void build() {
    n++; // consider additional '\0' character
    s.push_back(0);
    int h = (int)(*max_element(s.begin(), s.end())) + 1;
    init(h);
    for (int i = 0; i < n; i++) sa[i] = i, rnk[i] = s[i], tmp[i] = 0;
    block = 0;
    suffix_sort(h);
    for (block = 1; tmp[n - 1] != n - 1; block *= 2) suffix_sort(tmp[n - 1] + 1);
    n--;
    sa.erase(sa.begin());
    build_lcp_();
  }

  void build_lcp_() {
    for (int i = 0; i < n; i++) rnk[sa[i]] = i, lcp_[i] = 0;
    int last = 0; // last lcp_
    for (int i = 0; i < n; i++, last = max(lcp_[rnk[i - 1]] - 1, 0)) {
      if (rnk[i] == n - 1) continue;
      int j = sa[rnk[i] + 1]; // next suffix pos in suffix array
      while (i + last < n && j + last < n && s[i + last] == s[j + last]) last++;
      lcp_[rnk[i]] = last;
    }

    for(int i = 0; i < n; i++)
      T[i][0] = lcp_[i];
    for(int j = 1; j < 20; j++){
      for(int i = 0; i+(1<<j) <= n; i++){
        T[i][j] = min(T[i][j-1], T[i+(1<<(j-1))][j-1]);
      }
    }
  }

  int lcp(int i, int j){
    if(i > j) swap(i,j);
    if(i == j) return n-sa[i];
    if(j == n) return 0;

    j--;
    int k = __builtin_clz(1) - __builtin_clz(j-i+1);
    return min(T[i][k], T[j-(1<<k)+1][k]);
  }

  int operator[](int i) const {
    return sa[i];
  }
```

```cpp

  int length(int i) const {
    return n - sa[i];
  }

  int lcp(int i) const {
    return lcp_[i];
  }

  pair<int,int> range(int i, int sz){
    pair<int, int> res;
    {
      int l = 0, r = i+1;
      while(l < r){
        int mid = (l+r)/2;
        if(lcp(mid, i) >= sz) r = mid;
        else l = mid+1;
      }
      res.first = l;
    }
    {
      int l = i, r = n-1;
      while(l < r){
        int mid = (l+r+1)/2;
        if(lcp(mid, i) >= sz) l = mid;
        else r = mid-1;
      }
      res.second = l;
    }
    return res;
  }

  pair<int, int> range(int i) {
    return range(i, length(i));
  }
};
} // namespace lib

#endif
```

1.44. Symbolic

```cpp
#ifndef _LIB_SYMBOLIC
#define _LIB_SYMBOLIC
#include <bits/stdc++.h>

namespace lib {
using namespace std;
static int g_VAR_PTR = 0;

enum Operation { variable, literal, sum };

template <typename T> struct Variable;

template <typename T> struct BasicExp {
  using node = shared_ptr<BasicExp<T>>;
  using variable = Variable<T>;

  T coef = 1;
  Operation op;
  vector<node> children;
  variable var;

  BasicExp(Operation n_op, const vector<node> &n_children, T n_coef = 1);
  BasicExp(const T &v);

  BasicExp(const Variable<T> &v);

  bool has_children() const {
    return op != Operation::variable && op != Operation::literal;
  }

  Variable<T> get_variable() const { return var; }
};

template <typename T> using Expression = shared_ptr<BasicExp<T>>;

template <typename T, typename... Args>
Expression<T> make_exp(Args &&... args) {
  return make_shared<BasicExp<T>, Args...>(std::forward<Args>(args)...);
}

template <typename T> struct Variable {
  int id;

  static Variable<T> get_variable() { return {g_VAR_PTR++}; }

  static vector<Variable<T>> get_variables(int n) {
    vector<Variable<T>> vars(n);
    for (int i = 0; i < n; i++)
      vars[i] = get_variable();
    return vars;
  }

  static Expression<T> get_exp_variable() {
    return Variable<T>::get_variable().as_exp();
  }

  static vector<Expression<T>> get_exp_variables(int n) {
    vector<Expression<T>> vs(n);
    int i = 0;
    for (const auto &v : Variable<T>::get_variables(n)) {
      vs[i++] = v.as_exp();
    }
    return vs;
  }

  operator Expression<T>() const { return make_exp<T>(*this); }

  Expression<T> as_exp() const { return Expression<T>(*this); }

  bool operator<(const Variable<T> &rhs) const { return id < rhs.id; }
};

template <typename T>
BasicExp<T>::BasicExp(Operation n_op, const vector<node> &n_children, T
    n_coef)
    : op(n_op), children(n_children), coef(n_coef) {}

template <typename T> BasicExp<T>::BasicExp(const T &v) {
  op = Operation::literal;
  coef = v;
}

template <typename T> BasicExp<T>::BasicExp(const Variable<T> &v) {
  op = Operation::variable;
  var = v;
}

```

```cpp
template <typename T> Expression<T> &operator*=(Expression<T> &e, const T &x) {
  e->coef *= x;
  return e;
}

template <typename T>
Expression<T> operator*(const Expression<T> &e, const T &x) {
  auto res = make_exp<T>(*e);
  return res *= x;
}

template <typename T>
Expression<T> &operator+=(Expression<T> &e, const Expression<T> &rhs) {
  if (e->op == Operation::sum) {
    e->children.push_back(rhs);
  } else {
    e = make_exp<T>(Operation::sum, vector<Expression<T>>{e, rhs});
  }
  return e;
}

template <typename T>
Expression<T> &operator+=(Expression<T> &e, const Variable<T> &rhs) {
  return e += make_exp<T>(rhs);
}

template <typename T> Expression<T> &operator+=(Expression<T> &e, const T &x) {
  return e += make_exp<T>(x);
}

template <typename T>
Expression<T> operator+(const Expression<T> &e, const Expression<T> &rhs) {
  auto res = e->op == Operation::sum ? make_exp<T>(*e) : e;
  return res += rhs;
}

template <typename T>
Expression<T> operator+(const Expression<T> &e, const Variable<T> &rhs) {
  return e + make_exp<T>(rhs);
}

template <typename T>
Expression<T> operator+(const Expression<T> &e, const T &x) {
  return e + make_exp<T>(x);
}

template <typename T>
Expression<T> operator+(const Variable<T> &v, const Expression<T> &rhs) {
  return make_exp<T>(v) + rhs;
}

template <typename T>
Expression<T> operator+(const Variable<T> &v, const Variable<T> &rhs) {
  return make_exp<T>(v) + make_exp<T>(rhs);
}

template <typename T>
Expression<T> operator+(const Variable<T> &v, const T &x) {
  return make_exp<T>(v) + make_exp<T>(x);
}

template <typename T>
Expression<T> operator*(const Variable<T> &v, const T &x) {
  return make_exp<T>(v) * x;
}

template <typename T> struct ExpressionVisitor {
  void visit(const Expression<T> &e) {
    if (e->op == Operation::sum)
      this->visit_sum(e);
    else if (e->op == Operation::variable)
      this->visit_variable(e);
    else if (e->op == Operation::literal)
      this->visit_literal(e);
  }
  virtual void visit_children(const Expression<T> &e) {
    if (e->has_children()) {
      for (const Expression<T> &child : e->children)
        this->visit(child);
    }
  }

  virtual void visit_sum(const Expression<T> &e) { this->visit_children(e); }
  virtual void visit_variable(const Expression<T> &e) {}
  virtual void visit_literal(const Expression<T> &e) {}
};

template <typename T> struct VariableVisitor : ExpressionVisitor<T> {
  set<Variable<T>> seen;
  virtual void visit_variable(const Expression<T> &e) { seen.insert(e->var); }
};

template <typename T, typename S = T>
struct StackVisitor : ExpressionVisitor<T> {
  vector<S> sta;
  virtual void visit_children(const Expression<T> &e) override {
    sta.push_back(sta.empty() ? e->coef : sta.back() * e->coef);
    ExpressionVisitor<T>::visit_children(e);
    if (!sta.empty())
      sta.pop_back();
  }
  S top() const { return sta.empty() ? S(1) : sta.back(); }
};

template <typename T> struct EvalVisitor : StackVisitor<T> {
  map<Variable<T>, T> values;
  T result;
  T eval(const Expression<T> &e, const map<Variable<T>, T> &values) {
    result = T();
    this->values = values;
    this->visit(e);
    return result;
  }
  virtual void visit_variable(const Expression<T> &e) override {
    result += this->top() * e->coef * values[e->var];
  }
  virtual void visit_literal(const Expression<T> &e) override {
    result += this->top() * e->coef;
  }
};

enum ConstraintOperation {
  equals,
  different,
  greater,
  less,
  greater_eq,
```

```
  less_eq
};

template <typename T> struct Constraint {
  Expression<T> lhs, rhs;
  ConstraintOperation op;
  Constraint(const Expression<T> &a, const Expression<T> &b,
             ConstraintOperation op)
      : lhs(a), rhs(b), op(op) {}
};

template <typename T>
Constraint<T> operator==(const Expression<T> &a, const Expression<T> &b) {
  return Constraint<T>(a, b, ConstraintOperation::equals);
}

template <typename T>
Constraint<T> operator!=(const Expression<T> &a, const Expression<T> &b) {
  return Constraint<T>(a, b, ConstraintOperation::different);
}

template <typename T>
Constraint<T> operator>=(const Expression<T> &a, const Expression<T> &b) {
  return Constraint<T>(a, b, ConstraintOperation::greater_eq);
}

template <typename T>
Constraint<T> operator<=(const Expression<T> &a, const Expression<T> &b) {
  return Constraint<T>(a, b, ConstraintOperation::less_eq);
}

template <typename T>
Constraint<T> operator>(const Expression<T> &a, const Expression<T> &b) {
  return Constraint<T>(a, b, ConstraintOperation::greater);
}

template <typename T>
Constraint<T> operator<(const Expression<T> &a, const Expression<T> &b) {
  return Constraint<T>(a, b, ConstraintOperation::less);
}

template <typename T>
T eval(const Expression<T> &e, const map<Variable<T>, T> &values) {
  auto visitor = std::make_unique<EvalVisitor<T>>();
  return visitor->eval(e, values);
}

} // namespace lib

#endif
```

1.45. Template

```
#include <bits/stdc++.h>
#define int long long
using namespace std;

#define mp make_pair
#define mt make_tuple
#define pb push_back
#define ms(v, x) memset((v), (x), sizeof(v))
#define all(v) (v).begin(), (v).end()
#define ff first
#define ss second
#define iopt ios::sync_with_stdio(false); cin.tie(0)
#define untie(p, a, b) decltype(p.first) a = p.first, decltype(p.second) b =
    p.second
#define TESTCASE(tn) cout << "Case #" << tn << ": "

int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }

int floor2(int x, int y);
int ceil2(int x, int y) {
  if(y < 0) return ceil2(-x, -y);
  return x < 0 ? -floor2(-x, y) : (x + y - 1) / y;
}
int floor2(int x, int y) {
  if(y < 0) return floor2(-x, -y);
  return x < 0 ? -ceil2(-x, y) : x / y;
}

typedef pair<int, int> ii;
typedef long double LD;
typedef vector<int> vi;

#define TC_MAIN int32_t main() { iopt; int T; cin >> T; for(int i = 1; i <=
    T; i++) solve(i); }
```

1.46. Traits

```
#ifndef _LIB_TRAITS
#define _LIB_TRAITS
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace traits {

template <typename...> struct make_void { using type = void; };

template <typename... T> using void_t = typename make_void<T...>::type;

/// keep caide
template <typename Iterator>
using IteratorCategory = typename
    iterator_traits<Iterator>::iterator_category;

/// keep caide
template <typename Container>
using IteratorCategoryOf = IteratorCategory<typename Container::iterator>;

/// keep caide
template <typename Iterator>
using IteratorValue = typename iterator_traits<Iterator>::value_type;

/// keep caide
template <typename Container>
using IteratorValueOf = IteratorValue<typename Container::iterator>;

/// keep caide
template <typename Iterator>
using IsRandomIterator =
    is_base_of<random_access_iterator_tag, IteratorCategory<Iterator>>;

/// keep caide
template <typename Iterator>
using IsInputIterator =
    is_base_of<input_iterator_tag, IteratorCategory<Iterator>>;
```

```

/// keep caide
template <typename Iterator>
using IsBidirectionalIterator =
    is_base_of<bidirectional_iterator_tag, IteratorCategory<Iterator>>;

/// keep caide
template <typename Container>
using HasRandomIterator =
    is_base_of<random_access_iterator_tag, IteratorCategoryOf<Container>>;

/// keep caide
template <typename Container>
using HasInputIterator =
    is_base_of<input_iterator_tag, IteratorCategoryOf<Container>>;

/// keep caide
template <typename Container>
using HasBidirectionalIterator =
    is_base_of<bidirectional_iterator_tag, IteratorCategoryOf<Container>>;
} // namespace traits
} // namespace lib

#endif
```

1.47. Treap

```
#ifndef _LIB_TREAP
#define _LIB_TREAP
#include "Random.cpp"
#include "SegtreeImplicit.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace treap {
template <typename T> struct SearchResult {
  bool found;
  T node;
};

struct EmptyPushdown {
  template <typename Node>
  inline void operator()(Node &no, Node *ln, Node *rn) const {}
};

struct EmptyCombiner {
  template <typename Node>
  inline void operator()(Node &no, Node *ln, Node *rn) const {}
};

template <typename T, typename Less = std::less<T>> struct DefaultNode {
  T key;
  int y;

  DefaultNode() {}
  DefaultNode(T key)
      : key(key), y(rng_gen.uniform_int(numeric_limits<int>::max())) {}

  inline bool operator<(const DefaultNode &rhs) const {
    return Less()(key, rhs.key);
  }

  inline int priority() const { return y; }

  template <typename Combiner>
  inline static void combine(DefaultNode &no, DefaultNode *ln, DefaultNode *rn,
                             const Combiner &combiner) {
    combiner(no, ln, rn);
  }
};

template <typename T, typename Combiner = EmptyCombiner,
          typename Pushdown = EmptyPushdown, typename Less = std::less<T>,
          typename TreapNode = DefaultNode<T, Less>,
          template <class> class ManagerTemplate = seg::Implicit>
struct TreapManager {
  using NodeManager = ManagerTemplate<TreapNode>;
  typedef TreapNode tnode;
  typedef typename NodeManager::vnode vnode;

  Combiner combiner_fn;
  Pushdown pushdown_fn;
  NodeManager manager;

  inline vnode make(T key) { return manager.make(TreapNode(key)); }
  inline vnode null() const { return manager.invalid(); }
  inline void push(vnode no) {}
  inline void update(vnode no) {
    if (!manager.has(no))
      return;
    combiner_fn(manager.ref(no), manager.ptr(manager.left(no)),
                manager.ptr(manager.right(no)));
  }

  template <typename Checker> bool check(vnode no, const Checker &checker) {
    if (!manager.has(no))
      return false;
    return checker(manager.ref(no), manager.ptr(manager.left(no)),
                   manager.ptr(manager.right(no)));
  }

  template <typename Checker>
  vnode bsearch_last_impl(vnode no, const Checker &checker) {
    push(no);
    if (!manager.has(no))
      return null();
    if (check(manager.right(no), checker))
      return bsearch_last_impl(manager.right(no), checker);
    else if (check(no, checker))
      return no;
    else
      return bsearch_last_impl(manager.left(no), checker);
  }

  template <typename Folder, typename Checker>
  vnode bsearch_last_impl(vnode no, const Folder &folder,
                          const Checker &checker) {
    push(no);
    if (!manager.has(no))
      return null();
  }

  template <typename Checker>
  SearchResult<tnode> bsearch_last(vnode no, const Checker &checker) {
    auto res = bsearch_last_impl(no, checker);
    if (!manager.has(res))
      return {false};
```

```
    return {true, manager.value(res)};
  }

  vnode merge(vnode small, vnode large) {
    push(small), push(large);
    vnode res;
    if (!manager.has(small))
      res = manager.replace(small, large);
    else if (!manager.has(large))
      res = manager.replace(large, small);
    else {
      const auto &t_small = manager.ref(small);
      const auto &t_large = manager.ref(large);
      if (t_small.priority() > t_large.priority()) {
        res = manager.persist(small);
        merge(manager.right(res), large);
      } else {
        res = manager.persist(large);
        merge(small, manager.left(res));
      }
    }
    update(res);
    return res;
  }

  template <typename Checker>
  pair<vnode, vnode> split(vnode no, const Checker &checker) {
    push(no);
    if (!manager.has(no))
      return {null(), null()};
    pair<vnode, vnode> res;
    no = manager.persist(no);
    if (check(no, checker)) {
      auto sp = split(manager.right(no), checker);
      manager.replace(manager.right(no), sp.first);
      res = {no, sp.second};
    } else {
      auto sp = split(manager.left(no), checker);
      manager.replace(manager.left(no), sp.second);
      res = {sp.first, no};
    }
    update(no);
    return res;
  }

  template <typename Checker>
  pair<vnode, vnode> split_on_node(vnode no, const Checker &checker) {
    return split(no, [&checker](const TreapNode &no, TreapNode *ln,
                                TreapNode *rn) { return checker(no); });
  }

  pair<vnode, vnode> split_on_key(vnode no, T x) {
    return split_on_node(no, [&x](const TreapNode &no) { return no.key < x;
    });
  }
};
} // namespace treap
} // namespace lib

#endif
```

1.48. TwoSat

```
#ifndef _LIB_TWO_SAT
#define _LIB_TWO_SAT
#include "Graph.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace graph {
#define POS(x) (2*(x))
#define NEG(x) (2*(x)+1)
#define VAR(x) ((x) < 0 ? NEG(-(x)) : POS(x))

// TODO: reuse graph structure and extract tarjan
struct TwoSat {
  int n, sz;
  vector<vector<int>> adj;

  int tempo, cnt;
  vector<int> low, vis, from;
  stack<int> st;
  vector<bool> res;

  TwoSat(int n) : n(n), adj(2*n){}

  int add_dummy() {
    int res = adj.size();
    for(int i = 0; i < 2; i++)
      adj.push_back(vector<int>());
    return res;
  }

  int convert(int x) const { return 2*x; }
  void add_edge(int a, int b) { adj[a].push_back(b); }
  void or_clause(int a, int b){
    add_edge(a^1, b);
    add_edge(b^1, a);
  }

  void implication_clause(int a, int b){
    or_clause(a^1, b);
  }

  void literal_clause(int x) { or_clause(x, x); }
  void and_clause(int a, int b){
    literal_clause(a);
    literal_clause(b);
  }

  void xor_clause(int a, int b){
    or_clause(a, b);
    or_clause(a^1, b^1);
  }

  void nand_clause(int a, int b){
    or_clause(a^1, b^1);
  }

  void nor_clause(int a, int b){
    literal_clause(a^1);
    literal_clause(b^1);
  }

  void equals(int a, int b){
    implication_clause(a, b);
    implication_clause(b, a);
  }
```

```

  void max_one_clause(const vector<int> & v){
    vector<int> p;
    for(int i = 0; i < v.size(); i++)
      p.push_back(add_dummy());

    for(int i = 0; i < v.size(); i++){
      implication_clause(v[i], p[i]);
      if(i+1 < v.size()){
        implication_clause(p[i], p[i+1]);
        implication_clause(p[i], v[i+1]^1);
      }
    }
  }

  void clear(){
    for(int i = 0; i < adj.size(); i++)
      adj[i].clear();
  }

  void tarjan(int u){
    low[u] = vis[u] = ++tempo;
    st.push(u);

    for(int v : adj[u]){
      if(!vis[v]){
        tarjan(v);
        low[u] = min(low[u], low[v]);
      } else if(vis[v] > 0)
        low[u] = min(low[u], vis[v]);
    }

    if(low[u] == vis[u]){
      int k;
      do{
        k = st.top();
        st.pop();
        from[k] = cnt;
        vis[k] = -1;
      } while(k != u);
      cnt++;
    }
  }

  bool solve(){
    sz = adj.size();
    assert(sz%2 == 0);

    low.assign(sz, 0);
    vis.assign(sz, 0);
    tempo = 0;
    cnt = 0;
    from.assign(sz, -1);
    st = stack<int>();

    res.assign(n, true);

    for(int i = 0; i < sz; i++)
      if(!vis[i])
        tarjan(i);

    for(int i = 0; i < sz; i += 2){
      if(from[i] == from[i^1]) return false;
      else if(from[i] > from[i^1] && (i>>1) < n)
        res[i>>1] = false;
    }

    return true;
  }

  bool get(int i) const { return res[i]; }
};
} // namespace graph
} // namespace lib

#endif
```

1.49. VectorN

```
#ifndef _LIB_VECTOR_N
#define _LIB_VECTOR_N
#include <bits/stdc++.h>
#include "Traits.cpp"

#define VEC_CONST_OP(op, typ) \
  type operator op(const typ rhs) const { \
    auto res = *this; \
    return res op##= rhs; \
  }

#define VEC_BIN_OP(op) \
  type& operator op##=(const type& rhs) { \
    if(rhs.size() > this->size()) \
      this->resize(rhs.size()); \
    int sz = this->size(); \
    for(int i = 0; i < (int)rhs.size(); i++) \
      (*this)[i] op##= rhs[i]; \
    for(int i = rhs.size(); i < sz; i++) \
      (*this)[i] op##= 0; \
    return *this; \
  } \
  VEC_CONST_OP(op, type)

#define VEC_SINGLE_OP(op, typ) \
  type& operator op##=(const typ rhs) { \
    for(auto& x : *this) \
      x op##= rhs; \
    return *this; \
  } \
  VEC_CONST_OP(op, typ)


namespace lib {
using namespace std;
template<typename T>
struct VectorN : vector<T> {
  using type = VectorN<T>;

  template <
      typename Container,
      typename enable_if<traits::HasInputIterator<Container>::value>::type *
    = nullptr>
  VectorN(const Container &container)
      : vector<T>(container.begin(), container.end()) {}

  VectorN(const initializer_list<T> &v)
      : vector<T>(v.begin(), v.end()) {}

  template<typename... Args>
```

```
  VectorN( Args&&... args )
      : vector<T>(std::forward<Args>(args)...) {}

  VEC_BIN_OP(+)
  VEC_BIN_OP(-)
  VEC_BIN_OP(*)

  VEC_SINGLE_OP(+, T&)
  VEC_SINGLE_OP(-, T&)
  VEC_SINGLE_OP(*, T&)
  VEC_SINGLE_OP(/, T&)
  VEC_SINGLE_OP(^, int64_t)

  type operator-() const {
    auto res = *this;
    for(auto& x : res) x = -x;
    return res;
  }

  type operator%(int n) const {
    // TODO: get rid of this
    // return *const_cast<type*>(this);
    return *this;
  }
};
} // namespace lib

#endif
```

2. ds

2.1. LiChaoTree

```
#ifndef _LIB_LI_CHAO_TREE
#define _LIB_LI_CHAO_TREE

#include <bits/stdc++.h>

namespace lib {
using namespace std;

template <typename D, typename T> struct LiChaoTree {
  inline constexpr static T inf = numeric_limits<T>::max();

  using Fn = function<T(D)>;
  vector<Fn> fns;
  vector<D> xs;
  vector<int> t;

  template <typename U = D,
            typename enable_if<is_integral<U>::value>::type = nullptr>
  LiChaoTree(D left, D right) {
    assert(right > left);
    xs = vector<D>(right - left);
    iota(xs.begin(), xs.end(), left);
    init();
  }

  LiChaoTree(const vector<D>& xs_) : xs(xs_) {
    sort(xs.begin(), xs.end());
    xs.resize(unique(xs.begin(), xs.end()) - xs.begin());
    init();
  }

  void init() {
    t = vector<int>(xs.size() * 4);
    fns.clear();
    fns.push_back([](D x) { return numeric_limits<T>::max(); });
  }

  void add(const Fn &fn) {
    int i = fns.size();
    fns.push_back(fn);
    add(i, 1, 0, xs.size());
  }

  // r is exclusive
  void add(int i, int no, int l, int r) {
    while (1) {
      int mid = (l + r) / 2;
      bool l_wins = fns[i](xs[l]) < fns[t[no]](xs[l]);
      bool r_wins = fns[i](xs[r-1]) < fns[t[no]](xs[r-1]);
      if (l_wins == r_wins) {
        if (l_wins) swap(i, t[no]);
        return;
      }
      bool mid_wins = fns[i](xs[mid]) < fns[t[no]](xs[mid]);
      if (mid_wins)
        swap(i, t[no]);
      if (l + 1 == r)
        return;
      if (l_wins != mid_wins)
        no = 2 * no, r = mid;
      else
        no = 2 * no + 1, l = mid;
    }
  }

  int seg_l, seg_r, seg_idx;
  void add_segment(int no, int l, int r) {
    if (seg_l >= r || seg_r <= l) return;
    if (seg_l <= l && r <= seg_r) add(seg_idx, no, l, r);
    else {
      int mid = (l+r)/2;
      add_segment(2*no, l, mid);
      add_segment(2*no+1, mid, r);
    }
  }

  void add_segment(const Fn& fn, D a, D b) {
    int i = fns.size();
    fns.push_back(fn);
    int l = lower_bound(xs.begin(), xs.end(), a) - xs.begin();
    int r = lower_bound(xs.begin(), xs.end(), b) - xs.begin();
    if (l == r) return;
    seg_idx = i, seg_l = l, seg_r = r;
    add_segment(1, 0, xs.size());
  }

  T query(D x, int no, int l, int r) const {
    auto res = inf;
    while (1) {
      res = min(res, fns[t[no]](x));
      if (l + 1 == r)
        return res;
      int mid = (l + r) / 2;
      if (x < xs[mid])
        no = 2 * no, r = mid;
      else
```

```
        no = 2 * no + 1, l = mid;
      }
    }
  }

  T query(D x) const { return query(x, 1, 0, xs.size()); }
};
} // namespace lib

#endif
```

2.2. OrderedIntTree

```
#ifndef _LIB_ORDERED_INT_TREE
#define _LIB_ORDERED_INT_TREE
#include <bits/stdc++.h>

namespace lib {
using namespace std;

namespace ds {

template <typename T>
struct Node {
  int key;
  T data;
};

template<>
struct Node<void> {
  int key;
};

template <typename T = void>
struct OrderedIntTree {

};

}
} // namespace lib

#endif
```

2.3. StaticRMQ

```
#ifndef _LIB_STATIC_RMQ
#define _LIB_STATIC_RMQ
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace {
  inline int lsb(int x) { return x&-x; }
}

// Credits: hly1204
template<typename T, typename Cmp = std::less<T>>
struct StaticRMQ {
  Cmp cmp;
  vector<T> t1, t2, a;

  StaticRMQ() {}

  StaticRMQ(const vector<T>& a)
    : t1(a.size() + 1), t2(a.size() + 1), a(a) {
    copy(a.begin(), a.end(), t1.begin() + 1);
    copy(a.begin(), a.end(), t2.begin() + 1);
    build();
  }

  int size() const { return (int)t1.size() - 1; }

  T best(const T& a, const T& b) const {
    return cmp(a, b) ? a : b;
  }

  void build() {
    int n = size();
    for(int i = 1; i <= n; i++) {
      int b = lsb(i);
      if(i + b <= n) t1[i + b] = best(t1[i + b], t1[i]);
    }
    for(int i = n; i; i--) {
      int b = lsb(i);
      t2[i - b] = best(t2[i - b], t2[i]);
    }
  }

  // [l, r], 0-indexed
  T query(int l, int r) const {
    if(l == r) return a[l];
    ++l, ++r;
    T ans = best(a[l-1], a[r-1]);
    int x = l;
    for(; x + lsb(x) - 1 <= r; x += lsb(x))
      ans = best(ans, t2[x]);
    for(int y = r; y != 0 && y - lsb(y) + 1 >= l; y -= lsb(y))
      ans = best(ans, t1[y]);
    if(x <= r)
      ans = best(ans, a[x-1]);
    return ans;
  }
};
} // namespace lib

#endif
```

3. dsu

3.1. BinaryLifting

```
#ifndef _LIB_DSU_BINARY_LIFTING
#define _LIB_DSU_BINARY_LIFTING
#include <bits/stdc++.h>
#include "SpanningTree.cpp"

namespace lib {
using namespace std;
namespace dsu {

template<typename D>
struct BinaryLifting : public D {
  using D::parent;
  vector<vector<int>> P;
  int K;

  BinaryLifting() : D() {}
  BinaryLifting(int n) : D(n) {
```

```cpp
    P = decltype(P)(n, vector<int>(__lg(n)+1, -1));
    K = __lg(n)+1;
  }
  virtual void clear() override {
    D::clear();
    int n = P.size();
    P = decltype(P)(n, vector<int>(K, -1));
  }
  virtual int merge(int u, int v) override {
    if(!D::merge(u, v)) return 0;
    this->traverse_last_small([this](int u, int p, vector<int>&) {
      for(int& x : P[u]) x = -1;
      P[u][0] = p;
      for(int i = 1; i < K; i++) {
        if(P[u][i-1] == -1) break;
        P[u][i] = P[P[u][i-1]][i-1];
      }
    }, no_op_visitor);
    return 1;
  }
  int parent(int u, int k) {
    assert(k >= 0);
    for(int i = K-1; i >= 0; i--) {
      if(!((k>>i)&1)) continue;
      u = P[u][i];
      if(u == -1) return -1;
    }
    return u;
  }
};
} // namespace dsu
} // namespace lib

#endif
```

3.2. Compress

```cpp
#ifndef _LIB_DSU_COMPRESS
#define _LIB_DSU_COMPRESS
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace dsu {

template<typename D>
struct Compress : public D {
  using D::r;

  Compress() : D() {}
  Compress(int n) : D(n) {}

  virtual int get(int i) const override {
    return r[i] == i ? i : r[i] = get(r[i]);
  }
};
} // namespace dsu
} // namespace lib

#endif
```

3.3. DSU

```cpp
#ifndef _LIB_RANK_DSU
#define _LIB_RANK_DSU
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace dsu {
struct RankDSU {
  mutable vector<int> r, sz;
  pair<int, int> last_merge_ = {-1, -1};
  bool last_swapped_ = false;
  int merges = 0;
  RankDSU() {}
  RankDSU(int n) : r(n), sz(n, 1) {
    iota(r.begin(), r.end(), 0);
  }
  virtual void clear() {
    iota(r.begin(), r.end(), 0);
    fill(sz.begin(), sz.end(), 1);
    last_merge_ = {-1, -1};
    merges = 0;
  }
  virtual int get(int i) const {
    return r[i] == i ? i : get(r[i]);
  }
  int operator[](int i) const {
    return get(i);
  }
  pair<int, int> last_merge() const {
    return last_merge_;
  }
  int n_comps() const { return (int)r.size() - merges; }
  virtual void merged(int u, int v) {}
  virtual int merge(int u, int v) {
    u = get(u), v = get(v);
    if(u == v) return 0;
    last_swapped_ = false;
    if(sz[u] > sz[v]) swap(u, v), last_swapped_ = true;
    r[u] = v;
    sz[v] += sz[u];
    last_merge_ = {u, v};
    merges++;
    merged(u, v);
    return 1;
  }
};

template<template<class> class ...Ts>
struct ByRankImpl;

template<template<class> class T, template<class> class ...Ts>
struct ByRankImpl<T, Ts...> {
  using type = T<typename ByRankImpl<Ts...>::type>;
};

template<>
struct ByRankImpl<> {
  using type = RankDSU;
};

template<template<class> class ...Ts>
using ByRank = typename ByRankImpl<Ts...>::type;
} // namespace dsu
} // namespace lib
```

```

#endif
```

3.4. SpanningTree

```
#ifndef _LIB_DSU_SPANNING_TREE
#define _LIB_DSU_SPANNING_TREE
#include <bits/stdc++.h>
#include "../utils/LazyArray.cpp"

namespace lib {
using namespace std;
namespace dsu {

const auto no_op_visitor = [](int, int, const vector<int>&) -> void {};

template<typename D>
struct SpanningTree : public D {
  using D::last_swapped_;

  vector<vector<int>> adj;
  vector<int> pai, depth;
  LazyArray<char> vis;
  pair<int, int> last_edge_;

  SpanningTree() : D() {}
  SpanningTree(int n) : D(n), adj(n), pai(n, -1), vis(n, 0), depth(n, 0) {}
  virtual void clear() override {
    D::clear();
    for(int i = 0; i < adj.size(); i++)
      adj[i].clear();
    fill(pai.begin(), pai.end(), -1);
    fill(depth.begin(), depth.end(), 0);
    vis.clear();
    last_edge_ = {-1, -1};
  }
  virtual int merge(int u, int v) override {
    if(!D::merge(u, v)) return 0;
    if(last_swapped_)
      swap(u, v);
    last_edge_ = {u, v};
    vis.clear();
    fix_(u, v, depth[v]+1);
    adj[u].push_back(v);
    adj[v].push_back(u);
    return 1;
  }
  template<typename F, typename G>
  void traverse_last_small(const F& f, const G& g) {
    vis.clear();
    traverse_(last_edge_.first, last_edge_.second, f, g);
  }
  template<typename F, typename G>
  void traverse_(int u, int p, const F& f, const G& g) {
    if(vis.get(u)) return;
    vis[u] = 1;
    f(u, p, adj[u]);
    for(int v : adj[u]) {
      if(v == p || vis.get(v)) continue;
      traverse_(v, u, f, g);
    }
    g(u, p, adj[u]);
  }
  void fix_(int u, int p, int d) {
    if(vis.get(u)) return;
    vis[u] = 1;
    pai[u] = p;
    depth[u] = d;
    for(int v : adj[u]) {
      if(v == p || vis.get(v)) continue;
      fix_(v, u, d+1);
    }
  }
  pair<int, int> last_edge() const {
    return last_edge_;
  }
  int parent(int i) const {
    return pai[i];
  }
};
} // namespace dsu
} // namespace lib

#endif
```

3.5. Time

```
#ifndef _LIB_DSU_TIME
#define _LIB_DSU_TIME
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace dsu {

template<typename D>
struct Time : public D {
  using D::r;
  using D::sz;

  vector<int> t;
  int tempo = 0;
  Time() : D() {}
  Time(int n) : D(n), t(n, 1e9) {}
  virtual void clear() override {
    tempo = 0;
    fill(t.begin(), t.end(), (int)1e9);
  }
  int get(int i, int tt) const {
    return r[i] == i ? i : (t[i] <= tt ? get(r[i]) : i);
  }
  int get_merge_time(int u, int v) const {
    int ans = -1;
    while(u != v) {
      if(sz[u] < sz[v]) swap(u, v);
      ans = max(ans, t[v]);
      if(r[v] == v) return -1;
      v = r[v];
    }
    return ans;
  }
  Time& at_time(int tt) {
    assert(tt >= tempo);
    tempo = tt;
    return *this;
  }
  Time& tick() {
    return at_time(tempo+1);
```

```
  }
  virtual void merged(int u, int v) override {
    D::merged(u, v);
    t[u] = tempo;
  }
};
} // namespace dsu
} // namespace lib

#endif
```

4. graphs

4.1. BlockCut

```
#ifndef _LIB_BLOCK_CUT
#define _LIB_BLOCK_CUT
#include <bits/stdc++.h>
#include "../Graph.cpp"
#include "../utils/LazyArray.cpp"

namespace lib {
  using namespace std;
namespace graph {
template<typename V, typename E>
struct BlockCut {
  int n, m;
  Graph<V, E> g;
  int tempo = 0;
  vector<int> vis, low, seen;
  vector<int> st;
  LazyArray<char> seen_v;

  Graph<V, E> g2;
  int n2 = 0;

  BlockCut(const Graph<V, E>& g) : g(g) {
    n = g.size();
    m = g.edge_size();
    vis = low = vector<int>(n);
    seen = vector<int>(m);
    st.reserve(m);
    seen_v = LazyArray<char>(n, 0);

    g2 = Graph<V, E>(n);

    for(int i = 0; i < n; i++) {
      if(!vis[i]) {
        tarjan(i, -1);
        if (g.degree(i) == 0) {
          // Vertex is isolated, process separately.
          g2.add_vertex();
          g2.add_2edge(n + n2, i);
          n2++;
        }
      }
    }
  }
  Graph<V, E> graph() const { return g2; }

  int n_components() const { return n2; }
  vector<int> component(int i) const {
    vector<int> res;
    for(const auto& v : g2.n_edges(n + i))
      if (v.to < n)
        res.push_back(v.to);
    return res;
  }

  vector<int> get_vertices_(const vector<int>& e) {
    seen_v.clear();
    vector<int> comp;
    for(int kk : e) {
      auto ed = g.edge(kk);
      if(!seen_v.get(ed.from)) comp.push_back(ed.from), seen_v[ed.from] =
    true;
      if(!seen_v.get(ed.to)) comp.push_back(ed.to), seen_v[ed.to] = true;
    }
    return comp;
  }
  void process_component_(int k) {
    vector<int> e;
    int cur;
    do {
      cur = st.back(); st.pop_back();
      e.push_back(cur);
    } while(cur != k);
    auto comp = get_vertices_(e);
    g2.add_vertex();
    for(int w : comp) {
      g2.add_2edge(n + n2, w);
    }
    n2++;
  }
  void tarjan(int u, int p) {
    vis[u] = low[u] = ++tempo;
    auto nei = g.n_edges(u);
    for(int i = 0; i < nei.size(); i++) {
      int k = nei.index(i);
      int v = g.edge(k).to;

      if(!seen[k]) {
        seen[k] = seen[k^1] = 1;
        st.push_back(k);
      }

      if(!vis[v]) {
        tarjan(v, u);
        low[u] = min(low[u], low[v]);

        if(low[v] >= vis[u]) {
          process_component_(k);
        }
      } else {
        low[u] = min(low[u], vis[v]);
      }
    }
  }
};

template<typename V, typename E>
BlockCut<V, E> make_block_cut(const Graph<V, E>& g) {
  return BlockCut<V, E>(g);
}
} // namespace graph
} // namespace lib

#endif
```

4.2. Chordal

```
#ifndef _LIB_CHORDAL
#define _LIB_CHORDAL
#include <bits/stdc++.h>
#include "../utils/FastList.cpp"

namespace lib {
  using namespace std;
namespace graph {
namespace {
  using Elements = pair<vector<int>, int>;
  using SetList = lib::list::Node<Elements>;
  shared_ptr<SetList> make_set_list(int n = 0) {
    return make_shared<SetList>(Elements(vector<int>(n), 0));
  }
}
// No parallel edges or self-loops.
template<typename Graph>
vector<int> lex_bfs(const Graph& g) {
  int n = g.size();
  vector<int> res(n);
  vector<int> vis(n);
  vector<pair<shared_ptr<SetList>, int>> inv(n);
  auto data = make_set_list(n);
  for(int i = 0; i < n; i++) {
    data->val.first[i] = i;
    inv[i] = make_pair(data, i);
  }

  auto head = make_set_list();
  list::append(head.get(), data.get());

  for(int i = 0; i < n; i++) {
    auto no = head->next;
    assert(no != nullptr);
    assert(!no->val.first.empty());
    const int u = res[i] = no->val.first.back();
    no->val.first.pop_back();
    if(no->val.first.empty()) list::remove(no);
    vis[u] = 1;

    // Partition
    for(const auto& e : g.n_edges(u)) {
      int v = e.to;
      if(vis[v]) continue;
      auto st = inv[v].first;
      int sz = st->val.first.size();
      if(sz == 1) continue;
      auto idx = inv[v].second;
      swap(st->val.first[idx], st->val.first[sz - 1 - st->val.second]);
      swap(inv[v].second, inv[st->val.first[idx]].second);
      st->val.second++;
    }

    for(const auto& e : g.n_edges(u)) {
      int v = e.to;
      if(vis[v]) continue;
      auto st = inv[v].first;
      int st_sz = st->val.first.size();
      int size_new = st->val.second;
      assert(size_new <= st_sz);
      if(size_new == 0 || size_new == st_sz) {
        st->val.second = 0;
        continue;
      }
      auto new_data = make_set_list(size_new);
      for(int i = 0; i < size_new; i++) {
        new_data->val.first[i] = st->val.first[st_sz - size_new + i];
        inv[new_data->val.first[i]] = {new_data, i};
      }

      st->val.first.resize(st_sz - size_new);
      st->val.second = 0;

      // both st and new_data should have size > 0 at this point
      list::prepend(st.get(), new_data.get());
    }
  }

  return res;
}

template<typename Graph>
struct Chordal {
  mutable vector<int> vis, par;
  mutable vector<int> cyc;

  Graph g;
  vector<int> order, inv;
  mutable bool was_tested = false;
  Chordal(Graph g) : g(g) {
    order = lex_bfs(g);
    reverse(order.begin(), order.end());
    int n = g.size();
    inv = vector<int>(n);
    for(int i = 0; i < n; i++) inv[order[i]] = i;
  }

  bool is_valid() const {
    if(was_tested) return cyc.empty();
    int n = g.size();

    vector<vector<int>> adj(n);
    for(int i = 0; i < n; i++) {
      for(const auto& e : g.n_edges(i)) {
        adj[i].push_back(e.to);
      }
      sort(adj[i].begin(), adj[i].end());
    }

    for(int k = n-2; k >= 0; k--) {
      int i = order[k];
      pair<int, int> best = {1e9, -1};
      for(const auto& e : g.n_edges(i)) {
        if(inv[e.to] > k)
          best = min(best, {inv[e.to], e.to});
      }
      auto v = best.second;
      if(v == -1) continue;
      for(const auto& e : g.n_edges(i)) {
        if(inv[e.to] > inv[v])
          if(!binary_search(adj[v].begin(), adj[v].end(), e.to)) {
            was_tested = true;
            par.assign(n, -1), vis.assign(n, 0);
            queue<int> q;
            vis[e.to] = 1;
            q.push(e.to);
            while(!q.empty()) {
              int x = q.front(); q.pop();
```

```
            for(const auto& e2 : g.n_edges(x)) {
              int y = e2.to;
              if(vis[y]) continue;
              if(y == i) continue;
              if(y != v && binary_search(adj[i].begin(), adj[i].end(), y)) continue;
              vis[y] = 1;
              q.push(y);
              par[y] = x;
            }
          }
          cyc.clear();
          cyc.push_back(e.to);
          cyc.push_back(i);
          assert(vis[v]);
          for(auto x = v; x != e.to; x = par[x]) cyc.push_back(x);
          return false;
        }
      }
    }
    was_tested = true;
    return true;
  }

  vector<int> induced_cycle() const { return cyc; }

  vector<int> max_independent_set() const {
    int n = g.size();
    vis.assign(n, 0);

    vector<int> res;
    for(int i : order) {
      if(vis[i]) continue;
      res.push_back(i);
      for(const auto& e : g.n_edges(i)) {
        vis[e.to] = 1;
      }
    }
    return res;
  }
};

template<typename Graph>
Chordal<Graph> make_chordal(const Graph& g) {
  return Chordal<Graph>(g);
}
} // namespace graph
} // namespace lib

#endif
```

5. matroid

5.1. CographicMatroid

```
#ifndef _LIB_COGRAPHIC_MATROID
#define _LIB_COGRAPHIC_MATROID
#include <bits/stdc++.h>
#include "GraphicMatroid.cpp"

namespace lib {
  using namespace std;
struct CographicMatroid : GraphicMatroid {

  CographicMatroid(int n, std::function<pair<int, int>(int)> edge_fn_)
    : GraphicMatroid(n, edge_fn_) {}

  void build(const lambda::SubsetFilter& I_) override {
    GraphicMatroid::build(!I_);
  }
  void setup_exchange(int i) {
    setup();
  }
  bool can_exchange(int i, int j) {
    return can_add(j);
  }
  bool can_add(int i) {
    return !is_bridge(i);
  }
};
} // namespace lib

#endif
```

5.2. ColorMatroid

```
#ifndef _LIB_COLOR_MATROID
#define _LIB_COLOR_MATROID
#include <bits/stdc++.h>
#include "Matroid.cpp"

namespace lib {
  using namespace std;
struct ColorMatroid : Matroid {
  vector<int> cnt, limits;
  lambda::Map<int> color;
  ColorMatroid(vector<int> limits, const lambda::Map<int>& color_)
    : Matroid(), limits(limits), color(color_) {}
  ColorMatroid(int n, int K, const lambda::Map<int>& color_)
    : Matroid(), limits(n, K), color(color_) {}
  void setup() {
    cnt.assign(limits.size(), 0);
    for(int i = 0; i < ground_set_size(); i++)
      if(in_I(i))
        cnt[color(i)]++;
  }
  void setup_exchange(int i) {
    cnt[color(i)]--;
  }
  void finish_exchange(int i) {
    cnt[color(i)]++;
  }
  bool can_exchange(int i, int j) {
    return can_add(j);
  }
  bool can_add(int i) {
    int c = color(i);
    return cnt[c] < limits[c];
  }
  void print() const {
    for(int x : cnt) cout << x << " ";
    cout << endl;
  }
};
} // namespace lib

#endif
```

5.3. Compose

```cpp
#ifndef _LIB_COMPOSE_MATROID
#define _LIB_COMPOSE_MATROID
#include <bits/stdc++.h>
#include "Matroid.cpp"
#include "../Lambda.cpp"

namespace lib {
  using namespace std;
namespace matroid {
template<typename M>
struct Filter : Matroid {
  M mat;
  lambda::Filter filter_fn;
  lambda::SubsetMap<int> inv_fn;
  Filter(const M& mat_, const lambda::Filter& filter_fn_)
    : Matroid(), mat(mat_), filter_fn(filter_fn_) {}

  void build(const lambda::SubsetFilter& I_) override{
    Matroid::build(I_);
    auto subset = filter_fn.subset(I_.size());
    inv_fn = subset.take_inverse();
    mat.build(subset.take_from(I_));
  }

  void setup() { mat.setup(); }
  void setup_graph() { mat.setup_graph(); }
  void setup_exchange(int i) {
    mat.setup_exchange(inv_fn(i));
  }
  void finish_exchange(int i) {
    mat.finish_exchange(inv_fn(i));
  }

  bool can_add(int i) {
    if(!filter_fn(i)) return true;
    return mat.can_add(inv_fn(i));
  }
  bool can_exchange(int i, int j) {
    if(!filter_fn(i)) return can_add(j);
    if(!filter_fn(j)) return true;
    return mat.can_exchange(inv_fn(i), inv_fn(j));
  }
};

template<typename M>
Filter<M> make_filter(const M& mat, const lambda::Filter& fn) {
  return Filter<M>(mat, fn);
}
} // namespace matroid
} // namespace lib

#endif
```

5.4. GraphicMatroid

```cpp
#ifndef _LIB_GRAPHIC_MATROID
#define _LIB_GRAPHIC_MATROID
#include <bits/stdc++.h>
#include "Matroid.cpp"
#include "../utils/FastAdj.cpp"

namespace lib {
  using namespace std;
struct GraphicMatroid : Matroid {
  lambda::Map<pair<int, int>> edge;
  FastAdj<pair<int, int>> g;
  vector<int> comp, st, nd, low;
  vector<int> bridges;
  int tempo, comps;
  bool printer = true;

  GraphicMatroid(int n, const lambda::Map<pair<int, int>>& edge_)
    : Matroid(), edge(edge_), g(n, n) {}
  void setup() {
    g.clear();
    g.reserve(ground_set_size());
    for(int i = 0; i < ground_set_size(); i++)
      if(in_I(i)) {
        auto p = edge(i);
        g.add(p.first, {p.second, i});
        g.add(p.second, {p.first, i});
      }
    build_graph();
  }
  void build_graph() {
    int n = g.size();
    comp.assign(n, -1);
    st.assign(n, 0);
    nd.assign(n, 0);
    low.assign(n, 0);
    bridges.assign(ground_set_size(), 0);

    tempo = 0;
    comps = 0;
    for(int i = 0; i < n; i++) {
      if(comp[i] == -1) dfs(i, -1, comps++);
    }
  }
  void dfs(int u, int p, int c) {
    comp[u] = c;
    st[u] = low[u] = tempo++;
    for(auto e : g.n_edges(u)) {
      int v = e.first;
      if(v == p) {
        p = -1;
        continue;
      }
      if(comp[v] != -1) low[u] = min(low[u], st[v]);
      else {
        dfs(v, u, c);
        low[u] = min(low[u], low[v]);
        if(low[v] > st[u]) {
          bridges[e.second] = 1;
        }
      }
    }
    nd[u] = tempo++;
  }
  bool is_bridge(int i) {
    return bridges[i];
  }
  bool is_anc(int u, int v) {
    return st[u] <= st[v] && st[v] <= nd[u];
  }
  bool can_exchange(int i, int j) {
    auto e1 = edge(i);
    auto e2 = edge(j);
```

```
    if(st[e1.first] > st[e1.second]) swap(e1.first, e1.second);
    return is_anc(e1.second, e2.first) + is_anc(e1.second, e2.second) == 1;
  }
  bool can_add(int i) {
    auto e = edge(i);
    return comp[e.first] != comp[e.second];
  }
};
} // namespace lib

#endif
```

5.5. Matroid

```
#ifndef _LIB_MATROID
#define _LIB_MATROID
#include <bits/stdc++.h>
#include "../Lambda.cpp"

namespace lib {
struct Matroid {
  lambda::SubsetFilter I;
  bool in_I(int i) const {
    return I(i);
  }
  vector<bool> get_I() const {
    return I();
  }
  int ground_set_size() const { return I.size(); }

  /** docstring
   * Used to build a Matroid object from an M (independent set provider).
   */
  virtual void build(const lambda::SubsetFilter& I_) {
    I = I_;
  }

  void setup() {}
  void setup_graph() {}
  void setup_exchange(int i) {}
  void finish_exchange(int i) {}

  bool can_add(int i) { return false; }
  bool can_exchange(int i, int j) { return false; }

  void print_I() {
    for(int i = 0; i < I.size(); i++) cout << in_I(i) << " ";
    cout << endl;
  }
};
} // namespace lib

#endif
```

5.6. MatroidIntersection

```
#ifndef _LIB_MATROID_INTERSECTION
#define _LIB_MATROID_INTERSECTION
#include <bits/stdc++.h>
#include "../utils/FastAdj.cpp"
#include "../utils/FastQueue.cpp"
#include "../Lambda.cpp"

namespace lib {
template<typename M1, typename M2, typename W = int>
struct MatroidIntersection {
  int n;
  M1 m1;
  M2 m2;

  // aux vectors
  vector<int> vI;
  vector<int> I;
  vector<int> nd;
  FastQueue<int> q;
  vector<int> p;
  vector<int> ch;
  vector<int> in_q;
  vector<W> w;
  vector<W> dist;

  FastAdj<int> g;

  MatroidIntersection() : q(1) { init (); }
  MatroidIntersection(int n, const M1& m1, const M2& m2) : m1(m1), m2(m2),
    n(n), g(n+2, n), q(n) {
    init();
  }
  void set_weights(const vector<W>& w_) {
    assert(n == w_.size());
    w = w_;
  }
  int size() const { return n; }
  void init() {
    vI.reserve(n);
    p.assign(n, -1);
    I.assign(n, false);
    nd.assign(n, 0);
  }
  void setup_augment() {
    vI.clear();
    g.clear();
    for(int i = 0; i < n; i++) {
      if(I[i]) vI.push_back(i);
      nd[i] = 0;
    }
  }
  bool is_weighted() const {
    return !w.empty();
  }
  bool augment(int truncate = 1e9) {
    setup_augment();
    if(vI.size() == min(truncate, n)) return false;
    auto f = lambda::SubsetFilter(n, [this](int i) -> bool { return in_I(i);
    });
    m1.build(f), m2.build(f);
    m1.setup(), m2.setup();
    // Check potential starting and ending points of the path.
    // Also, return earlier if is both starting and ending point.
    for(int i = 0; i < n; i++) {
      if(I[i]) continue;
      if(m1.can_add(i)) nd[i] |= 1;
      if(m2.can_add(i)) nd[i] |= 2;
      if(nd[i] == 3 && !is_weighted()) {
        I[i] = true;
        return true;
      }
    }
```

```
    m1.setup_graph(), m2.setup_graph();
    for(int i : vI) {
      I[i] = false;
      m1.setup_exchange(i), m2.setup_exchange(i);
      for(int j = 0; j < n; j++) {
        if(I[j] || i == j) continue;
        if(m1.can_exchange(i, j)) g.add(i, j);
        if(m2.can_exchange(i, j)) g.add(j, i);
      }
      I[i] = true;
      m1.finish_exchange(i), m2.finish_exchange(i);
    }

    int st = is_weighted() ? weighted_sp() : unweighted_sp();
    if(st == -1) return false;
    I[st] ^= 1;
    while(p[st] != st) {
      st = p[st];
      I[st] ^= 1;
    }
    return true;
  }
  int unweighted_sp() {
    q.clear();
    p.assign(n, -1);
    for(int i = 0; i < n; i++)
      if(nd[i]&1) q.push(i), p[i] = i;

    int st = -1;
    while(!q.empty() && st == -1) {
      int u = q.pop();
      if(nd[u]&2) {
        st = u;
        break;
      }
      for(int v : g.n_edges(u)) {
        if(p[v] == -1) {
          p[v] = u;
          q.push(v);
        }
      }
    }
    return st;
  }
  int weighted_sp() {
    q.clear();
    in_q.assign(n, 0);
    p.assign(n, -1);
    const W oo = numeric_limits<W>::max() / 2;
    ch.assign(n, 1e9);
    dist.assign(n, oo);
    for(int i = 0; i < n; i++)
      if(nd[i]&1)
        dist[i] = -w[i], ch[i] = 0, p[i] = i, q.push(i), in_q[i] = 1;
    while(!q.empty()) {
      int i = q.pop();
      in_q[i] = 0;
      for(int v : g.n_edges(i)) {
        if(v == i) continue;
        W n_dist = dist[i] + (I[v] ? w[v] : -w[v]);
        int n_ch = ch[i] + 1;
        using ii = pair<W, int>;
        if(ii(n_dist, n_ch) < ii(dist[v], ch[v])) {
          dist[v] = n_dist;
          ch[v] = n_ch;
          p[v] = i;
          if(!in_q[v]) {
            in_q[v] = 1;
            q.push(v);
          }
        }
      }
    }

    pair<pair<W, int>, int> best = {{oo, 1e9}, -1};
    for(int i = 0; i < n; i++) {
      if(nd[i]&2) {
        best = min(best, {{dist[i], ch[i]}, i});
      }
    }
    return best.second;
  }
  vector<int> solve(int truncate = 1e9) {
    while(augment(truncate));
    return I;
  }
  W cost() const {
    W res = 0;
    for(int i = 0; i < n; i++) {
      if(I[i])
        res += is_weighted() ? w[i] : 1;
    }
    return res;
  }
  int cardinality() const {
    int res = 0;
    for(int i = 0; i < n; i++)
      res += I[i];
    return res;
  }
  bool in_I(int i) const {
    return I[i];
  }
  void flip(int i) {
    I[i] ^= 1;
  }
  const vector<int>& get_I() const {
    return I;
  }
};

template<typename M1, typename M2>
shared_ptr<MatroidIntersection<M1, M2>> make_matroid_intersection(int n,
    const M1& m1, const M2& m2) {
  return make_shared<MatroidIntersection<M1, M2>>(n, m1, m2);
}
template<typename W, typename M1, typename M2>
shared_ptr<MatroidIntersection<M1, M2, W>>
    make_weighted_matroid_intersection(int n, const M1& m1, const M2& m2,
    const lambda::Map<W>& f) {
  auto res = make_shared<MatroidIntersection<M1, M2, W>>(n, m1, m2);
  vector<W> w(n);
  for(int i = 0; i < n; i++) w[i] = f(i);
  res->set_weights(w);
  return res;
}
} // namespace lib

#endif
```

6. geometry

6.1. Caliper

```
#ifndef _LIB_GEOMETRY_CALIPER
#define _LIB_GEOMETRY_CALIPER
#include "Line2D.cpp"
#include "Polygon2D.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace geo {
namespace plane {
template <typename T, typename Large = T,
          typename enable_if<!is_integral<T>::value>::type * = nullptr,
          typename enable_if<!is_integral<T>::value>::type * = nullptr>
struct Caliper {
  typedef Point<T, Large> point;
  typedef Line<T, Large> line;
  point p;
  Large ang;
  Caliper(point a, Large alpha) : p(a) {
    ang = remainder(alpha, 2 * trig::PI);
    while (ang < 0)
      ang += 2 * trig::PI;
  }
  Large angle_to(const point &q) const {
    return remainder(arg(q - p) - ang, 2 * trig::PI);
  }
  void rotate(double theta) {
    ang += theta;
    while (ang > 2 * trig::PI)
      ang -= 2 * trig::PI;
    while (ang < 0)
      ang += 2 * trig::PI;
  }
  void move(const point &q) { p = q; }
  point versor() const { return point::polar(1.0, ang); }
  line as_line(Large scale = 1.0) const {
    return line(p, p + versor() * scale);
  }
  friend Large dist(const Caliper &a, const Caliper &b) {
    return dist(a.as_line(), b.p);
  }
};

template <typename T, typename Large = T> struct PolygonCalipers {
  constexpr static Large LIMIT = 4 * acosl(-1);

  typedef Point<T, Large> point;
  typedef Caliper<T, Large> caliper;
  typedef ConvexPolygon<T, Large> polygon;
  typedef pair<int, Large> descriptor;

  polygon poly;
  vector<caliper> calipers;
  vector<int> indices;
  vector<int> walked;
  Large angle_walked;

  PolygonCalipers(const polygon &poly, const vector<descriptor> &descriptors)
      : poly(poly), walked(descriptors.size()), angle_walked(0) {
    indices.reserve(descriptors.size());
    calipers.reserve(descriptors.size());
    for (size_t i = 0; i < descriptors.size(); i++) {
      calipers.emplace_back(poly[descriptors[i].first],
    descriptors[i].second);
      indices.emplace_back(descriptors[i].first);
    }
  }
  caliper operator[](int i) const { return calipers[i]; }
  int index(int i) const { return indices[i]; }
  bool has_next() const {
    return *min_element(walked.begin(), walked.end()) < poly.size() &&
           angle_walked < LIMIT;
  }
  Large angle_to_next(int i) const {
    int u = indices[i];
    return calipers[i].angle_to(poly[u + 1]);
  }
  void step_(int i) {
    int u = indices[i]++;
    indices[i] %= poly.size();
    calipers[i].move(poly[u + 1]);
    walked[i]++;
  }

  void next() {
    int i = 0;
    Large best = angle_to_next(0);
    for (size_t j = 1; j < calipers.size(); j++) {
      Large cur = angle_to_next(j);
      if (cur < best) {
        best = cur;
        i = j;
      }
    }
    Large alpha = angle_to_next(i);
    for (auto &caliper : calipers)
      caliper.rotate(alpha);
    step_(i);
    angle_walked += alpha;
  }
};
} // namespace plane

} // namespace geo
} // namespace lib

#endif
```

6.2. Circle2D

```
#ifndef _LIB_GEOMETRY_CIRCLE_2D
#define _LIB_GEOMETRY_CIRCLE_2D
#include "../utils/Annotation.cpp"
#include "Line2D.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace geo {
namespace plane {
template <typename T, typename Large = T> struct Barycentric {
  typedef Point<T, Large> point;
  point r1, r2, r3;
  T a, b, c;

```

```
  Barycentric(const point &r1, const point &r2, const point &r3, T a = 1,
              T b = 1, T c = 1)
      : r1(r1), r2(r2), r3(r3), a(a), b(b), c(c) {}
  point as_point() const { return (r1 * a + r2 * b + r3 * c) / (a + b + c); }

  static Barycentric centroid(const point &r1, const point &r2,
                              const point &r3) {
    return Barycentric(r1, r2, r3);
  }
  static Barycentric circumcenter(const point &r1, const point &r2,
                                  const point &r3) {
    Large a = norm_sq(r2 - r3), b = norm_sq(r3 - r1), c = norm_sq(r1 - r2);
    return Barycentric(r1, r2, r3, a * (b + c - a), b * (c + a - b),
                       c * (a + b - c));
  }
  static Barycentric incenter(const point &r1, const point &r2,
                              const point &r3) {
    return Barycentric(r1, r2, r3, norm(r2 - r3), norm(r1 - r3), norm(r1 -
    r2));
  }
  static Barycentric orthocenter(const point &r1, const point &r2,
                                 const point &r3) {
    Large a = norm_sq(r2 - r3), b = norm_sq(r3 - r1), c = norm_sq(r1 - r2);
    return Barycentric(r1, r2, r3, (a + b - c) * (c + a - b),
                       (b + c - a) * (a + b - c), (c + a - b) * (b + c - a));
  }
  static Barycentric excenter(const point &r1, const point &r2,
                              const point &r3) {
    return Barycentric(r1, r2, r3, -norm(r2 - r3), norm(r1 - r3),
                       norm(r1 - r2));
  }
};

template <typename T, typename Large = T> struct Circle {
  typedef Point<T, Large> point;
  typedef Line<T, Large> line;
  typedef Barycentric<Large> bary;
  typedef Segment<T, Large> segment;
  point center;
  T radius;

  Circle(point center, T radius) : center(center), radius(radius) {}
  Circle(const point &p1, const point &p2, const point &p3) {
    center = bary::circumcenter(p1, p2, p3).as_point();
    radius = dist(center, p1);
  }
  Circle(const point &p1, const point &p2) {
    center = (p1 + p2) / 2;
    radius = dist(center, p1);
  }
  bool crosses_x_axis(point p = point()) const {
    auto c = center - p;
    return GEOMETRY_COMPARE0(T, c.y + radius) >= 0 && GEOMETRY_COMPARE0(T,
    c.y - radius) < 0;
  }
  static Circle incircle(const point &p1, const point &p2, const point &p3) {
    point center = bary::incenter(p1, p2, p3).as_point();
    return Circle(center, dist(line(p1, p2), center));
  }
  friend pair&lt;segment, int&gt; intersect_segment(const Circle &c, const line
    &l) {
    point H = project(c.center, l);
    Large h = norm(H - c.center);
    if (GEOMETRY_COMPARE(Large, c.radius, h) < 0)
      return {{}, 0};
    Large norma = sqrtl(c.radius + h) * sqrtl(c.radius - h);
    point v = normalized(l.direction(), norma);
    segment res = segment(H - v, H + v);
    return {res, res.is_degenerate() ? 1 : 2};
  }
  friend Large intersection_area(const Circle &a, const Circle &b) {
    Large d = norm(a.center - b.center);
    if (GEOMETRY_COMPARE(Large, a.radius + b.radius, d) <= 0)
      return 0.0;
    if (GEOMETRY_COMPARE(Large, d, abs(a.radius - b.radius)) <= 0) {
      T r = min(a.radius, b.radius);
      return r * r * trig::PI;
    }

    auto compute = [d](Large ra, Large rb) {
      Large sup = rb * rb + d * d - ra * ra;
      Large alpha = trig::acos(sup / (2.0 * rb * d));
      Large s = alpha * rb * rb;
      Large t = rb * rb * trig::sin(alpha) * trig::cos(alpha);
      return s - t;
    };
    return compute(a.radius, b.radius) + compute(b.radius, a.radius);
  }
  static Large intersection_signed_area(T r, const point &a, const point &b)
    {
    Circle C(point(), r);
    auto ps = intersect_segment(C, line(a, b));
    if (!ps.second)
      return r * r * signed_angle(a, b) / 2;
    auto s = ps.first;
    bool outa = !contains(C, a), outb = !contains(C, b);
    if (outa && outb) {
      segment ab(a, b);
      if (ab.contains(s.a) && ab.contains(s.b))
        return (r * r * (signed_angle(a, b) - signed_angle(s.a, s.b)) +
                cross(s.a, s.b)) /
               2;
      return r * r * signed_angle(a, b) / 2;
    } else if (outa)
      return (r * r * signed_angle(a, s.a) + cross(s.a, b)) / 2;
    else if (outb)
      return (r * r * signed_angle(s.b, b) + cross(a, s.b)) / 2;
    else
      return cross(a, b) / 2;
  }
  friend vector<point> tangents(const Circle &C, const point &p) {
    return _tangents({p, T()}, C, {1});
  }
  friend vector<line> inner_tangents(const Circle& a, const Circle& b) {
    return _tangents(a, b, {-1});
  }
  friend vector<line> outer_tangents(const Circle& a, const Circle& b) {
    return _tangents(a, b, {1});
  }
  friend vector<line> _tangents(const Circle& a, const Circle& b, const
    initializer_list<int>& r_sgn) {
    vector<line> res;
    for(int r_s : r_sgn) {
      point d = b.center - a.center;
      Large dr = (a.radius - b.radius*r_s), d2 = norm_sq(d), h2 = d2 - dr*dr;
      if(GEOMETRY_COMPARE0(Large, d2) == 0) continue;
      if(GEOMETRY_COMPARE0(Large, h2) < 0) continue;
      for(T sgn : {-1, 1}) {
        point v = (d * dr + ortho(d) * sqrtl(h2) * sgn) / d2;
```

```
        res.push_back({a.center + v * a.radius, b.center + v * (b.radius * r_s)});
      }
      if(GEOMETRY_COMPARE0(Large, h2) == 0) res.pop_back();
    }
    return res;
  }
  friend vector<Note<line, int>> angular_tangents(const Circle& a, const vector<Circle>& v, vector<int>& sgn) {
    vector<Note<line, int>> res;
    res.reserve(4 * v.size());
    int i = 0;
    sgn = vector<int>(v.size());
    vector<bool> reversed(4);
    bool null_a = GEOMETRY_COMPARE0(T, a.radius) == 0;

    for(int i = 0; i < v.size(); i++) {
      bool null_i = GEOMETRY_COMPARE0(T, v[i].radius) == 0;
      assert(!null_a || !null_i);
      vector<line> tgts;
      if(null_a || null_i) tgts = _tangents(a, v[i], {1});
      else tgts = _tangents(a, v[i], {+1, -1});
      if(tgts.empty()) continue;

      fill(reversed.begin(), reversed.end(), false);
      int j = 0;
      for(auto& t : tgts) {
        // direct tangents
        if(ccw(t.b - t.a, a.center - t.a) < 0)
          swap(t.a, t.b), reversed[j] = true;
        res.push_back(make_note<line, int>(t, i));
        j++;
      }

      // check signal
      auto it = AngleComparator<RayDirection<line>, T, Large>::minByAngle(tgts.begin(), tgts.end());
      point ta = reversed[it - tgts.begin()] ? it->b : it->a;
      point dir = v[i].center - ta;
      sgn[i] = half_ccw(it->direction(), dir);
    }
    AngleComparator<RayDirection<line>, T, Large>::sortByAngle(res.begin(), res.end());
    return res;
  }
  friend bool contains(const Circle &c, const point &p) {
    return GEOMETRY_COMPARE(Large, dist(p, c.center), c.radius) <= 0;
  }
  friend bool contains(const Circle &c, const segment &s) {
    return GEOMETRY_COMPARE(Large, dist(s.a, c.center), c.radius) <= 0 &&
           GEOMETRY_COMPARE(Large, dist(s.b, c.center), c.radius) <= 0;
  }
  template <typename L>
  friend bool partially_contains(const Circle &c, const L &l) {
    return GEOMETRY_COMPARE(Large, dist(l, c.center), c.radius) <= 0;
  }
  template <typename L>
  friend bool has_unique_intersection(const Circle &c, const L &l) {
    return GEOMETRY_COMPARE(Large, dist(l, c.center), c.radius) == 0;
  }
  template <typename L>
  friend bool has_intersection(const Circle &c, const L &l) {
    return GEOMETRY_COMPARE(Large, dist(l, c.center), c.radius) <= 0;
  }
  friend bool has_intersection(const Circle &c, const segment &s) {
    return GEOMETRY_COMPARE(Large, dist(s, c.center), c.radius) <= 0 &&
            (GEOMETRY_COMPARE(Large, dist(s.a, c.center), c.radius) >= 0 ||
             GEOMETRY_COMPARE(Large, dist(s.b, c.center), c.radius) >= 0);
  }
};
} // namespace plane

template <typename T, typename Large = T>
struct CirclePlane : public CartesianPlane<T, Large> {
  typedef plane::Circle<T, Large> circle;
};

} // namespace geo
} // namespace lib

#endif
```

6.3. GeometryEpsilon

```
#ifndef _LIB_GEOMETRY_EPSILON
#define _LIB_GEOMETRY_EPSILON
#include "../Epsilon.cpp"
#include <bits/stdc++.h>

#define GEOMETRY_EPSILON(T, x) \
  template <> \
  lib::Epsilon<T> *lib::geo::GeometryEpsilon<T>::eps = \
      new lib::Epsilon<T>((x));

#define GEOMETRY_COMPARE0(T, x) GeometryEpsilon<T>()((x))
#define GEOMETRY_COMPARE(T, x, y) GeometryEpsilon<T>()((x), (y))

namespace lib {
using namespace std;
namespace geo {
template <typename T> struct GeometryEpsilon {
  static Epsilon<T> *eps;
  template <typename G> int operator()(G a, G b = 0) const {
    return (*eps)(a, b);
  }
};

GEOMETRY_EPSILON(int, 0);
GEOMETRY_EPSILON(long, 0);
GEOMETRY_EPSILON(long long, 0);
} // namespace geo
} // namespace lib

#endif
```

6.4. Line2D

```
#ifndef _LIB_GEOMETRY_LINE_2D
#define _LIB_GEOMETRY_LINE_2D
#include "GeometryEpsilon.cpp"
#include "Trigonometry.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
```

```cpp
namespace geo {
namespace plane {
namespace {
template <typename T> bool scalar_between(T a, T o, T b) {
  if (a > b)
    swap(a, b);
  return GEOMETRY_COMPARE(T, a, o) <= 0 && GEOMETRY_COMPARE(T, o, b) <= 0;
}

template <typename T> bool scalar_strictly_between(T a, T o, T b) {
  if (a > b)
    swap(a, b);
  int x = GEOMETRY_COMPARE(T, a, o);
  int y = GEOMETRY_COMPARE(T, o, b);
  return x <= 0 && y <= 0 && (x < 0 || y < 0);
}
} // namespace

template <typename T, typename Large = T> struct Point {
  T x, y;
  Point() : x(0), y(0) {}
  Point(T x, T y) : x(x), y(y) {}
  template <typename G, typename H> explicit operator Point<G, H>() const {
    return Point<G, H>((G)x, (G)y);
  }
  friend Point reversed(const Point &a) { return Point(a.y, a.x); }
  Point &operator+=(const Point &rhs) {
    x += rhs.x, y += rhs.y;
    return *this;
  }
  Point &operator-=(const Point &rhs) {
    x -= rhs.x, y -= rhs.y;
    return *this;
  }
  Point &operator*=(T k) {
    x *= k, y *= k;
    return *this;
  }
  Point &operator/=(T k) {
    x /= k, y /= k;
    return *this;
  }
  Point operator+(const Point &rhs) const {
    Point res = *this;
    return res += rhs;
  }
  Point operator-(const Point &rhs) const {
    Point res = *this;
    return res -= rhs;
  }
  Point operator*(T k) const {
    Point res = *this;
    return res *= k;
  }
  Point operator/(T k) const {
    Point res = *this;
    return res /= k;
  }
  Point operator-() const { return Point(-x, -y); }
  inline friend Point convolve(const Point &a, const Point &b) {
    return Point(a.x * b.x - a.y * b.y, a.x * b.y + b.x * a.y);
  }
  inline friend Large cross(const Point &a, const Point &b) {
    return (Large)a.x * b.y - (Large)a.y * b.x;
  }
  friend Large cross(const Point &a, const Point &b, const Point &c) {
    return cross(b - a, c - a);
  }
  inline friend Large dot(const Point &a, const Point &b) {
    return (Large)a.x * b.x + (Large)a.y * b.y;
  }
  friend int ccw(const Point &u, const Point &v) {
    return GEOMETRY_COMPARE0(Large, cross(u, v));
  }
  friend int ccw(const Point &a, const Point &b, const Point &c) {
    return ccw(b - a, c - a);
  }
  friend int half_ccw(const Point& u, const Point& v) {
    int dot_sgn = GEOMETRY_COMPARE0(Large, dot(u, v));
    int ccw_sgn = ccw(u, v);
    if(dot_sgn == 0) return ccw_sgn ? 1 : 0;
    return dot_sgn * ccw_sgn;
  }
  friend Large norm(const Point &a) { return sqrtl(dot(a, a)); }
  friend Large norm_sq(const Point &a) { return dot(a, a); }
  bool is_null() const { return GEOMETRY_COMPARE0(Large, norm_sq(*this)) ==
    0; }
  bool is_versor() const {
    return GEOMETRY_COMPARE(Large, norm_sq(*this), (Large)1) == 0;
  }
  static Point polar(Large d, Large theta) {
    return Point(trig::cos(theta) * d, trig::sin(theta) * d);
  }
  friend Point rotate(const Point &a, Large theta) {
    return convolve(a, polar((Large)1, theta));
  }
  friend Point ortho(const Point &a) { return Point(-a.y, a.x); }
  friend Large arg(const Point &a) { return trig::atan2(a.y, a.x); }
  friend Large signed_angle(const Point &v, const Point &w) {
    return remainder(arg(w) - arg(v), 2.0 * trig::PI);
  }
  friend Large angle(const Point &v, const Point &w) {
    return abs(signed_angle(v, w));
  }
  friend Large ccw_angle(const Point &v) {
    Large res = arg(v);
    if (res < 0)
      res += 2.0 * trig::PI;
    return res;
  }
  friend Large ccw_angle(const Point &v, const Point &w) {
    Large res = signed_angle(v, w);
    if (res < 0)
      res += 2.0 * trig::PI;
    return res;
  }
  inline friend Point normalized(const Point &a, Large k) {
    return a.is_null() ? Point() : a / norm(a) * k;
  }
  inline friend Point versor(const Point &a) { return normalized(a,
    (Large)1); }
  friend bool collinear(const Point &a, const Point &b) {
    return GEOMETRY_COMPARE0(Large, cross(a, b)) == 0;
  }
  friend bool collinear(const Point &a, const Point &b, const Point &c) {
    return collinear(b - a, c - a);
  }
  friend Point project(const Point &a, const Point &v) {
    return v / norm_sq(v) * dot(a, v);
  }
```

```
  template <typename G = T,
            typename enable_if<!is_integral<G>::value>::type * = nullptr>
  friend Point reflect(const Point &a, const Point &v) {
    Point n = versor(v);
    return a - n * 2 * dot(n, v);
  }
  friend bool between(const Point &a, const Point &b, const Point &c) {
    return collinear(a, b, c) &&
           GEOMETRY_COMPARE0(Large, dot(a - b, c - b)) <= 0;
  }
  friend bool strictly_between(const Point &a, const Point &b, const Point &c) {
    return collinear(a, b, c) &&
           GEOMETRY_COMPARE0(Large, dot(a - b, c - b)) < 0;
  }
  friend bool collinear_between(const Point a, const Point &o, const Point &b) {
    return scalar_between(a.x, o.x, b.x) && scalar_between(a.y, o.y, b.y);
  }
  friend bool collinear_strictly_between(const Point &a, const Point &o,
                                         const Point &b) {
    return scalar_between(a.x, o.x, b.x) && scalar_between(a.y, o.y, b.y);
  }
  friend Large dist(const Point &a, const Point &b) { return norm(a - b); }
  friend bool operator==(const Point &a, const Point &b) {
    return GEOMETRY_COMPARE(T, a.x, b.x) == 0 &&
           GEOMETRY_COMPARE(T, a.y, b.y) == 0;
  }
  friend bool operator!=(const Point &a, const Point &b) { return !(a == b); }
  friend bool operator<(const Point &a, const Point &b) {
    return tie(a.y, a.x) < tie(b.y, b.x);
  }
  friend bool operator>(const Point &a, const Point &b) {
    return tie(a.y, a.x) > tie(b.y, b.x);
  }
  friend bool operator>=(const Point &a, const Point &b) {
    return tie(a.y, a.x) >= tie(b.y, b.x);
  }
  friend bool operator<=(const Point &a, const Point &b) {
    return tie(a.y, a.x) <= tie(b.y, b.x);
  }
  friend istream &operator>>(istream &in, Point &p) { return in >> p.x >> p.y; }
  friend ostream &operator<<(ostream &out, const Point &p) {
    return out << p.x << " " << p.y;
  }
};

template <typename T, typename Large = T> struct Rectangle {
  typedef Point<T, Large> point;

  T minx, miny, maxx, maxy;
  Rectangle() {
    minx = miny = numeric_limits<T>::max();
    maxx = maxy = numeric_limits<T>::min();
  }

  Rectangle(const initializer_list<point> &points) : Rectangle() {
    for (const auto &p : points) {
      minx = min(minx, p.x);
      maxx = max(maxx, p.x);
      miny = min(miny, p.y);
      maxy = max(maxy, p.y);
    }
  }

  bool contains(const point &p) const {
    return GEOMETRY_COMPARE(T, minx, p.x) <= 0 &&
           GEOMETRY_COMPARE(T, p.x, maxx) <= 0 &&
           GEOMETRY_COMPARE(T, miny, p.y) <= 0 &&
           GEOMETRY_COMPARE(T, p.y, maxy) <= 0;
  }
};

template <typename T, typename Large = T> struct Line {
  typedef Point<T, Large> point;
  typedef Line<T, Large> line;
  point a, b;
  Line(point a, point b) : a(a), b(b) {}
  template <typename G = T,
            typename enable_if<!is_integral<G>::value>::type * = nullptr>
  Line(T A, T B, T C) {
    if (GEOMETRY_COMPARE0(Large, A))
      a = point(-C / A, 0), b = point((-C - B) / A, 1);
    else if (GEOMETRY_COMPARE0(Large, B))
      a = point(0, -C / B), b = point(1, (-C - A) / B);
    else
      assert(false);
  }
  template <typename G, typename H> explicit operator Line<G, H>() const {
    return Line<G, H>(Point<G, H>(a), Point<G, H>(b));
  }
  point direction() const { return b - a; }
  friend point project(const point &p, const line &v) {
    return project(p - v.a, v.b - v.a) + v.a;
  }
  friend bool collinear(const line &u, const line &v) {
    return collinear(u.a, u.b, v.a) && collinear(u.a, u.b, v.b);
  }
  bool contains(const point &p) const { return collinear(a, b, p); }
  friend bool parallel(const line &u, const line &v) {
    return collinear(u.b - u.a, v.b - v.a);
  }
  friend bool opposite(const line &l, const point &p1, const point &p2) {
    int x = GEOMETRY_COMPARE0(Large, cross(p1 - l.a, l.direction()));
    int y = GEOMETRY_COMPARE0(Large, cross(p2 - l.a, l.direction()));
    return x * y <= 0;
  }
  friend pair<point, bool> intersect(const line &l1, const line &l2) {
    Large c1 = cross(l2.a - l1.a, l1.b - l1.a);
    Large c2 = cross(l2.b - l1.a, l1.b - l1.a);
    if (GEOMETRY_COMPARE0(Large, c1 - c2) == 0)
      return {{}, false};
    return {(l2.b * c1 - l2.a * c2) / (c1 - c2), true};
  }
  friend bool has_unique_intersection(const line &l1, const line &l2) {
    return !parallel(l1, l2);
  }
  friend bool has_intersection(const line &l1, const line &l2) {
    return collinear(l1, l2) || has_unique_intersection(l1, l2);
  }
  friend Large dist(const line &l1, const point &p) {
    // TODO: improve this
    return dist(p, project(p, l1));
  }
  friend Large dist(const line &l1, const line &l2) {
    if (has_intersection(l1, l2))
      return 0;
    // TODO: improve this
```

```
    return dist(l1.a, project(l1.a, l2));
  }
};

template <typename T, typename Large = T> struct Ray {
  typedef Point<T, Large> point;
  typedef Line<T, Large> line;
  typedef Ray<T, Large> ray;
  point a, b;

  Ray(point a, point direction) : a(a), b(a + direction) {}

  static ray from_points(point a, point b) { return ray(a, b - a); }
  point direction() const { return b - a; }
  point direction_versor() const { return versor(direction()); }

  line as_line() const { return line(a, b); }
  explicit operator line() const { return as_line(); }

  template <typename G, typename H> explicit operator Ray<G, H>() const {
    return Ray<G, H>(Point<G, H>(a), Point<G, H>(b));
  }
  bool contains(const point &p) const {
    return collinear(a, b, p) &&
           GEOMETRY_COMPARE0(Large, dot(p - a, b - a)) >= 0;
  }
  bool strictly_contains(const point &p) const {
    return collinear(a, b, p) &&
           GEOMETRY_COMPARE0(Large, dot(p - a, b - a)) > 0;
  }
  bool collinear_contains(const point &p) const {
    point dir = direction();
    int dx = GEOMETRY_COMPARE0(T, dir.x);
    if (dx == 0)
      return GEOMETRY_COMPARE0(T, dir.y) * GEOMETRY_COMPARE0(T, p.y - a.y)
    >= 0;
    else
      return dx * GEOMETRY_COMPARE0(T, p.x - a.x) >= 0;
  }
  bool collinear_strictly_contains(const point &p) const {
    point dir = direction();
    int dx = GEOMETRY_COMPARE0(T, dir.x);
    if (dx == 0)
      return GEOMETRY_COMPARE0(T, dir.y) * GEOMETRY_COMPARE0(T, p.y - a.y) >
    0;
    else
      return dx * GEOMETRY_COMPARE0(T, p.x - a.x) > 0;
  }
  friend pair<point, bool> intersect(const ray &r, const line &l) {
    auto p = intersect(r.as_line(), l);
    if (!p.second)
      return {{}, false};
    if (!r.collinear_contains(p.first))
      return {{}, false};
    return p;
  }
  friend pair<point, bool> intersect(const ray &a, const ray &b) {
    auto p = intersect(a, b.as_line());
    if (!p.second)
      return {{}, false};
    if (!b.collinear_contains(p.first))
      return {{}, false};
    return p;
  }
  friend bool has_unique_intersection(const ray &r, const line &l) {
    if (!has_unique_intersection(r.as_line(), l))
      return false;
    int x = GEOMETRY_COMPARE0(Large, cross(r.direction(), l.direction()));
    int y = GEOMETRY_COMPARE0(Large, cross(r.a - l.a, l.direction()));
    return x * y <= 0;
  }
  friend bool has_intersection(const ray &r, const line &l) {
    return collinear(r.as_line(), l) || has_unique_intersection(r, l);
  }
  friend bool has_unique_intersection(const ray &r1, const ray &r2) {
    // TODO: not efficient
    return has_unique_intersection(r1, r2.as_line()) &&
           has_unique_intersection(r2, r1.as_line());
  }
  friend bool has_intersection(const ray &r1, const ray &r2) {
    return r1.contains(r2.a) || has_unique_intersection(r1, r2);
  }
  friend Large dist(const ray &r, const point &p) {
    if (GEOMETRY_COMPARE0(Large, dot(r.direction(), p - r.a)) < 0)
      return dist(p, r.a);
    return dist(r.as_line(), p);
  }
  friend Large dist(const ray &r, const line &l) {
    if (has_intersection(r, l))
      return Large(0);
    return dist(l, r.a);
  }
  friend Large dist(const ray &r1, const ray &r2) {
    if (has_intersection(r1, r2))
      return Large(0);
    return min(dist(r1, r2.a), dist(r2, r1.a));
  }
};

template <typename T, typename Large = T> struct Halfplane {
  typedef Point<T, Large> point;
  typedef Line<T, Large> line;
  typedef Ray<T, Large> ray;
  typedef Halfplane<T, Large> halfplane;
  point a, b;

  Halfplane(point a, point direction) : a(a), b(a + direction) {}

  static halfplane from_points(point a, point b) { return halfplane(a, b -
    a); }
  point direction() const { return b - a; }
  point direction_versor() const { return versor(direction()); }

  line as_line() const { return line(a, b); }
  explicit operator line() const { return as_line(); }

  ray as_ray() const { return ray(a, b); }
  explicit operator ray() const { return as_ray(); }

  template <typename G, typename H> explicit operator Halfplane<G, H>()
    const {
    return Halfplane<G, H>(Point<G, H>(a), Point<G, H>(b));
  }

  bool contains(const point& p) const {
    return ccw(a, b, p) <= 0;
  }
  bool strictly_contains(const point& p) const {
    return ccw(a, b, p) < 0;
  }
```

```
};

template <typename T, typename Large = T> struct Segment {
  typedef Point<T, Large> point;
  typedef Line<T, Large> line;
  typedef Segment<T, Large> segment;
  typedef Ray<T, Large> ray;
  point a, b;

  Segment() {}
  Segment(point a, point b) : a(a), b(b) {}
  line as_line() const { return line(a, b); }
  explicit operator line() const { return as_line(); }
  bool is_degenerate() const { return a == b; }

  template <typename G, typename H> explicit operator Segment<G, H>() const {
    return Segment<G, H>(Point<G, H>(a), Point<G, H>(b));
  }
  bool contains(const point &p) const { return between(a, p, b); }
  bool strictly_contains(const point &p) const {
    return strictly_between(a, p, b);
  }
  bool collinear_contains(const point &p) const {
    return collinear_between(a, p, b);
  }
  bool collinear_strictly_contains(const point &p) const {
    return collinear_strictly_between(a, p, b);
  }
  friend pair<point, bool> intersect(const segment &s, const line &l) {
    auto p = intersect(s.as_line(), l);
    if (!p.second)
      return {{}, false};
    if (!s.collinear_contains(p.first))
      return {{}, false};
    return p;
  }
  friend pair<point, bool> intersect(const segment &s, const ray &r) {
    auto p = intersect(s.as_line(), r.as_line());
    if (!p.second)
      return {{}, false};
    if (!s.collinear_contains(p.first) || !r.collinear_contains(p.first))
      return {{}, false};
    return p;
  }
  friend pair&lt;segment, int&gt; intersect_segment(segment s1, segment s2) {
    if (collinear(s1.as_line(), s2.as_line())) {
      if (s1.a > s1.b)
        swap(s1.a, s1.b);
      if (s2.a > s2.b)
        swap(s2.a, s2.b);
      segment res(max(s1.a, s2.a), min(s1.b, s2.b));
      return {res, int(res.a <= res.b) * 2};
    } else {
      auto p = intersect(s1, s2);
      return {segment(p.first, p.first), p.second};
    }
  }
  friend pair<point, bool> intersect(const segment &s1, const segment &s2) {
    auto p = intersect(s1, s2.as_line());
    if (!p.second)
      return {{}, false};
    if (!s2.collinear_contains(p.first))
      return {{}, false};
    return p;
  }
  friend bool has_unique_intersection(const segment &s, const line &l) {
    if (!has_unique_intersection(s.as_line(), l))
      return false;
    return opposite(l, s.a, s.b);
  }
  friend bool has_intersection(const segment &s, const line &l) {
    return collinear(s.as_line(), l) || has_unique_intersection(s, l);
  }
  friend bool has_unique_intersection(const segment &s, const ray &r) {
    if (!has_unique_intersection(r, s.as_line()))
      return false;
    return opposite(r.as_line(), s.a, s.b);
  }
  friend bool has_intersection(const segment &s, const ray &r) {
    return r.contains(s.a) || r.contains(s.b) || has_unique_intersection(s,
      r);
  }
  friend bool has_unique_intersection(const segment &s1, const segment &s2) {
    if (!has_unique_intersection(s1.as_line(), s2.as_line()))
      return false;
    return opposite(s2.as_line(), s1.a, s1.b) &&
           opposite(s1.as_line(), s2.a, s2.b);
  }
  friend bool has_intersection(const segment &s1, const segment &s2) {
    return s1.contains(s2.a) || s1.contains(s2.b) ||
           has_unique_intersection(s1, s2);
  }
  friend Large dist(const segment &s, const point &p) {
    if (GEOMETRY_COMPARE0(Large, dot(p - s.a, s.b - s.a)) <= 0)
      return dist(s.a, p);
    if (GEOMETRY_COMPARE0(Large, dot(p - s.b, s.a - s.b)) <= 0)
      return dist(s.b, p);
    return dist(s.as_line(), p);
  }
  friend Large dist(const segment &s, const line &l) {
    if (has_intersection(s, l))
      return Large(0);
    return min(dist(l, s.a), dist(l, s.b));
  }
  friend Large dist(const segment &s, const ray &r) {
    if (has_intersection(s, r))
      return Large(0);
    return min({dist(r, s.a), dist(r, s.b), dist(s, r.a)});
  }
  friend Large dist(const segment &s1, const segment &s2) {
    if (has_intersection(s1, s2))
      return Large(0);
    return min(
        {dist(s1, s2.a), dist(s1, s2.b), dist(s2, s1.a), dist(s2, s1.b)});
  }

  friend bool operator==(const segment &l1, const segment &l2) {
    return tie(l1.a, l1.b) == tie(l2.a, l2.b);
  }
  friend bool operator!=(const segment &l1, const segment &l2) {
    return !(l1 == l2);
  }
  friend bool operator<(const segment &l1, const segment &l2) {
    return tie(l1.a, l1.b) < tie(l2.a, l2.b);
  }
};

template <typename Direction, typename T, typename Large> struct
    AngleComparator {
  using type = typename Direction::type;
```

```cpp
  using point = Point<T, Large>;

  Direction dir;
  AngleComparator() {}
  AngleComparator(Direction dir) : dir(dir) {}
  bool operator()(const type &a, const type &b) const {
    return ccw(dir(a), dir(b)) > 0;
  }
  template <typename Iterator>
  static void sortByAngle(Iterator begin, Iterator end, const Direction& dir = Direction()) {
    AngleComparator cmp(dir);
    begin =
        partition(begin, end, [&dir](const type &p) { return dir(p).is_null(); });
    auto half =
        partition(begin, end, [&dir](const type &p) { return dir(p) > point(); });
    sort(begin, half, cmp);
    sort(half, end, cmp);
  }
  template <typename Iterator>
  static Iterator minByAngle(Iterator begin, Iterator end, const Direction& dir = Direction()) {
    AngleComparator cmp(dir);
    return min_element(begin, end, [&dir, &cmp](const type& a, const type& b) {
      bool part_a = dir(a) > point();
      bool part_b = dir(b) > point();
      if(part_a == part_b)
        return cmp(a, b);
      return part_a > part_b;
    });
  }
};
template <typename Ray> struct RayDirection {
  using point = typename Ray::point;
  using type = Ray;
  point operator()(const type& rhs) const {
    return rhs.direction();
  }
};
template <typename Point> struct PointDirection {
  using type = Point;
  Point pivot;
  PointDirection() : pivot() {}
  PointDirection(Point pivot) : pivot(pivot) {}
  Point operator()(const Point& rhs) const {
    return (rhs - pivot).direction();
  }
};
} // namespace plane

template <typename T, typename Large = T> struct CartesianPlane {
  typedef plane::Point<T, Large> point;
  typedef plane::Line<T, Large> line;
  typedef plane::Rectangle<T, Large> rectangle;
  typedef plane::Segment<T, Large> segment;
  typedef plane::Ray<T, Large> ray;
  typedef plane::Halfplane<T, Large> halfplane;

  template<typename Direction>
  using angle_comparator = plane::AngleComparator<Direction, T, Large>;
};

} // namespace geo
} // namespace lib

#endif
```

6.5. Polygon2D

```cpp
#ifndef _LIB_GEOMETRY_POLY_2D
#define _LIB_GEOMETRY_POLY_2D
#include "Circle2D.cpp"
#include "Line2D.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace geo {
namespace plane {

template <typename T, typename Large = T> struct ConvexHullComparator {
  typedef Point<T, Large> point;
  point pivot;
  ConvexHullComparator(point p) : pivot(p) {}
  template <typename G>
  bool operator()(const pair<point, G> &a, const pair<point, G> &b) const {
    int k = ccw(pivot, a.first, b.first);
    if (k == 0)
      return norm_sq(a.first) < norm_sq(b.first);
    return k > 0;
  }
};

template <typename T, typename Large = T> struct Polygon {
  typedef Point<T, Large> point;
  typedef Polygon<T, Large> polygon;
  typedef Circle<T, Large> circle;
  vector<point> p;

  Polygon() {}
  Polygon(const vector<point> &p) : p(p) {}
  template <typename G> Polygon(const vector<pair<point, G>> &g) : p(g.size()) {
    for (size_t i = 0; i < g.size(); i++)
      p[i] = g[i].first;
  }
  template <typename A, typename B> explicit operator Polygon<A, B>() const {
    vector<Point<A, B>> v(p.size());
    for (size_t i = 0; i < p.size(); i++)
      v[i] = Point<A, B>(p[i]);
    return Polygon<A, B>(v);
  }
  inline int index(int i) const {
    if (i >= size())
      i %= size();
    else if (i < 0) {
      i %= size();
      if (i < 0)
        i += size();
    }
    return i;
  }
  inline int size() const { return p.size(); }
  inline point &operator[](int i) { return p[index(i)]; }
  inline point operator[](int i) const { return p[index(i)]; }
  void erase(int i) { p.erase(p.begin() + index(i)); }
```

```
  polygon &operator+=(const point &pt) {
    for (auto &q : p)
      q += pt;
    return *this;
  }
  polygon &operator-=(const point &pt) {
    for (auto &q : p)
      q -= pt;
    return *this;
  }
  polygon &operator*=(const Large k) {
    for (auto &q : p)
      q *= k;
    return *this;
  }
  polygon &operator/=(const Large k) {
    for (auto &q : p)
      q /= k;
    return *this;
  }
  polygon operator-() const {
    polygon res = *this;
    for (auto &q : res.p)
      q = -q;
    return res;
  }
  void reserve(int n) { p.reserve(n); }
  bool is_ccw() const {
    int n = size();
    int i = min_element(p.begin(), p.end()) - p.begin();
    return ccw(p[i], p[i + 1], p[i - 1]) >= 0;
  }
  bool is_degenerate() const {
    int n = size();
    if (n < 3)
      return true;
    for (int i = 0; i < n; i++) {
      if (GEOMETRY_COMPARE0(Large, cross(p[i + 2] - p[i], p[i + 1] - p[i])))
        return false;
    }
    return true;
  }
  inline operator vector<point>() const { return p; }

  friend Large double_area(const Polygon &p) {
    int n = p.size();
    Large res = 0;
    for (int i = 0; i < n; i++) {
      res += cross(p[i], p[i + 1]);
    }
    return abs(res);
  }
  friend Large area(const Polygon &p) { return double_area(p) / 2; }
  friend Large perimeter(const Polygon &p) {
    int n = p.size();
    Large res = 0;
    for (int i = 0; i < n; i++)
      res += dist(p[i], p[i + 1]);
    return res;
  }

  int test(const point &p) const {
    const Polygon &poly = *this;
    int n = size();
    int wn = 0;
    for (int i = 0; i < n; i++) {
      if (p == poly[i])
        return 0;
      int j = i + 1;
      if (poly[i].y == p.y && poly[j].y == p.y) {
        if (min(poly[i].x, poly[j].x) <= p.x &&
            p.x <= max(poly[i].x, poly[j].x))
          return 0;
      } else {
        bool below = poly[i].y < p.y;
        if (below != (poly[j].y < p.y)) {
          auto sig = ccw(poly[i], poly[j], p);
          if (sig == 0)
            return 0;
          if (below == (sig > 0))
            wn += below ? 1 : -1;
        }
      }
    }
    return wn == 0 ? 1 : -1;
  }

  template <typename G>
  static vector<pair<point, G>> convex_hull(vector<pair<point, G>> p,
                                            bool keep_border = false) {
    if (p.size() <= 1)
      return p;
    sort(p.begin(), p.end());
    vector<pair<point, G>> res;
    res.reserve(p.size() + 1);
    for (int step = 0; step < 2; step++) {
      auto start = res.size();
      for (auto &q : p) {
        while (res.size() >= start + 2) {
          int sig = ccw(res[res.size() - 2].first, res.back().first, q.first);
          if ((sig == 0 && !keep_border) || sig < 0)
            res.pop_back();
          else
            break;
        }
        res.push_back(q);
      }
      res.pop_back();
      if (step == 0)
        reverse(p.begin(), p.end());
    }
    if (res.size() == 2 && res[0] == res[1])
      res.pop_back();
    return res;
  }

  static polygon convex_hull(const vector<point> &p, bool keep_border = false) {
    vector<pair<point, int>> v(p.size());
    for (size_t i = 0; i < p.size(); i++)
      v[i] = {p[i], i};
    auto res = convex_hull(v, keep_border);
    return polygon(res);
  }

  friend vector<polygon> triangulation(polygon poly) {
    if (poly.size() < 3)
      return {};
    vector<polygon> res;
```

```
    int ptr = 0;
    int n;
    while ((n = poly.size()) > 3) {
      for (int &i = ptr;; i++) {
        if (ccw(poly[i - 1], poly[i], poly[i + 1]) > 0) {
          auto trig = polygon({poly[i - 1], poly[i], poly[i + 1]});
          bool good = true;
          for (int j = 0; j < n; j++) {
            good &= trig.test(poly[j]) >= 0;
          }
          if (!good)
            continue;
          poly.erase(i--);
          res.push_back(trig);
          break;
        }
      }
    }
    res.push_back(poly);
    return res;
  }

  friend Large intersection_area(const Polygon &p, const circle &C) {
    Large res = 0;
    int n = p.size();
    for (int i = 0; i < n; i++) {
      res += circle::intersection_signed_area(C.radius, p[i + 1] - C.center,
                                              p[i] - C.center);
    }
    return abs(res);
  }
};

template <typename T, typename Large = T>
struct ConvexPolygon : public Polygon<T, Large> {
  typedef Point<T, Large> point;
  typedef Segment<T, Large> segment;
  typedef Line<T, Large> line;
  typedef Halfplane<T, Large> halfplane;
  typedef Circle<T, Large> circle;
  typedef AngleComparator<PointDirection<point>, T, Large> angle_comparator;
  using Polygon<T, Large>::p;
  int top;
  ConvexPolygon() {}
  ConvexPolygon(const vector<point> &p) : Polygon<T, Large>(p) {
    normalize(); }
  template <typename G>
  ConvexPolygon(const vector<pair<point, G>> &p) : Polygon<T, Large>(p) {
    normalize();
  }
  void normalize() {
    auto bottom = min_element(p.begin(), p.end());
    rotate(p.begin(), bottom, p.end());
    top = max_element(p.begin(), p.end()) - p.begin();
  }
  ConvexPolygon &operator+=(const point &pt) {
    for (auto &q : p)
      q += pt;
    return *this;
  }
  ConvexPolygon &operator-=(const point &pt) {
    for (auto &q : p)
      q -= pt;
    return *this;
  }
  ConvexPolygon &operator*=(const Large k) {
    for (auto &q : p)
      q *= k;
    return *this;
  }
  ConvexPolygon &operator/=(const Large k) {
    for (auto &q : p)
      q /= k;
    return *this;
  }
  ConvexPolygon operator-() const {
    ConvexPolygon res = *this;
    for (auto &q : res.p)
      q = -q;
    return res;
  }

  int test(const point &q) const {
    if (q < p[0] || q > p[top])
      return 1;
    auto sig = ccw(p[0], p[top], q);
    if (sig == 0) {
      if (q == p[0] || q == p[top])
        return 0;
      return top == 1 || top + 1 == this->size() ? 0 : -1;
    } else if (sig < 0) {
      auto it = lower_bound(p.begin() + 1, p.begin() + top, q);
      return ccw(it[-1], q, it[0]);
    } else {
      auto it = upper_bound(p.rbegin(), p.rend() - top - 1, q);
      auto pit_deref = it == p.rbegin() ? p[0] : it[-1];
      return ccw(*it, q, pit_deref);
    }
  }
  template <typename Function> int extreme(Function direction) const {
    int n = this->size(), left = 0, leftSig;
    const ConvexPolygon &poly = *this;
    auto vertex_cmp = [&poly, direction](int i, int j) {
      return ccw(poly[j] - poly[i], direction(poly[j]));
    };
    auto is_extreme = [n, vertex_cmp](int i, int &iSig) {
      return (iSig = vertex_cmp(i + 1, i)) >= 0 && vertex_cmp(i, i - 1) < 0;
    };
    for (int right = is_extreme(0, leftSig) ? 1 : n; left + 1 < right;) {
      int mid = (left + right) / 2, midSig;
      if (is_extreme(mid, midSig))
        return mid;
      if (leftSig != midSig ? leftSig < midSig
                          : leftSig == vertex_cmp(left, mid))
        right = mid;
      else
        left = mid, leftSig = midSig;
    }
    return poly.index(left);
  }
  void stab_extremes(const line &l, int &left, int &right) const {
    point direction = l.direction();
    right = extreme([&direction](const point &) { return direction; });
    left = extreme([&direction](const point &) { return -direction; });
  }
  friend vector<point> intersect(const ConvexPolygon &poly, const line &l) {
    point direction = l.direction();

    int left, right;
    poly.stab_extremes(l, left, right);
```

```
    auto vertex_cmp = [&l, &direction](const point &q) {
      return ccw(q - l.a, direction);
    };
    int rightSig = vertex_cmp(poly[right]), leftSig = vertex_cmp(poly[left]);
    if (rightSig < 0 || leftSig > 0)
      return {};
    auto intersectChain = [&l, &poly, vertex_cmp](int first, int last,
                                                   int firstSig) {
      int n = poly.size();
      while (poly.index(first + 1) != poly.index(last)) {
        int mid = (first + last + (first < last ? 0 : n)) / 2;
        mid = poly.index(mid);
        if (vertex_cmp(poly[mid]) == firstSig)
          first = mid;
        else
          last = mid;
      }
      return intersect(l, line(poly[first], poly[last]));
    };
    return {intersectChain(left, right, leftSig).first,
            intersectChain(right, left, rightSig).first};
  }
  friend bool has_intersection(const ConvexPolygon &p, const line &l) {
    point direction = l.direction();
    int left, right;
    p.stab_extremes(l, left, right);
    auto vertex_cmp = [&l, &direction](const point &q) {
      return ccw(q - l.a, direction);
    };
    int rightSig = vertex_cmp(p[right]), leftSig = vertex_cmp(p[left]);
    if (rightSig < 0 || leftSig > 0)
      return false;
    return true;
  }
  friend Large dist(const ConvexPolygon &p, const line &l) {
    point direction = l.direction();
    int left, right;
    p.stab_extremes(l, left, right);
    auto vertex_cmp = [&l, &direction](const point &q) {
      return ccw(q - l.a, direction);
    };
    int rightSig = vertex_cmp(p[right]), leftSig = vertex_cmp(p[left]);
    if (rightSig < 0 || leftSig > 0) {
      return min(dist(l, p[right]), dist(l, p[left]));
    } else {
      return 0;
    }
  }
  template <typename Function>
  friend void antipodals(const ConvexPolygon &poly, Function f) {
    if (poly.size() <= 1)
      return;
    if (poly.size() == 2)
      return void(f(0, 1));
    auto area = [&poly](int i, int j, int k) {
      return abs(cross(poly[i], poly[j], poly[k]));
    };
    auto func = [f, &poly](int i, int j) {
      return f(poly.index(i), poly.index(j));
    };

    int p = -1;
    int q = 0;
    while (area(p, p + 1, q + 1) > area(p, p + 1, q))
      q++;
    int p0 = 0;
    int q0 = q;
    while (poly.index(q) != p0) {
      p++;
      func(p, q);
      while (area(p, p + 1, q + 1) > area(p, p + 1, q)) {
        q++;
        if (poly.index(p) != poly.index(q0) || poly.index(q) != p0)
          func(p, q);
        else
          return;
      }
      if (area(p, p + 1, q + 1) == area(p, p + 1, q)) {
        if (poly.index(p) != poly.index(q0) || poly.index(q) != p0)
          func(p, q + 1);
        else
          func(p + 1, q);
      }
    }
  }
  friend ConvexPolygon minkowski_sum(const vector<ConvexPolygon> &v) {
    vector<point> vectors;
    point origin;
    for (auto &poly : v) {
      origin += poly[0];
      for (int i = 0; i < poly.size(); i++)
        vectors.push_back(poly[i + 1] - poly[i]);
    }
    angle_comparator::sortByAngle(vectors.begin(), vectors.end());
    auto last = point();
    if (!vectors.empty()) {
      last = vectors.back();
      vectors.pop_back();
    }
    vector<point> res;
    res.push_back(origin);
    for (auto &v : vectors) {
      res.push_back(res.back() + v);
      int n = res.size();
      if (n >= 3 && collinear(res[n - 3], res[n - 2], res[n - 1]))
        res.erase(res.begin() + n - 2);
    }
    int n = res.size();
    if (n >= 3 && collinear(res[n - 2], res[n - 1], res[0]))
      res.pop_back();
    if (res.size() >= 3 && collinear(res.back(), res[0], res[1]))
      res.erase(res.begin());
    return ConvexPolygon(res);
  }
  friend ConvexPolygon minkowski_sum(const ConvexPolygon &a,
                                     const ConvexPolygon &b) {
    vector<ConvexPolygon> v;
    v.push_back(a);
    v.push_back(b);
    return minkowski_sum(v);
  }
  friend ConvexPolygon intersect(const ConvexPolygon &a,
                                 const ConvexPolygon &b) {
    vector<point> candidates;
    auto consider = [&candidates](const ConvexPolygon &a,
                                  const ConvexPolygon &b) {
      for (int i = 0; i < a.size(); i++) {
        if (b.test(a[i]) <= 0)
          candidates.push_back(a[i]);
        segment s(a[i], a[i + 1]);
```

```
      vector<point> ps = intersect(b, s.as_line());
      for (auto p : ps) {
        if (s.contains(p))
          candidates.push_back(p);
      }
    }
  };
  consider(a, b);
  consider(b, a);
  auto res = ConvexPolygon(ConvexPolygon::convex_hull(candidates));
  return res;
}
friend Large intersection_area_or_dist(const ConvexPolygon &a,
                                       const ConvexPolygon &b) {
  ConvexPolygon inter = intersect(a, b);
  if (inter.size() > 0)
    return max(area(inter), Large(0));
  ConvexPolygon sum = minkowski_sum(a, -b);
  Large res = numeric_limits<Large>::max();
  for (int i = 0; i < sum.size(); i++) {
    res = min(res, dist(segment(sum[i], sum[i + 1]), point()));
  }
  return -res;
}
void cut(const halfplane& pl) {
  int n = this->size();
  if(n < 3) return;
  p.push_back(p[0]);

  auto pl_line = pl.as_line();

  vector<point> out;
  bool inside = pl.strictly_contains(p[0]);
  if(inside) out.push_back(p[0]);

  for(int i = 1; i <= n; i++) {
    if(pl.strictly_contains(p[i])) {
      if(!inside) {
        out.push_back(intersect(pl_line, line(p[i-1], p[i])).first);
      }
      out.push_back(p[i]);
      inside = true;
    } else {
      if(inside) {
        out.push_back(intersect(pl_line, line(p[i-1], p[i])).first);
      }
      inside = false;
    }
  }

  if(!out.empty() && out[0] == out.back()) out.pop_back();
  *this = ConvexPolygon(ConvexPolygon::convex_hull(out));
}
void cut(const ConvexPolygon &rhs) {
  for(int i = 0; i < rhs.size(); i++) {
    cut(halfplane::from_points(rhs[i], rhs[i+1]));
  }
}
};
} // namespace plane

template <typename T, typename Large = T>
struct PolygonPlane : public CirclePlane<T, Large> {
  typedef plane::Polygon<T, Large> polygon;
  typedef plane::ConvexPolygon<T, Large> convex_polygon;
};

} // namespace geo
} // namespace lib

#endif
```

6.6. Trigonometry

```
#ifndef _LIB_TRIGONOMETRY
#define _LIB_TRIGONOMETRY
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace geo {
namespace trig {
constexpr static long double PI =
    3.141592653589793238462643383279502884197169399375105820974944l;
double cos(double x) { return ::cos(x); }
double sin(double x) { return ::sin(x); }
double asin(double x) { return ::asin(x); }
double acos(double x) { return ::acos(x); }
double atan2(double y, double x) { return ::atan2(y, x); }
long double cos(long double x) { return ::cosl(x); }
long double sin(long double x) { return ::sinl(x); }
long double asin(long double x) { return ::asinl(x); }
long double acos(long double x) { return ::acosl(x); }
long double atan2(long double y, long double x) { return ::atan2l(y, x); }
} // namespace trig
} // namespace geo
} // namespace lib

#endif
```

7. polynomial

7.1. ExponentialSum

```
#ifndef _LIB_EXPONENTIAL_SUM
#define _LIB_EXPONENTIAL_SUM
#include <bits/stdc++.h>
#include "../Combinatorics.cpp"
#include "../Lagrange.cpp"

namespace lib {
using namespace std;

// given  : f(0)...f(k) (deg(f) = k), a
// return : \sum_{i=0...infty} a^i f(i)
template<typename Field>
Field exponential_sum_limit(const vector<Field>& f, Field a) {
  if(a == 0) return f[0];
  assert(a != 1);
  int m = f.size();
  vector<Field> g(m);
  Field acc = 1;
  for(int i = 0; i < m; i++) g[i] = f[i] * acc, acc *= a;
  for(int i = 1; i < m; i++) g[i] += g[i-1];
  Field c = 0;
  acc = 1;
  for(int i = 0; i < m; i++) {
    c += Combinatorics<Field>::nCr(m, i) * acc * g[m - i - 1];
```

```cpp
    acc *= -a;
  }
  c /= (1 - a)^m;
  return c;
}

// given  : f(0)...f(k) (deg(f) = k), a, n
// return : \sum_{i=0...n-1} a^i f(i)
template<typename Field>
Field exponential_sum(const vector<Field>& f, Field a, int64_t n) {
  if(n == 0) return 0;
  if(a == 0) return f[0];
  if(a == 1) {
    // Interpolate polynomial of deg == k + 1
    return linalg::lagrange_iota_sum(f, n);
  }
  int m = f.size();
  vector<Field> g(m);
  auto c = exponential_sum_limit(f, a);
  Field acc = 1;
  for(int i = 0; i < m; i++) g[i] = f[i] * acc, acc *= a;
  for(int i = 1; i < m; i++) g[i] += g[i-1];
  auto ai = Field(1) / a;
  acc = 1;
  for(int i = 0; i < m; i++) {
    g[i] = (g[i] - c) * acc;
    acc *= ai;
  }
  // Interpolate polynomial of deg == k
  auto tn = linalg::lagrange_iota(g, n - 1);
  return c + (a^(n-1)) * tn;
}

// given  : p, n
// return : (0^p, 1^p, ... , n^p)
template <typename Field>
vector<Field> monomials(int p, int n) {
  vector<Field> f(n + 1, Field(0));
  if (!p) {
    f[0] = 1;
    return std::move(f);
  }
  f[1] = 1;
  vector<bool> sieve(n + 1, false);
  vector<int> ps;
  for (int i = 2; i <= n; i++) {
    if (!sieve[i]) {
      f[i] = Field(i)^p;
      ps.push_back(i);
    }
    for (int j = 0; j < (int)ps.size() && i * ps[j] <= n; j++) {
      sieve[i * ps[j]] = 1;
      f[i * ps[j]] = f[i] * f[ps[j]];
      if (i % ps[j] == 0) break;
    }
  }
  return std::move(f);
}
} // namespace lib

#endif
```

7.2. MultipointEvaluation

```cpp
#ifndef _LIB_POLYNOMIAL_MULTIPOINT_EVALUATION
#define _LIB_POLYNOMIAL_MULTIPOINT_EVALUATION
#include "../PolynomialRing.cpp"
#include "../Traits.cpp"
#include <bits/stdc++.h>

namespace lib {
using namespace std;
namespace math {
namespace {
/// keep caide
using traits::IsInputIterator;
/// keep caide
using traits::HasInputIterator;

} // namespace
template <typename Poly> struct MultipointEvaluation {
  using field = typename Poly::field;
  int n;
  vector<field> w;
  vector<Poly> up, down;

  template <
      typename Iterator,
      typename enable_if<IsInputIterator<Iterator>::value>::type * = nullptr>
  MultipointEvaluation(Iterator begin, Iterator end) : w(distance(begin, end)) {
    int i = 0;
    for (auto it = begin; it != end; ++it, ++i)
      w[i] = *it;
    n = w.size();
    build();
  }

  template <
      typename Container,
      typename enable_if<HasInputIterator<Container>::value>::type * = nullptr>
  MultipointEvaluation(const Container &container)
      : MultipointEvaluation(container.begin(), container.end()) {}

  void build() {
    if(w.empty()) return;
    up = vector<Poly>(2 * n);
    down = vector<Poly>(2 * n);
    for(int i = 0; i < n; i++)
      up[i+n] = {-w[i], 1};
    for(int i = n-1; i; i--)
      up[i] = up[2*i] * up[2*i+1];
  }

  vector<field> eval(const Poly &p) {
    down[1] = p % up[1];
    for(int i = 2; i < 2*n; i++)
      down[i] = down[i/2] % up[i];
    vector<field> res(n);
    for(int i = 0; i < n; i++)
      res[i] = down[i+n][0];
    return res;
  }

  template<typename Iterator>
  Poly interp(const Iterator& begin, const Iterator& end) {
    assert(n == distance(begin, end));
    vector<field> a = eval(up[1].derivative());
```

```cpp
    auto it = begin;
    for(int i = 0; i < n; i++, ++it)
      down[i+n] = {*it / a[i]};
    for(int i = n-1; i; i--)
      down[i] = down[i*2] * up[i*2+1] + down[i*2+1] * up[i*2];
    return down[1];
  }

  template <
      typename Container,
      typename enable_if<traits::HasInputIterator<Container>::value>::type *
    = nullptr>
  Poly interp(const Container &container) {
    interp(container.begin(), container.end());
  }

};
} // namespace math
} // namespace lib

#endif
```

7.3. Transform

```cpp
#ifndef _LIB_POLYNOMIAL_TRANSFORM
#define _LIB_POLYNOMIAL_TRANSFORM
#include <bits/stdc++.h>

namespace lib {
using namespace std;
template<template <class> class T>
struct TransformMultiplication {
  template<typename Field>
  using Transform = T<Field>;

  template <typename Field>
  vector<Field> operator()(const vector<Field> &a,
                           const vector<Field> &b) const {
    return T<Field>::convolve(a, b);
  };

  template<typename Field>
  inline VectorN<Field> transform(int n, const vector<Field>& p) const {
    int np = next_power_of_two(n);
    return T<Field>::transform(p, np);
  }

  template<typename Field>
  inline vector<Field> itransform(int n, const vector<Field>& p) const {
    return T<Field>::itransform(p, n);
  }

  template <typename Field, typename Functor, typename ...Ts>
  inline vector<Field> on_transform(
    int n,
    Functor& f,
    const vector<Ts>&... vs) const {
    int np = next_power_of_two(n);
    return T<Field>::itransform(
      f(n, T<Field>::transform(vs, np)...), n);
  }
};
} // namespace lib

#endif
```