

1	unindexed	2			
1.1	BitTricks	2	1.42	Subset	45
1.2	Combinatorics	2	1.43	SuffixArray	46
1.3	Complex	2	1.44	Symbolic	47
1.4	DFT	3	1.45	Template	49
1.5	DSU	4	1.46	Traits	49
1.6	Epsilon	4	1.47	Treap	50
1.7	Euclid	5	1.48	TwoSat	51
1.8	FFT	6	1.49	VectorN	52
1.9	FHT	7	2	ds	53
1.10	FastMap	9	2.1	LiChaoTree	53
1.11	Fenwick	9	2.2	OrderedIntTree	54
1.12	Graph	10	2.3	StaticRMQ	54
1.13	HLD	12	3	dsu	54
1.14	Karatsuba	13	3.1	BinaryLifting	54
1.15	Lagrange	14	3.2	Compress	55
1.16	LinearProgram	14	3.3	DSU	55
1.17	LinearRecurrence	15	3.4	SpanningTree	56
1.18	LongMultiplication	17	3.5	Time	56
1.19	Math	17	4	graphs	57
1.20	Matrix	18	4.1	BlockCut	57
1.21	Maxflow	19	4.2	Chordal	58
1.22	ModularInteger	20	5	matroid	59
1.23	MontgomeryInteger	23	5.1	CographicMatroid	59
1.24	NTT	24	5.2	ColorMatroid	59
1.25	NumberTheory	25	5.3	Compose	60
1.26	OfflineRMQ	26	5.4	GraphicMatroid	60
1.27	PolynomialRing	26	5.5	Matroid	61
1.28	PowerSeries	31	5.6	MatroidIntersection	61
1.29	Random	31	6	geometry	63
1.30	RangeDSU	32	6.1	Caliper	63
1.31	RollingHash	32	6.2	Circle2D	63
1.32	Segtree	35	6.3	GeometryEpsilon	65
1.33	SegtreeBeats	37	6.4	Line2D	65
1.34	SegtreeFast	39	6.5	Polygon2D	70
1.35	SegtreeHLD	40	6.6	Trigonometry	74
1.36	SegtreeImplicit	42	7	polynomial	74
1.37	SegtreeLazy	42	7.1	ExponentialSum	74
1.38	SegtreeNormal	42	7.2	MultipointEvaluation	75
1.39	SegtreePersistent	43	7.3	Transform	76
1.40	SegtreeSplash	43			
1.41	Simplex	44			

1. unindexed

1.1. BitTricks

```

1 #ifndef _LIB_BIT_TRICKS
2 #define _LIB_BIT_TRICKS
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 long long next_power_of_two(long long n) {
7     if (n <= 0) return 1;
8     return 1LL << (sizeof(long long) * 8 - 1 - __builtin_clzll(n) +
9                     ((n & (n - 1LL)) != 0));
10 }
11 } // namespace lib
12
13 #endif

```

1.2. Combinatorics

```

1 #ifndef _LIB_COMBINATORICS
2 #define _LIB_COMBINATORICS
3 #include <bits/stdc++.h>
4 #include "BitTricks.cpp"
5
6 namespace lib {
7 using namespace std;
8 template<typename T>
9 struct Combinatorics {
10     static vector<T> fat;
11     static vector<T> inv;
12     static vector<T> ifat;
13
14     static T factorial(int i) {
15         ensure_fat(next_power_of_two(i));
16         return fat[i];
17     }
18
19     static T inverse(int i) {
20         ensure_inv(next_power_of_two(i));
21         return inv[i];
22     }
23
24     static T ifactorial(int i) {
25         ensure_ifat(next_power_of_two(i));
26         return ifat[i];
27     }
28
29     static T nCr(int n, int K) {
30         if(K > n) return 0;
31         ensure_fat(next_power_of_two(n));
32         ensure_ifat(next_power_of_two(n));
33         return fat[n] * ifat[n-K] * ifat[K];
34     }
35
36     static T arrangement(int n, int K) {
37         return nCr(n, K) * factorial(n);
38     }
39
40     static T nCr_rep(int n, int K) {
41         return interpolate(n - 1, K);
42     }
43
44     static T interpolate(int a, int b) {

```

```

45         return nCr(a+b, b);
46     }
47
48     static void ensure_fat(int i) {
49         int o = fat.size();
50         if(i < o) return;
51         fat.resize(i+1);
52         for(int j = o; j <= i; j++) fat[j] = fat[j-1]*j;
53     }
54
55     static void ensure_inv(int i) {
56         int o = inv.size();
57         if(i < o) return;
58         inv.resize(i+1);
59         for(int j = o; j <= i; j++) inv[j] = -(inv[T::mod%j] * (T::mod/j));
60     }
61
62     static void ensure_ifat(int i) {
63         int o = ifat.size();
64         if(i < o) return;
65         ifat.resize(i+1);
66         ensure_inv(i);
67         for(int j = o; j <= i; j++) ifat[j] = ifat[j-1]*inv[j];
68     }
69 };
70
71 template<typename T>
72 vector<T> Combinatorics<T>::fat = vector<T>(1, T(1));
73 template<typename T>
74 vector<T> Combinatorics<T>::inv = vector<T>(2, T(1));
75 template<typename T>
76 vector<T> Combinatorics<T>::ifat = vector<T>(1, T(1));
77 } // namespace lib
78
79 #endif

```

1.3. Complex

```

1 #ifndef _LIB_COMPLEX
2 #define _LIB_COMPLEX
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 template <typename T> struct Complex {
8     T re, im;
9     Complex(T a = T(), T b = T()) : re(a), im(b) {}
10     T real() const { return re; }
11     T imag() const { return im; }
12     explicit operator T() const { return re; }
13     template<typename G>
14     operator Complex<G>() const { return Complex<G>(re, im); }
15     Complex conj() const { return Complex(re, -im); }
16     void operator+=(const Complex<T> &rhs) { re += rhs.re, im += rhs.im; }
17     void operator-=(const Complex<T> &rhs) { re -= rhs.re, im -= rhs.im; }
18     void operator*=(const Complex<T> &rhs) {
19         tie(re, im) =
20             make_pair(re * rhs.re - im * rhs.im, re * rhs.im + im * rhs.re);
21     }
22     Complex<T> operator+(const Complex<T> &rhs) {
23         Complex<T> res = *this;
24         res += rhs;
25         return res;
26     }

```

```

27 Complex<T> operator-(const Complex<T> &rhs) {
28     Complex<T> res = *this;
29     res -= rhs;
30     return res;
31 }
32 Complex<T> operator*(const Complex<T> &rhs) {
33     Complex<T> res = *this;
34     res *= rhs;
35     return res;
36 }
37 Complex<T> operator-() const {
38     return {-re, -im};
39 }
40 void operator/=(const T x) { re /= x, im /= x; }
41 };
42 } // namespace lib
43
44 #endif

```

1.4. DFT

```

1 #ifndef _LIB_DFT
2 #define _LIB_DFT
3 #include <bits/stdc++.h>
4 #include "BitTricks.cpp"
5
6 namespace lib {
7     using namespace std;
8     namespace linalg {
9         template <typename Ring, typename Provider>
10         struct DFT {
11             static vector<int> rev;
12             static vector<Ring> fa;
13
14             // function used to precompute rev for fixed size fft (n is a power of two)
15             static void dft_rev(int n) {
16                 Provider()(n);
17                 int lbn = __builtin_ctz(n);
18                 if ((int)rev.size() < (1 << lbn))
19                     rev.resize(1 << lbn);
20                 int h = -1;
21                 for (int i = 1; i < n; i++) {
22                     if ((i & (i - 1)) == 0)
23                         h++;
24                     rev[i] = rev[i ^ (1 << h)] | (1 << (lbn - h - 1));
25                 }
26             }
27
28             static void dft_iter(Ring *p, int n) {
29                 Provider()w;
30                 for (int L = 2; L <= n; L <= 1) {
31                     for (int i = 0; i < n; i += L) {
32                         for (int j = 0; j < L / 2; j++) {
33                             Ring z = p[i + j + L / 2] * w[j + L / 2];
34                             p[i + j + L / 2] = p[i + j] - z;
35                             p[i + j] += z;
36                         }
37                     }
38                 }
39             }
40
41             static void swap(vector<Ring> &buf) { std::swap(fa, buf); }
42             static void _dft(Ring *p, int n) {
43                 dft_rev(n);

```

```

44         for (int i = 0; i < n; i++)
45             if (i < rev[i])
46                 std::swap(p[i], p[rev[i]]);
47         dft_iter(p, n);
48     }
49     static void _idft(Ring *p, int n) {
50         _dft(p, n);
51         reverse(p + 1, p + n);
52         Ring inv = Provider().inverse(n);
53         for (int i = 0; i < n; i++)
54             p[i] *= inv;
55     }
56
57     static void dft(int n) { _dft(fa.data(), n); }
58
59     static void idft(int n) { _idft(fa.data(), n); }
60
61     static void dft(vector<Ring> &v, int n) {
62         swap(v);
63         dft(n);
64         swap(v);
65     }
66     static void idft(vector<Ring> &v, int n) {
67         swap(v);
68         idft(n);
69         swap(v);
70     }
71
72     static int ensure(int a, int b = 0) {
73         int n = a+b;
74         n = next_power_of_two(n);
75         if ((int)fa.size() < n)
76             fa.resize(n);
77         return n;
78     }
79
80     static void clear(int n) { fill(fa.begin(), fa.begin() + n, 0); }
81
82     template<typename Iterator>
83     static void fill(Iterator begin, Iterator end) {
84         int n = ensure(distance(begin, end));
85         int i = 0;
86         for(auto it = begin; it != end; ++it) {
87             fa[i++] = *it;
88         }
89         for(; i < n; i++) fa[i] = Ring();
90     }
91 };
92
93 template<typename DF, typename U>
94 static vector<U> retrieve(int n) {
95     assert(n <= DF::fa.size());
96     vector<U> res(n);
97     for(int i = 0; i < n; i++) res[i] = (U)DF::fa[i];
98     return res;
99 }
100
101 template<typename Ring, typename Provider>
102 vector<int> DFT<Ring, Provider>::rev = vector<int>();
103
104 template<typename Ring, typename Provider>
105 vector<Ring> DFT<Ring, Provider>::fa = vector<Ring>();
106 }
107 // namespace lib
108

```

109 #endif

1.5. DSU

```

1 #ifndef _LIB_DSU
2 #define _LIB_DSU
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7
8 struct DSU {
9     vector<int> p, ptime, sz;
10    int tempo = 0;
11    int merges = 0;
12    pair<int, int> last_merge_ = {-1, -1};
13
14    DSU(int n = 0) : p(n), ptime(n, 1e9), sz(n, 1) { iota(p.begin(), p.end(),
15        0); }
16
17    int make_node() {
18        int i = p.size();
19        p.emplace_back(i);
20        ptime.emplace_back(0);
21        sz.emplace_back(1);
22        return i;
23    }
24
25    int get(int i, int at) const {
26        return p[i] == i ? i : (at >= ptime[i] ? get(p[i], at) : i);
27    }
28
29    int operator[](int i) const { return get(i, tempo); }
30
31    int merge(int u, int v) {
32        u = (*this)[u], v = (*this)[v];
33        if (u == v)
34            return 0;
35        if (sz[u] < sz[v])
36            swap(u, v);
37        p[v] = u;
38        ptime[v] = ++tempo;
39        sz[u] += sz[v];
40        last_merge_ = {v, u};
41        merges++;
42        return 1;
43    }
44
45    pair<int, int> last_merge() const {
46        return last_merge_;
47    }
48
49    int n_comps() const { return (int)p.size() - merges; }
50
51    struct CompressedDSU {
52        vector<int> p;
53        CompressedDSU(int n = 0) : p(n) { iota(p.begin(), p.end(), 0); }
54        int get(int i) {
55            return p[i] == i ? i : p[i] = get(p[i]);
56        }
57        int operator[](int i) { return get(i); }
58        int& parent(int i) { return p[i]; }
59    };

```

```

60 struct FastDSU {
61     vector<int> p, sz;
62     int merges = 0;
63     pair<int, int> last_merge_ = {-1, -1};
64     FastDSU(int n = 0) : p(n), sz(n, 1) { iota(p.begin(), p.end(), 0); }
65
66     int get(int i) {
67         return p[i] == i ? i : p[i] = get(p[i]);
68     }
69     int operator[](int i) { return get(i); }
70
71     int merge(int u, int v) {
72         u = get(u), v = get(v);
73         if (u == v) return 0;
74         if (sz[u] < sz[v])
75             swap(u, v);
76         p[v] = u;
77         sz[u] += sz[v];
78         merges++;
79         last_merge_ = {v, u};
80         return 1;
81     }
82     pair<int, int> last_merge() const {
83         return last_merge_;
84     }
85     int n_comps() const { return (int)p.size() - merges; }
86 };
87 } // namespace lib
88
89 #endif

```

1.6. Epsilon

```

1 #ifndef _LIB_EPSILON
2 #define _LIB_EPSILON
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7
8 template <typename T = double> struct Epsilon {
9     T eps;
10    constexpr Epsilon(T eps = 1e-9) : eps(eps) {}
11
12    template <typename G,
13        typename enable_if<is_floating_point<G>::value>::type * =
14        nullptr>
15    int operator()(G a, G b = 0) const {
16        return a + eps < b ? -1 : (b + eps < a ? 1 : 0);
17    }
18
19    template <typename G,
20        typename enable_if<!is_floating_point<G>::value>::type * =
21        nullptr>
22    int operator()(G a, G b = 0) const {
23        return a < b ? -1 : (a > b ? 1 : 0);
24    }
25
26    template <typename G,
27        typename enable_if<is_floating_point<G>::value>::type * =
28        nullptr>
29    bool null(G a) const {
30        return (*this)(a) == 0;
31    }
32 };

```

```

29     template <typename G,
30               typename enable_if<!is_floating_point<G>::value>::type * =
31               nullptr>
32     bool null(G a) const {
33         return a == 0;
34     }
35 };
36 } // namespace lib
37
38 #endif

```

1.7. Euclid

```

1  #ifndef _LIB_EUCLID
2  #define _LIB_EUCLID
3  #include <bits/stdc++.h>
4
5  namespace lib {
6  using namespace std;
7  namespace math {
8  namespace {
9  constexpr static size_t ULL_SIZE = sizeof(unsigned long long);
10
11  template <typename T>
12  using IsManageableInt =
13      typename conditional<is_integral<T>::value && sizeof(T) <= ULL_SIZE,
14                          true_type, false_type>::type;
15  } // namespace
16
17  template <typename T, typename US = T> struct Euclid {
18      template <typename U, typename V> static V safe_mod(U x, V m) {
19          x %= m;
20          if (x < 0)
21              x += m;
22          return x;
23      }
24
25      template <typename U,
26               typename enable_if<IsManageableInt<U>::value>::type * = nullptr>
27      static U safe_mult(U a, U b, U m) {
28          a = safe_mod(a, m), b = safe_mod(b, m);
29
30          if (!b) return 0;
31          int hi = 63 - __builtin_clzll((unsigned long long)b);
32          U res = 0;
33          for (int i = hi; i >= 0; i--) {
34              res = safe_mod(res * 2, m);
35              if ((b >> i) & 1)
36                  res = safe_mod(res + a, m);
37          }
38          return res;
39      }
40
41      template <typename U,
42               typename enable_if<!IsManageableInt<U>::value>::type * = nullptr>
43      static U safe_mult(U a, U b, U m) {
44          return a * b % m;
45      }
46
47      static T euclid_(T a, T b, T &x, T &y) {
48          if (a == 0) {
49              x = 0, y = 1;
50              return b;

```

```

51     }
52     T x1, y1;
53     T g = euclid_(b % a, a, x1, y1);
54     x = y1 - b / a * x1;
55     y = x1;
56     return g;
57 }
58
59 static T euclid(T a, T b, T &x, T &y) {
60     T g = euclid_(a, b, x, y);
61     if (g < 0)
62         g = -g, x = -x, y = -y;
63     return g;
64 }
65
66 static pair<T, T> crt(T a, T b, T m1, T m2) {
67     if (m1 < m2)
68         swap(m1, m2), swap(a, b);
69     T xx, yy;
70     T g = euclid(m1, m2, xx, yy);
71     if (safe_mod(a, g) != safe_mod(b, g))
72         return {0, 0};
73
74     T mod = m1 / g * m2;
75
76     T x = safe_mod<T>(xx, mod);
77     US s = safe_mult<T>(x, (b - a) / g, m2 / g) * m1 % mod;
78     T res = safe_mod<US, US>((US)a + s, mod);
79
80     return {safe_mod<T>(res, mod), mod};
81 }
82
83 static pair<T, T> crt(const vector<pair<T, T>> &equations) {
84     pair<T, T> acc = {0, 1};
85     for (const pair<T, T> &e : equations) {
86         acc = crt(acc.first, e.first, acc.second, e.second);
87         if (!acc.second)
88             return {0, 0};
89     }
90     return acc;
91 }
92
93 static bool diophantine_solution(T a, T b, T c, T& x0, T& y0, T& g) {
94     g = euclid(a, b, x0, y0);
95     if (c % g)
96         return false;
97     x0 *= c/g;
98     y0 *= c/g;
99     return true;
100 }
101
102 // Give solutions for diophantine in the form [x = x.first * k + x.second].
103 static bool diophantine_solutions(T a, T b, T c, pair<T, T>& x, pair<T,
104 T>& y) {
105     T g;
106     if (!diophantine_solution(a, b, c, x.second, y.second, g))
107         return false;
108     x.first = b / g;
109     y.first = -a / g;
110     return true;
111 }
112
113 // Give parameterized solution (in terms of k) to:
114 // a_1 * k + b_1 = ... = a_n * k + b_n, i.e., an equation for where those
115 // functions meet.

```

```

115 static bool linear_equality_system(const vector<pair<T, T>& v, pair<T,
116 T>& res) {
117     assert(!v.empty());
118     res = v[0];
119     for(int i = 1; i < v.size(); i++) {
120         pair<T, T> x, y;
121         if (!diophantine_solutions(res.first, -v[i].first, v[i].second -
122         res.second, x, y))
123             return false;
124         auto num = res.first * x.first;
125         if (num < 0) num = -num;
126         res = {
127             num,
128             safe_mod(res.second + safe_mult(res.first, x.second, num), num),
129         };
130     }
131     return true;
132 };
133 using LongCRT = Euclid<long long, unsigned long long>;
134 } // namespace math
135 } // namespace lib
136 #endif
137

```

1.8. FFT

```

1 #ifndef _LIB_FFT
2 #define _LIB_FFT
3 #include "DFT.cpp"
4 #include "Complex.cpp"
5 #include "geometry/Trigonometry.cpp"
6 #include <bits/stdc++.h>
7
8 namespace lib {
9 using namespace std;
10 namespace linalg {
11
12 template<typename T>
13 struct ComplexRootProvider {
14     typedef Complex<T> cd;
15     typedef Complex<long double> cld;
16     static vector<cd> w;
17     static vector<cld> wl;
18
19     static cld root(long double ang) {
20         return cld(geo::trig::cos(ang), geo::trig::sin(ang));
21     }
22
23     cd operator()(int n, int k) {
24         long double ang = 2.01 * geo::trig::PI / (n / k);
25         return root(ang);
26     }
27
28     void operator()(int n) {
29         n = max(n, 2);
30         int k = max((int)w.size(), 2);
31         if ((int)w.size() < n)
32             w.resize(n), wl.resize(n);
33         else
34             return;
35         w[0] = w[1] = cd(1.0, 0.0);
36         wl[0] = wl[1] = cld(1.0, 0.0);
37         for (; k < n; k *= 2) {

```

```

37         long double ang = 2.01 * geo::trig::PI / (2*k);
38         cld step = root(ang);
39         for(int i = k; i < 2*k; i++)
40             w[i] = wl[i] = (i&1) ? wl[i/2] * step : wl[i/2];
41     }
42 }
43 cd operator()(int i) {
44     return w[i];
45 }
46 cd inverse(int n) {
47     return cd(1.0 / n, 0.0);
48 }
49 };
50
51 template<typename T>
52 vector<Complex<T>> ComplexRootProvider<T>::w = vector<Complex<T>>();
53 template<typename T>
54 vector<Complex<long double>> ComplexRootProvider<T>::wl =
55     vector<Complex<long double>>();
56
57 template<typename T = double>
58 struct FFT : public DFT<Complex<T>, ComplexRootProvider<T>> {
59     using Parent = DFT<Complex<T>, ComplexRootProvider<T>>;
60     using Parent::fa;
61
62     template <typename U>
63     static void _convolve(const vector<U> &a, const vector<U> &b) {
64         typedef Complex<T> cd;
65         int n = Parent::ensure(a.size(), b.size());
66         for (size_t i = 0; i < (size_t)n; i++)
67             fa[i] = cd(i < a.size() ? (T)a[i] : T(),
68             i < b.size() ? (T)b[i] : T());
69         Parent::dft(n);
70         for (int i = 0; i < n; i++)
71             fa[i] *= fa[i];
72         Parent::idft(n);
73         for (int i = 0; i < n; i++)
74             fa[i] = cd(fa[i].imag() / 2, T());
75     }
76
77     template<typename U>
78     static vector<U> convolve(const vector<U>& a, const vector<U>& b) {
79         int sz = (int)a.size() + b.size() - 1;
80         _convolve(a, b);
81         return retrieve<Parent, U>(sz);
82     }
83
84     template<typename U>
85     static vector<U> convolve_rounded(const vector<U>& a, const vector<U>& b) {
86         int sz = (int)a.size() + b.size() - 1;
87         _convolve(a, b);
88         vector<U> res(sz);
89         for(int i = 0; i < sz; i++) res[i] = (U)(long long)(fa[i].real() + 0.5);
90         return res;
91     }
92
93     // TODO: use separate static buffers for this function
94     template <typename M>
95     static vector<M> convolve_mod(const vector<M> &a, const vector<M> &b) {
96         typedef typename M::type_int type_int;
97         typedef typename M::large_int large_int;
98         typedef Complex<T> cd;
99         typedef vector<cd> vcd;
100         static_assert(sizeof(M::mods) / sizeof(type_int) == 1,

```

```

101         "cant multiply with multiple mods");
102     type_int base = sqrtl(M::mods[0]) + 0.5;
103     M base_m = base;
104     int sza = a.size();
105     int szb = b.size();
106     int sz = sza+szb-1;
107     int n = next_power_of_two(sz);
108     Parent::dft_rev(n);
109
110     // establish buffers
111     vcd fa(n), fb(n), C1(n), C2(n);
112
113     for (int i = 0; i < n; i++)
114         fa[i] = i < sza ? cd((type_int)a[i] / base, (type_int)a[i] % base) :
115         cd();
116     for (int i = 0; i < n; i++)
117         fb[i] = i < szb ? cd((type_int)b[i] / base, (type_int)b[i] % base) :
118         cd();
119     Parent::dft(fa, n);
120     Parent::dft(fb, n);
121
122     for (int i = 0; i < n; i++) {
123         int j = i ? n - i : 0;
124         cd a1 = (fa[i] + fa[j].conj()) * cd(0.5, 0.0);
125         cd a2 = (fa[i] - fa[j].conj()) * cd(0.0, -0.5);
126         cd b1 = (fb[i] + fb[j].conj()) * cd(0.5, 0.0);
127         cd b2 = (fb[i] - fb[j].conj()) * cd(0.0, -0.5);
128         cd c11 = a1 * b1, c12 = a1 * b2;
129         cd c21 = a2 * b1, c22 = a2 * b2;
130         C1[j] = c11 + c12 * cd(0.0, 1.0);
131         C2[j] = c21 + c22 * cd(0.0, 1.0);
132     }
133     Parent::idft(C1, n), Parent::idft(C2, n);
134
135     vector<M> res(sz);
136     for (int i = 0; i < sz; i++) {
137         int j = i ? n - i : 0;
138         M x = large_int(C1[j].real() + 0.5);
139         M y1 = large_int(C1[j].imag() + 0.5);
140         M y2 = large_int(C2[j].real() + 0.5);
141         M z = large_int(C2[j].imag() + 0.5);
142         res[i] = x * base_m * base_m + (y1 + y2) * base_m + z;
143     }
144
145     return res;
146 }
147 // namespace linalg
148
149 namespace math {
150 struct FastMultiplication {
151     template<typename T>
152     using Transform = linalg::FFT<T>;
153     template<typename Field, typename U = double>
154     vector<Field> operator()(const vector<Field> &a,
155                             const vector<Field> &b) const {
156         return linalg::FFT<U>::convolve_rounded(a, b);
157     }
158 };
159
160 struct FFTMultiplication {
161     template<typename T>
162     using Transform = linalg::FFT<T>;
163     template<typename Field, typename U = double>
164     vector<Field> operator()(const vector<Field> &a,

```

```

164         const vector<Field> &b) const {
165         return linalg::FFT<U>::convolve(a, b);
166     }
167 };
168
169 struct SafeMultiplication {
170     template<typename T>
171     using Transform = linalg::FFT<T>;
172     template<typename Field, typename U = double>
173     vector<Field> operator()(const vector<Field> &a,
174                             const vector<Field> &b) const {
175         return linalg::FFT<U>::convolve_mod(a, b);
176     }
177 };
178 } // namespace math
179 } // namespace lib
180
181 #endif

```

1.9. FHT

```

1 #ifndef _LIB_FHT
2 #define _LIB_FHT
3 #include <bits/stdc++.h>
4 #include "BitTricks.cpp"
5 #include "NTT.cpp"
6 #include "polynomial/Transform.cpp"
7
8 namespace lib {
9 using namespace std;
10 namespace linalg {
11 template<typename Ring>
12 struct FHT {
13     using Provider = MintRootProvider<Ring>;
14     using T = Ring;
15     using U = make_unsigned_t<typename Ring::type_int>;
16     using U64 = make_unsigned_t<typename Ring::large_int>;
17     static vector<Ring> fa;
18     static const int MAX_LG_N = 30;
19     static vector<Ring> g, ig;
20
21     static void precompute() {
22         if(!g.empty()) return;
23         Provider();
24         g.resize(MAX_LG_N);
25         ig.resize(MAX_LG_N);
26         for(int i = 0; i < MAX_LG_N; i++) {
27             Ring w = Provider::g ^ ((Ring::mod-1) >> (i + 2)) * 3);
28             w = -w;
29             Ring iw = w.inverse();
30             g[i] = w;
31             ig[i] = iw;
32         }
33     }
34
35     static inline U& v(Ring& p) {
36         return (U&)p.data();
37     }
38
39     static inline U v(const Ring& p) {
40         return (U)p.data();
41     }
42
43     static void dft_iter(Ring *p, int n) {

```

```

44 // decimation-in-time
45 // natural to reverse ordering
46 for (int B = n >> 1; B; B >>= 1) {
47     Ring w = 1;
48     for (int i = 0, twiddle = 0; i < n; i += B * 2) {
49         for (int j = i; j < i + B; j++) {
50             Ring z = p[j + B] * w;
51             p[j + B] = p[j] - z;
52             p[j] += z;
53         }
54         w *= g[__builtin_ctz(++twiddle)];
55     }
56 }
57 }
58
59 static void idft_iter(Ring *p, int n) {
60     // decimation-in-frequency
61     // reverse to natural ordering
62     for (int B = 1; B < n; B <= 1) {
63         Ring w = 1;
64         for (int i = 0, twiddle = 0; i < n; i += B * 2) {
65             for (int j = i; j < i + B; j++) {
66                 Ring z = (p[j] - p[j + B]) * w;
67                 p[j] += p[j + B];
68                 p[j + B] = z;
69             }
70             w *= ig[__builtin_ctz(++twiddle)];
71         }
72     }
73 }
74
75 static void swap(vector<Ring> &buf) { std::swap(fa, buf); }
76 static void _dft(Ring *p, int n) {
77     precompute();
78     dft_iter(p, n);
79 }
80 static void _idft(Ring *p, int n) {
81     precompute();
82     idft_iter(p, n);
83     Ring inv = Provider().inverse(n);
84     for (int i = 0; i < n; i++)
85         p[i] *= inv;
86 }
87
88 static void dft(int n) { _dft(fa.data(), n); }
89
90 static void idft(int n) { _idft(fa.data(), n); }
91
92 static void dft(vector<Ring> &v, int n) {
93     swap(v);
94     dft(n);
95     swap(v);
96 }
97 static void idft(vector<Ring> &v, int n) {
98     swap(v);
99     idft(n);
100    swap(v);
101 }
102
103 static int ensure(int a, int b = 0) {
104     int n = a+b;
105     n = next_power_of_two(n);
106     if ((int)fa.size() < n)
107         fa.resize(n);
108     return n;

```

```

109 }
110
111 static void clear(int n) { fill(fa.begin(), fa.begin() + n, 0); }
112
113 template<typename Iterator>
114 static void fill(Iterator begin, Iterator end) {
115     int n = ensure(distance(begin, end));
116     int i = 0;
117     for(auto it = begin; it != end; ++it) {
118         fa[i++] = *it;
119     }
120     for(; i < n; i++) fa[i] = Ring();
121 }
122
123 static void _convolve(const vector<T> &a) {
124     int n = ensure(a.size(), a.size());
125     for (size_t i = 0; i < (size_t)n; i++)
126         fa[i] = i < a.size() ? a[i] : T();
127     dft(n);
128     for (int i = 0; i < n; i++)
129         fa[i] *= fa[i];
130     idft(n);
131 }
132
133 static void _convolve(const vector<T> &a, const vector<T> &b) {
134     if(std::addressof(a) == std::addressof(b))
135         return _convolve(a);
136     int n = ensure(a.size(), b.size());
137     for (size_t i = 0; i < (size_t)n; i++)
138         fa[i] = i < a.size() ? a[i] : T();
139     dft(n);
140     // TODO: have a buffer for this
141     auto fb = retrieve<FHT<T>, T>(n);
142     for(size_t i = 0; i < (size_t)n; i++)
143         fa[i] = i < b.size() ? b[i] : T();
144     dft(n);
145     for (int i = 0; i < n; i++)
146         fa[i] *= fb[i];
147     idft(n);
148 }
149
150 static vector<T> convolve(const vector<T> &a, const vector<T> &b) {
151     int sz = (int)a.size() + b.size() - 1;
152     _convolve(a, b);
153     return retrieve<FHT<T>, T>(sz);
154 }
155
156 static VectorN<T> transform(vector<T> a, int n) {
157     a.resize(n);
158     dft(a, n);
159     return a;
160 }
161
162 static vector<T> itransform(vector<T> a, int n) {
163     int sz = a.size();
164     idft(a, sz);
165     a.resize(min(n, sz));
166     return a;
167 }
168 };
169
170 template<typename Ring>
171 vector<Ring> FHT<Ring>::fa = vector<Ring>();
172 template<typename Ring>
173 vector<Ring> FHT<Ring>::g = vector<Ring>();

```



```

174 template<typename Ring>
175 vector<Ring> FHT<Ring>::ig = vector<Ring>();
176 }
177
178 using FHTMultiplication = TransformMultiplication<linalg::FHT>;
179 } // namespace lib
180
181 #endif

```

1.10. FastMap

```

1 #ifndef _LIB_FAST_MAP
2 #define _LIB_FAST_MAP
3 #include <bits/stdc++.h>
4
5 // Pretty much copied from:
6 //
7 // https://nyaannyaan.github.io/library/data-structure/hash-map-variable-length.hpp
8 namespace lib {
9 using namespace std;
10
11 template <typename Key, typename Val = Key>
12 struct FastMap {
13     using u32 = uint32_t;
14     using u64 = uint64_t;
15
16     u32 cap, s;
17     vector<Key> keys;
18     vector<Val> vals;
19     vector<bool> flag;
20     u64 r;
21     u32 shift;
22     Val DefaultValue;
23
24     static u64 rng() {
25         u64 m = chrono::duration_cast<chrono::nanoseconds>(
26             chrono::high_resolution_clock::now().time_since_epoch())
27             .count();
28         m ^= m >> 16;
29         m ^= m << 32;
30         return m;
31     }
32
33     void reallocate() {
34         cap <= 1;
35         vector<Key> k(cap);
36         vector<Val> v(cap);
37         vector<bool> f(cap);
38         u32 sh = shift - 1;
39         for (int i = 0; i < (int)flag.size(); i++) {
40             if (flag[i]) {
41                 u32 hash = (u64(keys[i]) * r) >> sh;
42                 while (f[hash]) hash = (hash + 1) & (cap - 1);
43                 k[hash] = keys[i];
44                 v[hash] = vals[i];
45                 f[hash] = 1;
46             }
47         }
48         keys.swap(k);
49         vals.swap(v);
50         flag.swap(f);
51         --shift;
52     }
53
54     explicit FastMap()
55         : cap(8),
56           s(0),
57           keys(cap),
58           vals(cap),
59           flag(cap),
60           r(rng()),
61           shift(64 - __lg(cap)),
62           DefaultValue(Val()) {}
63
64     Val& operator[](const Key& i) {
65         u32 hash = (u64(i) * r) >> shift;
66         while (true) {
67             if (!flag[hash]) {
68                 if (s + s / 4 >= cap) {
69                     reallocate();
70                     return (*this)[i];
71                 }
72                 keys[hash] = i;
73                 flag[hash] = 1;
74                 ++s;
75                 return vals[hash] = DefaultValue;
76             }
77             if (keys[hash] == i) return vals[hash];
78             hash = (hash + 1) & (cap - 1);
79         }
80
81         // exist -> return pointer of Val
82         // not exist -> return nullptr
83         const Val* find(const Key& i) const {
84             u32 hash = (u64(i) * r) >> shift;
85             while (true) {
86                 if (!flag[hash]) return nullptr;
87                 if (keys[hash] == i) return &(vals[hash]);
88                 hash = (hash + 1) & (cap - 1);
89             }
90         }
91
92         // return vector< pair<const Key&, val& > >
93         vector<pair<Key, Val>> enumerate() const {
94             vector<pair<Key, Val>> ret;
95             for (u32 i = 0; i < cap; ++i)
96                 if (flag[i]) ret.emplace_back(keys[i], vals[i]);
97             return ret;
98         }
99
100         int size() const { return s; }
101
102         // set default_value
103         void set_default(const Val& val) { DefaultValue = val; }
104     };
105 } // namespace lib
106
107 #endif

```

1.11. Fenwick

```

1 #ifndef _LIB_FENWICK
2 #define _LIB_FENWICK
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;

```

```

7  template<typename T>
8  struct Fenwick {
9      vector<int> t;
10     Fenwick(int n) : t(n+1) {}
11     int size() const { return t.size() - 1; }
12     void add(int i, T x) {
13         for(i++; i < t.size(); i += (i&-i))
14             t[i] += x;
15     }
16     T get(int i) const {
17         T res = 0;
18         for(i++; i > 0; i -= (i&-i))
19             res += t[i];
20         return res;
21     }
22     T get(int i, int j) const {
23         return get(j) - get(i - 1);
24     }
25     T from(int i) const {
26         return get(i, size() - 1);
27     }
28 };
29 } // namespace lib
30 #endif

```

1.12. Graph

```

1  #ifndef _LIB_GRAPH
2  #define _LIB_GRAPH
3  #include "Traits.cpp"
4  #include "utils/Wrappers.cpp"
5  #include <bits/stdc++.h>
6
7  namespace lib {
8  using namespace std;
9  namespace graph {
10     template <typename V = void, typename E = void, bool Directed = false>
11     struct GraphImpl {
12         typedef GraphImpl<V, E> self_type;
13         typedef vector<vector<int>> adj_list;
14         typedef Edge<E> edge_type;
15         typedef VertexWrapper<V> vertex_type;
16
17         const static bool directed = Directed;
18
19         vector<edge_type> edges;
20         adj_list adj;
21
22         vector<vertex_type> vertices;
23
24         class iterator {
25         public:
26             typedef iterator self_type;
27             typedef edge_type value_type;
28             typedef edge_type &reference;
29             typedef edge_type *pointer;
30             typedef std::forward_iterator_tag iterator_category;
31             typedef int difference_type;
32             iterator(vector<int> *adj, vector<edge_type> *edges, int ptr = 0)
33                 : adj_(adj), edges_(edges), ptr_(ptr) {}
34             self_type operator++() {
35                 ptr_++;
36                 return *this;
37             }

```

```

38     self_type operator++(int junk) {
39         self_type i = *this;
40         ptr_++;
41         return i;
42     }
43     reference operator*() { return (*edges_)[(adj_[ptr_])]; }
44     pointer operator->() { return &(*edges_)[(adj_[ptr_])]; }
45     bool operator==(const self_type &rhs) const {
46         return adj_ == rhs.adj_ && ptr_ == rhs.ptr_;
47     }
48     bool operator!=(const self_type &rhs) const { return !(*this == rhs); }
49
50 private:
51     vector<int> *adj_;
52     vector<edge_type> *edges_;
53     int ptr_;
54 };
55
56 class const_iterator {
57 public:
58     typedef const_iterator self_type;
59     typedef edge_type value_type;
60     typedef edge_type &reference;
61     typedef edge_type *pointer;
62     typedef std::forward_iterator_tag iterator_category;
63     typedef int difference_type;
64     const_iterator(vector<int> *adj, vector<edge_type> *edges, int ptr = 0)
65         : adj_(adj), edges_(edges), ptr_(ptr) {}
66     self_type operator++() {
67         ptr_++;
68         return *this;
69     }
70     self_type operator++(int junk) {
71         self_type i = *this;
72         ptr_++;
73         return i;
74     }
75     const value_type &operator*() { return (*edges_)[(adj_[ptr_])]; }
76     const value_type *operator->() { return &(*edges_)[(adj_[ptr_])]; }
77     bool operator==(const self_type &rhs) const {
78         return adj_ == rhs.adj_ && ptr_ == rhs.ptr_;
79     }
80     bool operator!=(const self_type &rhs) const { return !(*this == rhs); }
81
82 private:
83     vector<int> *adj_;
84     vector<edge_type> *edges_;
85     int ptr_;
86 };
87
88 struct iterable {
89     vector<int> *adj_;
90     vector<edge_type> *edges_;
91
92     iterable(vector<int> *adj, vector<edge_type> *edges)
93         : adj_(adj), edges_(edges) {}
94
95     inline iterator begin() { return iterator(adj_, edges_); }
96     inline iterator end() { return iterator(adj_, edges_, adj_>size()); }
97
98     inline const_iterator cbegin() const {
99         return const_iterator(adj_, edges_);
100     }
101     inline const_iterator cend() const {
102         return const_iterator(adj_, edges_, adj_>size());

```

```

103     }
104
105     inline const_iterator begin() const { return cbegin(); }
106     inline const_iterator end() const { return cend(); }
107
108     inline edge_type &operator[](int i) { return (*edges_)[(*adj_)[i]]; }
109     inline const edge_type &operator[](int i) const {
110         return (*edges_)[(*adj_)[i]];
111     }
112
113     inline int index(int i) const { return (*adj_)[i]; }
114     inline int size() const { return adj_>size(); }
115 };
116
117 GraphImpl() {}
118
119 template <typename S = V,
120         typename enable_if<is_void<S>::value>::type * = nullptr>
121 GraphImpl(size_t n) : adj(n) {}
122
123 template <typename S = V,
124         typename enable_if<!is_void<S>::value>::type * = nullptr>
125 GraphImpl(size_t n) : adj(n), vertices(n) {}
126
127 inline iterable n_edges(int i) { return iterable(&adj[i], &edges); }
128 inline const iterable n_edges(int i) const {
129     return iterable(const_cast<vector<int>*>(&adj[i]),
130                    const_cast<vector<edge_type>*>(&edges));
131 }
132 inline int degree(int i) const { return adj[i].size(); }
133
134 inline int size() const { return adj.size(); }
135 inline int edge_size() const { return edges.size(); }
136 inline edge_type &edge(int i) { return edges[i]; }
137 inline edge_type edge(int i) const { return edges[i]; }
138
139 inline vector<edge_type> all_edges() const { return edges; }
140
141 template <typename S = V,
142         typename enable_if<!is_void<S>::value>::type * = nullptr>
143 inline S &vertex(int i) {
144     return vertices[i];
145 }
146
147 template <typename S = V,
148         typename enable_if<!is_void<S>::value>::type * = nullptr>
149 inline V vertex(int i) const {
150     return vertices[i];
151 }
152
153 template <typename S = V,
154         typename enable_if<is_void<S>::value>::type * = nullptr>
155 inline void add_vertex() {
156     adj.emplace_back();
157 }
158
159 template <typename S = V,
160         typename enable_if<!is_void<S>::value>::type * = nullptr>
161 inline S &add_vertex() {
162     adj.emplace_back();
163     return vertices.emplace_back().data;
164 }
165
166 template <typename S = E,
167         typename enable_if<is_void<S>::value>::type * = nullptr>

```

```

168 inline void add_edge_(int u, int v) {
169     adj[u].push_back(edges.size());
170     edges.push_back({u, v});
171 }
172
173 template <typename S = E,
174         typename enable_if<!is_void<S>::value>::type * = nullptr>
175 inline S &add_edge_(int u, int v) {
176     adj[u].push_back(edges.size());
177     edges.push_back({u, v});
178     return edges.back().data;
179 }
180
181 void add_2edge(int u, int v) {
182     add_edge_(u, v);
183     add_edge_(v, u);
184 }
185
186 template <typename S = E,
187         typename enable_if<!is_void<S>::value>::type * = nullptr>
188 inline void add_2edge(int u, int v, const S &data) {
189     add_edge_(u, v) = data;
190     add_edge_(v, u) = data;
191 }
192
193 template <typename S = E,
194         typename enable_if<is_void<S>::value && Directed>::type * =
195         nullptr>
196 inline void add_edge(int u, int v) {
197     adj[u].push_back(edges.size());
198     edges.push_back({u, v});
199 }
200
201 template <typename S = E,
202         typename enable_if<!is_void<S>::value && Directed>::type * =
203         nullptr>
204 inline S &add_edge(int u, int v) {
205     adj[u].push_back(edges.size());
206     edges.push_back({u, v});
207     return edges.back().data;
208 }
209
210 template<typename V = void, typename E = void>
211 using Graph = GraphImpl<V, E, false>;
212
213 template<typename V = void, typename E = void>
214 using DirectedGraph = GraphImpl<V, E, true>;
215
216 template <typename V = void, typename E = void>
217 struct RootedForest : public DirectedGraph<V, E> {
218     typedef RootedForest<V, E> self_type;
219     using typename DirectedGraph<V, E>::adj_list;
220     using typename DirectedGraph<V, E>::edge_type;
221     using DirectedGraph<V, E>::DirectedGraph;
222     using DirectedGraph<V, E>::adj;
223     using DirectedGraph<V, E>::edge;
224     vector<int> p, pe;
225
226     void build_parents() {
227         if ((int)p.size() == this->size())
228             return;
229
230         int n = this->size();
231         stack<int> st;

```

```

231     vector<bool> vis(n);
232     p.assign(n, -1), pe.assign(n, -1);
233     for (int i = 0; i < n; i++) {
234         if (!vis[i]) {
235             st.push(i);
236             vis[i] = true;
237             while (!st.empty()) {
238                 int u = st.top();
239                 st.pop();
240
241                 for (int k : adj[u]) {
242                     int v = edge(k).to;
243                     vis[v] = true;
244                     st.push(v), pe[v] = k, p[v] = u;
245                 }
246             }
247         }
248     }
249 }
250
251 inline int parent(int i) const {
252     const_cast<self_type*>(this)->build_parents();
253     return p[i];
254 }
255
256 inline bool is_root(int i) const { return parent(i) != -1; }
257
258 inline edge_type &parent_edge(int i) {
259     build_parents();
260     return edge(pe[i]);
261 }
262 inline edge_type &parent_edge(int i) const {
263     const_cast<self_type*>(this)->build_parents();
264     return edge(pe[i]);
265 }
266
267 vector<int> roots() const {
268     vector<int> res;
269     const_cast<self_type*>(this)->build_parents();
270     int n = this->size();
271
272     for (int i = 0; i < n; i++)
273         if (p[i] == -1)
274             res.push_back(i);
275     return res;
276 }
277 };
278
279 template <typename V = void, typename E = void>
280 struct RootedTree : public RootedForest<V, E> {
281     using typename RootedForest<V, E>::adj_list;
282     int root;
283
284     RootedTree(int n, int root) : RootedForest<V, E>(n) {
285         assert(n > 0);
286         assert(root < n);
287         this->root = root;
288     }
289
290     RootedTree(const adj_list &adj, int root) : RootedForest<V, E>(adj) {
291         assert(adj.size() > 0);
292         assert(root < adj.size());
293         this->root = root;
294     }
295 };

```

```

296 namespace builders {
297 namespace {
298 template <typename F, typename G>
299 void dfs_rooted_forest(F &forest, const G &graph, int u, vector<bool> &vis) {
300     vis[u] = true;
301     for (const auto &ed : graph.n_edges(u)) {
302         int v = ed.to;
303         if (!vis[v]) {
304             forest.add_edge(u, v);
305             dfs_rooted_forest(forest, graph, v, vis);
306         }
307     }
308 }
309 }
310 } // namespace
311
312 template <typename A, typename B>
313 RootedForest<A, B> make_rooted_forest(const Graph<A, B> &graph,
314                                     const vector<int> &roots) {
315     RootedForest<A, B> res(graph.size());
316     vector<bool> vis(graph.size());
317     for (int i : roots)
318         if (!vis[i])
319             dfs_rooted_forest(res, graph, i, vis);
320     for (int i = 0; i < graph.size(); i++)
321         if (!vis[i])
322             dfs_rooted_forest(res, graph, i, vis);
323     return res;
324 }
325 } // namespace builders
326 } // namespace graph
327 } // namespace lib
328
329 #endif

```

1.13. HLD

```

1  #ifndef _LIB_HLD
2  #define _LIB_HLD
3  #include "Graph.cpp"
4  #include "Segtree.cpp"
5  #include "Traits.cpp"
6  #include <bits/stdc++.h>
7
8  namespace lib {
9  using namespace std;
10 namespace graph {
11 namespace {
12 void empty_lifter(int a, int b, bool inv) {}
13 } // namespace
14
15 template <typename G> struct HLD {
16     G graph;
17     vector<int> in, out, rin;
18     vector<int> L, sz, ch;
19     int tempo;
20
21     HLD(const G &g)
22         : graph(g), in(g.size()), out(g.size()), rin(g.size()), L(g.size()),
23           sz(g.size()), ch(g.size()) {
24         build();
25     }
26
27     inline int size() const { return graph.size(); }

```

```

28 void dfs0(int u) {
29     sz[u] = 1;
30     for (auto &k : graph.adj[u]) {
31         int v = graph.edge(k).to;
32         L[v] = L[u] + 1;
33         dfs0(v);
34         if (sz[v] > sz[graph.edge(graph.adj[u][0]).to])
35             swap(k, graph.adj[u][0]);
36         sz[u] += sz[v];
37     }
38 }
39
40 void dfs1(int u) {
41     in[u] = tempo++;
42     rin[in[u]] = u;
43
44     if (graph.adj[u].size() > 0) {
45         int v = graph.edge(graph.adj[u][0]).to;
46         ch[v] = ch[u];
47         dfs1(v);
48         for (size_t i = 1; i < graph.adj[u].size(); i++) {
49             v = graph.edge(graph.adj[u][i]).to;
50             ch[v] = v;
51             dfs1(v);
52         }
53     }
54     out[u] = tempo;
55 }
56
57 void build() {
58     vector<int> roots = graph.roots();
59     for (int i : roots)
60         dfs0(i);
61     tempo = 0;
62     for (int i : roots)
63         dfs1(i);
64 }
65
66 template <typename Lifter>
67 inline void operate_on_subtree(int u, Lifter &lifter) {
68     lifter(in[u], out[u] - 1, false);
69 }
70
71 template <typename T, typename QueryIssuer>
72 inline T query_on_subtree(int u, const QueryIssuer &issuer) {
73     return issuer(in[u], out[u] - 1);
74 }
75
76 template <typename Lifter>
77 inline void operate_on_subtree_edges(int u, Lifter &lifter) {
78     if (in[u] + 2 <= out[u])
79         lifter(in[u] + 1, out[u] - 1, false);
80 }
81
82 template <typename T, typename QueryIssuer>
83 inline void query_on_subtree_edges(int u, const QueryIssuer &issuer) {
84     assert(in[u] + 2 <= out[u]);
85     return issuer(in[u] + 1, out[u] - 1);
86 }
87
88 template <bool is_edge, typename Lifter>
89 int _query_path(int u, int v, Lifter &lifter) {
90     int inv = 0;
91     for (; ch[u] != ch[v]; u = graph.parent(ch[u])) {
92

```

```

93         if (L[ch[u]] < L[ch[v]])
94             swap(u, v), inv ^= 1;
95         lifter(in[ch[u]], in[u], (bool)inv);
96     }
97     if (L[u] > L[v])
98         swap(u, v), inv ^= 1;
99     inv ^= 1;
100     if (is_edge && in[u] + 1 <= in[v])
101         lifter(in[u] + 1, in[v], (bool)inv);
102     else if (!is_edge)
103         lifter(in[u], in[v], (bool)inv);
104     return u;
105 }
106
107 template <typename Lifter>
108 inline int operate_on_path(int u, int v, Lifter &lifter) {
109     return _query_path<false>(u, v, lifter);
110 }
111
112 template <typename Lifter>
113 inline int operate_on_path_edges(int u, int v, Lifter &lifter) {
114     return _query_path<true>(u, v, lifter);
115 }
116
117 template <typename Op> inline void operate_on_vertex(int u, Op &op) {
118     op(in[u]);
119 }
120
121 template <typename T, typename QueryIssuer>
122 inline T query_on_vertex(int u, const QueryIssuer &issuer) {
123     return issuer(in[u]);
124 }
125
126 inline int lca(int u, int v) {
127     return _query_path<false>(u, v, empty_lifter);
128 }
129
130 inline int dist(int u, int v) {
131     int uv = lca(u, v);
132     return L[u] + L[v] - 2 * L[uv];
133 }
134 };
135
136 template <typename G> HLD<G> make_hld(const G &graph) { return
137     HLD<G>(graph); }
138 } // namespace graph
139 } // namespace lib
140 #endif

```

1.14. Karatsuba

```

1 #ifndef _LIB_KARATSUBA
2 #define _LIB_KARATSUBA
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace math {
8 struct Karatsuba {
9     template <typename Field>
10     vector<Field> multiply(const vector<Field> &a, const vector<Field> &b)
11     const {
12         if (b.size() == 0)

```

```

12     return {};
13     if (b.size() == 1) {
14         vector<Field> res = a;
15         for (Field &res : a)
16             res *= b[0];
17     }
18
19     int shift = a.size() / 2;
20     vector<Field> a0 = a;
21     vector<Field> b0 = b;
22     a0.resize(min(shift, a.size()));
23     b0.resize(min(shift, b.size()));
24 }
25
26 template <typename Field>
27 vector<Field> operator()(const vector<Field> &a,
28                         const vector<Field> &b) const {
29     if (a.size() >= b.size())
30         return multiply(a, b);
31     else
32         return multiply(b, a);
33 }
34 };
35 } // namespace math
36 } // namespace lib
37
38 #endif

```

1.15. Lagrange

```

1 #ifndef _LIB_LAGRANGE
2 #define _LIB_LAGRANGE
3 #include <bits/stdc++.h>
4 #include "Combinatorics.cpp"
5
6 namespace lib {
7     using namespace std;
8     namespace linalg {
9         template <typename Field> struct PrefixLagrange {
10             vector<Field> pref, suf;
11             PrefixLagrange() {}
12
13             void ensure(int n) {
14                 int o = pref.size();
15                 if (n <= o)
16                     return;
17                 pref.resize(n), suf.resize(n);
18             }
19
20             template <typename T> Field eval(const vector<Field> &v, T x) {
21                 using C = Combinatorics<Field>;
22                 assert(!v.empty());
23                 int d = (int)v.size() - 1;
24                 if (x <= d)
25                     return v[x];
26
27                 ensure(d + 1);
28
29                 Field a = x;
30                 pref[0] = suf[d] = 1;
31                 for (T i = 0; i < d; i++)
32                     pref[i + 1] = pref[i] * a, a -= 1;
33                 for (T i = d; i; i--)
34                     suf[i - 1] = suf[i] * a, a += 1;

```

```

35     Field ans = 0;
36     for (int i = 0; i <= d; i++) {
37         Field l = pref[i] * suf[i] * C::ifactorial(i) * C::ifactorial(d-i) *
38             v[i];
39         if ((d + i) & 1)
40             l = -l;
41         ans += l;
42     }
43     return ans;
44 }
45 };
46
47 template<typename T, typename U>
48 T lagrange_iota(const vector<T>& f, U n) {
49     static PrefixLagrange<T> lag;
50     return lag.eval(f, n);
51 }
52
53 template<typename T, typename U>
54 T lagrange_iota_sum(const vector<T>& f, U n) {
55     int m = f.size();
56     vector<T> g(m + 1);
57     for(int i = 1; i <= m; i++)
58         g[i] = g[i-1] + f[i-1];
59     return lagrange_iota(g, n);
60 }
61 } // namespace linalg
62 } // namespace lib
63
64 #endif

```

1.16. LinearProgram

```

1 #ifndef _LIB_LINEAR_PROGRAM
2 #define _LIB_LINEAR_PROGRAM
3 #include "Simplex.cpp"
4 #include "Symbolic.cpp"
5 #include <bits/stdc++.h>
6
7 namespace lib {
8     using namespace std;
9     template <typename T = double> struct LinearProgram {
10         struct ConstraintVisitor : StackVisitor<T> {
11             const vector<Variable<T>> &vars;
12             const vector<Constraint<T>> &consts;
13
14             vector<vector<T>> A;
15             vector<T> b;
16             T mult;
17
18             ConstraintVisitor(const vector<Variable<T>> &vars,
19                             const vector<Constraint<T>> &consts)
20                 : vars(vars), consts(consts), mult(1) {
21                 A = vector<vector<T>>();
22                 b = vector<T>();
23             }
24
25             void populate() {
26                 for (int i = 0; i < consts.size(); i++) {
27                     const auto &constraint = consts[i];
28                     if (constraint.op == ConstraintOperation::less_eq)
29                         visit_constraint(constraint, 1);
30                     else if (constraint.op == ConstraintOperation::greater_eq)

```

```

31     visit_constraint(constraint, -1);
32     else if (constraint.op == ConstraintOperation::equals)
33         visit_constraint(constraint, 1), visit_constraint(constraint, -1);
34 }
35
36 // for(int i = 0; i < b.size(); i++) {
37 //     for(int j = 0; j < vars.size(); j++) {
38 //         cout << A[i][j] << " ";
39 //     }
40 //     cout << b[i] << endl;
41 // }
42 }
43
44 void visit_constraint(const Constraint<T> &constraint, T
constraint_mult) {
45     A.emplace_back(vars.size());
46     b.emplace_back();
47     mult *= constraint_mult;
48     this->visit(constraint.lhs);
49     mult = -mult;
50     this->visit(constraint.rhs);
51     mult = -mult;
52     mult *= constraint_mult;
53 }
54
55 int index(const Variable<T> &v) const {
56     return lower_bound(vars.begin(), vars.end(), v) - vars.begin();
57 }
58
59 virtual void visit_variable(const Expression<T> &e) override {
60     A.back()[index(e->var)] += this->top() * e->coef * mult;
61 }
62 virtual void visit_literal(const Expression<T> &e) override {
63     b.back() -= this->top() * e->coef * mult;
64 }
65 };
66
67 struct ObjectiveVisitor : StackVisitor<T> {
68     const vector<Variable<T>> &vars;
69     const Expression<T> &obj;
70
71     vector<T> c;
72     T mult;
73
74     ObjectiveVisitor(const vector<Variable<T>> &vars, const Expression<T>
&obj,
75                     T mult)
76         : vars(vars), obj(obj), mult(mult) {
77         c = vector<T>(vars.size());
78     }
79
80     void populate() {
81         this->visit(obj);
82         // cout << "---" << endl;
83         // for(int i = 0; i < vars.size(); i++)
84         //     cout << c[i] << " ";
85         // cout << endl;
86     }
87
88     int index(const Variable<T> &v) const {
89         return lower_bound(vars.begin(), vars.end(), v) - vars.begin();
90     }
91
92     virtual void visit_variable(const Expression<T> &e) override {
93         c[index(e->var)] += this->top() * e->coef * mult;

```

```

94     }
95 };
96
97 vector<Constraint<T>> constraints;
98 void add_constraint(Constraint<T> constraint) {
99     constraints.push_back(constraint);
100 }
101 set<Variable<T>> get_variables(const Expression<T> &obj) const {
102     auto visitor = make_unique<VariableVisitor<T>>();
103     for (const auto &c : constraints) {
104         visitor->visit(c.lhs);
105         visitor->visit(c.rhs);
106     }
107     visitor->visit(obj);
108     return visitor->seen;
109 }
110 map<Variable<T>, T> _solve(const Expression<T> &obj, T obj_mult = 1) {
111     const auto &variables = get_variables(obj);
112     vector<Variable<T>> vs(variables.begin(), variables.end());
113     auto visitor = make_unique<ConstraintVisitor>(vs, constraints);
114     visitor->populate();
115     auto objVisitor = make_unique<ObjectiveVisitor>(vs, obj, obj_mult);
116     objVisitor->populate();
117
118     LPSolver<T> solver(visitor->A, visitor->b, objVisitor->c);
119     vector<T> ans;
120     solver.Solve(ans);
121     if (ans.size() < vs.size())
122         return {};
123
124     map<Variable<T>, T> res;
125     for (int i = 0; i < vs.size(); i++)
126         res[vs[i]] = ans[i];
127     return res;
128 }
129
130 map<Variable<T>, T> maximize(const Expression<T> &obj) { return
_solve(obj); }
131
132 map<Variable<T>, T> minimize(const Expression<T> &obj) {
133     return _solve(obj, -1);
134 }
135 };
136 } // namespace lib
137
138 #endif

```

1.17. LinearRecurrence

```

1 #ifndef _LIB_LINEAR_RECURRENCE
2 #define _LIB_LINEAR_RECURRENCE
3 #include "PolynomialRing.cpp"
4 #include "Traits.cpp"
5 #include <bits/stdc++.h>
6
7 namespace lib {
8     using namespace std;
9     namespace linalg {
10         namespace {
11             using traits::HasRandomIterator;
12             using traits::IsRandomIterator;
13         } // namespace
14
15         template <typename P> struct BMSolver {

```

```

16 typedef BMSolver<P> type;
17 typedef typename P::field_type field_type;
18 typedef P poly_type;
19
20 vector<field_type> base;
21 vector<field_type> T;
22
23 template <
24     typename Iterator,
25     typename enable_if<IsRandomIterator<Iterator>::value::type * =
26         nullptr>
27 void solve(Iterator begin, Iterator end) {
28     auto get = [begin](int i) { return *(begin + i); };
29
30     int n = distance(begin, end);
31
32     vector<field_type> C = {1}, B = {1};
33     field_type b = 1;
34     int L = 0;
35
36     for (int i = 0, x = 1; i < n; i++, x++) {
37         // evaluate new element
38         field_type d = 0;
39         for (size_t j = 0; j < C.size(); j++)
40             d += get(i - j) * C[j];
41         if (d == 0)
42             continue;
43         if (2 * L <= i) {
44             auto tmp = C;
45             if (C.size() < B.size() + x)
46                 C.resize(B.size() + x);
47             field_type coef = d / b;
48             for (size_t j = 0; j < B.size(); j++)
49                 C[j + x] -= coef * B[j];
50             L = i + 1 - L;
51             B = tmp;
52             b = d;
53             x = 0;
54         } else {
55             if (C.size() < B.size() + x)
56                 C.resize(B.size() + x);
57             field_type coef = d / b;
58             for (size_t j = 0; j < B.size(); j++)
59                 C[j + x] -= coef * B[j];
60         }
61
62         T = vector<field_type>((int)C.size() - 1);
63         for (size_t i = 0; i < T.size(); i++)
64             T[i] = -C[i + 1];
65         base = vector<field_type>(begin, end);
66     }
67
68     template <
69         typename Container,
70         typename enable_if<HasRandomIterator<Container>::value::type * =
71             nullptr>
72     void solve(const Container &container) {
73         solve(container.begin(), container.end());
74     }
75
76     void solve(const initializer_list<field_type> &l) {
77         solve(l.begin(), l.end());
78     }

```

```

79 bool solved() const { return T.size() > 0 && base.size() >= T.size(); }
80
81 void ensure(int nsz) const {
82     auto *self = const_cast<type *>(this);
83     for (int j = base.size(); j < nsz; j++) {
84         field_type acc = 0;
85         for (int i = 0; i < (int)T.size(); i++)
86             acc += base[j - i - 1] * T[i];
87         self->base.push_back(acc);
88     }
89 }
90
91 poly_type mod_function() const {
92     poly_type res;
93     int m = T.size();
94     res[m] = 1;
95     for (int i = 0; i < m; i++)
96         res[i] = -T[m - i - 1];
97     return res;
98 }
99
100 vector<field_type> compute(long long K, int n) {
101     assert(n > 0);
102     assert(solved());
103     vector<field_type> res;
104     int N = T.size();
105     int cons = min(n, N);
106
107     if (K < (int)base.size()) {
108         for (int j = 0; j < n && K + j < (int)base.size(); j++)
109             res.push_back({base[K + j]});
110
111         while ((int)res.size() < cons) {
112             field_type acc = 0;
113             int sz = res.size();
114             int mid = min(sz, N);
115             for (int i = 0; i < mid; i++)
116                 acc += res[sz - i - 1] * T[i];
117             sz = base.size();
118             for (int i = mid; i < N; i++)
119                 acc += base[sz - 1 - (i - mid)] * T[i];
120             res.push_back(acc);
121         }
122     } else {
123         ensure(cons + N - 1);
124
125         poly_type x = poly_type::kth(K, mod_function());
126
127         for (int j = 0; j < cons; j++) {
128             field_type acc = 0;
129             for (int i = 0; i < N; i++)
130                 acc += x[i] * base[i + j];
131             res.push_back(acc);
132         }
133     }
134
135     for (int j = res.size(); j < n; j++) {
136         field_type acc = 0;
137         for (int i = 0; i < N; i++)
138             acc += res[j - i - 1] * T[i];
139         res.push_back(acc);
140     }
141     return res;
142 }
143

```



```

144 field_type compute(long long K) { return compute(K, 1)[0]; }
145 };
146
147 template<typename Poly>
148 struct LinearRecurrence {
149     typedef LinearRecurrence<Poly> type;
150     typedef typename Poly::field field_type;
151     typedef Poly poly_type;
152
153     poly_type P, Q;
154
155     LinearRecurrence(const vector<field_type>& base, vector<field_type> T) {
156         assert(base.size() == T.size());
157         assert(T.back() != field_type());
158         for(auto& x : T) x = -x;
159         T.insert(T.begin(), field_type(1));
160         Q = poly_type(T);
161         P = poly_type(base) % T.size() * Q % ((int)T.size() - 1);
162     }
163
164     template<typename I>
165     field_type compute(I N) {
166         auto P1 = P;
167         auto Q1 = Q;
168         while(N) {
169             auto Q2 = Q1;
170             for(int i = 1; i < Q2.size(); i += 2) Q2[i] = -Q2[i];
171             auto U = P1 * Q2;
172             P1 = poly_type();
173             for(int i = N % 2, j = 0; j < Q.degree(); j++, i += 2)
174                 P1[j] = U[i];
175             auto A = Q1 * Q2;
176             Q1 = poly_type();
177             for(int i = 0, j = 0; j <= Q.degree(); j++, i += 2)
178                 Q1[j] = A[i];
179             N /= 2;
180             if(N < P.size()) break;
181         }
182         return (P1 * Q1.inverse())[N];
183     }
184 };
185 } // namespace linalg
186 } // namespace lib
187
188 #endif

```

1.18. LongMultiplication

```

1 #ifndef _LIB_LONG_MULTIPLICATION
2 #define _LIB_LONG_MULTIPLICATION
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace math {
8 struct NaiveMultiplication {
9     template<typename T>
10     using Transform = void;
11
12     template <typename Field>
13     vector<Field> operator()(const vector<Field> &a,
14                             const vector<Field> &b) const {
15         vector<Field> res(a.size() + b.size());
16         for (size_t i = 0; i < a.size(); i++) {

```

```

17         for (size_t j = 0; j < b.size(); j++) {
18             res[i + j] += a[i] * b[j];
19         }
20     }
21     return res;
22 }
23 };
24
25 template <typename Mult, typename Field>
26 vector<Field> shift_conv(const vector<Field> &a, vector<Field> b) {
27     if (b.empty())
28         return {};
29     reverse(b.begin(), b.end());
30     int n = a.size();
31     int m = b.size();
32
33     auto res = Mult()(a, b);
34     return vector<Field>(res.begin() + m - 1, res.end());
35 }
36 } // namespace math
37 } // namespace lib
38
39 #endif

```

1.19. Math

```

1 #ifndef _LIB_MATH
2 #define _LIB_MATH
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace math {
8
9     /// caide keep
10     template <typename Type> struct DefaultPowerOp {
11         Type operator()() const { return Type(1); }
12         Type operator()(const Type &a) const { return a; }
13         void operator()(Type &x, const Type &a, long long cur) const {
14             x *= x;
15             if (cur & 1)
16                 x *= a;
17         }
18     };
19
20     template <typename Type, typename Op>
21     Type generic_power(const Type &a, long long n, Op op) {
22         if (n == 0)
23             return op();
24         Type res = op(a);
25         int hi = 63 - __builtin_clzll(n);
26         for (int i = hi - 1; ~i; i--) {
27             op(res, a, n >> i);
28         }
29         return res;
30     }
31
32     template <typename Type> Type generic_power(const Type &a, long long n) {
33         return generic_power(a, n, DefaultPowerOp<Type>());
34     }
35 } // namespace math
36 } // namespace lib
37
38 #endif

```

1.20. Matrix

```

1  #ifndef _LIB_MATRIX
2  #define _LIB_MATRIX
3  #include <bits/stdc++.h>
4
5  namespace lib {
6  using namespace std;
7  namespace linalg {
8
9  template <typename T>
10 struct MultCombiner {
11     inline constexpr static T default_value = 0;
12     void operator()(T& x, const T& a, const T& b) {
13         x += a * b;
14     }
15 };
16
17 template <typename T, T def = numeric_limits<T>::max(), typename Cmp =
18     less<T>>
19 struct OptCombiner {
20     inline constexpr static T default_value = def;
21     void operator()(T& x, const T& a, const T& b) {
22         x = Cmp()(a, b) ? a : b;
23     }
24 };
25
26 template <typename T, T def = numeric_limits<T>::max(), typename Cmp =
27     less<T>>
28 struct OptSumCombiner {
29     inline constexpr static T default_value = def;
30     void operator()(T& x, const T& a, const T& b) {
31         auto sum = a + b;
32         x = Cmp()(x, sum) ? x : sum;
33     }
34 };
35
36 template <typename T, typename Cmp = less<T>>
37 struct SafeOptSumCombiner {
38     inline constexpr static T default_value = numeric_limits<T>::max();
39     void operator()(T& x, const T& a, const T& b) {
40         if (a == default_value || b == default_value) return;
41         T sum;
42         if (!__builtin_add_overflow(a, b, &sum))
43             x = Cmp()(x, sum) ? x : sum;
44     }
45 };
46
47 template <typename T, typename Combiner = MultCombiner<T>> struct Matrix {
48     inline constexpr static T default_value = Combiner::default_value;
49     typedef long long large_int;
50     typedef initializer_list<initializer_list<T>> nested_list;
51     vector<T> g;
52     int n, m;
53
54     Matrix() {}
55     Matrix(int n, int m) : g(n * m), n(n), m(m) {}
56     Matrix(const nested_list &l) : Matrix(l.size(), l.begin()->size()) {
57         auto it1 = l.begin();
58         for (int i = 0; i < n; i++, ++it1) {
59             assert((int)it1->size() == m);
60             auto it2 = it1->begin();
61             for (int j = 0; j < m; j++, ++it2) {
62                 (*this)(i, j) = *it2;
63             }
64         }
65     }
66 };

```

```

62     }
63 }
64
65 inline int rows() const { return n; }
66 inline int cols() const { return m; }
67 inline int size() const { return n * m; }
68 inline bool is_square() const { return n == m; }
69 T operator()(const int i, const int j) const { return g[i * m + j]; }
70 T &operator()(const int i, const int j) { return g[i * m + j]; }
71
72 Matrix t() const {
73     Matrix res(m, n);
74     for (int i = 0; i < m; i++) {
75         for (int j = 0; j < n; j++) {
76             res(i, j) = (*this)(j, i);
77         }
78     }
79     return res;
80 }
81
82 Matrix &operator+=(const Matrix &rhs) {
83     assert(n == rhs.n && m == rhs.m);
84     int sz = size();
85     for (int i = 0; i < sz; i++)
86         g[i] += rhs.g[i];
87     return *this;
88 }
89
90 Matrix &operator-=(const Matrix &rhs) {
91     assert(n == rhs.n && m == rhs.m);
92     int sz = size();
93     for (int i = 0; i < sz; i++)
94         g[i] -= rhs.g[i];
95     return *this;
96 }
97
98 Matrix &operator*=(const Matrix &rhs) {
99     assert(n == rhs.n && m == rhs.m);
100     int sz = size();
101     for (int i = 0; i < sz; i++)
102         g[i] *= rhs.g[i];
103     return *this;
104 }
105
106 Matrix operator-() const {
107     Matrix res = *this;
108     for (T &t : g)
109         t = -t;
110     return res;
111 }
112
113 friend Matrix operator+(const Matrix &lhs, const Matrix &rhs) {
114     Matrix res = lhs;
115     return res += rhs;
116 }
117
118 friend Matrix<T> operator-(const Matrix &lhs, const Matrix &rhs) {
119     Matrix res = lhs;
120     return res -= rhs;
121 }
122
123 friend Matrix operator*(const Matrix &lhs, const Matrix &rhs) {
124     Matrix res = lhs;
125     return res *= rhs;
126 }
127
128 friend Matrix operator%(const Matrix &lhs, const Matrix &rhs) {
129     assert(lhs.m == rhs.n);
130     auto res = Matrix::same(lhs.n, rhs.m, Combiner::default_value);
131     Combiner combiner;
132     for (int i = 0; i < lhs.n; i++) {

```

```

127     for (int k = 0; k < lhs.m; k++) {
128         for (int j = 0; j < rhs.m; j++) {
129             combiner(res(i, j), lhs(i, k), rhs(k, j));
130         }
131     }
132 }
133 return res;
134 }
135
136 static Matrix id(int n) {
137     Matrix res(n, n);
138     for (int i = 0; i < n; i++)
139         res(i, i) = 1;
140     return res;
141 }
142
143 static Matrix ones(int n, int m) {
144     return same(n, m, 1);
145 }
146
147 static Matrix same(int n, int m, T x) {
148     Matrix res(n, m);
149     res.fill(x);
150     return res;
151 }
152
153 static Matrix _power(const Matrix &a, large_int p) {
154     if (p == 1)
155         return a;
156     Matrix res = power(a, p >> 1);
157     res = res % res;
158     if (p & 1)
159         res = res % a;
160     return res;
161 }
162
163 static Matrix power(const Matrix &a, large_int p) {
164     assert(p >= 0);
165     if (p == 0) {
166         assert(a.is_square());
167         return Matrix::id(a.n);
168     } else if (p == 1)
169         return a;
170     else
171         return _power(a, p);
172 }
173
174 friend Matrix operator^(const Matrix &lhs, const large_int rhs) {
175     return Matrix::power(lhs, rhs);
176 }
177
178 inline void fill(T x) {
179     for (T &t : g)
180         t = x;
181 }
182
183 friend bool operator==(const Matrix &lhs, const Matrix &rhs) {
184     assert(lhs.n == rhs.n && lhs.m == rhs.m);
185     int sz = size();
186     for (int i = 0; i < sz; i++)
187         if (lhs.g[i] != rhs.g[i])
188             return false;
189     return true;
190 }
191 friend bool operator!=(const Matrix &lhs, const Matrix &rhs) {

```

```

192     return !(lhs == rhs);
193 }
194
195 friend istream &operator>>(istream &input, Matrix &var) {
196     for (T &t : var.g)
197         input >> t;
198     return input;
199 }
200 friend ostream &operator<<(ostream &output, Matrix &var) {
201     for (int i = 0; i < var.n; i++) {
202         if (i == 0)
203             output << "[";
204         else
205             output << " ";
206         for (int j = 0; j < var.m; j++) {
207             if (j)
208                 output << " ";
209             output << var(i, j);
210         }
211         output << "\n";
212     }
213     return output << "]";
214 }
215 };
216 } // namespace linalg
217 } // namespace lib
218
219 #endif

```

1.21. Maxflow

```

1  #ifndef _LIB_MAX_FLOW
2  #define _LIB_MAX_FLOW
3  #include "Graph.cpp"
4  #include <bits/stdc++.h>
5  // TODO: L-R flow
6
7  namespace lib {
8      using namespace std;
9      namespace flow {
10         template <typename T, typename E> struct Edge {
11             T cap;
12             bool original;
13             E label;
14         };
15         template <typename T> struct Edge<T, void> {
16             T cap;
17             bool original;
18         };
19
20         template <typename T, typename E = void> struct Maxflow {
21             typedef Maxflow<T, E> type;
22             typedef Edge<T, E> flow_edge_type;
23             typedef lib::graph::DirectedGraph<void, flow_edge_type> graph;
24             using edge_type = typename graph::edge_type;
25
26             graph g;
27             int source, sink;
28             vector<bool> visited;
29             vector<int> dist;
30             vector<size_t> used;
31
32             explicit Maxflow(int n) : g(n), source(n - 2), sink(n - 1) { assert(n >=
33                 2); }

```

```

33 void setup(int a, int b) { source = a, sink = b; }
34 void add_fake_edge(int u, int v, T weight) {
35     g.add_edge(u, v) = {weight, false};
36     g.add_edge(v, u) = {0, false};
37 }
38 template <typename S = E,
39           typename enable_if<is_void<S>::value>::type * = nullptr>
40 void add_edge(int u, int v, T weight = 1) {
41     g.add_edge(u, v) = {weight, true};
42     g.add_edge(v, u) = {0, true};
43 }
44 template <typename S = E,
45           typename enable_if<!is_void<S>::value>::type * = nullptr>
46 void add_edge(int u, int v, T weight = 1, S data = S()) {
47     g.add_edge(u, v) = {weight, true, data};
48     g.add_edge(v, u) = {0, true, S()};
49 }
50 inline int size() const { return g.size(); }
51 inline int edge_size() const { return g.edge_size(); }
52 edge_type reverse(int i) const { return g.edge(i ^ 1); }
53 edge_type edge(int i) const { return g.edge(i); }
54 flow_edge_type &flow_edge(int i) { return g.edge(i).data; }
55 flow_edge_type &reverse_flow_edge(int i) { return g.edge(i ^ 1).data; }
56
57 bool layered_bfs() {
58     int n = size();
59     dist.assign(n, -1);
60     dist[source] = 0;
61     vector<int> q;
62     q.reserve(n);
63     q.push_back(source);
64
65     for (size_t i = 0; i < q.size(); i++) {
66         int u = q[i];
67         if (u == sink)
68             break;
69         for (const auto &e : g.n_edges(u)) {
70             if (dist[e.to] == -1 && e.data.cap > 0) {
71                 dist[e.to] = dist[u] + 1;
72                 q.push_back(e.to);
73             }
74         }
75     }
76
77     return dist[sink] != -1;
78 }
79
80 T augmenting_path(const int u, const T bottle) {
81     if (!bottle)
82         return 0;
83     if (u == sink)
84         return bottle;
85     for (size_t &i = used[u]; i < g.adj[u].size(); i++) {
86         int x = g.adj[u][i];
87         auto &e = g.edge(x);
88         if (dist[e.to] != dist[u] + 1)
89             continue;
90         T cf = augmenting_path(e.to, min(bottle, e.data.cap));
91         e.data.cap -= cf;
92         g.edge(x ^ 1).data.cap += cf;
93         if (cf)
94             return cf;
95     }
96     return 0;
97 }

```

```

98
99 T blocking_flow() {
100     if (!layered_bfs())
101         return 0;
102     used.assign(size(), 0);
103     T aug, flow = 0;
104     while ((aug = augmenting_path(source, numeric_limits<T>::max())))
105         flow += aug;
106     return flow;
107 }
108
109 T maxflow() {
110     T aug, flow = 0;
111     while ((aug = blocking_flow()))
112         flow += aug;
113     return flow;
114 }
115
116 vector<bool> mincut() const {
117     int n = size();
118     vector<bool> vis(n);
119     vector<int> q;
120     q.reserve(n);
121     q.push_back(source);
122     vis[source] = true;
123     for (size_t i = 0; i < q.size(); i++) {
124         int u = q[i];
125         for (const auto &e : g.n_edges(u)) {
126             if (e.data.cap > 0 && !vis[e.to]) {
127                 q.push_back(e.to);
128                 vis[e.to] = true;
129             }
130         }
131     }
132     return vis;
133 }
134 };
135 } // namespace flow
136 } // namespace lib
137
138 #endif

```

1.22. ModularInteger

```

1 #ifndef _LIB_MODULAR_INTEGER
2 #define _LIB_MODULAR_INTEGER
3 #include "NumberTheory.cpp"
4 #include <bits/stdc++.h>
5
6 #if __cplusplus < 201300
7 #error required(c++14)
8 #endif
9
10 namespace lib {
11 using namespace std;
12 namespace {
13 template <typename T, T... Mods> struct ModularIntegerBase {
14     typedef ModularIntegerBase<T, Mods...> type;
15
16     T x[sizeof...(Mods)];
17     friend ostream &operator<<(ostream &output, const type &var) {
18         output << "(";
19         for (int i = 0; i < sizeof...(Mods); i++) {
20             if (i)

```

```

21     output << ", ";
22     output << var.x[i];
23 }
24 return output << ")";
25 }
26 };
27
28 template <typename T, T Mod> struct ModularIntegerBase<T, Mod> {
29     typedef ModularIntegerBase<T, Mod> type;
30     constexpr static T mod = Mod;
31
32     T x[1];
33
34     T& data() { return this->x[0]; }
35     T data() const { return this->x[0]; }
36     explicit operator int() const { return this->x[0]; }
37     explicit operator int64_t() const { return this->x[0]; }
38     explicit operator double() const { return this->x[0]; }
39     explicit operator long double() const { return this->x[0]; }
40     friend ostream &operator<<(ostream &output, const type &var) {
41         return output << var.x[0];
42     }
43 };
44
45 template<typename T, typename U, T... Mods>
46 struct InversesTable {
47     constexpr static size_t n_mods = sizeof...(Mods);
48     constexpr static T mods[sizeof...(Mods)] = {Mods...};
49     constexpr static int n_inverses = 1e6 + 10;
50
51     T v[n_inverses][n_mods];
52     T max_x;
53
54     InversesTable() : v(), max_x(n_inverses) {
55         for(int j = 0; j < sizeof...(Mods); j++)
56             v[1][j] = 1, max_x = min(max_x, mods[j]);
57         for(int i = 2; i < max_x; i++) {
58             for(int j = 0; j < sizeof...(Mods); j++) {
59                 v[i][j] = mods[j] - (T)((U)(mods[j] / i) * v[mods[j] % i][j] %
60                     mods[j]);
61             }
62         }
63     };
64
65     // Make available for linkage.
66     template <typename T, class U, T... Mods>
67     constexpr T InversesTable<T, U, Mods...>::mods[];
68
69     template <typename T, class Enable, T... Mods>
70     struct ModularIntegerImpl : ModularIntegerBase<T, Mods...> {
71         typedef ModularIntegerImpl<T, Enable, Mods...> type;
72         typedef T type_int;
73         typedef uint64_t large_int;
74         constexpr static size_t n_mods = sizeof...(Mods);
75         constexpr static T mods[sizeof...(Mods)] = {Mods...};
76         using ModularIntegerBase<T, Mods...>::x;
77         using Inverses = InversesTable<T, large_int, Mods...>;
78
79         struct Less {
80             bool operator()(const type &lhs, const type &rhs) const {
81                 for (size_t i = 0; i < sizeof...(Mods); i++)
82                     if (lhs.x[i] != rhs.x[i])
83                         return lhs.x[i] < rhs.x[i];
84                 return false;

```

```

85     };
86 };
87 typedef Less less;
88
89
90 constexpr ModularIntegerImpl() {
91     for (size_t i = 0; i < sizeof...(Mods); i++)
92         x[i] = T();
93 }
94 constexpr ModularIntegerImpl(large_int y) {
95     for (size_t i = 0; i < sizeof...(Mods); i++) {
96         x[i] = y % mods[i];
97         if (x[i] < 0)
98             x[i] += mods[i];
99     }
100 }
101 static type with_remainders(T y[sizeof...(Mods)]) {
102     type res;
103     for (size_t i = 0; i < sizeof...(Mods); i++)
104         res.x[i] = y[i];
105     res.normalize();
106     return res;
107 }
108
109 inline void normalize() {
110     for (size_t i = 0; i < sizeof...(Mods); i++)
111         if ((x[i] %= mods[i]) < 0)
112             x[i] += mods[i];
113 }
114
115 inline T operator[](int i) const { return x[i]; }
116
117 inline T multiply(T a, T b, T mod) const { return (large_int)a * b % mod; }
118
119 inline T inv(T a, T mod) const { return static_cast<T>(nt::inverse(a,
120     mod)); }
121
122 inline T invi(T a, int i) const {
123     const static Inverses inverses = Inverses();
124     if (a < inverses.max_x)
125         return inverses.v[a][i];
126     return inv(a, mods[i]);
127 }
128
129 type inverse() const {
130     T res[sizeof...(Mods)];
131     for (size_t i = 0; i < sizeof...(Mods); i++)
132         res[i] = invi(x[i], i);
133     return type::with_remainders(res);
134 }
135
136 template <typename U> T power_(T a, U p, T mod) {
137     if (mod == 1)
138         return T();
139     if (p < 0) {
140         if (a == 0)
141             throw domain_error("0^p with negative p is invalid");
142         p = -p;
143         a = inv(a, mod);
144     }
145     if (p == 0)
146         return T(1);
147     if (p == 1)
148         return a;
149     T res = 1;

```

```

149     while (p > 0) {
150         if (p & 1)
151             res = multiply(res, a, mod);
152         p >>= 1;
153         a = multiply(a, a, mod);
154     }
155     return res;
156 }
157
158 inline type &operator+=(const type &rhs) {
159     for (size_t i = 0; i < sizeof...(Mods); i++)
160         if ((x[i] += rhs.x[i]) >= mods[i])
161             x[i] -= mods[i];
162     return *this;
163 }
164 inline type &operator-=(const type &rhs) {
165     for (size_t i = 0; i < sizeof...(Mods); i++)
166         if ((x[i] -= rhs.x[i]) < 0)
167             x[i] += mods[i];
168     return *this;
169 }
170 inline type &operator*=(const type &rhs) {
171     for (size_t i = 0; i < sizeof...(Mods); i++)
172         x[i] = multiply(x[i], rhs.x[i], mods[i]);
173     return *this;
174 }
175 inline type &operator/=(const type &rhs) {
176     for (size_t i = 0; i < sizeof...(Mods); i++)
177         x[i] = multiply(x[i], invi(rhs.x[i], i), mods[i]);
178     return *this;
179 }
180
181 inline type &operator+=(T rhs) {
182     for (size_t i = 0; i < sizeof...(Mods); i++)
183         if ((x[i] += rhs) >= mods[i])
184             x[i] -= mods[i];
185     return *this;
186 }
187
188 type &operator-=(T rhs) {
189     for (size_t i = 0; i < sizeof...(Mods); i++)
190         if ((x[i] -= rhs) < 0)
191             x[i] += mods[i];
192     return *this;
193 }
194
195 type &operator*=(T rhs) {
196     for (size_t i = 0; i < sizeof...(Mods); i++)
197         x[i] = multiply(x[i], rhs, mods[i]);
198     return *this;
199 }
200
201 type &operator/=(T rhs) {
202     for (size_t i = 0; i < sizeof...(Mods); i++)
203         x[i] = multiply(invi(rhs, i), x[i], mods[i]);
204     return *this;
205 }
206
207 type &operator^=(large_int p) {
208     for (size_t i = 0; i < sizeof...(Mods); i++)
209         x[i] = power_(x[i], p, mods[i]);
210     return *this;
211 }
212
213 type &operator++() {

```

```

214     for (size_t i = 0; i < sizeof...(Mods); i++)
215         if ((++x[i]) >= mods[i])
216             x[i] -= mods[i];
217     return *this;
218 }
219 type &operator--() {
220     for (size_t i = 0; i < sizeof...(Mods); i++)
221         if ((--x[i]) < 0)
222             x[i] += mods[i];
223     return *this;
224 }
225 type operator++(int unused) {
226     type res = *this;
227     ++(*this);
228     return res;
229 }
230 type operator--(int unused) {
231     type res = *this;
232     --(*this);
233     return res;
234 }
235
236 friend type operator+(const type &lhs, const type &rhs) {
237     type res = lhs;
238     return res += rhs;
239 }
240 friend type operator-(const type &lhs, const type &rhs) {
241     type res = lhs;
242     return res -= rhs;
243 }
244 friend type operator*(const type &lhs, const type &rhs) {
245     type res = lhs;
246     return res *= rhs;
247 }
248 friend type operator/(const type &lhs, const type &rhs) {
249     type res = lhs;
250     return res /= rhs;
251 }
252
253 friend type operator+(const type &lhs, T rhs) {
254     type res = lhs;
255     return res += rhs;
256 }
257
258 friend type operator-(const type &lhs, T rhs) {
259     type res = lhs;
260     return res -= rhs;
261 }
262
263 friend type operator*(const type &lhs, T rhs) {
264     type res = lhs;
265     return res *= rhs;
266 }
267
268 friend type operator/(const type &lhs, T rhs) {
269     type res = lhs;
270     return res /= rhs;
271 }
272
273 friend type operator^(const type &lhs, large_int rhs) {
274     type res = lhs;
275     return res ^= rhs;
276 }
277
278 friend type power(const type &lhs, large_int rhs) { return lhs ^ rhs; }

```

```

279
280 type operator-() const {
281     type res = *this;
282     for (size_t i = 0; i < sizeof...(Mods); i++)
283         if (res.x[i])
284             res.x[i] = mods[i] - res.x[i];
285     return res;
286 }
287
288 friend bool operator==(const type &lhs, const type &rhs) {
289     for (size_t i = 0; i < sizeof...(Mods); i++)
290         if (lhs.x[i] != rhs.x[i])
291             return false;
292     return true;
293 }
294 friend bool operator!=(const type &lhs, const type &rhs) {
295     return !(lhs == rhs);
296 }
297
298 friend istream &operator>>(istream &input, type &var) {
299     T y;
300     cin >> y;
301     var = y;
302     return input;
303 }
304 };
305 } // namespace
306
307 // Explicitly make constexpr available for linkage.
308 template <typename T, class Enable, T... Mods>
309 constexpr T ModularIntegerImpl<T, Enable, Mods...>::mods[];
310
311 template <typename T, T... Mods>
312 using ModularInteger =
313     ModularIntegerImpl<T, typename enable_if<is_integral<T>::value>::type,
314         Mods...>;
315
316 template <int32_t... Mods> using Mint32 = ModularInteger<int32_t, Mods...>;
317
318 template <int64_t... Mods> using Mint64 = ModularInteger<int64_t, Mods...>;
319
320 using MintP = Mint32<(int32_t)1e9+7>;
321 using MintNTT = Mint32<998244353>;
322 } // namespace lib
323
324 #endif

```

1.23. MontgomeryInteger

```

1 #ifndef _LIB_MONTGOMERY_INTEGER
2 #define _LIB_MONTGOMERY_INTEGER
3 #include <bits/stdc++.h>
4
5 #if __cplusplus < 201300
6 #error required(c++14)
7 #endif
8
9 namespace lib {
10 using namespace std;
11 namespace {
12 template <typename U, U Mod>
13 struct MontgomeryIntegerImpl {
14     using S = make_signed_t<U>;
15     using T = make_unsigned_t<U>;

```

```

16 typedef MontgomeryIntegerImpl<U, Mod> type;
17 constexpr static T mod = (T)Mod;
18
19 T x;
20 typedef U type_int;
21 typedef uint64_t large_int;
22
23 constexpr static T get_r() {
24     T ret = Mod;
25     for(int i = 0; i < 4; i++)
26         ret *= 2 - mod * ret;
27     return ret;
28 }
29
30 constexpr static T r = get_r();
31 constexpr static T n2 = -large_int(mod) % mod;
32 static_assert(r * mod == 1, "assert(r * mod == 1)");
33 static_assert(mod < (1 << 30), "assert(mod < 2^30)");
34 static_assert(mod % 2 == 1, "assert(mod % 2 == 1)");
35
36 constexpr MontgomeryIntegerImpl() : x(0) {}
37 constexpr MontgomeryIntegerImpl(large_int y)
38     : x(reduce(large_int(y % mod + mod) * n2)) {}
39
40 constexpr inline static T reduce(large_int y) {
41     return (y + large_int(T(y) * T(-r)) * mod) >> 32;
42 }
43
44 constexpr inline type &operator+=(const type &rhs) {
45     if(S(x += rhs.x - 2 * mod) < 0) x += 2 * mod;
46     return *this;
47 }
48 constexpr inline type &operator-=(const type &rhs) {
49     if(S(x -= rhs.x) < 0) x += 2 * mod;
50     return *this;
51 }
52 constexpr inline type &operator*=(const type &rhs) {
53     x = reduce(large_int(x) * rhs.x);
54     return *this;
55 }
56 constexpr inline type &operator/=(const type &rhs) {
57     return *this *= rhs.inverse();
58 }
59
60 constexpr inline type inverse() const {
61     return (*this).power(large_int(mod - 2));
62 }
63
64 constexpr type &operator^=(large_int p) {
65     return *this = power(p);
66 }
67
68 constexpr type &operator++() {
69     return *this += type(1);
70 }
71 constexpr type &operator--() {
72     return *this -= type(1);
73 }
74 constexpr type operator++(int unused) {
75     type res = *this;
76     ++(*this);
77     return res;
78 }
79 constexpr type operator--(int unused) {
80     type res = *this;

```

```

81     --(*this);
82     return res;
83 }
84
85 friend constexpr type operator+(const type &lhs, const type &rhs) {
86     type res = lhs;
87     return res += rhs;
88 }
89 friend constexpr type operator-(const type &lhs, const type &rhs) {
90     type res = lhs;
91     return res -= rhs;
92 }
93 friend constexpr type operator*(const type &lhs, const type &rhs) {
94     type res = lhs;
95     return res *= rhs;
96 }
97 friend constexpr type operator/(const type &lhs, const type &rhs) {
98     type res = lhs;
99     return res /= rhs;
100 }
101
102 friend constexpr type operator^(const type &lhs, large_int rhs) {
103     type res = lhs;
104     return res ^= rhs;
105 }
106
107 friend constexpr type power(const type &lhs, large_int rhs) {
108     return lhs.power(rhs);
109 }
110
111 constexpr type operator-() const {
112     return type() - *this;
113 }
114
115 constexpr type power(large_int rhs) const {
116     type ret(1), mul(*this);
117     while(rhs > 0) {
118         if(rhs&1) ret *= mul;
119         mul *= mul;
120         rhs /= 2;
121     }
122     return ret;
123 }
124
125 constexpr T get() const {
126     T ret = reduce(x);
127     return ret >= mod ? ret - mod : ret;
128 }
129
130 friend bool operator==(const type &lhs, const type &rhs) {
131     return lhs.get() == rhs.get();
132 }
133 friend bool operator!=(const type &lhs, const type &rhs) {
134     return !(lhs == rhs);
135 }
136
137 explicit operator int() const { return get(); }
138 explicit operator int64_t() const { return get(); }
139 explicit operator long long() const { return get(); }
140 explicit operator double() const { return get(); }
141 explicit operator long double() const { return get(); }
142 friend ostream &operator<<(ostream &output, const type &var) {
143     return output << var.get();
144 }
145

```

```

146     friend istream &operator>>(istream &input, type &var) {
147         T y;
148         cin >> y;
149         var = type(y);
150         return input;
151     }
152 };
153 } // namespace
154
155 template <typename T, T... Mods>
156 using MontgomeryInteger = MontgomeryIntegerImpl<T, Mods...>;
157
158 template <int32_t... Mods> using Mont32 = MontgomeryInteger<int32_t,
159     Mods...>;
160
161 using MontP = Mont32<(int32_t)1e9+7>;
162 using MontNTT = Mont32<998244353>;
163 } // namespace lib
164
165 #endif

```

1.24. NTT

```

1  #ifndef _LIB_NTT
2  #define _LIB_NTT
3  #include <bits/stdc++.h>
4  #include "DFT.cpp"
5  #include "NumberTheory.cpp"
6  #include "VectorN.cpp"
7
8  namespace lib {
9      using namespace std;
10     namespace linalg {
11         template<typename T>
12         struct MintRootProvider {
13             static size_t max_sz;
14             static T g;
15             static vector<T> w;
16
17             MintRootProvider() {
18                 if(g == 0) {
19                     auto acc = T::mod-1;
20                     while(acc % 2 == 0) acc /= 2, max_sz++;
21
22                     auto factors = nt::factors(T::mod - 1);
23                     for(g = 2; (typename T::type_int)g < T::mod; g++) {
24                         bool ok = true;
25                         for(auto f : factors) {
26                             if(power(g, (T::mod-1)/f) == 1) {
27                                 ok = false;
28                                 break;
29                             }
30                         }
31                         if(ok) break;
32                     }
33                     assert(g != 0);
34                 }
35             }
36
37             pair<T, T> roots(int num, int den) {
38                 auto p = g ^ ((long long)(T::mod - 1) / den * num);
39                 return {p, p.inverse()};
40             }

```



```

41 T operator()(int n, int k) {
42     return power(g, (T::mod-1)/(n/k));
43 }
44
45 void operator()(int n) {
46     n = max(n, 2);
47     int k = max((int)w.size(), 2);
48     assert(n <= (1LL << max_sz));
49     if ((int)w.size() < n)
50         w.resize(n);
51     else
52         return;
53     w[0] = w[1] = 1;
54     for (; k < n; k *= 2) {
55         T step = power(g, (T::mod-1)/(2*k));
56         for(int i = k; i < 2*k; i++)
57             w[i] = (i&1) ? w[i/2] * step : w[i/2];
58     }
59 }
60 T operator[](int i) {
61     return w[i];
62 }
63
64 T inverse(int n) {
65     return T(1) / n;
66 }
67 };
68
69 template<typename T>
70 size_t MintRootProvider<T>::max_sz = 1;
71 template<typename T>
72 T MintRootProvider<T>::g = T();
73 template<typename T>
74 vector<T> MintRootProvider<T>::w = vector<T>();
75
76 template<typename T>
77 struct NTT : public DFT<T, MintRootProvider<T>> {
78     using Parent = DFT<T, MintRootProvider<T>>;
79     using Parent::fa;
80     using Parent::dft;
81     using Parent::idft;
82
83     static void _convolve(const vector<T> &a) {
84         int n = Parent::ensure(a.size(), a.size());
85         for (size_t i = 0; i < (size_t)n; i++)
86             fa[i] = i < a.size() ? a[i] : T();
87         Parent::dft(n);
88         for (int i = 0; i < n; i++)
89             fa[i] *= fa[i];
90         Parent::idft(n);
91     }
92
93     static void _convolve(const vector<T> &a, const vector<T> &b) {
94         if(std::addressof(a) == std::addressof(b))
95             return _convolve(a);
96         int n = Parent::ensure(a.size(), b.size());
97         for (size_t i = 0; i < (size_t)n; i++)
98             fa[i] = i < a.size() ? a[i] : T();
99         Parent::dft(n);
100         // TODO: have a buffer for this
101         auto fb = retrieve<Parent, T>(n);
102         for (size_t i = 0; i < (size_t)n; i++)
103             fa[i] = i < b.size() ? b[i] : T();
104         Parent::dft(n);
105         for (int i = 0; i < n; i++)

```

```

106         fa[i] *= fb[i];
107         Parent::idft(n);
108     }
109
110     static vector<T> convolve(const vector<T>& a, const vector<T>& b) {
111         int sz = (int)a.size() + b.size() - 1;
112         _convolve(a, b);
113         return retrieve<Parent, T>(sz);
114     }
115
116     static VectorN<T> transform(vector<T> a, int n) {
117         a.resize(n);
118         Parent::dft(a, n);
119         return a;
120     }
121
122     static vector<T> itransform(vector<T> a, int n) {
123         int sz = a.size();
124         Parent::idft(a, sz);
125         a.resize(min(n, sz));
126         return a;
127     }
128 };
129
130 struct NTTMultiplication {
131     template<typename T>
132     using Transform = linalg::NTT<T>;
133
134     template<typename Field>
135     vector<Field> operator()(const vector<Field> &a,
136                             const vector<Field> &b) const {
137         return linalg::NTT<Field>::convolve(a, b);
138     };
139
140     template<typename Field>
141     inline VectorN<Field> transform(int n, const vector<Field>& p) const {
142         int np = next_power_of_two(n);
143         return linalg::NTT<Field>::transform(p, np);
144     }
145
146     template<typename Field>
147     inline vector<Field> itransform(int n, const vector<Field>& p) const {
148         return linalg::NTT<Field>::itransform(p, n);
149     }
150
151     template<typename Field, typename Functor, typename ...Ts>
152     inline vector<Field> on_transform(
153         int n,
154         Functor& f,
155         const vector<Ts>&&... vs) const {
156         int np = next_power_of_two(n);
157         return linalg::NTT<Field>::itransform(
158             f(n, linalg::NTT<Field>::transform(vs, np)...), n);
159     }
160 };
161 // namespace lib
162
163 #endif

```

1.25. NumberTheory

```

1 #ifndef _LIB_NUMBER_THEORY
2 #define _LIB_NUMBER_THEORY

```

```

3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace nt {
8 int64_t inverse(int64_t a, int64_t b) {
9     long long b0 = b, t, q;
10    long long x0 = 0, x1 = 1;
11    if (b == 1)
12        return 1;
13    while (a > 1) {
14        q = a / b;
15        t = b, b = a % b, a = t;
16        t = x0, x0 = x1 - q * x0, x1 = t;
17    }
18    if (x1 < 0)
19        x1 += b0;
20    return x1;
21 }
22 template<typename T, typename U>
23 T powmod (T a, U b, U p) {
24     int res = 1;
25     while (b)
26         if (b & 1)
27             res = (int) (res * 1ll * a % p), --b;
28         else
29             a = (int) (a * 1ll * a % p), b >>= 1;
30     return res;
31 }
32 template<typename T>
33 vector<T> factors(T n) {
34     vector<T> f;
35     for(T i = 2; i*i <= n; i++) {
36         if(n % i == 0) f.push_back(i);
37         while(n % i == 0) n /= i;
38     }
39     if(n > 1) f.push_back(n);
40     return f;
41 }
42 } // namespace nt
43 } // namespace lib
44
45 #endif

```

1.26. OfflineRMQ

```

1 #ifndef _LIB_OFFLINE_RMQ
2 #define _LIB_OFFLINE_RMQ
3 #include <bits/stdc++.h>
4 #include "DSU.cpp"
5
6 namespace lib {
7 using namespace std;
8
9 // O(n + qlogn)
10 template<typename T, typename U = T>
11 vector<T> offline_rmq(const vector<T>& v, const vector<pair<U, U>>& qrs) {
12     int n = v.size();
13     vector<vector<pair<U, int>>> cont(n);
14     for(int i = 0; i < (int)qrs.size(); i++) {
15         auto p = qrs[i];
16         cont[p.second].push_back({p.first, i});
17     }
18     vector<T> ans(qrs.size());

```

```

19 CompressedDSU dsu(n);
20 vector<U> s;
21 for(int i = 0; i < n; i++) {
22     while(!s.empty() && v[s.back()] > v[i]) {
23         dsu.parent(s.back()) = i;
24         s.pop_back();
25     }
26     s.push_back(i);
27     for(auto p : cont[i]) {
28         ans[p.second] = v[dsu[p.first]];
29     }
30 }
31 return ans;
32 }
33 } // namespace lib
34
35 #endif

```

1.27. PolynomialRing

```

1 #ifndef _LIB_POLYNOMIAL_RING
2 #define _LIB_POLYNOMIAL_RING
3 #include "Epsilon.cpp"
4 #include "Math.cpp"
5 #include "ModularInteger.cpp"
6 #include "Traits.cpp"
7 #include "LongMultiplication.cpp"
8 #include "VectorN.cpp"
9 #include <bits/stdc++.h>
10
11 namespace lib {
12 using namespace std;
13 namespace math {
14 namespace poly {
15
16 namespace {
17 /// keep caide
18 using traits::IsInputIterator;
19 /// keep caide
20 using traits::HasInputIterator;
21 } // namespace
22
23 namespace detail {
24 template<class>
25 struct sfinae_true : std::true_type{};
26
27 template<class T, class Field, class Func>
28 static auto test_transform(int)
29     -> sfinae_true<decltype(
30         std::declval<T>().template on_transform<Field>(std::declval<int>()),
31         std::declval<Func>())>{};
32
33 template<class, class Field, class Func>
34 static auto test_transform(long) -> std::false_type;
35 } // detail::
36
37 template<class T, class Field, class Func =
38     std::function<VectorN<Field>(int)>>
39 struct has_transform : decltype(detail::test_transform<T, Field, Func>(0)){};
40
41 template<typename P> struct DefaultPowerOp {
42     int mod;
43     DefaultPowerOp(int mod) : mod(mod) {}

```

```

42 P operator()() const { return P(1); }
43 P operator()(const P &a) const { return a % mod; }
44 void operator()(P &x, const P &a, long long cur) const {
45     (x *= x) %= mod;
46     if (cur & 1)
47         (x *= a) %= mod;
48 }
49 };
50
51 template <typename P> struct ModPowerOp {
52     const P &mod;
53     ModPowerOp(const P &p) : mod(p) {}
54     P operator()() const { return P(1); }
55     P operator()(const P &a) { return a % mod; }
56     void operator()(P &x, const P &a, long long cur) const {
57         (x *= x) %= mod;
58         if (cur & 1)
59             (x *= a) %= mod;
60     }
61 };
62
63 template <typename P> struct ModShiftPowerOp {
64     const P &mod;
65     ModShiftPowerOp(const P &p) : mod(p) {}
66     P operator()() const { return P(1); }
67     P operator()(const P &a) { return a % mod; }
68     void operator()(P &x, const P &a, long long cur) const {
69         // if (cur < mod.degree())
70         //     x = P::kth(cur);
71         if (cur & 1)
72             (x *= (x << 1)) %= mod;
73         else
74             (x *= x) %= mod;
75     }
76 };
77
78 struct DefaultDivmod;
79 struct NaiveDivmod;
80
81 template <typename Field, typename Mult, typename Divmod = DefaultDivmod>
82 struct Polynomial {
83     constexpr static int Magic = 64;
84     constexpr static bool NaiveMod = is_same<Divmod, NaiveDivmod>::value;
85     constexpr static bool HasTransform = has_transform<Mult, Field>::value;
86     using Transform = typename Mult::template Transform<Field>;
87
88     typedef Polynomial<Field, Mult, Divmod> type;
89     typedef Field field;
90     vector<Field> p;
91
92     Polynomial() : p(0) {}
93     explicit Polynomial(Field x) : p(1, x) {}
94
95     template <
96         typename Iterator,
97         typename enable_if<IsInputIterator<Iterator>::value>::type * = nullptr>
98     Polynomial(Iterator begin, Iterator end) : p(distance(begin, end)) {
99         int i = 0;
100         for (auto it = begin; it != end; ++it, ++i)
101             p[i] = *it;
102         normalize();
103     }
104
105     template <
106         typename Container,

```

```

107         typename enable_if<HasInputIterator<Container>::value>::type * =
            nullptr>
108     Polynomial(const Container &container)
109         : Polynomial(container.begin(), container.end()) {}
110
111     Polynomial(const initializer_list<Field> &v)
112         : Polynomial(v.begin(), v.end()) {}
113
114     static type from_root(const Field &root) { return Polynomial({-root, 1}); }
115
116     void normalize() const {
117         type *self = const_cast<type *>(this);
118         int sz = self->p.size();
119         while (sz > 0 && Epsilon<>().null(self->p[sz - 1]))
120             sz--;
121         if (sz != (int)self->p.size())
122             self->p.resize(sz);
123     }
124
125     inline int size() const { return p.size(); }
126     inline int degree() const { return max((int)p.size() - 1, 0); }
127     bool null() const {
128         for (Field x : p)
129             if (!Epsilon<>().null(x))
130                 return false;
131         return true;
132     }
133
134     const vector<Field>& data() const {
135         return p;
136     }
137
138     Field eval(Field x) const {
139         Field pw = 1;
140         Field res = 0;
141         for (Field c : p) {
142             res += pw * c;
143             pw *= x;
144         }
145         return res;
146     }
147
148     inline Field operator[](const int i) const {
149         if (i >= size())
150             return 0;
151         return p[i];
152     }
153
154     inline Field &operator[](const int i) {
155         if (i >= size())
156             p.resize(i + 1);
157         return p[i];
158     }
159
160     Field operator()(const Field &x) const {
161         if (null())
162             return Field();
163         Field acc = p.back();
164         for (int i = (int)size() - 2; i >= 0; i--) {
165             acc *= x;
166             acc += p[i];
167         }
168         return acc;
169     }
170
171     type substr(int i, int sz) const {

```

```

171     int j = min(sz + i, size());
172     i = min(i, size());
173     if(i >= j) return type();
174     return type(begin(p)+i, begin(p)+j);
175 }
176
177 type &operator+=(const type &rhs) {
178     if (rhs.size() > size())
179         p.resize(rhs.size());
180     int sz = size();
181     for (int i = 0; i < sz; i++)
182         p[i] += rhs[i];
183     normalize();
184     return *this;
185 }
186
187 type &operator-=(const type &rhs) {
188     if (rhs.size() > size())
189         p.resize(rhs.size());
190     int sz = size();
191     for (int i = 0; i < sz; i++)
192         p[i] -= rhs[i];
193     normalize();
194     return *this;
195 }
196
197 static vector<Field> multiply(const vector<Field>& a, const vector<Field>&
198     b) {
199     if(min(a.size(), b.size()) < Magic)
200         return NaiveMultiplication()(a, b);
201     return Mult()(a, b);
202 }
203
204 type &operator*=(const type &rhs) {
205     p = multiply(p, rhs.p);
206     normalize();
207     return *this;
208 }
209
210 type &operator*=(const Field &rhs) {
211     int sz = size();
212     for (int i = 0; i < sz; i++)
213         p[i] *= rhs;
214     normalize();
215     return *this;
216 }
217
218 type &operator/=(const Field &rhs) {
219     int sz = size();
220     for (int i = 0; i < sz; i++)
221         p[i] /= rhs;
222     normalize();
223     return *this;
224 }
225
226 type &operator<=<(const int rhs) {
227     if (rhs < 0)
228         return *this >>= rhs;
229     if (rhs == 0)
230         return *this;
231     int sz = size();
232     p.resize(sz + rhs);
233     for (int i = sz - 1; i >= 0; i--)
234         p[i + rhs] = p[i];
235     fill_n(p.begin(), rhs, 0);

```

```

235     return *this;
236 }
237
238 type &operator>=(const int rhs) {
239     if (rhs < 0)
240         return *this <=<= rhs;
241     if (rhs == 0)
242         return *this;
243     int sz = size();
244     if (rhs >= sz) {
245         p.clear();
246         return *this;
247     }
248     for (int i = rhs; i < sz; i++)
249         p[i - rhs] = p[i];
250     p.resize(sz - rhs);
251     return *this;
252 }
253
254 type &operator%=(const int rhs) {
255     if (rhs < size())
256         p.resize(rhs);
257     normalize();
258     return *this;
259 }
260
261 type &operator/=(const type &rhs) { return *this = *this / rhs; }
262
263 type operator%=(const type &rhs) { return *this = *this % rhs; }
264
265 type operator+(const type &rhs) const {
266     type res = *this;
267     return res += rhs;
268 }
269
270 type operator-(const type &rhs) const {
271     type res = *this;
272     return res -= rhs;
273 }
274
275 type operator*(const type &rhs) const { return type(multiply(p, rhs.p)); }
276
277 type operator*(const Field &rhs) const {
278     type res = *this;
279     return res *= rhs;
280 }
281
282 type operator/(const Field &rhs) const {
283     type res = *this;
284     return res /= rhs;
285 }
286
287 type operator<<(const int rhs) const {
288     type res = *this;
289     return res <<= rhs;
290 }
291
292 type operator>>(const int rhs) const {
293     type res = *this;
294     return res >>= rhs;
295 }
296
297 type operator%(const int rhs) const {
298     return Polynomial(p.begin(), p.begin() + min(rhs, size()));
299 }

```

```

300 type operator/(const type &rhs) const {
301     return type::divmod(*this, rhs).first;
302 }
303
304 type operator%(const type &rhs) const {
305     return type::divmod(*this, rhs).second;
306 }
307
308 bool operator==(const type &rhs) const {
309     normalize();
310     rhs.normalize();
311     return p == rhs.p;
312 }
313
314 template <// Used in SFINAE.
315     typename U = Field,
316     enable_if_t<has_transform<Mult, U>::value>* = nullptr>
317 inline VectorN<U> transform(int n) {
318     return Mult().template transform<U>(n, p);
319 }
320
321 template <// Used in SFINAE.
322     typename U = Field,
323     enable_if_t<has_transform<Mult, U>::value>* = nullptr>
324 inline static type itransform(int n, const vector<U>& v) {
325     return Mult().template itransform<U>(n, v);
326 }
327
328 template <typename Functor,
329     // Used in SFINAE.
330     typename U = Field,
331     enable_if_t<has_transform<Mult, U>::value>* = nullptr,
332     typename ...Ts>
333 inline static type on_transform(
334     int n,
335     Functor f,
336     const Ts&... vs) {
337     if(n < Magic)
338         return f(n, vs...);
339     return Mult().template on_transform<U>(n, f, vs.p...);
340 }
341
342 template <typename Functor,
343     // Used in SFINAE.
344     typename U = Field,
345     enable_if_t<!has_transform<Mult, U>::value>* = nullptr,
346     typename ...Ts>
347 inline static type on_transform(
348     int n,
349     Functor f,
350     const Ts&... vs) {
351     return f(n, vs...);
352 }
353
354 template <
355     // Used in SFINAE.
356     typename U = Field,
357     enable_if_t<has_transform<Mult, U>::value>* = nullptr>
358 type inverse(int m) const {
359     if(null()) return *this;
360     type r = {Field(1) / p[0]};
361     r.p.reserve(m);
362     for(int i = 1; i < m; i *= 2) {
363         int n = 2 * i;

```

```

365     vector<U> f = (*this % n).p; f.resize(n);
366     vector<U> g = r.p; g.resize(n);
367     Transform::dft(f, n);
368     Transform::dft(g, n);
369     for(int j = 0; j < n; j++) f[j] *= g[j];
370     Transform::idft(f, n);
371     for(int j = 0; j < i; j++) f[j] = 0;
372     Transform::dft(f, n);
373     for(int j = 0; j < n; j++) f[j] *= g[j];
374     Transform::idft(f, n);
375     for(int j = i; j < min(n, m); j++)
376         r[j] = -f[j];
377     }
378     return r;
379 }
380
381 type inverse_slow(int m) const {
382     if(null()) return *this;
383     type b = {Field(1) / p[0]};
384     b.p.reserve(2 * m);
385     for(int i = 1; i < m; i *= 2) {
386         int n = min(2 * i, m);
387         auto bb = b * b % n;
388         b += b;
389         b -= *this % n * bb;
390         b %= n;
391     }
392     return b % m;
393 }
394
395 template <
396     // Used in SFINAE.
397     typename U = Field,
398     enable_if_t<!has_transform<Mult, U>::value>* = nullptr>
399 type inverse(int m) const {
400     return inverse_slow(m);
401 }
402
403 type inverse() const {
404     return inverse(size());
405 }
406
407 type reciprocal() const {
408     normalize();
409     return type(p.rbegin(), p.rend());
410 }
411
412 type integral() const {
413     int sz = size();
414     if (sz == 0)
415         return {};
416     type res = *this;
417     for (int i = sz; i; i--) {
418         res[i] = res[i - 1] / i;
419     }
420     res[0] = 0;
421     res.normalize();
422     return res;
423 }
424
425 type derivative() const {
426     int sz = size();
427     if (sz == 0)
428         return {};
429     type res = *this;

```

```

430     for (int i = 0; i + 1 < sz; i++) {
431         res[i] = res[i + 1] * (i + 1);
432     }
433     res.p.back() = 0;
434     res.normalize();
435     return res;
436 }
437
438 type mulx(field x) const { // component-wise multiplication with x^k
439     field cur = 1;
440     type res(*this);
441     for(auto& c : res.p)
442         c *= cur, cur *= x;
443     return res;
444 }
445
446 type mulx_sq(field x) const { // component-wise multiplication with x^{k^2}
447     field cur = x;
448     field total = 1;
449     field xx = x * x;
450     type res(*this);
451     for(auto& c : res.p)
452         c *= total, total *= cur, cur *= xx;
453     return res;
454 }
455
456 static pair<type, type> divmod(const type &a, const type &b) {
457     if (NaiveMod || min(a.size(), b.size()) < Magic)
458         return naive_divmod(a, b);
459     a.normalize();
460     b.normalize();
461     int m = a.size();
462     int n = b.size();
463     if (m < n)
464         return {Polynomial(), a};
465     int sz = m - n + 1;
466     type ar = a.reciprocal() % sz;
467     type br = b.reciprocal() % sz;
468     type q = (ar * br.inverse(sz) % sz).reciprocal();
469     type r = a - b * q;
470     return {q, r % (n-1)};
471 }
472
473 static pair<type, type> naive_divmod(const type &a, const type &b) {
474     type res = a;
475     int a_deg = a.degree();
476     int b_deg = b.degree();
477     Field normalizer = Field(1) / b[b_deg];
478     for (int i = 0; i < a_deg - b_deg + 1; i++) {
479         Field coef = (res[a_deg - i] * normalizer);
480         if (coef != 0) {
481             for (int j = 1; j <= b_deg; j++) {
482                 res[a_deg - i - j] += -b[b_deg - j] * coef;
483             }
484         }
485     }
486     return {res >> b_deg, res % b_deg};
487 }
488
489 vector<Field> czt_even(Field z, int n) const { // P(1), P(z^2), P(z^4),
490     ..., P(z^{2(n-1)})
491     int m = degree();
492     if(null()) {
493         return vector<Field>(n);
494     }
495     vector<Field> vv(m + n);
496     Field zi = Field(1) / z;

```

```

497     Field zz = zi * zi;
498     Field cur = zi;
499     Field total = 1;
500     for(int i = 0; i <= max(n - 1, m); i++) {
501         if(i <= m) {vv[m - i] = total;}
502         if(i < n) {vv[m + i] = total;}
503         total *= cur;
504         cur *= zz;
505     }
506     type w = (mulx_sq(z) * vv).substr(m, n).mulx_sq(z);
507     vector<Field> res(n);
508     for(int i = 0; i < n; i++) {
509         res[i] = w[i];
510     }
511     return res;
512 }
513
514 vector<Field> czt(Field z, int n) const {
515     auto even = czt_even(z, (n+1)/2);
516     auto odd = mulx(z).czt_even(z, n/2);
517     vector<Field> ans(n);
518     for(int i = 0; i < n/2; i++) {
519         ans[2*i] = even[i];
520         ans[2*i+1] = odd[i];
521     }
522     if(n&1) {
523         ans.back() = even.back();
524     }
525     return ans;
526 }
527
528 friend type kmul(const vector<type>& polys, int l, int r) {
529     if(l == r) return polys[l];
530     int mid = (l+r)/2;
531     return kmul(polys, l, mid) * kmul(polys, mid+1, r);
532 }
533
534 friend type kmul(const vector<type>& polys) {
535     if(polys.empty()) return type();
536     return kmul(polys, 0, (int)polys.size() - 1);
537 }
538
539 static type power(const type &a, long long n, const int mod) {
540     return math::generic_power<type>(a, n, DefaultPowerOp<type>(mod));
541 }
542
543 static type power(const type &a, long long n, const type &mod) {
544     return math::generic_power<type>(a, n, ModPowerOp<type>(mod));
545 }
546
547 static type kth(int K) { return type(1) << K; }
548
549 static type kth(long long K, const type &mod) {
550     return math::generic_power<type>(type(1) << 1, K,
551                                     ModShiftPowerOp<type>(mod));
552 }
553
554 friend ostream &operator<<(ostream &output, const type &var) {
555     output << "[";
556     int sz = var.size();
557     for (int i = sz - 1; i >= 0; i--) {
558         output << var[i];
559         if (i)
560             output << " ";
561     }
562     return output << "]";
563 }
564
565 };
566
567 } // namespace poly

```

```

559 /// keep caide
560 using poly::Polynomial;
561 } // namespace math
562 } // namespace lib
563
564 #endif

```

1.28. PowerSeries

```

1  #ifndef _LIB_POWER_SERIES
2  #define _LIB_POWER_SERIES
3  #include "BitTricks.cpp"
4  #include "PolynomialRing.cpp"
5  #include <bits/stdc++.h>
6
7  namespace lib {
8  using namespace std;
9  namespace series {
10
11  template <typename P> P ln(const P &p, int n);
12
13  template <typename P> P inverse(P p, int n) {
14      return p.inverse(n);
15  }
16
17  template <typename P> P ln(const P &p, int n) {
18      return (p.derivative() * inverse(p, n) % n).integral() % n;
19  }
20
21  /// \sum ln(1 + x^K), where K are elements of v.
22  template<typename P, typename I>
23  P ln_lpx(const vector<I>& v, int n) {
24      using Field = typename P::field;
25      vector<I> h(n);
26      vector<Field> res(n);
27      for(auto x : v) if(x < n) h[x]++;
28      res[0] = h[0];
29      for(int i = 1; i < n; i++) {
30          if(!h[i]) continue;
31          for(int j = 0, k = i; k < n; k += i, j++) {
32              Field c = Field(1) / Field(j + 1);
33              if(j&1) c = -c;
34              res[k] += c * h[i];
35          }
36      }
37      return P(res);
38  }
39
40  template<typename P> pair<P, P> exp2(P p, int n) {
41      assert(p[0] == 0);
42      P f{1}, g{1};
43      for(int i = 1; i <= n; i*=2) {
44          g = g * 2 - (g*g*i*f)%i;
45          P q = (p % i).derivative();
46          q += g * (f.derivative() - f * q) % (2 * i - 1);
47          f += f * (p % (2 * i) - q.integral()) % (2 * i);
48      }
49      return {f % n, g % n};
50  }
51
52  /// p[0] must be null
53  template <typename P> P exp(P p, int n) {
54      return exp2(p, n).first;
55  }

```

```

56 template <typename P> P power(const P &p, long long k, int n) {
57     int m = p.size();
58     for(int i = 0; i < m; i++) {
59         if(p[i] == 0) continue;
60         if(i > 0 && k > n / i) return {};
61         auto rev = typename P::field(1) / p[i];
62         auto D = (p * rev) >> i;
63         int sz = n - i * k;
64         D = exp(ln(D, sz) * k, sz) * (p[i] ^ k);
65         if(i == 0) return D % n;
66         long long S = k * i;
67         D <<= S;
68         return D % n;
69     }
70 }
71 return {};
72 }
73 } // namespace series
74 } // namespace lib
75
76 #endif

```

1.29. Random

```

1  #ifndef _LIB_RANDOM
2  #define _LIB_RANDOM
3  #include <bits/stdc++.h>
4
5  namespace lib {
6  using namespace std;
7  namespace rng {
8  struct Generator {
9      mt19937 rng;
10     Generator() {
11         seed_seq seq {
12             (uint64_t) chrono::duration_cast<chrono::nanoseconds>(
13                 chrono::high_resolution_clock::now().time_since_epoch())
14                 .count(),
15             #if __cplusplus > 201300
16                 (uint64_t)make_unique<char>().get(),
17             #else
18                 (uint64_t)unique_ptr<char>(new char).get(),
19             #endif
20                 (uint64_t)__builtin_ia32_rdtsc()
21             };
22         rng = mt19937(seq);
23     }
24     Generator(seed_seq &seq) : rng(seq) {}
25
26     template <typename T,
27             typename enable_if<is_integral<T>::value>::type * = nullptr>
28     inline T uniform_int(T L, T R) {
29         return uniform_int_distribution<T>(L, R)(rng);
30     }
31
32     template <typename T> inline T uniform_int(T N) {
33         return uniform_int(T(), N - 1);
34     }
35
36     template <typename T> inline T uniform_real(T N) {
37         return uniform_real(0.0, static_cast<double>(N));
38     }
39
40     template <typename T> inline T uniform_real(T L, T R) {

```

```

41     return uniform_real_distribution<double>(static_cast<double>(L),
42                                             static_cast<double>(R)) (rng);
43 }
44
45 inline double uniform_real() { return uniform_real(0.0, 1.0); }
46 };
47
48 static Generator gen = Generator();
49 } // namespace rng
50 static rng::Generator &rng_gen = rng::gen;
51 } // namespace lib
52
53 #endif

```

1.30. RangeDSU

```

1 #ifndef _LIB_RANGE_DSU
2 #define _LIB_RANGE_DSU
3 #include <bits/stdc++.h>
4 #include "SegtreeFast.cpp"
5 #include "DSU.cpp"
6
7 namespace lib {
8 using namespace std;
9 struct RangeDSU {
10     struct NodeImpl {
11         int low, high;
12         int low_inv, high_inv;
13         friend NodeImpl operator+(const NodeImpl& a, const NodeImpl& b) {
14             NodeImpl res = a;
15             if(b.low < res.low) res.low = b.low, res.low_inv = b.low_inv;
16             if(b.high > res.high) res.high = b.high, res.high_inv = b.high_inv;
17             return res;
18         }
19     };
20     using Node = seg::Active<NodeImpl>;
21
22     seg::SegtreeFast<Node, seg::CombineFolder<Node>> sg;
23     FastDSU dsu;
24     vector<vector<int>> inv;
25
26     RangeDSU(int n) : sg(seg::make_builder(n)), dsu(n), inv(n) {
27         // TODO: optimize
28         for(int i = 0; i < n; i++) {
29             sg.update_element(i, seg::SetUpdater<NodeImpl>(node_impl(i)));
30             inv[i].push_back(i);
31         }
32     }
33
34     NodeImpl node_impl(int i) {
35         int u = dsu[i];
36         return NodeImpl{u, u, i, i};
37     }
38
39     void activate(int i) {
40         sg.update_element(i, seg::ActiveUpdater<Node>(true));
41     }
42
43     void deactivate(int i) {
44         sg.update_element(i, seg::ActiveUpdater<Node>(false));
45     }
46
47     int operator[](int i) {
48         return dsu[i];

```

```

49     }
50
51     bool merge(int u, int v) {
52         if(!dsu.merge(u, v)) return false;
53         tie(u, v) = dsu.last_merge();
54         for(int x : inv[u]) {
55             inv[v].push_back(x);
56             sg.update_element(x, seg::SetUpdater<NodeImpl>(node_impl(x)));
57         }
58         return true;
59     }
60
61     int merge_range(int i, int j, int x) {
62         x = dsu[x];
63         Node res = sg.query<Node>(i, j, seg::CombineFolder<Node>());
64         if(!res.is_active()) return -1;
65         if(res.low != x) {
66             merge(res.low, x);
67             return res.low_inv;
68         }
69         if(res.high != x) {
70             merge(res.high, x);
71             return res.high_inv;
72         }
73         return -1;
74     }
75
76     void merge_all_range(int i, int j, int x) {
77         while(merge_range(i, j, x) != -1);
78     }
79
80     pair<int, int> last_merge() const { return dsu.last_merge(); }
81     vector<int> last_move() const { return inv[last_merge().first]; }
82 };
83 } // namespace lib
84
85 #endif

```

1.31. RollingHash

```

1 #ifndef _LIB_ROLLING_HASH
2 #define _LIB_ROLLING_HASH
3 #include "ModularInteger.cpp"
4 #include "Random.cpp"
5 #include "Traits.cpp"
6 #include <bits/stdc++.h>
7
8 namespace lib {
9 using namespace std;
10 namespace hashing {
11     namespace {
12         using traits::HasBidirectionalIterator;
13         using traits::HasInputIterator;
14         using traits::IsBidirectionalIterator;
15         using traits::IsInputIterator;
16         using traits::IsRandomIterator;
17     } // namespace
18
19     const static int DEFAULT_MAX_POWERS = 1e6 + 5;
20     const static int GOOD_MOD1 = (int)1e9 + 7;
21     const static int GOOD_MOD2 = (int)1e9 + 9;
22
23     template <typename T, T... Mods> struct BaseProvider {
24         typedef BaseProvider<T, Mods...> type;

```



```

25 typedef ModularInteger<T, Mods...> mint_type;
26
27 mint_type b;
28 vector<mint_type> powers;
29 int max_powers = 0;
30
31 BaseProvider(T bases[sizeof...(Mods)]) : powers(1, 1) {
32     b = mint_type::with_remainders(bases);
33 }
34 BaseProvider() : powers(1, 1) {
35     T bases[sizeof...(Mods)];
36     for (size_t i = 0; i < sizeof...(Mods); i++)
37         bases[i] = rng::gen.uniform_int(mint_type::mods[i]);
38     b = mint_type::with_remainders(bases);
39 }
40
41 void set_max_powers(int x) { max_powers = x; }
42
43 inline operator mint_type() const { return b; }
44 inline T operator()(int i) { return b[i]; }
45
46 void ensure(int p) const {
47     type *self = const_cast<type *>(this);
48     int cur = powers.size();
49     if (p > cur)
50         self->powers.resize(max(2 * cur, p));
51     else
52         return;
53     int nsz = powers.size();
54     for (int i = cur; i < nsz; i++)
55         self->powers[i] = powers[i - 1] * b;
56 }
57
58 mint_type power(int p) const {
59     if (p >= max_powers)
60         return b ^ p;
61     ensure(p + 1);
62     return powers[p];
63 }
64 T power(int p, int i) const { return power(p)[i]; }
65 };
66
67 template <typename T, T... Mods> struct RollingHash {
68     typedef RollingHash<T, Mods...> type;
69     typedef ModularInteger<T, Mods...> mint_type;
70     typedef BaseProvider<T, Mods...> base_type;
71
72     vector<mint_type> hs;
73
74     struct Hash {
75         mint_type x;
76         int n;
77
78         struct Less {
79             typename mint_type::less mint_less;
80             bool operator()(const Hash &lhs, const Hash &rhs) const {
81                 if (lhs.n == rhs.n)
82                     return mint_less(lhs.x, rhs.x);
83                 return lhs.n < rhs.n;
84             }
85         };
86         typedef Less less;
87
88         Hash() : n(0) {}
89         explicit Hash(mint_type y) : x(y), n(1) {}

```

```

90     Hash(mint_type y, int n) : x(y), n(n) { assert(n >= 0); }
91
92     explicit operator mint_type() const { return x; }
93
94     friend bool operator==(const Hash &lhs, const Hash &rhs) {
95         return tie(lhs.n, lhs.x) == tie(rhs.n, rhs.x);
96     }
97     friend bool operator!=(const Hash &lhs, const Hash &rhs) {
98         return !(lhs == rhs);
99     }
100     friend ostream &operator<<(ostream &output, const Hash &var) {
101         return output << var.x << "{" << var.n << "}";
102     }
103 };
104
105 struct Cat {
106     shared_ptr<base_type> base;
107     Cat(const shared_ptr<base_type> &base) : base(base) {}
108
109     template <
110         typename Iterator,
111         typename enable_if<IsInputIterator<Iterator>::value>::type * =
112             nullptr>
113     Hash operator()(Iterator begin, Iterator end) {
114         Hash res;
115         for (auto it = begin; it != end; ++it) {
116             res.n += it->n;
117             res.x *= base->power(it->n);
118             res.x += it->x;
119         }
120         return res;
121     }
122
123     Hash operator()(const initializer_list<Hash> &hashes) {
124         return (*this)(hashes.begin(), hashes.end());
125     }
126
127     template <class... Args> Hash operator()(Args... args) {
128         return (*this)({args...});
129     }
130
131     template <class... Args>
132     pair<Hash, Hash> cat(const pair<Args, Args> &... args) {
133         initializer_list<Hash> fwd_list = {args.first...};
134         initializer_list<Hash> bwd_list = {args.second...};
135         return {cat(fwd_list.begin(), fwd_list.end()),
136             cat(bwd_list.rbegin(), bwd_list.rend())};
137     }
138 };
139
140 Cat cat;
141
142 RollingHash(const shared_ptr<base_type> &base) : hs(1), cat(base) {}
143
144 template <
145     typename Container,
146     typename enable_if<HasInputIterator<Container>::value>::type * =
147         nullptr>
148 RollingHash(const Container &container, const shared_ptr<base_type> &base)
149     : hs(1), cat(base) {
150     (*this) += container;
151 }
152
153 template <
154     typename Iterator,

```

```

153     typename enable_if<IsInputIterator<Iterator>::value>::type * = nullptr>
154 RollingHash(Iterator begin, Iterator end, const shared_ptr<base_type>
    &base)
155     : hs(1), cat(base) {
156     append(begin, end);
157 }
158
159 inline int size() const { return (int)hs.size() - 1; }
160
161 template <
162     typename Iterator,
163     typename enable_if<IsRandomIterator<Iterator>::value>::type * =
    nullptr>
164 void append(Iterator begin, Iterator end) {
165     int i = hs.size();
166     hs.resize(hs.size() + distance(begin, end));
167     for (auto it = begin; it != end; ++it, ++i)
168         hs[i] = hs[i - 1] * (*cat.base) + mint_type(*it);
169 }
170
171 template <
172     typename Iterator,
173     typename enable_if<!IsRandomIterator<Iterator>::value>::type * =
    nullptr>
174 void append(Iterator begin, Iterator end) {
175     for (auto it = begin; it != end; ++it)
176         (*this) += *it;
177 }
178
179 template <typename U> void append(U rhs) { (*this) += rhs; }
180
181 template <typename U,
182     typename enable_if<is_integral<U>::value>::type * = nullptr>
183 RollingHash &operator+=(U rhs) {
184     hs.push_back(mint_type(rhs) + hs.back() * (*cat.base));
185     return *this;
186 }
187
188 template <
189     typename Container,
190     typename enable_if<HasInputIterator<Container>::value>::type * =
    nullptr>
191 RollingHash &operator+=(const Container &rhs) {
192     append(rhs.begin(), rhs.end());
193     return *this;
194 }
195
196 inline void pop() {
197     assert(size() > 0);
198     hs.pop_back();
199 }
200
201 Hash prefix(int n) const {
202     n = min(n, size());
203     return Hash(hs[n], n);
204 }
205
206 Hash operator()(int i, int j) const {
207     return Hash(hs[j + 1] - hs[i] * (cat.base->power(j - i + 1)), j - i + 1);
208 }
209
210 Hash suffix(int n) const {
211     int sz = size();
212     n = min(n, sz);
213     return (*this)(sz - n, sz - 1);

```

```

214 }
215
216 pair<Hash, Hash> border(int n) const { return {prefix(n), suffix(n)}; }
217
218 Hash substr(int i) const {
219     i = min(i, size());
220     return (*this)(i, size() - 1);
221 }
222
223 Hash substr(int i, int j) const { return (*this)(i, j); }
224
225 Hash all() const { return Hash(hs.back(), size()); }
226
227 friend int lcp(const type &lhs, const type &rhs) {
228     int l = 0, r = min(lhs.size(), rhs.size());
229     while (l < r) {
230         int mid = (l + r) / 2;
231         if (lhs.hs[mid + 1] != rhs.hs[mid + 1])
232             r = mid;
233         else
234             l = mid + 1;
235     }
236     return l;
237 }
238
239 friend bool operator<(const type &lhs, const type &rhs) {
240     int l = lcp(lhs, rhs);
241     if (l == min(lhs.size(), rhs.size()))
242         return lhs.size() < rhs.size();
243     return lhs(l, l) < rhs(l, l);
244 }
245 };
246
247 template <typename T, T... Mods> struct BidirectionalRollingHash {
248     typedef RollingHash<T, Mods...> type;
249     using Hash = typename type::Hash;
250     using base_type = typename type::base_type;
251     using Cat = typename type::Cat;
252
253     type fwd, bwd;
254     typename type::Cat cat;
255
256     template <typename Container,
257         typename
258         enable_if<HasBidirectionalIterator<Container>::value>::type
            * = nullptr>
259 BidirectionalRollingHash(const Container &container,
260         const shared_ptr<base_type> &base)
261         : BidirectionalRollingHash<T, Mods...>(container.begin(),
            container.end(),
262             base) {}
263
264     template <typename Iterator,
265         typename
266         enable_if<IsBidirectionalIterator<Iterator>::value>::type
            * = nullptr>
267 BidirectionalRollingHash(Iterator begin, Iterator end,
268         const shared_ptr<base_type> &base)
269         : fwd(begin, end, base),
270         bwd(make_reverse_iterator(end), make_reverse_iterator(begin), base),
271         cat(base) {}
272
273     inline Hash forward(int i, int j) const { return fwd(i, j); }
274
275     inline Hash backward(int i, int j) const {

```

```

276     int n = fwd.size();
277     return bwd(n - j - 1, n - i - 1);
278 }
279
280 inline pair<Hash, Hash> operator()(int i, int j) const {
281     return {forward(i, j), backward(i, j)};
282 }
283 };
284
285 template <typename R> struct HashProvider {
286     typedef R Roll;
287     typedef typename R::base_type base_type;
288     typedef typename R::Hash Hash;
289
290     typename R::Cat cat;
291     HashProvider() : cat(make_shared<base_type>()) {}
292     explicit HashProvider(base_type base) : cat(make_shared<base_type>(base))
293     {}
294
295     template <class... Args> R operator()(Args... args) {
296         return R(args..., cat.base);
297     }
298 };
299
300 template <typename R> struct PowerHashProvider : HashProvider<R> {
301     using typename HashProvider<R>::base_type;
302     using HashProvider<R>::cat;
303
304     PowerHashProvider() : PowerHashProvider<R>(base_type()) {}
305     PowerHashProvider(base_type base) : HashProvider<R>(base) {
306         cat.base->set_max_powers(DEFAULT_MAX_POWERS);
307     }
308 };
309
310 template <int32_t... Mods> using Roll32 = RollingHash<int32_t, Mods...>;
311
312 template <int64_t... Mods> using Roll64 = RollingHash<int64_t, Mods...>;
313
314 template <int32_t... Mods>
315 using Biroll32 = BidirectionalRollingHash<int32_t, Mods...>;
316
317 template <int64_t... Mods>
318 using Biroll64 = BidirectionalRollingHash<int64_t, Mods...>;
319
320 using DefaultProvider = PowerHashProvider<Roll32<GOOD_MOD1, GOOD_MOD2>>;
321 using BiDefaultProvider = PowerHashProvider<Biroll32<GOOD_MOD1, GOOD_MOD2>>;
322 } // namespace hashing
323 } // namespace lib
324 #endif

```

1.32. Segtree

```

1 #ifndef _LIB_SEGTREE
2 #define _LIB_SEGTREE
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace seg {
8 struct LeafBuilder {
9     template <typename Node> void operator()(Node &no, int i) const {}
10     inline pair<int, int> range() const { return {0, 0}; }
11     bool should_build() const { return true; }

```

```

12 };
13
14 struct EmptyLeafBuilder : LeafBuilder {
15     int n;
16     explicit EmptyLeafBuilder(int n) : n(n) {}
17     inline pair<int, int> range() const { return {0, n - 1}; }
18     bool should_build() const { return true; }
19 };
20
21 struct ImplicitBuilder : LeafBuilder {
22     int L, R;
23     explicit ImplicitBuilder(int L, int R) : L(L), R(R) {}
24     inline pair<int, int> range() const { return {L, R}; }
25     bool should_build() const { return false; }
26 };
27
28 // TODO: NOT IMPLEMENTED
29 template <typename DefaultNode>
30 struct ImplicitWithDefaultBuilder : LeafBuilder {
31     int L, R;
32     DefaultNode default_node;
33     explicit ImplicitWithDefaultBuilder(int L, int R, DefaultNode def)
34         : L(L), R(R), default_node(def) {}
35
36     template <typename Node> inline void operator()(Node &no, int i) const {
37         no = default_node;
38     }
39
40     inline pair<int, int> range() const { return {L, R}; }
41     bool should_build() const { return false; }
42 };
43
44 template <typename RandomIterator> struct RangeLeafBuilder : LeafBuilder {
45     RandomIterator begin, end;
46     explicit RangeLeafBuilder(RandomIterator begin, RandomIterator end)
47         : begin(begin), end(end) {}
48
49     template <typename Node> inline void operator()(Node &no, int i) const {
50         no = *(begin + i);
51     }
52
53     inline pair<int, int> range() const { return {0, end - begin - 1}; }
54 };
55
56 template <typename F> struct LambdaLeafBuilder : LeafBuilder {
57     F f;
58     pair<int, int> rng;
59     explicit LambdaLeafBuilder(F f, pair<int, int> range)
60         : f(f), rng(range) {}
61
62     template <typename Node> inline void operator()(Node &no, int i) const {
63         no = f(i);
64     }
65
66     inline pair<int, int> range() const { return rng; }
67 };
68
69 EmptyLeafBuilder make_builder(int n) { return EmptyLeafBuilder(n); }
70
71 template <typename RandomIterator>
72 RangeLeafBuilder<RandomIterator> make_builder(RandomIterator begin,
73                                             RandomIterator end) {
74     return RangeLeafBuilder<RandomIterator>(begin, end);
75 }
76

```

```

77 template <typename T>
78 RangeLeafBuilder<typename vector<T>::const_iterator>
79 make_builder(const vector<T> &v) {
80     return RangeLeafBuilder<typename vector<T>::const_iterator>(v.begin(),
81                                                                v.end());
82 }
83
84 template<typename T>
85 LambdaLeafBuilder<std::function<T(int)>>
86 make_builder(std::function<T(int)> f, pair<int, int> range) {
87     return LambdaLeafBuilder<std::function<T(int)>>(f, range);
88 }
89
90 template <typename T> struct CombineFolder {
91     inline T operator()() const { return T(); }
92
93     template <typename Node> inline T operator()(const Node &no) const {
94         return T(no);
95     }
96
97     inline T operator()(const T &a, const T &b) const { return a + b; }
98 };
99
100 template <typename T> struct EmptyFolder : CombineFolder<T> {
101     using CombineFolder<T>::operator();
102
103     template <typename Node> inline T operator()(const Node &no) const {
104         return T();
105     }
106     inline T operator()(const T &a, const T &b) const { return T(); }
107 };
108
109 template <typename T> struct SumFolder : CombineFolder<T> {};
110
111 template <typename T> struct ProductFolder : CombineFolder<T> {
112     using CombineFolder<T>::operator();
113     inline T operator()() const { return T(1); }
114     inline T operator()(const T &a, const T &b) const { return a * b; }
115 };
116
117 template <typename T> struct MaxFolder : CombineFolder<T> {
118     using CombineFolder<T>::operator();
119     inline T operator()() const { return numeric_limits<T>::min(); }
120     inline T operator()(const T &a, const T &b) const { return max(a, b); }
121 };
122
123 template <typename T> struct MinFolder : CombineFolder<T> {
124     using CombineFolder<T>::operator();
125     inline T operator()() const { return numeric_limits<T>::max(); }
126     inline T operator()(const T &a, const T &b) const { return min(a, b); }
127 };
128
129 template <typename T> struct SingleValueUpdater {
130     T value;
131     explicit SingleValueUpdater(T val) : value(val) {}
132 };
133
134 template <typename T> struct SetUpdater : SingleValueUpdater<T> {
135     using SingleValueUpdater<T>::SingleValueUpdater;
136
137     template <typename Node> inline void operator()(Node &no) const {
138         no = this->value;
139     }
140 };
141

```

```

142 template <typename T> struct AddUpdater : SingleValueUpdater<T> {
143     using SingleValueUpdater<T>::SingleValueUpdater;
144
145     template <typename Node> inline void operator()(Node &no) const {
146         no += this->value;
147     }
148 };
149
150 template <typename T> struct MultUpdater : SingleValueUpdater<T> {
151     using SingleValueUpdater<T>::SingleValueUpdater;
152
153     template <typename Node> inline void operator()(Node &no) const {
154         no *= this->value;
155     }
156 };
157
158 struct EmptyPushdown {
159     template<typename Node>
160     inline bool dirty(const Node& no) const { return false; }
161
162     template<typename Node>
163     inline void operator()(Node& no, int l, int r,
164                          Node* ln, Node* rn) const {}
165 };
166
167 template<typename Node>
168 struct Active : public Node {
169     bool active_ = false;
170     Active& operator=(const Node& no) {
171         Node::operator=(no);
172         return *this;
173     }
174     bool is_active() const { return active_; }
175     Active& activate() {
176         active_ = true;
177         return *this;
178     }
179     Active& deactivate() {
180         active_ = false;
181         return *this;
182     }
183     void toggle() {
184         active_ = !active_;
185     }
186     friend Active<Node> operator+(const Active<Node>& a, const Active<Node>&
187                                b) {
188         if(!a.active_) return b;
189         else if(!b.active_) return a;
190         Active<Node> res;
191         res = Node(a) + Node(b);
192         return res.activate();
193     }
194 };
195
196 template <typename T>
197 struct ActiveUpdater {
198     bool flag;
199
200     ActiveUpdater(bool f) : flag(f) {}
201
202     template <typename Node> inline void operator()(Node &no) const {
203         no.active_ = flag;
204     }
205 };
206 // namespace seg

```

```

206 } // namespace lib
207
208 #endif

```

1.33. SegtreeBeats

```

1  #ifndef _LIB_SEGTREE_BEATS
2  #define _LIB_SEGTREE_BEATS
3  #include "Segtree.cpp"
4  #include <bits/stdc++.h>
5
6  namespace lib {
7  using namespace std;
8  namespace seg {
9  struct DefaultBreakCond {
10     template <typename Node>
11     inline bool operator()(const Node &no, int l, int r, int i, int j) const {
12         return i > r || j < l;
13     }
14 };
15
16 struct DefaultTagCond {
17     template <typename Node>
18     inline bool operator()(const Node &no, int l, int r, int i, int j) const {
19         return i <= l && r <= j;
20     }
21 };
22
23 template <typename T> struct SearchResult {
24     bool found;
25     int pos;
26     T value;
27
28     static SearchResult<T> not_found(T acc = T()) { return {false, 0, acc}; }
29 };
30
31 struct PrefixSearch;
32 struct SuffixSearch;
33
34 template <typename Direction> using IsSuffix = is_same<Direction,
35     SuffixSearch>;
36
37 template <typename Node> struct InMemoryNodeManager {
38     typedef int vnode;
39     vector<Node> t;
40
41     template <typename Builder> void initialize(const Builder &builder) {
42         int L, R;
43         tie(L, R) = builder.range();
44         t = vector<Node>(4 * (R - L + 1));
45
46         inline bool has(vnode no) { return true; }
47         inline vnode root() { return 1; }
48         inline vnode new_root(vnode no) { return no; }
49         inline vnode left(vnode no) { return no << 1; }
50         inline vnode right(vnode no) { return no << 1 | 1; }
51         inline Node &ref(vnode no) { return t[no]; }
52         inline Node *ptr(vnode no) { return &t[no]; }
53         inline Node value(vnode no) { return t[no]; }
54
55         inline vnode persist(vnode no) { return no; }
56         inline void ensure_left(vnode no) {}
57         inline void ensure_right(vnode no) {}

```

```

58     };
59
60     template <
61         typename Node, typename NodeManager, typename CombinerFn =
62             EmptyFolder<int>,
63         typename PushdownFn = EmptyPushdown, typename BreakCond =
64             DefaultBreakCond,
65         typename TagCond = DefaultTagCond>
66     struct SegtreeImpl {
67         typedef typename NodeManager::vnode vnode;
68         constexpr static bool has_lazy = !is_same<PushdownFn,
69             EmptyPushdown>::value;
70         constexpr static bool is_implicit =
71             !is_same<NodeManager, InMemoryNodeManager<Node>>::value;
72
73         CombinerFn combiner_fn;
74         PushdownFn pushdown_fn;
75         BreakCond break_cond;
76         TagCond tag_cond;
77         NodeManager manager;
78
79         int L, R;
80
81         template <typename Builder> explicit SegtreeImpl(const Builder &builder) {
82             tie(L, R) = builder.range();
83             assert(L <= R);
84             manager.initialize(builder);
85             if (builder.should_build())
86                 build(builder);
87
88             inline vnode root() { return manager.root(); }
89             inline int split(int l, int r) { return l + (r - l) / 2; }
90
91             template <typename Builder>
92             vnode build(const Builder &builder, vnode no, int l, int r) {
93                 no = manager.persist(no);
94                 if (l == r) {
95                     builder(manager.ref(no), l);
96                 } else {
97                     int mid = split(l, r);
98                     build(builder, manager.left(no), l, mid);
99                     build(builder, manager.right(no), mid + 1, r);
100                     manager.ref(no) = combiner_fn(manager.value(manager.left(no)),
101                         manager.value(manager.right(no)));
102                 }
103                 return no;
104             }
105
106             template <typename Builder> vnode build(const Builder &builder) {
107                 return manager.new_root(build(builder, root(), L, R));
108             }
109
110             inline int size() const { return R - L + 1; }
111
112             void push(vnode no, int l, int r) {
113                 if (!has_lazy) return;
114                 if (!pushdown_fn.dirty(manager.ref(no)))
115                     return;
116                 if (l == r) {
117                     pushdown_fn(manager.ref(no), l, r, nullptr, nullptr);
118                     return;
119                 }
120                 manager.ensure_left(no);
121                 manager.ensure_right(no);

```

```

120     vnode lno = manager.persist(manager.left(no));
121     vnode rno = manager.persist(manager.right(no));
122     pushdown_fn(manager.ref(no), l, r, manager.ptr(lno), manager.ptr(rno));
123 }
124
125 template <typename T, typename Folder>
126 T query(vnode no, int l, int r, int i, int j, const Folder &folder) {
127     if (!manager.has(no))
128         return folder();
129     if (j < l || i > r)
130         return folder();
131     push(no, l, r);
132     if (i <= l && r <= j)
133         return folder(manager.ref(no));
134     int mid = split(l, r);
135     return folder(query<T>(manager.left(no), l, mid, i, j, folder),
136                    query<T>(manager.right(no), mid + 1, r, i, j, folder));
137 }
138
139 template <typename T, typename Folder>
140 inline T query(vnode root, int i, int j, const Folder &folder) {
141     return query<T>(root, L, R, i, j, folder);
142 }
143
144 template <typename T, typename Folder>
145 inline T query(int i, int j, const Folder &folder) {
146     return query<T>(root(), i, j, folder);
147 }
148
149 template <typename Updater>
150 vnode update(vnode no, int l, int r, int i, int j, const Updater &updater)
151 {
152     push(no, l, r);
153     if (break_cond(manager.ref(no), l, r, i, j)) {
154         return no;
155     }
156     no = manager.persist(no);
157     if (tag_cond(manager.ref(no), l, r, i, j)) {
158         updater(manager.ref(no));
159         push(no, l, r);
160         return no;
161     }
162     int mid = split(l, r);
163     update(manager.left(no), l, mid, i, j, updater);
164     update(manager.right(no), mid + 1, r, i, j, updater);
165     manager.ref(no) = combiner_fn(manager.value(manager.left(no)),
166                                   manager.value(manager.right(no)));
167     return no;
168 }
169
170 template <typename Updater>
171 inline vnode update(vnode root, int i, int j, const Updater &updater) {
172     return manager.new_root(update(root, L, R, i, j, updater));
173 }
174
175 template <typename Updater>
176 inline vnode update(int i, int j, const Updater &updater) {
177     return update(root(), i, j, updater);
178 }
179
180 template <typename Beater, typename U = NodeManager,
181          typename enable_if<
182             is_same<U, InMemoryNodeManager<Node>>::value::type * =
183             nullptr>
184 void beat(vnode no, int l, int r, int i, int j, const Beater &beater) {

```

```

183     push(no, l, r);
184     if (break_cond(manager.ref(no), l, r, i, j) ||
185         beater.stop(manager.ref(no), l, r, i, j)) {
186         return;
187     }
188     if (tag_cond(manager.ref(no), l, r, i, j) &&
189         beater.tag(manager.ref(no), l, r, i, j)) {
190         beater(manager.ref(no));
191         push(no, l, r);
192         return;
193     }
194     int mid = split(l, r);
195     beat(manager.left(no), l, mid, i, j, beater);
196     beat(manager.right(no), mid + 1, r, i, j, beater);
197     manager.ref(no) = combiner_fn(manager.value(manager.left(no)),
198                                   manager.value(manager.right(no)));
199 }
200
201 template <typename Beater>
202 inline void beat(int i, int j, const Beater &beater) {
203     beat(root(), L, R, i, j, beater);
204 }
205
206 template <typename T, typename Direction, typename Folder, typename
207     Checker>
208 SearchResult<T> bsearch_first(vnode no, int l, int r, int i, int j,
209                               const Folder &folder, const Checker &checker,
210                               T acc) {
211     if (manager.has(no))
212         push(no, l, r);
213     if (j < l || i > r)
214         return SearchResult<T>::not_found(folder());
215     if (!manager.has(no)) {
216         auto value = folder(acc, folder());
217         if (checker(value))
218             return {true, IsSuffix<Direction>::value ? r : l, value};
219         else
220             return SearchResult<T>::not_found(folder());
221     }
222     int mid = split(l, r);
223     if (i <= l && r <= j) {
224         auto b_value = folder(acc, manager.value(no));
225         if (!checker(b_value))
226             return SearchResult<T>::not_found(manager.value(no));
227         if (l == r)
228             return {true, l, b_value};
229     }
230     if (!IsSuffix<Direction>::value) {
231         auto res_left = bsearch_first<T, Direction>(manager.left(no), l, mid,
232                                                     i,
233                                                     j, folder, checker, acc);
234         if (res_left.found)
235             return res_left;
236         return bsearch_first<T, Direction>(manager.right(no), mid + 1, r, i, j,
237                                             folder, checker,
238                                             folder(acc, res_left.value));
239     } else {
240         auto res_right = bsearch_first<T, Direction>(
241             manager.right(no), mid + 1, r, i, j, folder, checker, acc);
242         if (res_right.found)
243             return res_right;
244         return bsearch_first<T, Direction>(manager.left(no), l, mid, i, j,
245                                             folder,
246                                             checker, folder(acc,
247 res_right.value));

```

```

244 }
245 }
246
247 template <typename T, typename Direction, typename Folder, typename
Checker>
248 inline SearchResult<T> bsearch_first(vnode root, int i, int j,
const Folder &folder,
249 const Checker &checker) {
250     auto res = bsearch_first<T, Direction>(root, L, R, i, j, folder, checker,
251     folder());
252     if (!res.found)
253         res.pos = IsSuffix<Direction>::value ? i - 1 : j + 1;
254     return res;
255 }
256
257 template <typename T, typename Direction, typename Folder, typename
Checker>
258 inline SearchResult<T> bsearch_first(int i, int j, const Folder &folder,
const Checker &checker) {
259     return bsearch_first<T, Direction>(root(), i, j, folder, checker);
260 }
261
262 template <typename T, typename Direction, typename Folder, typename
Checker>
263 inline SearchResult<T> bsearch_last(vnode root, int i, int j,
const Folder &folder,
264 const Checker &checker) {
265     auto res = bsearch_first<T, Direction>(
266     root, i, j, folder, [&checker](T x) { return !checker(x); });
267     if (!IsSuffix<Direction>::value) {
268         if (res.pos == i)
269             res.found = false;
270         res.pos--;
271     } else {
272         if (res.pos == j)
273             res.found = false;
274         res.pos++;
275     }
276     return res;
277 }
278
279 template <typename T, typename Direction, typename Folder, typename
Checker>
280 inline SearchResult<T> bsearch_last(int i, int j, const Folder &folder,
const Checker &checker) {
281     return bsearch_last<T, Direction>(root(), i, j, folder, checker);
282 }
283
284 template <typename Node, typename CombinerFn = EmptyFolder<int>,
typename PushdownFn = EmptyPushdown,
285 typename BreakCond = DefaultBreakCond,
286 typename TagCond = DefaultTagCond>
287 struct SegtreeBeats : SegtreeImpl<Node, InMemoryNodeManager<Node>,
CombinerFn,
288 PushdownFn, BreakCond, TagCond> {
289
290     template <typename Builder>
291     explicit SegtreeBeats(const Builder &builder)
292     : SegtreeImpl<Node, InMemoryNodeManager<Node>, CombinerFn, PushdownFn,
293     BreakCond, TagCond>(builder) {}
294 };
295
296 template <typename Node> using Explicit = InMemoryNodeManager<Node>;
297 } // namespace seg

```

```

304 } // namespace lib
305
306 #endif

```

1.34. SegtreeFast

```

1 #ifndef _LIB_SEGTREE_FAST
2 #define _LIB_SEGTREE_FAST
3 #include "Segtree.cpp"
4 #include <bits/stdc++.h>
5
6 namespace lib {
7     using namespace std;
8     namespace seg {
9         template <typename Node, typename CombinerFn> struct SegtreeFastBase {
10             const static int MULTIPLIER = 2;
11
12             CombinerFn combiner_fn;
13
14             vector<Node> t;
15             int L, n;
16
17             SegtreeFastBase() {}
18             template <typename Builder> explicit SegtreeFastBase(const Builder
&builder) {
19                 pair<int, int> range = builder.range();
20                 L = range.first;
21                 n = range.second - range.first + 1;
22                 assert(n > 0);
23                 t = vector<Node>(n * MULTIPLIER);
24                 build(builder);
25             }
26
27             template <typename Builder> void build(const Builder &builder) {
28                 for (int i = n; i < 2 * n; i++)
29                     builder(t[i], L + i - n);
30                 for (int i = n - 1; i > 0; i--)
31                     t[i] = combiner_fn(t[i << 1], t[i << 1 | 1]);
32             }
33
34             template <typename Rebuilder> void rebuild(const Rebuilder &rebuilder) {
35                 for (int i = n; i < 2 * n; i++)
36                     rebuilder(t[i]);
37                 for (int i = n - 1; i > 0; i--)
38                     rebuilder(t[i], t[i << 1], t[i << 1 | 1]);
39             }
40         };
41
42         template <typename Node, typename CombinerFn>
43         struct SegtreeFast : SegtreeFastBase<Node, CombinerFn> {
44             typedef SegtreeFastBase<Node, CombinerFn> Base;
45             using Base::combiner_fn;
46             using Base::L;
47             using Base::n;
48             using Base::SegtreeFastBase;
49             using Base::t;
50
51             template <typename Updater>
52             void update_element(int i, const Updater &updater) {
53                 i -= L;
54                 assert(i >= 0);
55                 for (updater(t[i += n]); i /= 2;)
56                     t[i] = combiner_fn(t[i << 1], t[i << 1 | 1]);
57             }

```

```

58
59 template <typename T, typename Folder>
60 T query(int i, int j, const Folder &folder) {
61     // input is [i, j]
62     i -= L, j -= L;
63     assert(i >= 0 && j >= 0);
64     i += n, j += n;
65     if (i == j)
66         return folder(t[i]);
67     T resl = folder(t[i]), resr = folder(t[j]);
68
69     // now it is [i, j]
70     i++;
71     while (i < j) {
72         if (i & 1)
73             resl = folder(resl, folder(t[i++]));
74         if (j & 1)
75             resr = folder(folder(t[--j]), resr);
76         i /= 2, j /= 2;
77     }
78     return folder(resl, resr);
79 }
80 };
81
82 template <typename Node>
83 struct SegtreeFastSplash : SegtreeFastBase<Node, EmptyFolder<Node>> {
84     typedef SegtreeFastBase<Node, EmptyFolder<Node>> Base;
85     using Base::L;
86     using Base::n;
87     using Base::SegtreeFastBase;
88     using Base::t;
89
90     template <typename T, typename Folder>
91     T query_element(int i, const Folder &folder) {
92         i -= L;
93         assert(i >= 0);
94         T res = folder(t[i += n]);
95         while (i != 2) {
96             res = folder(folder(t[i]), res);
97         }
98         return res;
99     }
100 }
101
102 template <typename Updater>
103 void splash(int i, int j, const Updater &updater) {
104     // input is [i, j]
105     i -= L, j -= L;
106     assert(i >= 0 && j >= 0);
107     // now it is [i, j]
108     i += n, j += n + 1;
109
110     while (i < j) {
111         if (i & 1)
112             updater(t[i++]);
113         if (j & 1)
114             updater(t[--j]);
115         i /= 2, j /= 2;
116     }
117 }
118 };
119
120 } // namespace seg
121 } // namespace lib
122

```

```

123 #endif

```

1.35. SegtreeHLD

```

1 #ifndef _LIB_RANGE_HLD
2 #define _LIB_RANGE_HLD
3 #include "HLD.cpp"
4 #include <bits/stdc++.h>
5
6 namespace lib {
7     using namespace std;
8     namespace graph {
9         namespace range {
10             template <typename Builder, typename H> struct BuilderWrapper {
11                 H *hld;
12                 Builder builder;
13
14                 explicit BuilderWrapper(H *hld, const Builder &builder)
15                     : hld(hld), builder(builder) {}
16
17                 template <typename Node> void operator()(Node &no, int i) const {
18                     builder(no, hld->rin[i]);
19                 }
20
21                 pair<int, int> range() const { return {0, hld->size() - 1}; }
22             };
23
24             template <typename Builder, typename H>
25             struct RebuilderWrapper : BuilderWrapper<Builder, H> {
26                 using BuilderWrapper<Builder, H>::BuilderWrapper;
27                 using BuilderWrapper<Builder, H>::builder;
28                 template <typename Node>
29                 void operator()(Node &no, const Node &left, const Node &right) const {
30                     builder(no, left, right);
31                 }
32             };
33
34             template <typename S, typename T, typename Folder> struct QueryIssuer {
35                 S &seg;
36                 const Folder &folder;
37                 QueryIssuer(S &seg, const Folder &folder) : seg(seg), folder(folder) {}
38                 T operator()(int i) const { seg.template query_element<T>(i, folder); }
39                 T operator()(int i, int j) const {
40                     return seg.template query<T>(i, j, folder);
41                 }
42             };
43
44             template <typename S, typename T, typename Folder> struct QueryLifter {
45                 QueryIssuer<S, T, Folder> issuer;
46                 T res;
47                 QueryLifter(S &seg, const Folder &folder)
48                     : issuer(seg, folder), res(folder()) {}
49                 void operator()(int i, int j, bool) {
50                     res = issuer.folder(res, issuer(i, j));
51                 }
52                 T result() const { return res; }
53             };
54
55             template <typename S, typename T, typename Folder>
56             struct OrderedQueryLifter : QueryLifter<S, T, Folder> {
57                 using QueryLifter<S, T, Folder>::issuer;
58                 T resl, resr;
59
60                 OrderedQueryLifter(S &seg, const Folder &folder)

```



```

61     : QueryLifter<S, T, Folder>(seg, folder), resl(folder()),
62     resr(folder()) {
63 }
64 void operator()(int i, int j, bool right) {
65     if (right)
66         resr = issuer.folder(issuer(i, j), resr);
67     else
68         resl = issuer.folder(resl, issuer(i, j));
69 }
70 T result() const { return issuer.folder(resl, resr); }
71 };
72
73 template <typename S, typename Updater> struct UpdateIssuer {
74     S &seg;
75     const Updater &updater;
76     UpdateIssuer(S &seg, const Updater &updater) : seg(seg), updater(updater)
77     {}
78     void operator()(int i, int j) { seg.update(i, j, updater); }
79     void operator()(int i, int j, bool) { (*this)(i, j); }
80 };
81
82 template <typename S, typename Updater> struct SplashIssuer {
83     S &seg;
84     const Updater &updater;
85     SplashIssuer(S &seg, const Updater &updater) : seg(seg), updater(updater)
86     {}
87     void operator()(int i, int j) { seg.splash(i, j, updater); }
88     void operator()(int i, int j, bool) { (*this)(i, j); }
89 };
90
91 template <typename S, typename Beater> struct BeatIssuer {
92     S &seg;
93     const Beater &beater;
94     BeatIssuer(S &seg, const Beater &beater) : seg(seg), beater(beater) {}
95     void operator()(int i, int j) { seg.beat(i, j, beater); }
96     void operator()(int i, int j, bool) { (*this)(i, j); }
97 };
98 // namespace range
99
100 template <typename S, typename G> struct RangeHLD : HLD<G> {
101     typedef seg::EmptyLeafBuilder empty_builder;
102
103     template <typename Builder>
104     using builder_wrapper = range::BuilderWrapper<Builder, HLD<G>>;
105     template <typename Rebuilder>
106     using rebuilder_wrapper = range::RebuilderWrapper<Rebuilder, HLD<G>>;
107
108     S seg;
109
110     explicit RangeHLD(const G &graph)
111         : HLD<G>(graph),
112         seg(builder_wrapper<empty_builder>(this,
113             empty_builder(this->size()))) {}
114
115     template <typename Builder>
116     RangeHLD(const G &graph, const Builder &builder)
117         : HLD<G>(graph), seg(builder_wrapper<Builder>(this, builder)) {}
118
119     template <typename Builder> void build(const Builder &builder) {
120         seg.build(builder_wrapper<Builder>(builder));
121     }
122
123     template <typename Rebuilder> void rebuild(const Rebuilder &rebuilder) {

```

```

122     seg.rebuild(rebuilder_wrapper<Rebuilder>(rebuilder));
123 }
124
125 template <typename T, typename Folder>
126 inline T query_subtree(int u, const Folder &folder) {
127     return this->template query_on_subtree<T>(
128         u, range::QueryIssuer<S, T, Folder>(seg, folder));
129 }
130
131 template <typename T, typename Folder>
132 inline T query_subtree_edges(int u, const Folder &folder) {
133     return this->template query_on_subtree_edges(
134         u, range::QueryIssuer<S, T, Folder>(seg, folder));
135 }
136
137 template <typename T, typename Folder>
138 inline T query_vertex(int u, const Folder &folder) {
139     return this->template query_on_vertex(
140         u, range::QueryIssuer<S, T, Folder>(seg, folder));
141 }
142
143 template <typename T, typename Folder>
144 T query_path(int u, int v, const Folder &folder) {
145     auto lifter = range::OrderedQueryLifter<S, T, Folder>(seg, folder);
146     this->template operate_on_path(u, v, lifter);
147     return lifter.result();
148 }
149
150 template <typename T, typename Folder>
151 T query_path_edges(int u, int v, const Folder &folder) {
152     auto lifter = range::OrderedQueryLifter<S, T, Folder>(seg, folder);
153     this->template operate_on_path_edges(u, v, lifter);
154     return lifter.result();
155 }
156
157 template <typename Updater>
158 inline void update_subtree(int u, const Updater &updater) {
159     auto issuer = range::UpdateIssuer<S, Updater>(seg, updater);
160     this->template operate_on_subtree(u, issuer);
161 }
162
163 template <typename Updater>
164 inline void update_subtree_edges(int u, const Updater &updater) {
165     auto issuer = range::UpdateIssuer<S, Updater>(seg, updater);
166     this->template operate_on_subtree_edges(u, issuer);
167 }
168
169 template <typename Updater>
170 inline void update_path(int u, int v, const Updater &updater) {
171     auto issuer = range::UpdateIssuer<S, Updater>(seg, updater);
172     this->template operate_on_path(u, v, issuer);
173 }
174
175 template <typename Updater>
176 inline void update_path_edges(int u, int v, const Updater &updater) {
177     auto issuer = range::UpdateIssuer<S, Updater>(seg, updater);
178     this->template operate_on_path_edges(u, v, issuer);
179 }
180
181 template <typename Beater>
182 inline void beat_subtree(int u, const Beater &beater) {
183     auto issuer = range::BeatIssuer<S, Beater>(seg, beater);
184     this->template operate_on_subtree(u, issuer);
185 }
186

```

```

187 template <typename Beater>
188 inline void beat_subtree_edges(int u, const Beater &beater) {
189     auto issuer = range::BeatIssuer<S, Beater>(seg, beater);
190     this->template operate_on_subtree_edges(u, issuer);
191 }
192
193 template <typename Beater>
194 inline void beat_path(int u, int v, const Beater &beater) {
195     auto issuer = range::BeatIssuer<S, Beater>(seg, beater);
196     this->template operate_on_path(u, v, issuer);
197 }
198
199 template <typename Beater>
200 inline void beat_path_edges(int u, int v, const Beater &beater) {
201     auto issuer = range::BeatIssuer<S, Beater>(seg, beater);
202     this->template operate_on_path_edges(u, v, issuer);
203 }
204
205 // TODO: FIX THOSE
206 template <typename Updater>
207 inline void update_element(int idx, const Updater &updater) {
208     seg.update_element(idx, updater);
209 }
210
211 template <typename Updater>
212 inline void splash(int i, int j, const Updater &updater) {
213     seg.splash(i, j, updater);
214 }
215 };
216
217 template <typename S, typename G>
218 RangeHLD<S, G> make_range_hld(const G &graph) {
219     return RangeHLD<S, G>(graph);
220 }
221
222 template <typename S, typename G, typename Builder>
223 RangeHLD<S, G> make_range_hld(const G &graph, const Builder &builder) {
224     return RangeHLD<S, G>(graph, builder);
225 }
226
227 } // namespace graph
228 } // namespace lib
229
230 #endif

```

1.36. SegtreeImplicit

```

1 #ifndef _LIB_SEGTREE_IMPLICIT
2 #define _LIB_SEGTREE_IMPLICIT
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace seg {
8
9 template <typename Node> struct ImplicitNodeManager {
10     struct NodeWrapper {
11         Node no;
12         NodeWrapper *left = nullptr;
13         NodeWrapper *right = nullptr;
14     };
15
16     struct VirtualNode {
17         NodeWrapper *cur = nullptr, **edge = nullptr;

```

```

18     };
19
20     typedef VirtualNode vnode;
21
22     vnode r = {new NodeWrapper()};
23
24     template <typename Builder> void initialize(const Builder &builder) {}
25
26     inline bool has(vnode no) const { return no.cur; }
27     inline vnode root() { return r; }
28     inline vnode new_root(vnode no) { return r = no; }
29     inline vnode left(vnode no) { return {no.cur->left, &(no.cur->left)}; }
30     inline vnode right(vnode no) { return {no.cur->right, &(no.cur->right)}; }
31     inline Node &ref(vnode no) { return no.cur->no; }
32     inline Node *ptr(vnode no) { return &(no.cur->no); }
33     inline Node value(vnode no) { return no.cur->no; }
34
35     inline vnode persist(vnode no) {
36         if (no.cur)
37             return no;
38         vnode res = no;
39         res.cur = *res.edge = new NodeWrapper();
40         return res;
41     }
42     inline void ensure_left(vnode no) {
43         if (!no.cur->left)
44             no.cur->left = new NodeWrapper();
45     }
46     inline void ensure_right(vnode no) {
47         if (!no.cur->right)
48             no.cur->right = new NodeWrapper();
49     }
50 };
51
52 template <typename Node> using Implicit = ImplicitNodeManager<Node>;
53 } // namespace seg
54 } // namespace lib
55
56 #endif

```

1.37. SegtreeLazy

```

1 #ifndef _LIB_SEGTREE_LAZY
2 #define _LIB_SEGTREE_LAZY
3 #include "SegtreeBeats.cpp"
4 #include <bits/stdc++.h>
5
6 namespace lib {
7 using namespace std;
8 namespace seg {
9     template <typename Node, typename CombinerFn, typename PushdownFn,
10             typename NodeManager = Explicit<Node>>
11     struct SegtreeLazy : SegtreeImpl<Node, NodeManager, CombinerFn, PushdownFn> {
12         typedef SegtreeImpl<Node, NodeManager, CombinerFn, PushdownFn> Base;
13         using Base::SegtreeImpl;
14         using typename Base::vnode;
15     };
16 } // namespace seg
17 } // namespace lib
18
19 #endif

```

1.38. SegtreeNormal

```

1 #ifndef _LIB_SEGTREE_NORMAL
2 #define _LIB_SEGTREE_NORMAL
3 #include "SegtreeBeats.cpp"
4 #include <bits/stdc++.h>
5
6 namespace lib {
7 using namespace std;
8 namespace seg {
9 template <typename Node, typename CombinerFn,
10         typename NodeManager = Explicit<Node>>
11 struct SegtreeNormal : SegtreeImpl<Node, NodeManager, CombinerFn> {
12     typedef SegtreeImpl<Node, NodeManager, CombinerFn> Base;
13     using Base::combiner_fn;
14     using Base::L;
15     using Base::manager;
16     using Base::R;
17     using Base::SegtreeImpl;
18     using Base::split;
19     using typename Base::vnode;
20
21     template <typename Updater>
22     vnode update_element(vnode no, int l, int r, int idx,
23                         const Updater &updater) {
24         no = manager.persist(no);
25         if (l == r)
26             updater(manager.ref(no));
27         else {
28             int mid = split(l, r);
29             if (idx <= mid)
30                 update_element(manager.left(no), l, mid, idx, updater);
31             else
32                 update_element(manager.right(no), mid + 1, r, idx, updater);
33             auto left_no = manager.left(no);
34             auto right_no = manager.right(no);
35             auto left_value =
36                 manager.has(left_no) ? manager.value(left_no) : combiner_fn();
37             auto right_value =
38                 manager.has(right_no) ? manager.value(right_no) : combiner_fn();
39             manager.ref(no) = combiner_fn(left_value, right_value);
40         }
41         return no;
42     }
43
44     template <typename Updater>
45     inline vnode update_element(vnode root, int idx, const Updater &updater) {
46         return manager.new_root(update_element(root, L, R, idx, updater));
47     }
48
49     template <typename Updater>
50     inline vnode update_element(int idx, const Updater &updater) {
51         return update_element(this->root(), idx, updater);
52     }
53 };
54 } // namespace seg
55 } // namespace lib
56
57 #endif

```

1.39. SegtreePersistent

```

1 #ifndef _LIB_SEGTREE_PERSISTENT
2 #define _LIB_SEGTREE_PERSISTENT
3 #include "SegtreeImplicit.cpp"

```

```

4 #include <bits/stdc++.h>
5
6 namespace lib {
7 using namespace std;
8 namespace seg {
9
10 template <typename Node>
11 struct PersistentNodeManager : ImplicitNodeManager<Node> {
12     using typename ImplicitNodeManager<Node>::vnode;
13     using typename ImplicitNodeManager<Node>::NodeWrapper;
14
15     inline vnode persist(vnode no) {
16         vnode res = no;
17         res.cur = no.cur ? new NodeWrapper(*no.cur) : new NodeWrapper();
18         if (res.edge)
19             *res.edge = res.cur;
20         return res;
21     }
22 };
23
24 template <typename Node> using Persistent = PersistentNodeManager<Node>;
25 } // namespace seg
26 } // namespace lib
27
28 #endif

```

1.40. SegtreeSplash

```

1 #ifndef _LIB_SEGTREE_SPLASH
2 #define _LIB_SEGTREE_SPLASH
3 #include "SegtreeBeats.cpp"
4 #include <bits/stdc++.h>
5
6 namespace lib {
7 using namespace std;
8 namespace seg {
9 template <typename Node, typename NodeManager = Explicit<Node>>
10 struct SegtreeSplash : SegtreeBeats<Node, NodeManager, EmptyFolder<void>> {
11     typedef SegtreeBeats<Node, NodeManager, EmptyFolder<void>> Base;
12     using Base::L;
13     using Base::manager;
14     using Base::R;
15     using Base::SegtreeBeats;
16     using Base::split;
17     using typename Base::vnode;
18
19     template <typename T, typename Folder>
20     T query_element(vnode no, int l, int r, int idx, const Folder &folder) {
21         if (!manager.has(no))
22             return folder();
23         T res = folder(manager.ref(no));
24         if (l != r) {
25             int mid = split(l, r);
26             if (idx <= mid)
27                 res = folder(res,
28                             query_element<T>(manager.left(no), l, mid, idx,
29                                             folder));
30             else
31                 res = folder(
32                     res, query_element<T>(manager.right(no), mid + 1, r, idx,
33                                         folder));
34         }
35         return res;
36     }
37 }
38 }
39 }

```

```

35
36 template <typename T, typename Folder>
37 inline T query_element(vnode root, int idx, const Folder &folder) {
38     return query_element<T>(root, L, R, idx, folder);
39 }
40
41 template <typename T, typename Folder>
42 inline T query_element(int idx, const Folder &folder) {
43     return query_element<T>(this->root(), idx, folder);
44 }
45
46 template <typename Updater>
47 vnode splash(vnode no, int l, int r, int i, int j, const Updater &updater)
48 {
49     no = manager.persist(no);
50     if (tag_cond(manager.ref(no), l, r, i, j)) {
51         updater(manager.ref(no));
52         return no;
53     }
54     int mid = split(l, r);
55     if (j <= mid) {
56         manager.ensure_left(no);
57         splash(manager.left(no), l, mid, i, j, updater);
58     } else if (i > mid) {
59         manager.ensure_right(no);
60         splash(manager.right(no), mid + 1, r, i, j, updater);
61     } else {
62         manager.ensure_left(no), manager.ensure_right(no);
63         splash(manager.left(no), l, mid, i, j, updater);
64         splash(manager.right(no), mid + 1, r, i, j, updater);
65     }
66     return no;
67 }
68
69 template <typename Updater>
70 inline vnode splash(vnode root, int i, int j, const Updater &updater) {
71     return manager.new_root(splash(root, L, R, i, j, updater));
72 }
73
74 template <typename Updater>
75 inline vnode splash(int i, int j, const Updater &updater) {
76     return splash(this->root(), i, j, updater);
77 }
78 };
79 // namespace seg
80 // namespace lib
81 #endif

```

1.41. Simplex

```

1 #ifndef _LIB_SIMPLEX
2 #define _LIB_SIMPLEX
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 template <typename DOUBLE> struct LPSolver {
8     typedef vector<DOUBLE> VD;
9     typedef vector<VD> VVD;
10    typedef vector<int> VI;
11
12    constexpr static DOUBLE EPS = 1e-9;
13

```

```

14    int m, n;
15    VI B, N;
16    VVD D;
17
18    LPSolver(const VVD &A, const VD &b, const VD &c)
19        : m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
20        for (int i = 0; i < m; i++)
21            for (int j = 0; j < n; j++)
22                D[i][j] = A[i][j];
23        for (int i = 0; i < m; i++) {
24            B[i] = n + i;
25            D[i][n] = -1;
26            D[i][n + 1] = b[i];
27        }
28        for (int j = 0; j < n; j++) {
29            N[j] = j;
30            D[m][j] = -c[j];
31        }
32        N[n] = -1;
33        D[m + 1][n] = 1;
34    }
35
36    void Pivot(int r, int s) {
37        for (int i = 0; i < m + 2; i++)
38            if (i != r)
39                for (int j = 0; j < n + 2; j++)
40                    if (j != s)
41                        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
42        for (int j = 0; j < n + 2; j++)
43            if (j != s)
44                D[r][j] /= D[r][s];
45        for (int i = 0; i < m + 2; i++)
46            if (i != r)
47                D[i][s] /= -D[r][s];
48        D[r][s] = 1.0 / D[r][s];
49        swap(B[r], N[s]);
50    }
51
52    bool Simplex(int phase) {
53        int x = phase == 1 ? m + 1 : m;
54        while (true) {
55            int s = -1;
56            for (int j = 0; j <= n; j++) {
57                if (phase == 2 && N[j] == -1)
58                    continue;
59                if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] <
N[s])
60                    s = j;
61            }
62            if (D[x][s] > -EPS)
63                return true;
64            int r = -1;
65            for (int i = 0; i < m; i++) {
66                if (D[i][s] < EPS)
67                    continue;
68                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])
69                    r = i;
70            }
71            if (r == -1)
72                return false;
73            Pivot(r, s);
74        }
75    }
76 }

```

```

77 DOUBLE Solve(VD &x) {
78     int r = 0;
79     for (int i = 1; i < m; i++)
80         if (D[i][n + 1] < D[r][n + 1])
81             r = i;
82     if (D[r][n + 1] < -EPS) {
83         Pivot(r, n);
84         if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
85             return numeric_limits<DOUBLE>::infinity();
86         for (int i = 0; i < m; i++)
87             if (B[i] == -1) {
88                 int s = -1;
89                 for (int j = 0; j <= n; j++)
90                     if (s == -1 || D[i][j] < D[i][s] ||
91                         D[i][j] == D[i][s] && N[j] < N[s])
92                         s = j;
93                 Pivot(i, s);
94             }
95     }
96     if (!Simplex(2))
97         return numeric_limits<DOUBLE>::infinity();
98     x = VD(n);
99     for (int i = 0; i < m; i++)
100         if (B[i] < n)
101             x[B[i]] = D[i][n + 1];
102     return D[m][n + 1];
103 }
104 };
105 } // namespace lib
106
107 #endif
108

```

1.42. Subset

```

1 #ifndef _LIB_SUBSET
2 #define _LIB_SUBSET
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 // Source: https://github.com/NyaanNyaan/library/tree/master/set-function
8
9 template <typename T>
10 void superset_zeta_transform(vector<T>& f) {
11     int n = f.size();
12     assert((n & (n - 1)) == 0);
13     for (int i = 1; i < n; i <= 1) {
14         for (int j = 0; j < n; j++) {
15             if ((j & i) == 0) {
16                 f[j] += f[j | i];
17             }
18         }
19     }
20 }
21
22 template <typename T>
23 void superset_mobius_transform(vector<T>& f) {
24     int n = f.size();
25     assert((n & (n - 1)) == 0);
26     for (int i = 1; i < n; i <= 1) {
27         for (int j = 0; j < n; j++) {
28             if ((j & i) == 0) {
29                 f[j] -= f[j | i];

```

```

30     }
31 }
32 }
33 }
34
35 template <typename T>
36 void subset_zeta_transform(vector<T>& f) {
37     int n = f.size();
38     assert((n & (n - 1)) == 0);
39     for (int i = 1; i < n; i <= 1) {
40         for (int j = 0; j < n; j++) {
41             if ((j & i) == 0) {
42                 f[j | i] += f[j];
43             }
44         }
45     }
46 }
47
48 template <typename T>
49 void subset_mobius_transform(vector<T>& f) {
50     int n = f.size();
51     assert((n & (n - 1)) == 0);
52     for (int i = 1; i < n; i <= 1) {
53         for (int j = 0; j < n; j++) {
54             if ((j & i) == 0) {
55                 f[j | i] -= f[j];
56             }
57         }
58     }
59 }
60
61 template <typename T>
62 vector<T> or_convolution(vector<T> a, vector<T> b) {
63     assert(a.size() == b.size());
64     subset_zeta_transform(a);
65     subset_zeta_transform(b);
66     for (int i = 0; i < (int)a.size(); i++) a[i] *= b[i];
67     subset_mobius_transform(a);
68     return a;
69 }
70
71 template <typename T>
72 vector<T> and_convolution(vector<T> a, vector<T> b) {
73     assert(a.size() == b.size());
74     superset_zeta_transform(a);
75     superset_zeta_transform(b);
76     for (int i = 0; i < (int)a.size(); i++) a[i] *= b[i];
77     superset_mobius_transform(a);
78     return a;
79 }
80
81 template <typename T>
82 vector<vector<T>> ranked_zeta_transform(const vector<T>& f) {
83     int N = f.size();
84     assert((N & (N - 1)) == 0);
85     int R = __builtin_ctz(N);
86     vector<vector<T>> F(R + 1, vector<T>(N));
87     for (int i = 0; i < N; i++)
88         F[__builtin_popcount(i)][i] = f[i];
89     for (int i = 0; i <= R; i++)
90         subset_zeta_transform(F[i]);
91     return F;
92 }
93
94 template <typename T>

```

```

95 vector<T> subset_convolution(const vector<T>& a, const vector<T>& b, int
    offset = 0) {
96     int N = a.size();
97     assert(N == b.size());
98     assert((N & (N-1)) == 0);
99     int R = __builtin_ctz(N);
100
101     auto A = ranked_zeta_transform(a), B = ranked_zeta_transform(b);
102     auto C = vector<vector<T>>(R + 1, vector<T>(N));
103
104     for(int m = 0; m < N; m++) {
105         for(int i = 0; i <= R; i++) {
106             for(int j = offset; j <= i; j++) {
107                 C[i][m] += A[j][m] * B[i + offset - j][m];
108             }
109         }
110     }
111
112     for(int i = 0; i <= R; i++)
113         subset_mobius_transform(C[i]);
114     vector<T> res(N);
115     for(int i = 0; i < N; i++)
116         res[i] = C[__builtin_popcount(i)][i];
117     return res;
118 }
119 } // namespace lib
120
121 #endif

```

1.43. SuffixArray

```

1  #ifndef _LIB_SUFFIX_ARRAY
2  #define _LIB_SUFFIX_ARRAY
3  #include <bits/stdc++.h>
4
5  namespace lib {
6  using namespace std;
7  template<typename C>
8  struct SuffixArray {
9      int n, block;
10     vector<C> s;
11     vector<int> sa, rnk, tmp, aux, lcp_;
12     vector<vector<int>> T;
13
14     void init(int h) {
15         h = max(h, n);
16         sa = vector<int>(h+3), rnk = vector<int>(h+3),
17         tmp = vector<int>(h+3), aux = vector<int>(h+3),
18         lcp_ = vector<int>(h+3);
19         T = vector<vector<int>>(n + 3, vector<int>(__lg(n) + 1));
20     }
21
22     SuffixArray(vector<C> s_) : s(s_), n(s_.size()) { build(); }
23     SuffixArray(string s_) {
24         s = vector<C>(s_.size());
25         n = s_.size();
26         for(int i = 0; i < n; i++) s[i] = s_[i];
27         build();
28     }
29
30     bool suffix_cmp(int i, int j) {
31         if (rnk[i] != rnk[j]) return rnk[i] < rnk[j];
32         i += block, j += block;
33         if (i >= n) i -= n;

```

```

34         if (j >= n) j -= n;
35         return rnk[i] < rnk[j];
36     }
37     void suffix_sort(int h) {
38         for (int i = 0; i < n; i++) {
39             aux[i] = sa[i] - block;
40             if (aux[i] < 0) aux[i] += n;
41         }
42         for (int i = 0; i < h; i++) tmp[i] = 0;
43         for (int i = 0; i < n; i++) tmp[rnk[aux[i]]]++;
44         for (int i = 0; i < h - 1; i++) tmp[i + 1] += tmp[i];
45         for (int i = n - 1; i >= 0; i--) sa[--tmp[rnk[aux[i]]]] = aux[i];
46         tmp[0] = 0;
47         for (int i = 0; i < n - 1; i++) tmp[i + 1] = tmp[i] + suffix_cmp(sa[i],
            sa[i + 1]);
48         for (int i = 0; i < n; i++) rnk[sa[i]] = tmp[i];
49     }
50     void build() {
51         n++; // consider additional '\0' character
52         s.push_back(0);
53         int h = (int)(*max_element(s.begin(), s.end())) + 1;
54         init(h);
55         for (int i = 0; i < n; i++) sa[i] = i, rnk[i] = s[i], tmp[i] = 0;
56         block = 0;
57         suffix_sort(h);
58         for (block = 1; tmp[n - 1] != n - 1; block *= 2) suffix_sort(tmp[n - 1]
            + 1);
59         n--;
60         sa.erase(sa.begin());
61         build_lcp_();
62     }
63
64     void build_lcp_() {
65         for (int i = 0; i < n; i++) rnk[sa[i]] = i, lcp_[i] = 0;
66         int last = 0; // last lcp_
67         for (int i = 0; i < n; i++, last = max(lcp_[rnk[i] - 1] - 1, 0)) {
68             if (rnk[i] == n - 1) continue;
69             int j = sa[rnk[i] + 1]; // next suffix pos in suffix array
70             while (i + last < n && j + last < n && s[i + last] == s[j + last])
71                 last++;
72             lcp_[rnk[i]] = last;
73         }
74
75         for(int i = 0; i < n; i++)
76             T[i][0] = lcp_[i];
77         for(int j = 1; j < 20; j++){
78             for(int i = 0; i+(1<<j) <= n; i++){
79                 T[i][j] = min(T[i][j-1], T[i+(1<<(j-1))][j-1]);
80             }
81         }
82
83         int lcp(int i, int j){
84             if(i > j) swap(i, j);
85             if(i == j) return n-sa[i];
86             if(j == n) return 0;
87
88             j--;
89             int k = __builtin_clz(1) - __builtin_clz(j-i+1);
90             return min(T[i][k], T[j-(1<<k)+1][k]);
91         }
92
93         int operator[](int i) const {
94             return sa[i];
95         }

```

```

96
97 int length(int i) const {
98     return n - sa[i];
99 }
100
101 int lcp(int i) const {
102     return lcp_[i];
103 }
104
105 pair<int,int> range(int i, int sz){
106     pair<int, int> res;
107     {
108         int l = 0, r = i+1;
109         while(l < r){
110             int mid = (l+r)/2;
111             if(lcp(mid, i) >= sz) r = mid;
112             else l = mid+1;
113         }
114         res.first = l;
115     }
116     {
117         int l = i, r = n-1;
118         while(l < r){
119             int mid = (l+r+1)/2;
120             if(lcp(mid, i) >= sz) l = mid;
121             else r = mid-1;
122         }
123         res.second = l;
124     }
125     return res;
126 }
127
128 pair<int, int> range(int i) {
129     return range(i, length(i));
130 }
131 };
132 } // namespace lib
133
134 #endif

```

1.44. Symbolic

```

1 #ifndef _LIB_SYMBOLIC
2 #define _LIB_SYMBOLIC
3 #include <bits/stdc++.h>
4
5 namespace lib {
6     using namespace std;
7     static int g_VAR_PTR = 0;
8
9     enum Operation { variable, literal, sum };
10
11     template <typename T> struct Variable;
12
13     template <typename T> struct BasicExp {
14         using node = shared_ptr<BasicExp<T>>;
15         using variable = Variable<T>;
16
17         T coef = 1;
18         Operation op;
19         vector<node> children;
20         variable var;
21
22         BasicExp(Operation n_op, const vector<node> &n_children, T n_coef = 1);

```

```

23         BasicExp(const T &v);
24
25         BasicExp(const Variable<T> &v);
26
27         bool has_children() const {
28             return op != Operation::variable && op != Operation::literal;
29         }
30
31         Variable<T> get_variable() const { return var; }
32     };
33
34     template <typename T> using Expression = shared_ptr<BasicExp<T>>;
35
36     template <typename T, typename... Args>
37     Expression<T> make_exp(Args &&... args) {
38         return make_shared<BasicExp<T>, Args...>(std::forward<Args>(args)...);
39     }
40
41     template <typename T> struct Variable {
42         int id;
43
44         static Variable<T> get_variable() { return {g_VAR_PTR++}; }
45
46         static vector<Variable<T>> get_variables(int n) {
47             vector<Variable<T>> vars(n);
48             for (int i = 0; i < n; i++)
49                 vars[i] = get_variable();
50             return vars;
51         }
52
53         static Expression<T> get_exp_variable() {
54             return Variable<T>::get_variable().as_exp();
55         }
56
57         static vector<Expression<T>> get_exp_variables(int n) {
58             vector<Expression<T>> vs(n);
59             int i = 0;
60             for (const auto &v : Variable<T>::get_variables(n)) {
61                 vs[i++] = v.as_exp();
62             }
63             return vs;
64         }
65
66         operator Expression<T>() const { return make_exp<T>(*this); }
67
68         Expression<T> as_exp() const { return Expression<T>(*this); }
69
70         bool operator<(const Variable<T> &rhs) const { return id < rhs.id; }
71     };
72
73     template <typename T>
74     BasicExp<T>::BasicExp(Operation n_op, const vector<node> &n_children, T
75         n_coef)
76         : op(n_op), children(n_children), coef(n_coef) {}
77
78     template <typename T> BasicExp<T>::BasicExp(const T &v) {
79         op = Operation::literal;
80         coef = v;
81     }
82
83     template <typename T> BasicExp<T>::BasicExp(const Variable<T> &v) {
84         op = Operation::variable;
85         var = v;
86     }

```

```

87 template <typename T> Expression<T> &operator+=(Expression<T> &e, const T
    &x) {
88     e->coef *= x;
89     return e;
90 }
91
92 template <typename T>
93 Expression<T> operator*(const Expression<T> &e, const T &x) {
94     auto res = make_exp<T>(*e);
95     return res *= x;
96 }
97
98 template <typename T>
99 Expression<T> &operator+=(Expression<T> &e, const Expression<T> &rhs) {
100     if (e->op == Operation::sum) {
101         e->children.push_back(rhs);
102     } else {
103         e = make_exp<T>(Operation::sum, vector<Expression<T>>{e, rhs});
104     }
105     return e;
106 }
107
108 template <typename T>
109 Expression<T> &operator+=(Expression<T> &e, const Variable<T> &rhs) {
110     return e += make_exp<T>(rhs);
111 }
112
113 template <typename T> Expression<T> &operator+=(Expression<T> &e, const T
    &x) {
114     return e += make_exp<T>(x);
115 }
116
117 template <typename T>
118 Expression<T> operator+(const Expression<T> &e, const Expression<T> &rhs) {
119     auto res = e->op == Operation::sum ? make_exp<T>(*e) : e;
120     return res += rhs;
121 }
122
123 template <typename T>
124 Expression<T> operator+(const Expression<T> &e, const Variable<T> &rhs) {
125     return e + make_exp<T>(rhs);
126 }
127
128 template <typename T>
129 Expression<T> operator+(const Expression<T> &e, const T &x) {
130     return e + make_exp<T>(x);
131 }
132
133 template <typename T>
134 Expression<T> operator+(const Variable<T> &v, const Expression<T> &rhs) {
135     return make_exp<T>(v) + rhs;
136 }
137
138 template <typename T>
139 Expression<T> operator+(const Variable<T> &v, const Variable<T> &rhs) {
140     return make_exp<T>(v) + make_exp<T>(rhs);
141 }
142
143 template <typename T>
144 Expression<T> operator+(const Variable<T> &v, const T &x) {
145     return make_exp<T>(v) + make_exp<T>(x);
146 }
147
148 template <typename T>
149 Expression<T> operator*(const Variable<T> &v, const T &x) {

```

```

150     return make_exp<T>(v) * x;
151 }
152
153 template <typename T> struct ExpressionVisitor {
154     void visit(const Expression<T> &e) {
155         if (e->op == Operation::sum)
156             this->visit_sum(e);
157         else if (e->op == Operation::variable)
158             this->visit_variable(e);
159         else if (e->op == Operation::literal)
160             this->visit_literal(e);
161     }
162     virtual void visit_children(const Expression<T> &e) {
163         if (e->has_children()) {
164             for (const Expression<T> &child : e->children)
165                 this->visit(child);
166         }
167     }
168
169     virtual void visit_sum(const Expression<T> &e) { this->visit_children(e); }
170     virtual void visit_variable(const Expression<T> &e) {}
171     virtual void visit_literal(const Expression<T> &e) {}
172 };
173
174 template <typename T> struct VariableVisitor : ExpressionVisitor<T> {
175     set<Variable<T>> seen;
176     virtual void visit_variable(const Expression<T> &e) { seen.insert(e->var); }
177 };
178
179 template <typename T, typename S = T>
180 struct StackVisitor : ExpressionVisitor<T> {
181     vector<S> sta;
182     virtual void visit_children(const Expression<T> &e) override {
183         sta.push_back(sta.empty() ? e->coef : sta.back() * e->coef);
184         ExpressionVisitor<T>::visit_children(e);
185         if (!sta.empty())
186             sta.pop_back();
187     }
188     S top() const { return sta.empty() ? S(1) : sta.back(); }
189 };
190
191 template <typename T> struct EvalVisitor : StackVisitor<T> {
192     map<Variable<T>, T> values;
193     T result;
194     T eval(const Expression<T> &e, const map<Variable<T>, T> &values) {
195         result = T();
196         this->values = values;
197         this->visit(e);
198         return result;
199     }
200     virtual void visit_variable(const Expression<T> &e) override {
201         result += this->top() * e->coef * values[e->var];
202     }
203     virtual void visit_literal(const Expression<T> &e) override {
204         result += this->top() * e->coef;
205     }
206 };
207
208 enum ConstraintOperation {
209     equals,
210     different,
211     greater,
212     less,
213     greater_eq,

```



```

214 less_eq
215 };
216
217 template <typename T> struct Constraint {
218     Expression<T> lhs, rhs;
219     ConstraintOperation op;
220     Constraint(const Expression<T> &a, const Expression<T> &b,
221               ConstraintOperation op)
222         : lhs(a), rhs(b), op(op) {}
223 };
224
225 template <typename T>
226 Constraint<T> operator==(const Expression<T> &a, const Expression<T> &b) {
227     return Constraint<T>(a, b, ConstraintOperation::equals);
228 }
229
230 template <typename T>
231 Constraint<T> operator!=(const Expression<T> &a, const Expression<T> &b) {
232     return Constraint<T>(a, b, ConstraintOperation::different);
233 }
234
235 template <typename T>
236 Constraint<T> operator>=(const Expression<T> &a, const Expression<T> &b) {
237     return Constraint<T>(a, b, ConstraintOperation::greater_eq);
238 }
239
240 template <typename T>
241 Constraint<T> operator<=(const Expression<T> &a, const Expression<T> &b) {
242     return Constraint<T>(a, b, ConstraintOperation::less_eq);
243 }
244
245 template <typename T>
246 Constraint<T> operator>(const Expression<T> &a, const Expression<T> &b) {
247     return Constraint<T>(a, b, ConstraintOperation::greater);
248 }
249
250 template <typename T>
251 Constraint<T> operator<(const Expression<T> &a, const Expression<T> &b) {
252     return Constraint<T>(a, b, ConstraintOperation::less);
253 }
254
255 template <typename T>
256 T eval(const Expression<T> &e, const map<Variable<T>, T> &values) {
257     auto visitor = std::make_unique<EvalVisitor<T>>();
258     return visitor->eval(e, values);
259 }
260
261 } // namespace lib
262
263 #endif

```

1.45. Template

```

1 #include <bits/stdc++.h>
2 #define int long long
3 using namespace std;
4
5 #define mp make_pair
6 #define mt make_tuple
7 #define pb push_back
8 #define ms(v, x) memset((v), (x), sizeof(v))
9 #define all(v) (v).begin(), (v).end()
10 #define ff first
11 #define ss second

```

```

12 #define iopt ios::sync_with_stdio(false); cin.tie(0)
13 #define untie(p, a, b) decltype(p.first) a = p.first, decltype(p.second) b =
14     p.second
15 #define TESTCASE(tn) cout << "Case #" << tn << ": "
16
17 int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
18
19 int floor2(int x, int y);
20 int ceil2(int x, int y) {
21     if(y < 0) return ceil2(-x, -y);
22     return x < 0 ? -floor2(-x, y) : (x + y - 1) / y;
23 }
24 int floor2(int x, int y) {
25     if(y < 0) return floor2(-x, -y);
26     return x < 0 ? -ceil2(-x, y) : x / y;
27 }
28
29 typedef pair<int, int> ii;
30 typedef long double LD;
31 typedef vector<int> vi;
32
33 #define TC_MAIN int32_t main() { iopt; int T; cin >> T; for(int i = 1; i <=
34     T; i++) solve(i); }

```

1.46. Traits

```

1 #ifndef _LIB_TRAITS
2 #define _LIB_TRAITS
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace traits {
8
9 template <typename...> struct make_void { using type = void; };
10
11 template <typename... T> using void_t = typename make_void<T...>::type;
12
13 /// keep caide
14 template <typename Iterator>
15 using IteratorCategory = typename
16     iterator_traits<Iterator>::iterator_category;
17
18 /// keep caide
19 template <typename Container>
20 using IteratorCategoryOf = IteratorCategory<typename Container::iterator>;
21
22 /// keep caide
23 template <typename Iterator>
24 using IteratorValue = typename iterator_traits<Iterator>::value_type;
25
26 /// keep caide
27 template <typename Container>
28 using IteratorValueOf = IteratorValue<typename Container::iterator>;
29
30 /// keep caide
31 template <typename Iterator>
32 using IsRandomIterator =
33     is_base_of<random_access_iterator_tag, IteratorCategory<Iterator>>;
34
35 /// keep caide
36 template <typename Iterator>
37 using IsInputIterator =
38     is_base_of<input_iterator_tag, IteratorCategory<Iterator>>;

```

```

38
39 /// keep caide
40 template <typename Iterator>
41 using IsBidirectionalIterator =
42     is_base_of<bidirectional_iterator_tag, IteratorCategory<Iterator>>;
43
44 /// keep caide
45 template <typename Container>
46 using HasRandomIterator =
47     is_base_of<random_access_iterator_tag, IteratorCategoryOf<Container>>;
48
49 /// keep caide
50 template <typename Container>
51 using HasInputIterator =
52     is_base_of<input_iterator_tag, IteratorCategoryOf<Container>>;
53
54 /// keep caide
55 template <typename Container>
56 using HasBidirectionalIterator =
57     is_base_of<bidirectional_iterator_tag, IteratorCategoryOf<Container>>;
58 } // namespace traits
59 } // namespace lib
60
61 #endif

```

1.47. Treap

```

1 #ifndef _LIB_TREAP
2 #define _LIB_TREAP
3 #include "Random.cpp"
4 #include "SegtreeImplicit.cpp"
5 #include <bits/stdc++.h>
6
7 namespace lib {
8 using namespace std;
9 namespace treap {
10 template <typename T> struct SearchResult {
11     bool found;
12     T node;
13 };
14
15 struct EmptyPushdown {
16     template <typename Node>
17     inline void operator()(Node &no, Node *ln, Node *rn) const {}
18 };
19
20 struct EmptyCombiner {
21     template <typename Node>
22     inline void operator()(Node &no, Node *ln, Node *rn) const {}
23 };
24
25 template <typename T, typename Less = std::less<T>> struct DefaultNode {
26     T key;
27     int y;
28
29     DefaultNode() {}
30     DefaultNode(T key)
31         : key(key), y(rng_gen.uniform_int(numeric_limits<int>::max())) {}
32
33     inline bool operator<(const DefaultNode &rhs) const {
34         return Less()(key, rhs.key);
35     }
36
37     inline int priority() const { return y; }

```

```

38
39 template <typename Combiner>
40 inline static void combine(DefaultNode &no, DefaultNode *ln, DefaultNode
41     *rn,
42         const Combiner &combiner) {
43     combiner(no, ln, rn);
44 }
45
46 template <typename T, typename Combiner = EmptyCombiner,
47     typename Pushdown = EmptyPushdown, typename Less = std::less<T>,
48     typename TreapNode = DefaultNode<T, Less>,
49     template <class> class ManagerTemplate = seg::Implicit>
50 struct TreapManager {
51     using NodeManager = ManagerTemplate<TreapNode>;
52     typedef TreapNode tnode;
53     typedef typename NodeManager::vnode vnode;
54
55     Combiner combiner_fn;
56     Pushdown pushdown_fn;
57     NodeManager manager;
58
59     inline vnode make(T key) { return manager.make(TreapNode(key)); }
60     inline vnode null() const { return manager.invalid(); }
61     inline void push(vnode no) {}
62     inline void update(vnode no) {
63         if (!manager.has(no))
64             return;
65         combiner_fn(manager.ref(no), manager.ptr(manager.left(no)),
66             manager.ptr(manager.right(no)));
67     }
68
69     template <typename Checker> bool check(vnode no, const Checker &checker) {
70         if (!manager.has(no))
71             return false;
72         return checker(manager.ref(no), manager.ptr(manager.left(no)),
73             manager.ptr(manager.right(no)));
74     }
75
76     template <typename Checker>
77     vnode bsearch_last_impl(vnode no, const Checker &checker) {
78         push(no);
79         if (!manager.has(no))
80             return null();
81         if (check(manager.right(no), checker))
82             return bsearch_last_impl(manager.right(no), checker);
83         else if (check(no, checker))
84             return no;
85         else
86             return bsearch_last_impl(manager.left(no), checker);
87     }
88
89     template <typename Folder, typename Checker>
90     vnode bsearch_last_impl(vnode no, const Folder &folder,
91         const Checker &checker) {
92         push(no);
93         if (!manager.has(no))
94             return null();
95     }
96
97     template <typename Checker>
98     SearchResult<tnode> bsearch_last(vnode no, const Checker &checker) {
99         auto res = bsearch_last_impl(no, checker);
100         if (!manager.has(res))
101             return {false};

```

```

102     return {true, manager.value(res)};
103 }
104
105 vnode merge(vnode small, vnode large) {
106     push(small), push(large);
107     vnode res;
108     if (!manager.has(small))
109         res = manager.replace(small, large);
110     else if (!manager.has(large))
111         res = manager.replace(large, small);
112     else {
113         const auto &t_small = manager.ref(small);
114         const auto &t_large = manager.ref(large);
115         if (t_small.priority() > t_large.priority()) {
116             res = manager.persist(small);
117             merge(manager.right(res), large);
118         } else {
119             res = manager.persist(large);
120             merge(small, manager.left(res));
121         }
122     }
123     update(res);
124     return res;
125 }
126
127 template <typename Checker>
128 pair<vnode, vnode> split(vnode no, const Checker &checker) {
129     push(no);
130     if (!manager.has(no))
131         return {null(), null()};
132     pair<vnode, vnode> res;
133     no = manager.persist(no);
134     if (check(no, checker)) {
135         auto sp = split(manager.right(no), checker);
136         manager.replace(manager.right(no), sp.first);
137         res = {no, sp.second};
138     } else {
139         auto sp = split(manager.left(no), checker);
140         manager.replace(manager.left(no), sp.second);
141         res = {sp.first, no};
142     }
143     update(no);
144     return res;
145 }
146
147 template <typename Checker>
148 pair<vnode, vnode> split_on_node(vnode no, const Checker &checker) {
149     return split(no, [&checker](const TreapNode &no, TreapNode *ln,
150                               TreapNode *rn) { return checker(no); });
151 }
152
153 pair<vnode, vnode> split_on_key(vnode no, T x) {
154     return split_on_node(no, [&x](const TreapNode &no) { return no.key < x;
155 });
156 };
157 } // namespace treap
158 } // namespace lib
159
160 #endif

```

1.48. TwoSat

1 #ifndef _LIB_TWO_SAT

```

2 #define _LIB_TWO_SAT
3 #include "Graph.cpp"
4 #include <bits/stdc++.h>
5
6 namespace lib {
7     using namespace std;
8     namespace graph {
9         #define POS(x) (2*(x))
10        #define NEG(x) (2*(x)+1)
11        #define VAR(x) ((x) < 0 ? NEG(-(x)) : POS(x))
12
13        // TODO: reuse graph structure and extract tarjan
14        struct TwoSat {
15            int n, sz;
16            vector<vector<int>>> adj;
17
18            int tempo, cnt;
19            vector<int> low, vis, from;
20            stack<int> st;
21            vector<bool> res;
22
23            TwoSat(int n) : n(n), adj(2*n){}
24
25            int add_dummy() {
26                int res = adj.size();
27                for(int i = 0; i < 2; i++)
28                    adj.push_back(vector<int>());
29                return res;
30            }
31
32            int convert(int x) const { return 2*x; }
33            void add_edge(int a, int b) { adj[a].push_back(b); }
34            void or_clause(int a, int b){
35                add_edge(a^1, b);
36                add_edge(b^1, a);
37            }
38
39            void implication_clause(int a, int b){
40                or_clause(a^1, b);
41            }
42
43            void literal_clause(int x) { or_clause(x, x); }
44            void and_clause(int a, int b){
45                literal_clause(a);
46                literal_clause(b);
47            }
48
49            void xor_clause(int a, int b){
50                or_clause(a, b);
51                or_clause(a^1, b^1);
52            }
53
54            void nand_clause(int a, int b){
55                or_clause(a^1, b^1);
56            }
57
58            void nor_clause(int a, int b){
59                literal_clause(a^1);
60                literal_clause(b^1);
61            }
62
63            void equals(int a, int b){
64                implication_clause(a, b);
65                implication_clause(b, a);
66            }

```

```

67
68 void max_one_clause(const vector<int> & v){
69     vector<int> p;
70     for(int i = 0; i < v.size(); i++)
71         p.push_back(add_dummy());
72
73     for(int i = 0; i < v.size(); i++){
74         implication_clause(v[i], p[i]);
75         if(i+1 < v.size()){
76             implication_clause(p[i], p[i+1]);
77             implication_clause(p[i], v[i+1]^1);
78         }
79     }
80 }
81
82 void clear(){
83     for(int i = 0; i < adj.size(); i++)
84         adj[i].clear();
85 }
86
87 void tarjan(int u){
88     low[u] = vis[u] = ++tempo;
89     st.push(u);
90
91     for(int v : adj[u]){
92         if(!vis[v]){
93             tarjan(v);
94             low[u] = min(low[u], low[v]);
95         } else if(vis[v] > 0)
96             low[u] = min(low[u], vis[v]);
97     }
98
99     if(low[u] == vis[u]){
100         int k;
101         do{
102             k = st.top();
103             st.pop();
104             from[k] = cnt;
105             vis[k] = -1;
106         } while(k != u);
107         cnt++;
108     }
109 }
110
111 bool solve(){
112     sz = adj.size();
113     assert(sz%2 == 0);
114
115     low.assign(sz, 0);
116     vis.assign(sz, 0);
117     tempo = 0;
118     cnt = 0;
119     from.assign(sz, -1);
120     st = stack<int>();
121
122     res.assign(n, true);
123
124     for(int i = 0; i < sz; i++)
125         if(!vis[i])
126             tarjan(i);
127
128     for(int i = 0; i < sz; i += 2){
129         if(from[i] == from[i^1]) return false;
130         else if(from[i] > from[i^1] && (i>>1) < n)
131             res[i>>1] = false;

```

```

132     }
133
134     return true;
135 }
136
137 bool get(int i) const { return res[i]; }
138 };
139 } // namespace graph
140 } // namespace lib
141
142 #endif

```

1.49. VectorN

```

1 #ifndef _LIB_VECTOR_N
2 #define _LIB_VECTOR_N
3 #include <bits/stdc++.h>
4 #include "Traits.cpp"
5
6 #define VEC_CONST_OP(op, typ) \
7     type operator op(const typ rhs) const { \
8         auto res = *this; \
9         return res op##= rhs; \
10     }
11
12 #define VEC_BIN_OP(op) \
13     type& operator op##=(const type& rhs) { \
14         if(rhs.size() > this->size()) \
15             this->resize(rhs.size()); \
16         int sz = this->size(); \
17         for(int i = 0; i < (int)rhs.size(); i++) \
18             (*this)[i] op##= rhs[i]; \
19         for(int i = rhs.size(); i < sz; i++) \
20             (*this)[i] op##= 0; \
21         return *this; \
22     } \
23     VEC_CONST_OP(op, type)
24
25 #define VEC_SINGLE_OP(op, typ) \
26     type& operator op##=(const typ rhs) { \
27         for(auto& x : *this) \
28             x op##= rhs; \
29         return *this; \
30     } \
31     VEC_CONST_OP(op, typ)
32
33
34 namespace lib {
35 using namespace std;
36 template<typename T>
37 struct VectorN : vector<T> {
38     using type = VectorN<T>;
39
40     template <
41         typename Container,
42         typename enable_if<traits::HasInputIterator<Container>::value>::type *
43             = nullptr>
44     VectorN(const Container &container)
45         : vector<T>(container.begin(), container.end()) {}
46
47     VectorN(const initializer_list<T> &v)
48         : vector<T>(v.begin(), v.end()) {}
49
50     template<typename... Args>

```

```

50 VectorN( Args&&... args )
51 : vector<T>(std::forward<Args>(args)...) {}
52
53 VEC_BIN_OP(+)
54 VEC_BIN_OP(-)
55 VEC_BIN_OP(*)
56
57 VEC_SINGLE_OP(+, T&)
58 VEC_SINGLE_OP(-, T&)
59 VEC_SINGLE_OP(*, T&)
60 VEC_SINGLE_OP(/, T&)
61 VEC_SINGLE_OP(^, int64_t)
62
63 type operator-() const {
64     auto res = *this;
65     for(auto& x : res) x = -x;
66     return res;
67 }
68
69 type operator%(int n) const {
70     // TODO: get rid of this
71     // return *const_cast<type*>(this);
72     return *this;
73 }
74 };
75 } // namespace lib
76
77 #endif

```

2. ds

2.1. LiChaoTree

```

1  #ifndef _LIB_LI_CHAO_TREE
2  #define _LIB_LI_CHAO_TREE
3
4  #include <bits/stdc++.h>
5
6  namespace lib {
7  using namespace std;
8
9  template <typename D, typename T> struct LiChaoTree {
10     inline constexpr static T inf = numeric_limits<T>::max();
11
12     using Fn = function<T(D)>;
13     vector<Fn> fns;
14     vector<D> xs;
15     vector<int> t;
16
17     template <typename U = D,
18             typename enable_if<is_integral<U>::value>::type = nullptr>
19     LiChaoTree(D left, D right) {
20         assert(right > left);
21         xs = vector<D>(right - left);
22         iota(xs.begin(), xs.end(), left);
23         init();
24     }
25
26     LiChaoTree(const vector<D>& xs_) : xs(xs_) {
27         sort(xs.begin(), xs.end());
28         xs.resize(unique(xs.begin(), xs.end()) - xs.begin());
29         init();
30     }
31

```

```

32 void init() {
33     t = vector<int>(xs.size() * 4);
34     fns.clear();
35     fns.push_back([](D x) { return numeric_limits<T>::max(); });
36 }
37
38 void add(const Fn &fn) {
39     int i = fns.size();
40     fns.push_back(fn);
41     add(i, 1, 0, xs.size());
42 }
43
44 // r is exclusive
45 void add(int i, int no, int l, int r) {
46     while (1) {
47         int mid = (l + r) / 2;
48         bool l_wins = fns[i](xs[l]) < fns[t[no]](xs[l]);
49         bool r_wins = fns[i](xs[r-1]) < fns[t[no]](xs[r-1]);
50         if (l_wins == r_wins) {
51             if (l_wins) swap(i, t[no]);
52             return;
53         }
54         bool mid_wins = fns[i](xs[mid]) < fns[t[no]](xs[mid]);
55         if (mid_wins)
56             swap(i, t[no]);
57         if (l + 1 == r)
58             return;
59         if (l_wins != mid_wins)
60             no = 2 * no, r = mid;
61         else
62             no = 2 * no + 1, l = mid;
63     }
64 }
65
66 int seg_l, seg_r, seg_idx;
67 void add_segment(int no, int l, int r) {
68     if (seg_l >= r || seg_r <= l) return;
69     if (seg_l <= l && r <= seg_r) add(seg_idx, no, l, r);
70     else {
71         int mid = (l+r)/2;
72         add_segment(2*no, l, mid);
73         add_segment(2*no+1, mid, r);
74     }
75 }
76
77 void add_segment(const Fn& fn, D a, D b) {
78     int i = fns.size();
79     fns.push_back(fn);
80     int l = lower_bound(xs.begin(), xs.end(), a) - xs.begin();
81     int r = lower_bound(xs.begin(), xs.end(), b) - xs.begin();
82     if (l == r) return;
83     seg_idx = i, seg_l = l, seg_r = r;
84     add_segment(1, 0, xs.size());
85 }
86
87 T query(D x, int no, int l, int r) const {
88     auto res = inf;
89     while (1) {
90         res = min(res, fns[t[no]](x));
91         if (l + 1 == r)
92             return res;
93         int mid = (l + r) / 2;
94         if (x < xs[mid])
95             no = 2 * no, r = mid;
96         else

```

```

97         no = 2 * no + 1, l = mid;
98     }
99 }
100
101 T query(D x) const { return query(x, 1, 0, xs.size()); }
102 };
103 } // namespace lib
104
105 #endif

```

2.2. OrderedIntTree

```

1  #ifndef _LIB_ORDERED_INT_TREE
2  #define _LIB_ORDERED_INT_TREE
3  #include <bits/stdc++.h>
4
5  namespace lib {
6  using namespace std;
7
8  namespace ds {
9
10     template <typename T>
11     struct Node {
12         int key;
13         T data;
14     };
15
16     template<>
17     struct Node<void> {
18         int key;
19     };
20
21     template <typename T = void>
22     struct OrderedIntTree {
23
24     };
25
26     }
27 } // namespace lib
28
29 #endif

```

2.3. StaticRMQ

```

1  #ifndef _LIB_STATIC_RMQ
2  #define _LIB_STATIC_RMQ
3  #include <bits/stdc++.h>
4
5  namespace lib {
6  using namespace std;
7  namespace {
8      inline int lsb(int x) { return x&-x; }
9  }
10
11  // Credits: hly1204
12  template<typename T, typename Cmp = std::less<T>>
13  struct StaticRMQ {
14      Cmp cmp;
15      vector<T> t1, t2, a;
16
17      StaticRMQ() {}
18
19      StaticRMQ(const vector<T>& a)

```

```

20         : t1(a.size() + 1), t2(a.size() + 1), a(a) {
21             copy(a.begin(), a.end(), t1.begin() + 1);
22             copy(a.begin(), a.end(), t2.begin() + 1);
23             build();
24         }
25
26     int size() const { return (int)t1.size() - 1; }
27
28     T best(const T& a, const T& b) const {
29         return cmp(a, b) ? a : b;
30     }
31
32     void build() {
33         int n = size();
34         for(int i = 1; i <= n; i++) {
35             int b = lsb(i);
36             if(i + b <= n) t1[i + b] = best(t1[i + b], t1[i]);
37         }
38         for(int i = n; i; i--) {
39             int b = lsb(i);
40             t2[i - b] = best(t2[i - b], t2[i]);
41         }
42     }
43
44     // [l, r], 0-indexed
45     T query(int l, int r) const {
46         if(l == r) return a[l];
47         ++l, ++r;
48         T ans = best(a[l-1], a[r-1]);
49         int x = l;
50         for(; x + lsb(x) - 1 <= r; x += lsb(x))
51             ans = best(ans, t2[x]);
52         for(int y = r; y != 0 && y - lsb(y) + 1 >= l; y -= lsb(y))
53             ans = best(ans, t1[y]);
54         if(x <= r)
55             ans = best(ans, a[x-1]);
56         return ans;
57     }
58 };
59 } // namespace lib
60
61 #endif

```

3. dsu

3.1. BinaryLifting

```

1  #ifndef _LIB_DSU_BINARY_LIFTING
2  #define _LIB_DSU_BINARY_LIFTING
3  #include <bits/stdc++.h>
4  #include "SpanningTree.cpp"
5
6  namespace lib {
7  using namespace std;
8  namespace dsu {
9
10     template<typename D>
11     struct BinaryLifting : public D {
12         using D::parent;
13         vector<vector<int>>> P;
14         int K;
15
16         BinaryLifting() : D() {}
17         BinaryLifting(int n) : D(n) {

```

```

18     P = decltype(P)(n, vector<int>(__lg(n)+1, -1));
19     K = __lg(n)+1;
20 }
21 virtual void clear() override {
22     D::clear();
23     int n = P.size();
24     P = decltype(P)(n, vector<int>(K, -1));
25 }
26 virtual int merge(int u, int v) override {
27     if(!D::merge(u, v)) return 0;
28     this->traverse_last_small([this](int u, int p, vector<int>&) {
29         for(int& x : P[u]) x = -1;
30         P[u][0] = p;
31         for(int i = 1; i < K; i++) {
32             if(P[u][i-1] == -1) break;
33             P[u][i] = P[P[u][i-1]][i-1];
34         }
35     }, no_op_visitor);
36     return 1;
37 }
38 int parent(int u, int k) {
39     assert(k >= 0);
40     for(int i = K-1; i >= 0; i--) {
41         if(!((k>>i)&1)) continue;
42         u = P[u][i];
43         if(u == -1) return -1;
44     }
45     return u;
46 }
47 };
48 } // namespace dsu
49 } // namespace lib
50
51 #endif

```

3.2. Compress

```

1 #ifndef _LIB_DSU_COMPRESS
2 #define _LIB_DSU_COMPRESS
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace dsu {
8
9 template<typename D>
10 struct Compress : public D {
11     using D::r;
12
13     Compress() : D() {}
14     Compress(int n) : D(n) {}
15
16     virtual int get(int i) const override {
17         return r[i] == i ? i : r[i] = get(r[i]);
18     }
19 };
20 } // namespace dsu
21 } // namespace lib
22
23 #endif

```

3.3. DSU

```

1 #ifndef _LIB_RANK_DSU
2 #define _LIB_RANK_DSU
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace dsu {
8 struct RankDSU {
9     mutable vector<int> r, sz;
10     pair<int, int> last_merge_ = {-1, -1};
11     bool last_swapped_ = false;
12     int merges = 0;
13     RankDSU() {}
14     RankDSU(int n) : r(n), sz(n, 1) {
15         iota(r.begin(), r.end(), 0);
16     }
17     virtual void clear() {
18         iota(r.begin(), r.end(), 0);
19         fill(sz.begin(), sz.end(), 1);
20         last_merge_ = {-1, -1};
21         merges = 0;
22     }
23     virtual int get(int i) const {
24         return r[i] == i ? i : get(r[i]);
25     }
26     int operator[](int i) const {
27         return get(i);
28     }
29     pair<int, int> last_merge() const {
30         return last_merge_;
31     }
32     int n_comps() const { return (int)r.size() - merges; }
33     virtual void merged(int u, int v) {}
34     virtual int merge(int u, int v) {
35         u = get(u), v = get(v);
36         if(u == v) return 0;
37         last_swapped_ = false;
38         if(sz[u] > sz[v]) swap(u, v), last_swapped_ = true;
39         r[u] = v;
40         sz[v] += sz[u];
41         last_merge_ = {u, v};
42         merges++;
43         merged(u, v);
44         return 1;
45     }
46 };
47
48 template<template<class> class ...Ts>
49 struct ByRankImpl;
50
51 template<template<class> class T, template<class> class ...Ts>
52 struct ByRankImpl<T, Ts...> {
53     using type = T<typename ByRankImpl<Ts...>::type>;
54 };
55
56 template<>
57 struct ByRankImpl<> {
58     using type = RankDSU;
59 };
60
61 template<template<class> class ...Ts>
62 using ByRank = typename ByRankImpl<Ts...>::type;
63 } // namespace dsu
64 } // namespace lib

```

```
65 #endif
66
```

3.4. SpanningTree

```
1 #ifndef _LIB_DSU_SPANNING_TREE
2 #define _LIB_DSU_SPANNING_TREE
3 #include <bits/stdc++.h>
4 #include "../utils/LazyArray.cpp"
5
6 namespace lib {
7 using namespace std;
8 namespace dsu {
9
10 const auto no_op_visitor = [] (int, int, const vector<int>&) -> void {};
11
12 template<typename D>
13 struct SpanningTree : public D {
14     using D::last_swapped_;
15
16     vector<vector<int>>> adj;
17     vector<int> pai, depth;
18     LazyArray<char> vis;
19     pair<int, int> last_edge_;
20
21     SpanningTree() : D() {}
22     SpanningTree(int n) : D(n), adj(n), pai(n, -1), vis(n, 0), depth(n, 0) {}
23     virtual void clear() override {
24         D::clear();
25         for(int i = 0; i < adj.size(); i++)
26             adj[i].clear();
27         fill(pai.begin(), pai.end(), -1);
28         fill(depth.begin(), depth.end(), 0);
29         vis.clear();
30         last_edge_ = {-1, -1};
31     }
32     virtual int merge(int u, int v) override {
33         if(!D::merge(u, v)) return 0;
34         if(last_swapped_)
35             swap(u, v);
36         last_edge_ = {u, v};
37         vis.clear();
38         fix_(u, v, depth[v]+1);
39         adj[u].push_back(v);
40         adj[v].push_back(u);
41         return 1;
42     }
43     template<typename F, typename G>
44     void traverse_last_small(const F& f, const G& g) {
45         vis.clear();
46         traverse_(last_edge_.first, last_edge_.second, f, g);
47     }
48     template<typename F, typename G>
49     void traverse_(int u, int p, const F& f, const G& g) {
50         if(vis.get(u)) return;
51         vis[u] = 1;
52         f(u, p, adj[u]);
53         for(int v : adj[u]) {
54             if(v == p || vis.get(v)) continue;
55             traverse_(v, u, f, g);
56         }
57         g(u, p, adj[u]);
58     }
59     void fix_(int u, int p, int d) {
```

```
60     if(vis.get(u)) return;
61     vis[u] = 1;
62     pai[u] = p;
63     depth[u] = d;
64     for(int v : adj[u]) {
65         if(v == p || vis.get(v)) continue;
66         fix_(v, u, d+1);
67     }
68 }
69 pair<int, int> last_edge() const {
70     return last_edge_;
71 }
72 int parent(int i) const {
73     return pai[i];
74 }
75 };
76 } // namespace dsu
77 } // namespace lib
78
79 #endif
```

3.5. Time

```
1 #ifndef _LIB_DSU_TIME
2 #define _LIB_DSU_TIME
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace dsu {
8
9     template<typename D>
10     struct Time : public D {
11         using D::r;
12         using D::sz;
13
14         vector<int> t;
15         int tempo = 0;
16         Time() : D() {}
17         Time(int n) : D(n), t(n, 1e9) {}
18         virtual void clear() override {
19             tempo = 0;
20             fill(t.begin(), t.end(), (int)1e9);
21         }
22         int get(int i, int tt) const {
23             return r[i] == i ? i : (t[i] <= tt ? get(r[i]) : i);
24         }
25         int get_merge_time(int u, int v) const {
26             int ans = -1;
27             while(u != v) {
28                 if(sz[u] < sz[v]) swap(u, v);
29                 ans = max(ans, t[v]);
30                 if(r[v] == v) return -1;
31                 v = r[v];
32             }
33             return ans;
34         }
35         Time& at_time(int tt) {
36             assert(tt >= tempo);
37             tempo = tt;
38             return *this;
39         }
40         Time& tick() {
41             return at_time(tempo+1);
```



```

42     }
43     virtual void merged(int u, int v) override {
44         D::merged(u, v);
45         t[u] = tempo;
46     }
47 };
48 } // namespace dsu
49 } // namespace lib
50
51 #endif

```

4. graphs

4.1. BlockCut

```

1  #ifndef _LIB_BLOCK_CUT
2  #define _LIB_BLOCK_CUT
3  #include <bits/stdc++.h>
4  #include "../Graph.cpp"
5  #include "../utils/LazyArray.cpp"
6
7  namespace lib {
8      using namespace std;
9  namespace graph {
10     template<typename V, typename E>
11     struct BlockCut {
12         int n, m;
13         Graph<V, E> g;
14         int tempo = 0;
15         vector<int> vis, low, seen;
16         vector<int> st;
17         LazyArray<char> seen_v;
18
19         Graph<V, E> g2;
20         int n2 = 0;
21
22         BlockCut(const Graph<V, E>& g) : g(g) {
23             n = g.size();
24             m = g.edge_size();
25             vis = low = vector<int>(n);
26             seen = vector<int>(m);
27             st.reserve(m);
28             seen_v = LazyArray<char>(n, 0);
29
30             g2 = Graph<V, E>(n);
31
32             for(int i = 0; i < n; i++) {
33                 if(!vis[i]) {
34                     tarjan(i, -1);
35                     if(g.degree(i) == 0) {
36                         // Vertex is isolated, process separately.
37                         g2.add_vertex();
38                         g2.add_2edge(n + n2, i);
39                         n2++;
40                     }
41                 }
42             }
43         }
44         Graph<V, E> graph() const { return g2; }
45
46         int n_components() const { return n2; }
47         vector<int> component(int i) const {
48             vector<int> res;
49             for(const auto& v : g2.n_edges(n + i))

```

```

50             if (v.to < n)
51                 res.push_back(v.to);
52             return res;
53         }
54
55         vector<int> get_vertices_(const vector<int>& e) {
56             seen_v.clear();
57             vector<int> comp;
58             for(int kk : e) {
59                 auto ed = g.edge(kk);
60                 if(!seen_v.get(ed.from)) comp.push_back(ed.from), seen_v[ed.from] =
61                     true;
62                 if(!seen_v.get(ed.to)) comp.push_back(ed.to), seen_v[ed.to] = true;
63             }
64             return comp;
65         }
66         void process_component_(int k) {
67             vector<int> e;
68             int cur;
69             do {
70                 cur = st.back(); st.pop_back();
71                 e.push_back(cur);
72             } while(cur != k);
73             auto comp = get_vertices_(e);
74             g2.add_vertex();
75             for(int w : comp) {
76                 g2.add_2edge(n + n2, w);
77             }
78             n2++;
79         }
80         void tarjan(int u, int p) {
81             vis[u] = low[u] = ++tempo;
82             auto nei = g.n_edges(u);
83             for(int i = 0; i < nei.size(); i++) {
84                 int k = nei.index(i);
85                 int v = g.edge(k).to;
86
87                 if(!seen[k]) {
88                     seen[k] = seen[k^1] = 1;
89                     st.push_back(k);
90                 }
91
92                 if(!vis[v]) {
93                     tarjan(v, u);
94                     low[u] = min(low[u], low[v]);
95
96                     if(low[v] >= vis[u]) {
97                         process_component_(k);
98                     }
99                     else {
100                         low[u] = min(low[u], vis[v]);
101                     }
102                 }
103             }
104         }
105     };
106
107     template<typename V, typename E>
108     BlockCut<V, E> make_block_cut(const Graph<V, E>& g) {
109         return BlockCut<V, E>(g);
110     }
111 } // namespace graph
112 } // namespace lib
113
114 #endif

```



```

129         for(const auto& e2 : g.n_edges(x)) {
130             int y = e2.to;
131             if(vis[y]) continue;
132             if(y == i) continue;
133             if(y != v && binary_search(adj[i].begin(), adj[i].end(), y))
continue;
134             vis[y] = 1;
135             q.push(y);
136             par[y] = x;
137         }
138     }
139     cyc.clear();
140     cyc.push_back(e.to);
141     cyc.push_back(i);
142     assert(vis[v]);
143     for(auto x = v; x != e.to; x = par[x]) cyc.push_back(x);
144     return false;
145 }
146 }
147 }
148 was_tested = true;
149 return true;
150 }
151
152 vector<int> induced_cycle() const { return cyc; }
153
154 vector<int> max_independent_set() const {
155     int n = g.size();
156     vis.assign(n, 0);
157
158     vector<int> res;
159     for(int i : order) {
160         if(vis[i]) continue;
161         res.push_back(i);
162         for(const auto& e : g.n_edges(i)) {
163             vis[e.to] = 1;
164         }
165     }
166     return res;
167 }
168 };
169
170 template<typename Graph>
171 Chordal<Graph> make_chordal(const Graph& g) {
172     return Chordal<Graph>(g);
173 }
174 } // namespace graph
175 } // namespace lib
176
177 #endif

```

5. matroid

5.1. CographicMatroid

```

1 #ifndef _LIB_COGRAPHIC_MATROID
2 #define _LIB_COGRAPHIC_MATROID
3 #include <bits/stdc++.h>
4 #include "GraphicMatroid.cpp"
5
6 namespace lib {
7     using namespace std;
8     struct CographicMatroid : GraphicMatroid {
9

```

```

10     CographicMatroid(int n, std::function<pair<int, int>(int)> edge_fn_)
11         : GraphicMatroid(n, edge_fn_) {}
12
13     void build(const lambda::SubsetFilter& I_) override {
14         GraphicMatroid::build(!I_);
15     }
16     void setup_exchange(int i) {
17         setup();
18     }
19     bool can_exchange(int i, int j) {
20         return can_add(j);
21     }
22     bool can_add(int i) {
23         return !is_bridge(i);
24     }
25 };
26 } // namespace lib
27
28 #endif

```

5.2. ColorMatroid

```

1 #ifndef _LIB_COLOR_MATROID
2 #define _LIB_COLOR_MATROID
3 #include <bits/stdc++.h>
4 #include "Matroid.cpp"
5
6 namespace lib {
7     using namespace std;
8     struct ColorMatroid : Matroid {
9         vector<int> cnt, limits;
10         lambda::Map<int> color;
11         ColorMatroid(vector<int> limits, const lambda::Map<int>& color_)
12             : Matroid(), limits(limits), color(color_) {}
13         ColorMatroid(int n, int K, const lambda::Map<int>& color_)
14             : Matroid(), limits(n, K), color(color_) {}
15         void setup() {
16             cnt.assign(limits.size(), 0);
17             for(int i = 0; i < ground_set_size(); i++)
18                 if(in_I(i))
19                     cnt[color(i)]++;
20         }
21         void setup_exchange(int i) {
22             cnt[color(i)]--;
23         }
24         void finish_exchange(int i) {
25             cnt[color(i)]++;
26         }
27         bool can_exchange(int i, int j) {
28             return can_add(j);
29         }
30         bool can_add(int i) {
31             int c = color(i);
32             return cnt[c] < limits[c];
33         }
34         void print() const {
35             for(int x : cnt) cout << x << " ";
36             cout << endl;
37         }
38     };
39 } // namespace lib
40
41 #endif

```

5.3. Compose

```

1 #ifndef _LIB_COMPOSE_MATROID
2 #define _LIB_COMPOSE_MATROID
3 #include <bits/stdc++.h>
4 #include "Matroid.cpp"
5 #include "../Lambda.cpp"
6
7 namespace lib {
8     using namespace std;
9     namespace matroid {
10         template<typename M>
11         struct Filter : Matroid {
12             M mat;
13             lambda::Filter filter_fn;
14             lambda::SubsetMap<int> inv_fn;
15             Filter(const M& mat_, const lambda::Filter& filter_fn_)
16                 : Matroid(), mat(mat_), filter_fn(filter_fn_) {}
17
18             void build(const lambda::SubsetFilter& I_) override {
19                 Matroid::build(I_);
20                 auto subset = filter_fn.subset(I_.size());
21                 inv_fn = subset.take_inverse();
22                 mat.build(subset.take_from(I_));
23             }
24
25             void setup() { mat.setup(); }
26             void setup_graph() { mat.setup_graph(); }
27             void setup_exchange(int i) {
28                 mat.setup_exchange(inv_fn(i));
29             }
30             void finish_exchange(int i) {
31                 mat.finish_exchange(inv_fn(i));
32             }
33
34             bool can_add(int i) {
35                 if(!filter_fn(i)) return true;
36                 return mat.can_add(inv_fn(i));
37             }
38             bool can_exchange(int i, int j) {
39                 if(!filter_fn(i)) return can_add(j);
40                 if(!filter_fn(j)) return true;
41                 return mat.can_exchange(inv_fn(i), inv_fn(j));
42             }
43 };
44
45 template<typename M>
46 Filter<M> make_filter(const M& mat, const lambda::Filter& fn) {
47     return Filter<M>(mat, fn);
48 }
49 // namespace matroid
50 // namespace lib
51
52 #endif

```

5.4. GraphicMatroid

```

1 #ifndef _LIB_GRAPHIC_MATROID
2 #define _LIB_GRAPHIC_MATROID
3 #include <bits/stdc++.h>
4 #include "Matroid.cpp"
5 #include "../utils/FastAdj.cpp"
6
7 namespace lib {

```

```

8     using namespace std;
9     struct GraphicMatroid : Matroid {
10         lambda::Map<pair<int, int>>> edge;
11         FastAdj<pair<int, int>> g;
12         vector<int> comp, st, nd, low;
13         vector<int> bridges;
14         int tempo, comps;
15         bool printer = true;
16
17         GraphicMatroid(int n, const lambda::Map<pair<int, int>>& edge_)
18             : Matroid(), edge(edge_), g(n, n) {}
19         void setup() {
20             g.clear();
21             g.reserve(ground_set_size());
22             for(int i = 0; i < ground_set_size(); i++)
23                 if(in_I(i)) {
24                     auto p = edge(i);
25                     g.add(p.first, {p.second, i});
26                     g.add(p.second, {p.first, i});
27                 }
28             build_graph();
29         }
30         void build_graph() {
31             int n = g.size();
32             comp.assign(n, -1);
33             st.assign(n, 0);
34             nd.assign(n, 0);
35             low.assign(n, 0);
36             bridges.assign(ground_set_size(), 0);
37
38             tempo = 0;
39             comps = 0;
40             for(int i = 0; i < n; i++) {
41                 if(comp[i] == -1) dfs(i, -1, comps++);
42             }
43         }
44         void dfs(int u, int p, int c) {
45             comp[u] = c;
46             st[u] = low[u] = tempo++;
47             for(auto e : g.n_edges(u)) {
48                 int v = e.first;
49                 if(v == p) {
50                     p = -1;
51                     continue;
52                 }
53                 if(comp[v] != -1) low[u] = min(low[u], st[v]);
54                 else {
55                     dfs(v, u, c);
56                     low[u] = min(low[u], low[v]);
57                     if(low[v] > st[u]) {
58                         bridges[e.second] = 1;
59                     }
60                 }
61             }
62             nd[u] = tempo++;
63         }
64         bool is_bridge(int i) {
65             return bridges[i];
66         }
67         bool is_anc(int u, int v) {
68             return st[u] <= st[v] && st[v] <= nd[u];
69         }
70         bool can_exchange(int i, int j) {
71             auto e1 = edge(i);
72             auto e2 = edge(j);

```

```

73     if(st[e1.first] > st[e1.second]) swap(e1.first, e1.second);
74     return is_anc(e1.second, e2.first) + is_anc(e1.second, e2.second) == 1;
75 }
76 bool can_add(int i) {
77     auto e = edge(i);
78     return comp[e.first] != comp[e.second];
79 }
80 };
81 } // namespace lib
82
83 #endif

```

5.5. Matroid

```

1  #ifndef _LIB_MATROID
2  #define _LIB_MATROID
3  #include <bits/stdc++.h>
4  #include "../Lambda.cpp"
5
6  namespace lib {
7  struct Matroid {
8      lambda::SubsetFilter I;
9      bool in_I(int i) const {
10         return I(i);
11     }
12     vector<bool> get_I() const {
13         return I();
14     }
15     int ground_set_size() const { return I.size(); }
16
17     /** docstring
18      * Used to build a Matroid object from an M (independent set provider).
19      */
20     virtual void build(const lambda::SubsetFilter& I_) {
21         I = I_;
22     }
23
24     void setup() {}
25     void setup_graph() {}
26     void setup_exchange(int i) {}
27     void finish_exchange(int i) {}
28
29     bool can_add(int i) { return false; }
30     bool can_exchange(int i, int j) { return false; }
31
32     void print_I() {
33         for(int i = 0; i < I.size(); i++) cout << in_I(i) << " ";
34         cout << endl;
35     }
36 };
37 } // namespace lib
38
39 #endif

```

5.6. MatroidIntersection

```

1  #ifndef _LIB_MATROID_INTERSECTION
2  #define _LIB_MATROID_INTERSECTION
3  #include <bits/stdc++.h>
4  #include "../utils/FastAdj.cpp"
5  #include "../utils/FastQueue.cpp"
6  #include "../Lambda.cpp"

```

```

8  namespace lib {
9  template<typename M1, typename M2, typename W = int>
10 struct MatroidIntersection {
11     int n;
12     M1 m1;
13     M2 m2;
14
15     // aux vectors
16     vector<int> vI;
17     vector<int> I;
18     vector<int> nd;
19     FastQueue<int> q;
20     vector<int> p;
21     vector<int> ch;
22     vector<int> in_q;
23     vector<W> w;
24     vector<W> dist;
25
26     FastAdj<int> g;
27
28     MatroidIntersection() : q(1) { init (); }
29     MatroidIntersection(int n, const M1& m1, const M2& m2) : m1(m1), m2(m2),
30         n(n), g(n+2, n), q(n) {
31         init();
32     }
33     void set_weights(const vector<W>& w_) {
34         assert(n == w_.size());
35         w = w_;
36     }
37     int size() const { return n; }
38     void init() {
39         vI.reserve(n);
40         p.assign(n, -1);
41         I.assign(n, false);
42         nd.assign(n, 0);
43     }
44     void setup_augment() {
45         vI.clear();
46         g.clear();
47         for(int i = 0; i < n; i++) {
48             if(I[i]) vI.push_back(i);
49             nd[i] = 0;
50         }
51     }
52     bool is_weighted() const {
53         return !w.empty();
54     }
55     bool augment(int truncate = 1e9) {
56         setup_augment();
57         if(vI.size() == min(truncate, n)) return false;
58         auto f = lambda::SubsetFilter(n, [this](int i) -> bool { return in_I(i); });
59         m1.build(f), m2.build(f);
60         m1.setup(), m2.setup();
61         // Check potential starting and ending points of the path.
62         // Also, return earlier if is both starting and ending point.
63         for(int i = 0; i < n; i++) {
64             if(I[i]) continue;
65             if(m1.can_add(i)) nd[i] |= 1;
66             if(m2.can_add(i)) nd[i] |= 2;
67             if(nd[i] == 3 && !is_weighted()) {
68                 I[i] = true;
69                 return true;
70             }
71         }
72     }
73 };
74 }

```

```

71 m1.setup_graph(), m2.setup_graph();
72 for(int i : v1) {
73     I[i] = false;
74     m1.setup_exchange(i), m2.setup_exchange(i);
75     for(int j = 0; j < n; j++) {
76         if(I[j] || i == j) continue;
77         if(m1.can_exchange(i, j)) g.add(i, j);
78         if(m2.can_exchange(i, j)) g.add(j, i);
79     }
80     I[i] = true;
81     m1.finish_exchange(i), m2.finish_exchange(i);
82 }
83
84 int st = is_weighted() ? weighted_sp() : unweighted_sp();
85 if(st == -1) return false;
86 I[st] ^= 1;
87 while(p[st] != st) {
88     st = p[st];
89     I[st] ^= 1;
90 }
91 return true;
92 }
93 int unweighted_sp() {
94     q.clear();
95     p.assign(n, -1);
96     for(int i = 0; i < n; i++)
97         if(nd[i]&1) q.push(i), p[i] = i;
98
99     int st = -1;
100     while(!q.empty() && st == -1) {
101         int u = q.pop();
102         if(nd[u]&2) {
103             st = u;
104             break;
105         }
106         for(int v : g.n_edges(u)) {
107             if(p[v] == -1) {
108                 p[v] = u;
109                 q.push(v);
110             }
111         }
112     }
113     return st;
114 }
115 int weighted_sp() {
116     q.clear();
117     in_q.assign(n, 0);
118     p.assign(n, -1);
119     const W oo = numeric_limits<W>::max() / 2;
120     ch.assign(n, 1e9);
121     dist.assign(n, oo);
122     for(int i = 0; i < n; i++)
123         if(nd[i]&1)
124             dist[i] = -w[i], ch[i] = 0, p[i] = i, q.push(i), in_q[i] = 1;
125     while(!q.empty()) {
126         int i = q.pop();
127         in_q[i] = 0;
128         for(int v : g.n_edges(i)) {
129             if(v == i) continue;
130             W n_dist = dist[i] + (I[v] ? w[v] : -w[v]);
131             int n_ch = ch[i] + 1;
132             using ii = pair<W, int>;
133             if(ii(n_dist, n_ch) < ii(dist[v], ch[v])) {
134                 dist[v] = n_dist;
135                 ch[v] = n_ch;

```

```

136         p[v] = i;
137         if(!in_q[v]) {
138             in_q[v] = 1;
139             q.push(v);
140         }
141     }
142 }
143
144 pair<pair<W, int>, int> best = {{oo, 1e9}, -1};
145 for(int i = 0; i < n; i++) {
146     if(nd[i]&2) {
147         best = min(best, {{dist[i], ch[i]}, i});
148     }
149 }
150 return best.second;
151 }
152
153 vector<int> solve(int truncate = 1e9) {
154     while(augment(truncate));
155     return I;
156 }
157 W cost() const {
158     W res = 0;
159     for(int i = 0; i < n; i++) {
160         if(I[i])
161             res += is_weighted() ? w[i] : 1;
162     }
163     return res;
164 }
165 int cardinality() const {
166     int res = 0;
167     for(int i = 0; i < n; i++)
168         res += I[i];
169     return res;
170 }
171 bool in_I(int i) const {
172     return I[i];
173 }
174 void flip(int i) {
175     I[i] ^= 1;
176 }
177 const vector<int>& get_I() const {
178     return I;
179 }
180 };
181
182 template<typename M1, typename M2>
183 shared_ptr<MatroidIntersection<M1, M2>> make_matroid_intersection(int n,
184     const M1& m1, const M2& m2) {
185     return make_shared<MatroidIntersection<M1, M2>>(n, m1, m2);
186 }
187 template<typename W, typename M1, typename M2>
188 shared_ptr<MatroidIntersection<M1, M2, W>>
189     make_weighted_matroid_intersection(int n, const M1& m1, const M2& m2,
190     const lambda::Map<W>& f) {
191     auto res = make_shared<MatroidIntersection<M1, M2, W>>(n, m1, m2);
192     vector<W> w(n);
193     for(int i = 0; i < n; i++) w[i] = f(i);
194     res->set_weights(w);
195     return res;
196 }
197 } // namespace lib
198 #endif

```

6. geometry

6.1. Caliper

```

1 #ifndef _LIB_GEOMETRY_CALIPER
2 #define _LIB_GEOMETRY_CALIPER
3 #include "Line2D.cpp"
4 #include "Polygon2D.cpp"
5 #include <bits/stdc++.h>
6
7 namespace lib {
8 using namespace std;
9 namespace geo {
10 namespace plane {
11 template <typename T, typename Large = T,
12         typename enable_if<!is_integral<T>::value>::type * = nullptr,
13         typename enable_if<!is_integral<T>::value>::type * = nullptr>
14 struct Caliper {
15     typedef Point<T, Large> point;
16     typedef Line<T, Large> line;
17     point p;
18     Large ang;
19     Caliper(point a, Large alpha) : p(a) {
20         ang = remainder(alpha, 2 * trig::PI);
21         while (ang < 0)
22             ang += 2 * trig::PI;
23     }
24     Large angle_to(const point &q) const {
25         return remainder(arg(q - p) - ang, 2 * trig::PI);
26     }
27     void rotate(double theta) {
28         ang += theta;
29         while (ang > 2 * trig::PI)
30             ang -= 2 * trig::PI;
31         while (ang < 0)
32             ang += 2 * trig::PI;
33     }
34     void move(const point &q) { p = q; }
35     point versor() const { return point::polar(1.0, ang); }
36     line as_line(Large scale = 1.0) const {
37         return line(p, p + versor() * scale);
38     }
39     friend Large dist(const Caliper &a, const Caliper &b) {
40         return dist(a.as_line(), b.p);
41     }
42 };
43
44 template <typename T, typename Large = T> struct PolygonCalipers {
45     constexpr static Large LIMIT = 4 * acosl(-1);
46
47     typedef Point<T, Large> point;
48     typedef Caliper<T, Large> caliper;
49     typedef ConvexPolygon<T, Large> polygon;
50     typedef pair<int, Large> descriptor;
51
52     polygon poly;
53     vector<caliper> calipers;
54     vector<int> indices;
55     vector<int> walked;
56     Large angle_walked;
57
58     PolygonCalipers(const polygon &poly, const vector<descriptor> &descriptors)
59         : poly(poly), walked(descriptors.size()), angle_walked(0) {
60         indices.reserve(descriptors.size());
61         calipers.reserve(descriptors.size());

```

```

62         for (size_t i = 0; i < descriptors.size(); i++) {
63             calipers.emplace_back(poly[descriptors[i].first],
64                                   descriptors[i].second);
65             indices.emplace_back(descriptors[i].first);
66         }
67     caliper operator[](int i) const { return calipers[i]; }
68     int index(int i) const { return indices[i]; }
69     bool has_next() const {
70         return *min_element(walked.begin(), walked.end()) < poly.size() &&
71                angle_walked < LIMIT;
72     }
73     Large angle_to_next(int i) const {
74         int u = indices[i];
75         return calipers[i].angle_to(poly[u + 1]);
76     }
77     void step_(int i) {
78         int u = indices[i]++;
79         indices[i] %= poly.size();
80         calipers[i].move(poly[u + 1]);
81         walked[i]++;
82     }
83
84     void next() {
85         int i = 0;
86         Large best = angle_to_next(0);
87         for (size_t j = 1; j < calipers.size(); j++) {
88             Large cur = angle_to_next(j);
89             if (cur < best) {
90                 best = cur;
91                 i = j;
92             }
93         }
94         Large alpha = angle_to_next(i);
95         for (auto &caliper : calipers)
96             caliper.rotate(alpha);
97         step_(i);
98         angle_walked += alpha;
99     }
100 };
101 } // namespace plane
102
103 } // namespace geo
104 } // namespace lib
105
106 #endif

```

6.2. Circle2D

```

1 #ifndef _LIB_GEOMETRY_CIRCLE_2D
2 #define _LIB_GEOMETRY_CIRCLE_2D
3 #include "../utils/Annotation.cpp"
4 #include "Line2D.cpp"
5 #include <bits/stdc++.h>
6
7 namespace lib {
8 using namespace std;
9 namespace geo {
10 namespace plane {
11 template <typename T, typename Large = T> struct Barycentric {
12     typedef Point<T, Large> point;
13     point r1, r2, r3;
14     T a, b, c;
15

```

```

16 Barycentric(const point &r1, const point &r2, const point &r3, T a = 1,
17             T b = 1, T c = 1)
18     : r1(r1), r2(r2), r3(r3), a(a), b(b), c(c) {}
19 point as_point() const { return (r1 * a + r2 * b + r3 * c) / (a + b + c); }
20
21 static Barycentric centroid(const point &r1, const point &r2,
22                             const point &r3) {
23     return Barycentric(r1, r2, r3);
24 }
25 static Barycentric circumcenter(const point &r1, const point &r2,
26                                 const point &r3) {
27     Large a = norm_sq(r2 - r3), b = norm_sq(r3 - r1), c = norm_sq(r1 - r2);
28     return Barycentric(r1, r2, r3, a * (b + c - a), b * (c + a - b),
29                         c * (a + b - c));
30 }
31 static Barycentric incenter(const point &r1, const point &r2,
32                             const point &r3) {
33     return Barycentric(r1, r2, r3, norm(r2 - r3), norm(r1 - r3), norm(r1 -
34     r2));
35 }
36 static Barycentric orthocenter(const point &r1, const point &r2,
37                                const point &r3) {
38     Large a = norm_sq(r2 - r3), b = norm_sq(r3 - r1), c = norm_sq(r1 - r2);
39     return Barycentric(r1, r2, r3, (a + b - c) * (c + a - b),
40                         (b + c - a) * (a + b - c), (c + a - b) * (b + c - a));
41 }
42 static Barycentric excenter(const point &r1, const point &r2,
43                             const point &r3) {
44     return Barycentric(r1, r2, r3, -norm(r2 - r3), norm(r1 - r3),
45                         norm(r1 - r2));
46 }
47 };
48 template <typename T, typename Large = T> struct Circle {
49     typedef Point<T, Large> point;
50     typedef Line<T, Large> line;
51     typedef Barycentric<Large> bary;
52     typedef Segment<T, Large> segment;
53     point center;
54     T radius;
55
56     Circle(point center, T radius) : center(center), radius(radius) {}
57     Circle(const point &p1, const point &p2, const point &p3) {
58         center = bary::circumcenter(p1, p2, p3).as_point();
59         radius = dist(center, p1);
60     }
61     Circle(const point &p1, const point &p2) {
62         center = (p1 + p2) / 2;
63         radius = dist(center, p1);
64     }
65     bool crosses_x_axis(point p = point()) const {
66         auto c = center - p;
67         return GEOMETRY_COMPARE0(T, c.y + radius) >= 0 && GEOMETRY_COMPARE0(T,
68         c.y - radius) < 0;
69     }
70     static Circle incircle(const point &p1, const point &p2, const point &p3) {
71         point center = bary::incenter(p1, p2, p3).as_point();
72         return Circle(center, dist(line(p1, p2), center));
73     }
74     friend pair<segment, int> intersect_segment(const Circle &c, const line
75     &l) {
76         point H = project(c.center, l);
77         Large h = norm(H - c.center);
78         if (GEOMETRY_COMPARE(Large, c.radius, h) < 0)
79             return {{}, 0};
80
81         Large norma = sqrtl(c.radius + h) * sqrtl(c.radius - h);
82         point v = normalized(l.direction(), norma);
83         segment res = segment(H - v, H + v);
84         return {res, res.is_degenerate() ? 1 : 2};
85     }
86 }
87 friend Large intersection_area(const Circle &a, const Circle &b) {
88     Large d = norm(a.center - b.center);
89     if (GEOMETRY_COMPARE(Large, a.radius + b.radius, d) <= 0)
90         return 0.0;
91     if (GEOMETRY_COMPARE(Large, d, abs(a.radius - b.radius)) <= 0) {
92         T r = min(a.radius, b.radius);
93         return r * r * trig::PI;
94     }
95
96     auto compute = [d](Large ra, Large rb) {
97         Large sup = rb * rb + d * d - ra * ra;
98         Large alpha = trig::acos(sup / (2.0 * rb * d));
99         Large s = alpha * rb * rb;
100         Large t = rb * rb * trig::sin(alpha) * trig::cos(alpha);
101         return s - t;
102     };
103     return compute(a.radius, b.radius) + compute(b.radius, a.radius);
104 }
105 static Large intersection_signed_area(T r, const point &a, const point &b)
106 {
107     Circle C(point(), r);
108     auto ps = intersect_segment(C, line(a, b));
109     if (!ps.second)
110         return r * r * signed_angle(a, b) / 2;
111     auto s = ps.first;
112     bool outa = !contains(C, a), outb = !contains(C, b);
113     if (outa && outb) {
114         segment ab(a, b);
115         if (ab.contains(s.a) && ab.contains(s.b))
116             return (r * r * (signed_angle(a, b) - signed_angle(s.a, s.b)) +
117                     cross(s.a, s.b)) /
118                     2;
119         return r * r * signed_angle(a, b) / 2;
120     } else if (outa)
121         return (r * r * signed_angle(a, s.a) + cross(s.a, b)) / 2;
122     else if (outb)
123         return (r * r * signed_angle(s.b, b) + cross(a, s.b)) / 2;
124     else
125         return cross(a, b) / 2;
126 }
127 }
128 friend vector<point> tangents(const Circle &c, const point &p) {
129     return _tangents({p, T()}, c, {1});
130 }
131 friend vector<line> inner_tangents(const Circle& a, const Circle& b) {
132     return _tangents(a, b, {-1});
133 }
134 friend vector<line> outer_tangents(const Circle& a, const Circle& b) {
135     return _tangents(a, b, {1});
136 }
137 friend vector<line> _tangents(const Circle& a, const Circle& b, const
138     initializer_list<int>& r_sgn) {
139     vector<line> res;
140     for(int r_s : r_sgn) {
141         point d = b.center - a.center;
142         Large dr = (a.radius - b.radius*r_s), d2 = norm_sq(d), h2 = d2 - dr*dr;
143         if(GEOMETRY_COMPARE0(Large, d2) == 0) continue;
144         if(GEOMETRY_COMPARE0(Large, h2) < 0) continue;
145         for(T sgn : {-1, 1}) {
146             point v = (d * dr + ortho(d) * sqrtl(h2) * sgn) / d2;

```



```

140     res.push_back({a.center + v * a.radius, b.center + v * (b.radius *
141     r_s)});
142     }
143     if(GEOMETRY_COMPARE0(Large, h2) == 0) res.pop_back();
144     }
145     return res;
146 }
147 friend vector<Note<line, int>> angular_tangents(const Circle& a, const
148 vector<Circle>& v, vector<int>& sgn) {
149     vector<Note<line, int>> res;
150     res.reserve(4 * v.size());
151     int i = 0;
152     sgn = vector<int>(v.size());
153     vector<bool> reversed(4);
154     bool null_a = GEOMETRY_COMPARE0(T, a.radius) == 0;
155
156     for(int i = 0; i < v.size(); i++) {
157         bool null_i = GEOMETRY_COMPARE0(T, v[i].radius) == 0;
158         assert(!null_a || !null_i);
159         vector<line> tgts;
160         if(null_a || null_i) tgts = _tangents(a, v[i], {1});
161         else tgts = _tangents(a, v[i], {+1, -1});
162         if(tgts.empty()) continue;
163
164         fill(reversed.begin(), reversed.end(), false);
165         int j = 0;
166         for(auto& t : tgts) {
167             // direct tangents
168             if(ccw(t.b - t.a, a.center - t.a) < 0)
169                 swap(t.a, t.b), reversed[j] = true;
170             res.push_back(make_note<line, int>(t, i));
171             j++;
172         }
173
174         // check signal
175         auto it = AngleComparator<RayDirection<line>, T,
176         Large>::minByAngle(tgts.begin(), tgts.end());
177         point ta = reversed[it - tgts.begin()] ? it->b : it->a;
178         point dir = v[i].center - ta;
179         sgn[i] = half_ccw(it->direction(), dir);
180     }
181     AngleComparator<RayDirection<line>, T, Large>::sortByAngle(res.begin(),
182     res.end());
183     return res;
184 }
185 friend bool contains(const Circle &c, const point &p) {
186     return GEOMETRY_COMPARE(Large, dist(p, c.center), c.radius) <= 0;
187 }
188 friend bool contains(const Circle &c, const segment &s) {
189     return GEOMETRY_COMPARE(Large, dist(s.a, c.center), c.radius) <= 0 &&
190     GEOMETRY_COMPARE(Large, dist(s.b, c.center), c.radius) <= 0;
191 }
192 template <typename L>
193 friend bool partially_contains(const Circle &c, const L &l) {
194     return GEOMETRY_COMPARE(Large, dist(l, c.center), c.radius) <= 0;
195 }
196 template <typename L>
197 friend bool has_unique_intersection(const Circle &c, const L &l) {
198     return GEOMETRY_COMPARE(Large, dist(l, c.center), c.radius) == 0;
199 }
200 template <typename L>
201 friend bool has_intersection(const Circle &c, const L &l) {
202     return GEOMETRY_COMPARE(Large, dist(l, c.center), c.radius) <= 0;
203 }
204 friend bool has_intersection(const Circle &c, const segment &s) {

```

```

201     return GEOMETRY_COMPARE(Large, dist(s, c.center), c.radius) <= 0 &&
202     (GEOMETRY_COMPARE(Large, dist(s.a, c.center), c.radius) >= 0 ||
203     GEOMETRY_COMPARE(Large, dist(s.b, c.center), c.radius) >= 0);
204 }
205 };
206 } // namespace plane
207
208 template <typename T, typename Large = T>
209 struct CirclePlane : public CartesianPlane<T, Large> {
210     typedef plane::Circle<T, Large> circle;
211 };
212
213 } // namespace geo
214 } // namespace lib
215
216 #endif

```

6.3. GeometryEpsilon

```

1 #ifndef _LIB_GEOMETRY_EPSILON
2 #define _LIB_GEOMETRY_EPSILON
3 #include "../Epsilon.cpp"
4 #include <bits/stdc++.h>
5
6 #define GEOMETRY_EPSILON(T, x)
7     \
8     template <>
9     \
10    lib::Epsilon<T> *lib::geo::GeometryEpsilon<T>::eps =
11    \
12    new lib::Epsilon<T>((x));
13
14 #define GEOMETRY_COMPARE0(T, x) GeometryEpsilon<T>()((x))
15 #define GEOMETRY_COMPARE(T, x, y) GeometryEpsilon<T>()((x), (y))
16
17 namespace lib {
18     using namespace std;
19     namespace geo {
20         template <typename T> struct GeometryEpsilon {
21             static Epsilon<T> *eps;
22             template <typename G> int operator()(G a, G b = 0) const {
23                 return (*eps)(a, b);
24             }
25         };
26
27         GEOMETRY_EPSILON(int, 0);
28         GEOMETRY_EPSILON(long, 0);
29         GEOMETRY_EPSILON(long long, 0);
30     } // namespace geo
31 } // namespace lib
32
33 #endif

```

6.4. Line2D

```

1 #ifndef _LIB_GEOMETRY_LINE_2D
2 #define _LIB_GEOMETRY_LINE_2D
3 #include "GeometryEpsilon.cpp"
4 #include "Trigonometry.cpp"
5 #include <bits/stdc++.h>
6
7 namespace lib {
8     using namespace std;

```

```

9 namespace geo {
10 namespace plane {
11 namespace {
12 template <typename T> bool scalar_between(T a, T o, T b) {
13     if (a > b)
14         swap(a, b);
15     return GEOMETRY_COMPARE(T, a, o) <= 0 && GEOMETRY_COMPARE(T, o, b) <= 0;
16 }
17
18 template <typename T> bool scalar_strictly_between(T a, T o, T b) {
19     if (a > b)
20         swap(a, b);
21     int x = GEOMETRY_COMPARE(T, a, o);
22     int y = GEOMETRY_COMPARE(T, o, b);
23     return x <= 0 && y <= 0 && (x < 0 || y < 0);
24 }
25 } // namespace
26
27 template <typename T, typename Large = T> struct Point {
28     T x, y;
29     Point() : x(0), y(0) {}
30     Point(T x, T y) : x(x), y(y) {}
31     template <typename G, typename H> explicit operator Point<G, H>() const {
32         return Point<G, H>((G)x, (G)y);
33     }
34     friend Point reversed(const Point &a) { return Point(a.y, a.x); }
35     Point &operator+=(const Point &rhs) {
36         x += rhs.x, y += rhs.y;
37         return *this;
38     }
39     Point &operator-=(const Point &rhs) {
40         x -= rhs.x, y -= rhs.y;
41         return *this;
42     }
43     Point &operator*=(T k) {
44         x *= k, y *= k;
45         return *this;
46     }
47     Point &operator/=(T k) {
48         x /= k, y /= k;
49         return *this;
50     }
51     Point operator+(const Point &rhs) const {
52         Point res = *this;
53         return res += rhs;
54     }
55     Point operator-(const Point &rhs) const {
56         Point res = *this;
57         return res -= rhs;
58     }
59     Point operator*(T k) const {
60         Point res = *this;
61         return res *= k;
62     }
63     Point operator/(T k) const {
64         Point res = *this;
65         return res /= k;
66     }
67     Point operator-() const { return Point(-x, -y); }
68     inline friend Point convolve(const Point &a, const Point &b) {
69         return Point(a.x * b.x - a.y * b.y, a.x * b.y + b.x * a.y);
70     }
71     inline friend Large cross(const Point &a, const Point &b) {
72         return (Large)a.x * b.y - (Large)a.y * b.x;
73     }

```

```

74     friend Large cross(const Point &a, const Point &b, const Point &c) {
75         return cross(b - a, c - a);
76     }
77     inline friend Large dot(const Point &a, const Point &b) {
78         return (Large)a.x * b.x + (Large)a.y * b.y;
79     }
80     friend int ccw(const Point &u, const Point &v) {
81         return GEOMETRY_COMPARE0(Large, cross(u, v));
82     }
83     friend int ccw(const Point &a, const Point &b, const Point &c) {
84         return ccw(b - a, c - a);
85     }
86     friend int half_ccw(const Point &u, const Point &v) {
87         int dot_sgn = GEOMETRY_COMPARE0(Large, dot(u, v));
88         int ccw_sgn = ccw(u, v);
89         if (dot_sgn == 0) return ccw_sgn ? 1 : 0;
90         return dot_sgn * ccw_sgn;
91     }
92     friend Large norm(const Point &a) { return sqrtl(dot(a, a)); }
93     friend Large norm_sq(const Point &a) { return dot(a, a); }
94     bool is_null() const { return GEOMETRY_COMPARE0(Large, norm_sq(*this)) == 0; }
95     bool is_versor() const {
96         return GEOMETRY_COMPARE(Large, norm_sq(*this), (Large)1) == 0;
97     }
98     static Point polar(Large d, Large theta) {
99         return Point(trig::cos(theta) * d, trig::sin(theta) * d);
100     }
101     friend Point rotate(const Point &a, Large theta) {
102         return convolve(a, polar((Large)1, theta));
103     }
104     friend Point ortho(const Point &a) { return Point(-a.y, a.x); }
105     friend Large arg(const Point &a) { return trig::atan2(a.y, a.x); }
106     friend Large signed_angle(const Point &v, const Point &w) {
107         return remainder(arg(w) - arg(v), 2.0 * trig::PI);
108     }
109     friend Large angle(const Point &v, const Point &w) {
110         return abs(signed_angle(v, w));
111     }
112     friend Large ccw_angle(const Point &v) {
113         Large res = arg(v);
114         if (res < 0)
115             res += 2.0 * trig::PI;
116         return res;
117     }
118     friend Large ccw_angle(const Point &v, const Point &w) {
119         Large res = signed_angle(v, w);
120         if (res < 0)
121             res += 2.0 * trig::PI;
122         return res;
123     }
124     inline friend Point normalized(const Point &a, Large k) {
125         return a.is_null() ? Point() : a / norm(a) * k;
126     }
127     inline friend Point versor(const Point &a) { return normalized(a, (Large)1); }
128     friend bool collinear(const Point &a, const Point &b) {
129         return GEOMETRY_COMPARE0(Large, cross(a, b)) == 0;
130     }
131     friend bool collinear(const Point &a, const Point &b, const Point &c) {
132         return collinear(b - a, c - a);
133     }
134     friend Point project(const Point &a, const Point &v) {
135         return v / norm_sq(v) * dot(a, v);
136     }

```

```

137 template <typename G = T,
138           typename enable_if<!is_integral<G>::value>::type * = nullptr>
139 friend Point reflect(const Point &a, const Point &v) {
140     Point n = versor(v);
141     return a - n * 2 * dot(n, v);
142 }
143 friend bool between(const Point &a, const Point &b, const Point &c) {
144     return collinear(a, b, c) &&
145         GEOMETRY_COMPARE0(Large, dot(a - b, c - b)) <= 0;
146 }
147 friend bool strictly_between(const Point &a, const Point &b, const Point
148                             &c) {
149     return collinear(a, b, c) &&
150         GEOMETRY_COMPARE0(Large, dot(a - b, c - b)) < 0;
151 }
152 friend bool collinear_between(const Point a, const Point &o, const Point
153                             &b) {
154     return scalar_between(a.x, o.x, b.x) && scalar_between(a.y, o.y, b.y);
155 }
156 friend bool collinear_strictly_between(const Point &a, const Point &o,
157                                       const Point &b) {
158     return scalar_between(a.x, o.x, b.x) && scalar_between(a.y, o.y, b.y);
159 }
160 friend Large dist(const Point &a, const Point &b) { return norm(a - b); }
161 friend bool operator==(const Point &a, const Point &b) {
162     return GEOMETRY_COMPARE(T, a.x, b.x) == 0 &&
163         GEOMETRY_COMPARE(T, a.y, b.y) == 0;
164 }
165 friend bool operator!=(const Point &a, const Point &b) { return !(a == b); }
166 }
167 friend bool operator<(const Point &a, const Point &b) {
168     return tie(a.y, a.x) < tie(b.y, b.x);
169 }
170 friend bool operator>(const Point &a, const Point &b) {
171     return tie(a.y, a.x) > tie(b.y, b.x);
172 }
173 friend bool operator>=(const Point &a, const Point &b) {
174     return tie(a.y, a.x) >= tie(b.y, b.x);
175 }
176 friend bool operator<=(const Point &a, const Point &b) {
177     return tie(a.y, a.x) <= tie(b.y, b.x);
178 }
179 friend istream &operator>>(istream &in, Point &p) { return in >> p.x >>
180 p.y; }
181 friend ostream &operator<<(ostream &out, const Point &p) {
182     return out << p.x << " " << p.y;
183 }
184 };
185
186 template <typename T, typename Large = T> struct Rectangle {
187     typedef Point<T, Large> point;
188
189     T minx, miny, maxx, maxy;
190     Rectangle() {
191         minx = miny = numeric_limits<T>::max();
192         maxx = maxy = numeric_limits<T>::min();
193     }
194     Rectangle(const initializer_list<point> &points) : Rectangle() {
195         for (const auto &p : points) {
196             minx = min(minx, p.x);
197             maxx = max(maxx, p.x);
198             miny = min(miny, p.y);
199             maxy = max(maxy, p.y);
200         }
201     }
202 };

```

```

203 }
204
205 bool contains(const point &p) const {
206     return GEOMETRY_COMPARE(T, minx, p.x) <= 0 &&
207         GEOMETRY_COMPARE(T, p.x, maxx) <= 0 &&
208         GEOMETRY_COMPARE(T, miny, p.y) <= 0 &&
209         GEOMETRY_COMPARE(T, p.y, maxy) <= 0;
210 }
211 };
212
213 template <typename T, typename Large = T> struct Line {
214     typedef Point<T, Large> point;
215     typedef Line<T, Large> line;
216     point a, b;
217     Line(point a, point b) : a(a), b(b) {}
218     template <typename G = T,
219             typename enable_if<!is_integral<G>::value>::type * = nullptr>
220     Line(T A, T B, T C) {
221         if (GEOMETRY_COMPARE0(Large, A))
222             a = point(-C / A, 0), b = point((-C - B) / A, 1);
223         else if (GEOMETRY_COMPARE0(Large, B))
224             a = point(0, -C / B), b = point(1, (-C - A) / B);
225         else
226             assert(false);
227     }
228     template <typename G, typename H> explicit operator Line<G, H>() const {
229         return Line<G, H>(Point<G, H>(a), Point<G, H>(b));
230     }
231     point direction() const { return b - a; }
232     friend point project(const point &p, const line &v) {
233         return project(p - v.a, v.b - v.a) + v.a;
234     }
235     friend bool collinear(const line &u, const line &v) {
236         return collinear(u.a, u.b, v.a) && collinear(u.a, u.b, v.b);
237     }
238     bool contains(const point &p) const { return collinear(a, b, p); }
239     friend bool parallel(const line &u, const line &v) {
240         return collinear(u.b - u.a, v.b - v.a);
241     }
242     friend bool opposite(const line &l1, const point &p1, const point &p2) {
243         int x = GEOMETRY_COMPARE0(Large, cross(p1 - l1.a, l1.direction()));
244         int y = GEOMETRY_COMPARE0(Large, cross(p2 - l1.a, l1.direction()));
245         return x * y <= 0;
246     }
247     friend pair<point, bool> intersect(const line &l1, const line &l2) {
248         Large c1 = cross(l2.a - l1.a, l1.b - l1.a);
249         Large c2 = cross(l2.b - l1.a, l1.b - l1.a);
250         if (GEOMETRY_COMPARE0(Large, c1 - c2) == 0)
251             return {{}, false};
252         return {(l2.b * c1 - l2.a * c2) / (c1 - c2), true};
253     }
254     friend bool has_unique_intersection(const line &l1, const line &l2) {
255         return !parallel(l1, l2);
256     }
257     friend bool has_intersection(const line &l1, const line &l2) {
258         return collinear(l1, l2) || has_unique_intersection(l1, l2);
259     }
260     friend Large dist(const line &l1, const point &p) {
261         // TODO: improve this
262         return dist(p, project(p, l1));
263     }
264     friend Large dist(const line &l1, const line &l2) {
265         if (has_intersection(l1, l2))
266             return 0;
267         // TODO: improve this
268     }
269 };

```

```

263     return dist(l1.a, project(l1.a, l2));
264 }
265 };
266
267 template <typename T, typename Large = T> struct Ray {
268     typedef Point<T, Large> point;
269     typedef Line<T, Large> line;
270     typedef Ray<T, Large> ray;
271     point a, b;
272
273     Ray(point a, point direction) : a(a), b(a + direction) {}
274
275     static ray from_points(point a, point b) { return ray(a, b - a); }
276     point direction() const { return b - a; }
277     point direction_versor() const { return versor(direction()); }
278
279     line as_line() const { return line(a, b); }
280     explicit operator line() const { return as_line(); }
281
282     template <typename G, typename H> explicit operator Ray<G, H>() const {
283         return Ray<G, H>(Point<G, H>(a), Point<G, H>(b));
284     }
285     bool contains(const point &p) const {
286         return collinear(a, b, p) &&
287             GEOMETRY_COMPARE0(Large, dot(p - a, b - a)) >= 0;
288     }
289     bool strictly_contains(const point &p) const {
290         return collinear(a, b, p) &&
291             GEOMETRY_COMPARE0(Large, dot(p - a, b - a)) > 0;
292     }
293     bool collinear_contains(const point &p) const {
294         point dir = direction();
295         int dx = GEOMETRY_COMPARE0(T, dir.x);
296         if (dx == 0)
297             return GEOMETRY_COMPARE0(T, dir.y) * GEOMETRY_COMPARE0(T, p.y - a.y)
298             >= 0;
299         else
300             return dx * GEOMETRY_COMPARE0(T, p.x - a.x) >= 0;
301     }
302     bool collinear_strictly_contains(const point &p) const {
303         point dir = direction();
304         int dx = GEOMETRY_COMPARE0(T, dir.x);
305         if (dx == 0)
306             return GEOMETRY_COMPARE0(T, dir.y) * GEOMETRY_COMPARE0(T, p.y - a.y) >
307             0;
308         else
309             return dx * GEOMETRY_COMPARE0(T, p.x - a.x) > 0;
310     }
311     friend pair<point, bool> intersect(const ray &r, const line &l) {
312         auto p = intersect(r.as_line(), l);
313         if (!p.second)
314             return {{}, false};
315         if (!r.collinear_contains(p.first))
316             return {{}, false};
317         return p;
318     }
319     friend pair<point, bool> intersect(const ray &a, const ray &b) {
320         auto p = intersect(a.as_line(), b.as_line());
321         if (!p.second)
322             return {{}, false};
323         if (!b.collinear_contains(p.first))
324             return {{}, false};
325         return p;
326     }
327     friend bool has_unique_intersection(const ray &r, const line &l) {

```

```

326     if (!has_unique_intersection(r.as_line(), l))
327         return false;
328     int x = GEOMETRY_COMPARE0(Large, cross(r.direction(), l.direction()));
329     int y = GEOMETRY_COMPARE0(Large, cross(r.a - l.a, l.direction()));
330     return x * y <= 0;
331 }
332 friend bool has_intersection(const ray &r, const line &l) {
333     return collinear(r.as_line(), l) || has_unique_intersection(r, l);
334 }
335 friend bool has_unique_intersection(const ray &r1, const ray &r2) {
336     // TODO: not efficient
337     return has_unique_intersection(r1, r2.as_line()) &&
338         has_unique_intersection(r2, r1.as_line());
339 }
340 friend bool has_intersection(const ray &r1, const ray &r2) {
341     return r1.contains(r2.a) || has_unique_intersection(r1, r2);
342 }
343 friend Large dist(const ray &r, const point &p) {
344     if (GEOMETRY_COMPARE0(Large, dot(r.direction(), p - r.a)) < 0)
345         return dist(p, r.a);
346     return dist(r.as_line(), p);
347 }
348 friend Large dist(const ray &r, const line &l) {
349     if (has_intersection(r, l))
350         return Large(0);
351     return dist(l, r.a);
352 }
353 friend Large dist(const ray &r1, const ray &r2) {
354     if (has_intersection(r1, r2))
355         return Large(0);
356     return min(dist(r1, r2.a), dist(r2, r1.a));
357 }
358 };
359
360 template <typename T, typename Large = T> struct Halfplane {
361     typedef Point<T, Large> point;
362     typedef Line<T, Large> line;
363     typedef Ray<T, Large> ray;
364     typedef Halfplane<T, Large> halfplane;
365     point a, b;
366
367     Halfplane(point a, point direction) : a(a), b(a + direction) {}
368
369     static halfplane from_points(point a, point b) { return halfplane(a, b -
370         a); }
371     point direction() const { return b - a; }
372     point direction_versor() const { return versor(direction()); }
373
374     line as_line() const { return line(a, b); }
375     explicit operator line() const { return as_line(); }
376
377     ray as_ray() const { return ray(a, b); }
378     explicit operator ray() const { return as_ray(); }
379
380     template <typename G, typename H> explicit operator Halfplane<G, H>()
381     const {
382         return Halfplane<G, H>(Point<G, H>(a), Point<G, H>(b));
383     }
384
385     bool contains(const point& p) const {
386         return ccw(a, b, p) <= 0;
387     }
388     bool strictly_contains(const point& p) const {
389         return ccw(a, b, p) < 0;
390     }

```

```

389 };
390
391 template <typename T, typename Large = T> struct Segment {
392     typedef Point<T, Large> point;
393     typedef Line<T, Large> line;
394     typedef Segment<T, Large> segment;
395     typedef Ray<T, Large> ray;
396     point a, b;
397
398     Segment() {}
399     Segment(point a, point b) : a(a), b(b) {}
400     line as_line() const { return line(a, b); }
401     explicit operator line() const { return as_line(); }
402     bool is_degenerate() const { return a == b; }
403
404     template <typename G, typename H> explicit operator Segment<G, H>() const {
405         return Segment<G, H>(Point<G, H>(a), Point<G, H>(b));
406     }
407     bool contains(const point &p) const { return between(a, p, b); }
408     bool strictly_contains(const point &p) const {
409         return strictly_between(a, p, b);
410     }
411     bool collinear_contains(const point &p) const {
412         return collinear_between(a, p, b);
413     }
414     bool collinear_strictly_contains(const point &p) const {
415         return collinear_strictly_between(a, p, b);
416     }
417     friend pair<point, bool> intersect(const segment &s, const line &l) {
418         auto p = intersect(s.as_line(), l);
419         if (!p.second)
420             return {{}, false};
421         if (!s.collinear_contains(p.first))
422             return {{}, false};
423         return p;
424     }
425     friend pair<point, bool> intersect(const segment &s, const ray &r) {
426         auto p = intersect(s.as_line(), r.as_line());
427         if (!p.second)
428             return {{}, false};
429         if (!s.collinear_contains(p.first) || !r.collinear_contains(p.first))
430             return {{}, false};
431         return p;
432     }
433     friend pair<segment, int> intersect_segment(segment s1, segment s2) {
434         if (collinear(s1.as_line(), s2.as_line())) {
435             if (s1.a > s1.b)
436                 swap(s1.a, s1.b);
437             if (s2.a > s2.b)
438                 swap(s2.a, s2.b);
439             segment res(max(s1.a, s2.a), min(s1.b, s2.b));
440             return {res, int(res.a <= res.b) * 2};
441         } else {
442             auto p = intersect(s1, s2);
443             return {segment(p.first, p.first), p.second};
444         }
445     }
446     friend pair<point, bool> intersect(const segment &s1, const segment &s2) {
447         auto p = intersect(s1, s2.as_line());
448         if (!p.second)
449             return {{}, false};
450         if (!s2.collinear_contains(p.first))
451             return {{}, false};
452         return p;
453     }

```

```

454     friend bool has_unique_intersection(const segment &s, const line &l) {
455         if (!has_unique_intersection(s.as_line(), l))
456             return false;
457         return opposite(l, s.a, s.b);
458     }
459     friend bool has_intersection(const segment &s, const line &l) {
460         return collinear(s.as_line(), l) || has_unique_intersection(s, l);
461     }
462     friend bool has_unique_intersection(const segment &s, const ray &r) {
463         if (!has_unique_intersection(r, s.as_line()))
464             return false;
465         return opposite(r.as_line(), s.a, s.b);
466     }
467     friend bool has_intersection(const segment &s, const ray &r) {
468         return r.contains(s.a) || r.contains(s.b) || has_unique_intersection(s, r);
469     }
470     friend bool has_unique_intersection(const segment &s1, const segment &s2) {
471         if (!has_unique_intersection(s1.as_line(), s2.as_line()))
472             return false;
473         return opposite(s2.as_line(), s1.a, s1.b) &&
474             opposite(s1.as_line(), s2.a, s2.b);
475     }
476     friend bool has_intersection(const segment &s1, const segment &s2) {
477         return s1.contains(s2.a) || s1.contains(s2.b) ||
478             has_unique_intersection(s1, s2);
479     }
480     friend Large dist(const segment &s, const point &p) {
481         if (GEOMETRY_COMPARE0(Large, dot(p - s.a, s.b - s.a)) <= 0)
482             return dist(s.a, p);
483         if (GEOMETRY_COMPARE0(Large, dot(p - s.b, s.a - s.b)) <= 0)
484             return dist(s.b, p);
485         return dist(s.as_line(), p);
486     }
487     friend Large dist(const segment &s, const line &l) {
488         if (has_intersection(s, l))
489             return Large(0);
490         return min(dist(l, s.a), dist(l, s.b));
491     }
492     friend Large dist(const segment &s, const ray &r) {
493         if (has_intersection(s, r))
494             return Large(0);
495         return min({dist(r, s.a), dist(r, s.b), dist(s, r.a)});
496     }
497     friend Large dist(const segment &s1, const segment &s2) {
498         if (has_intersection(s1, s2))
499             return Large(0);
500         return min(
501             {dist(s1, s2.a), dist(s1, s2.b), dist(s2, s1.a), dist(s2, s1.b)});
502     }
503
504     friend bool operator==(const segment &l1, const segment &l2) {
505         return tie(l1.a, l1.b) == tie(l2.a, l2.b);
506     }
507     friend bool operator!=(const segment &l1, const segment &l2) {
508         return !(l1 == l2);
509     }
510     friend bool operator<(const segment &l1, const segment &l2) {
511         return tie(l1.a, l1.b) < tie(l2.a, l2.b);
512     }
513 };
514
515 template <typename Direction, typename T, typename Large> struct
516     AngleComparator {
517     using type = typename Direction::type;

```

```

517 using point = Point<T, Large>;
518
519 Direction dir;
520 AngleComparator() {}
521 AngleComparator(Direction dir) : dir(dir) {}
522 bool operator()(const type &a, const type &b) const {
523     return ccw(dir(a), dir(b)) > 0;
524 }
525 template <typename Iterator>
526 static void sortByAngle(Iterator begin, Iterator end, const Direction& dir
    = Direction()) {
527     AngleComparator cmp(dir);
528     begin =
529         partition(begin, end, [&dir](const type &p) { return
            dir(p).is_null(); });
530     auto half =
531         partition(begin, end, [&dir](const type &p) { return dir(p) >
            point(); });
532     sort(begin, half, cmp);
533     sort(half, end, cmp);
534 }
535 template <typename Iterator>
536 static Iterator minByAngle(Iterator begin, Iterator end, const Direction&
    dir = Direction()) {
537     AngleComparator cmp(dir);
538     return min_element(begin, end, [&dir, &cmp](const type& a, const type&
        b) {
539         bool part_a = dir(a) > point();
540         bool part_b = dir(b) > point();
541         if(part_a == part_b)
542             return cmp(a, b);
543         return part_a > part_b;
544     });
545 }
546 };
547 template <typename Ray> struct RayDirection {
548     using point = typename Ray::point;
549     using type = Ray;
550     point operator()(const type& rhs) const {
551         return rhs.direction();
552     }
553 };
554 template <typename Point> struct PointDirection {
555     using type = Point;
556     Point pivot;
557     PointDirection() : pivot() {}
558     PointDirection(Point pivot) : pivot(pivot) {}
559     Point operator()(const Point& rhs) const {
560         return (rhs - pivot).direction();
561     }
562 };
563 } // namespace plane
564
565 template <typename T, typename Large = T> struct CartesianPlane {
566     typedef plane::Point<T, Large> point;
567     typedef plane::Line<T, Large> line;
568     typedef plane::Rectangle<T, Large> rectangle;
569     typedef plane::Segment<T, Large> segment;
570     typedef plane::Ray<T, Large> ray;
571     typedef plane::Halfplane<T, Large> halfplane;
572
573     template<typename Direction>
574     using angle_comparator = plane::AngleComparator<Direction, T, Large>;
575 };

```

```

577 } // namespace geo
578 } // namespace lib
579
580 #endif

```

6.5. Polygon2D

```

1  #ifndef _LIB_GEOMETRY_POLY_2D
2  #define _LIB_GEOMETRY_POLY_2D
3  #include "Circle2D.cpp"
4  #include "Line2D.cpp"
5  #include <bits/stdc++.h>
6
7  namespace lib {
8  using namespace std;
9  namespace geo {
10     namespace plane {
11
12     template <typename T, typename Large = T> struct ConvexHullComparator {
13         typedef Point<T, Large> point;
14         point pivot;
15         ConvexHullComparator(point p) : pivot(p) {}
16         template <typename G>
17         bool operator()(const pair<point, G> &a, const pair<point, G> &b) const {
18             int k = ccw(pivot, a.first, b.first);
19             if (k == 0)
20                 return norm_sq(a.first) < norm_sq(b.first);
21             return k > 0;
22         }
23     };
24
25     template <typename T, typename Large = T> struct Polygon {
26         typedef Point<T, Large> point;
27         typedef Polygon<T, Large> polygon;
28         typedef Circle<T, Large> circle;
29         vector<point> p;
30
31         Polygon() {}
32         Polygon(const vector<point> &p) : p(p) {}
33         template <typename G> Polygon(const vector<pair<point, G>> &g) :
            p(g.size()) {
34             for (size_t i = 0; i < g.size(); i++)
35                 p[i] = g[i].first;
36         }
37         template <typename A, typename B> explicit operator Polygon<A, B>() const {
38             vector<Point<A, B>> v(p.size());
39             for (size_t i = 0; i < p.size(); i++)
40                 v[i] = Point<A, B>(p[i]);
41             return Polygon<A, B>(v);
42         }
43         inline int index(int i) const {
44             if (i >= size())
45                 i %= size();
46             else if (i < 0) {
47                 i %= size();
48                 if (i < 0)
49                     i += size();
50             }
51             return i;
52         }
53         inline int size() const { return p.size(); }
54         inline point &operator[](int i) { return p[index(i)]; }
55         inline point operator[](int i) const { return p[index(i)]; }
56         void erase(int i) { p.erase(p.begin() + index(i)); }

```

```

57 polygon &operator+=(const point &pt) {
58     for (auto &q : p)
59         q += pt;
60     return *this;
61 }
62 polygon &operator-=(const point &pt) {
63     for (auto &q : p)
64         q -= pt;
65     return *this;
66 }
67 polygon &operator*=(const Large k) {
68     for (auto &q : p)
69         q *= k;
70     return *this;
71 }
72 polygon &operator/=(const Large k) {
73     for (auto &q : p)
74         q /= k;
75     return *this;
76 }
77 polygon operator-() const {
78     polygon res = *this;
79     for (auto &q : res.p)
80         q = -q;
81     return res;
82 }
83 void reserve(int n) { p.reserve(n); }
84 bool is_ccw() const {
85     int n = size();
86     int i = min_element(p.begin(), p.end()) - p.begin();
87     return ccw(p[i], p[i + 1], p[i - 1]) >= 0;
88 }
89 bool is_degenerate() const {
90     int n = size();
91     if (n < 3)
92         return true;
93     for (int i = 0; i < n; i++) {
94         if (GEOMETRY_COMPARE0(Large, cross(p[i + 2] - p[i], p[i + 1] - p[i])))
95             return false;
96     }
97     return true;
98 }
99 inline operator vector<point>() const { return p; }
100
101 friend Large double_area(const Polygon &p) {
102     int n = p.size();
103     Large res = 0;
104     for (int i = 0; i < n; i++) {
105         res += cross(p[i], p[i + 1]);
106     }
107     return abs(res);
108 }
109 friend Large area(const Polygon &p) { return double_area(p) / 2; }
110 friend Large perimeter(const Polygon &p) {
111     int n = p.size();
112     Large res = 0;
113     for (int i = 0; i < n; i++)
114         res += dist(p[i], p[i + 1]);
115     return res;
116 }
117
118 int test(const point &p) const {
119     const Polygon &poly = *this;
120     int n = size();
121     int wn = 0;

```

```

122 for (int i = 0; i < n; i++) {
123     if (p == poly[i])
124         return 0;
125     int j = i + 1;
126     if (poly[i].y == p.y && poly[j].y == p.y) {
127         if (min(poly[i].x, poly[j].x) <= p.x &&
128             p.x <= max(poly[i].x, poly[j].x))
129             return 0;
130     } else {
131         bool below = poly[i].y < p.y;
132         if (below != (poly[j].y < p.y)) {
133             auto sig = ccw(poly[i], poly[j], p);
134             if (sig == 0)
135                 return 0;
136             if (below == (sig > 0))
137                 wn += below ? 1 : -1;
138         }
139     }
140 }
141 return wn == 0 ? 1 : -1;
142 }
143
144 template <typename G>
145 static vector<pair<point, G>> convex_hull(vector<pair<point, G>> p,
146                                           bool keep_border = false) {
147     if (p.size() <= 1)
148         return p;
149     sort(p.begin(), p.end());
150     vector<pair<point, G>> res;
151     res.reserve(p.size() + 1);
152     for (int step = 0; step < 2; step++) {
153         auto start = res.size();
154         for (auto &q : p) {
155             while (res.size() >= start + 2) {
156                 int sig = ccw(res[res.size() - 2].first, res.back().first,
157                             q.first);
158                 if ((sig == 0 && !keep_border) || sig < 0)
159                     res.pop_back();
160                 else
161                     break;
162             }
163             res.push_back(q);
164         }
165         res.pop_back();
166         if (step == 0)
167             reverse(p.begin(), p.end());
168     }
169     if (res.size() == 2 && res[0] == res[1])
170         res.pop_back();
171     return res;
172 }
173
174 static polygon convex_hull(const vector<point> &p, bool keep_border =
175                             false) {
176     vector<pair<point, int>> v(p.size());
177     for (size_t i = 0; i < p.size(); i++)
178         v[i] = {p[i], i};
179     auto res = convex_hull(v, keep_border);
180     return polygon(res);
181 }
182
183 friend vector<polygon> triangulation(polygon poly) {
184     if (poly.size() < 3)
185         return {};
186     vector<polygon> res;

```

```

185     int ptr = 0;
186     int n;
187     while ((n = poly.size()) > 3) {
188         for (int &i = ptr; i++) {
189             if (ccw(poly[i - 1], poly[i], poly[i + 1]) > 0) {
190                 auto trig = polygon({poly[i - 1], poly[i], poly[i + 1]});
191                 bool good = true;
192                 for (int j = 0; j < n; j++) {
193                     good &= trig.test(poly[j]) >= 0;
194                 }
195                 if (!good)
196                     continue;
197                 poly.erase(i--);
198                 res.push_back(trig);
199                 break;
200             }
201         }
202     }
203     res.push_back(poly);
204     return res;
205 }
206
207 friend Large intersection_area(const Polygon &p, const circle &C) {
208     Large res = 0;
209     int n = p.size();
210     for (int i = 0; i < n; i++) {
211         res += circle::intersection_signed_area(C.radius, p[i + 1] - C.center,
212         p[i] - C.center);
213     }
214     return abs(res);
215 }
216 };
217
218 template <typename T, typename Large = T>
219 struct ConvexPolygon : public Polygon<T, Large> {
220     typedef Point<T, Large> point;
221     typedef Segment<T, Large> segment;
222     typedef Line<T, Large> line;
223     typedef Halfplane<T, Large> halfplane;
224     typedef Circle<T, Large> circle;
225     typedef AngleComparator<PointDirection<point>, T, Large> angle_comparator;
226     using Polygon<T, Large>::p;
227     int top;
228     ConvexPolygon() {}
229     ConvexPolygon(const vector<point> &p) : Polygon<T, Large>(p) {
230         normalize();
231     }
232     template <typename G>
233     ConvexPolygon(const vector<pair<point, G>> &p) : Polygon<T, Large>(p) {
234         normalize();
235     }
236     void normalize() {
237         auto bottom = min_element(p.begin(), p.end());
238         rotate(p.begin(), bottom, p.end());
239         top = max_element(p.begin(), p.end()) - p.begin();
240     }
241     ConvexPolygon &operator+=(const point &pt) {
242         for (auto &q : p)
243             q += pt;
244         return *this;
245     }
246     ConvexPolygon &operator-=(const point &pt) {
247         for (auto &q : p)
248             q -= pt;
249         return *this;
250     }
251 }

```

```

249 ConvexPolygon &operator*=(const Large k) {
250     for (auto &q : p)
251         q *= k;
252     return *this;
253 }
254 ConvexPolygon &operator/=(const Large k) {
255     for (auto &q : p)
256         q /= k;
257     return *this;
258 }
259 ConvexPolygon operator-() const {
260     ConvexPolygon res = *this;
261     for (auto &q : res.p)
262         q = -q;
263     return res;
264 }
265
266 int test(const point &q) const {
267     if (q < p[0] || q > p[top])
268         return 1;
269     auto sig = ccw(p[0], p[top], q);
270     if (sig == 0) {
271         if (q == p[0] || q == p[top])
272             return 0;
273         return top == 1 || top + 1 == this->size() ? 0 : -1;
274     } else if (sig < 0) {
275         auto it = lower_bound(p.begin() + 1, p.begin() + top, q);
276         return ccw(it[-1], q, it[0]);
277     } else {
278         auto it = upper_bound(p.rbegin(), p.rend() - top - 1, q);
279         auto pit_deref = it == p.rbegin() ? p[0] : it[-1];
280         return ccw(*it, q, pit_deref);
281     }
282 }
283
284 template <typename Function> int extreme(Function direction) const {
285     int n = this->size(), left = 0, leftSig;
286     const ConvexPolygon &poly = *this;
287     auto vertex_cmp = [&poly, direction](int i, int j) {
288         return ccw(poly[j] - poly[i], direction(poly[j]));
289     };
290     auto is_extreme = [n, vertex_cmp](int i, int &iSig) {
291         return (iSig = vertex_cmp(i + 1, i)) >= 0 && vertex_cmp(i, i - 1) < 0;
292     };
293     for (int right = is_extreme(0, leftSig) ? 1 : n; left + 1 < right; ) {
294         int mid = (left + right) / 2, midSig;
295         if (is_extreme(mid, midSig))
296             return mid;
297         if (leftSig != midSig ? leftSig < midSig
298             : leftSig == vertex_cmp(left, mid))
299             right = mid;
300         else
301             left = mid, leftSig = midSig;
302     }
303     return poly.index(left);
304 }
305 void stab_extremes(const line &l, int &left, int &right) const {
306     point direction = l.direction();
307     right = extreme([&direction](const point &) { return direction; });
308     left = extreme([&direction](const point &) { return -direction; });
309 }
310 friend vector<point> intersect(const ConvexPolygon &poly, const line &l) {
311     point direction = l.direction();
312     int left, right;
313     poly.stab_extremes(l, left, right);

```



```

314 auto vertex_cmp = [&l, &direction](const point &q) {
315     return ccw(q - l.a, direction);
316 };
317 int rightSig = vertex_cmp(poly[right]), leftSig = vertex_cmp(poly[left]);
318 if (rightSig < 0 || leftSig > 0)
319     return {};
320 auto intersectChain = [&l, &poly, vertex_cmp](int first, int last,
321     int firstSig) {
322     int n = poly.size();
323     while (poly.index(first + 1) != poly.index(last)) {
324         int mid = (first + last + (first < last ? 0 : n)) / 2;
325         mid = poly.index(mid);
326         if (vertex_cmp(poly[mid]) == firstSig)
327             first = mid;
328         else
329             last = mid;
330     }
331     return intersect(l, line(poly[first], poly[last]));
332 };
333 return {intersectChain(left, right, leftSig).first,
334     intersectChain(right, left, rightSig).first};
335 }
336 friend bool has_intersection(const ConvexPolygon &p, const line &l) {
337     point direction = l.direction();
338     int left, right;
339     p.stab_extremes(l, left, right);
340     auto vertex_cmp = [&l, &direction](const point &q) {
341         return ccw(q - l.a, direction);
342     };
343     int rightSig = vertex_cmp(p[right]), leftSig = vertex_cmp(p[left]);
344     if (rightSig < 0 || leftSig > 0)
345         return false;
346     return true;
347 }
348 friend Large dist(const ConvexPolygon &p, const line &l) {
349     point direction = l.direction();
350     int left, right;
351     p.stab_extremes(l, left, right);
352     auto vertex_cmp = [&l, &direction](const point &q) {
353         return ccw(q - l.a, direction);
354     };
355     int rightSig = vertex_cmp(p[right]), leftSig = vertex_cmp(p[left]);
356     if (rightSig < 0 || leftSig > 0) {
357         return min(dist(l, p[right]), dist(l, p[left]));
358     } else {
359         return 0;
360     }
361 }
362 template <typename Function>
363 friend void antipodals(const ConvexPolygon &poly, Function f) {
364     if (poly.size() <= 1)
365         return;
366     if (poly.size() == 2)
367         return void(f(0, 1));
368     auto area = [&poly](int i, int j, int k) {
369         return abs(cross(poly[i], poly[j], poly[k]));
370     };
371     auto func = [f, &poly](int i, int j) {
372         return f(poly.index(i), poly.index(j));
373     };
374
375     int p = -1;
376     int q = 0;
377     while (area(p, p + 1, q + 1) > area(p, p + 1, q))
378         q++;

```

```

379 int p0 = 0;
380 int q0 = q;
381 while (poly.index(q) != p0) {
382     p++;
383     func(p, q);
384     while (area(p, p + 1, q + 1) > area(p, p + 1, q)) {
385         q++;
386         if (poly.index(p) != poly.index(q0) || poly.index(q) != p0)
387             func(p, q);
388         else
389             return;
390     }
391     if (area(p, p + 1, q + 1) == area(p, p + 1, q)) {
392         if (poly.index(p) != poly.index(q0) || poly.index(q) != p0)
393             func(p, q + 1);
394         else
395             func(p + 1, q);
396     }
397 }
398 }
399 friend ConvexPolygon minkowski_sum(const vector<ConvexPolygon> &v) {
400     vector<point> vectors;
401     point origin;
402     for (auto &poly : v) {
403         origin += poly[0];
404         for (int i = 0; i < poly.size(); i++)
405             vectors.push_back(poly[i + 1] - poly[i]);
406     }
407     angle_comparator::sortByAngle(vectors.begin(), vectors.end());
408     auto last = point();
409     if (!vectors.empty()) {
410         last = vectors.back();
411         vectors.pop_back();
412     }
413     vector<point> res;
414     res.push_back(origin);
415     for (auto &v : vectors) {
416         res.push_back(res.back() + v);
417         int n = res.size();
418         if (n >= 3 && collinear(res[n - 3], res[n - 2], res[n - 1]))
419             res.erase(res.begin() + n - 2);
420     }
421     int n = res.size();
422     if (n >= 3 && collinear(res[n - 2], res[n - 1], res[0]))
423         res.pop_back();
424     if (res.size() >= 3 && collinear(res.back(), res[0], res[1]))
425         res.erase(res.begin());
426     return ConvexPolygon(res);
427 }
428 friend ConvexPolygon minkowski_sum(const ConvexPolygon &a,
429     const ConvexPolygon &b) {
430     vector<ConvexPolygon> v;
431     v.push_back(a);
432     v.push_back(b);
433     return minkowski_sum(v);
434 }
435 friend ConvexPolygon intersect(const ConvexPolygon &a,
436     const ConvexPolygon &b) {
437     vector<point> candidates;
438     auto consider = [&candidates](const ConvexPolygon &a,
439         const ConvexPolygon &b) {
440         for (int i = 0; i < a.size(); i++) {
441             if (b.test(a[i]) <= 0)
442                 candidates.push_back(a[i]);
443             segment s(a[i], a[i + 1]);

```

```

444     vector<point> ps = intersect(b, s.as_line());
445     for (auto p : ps) {
446         if (s.contains(p))
447             candidates.push_back(p);
448     }
449 }
450 };
451 consider(a, b);
452 consider(b, a);
453 auto res = ConvexPolygon(ConvexPolygon::convex_hull(candidates));
454 return res;
455 }
456 friend Large intersection_area_or_dist(const ConvexPolygon &a,
457                                       const ConvexPolygon &b) {
458     ConvexPolygon inter = intersect(a, b);
459     if (inter.size() > 0)
460         return max(area(inter), Large(0));
461     ConvexPolygon sum = minkowski_sum(a, -b);
462     Large res = numeric_limits<Large>::max();
463     for (int i = 0; i < sum.size(); i++) {
464         res = min(res, dist(segment(sum[i], sum[i + 1]), point()));
465     }
466     return -res;
467 }
468 void cut(const halfplane& pl) {
469     int n = this->size();
470     if(n < 3) return;
471     p.push_back(p[0]);
472
473     auto pl_line = pl.as_line();
474
475     vector<point> out;
476     bool inside = pl.strictly_contains(p[0]);
477     if(inside) out.push_back(p[0]);
478
479     for(int i = 1; i <= n; i++) {
480         if(pl.strictly_contains(p[i])) {
481             if(!inside) {
482                 out.push_back(intersect(pl_line, line(p[i-1], p[i])).first);
483             }
484             out.push_back(p[i]);
485             inside = true;
486         } else {
487             if(inside) {
488                 out.push_back(intersect(pl_line, line(p[i-1], p[i])).first);
489             }
490             inside = false;
491         }
492     }
493
494     if(!out.empty() && out[0] == out.back()) out.pop_back();
495     *this = ConvexPolygon(ConvexPolygon::convex_hull(out));
496 }
497 void cut(const ConvexPolygon &rhs) {
498     for(int i = 0; i < rhs.size(); i++) {
499         cut(halfplane::from_points(rhs[i], rhs[i+1]));
500     }
501 }
502 };
503 } // namespace plane
504
505 template <typename T, typename Large = T>
506 struct PolygonPlane : public CirclePlane<T, Large> {
507     typedef plane::Polygon<T, Large> polygon;
508     typedef plane::ConvexPolygon<T, Large> convex_polygon;

```

```

509 };
510
511 } // namespace geo
512 } // namespace lib
513
514 #endif

```

6.6. Trigonometry

```

1 #ifndef _LIB_TRIGONOMETRY
2 #define _LIB_TRIGONOMETRY
3 #include <bits/stdc++.h>
4
5 namespace lib {
6 using namespace std;
7 namespace geo {
8 namespace trig {
9 constexpr static long double PI =
10     3.1415926535897932384626433832795028841971693993751058209749441;
11 double cos(double x) { return ::cos(x); }
12 double sin(double x) { return ::sin(x); }
13 double asin(double x) { return ::asin(x); }
14 double acos(double x) { return ::acos(x); }
15 double atan2(double y, double x) { return ::atan2(y, x); }
16 long double cos(long double x) { return ::cosl(x); }
17 long double sin(long double x) { return ::sinl(x); }
18 long double asin(long double x) { return ::asinl(x); }
19 long double acos(long double x) { return ::acosl(x); }
20 long double atan2(long double y, long double x) { return ::atan2l(y, x); }
21 } // namespace trig
22 } // namespace geo
23 } // namespace lib
24 #endif

```

7. polynomial

7.1. ExponentialSum

```

1 #ifndef _LIB_EXPONENTIAL_SUM
2 #define _LIB_EXPONENTIAL_SUM
3 #include <bits/stdc++.h>
4 #include "../Combinatorics.cpp"
5 #include "../Lagrange.cpp"
6
7 namespace lib {
8 using namespace std;
9
10 // given : f(0)...f(k) (deg(f) = k), a
11 // return : \sum_{i=0...infy} a^i f(i)
12 template<typename Field>
13 Field exponential_sum_limit(const vector<Field>& f, Field a) {
14     if(a == 0) return f[0];
15     assert(a != 1);
16     int m = f.size();
17     vector<Field> g(m);
18     Field acc = 1;
19     for(int i = 0; i < m; i++) g[i] = f[i] * acc, acc *= a;
20     for(int i = 1; i < m; i++) g[i] += g[i-1];
21     Field c = 0;
22     acc = 1;
23     for(int i = 0; i < m; i++) {
24         c += Combinatorics<Field>::nCr(m, i) * acc * g[m - i - 1];

```

```

25     acc *= -a;
26 }
27 c /= (1 - a)^m;
28 return c;
29 }
30
31 // given : f(0)...f(k) (deg(f) = k), a, n
32 // return : \sum_{i=0...n-1} a^i f(i)
33 template<typename Field>
34 Field exponential_sum(const vector<Field>& f, Field a, int64_t n) {
35     if(n == 0) return 0;
36     if(a == 0) return f[0];
37     if(a == 1) {
38         // Interpolate polynomial of deg == k + 1
39         return linalg::lagrange_iota_sum(f, n);
40     }
41     int m = f.size();
42     vector<Field> g(m);
43     auto c = exponential_sum_limit(f, a);
44     Field acc = 1;
45     for(int i = 0; i < m; i++) g[i] = f[i] * acc, acc *= a;
46     for(int i = 1; i < m; i++) g[i] += g[i-1];
47     auto ai = Field(1) / a;
48     acc = 1;
49     for(int i = 0; i < m; i++) {
50         g[i] = (g[i] - c) * acc;
51         acc *= ai;
52     }
53     // Interpolate polynomial of deg == k
54     auto tn = linalg::lagrange_iota(g, n - 1);
55     return c + (a^(n-1)) * tn;
56 }
57
58 // given : p, n
59 // return : (0^p, 1^p, ... , n^p)
60 template<typename Field>
61 vector<Field> monomials(int p, int n) {
62     vector<Field> f(n + 1, Field(0));
63     if (!p) {
64         f[0] = 1;
65         return std::move(f);
66     }
67     f[1] = 1;
68     vector<bool> sieve(n + 1, false);
69     vector<int> ps;
70     for (int i = 2; i <= n; i++) {
71         if (!sieve[i]) {
72             f[i] = Field(i)^p;
73             ps.push_back(i);
74         }
75         for (int j = 0; j < (int)ps.size() && i * ps[j] <= n; j++) {
76             sieve[i * ps[j]] = 1;
77             f[i * ps[j]] = f[i] * f[ps[j]];
78             if (i % ps[j] == 0) break;
79         }
80     }
81     return std::move(f);
82 }
83 } // namespace lib
84
85 #endif

```

7.2. MultipointEvaluation

```

1  #ifndef _LIB_POLYNOMIAL_MULTIPPOINT_EVALUATION
2  #define _LIB_POLYNOMIAL_MULTIPPOINT_EVALUATION
3  #include "../PolynomialRing.cpp"
4  #include "../Traits.cpp"
5  #include <bits/stdc++.h>
6
7  namespace lib {
8      using namespace std;
9      namespace math {
10         namespace {
11             /// keep caide
12             using traits::IsInputIterator;
13             /// keep caide
14             using traits::HasInputIterator;
15         } // namespace
16         template<typename Poly> struct MultipointEvaluation {
17             using field = typename Poly::field;
18             int n;
19             vector<field> w;
20             vector<Poly> up, down;
21
22             template<
23                 typename Iterator,
24                 typename enable_if<IsInputIterator<Iterator>::value>::type * = nullptr>
25             MultipointEvaluation(Iterator begin, Iterator end) : w(distance(begin,
26                 end)) {
27                 int i = 0;
28                 for (auto it = begin; it != end; ++it, ++i)
29                     w[i] = *it;
30                 n = w.size();
31                 build();
32             }
33
34             template<
35                 typename Container,
36                 typename enable_if<HasInputIterator<Container>::value>::type * =
37                     nullptr>
38             MultipointEvaluation(const Container &container)
39                 : MultipointEvaluation(container.begin(), container.end()) {}
40
41             void build() {
42                 if(w.empty()) return;
43                 up = vector<Poly>(2 * n);
44                 down = vector<Poly>(2 * n);
45                 for(int i = 0; i < n; i++)
46                     up[i+n] = {-w[i], 1};
47                 for(int i = n-1; i; i--)
48                     up[i] = up[2*i] * up[2*i+1];
49
50                 vector<field> eval(const Poly &p) {
51                     down[1] = p % up[1];
52                     for(int i = 2; i < 2*n; i++)
53                         down[i] = down[i/2] % up[i];
54                     vector<field> res(n);
55                     for(int i = 0; i < n; i++)
56                         res[i] = down[i+n][0];
57                     return res;
58                 }
59
60                 template<typename Iterator>
61                 Poly interp(const Iterator& begin, const Iterator& end) {
62                     assert(n == distance(begin, end));
63                     vector<field> a = eval(up[1].derivative());

```

```

64     auto it = begin;
65     for(int i = 0; i < n; i++, ++it)
66         down[i+n] = {*it / a[i]};
67     for(int i = n-1; i; i--)
68         down[i] = down[i*2] * up[i*2+1] + down[i*2+1] * up[i*2];
69     return down[1];
70 }
71
72 template <
73     typename Container,
74     typename enable_if<traits::HasInputIterator<Container>::value>::type *
75     = nullptr>
76 Poly interp(const Container &container) {
77     interp(container.begin(), container.end());
78 }
79 };
80 } // namespace math
81 } // namespace lib
82
83 #endif

```

```

41 #endif

```

7.3. Transform

```

1  #ifndef _LIB_POLYNOMIAL_TRANSFORM
2  #define _LIB_POLYNOMIAL_TRANSFORM
3  #include <bits/stdc++.h>
4
5  namespace lib {
6  using namespace std;
7  template<template <class> class T>
8  struct TransformMultiplication {
9      template<typename Field>
10      using Transform = T<Field>;
11
12      template <typename Field>
13      vector<Field> operator()(const vector<Field> &a,
14                             const vector<Field> &b) const {
15          return T<Field>::convolve(a, b);
16      };
17
18      template<typename Field>
19      inline VectorN<Field> transform(int n, const vector<Field> &p) const {
20          int np = next_power_of_two(n);
21          return T<Field>::transform(p, np);
22      }
23
24      template<typename Field>
25      inline vector<Field> itransform(int n, const vector<Field> &p) const {
26          return T<Field>::itransform(p, n);
27      }
28
29      template <typename Field, typename Functor, typename ...Ts>
30      inline vector<Field> on_transform(
31          int n,
32          Functor& f,
33          const vector<Ts> &... vs) const {
34          int np = next_power_of_two(n);
35          return T<Field>::itransform(
36              f(n, T<Field>::transform(vs, np)...), n);
37      }
38  };
39 } // namespace lib
40

```