

My Codebook

Felix Huang

September 1, 2022

Contents

1	Data-structures	1
1.1	DSU.h	1
1.2	Fenwick.h	2
1.3	HashMap.h	2
1.4	Segtree.h	2
1.5	LazySegtree.h	4
1.6	OrderStatisticTree.h	5
1.7	SparseTable.h	6
1.8	ConvexHullTrick.h	6
1.9	Treap.h	6
2	Combinatorial	7
2.1	Combination.h	7
2.2	CountInversions.h	7
3	Number-theory	7
3.1	ExtendGCD.h	7
3.2	InvGCD.h	8
3.3	StaticModint.h	8
3.4	DynamicModint.h	9
3.5	CRT.h	11
3.6	LinearSieve.h	11
3.7	ModInverses.h	12
3.8	ModPow.h	12
3.9	IsPrime.h	12
3.10	PrimitiveRoot.h	12
3.11	FloorSum.h	13
4	Numerical	13
4.1	Barrett.h	13
4.2	BitTransform.h	13
4.3	Poly.h	15
5	Geometry	17
5.1	Point.h	17
5.2	ConvexHull.h	18
6	Graph	19
6.1	LCA.h	19
6.2	HLD.h	20
6.3	TwoSat.h	21
6.4	Dinic.h	21
6.5	MCMF.h	22
7	String	22
7.1	SuffixArray.h	22

7.2	LCP.h	24
7.3	KMP.h	25
7.4	DynamicKMP.h	25
7.5	Zfunc.h	25
7.6	RollingHash.h	25
7.7	Manacher.h	26
7.8	Trie.h	26
7.9	AhoCorasick.h	27
8	Misc	28
8.1	Timer.h	28
8.2	Random.h	28
8.3	Debug.h	29
8.4	Discrete.h	29

1 Data-structures

1.1 DSU.h

```
1 class DSU {
2 public:
3     DSU() : DSU(0) {}
4
5     DSU(int _n) : n(_n), _size(vector<int>(n, -1))
6     → {}
7
8     inline int leader(int u) {
9         assert(0 <= u && u < n);
10        return (_size[u] < 0 ? u : (_size[u] =
11        → leader(_size[u])));
12    }
13
14    bool merge(int a, int b) {
15        assert(0 <= a && a < n);
16        assert(0 <= b && b < n);
17        a = leader(a);
18        b = leader(b);
19        if(a == b) {
20            return false;
21        }
22        if(-_size[a] < -_size[b]) {
23            swap(a, b);
24        }
25        _size[a] += _size[b];
26        _size[b] = a;
27        return true;
28    }
29
30    inline int size(int u) {
31        assert(0 <= u && u < n);
```

```

30     return _size[leader(u)];
31 }
32
33 inline bool same(int a, int b) {
34     assert(0 <= a && a < n);
35     assert(0 <= b && b < n);
36     return leader(a) == leader(b);
37 }
38
39 vector<vector<int>> groups() {
40     vector<int> leader_buf(n), group_size(n);
41     for(int i = 0; i < n; i++) {
42         leader_buf[i] = leader(i);
43         group_size[leader_buf[i]]++;
44     }
45     vector<vector<int>> result(n);
46     for(int i = 0; i < n; i++) {
47         result[i].reserve(group_size[i]);
48     }
49     for(int i = 0; i < n; i++) {
50         result[leader_buf[i]].push_back(i);
51     }
52     result.erase(remove_if(result.begin(),
↪ result.end(), [](const vector<int>& v) {
53         return v.empty();
54     }), result.end());
55     return result;
56 }
57
58 private:
59     int n;
60     vector<int> _size;
61 };
62

```

1.2 Fenwick.h

```

1 template<class T>
2 class fenwick {
3 public:
4     fenwick() : fenwick(0) {}
5
6     fenwick(int _n) : n(_n), data(_n) {}
7
8     void add(int p, T x) {
9         assert(0 <= p && p < n);
10        while(p < n) {
11            data[p] += x;
12            p |= (p + 1);
13        }
14    }
15
16    T get(int p) {
17        assert(0 <= p && p < n);
18        T res{};
19        while(p >= 0) {
20            res += data[p];
21            p = (p & (p + 1)) - 1;
22        }
23        return res;
24    }
25
26    T sum(int l, int r) {

```

```

27         return get(r) - (l ? get(l - 1) : T{});
28     }
29
30 private:
31     int n;
32     vector<T> data;
33 };
34

```

1.3 HashMap.h

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 using namespace __gnu_pbds;
3
4 struct splitmix64_hash {
5     static unsigned long long splitmix64(unsigned
↪ long long x) {
6         x += 0x9e3779b97f4a7c15;
7         x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
8         x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
9         return x ^ (x >> 31);
10    }
11
12    unsigned long long operator()(unsigned long
↪ long x) const {
13        static const unsigned long long
↪ FIXED_RANDOM =
↪ chrono::steady_clock::now().time_since_epoch().count()
14        return splitmix64(x + FIXED_RANDOM);
15    }
16 };
17
18 template<class T, class U, class H =
↪ splitmix64_hash> using hash_map =
↪ gp_hash_table<T, U, H>;
19 template<class T, class H = splitmix64_hash> using
↪ hash_set = hash_map<T, null_type, H>;
20

```

1.4 Segtree.h

```

1 // @param n `0 <= n`
2 // @return minimum non-negative `x` s.t. `n <=
↪ 2**x`
3 int ceil_pow2(int n) {
4     int x = 0;
5     while((1U << x) < (unsigned int)(n)) {
6         x++;
7     }
8     return x;
9 }
10
11 template<class T, T (*e)(), T (*op)(T, T)>
12 class segtree {
13 public:
14     segtree() : segtree(0) {}
15
16     segtree(int _n) : segtree(vector<T>(_n, e()))
↪ {}
17

```

```

18     segtree(const vector<T>& arr):
    ↪ n(int(arr.size())) {
19         log = ceil_pow2(n);
20         size = 1 << log;
21         st.resize(size << 1, e());
22         for(int i = 0; i < n; ++i) {
23             st[size + i] = arr[i];
24         }
25         for(int i = size - 1; i; --i) {
26             update(i);
27         }
28     }
29
30     void set(int p, T val) {
31         assert(0 <= p && p < n);
32         p += size;
33         st[p] = val;
34         for(int i = 1; i <= log; ++i) {
35             update(p >> i);
36         }
37     }
38
39     inline T get(int p) const {
40         assert(0 <= p && p < n);
41         return st[p + size];
42     }
43
44     inline T operator[](int p) const {
45         return get(p);
46     }
47
48     T prod(int l, int r) const {
49         assert(0 <= l && l <= r && r <= n);
50         T sml = e(), smr = e();
51         l += size;
52         r += size;
53         while(l < r) {
54             if(l & 1) {
55                 sml = op(sml, st[l++]);
56             }
57             if(r & 1) {
58                 smr = op(st[--r], smr);
59             }
60             l >>= 1;
61             r >>= 1;
62         }
63         return op(sml, smr);
64     }
65
66     inline T all_prod() const { return st[1]; }
67
68     template<bool (*f)(T)> int max_right(int l)
    ↪ const {
69         return max_right(l, [](T x) { return f(x);
    ↪ });
70     }
71
72     template<class F> int max_right(int l, F f)
    ↪ const {
73         assert(0 <= l && l <= n);
74         assert(f(e()));
75         if(l == n) {
76             return n;
77         }
78         l += size;

```

```

79         T sm = e();
80         do {
81             while(!(l & 1)) {
82                 l >>= 1;
83             }
84             if(!f(op(sm, st[l]))) {
85                 while(l < size) {
86                     l <<= 1;
87                     if(f(op(sm, st[l]))) {
88                         sm = op(sm, st[l]);
89                         l++;
90                     }
91                 }
92                 return l - size;
93             }
94             sm = op(sm, st[l]);
95             l++;
96         } while((l & -l) != l);
97         return n;
98     }
99
100     template<bool (*f)(T)> int min_left(int r)
    ↪ const {
101         return min_left(r, [](T x) { return f(x);
    ↪ });
102     }
103
104     template<class F> int min_left(int r, F f)
    ↪ const {
105         assert(0 <= r && r <= n);
106         assert(f(e()));
107         if(r == 0) {
108             return 0;
109         }
110         r += size;
111         T sm = e();
112         do {
113             r--;
114             while(r > 1 && (r & 1)) {
115                 r >>= 1;
116             }
117             if(!f(op(st[r], sm))) {
118                 while(r < size) {
119                     r = r << 1 | 1;
120                     if(f(op(st[r], sm))) {
121                         sm = op(st[r], sm);
122                         r--;
123                     }
124                 }
125                 return r + 1 - size;
126             }
127             sm = op(st[r], sm);
128         } while((r & -r) != r);
129         return 0;
130     }
131
132 private:
133     int n, size, log;
134     vector<T> st;
135
136     inline void update(int v) { st[v] = op(st[v <<
    ↪ 1], st[v << 1 | 1]); }
137 };
138

```

1.5 LazySegtree.h

```
1 // @param n `0 <= n`
2 // @return minimum non-negative `x` s.t. `n <= 2**x`
3 int ceil_pow2(int n) {
4     int x = 0;
5     while((1U << x) < (unsigned int)(n)) {
6         x++;
7     }
8     return x;
9 }
10
11 // Source: ac-library/atcoder/lazysegtree.hpp
12 template<class S,
13         S (*e)(),
14         S (*op)(S, S),
15         class F,
16         F (*id)(),
17         S (*mapping)(F, S),
18         F (*composition)(F, F)>
19 class lazy_segtree {
20 public:
21     lazy_segtree() : lazy_segtree(0) {}
22
23     explicit lazy_segtree(int _n) :
24     ↪ lazy_segtree(vector<S>(_n, e())) {}
25
26     explicit lazy_segtree(const vector<S>& v) :
27     ↪ n(int(v.size())) {
28         log = ceil_pow2(n);
29         size = 1 << log;
30         d = vector<S>(size << 1, e());
31         lz = vector<F>(size, id());
32         for(int i = 0; i < n; i++) {
33             d[size + i] = v[i];
34         }
35         for(int i = size - 1; i; --i) {
36             update(i);
37         }
38     }
39
40     void set(int p, S x) {
41         assert(0 <= p && p < n);
42         p += size;
43         for(int i = log; i; --i) {
44             push(p >> i);
45         }
46         d[p] = x;
47         for(int i = 1; i <= log; ++i) {
48             update(p >> i);
49         }
50     }
51
52     S get(int p) {
53         assert(0 <= p && p < n);
54         p += size;
55         for(int i = log; i; i--) {
56             push(p >> i);
57         }
58         return d[p];
59     }
60
61     S operator[](int p) {
```

```
62         return get(p);
63     }
64
65     S prod(int l, int r) {
66         assert(0 <= l && l <= r && r <= n);
67         if(l == r) {
68             return e();
69         }
70         l += size;
71         r += size;
72         for(int i = log; i; i--) {
73             if(((l >> i) << i) != 1) {
74                 push(l >> i);
75             }
76             if(((r >> i) << i) != r) {
77                 push(r >> i);
78             }
79         }
80         S sml = e(), smr = e();
81         while(l < r) {
82             if(l & 1) {
83                 sml = op(sml, d[l++]);
84             }
85             if(r & 1) {
86                 smr = op(d[--r], smr);
87             }
88             l >>= 1;
89             r >>= 1;
90         }
91         return op(sml, smr);
92     }
93
94     S all_prod() const { return d[1]; }
95
96     void apply(int p, F f) {
97         assert(0 <= p && p < n);
98         p += size;
99         for(int i = log; i; i--) {
100             push(p >> i);
101         }
102         d[p] = mapping(f, d[p]);
103         for(int i = 1; i <= log; i++) {
104             update(p >> i);
105         }
106     }
107
108     void apply(int l, int r, F f) {
109         assert(0 <= l && l <= r && r <= n);
110         if(l == r) {
111             return;
112         }
113         l += size;
114         r += size;
115         for(int i = log; i; i--) {
116             if(((l >> i) << i) != 1) {
117                 push(l >> i);
118             }
119             if(((r >> i) << i) != r) {
120                 push((r - 1) >> i);
121             }
122         }
123         {
124             int l2 = l, r2 = r;
125             while(l < r) {
126                 if(l & 1) {
127                     all_apply(l++, f);
128                 }
129                 if(r & 1) {
130                     all_apply(--r, f);
131                 }
132                 l >>= 1;
133                 r >>= 1;
134             }
135         }
136     }
137 }
```

```

125         }
126         if(r & 1) {
127             all_apply(--r, f);
128         }
129         l >>= 1;
130         r >>= 1;
131     }
132     l = l2;
133     r = r2;
134 }
135 for(int i = 1; i <= log; i++) {
136     if(((l >> i) << i) != 1) {
137         update(l >> i);
138     }
139     if(((r >> i) << i) != r) {
140         update((r - 1) >> i);
141     }
142 }
143 }
144
145 template<bool (*g)(S)> int max_right(int l) {
146     return max_right(l, [](S x) { return g(x);
→ });
147 }
148
149 template<class G> int max_right(int l, G g) {
150     assert(0 <= l && l <= n);
151     assert(g(e()));
152     if(l == n) {
153         return n;
154     }
155     l += size;
156     for(int i = log; i; i--) {
157         push(l >> i);
158     }
159     S sm = e();
160     do {
161         while(!(l & 1)) {
162             l >>= 1;
163         }
164         if(!g(op(sm, d[l]))) {
165             while(l < size) {
166                 push(l);
167                 l <<= 1;
168                 if(g(op(sm, d[l]))) {
169                     sm = op(sm, d[l]);
170                     l++;
171                 }
172             }
173             return l - size;
174         }
175         sm = op(sm, d[l]);
176         l++;
177     } while((l & -l) != 1);
178     return n;
179 }
180
181 template<bool (*g)(S)> int min_left(int r) {
182     return min_left(r, [](S x) { return g(x);
→ });
183 }
184
185 template<class G> int min_left(int r, G g) {
186     assert(0 <= r && r <= n);
187     assert(g(e()));

```

```

188     if(r == 0) {
189         return 0;
190     }
191     r += size;
192     for(int i = log; i >= 1; i--) {
193         push((r - 1) >> i);
194     }
195     S sm = e();
196     do {
197         r--;
198         while(r > 1 && (r & 1)) {
199             r >>= 1;
200         }
201         if(!g(op(d[r], sm))) {
202             while(r < size) {
203                 push(r);
204                 r = r << 1 | 1;
205                 if(g(op(d[r], sm))) {
206                     sm = op(d[r], sm);
207                     r--;
208                 }
209             }
210             return r + 1 - size;
211         }
212         sm = op(d[r], sm);
213     } while((r & -r) != r);
214     return 0;
215 }
216
217 private:
218     int n, size, log;
219     vector<S> d;
220     vector<F> lz;
221
222     inline void update(int k) { d[k] = op(d[k <<
→ 1], d[k << 1 | 1]); }
223
224     void all_apply(int k, F f) {
225         d[k] = mapping(f, d[k]);
226         if(k < size) {
227             lz[k] = composition(f, lz[k]);
228         }
229     }
230
231     void push(int k) {
232         all_apply(k << 1, lz[k]);
233         all_apply(k << 1 | 1, lz[k]);
234         lz[k] = id();
235     }
236 };
237

```

1.6 OrderStatisticTree.h

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 using namespace __gnu_pbds;
3
4 template<class T, class Comp = less<T>> using
→ ordered_set = tree<T, null_type, Comp,
→ rb_tree_tag,
→ tree_order_statistics_node_update>;

```

```

5 template<class T> using ordered_multiset =
  ↳ ordered_set<T, less_equal<T>>;

```

1.7 SparseTable.h

```

1 template<class T, T (*op)(T, T)>
2 class sparse_table {
3 public:
4     sparse_table() : n(0) {}
5
6     sparse_table(const vector<T>& a) {
7         n = static_cast<int>(a.size());
8         int max_log = 32 - __builtin_clz(n);
9         mat.resize(max_log);
10        mat[0] = a;
11        for(int j = 1; j < max_log; ++j) {
12            mat[j].resize(n - (1 << j) + 1);
13            for(int i = 0; i <= n - (1 << j); ++i)
14                ↳ mat[j][i] = op(mat[j - 1][i], mat[j
15                ↳ - 1][i + (1 << (j - 1))]);
16        }
17    }
18
19    inline T prod(int from, int to) const {
20        assert(0 <= from && from <= to && to <= n -
21        ↳ 1);
22        int lg = 31 - __builtin_clz(to - from + 1);
23        return op(mat[lg][from], mat[lg][to - (1 <<
24        ↳ lg) + 1]);
25    }
26
27    inline T operator[](int p) const {
28        assert(0 <= p && p < n);
29        return mat[0][p];
30    }
31
32 private:
33     int n;
34     vector<vector<T>> mat;
35 };

```

1.8 ConvexHullTrick.h

```

1 // Source:
2 ↳ https://github.com/kth-competitive-programming/kactl/blob/main/content/data-structures/LineContainer.h
3 struct Line_t {
4     mutable long long k, m, p;
5     bool operator<(const Line_t& o) const { return
6     ↳ k < o.k; }
7     bool operator<(long long x) const { return p <
8     ↳ x; }
9 };
10
11 // returns maximum (with minimum use negative
12 ↳ coefficient and constant)
13 struct CHT : multiset<Line_t, less<>> {
14     // (for doubles, use inf = 1/.0, div(a,b) =
15     ↳ a/b)

```

```

11 static const long long inf = LLONG_MAX;
12 long long div(long long a, long long b) { //
13 ↳ floored division
14     return a / b - ((a ^ b) < 0 && a % b);
15 }
16 bool isect(iterator x, iterator y) {
17     if(y == end()) {
18         x->p = inf;
19         return 0;
20     }
21     if(x->k == y->k) {
22         x->p = (x->m > y->m ? inf : -inf);
23     } else {
24         x->p = div(y->m - x->m, x->k - y->k);
25     }
26     return x->p >= y->p;
27 }
28 void insert_line(long long k, long long m) {
29     auto z = insert({k, m, 0}), y = z++, x = y;
30     while(isect(y, z)) {
31         z = erase(z);
32     }
33     if(x != begin() && isect(--x, y)) {
34         isect(x, y = erase(y));
35     }
36     while((y = x) != begin() && (--x)->p >=
37     ↳ y->p) {
38         isect(x, erase(y));
39     }
40 }
41 long long eval(long long x) {
42     assert(!empty());
43     auto l = *lower_bound(x);
44     return l.k * x + l.m;
45 }

```

1.9 Treap.h

```

1 mt19937_64
2 ↳ rng(chrono::steady_clock::now().time_since_epoch().
3
4 struct Node {
5     long long val;
6     long long sum;
7     bool rev;
8     int size;
9     int pri;
10
11     Node* l;
12     Node* r;
13
14     Node(long long x) : val(x), sum(x), rev(false),
15     ↳ size(1), pri(rng()), l(NULL), r(NULL) {}
16 };
17
18 inline int size(Node*& v) {
19     return (v ? v->size : 0);
20 }
21
22 void pull(Node*& v) {
23     v->size = 1 + size(v->l) + size(v->r);

```

```

22     v->sum = v->val + (v->l ? v->l->sum : 0) +
    ↪ (v->r ? v->r->sum : 0);
23 }
24
25 void push(Node*& v) {
26     if(v->rev) {
27         swap(v->l, v->r);
28         if(v->l) {
29             v->l->rev = !v->l->rev;
30         }
31         if(v->r) {
32             v->r->rev = !v->r->rev;
33         }
34         v->rev = false;
35     }
36 }
37
38 Node* merge(Node* a, Node* b) {
39     if(!a || !b) {
40         return (a ? a : b);
41     }
42     push(a);
43     push(b);
44     if(a->pri > b->pri) {
45         a->r = merge(a->r, b);
46         pull(a);
47         return a;
48     } else {
49         b->l = merge(a, b->l);
50         pull(b);
51         return b;
52     }
53 }
54
55 void split(Node* v, Node*& a, Node*& b, int k) {
56     if(k == 0) {
57         a = NULL;
58         b = v;
59         return;
60     }
61     push(v);
62     if(size(v->l) >= k) {
63         b = v;
64         split(v->l, a, v->l, k);
65         pull(b);
66     } else {
67         a = v;
68         split(v->r, v->r, b, k - size(v->l) - 1);
69         pull(a);
70     }
71 }
72

```

2 Combinatorial

2.1 Combination.h

```

1 vector<mint> fact{1}, inv_fact{1};
2
3 void init_fact(int n) {
4     while((int) fact.size() <= n) {

```

```

5         fact.push_back(fact.back() * (int)
    ↪ fact.size());
6     }
7     int sz = (int) inv_fact.size();
8     if(sz >= n + 1) {
9         return;
10    }
11    inv_fact.resize(n + 1);
12    inv_fact[n] = 1 / fact.back();
13    for(int i = n - 1; i >= sz; --i) {
14        inv_fact[i] = inv_fact[i + 1] * (i + 1);
15    }
16 }
17
18 mint C(int n, int k) {
19     if(k < 0 || k > n) {
20         return 0;
21     }
22     init_fact(n);
23     return fact[n] * inv_fact[k] * inv_fact[n - k];
24 }
25
26 mint P(int n, int k) {
27     if(k < 0 || k > n) {
28         return 0;
29     }
30     init_fact(n);
31     return fact[n] * inv_fact[n - k];
32 }
33

```

2.2 CountInversions.h

```

1 template<class T>
2 long long countInversions(vector<T> a) {
3     int n = (int) a.size();
4     a = ordered_compress(a);
5     fenwick<int> fenw(n + 1);
6     long long ans = 0;
7     for(int i = 0; i < n; ++i) {
8         ans += fenw.sum(a[i] + 1, n);
9         fenw.add(a[i], 1);
10    }
11    return ans;
12 }

```

3 Number-theory

3.1 ExtendGCD.h

```

1 // find x, y, gcd for ax + by = gcd(a, b)
2 long long ext_gcd(long long a, long long b, long
    ↪ long& x, long long& y) {
3     if(b == 0) {
4         x = 1;
5         y = 0;
6         return a;
7     }
8     long long x2, y2;
9     long long c = a % b;

```

```

10     if(c < 0) {
11         c += b;
12     }
13     long long g = ext_gcd(b, c, x2, y2);
14     x = y2;
15     y = x2 - (a / b) * y2;
16     return g;
17 }
18

```

3.2 InvGCD.h

```

1  /*
2  param  $1 \leq b$ 
3  return  $g, x$  s.t.
4       $g = \gcd(a, b)$ 
5       $ax = g \pmod{b}$ 
6       $0 \leq x < \frac{b}{g}$ 
7  */
8  constexpr pair<long long, long long> inv_gcd(long
  ↪ long a, long long b) {
9      a %= b;
10     if(a < 0) {
11         a += b;
12     }
13
14     if(a == 0) return {b, 0};
15
16     long long s = b, t = a;
17     long long m0 = 0, m1 = 1;
18
19     while(t) {
20         long long u = s / t;
21         s -= t * u;
22         m0 -= m1 * u;
23
24         // swap(s, t);
25         // swap(m0, m1);
26         auto tmp = s;
27         s = t;
28         t = tmp;
29         tmp = m0;
30         m0 = m1;
31         m1 = tmp;
32     }
33     if(m0 < 0) m0 += b / s;
34     return {s, m0};
35 }
36

```

3.3 StaticModint.h

```

1  template<int m>
2  class static_modint {
3  public:
4      static constexpr int mod() {
5          return m;
6      }
7
8      static_modint() : value(0) {}
9

```

```

10     static_modint(long long v) {
11         v %= mod();
12         if(v < 0) {
13             v += mod();
14         }
15         value = v;
16     }
17
18     const int& operator()() const {
19         return value;
20     }
21
22     template<class T>
23     explicit operator T() const {
24         return static_cast<T>(value);
25     }
26
27     static_modint& operator+=(const static_modint&
  ↪ rhs) {
28         value += rhs.value;
29         if(value >= mod()) {
30             value -= mod();
31         }
32         return *this;
33     }
34
35     static_modint& operator-=(const static_modint&
  ↪ rhs) {
36         value -= rhs.value;
37         if(value < 0) {
38             value += mod();
39         }
40         return *this;
41     }
42
43     static_modint& operator*=(const static_modint&
  ↪ rhs) {
44         value = (long long) value * rhs.value %
  ↪ mod();
45         return *this;
46     }
47
48     static_modint& operator/=(const static_modint&
  ↪ rhs) {
49         auto eg = inv_gcd(rhs.value, mod());
50         assert(eg.first == 1);
51         return *this *= eg.second;
52     }
53
54     template<class T>
55     static_modint& operator+=(const T& rhs) {
56         return *this += static_modint(rhs);
57     }
58
59     template<class T>
60     static_modint& operator-=(const T& rhs) {
61         return *this -= static_modint(rhs);
62     }
63
64     template<class T>
65     static_modint& operator*=(const T& rhs) {
66         return *this *= static_modint(rhs);
67     }
68
69     template<class T>

```



```

70 static_modint& operator/=(const T& rhs) {
71     return *this /= static_modint(rhs);
72 }
73
74 static_modint operator+() const {
75     return *this;
76 }
77
78 static_modint operator-() const {
79     return static_modint() - *this;
80 }
81
82 static_modint& operator++() {
83     return *this += 1;
84 }
85
86 static_modint& operator--() {
87     return *this -= 1;
88 }
89
90 static_modint operator++(int) {
91     static_modint res(*this);
92     *this += 1;
93     return res;
94 }
95
96 static_modint operator--(int) {
97     static_modint res(*this);
98     *this -= 1;
99     return res;
100 }
101
102 static_modint operator+(const static_modint&
→ rhs) {
103     return static_modint(*this) += rhs;
104 }
105
106 static_modint operator-(const static_modint&
→ rhs) {
107     return static_modint(*this) -= rhs;
108 }
109
110 static_modint operator*(const static_modint&
→ rhs) {
111     return static_modint(*this) *= rhs;
112 }
113
114 static_modint operator/(const static_modint&
→ rhs) {
115     return static_modint(*this) /= rhs;
116 }
117
118 inline bool operator==(const static_modint&
→ rhs) const {
119     return value == rhs();
120 }
121
122 inline bool operator!=(const static_modint&
→ rhs) const {
123     return !(*this == rhs);
124 }
125
126 private:
127     int value;
128 };

```

```

129
130 template<int m, class T> static_modint<m>
→ operator+(const T& lhs, const static_modint<m>&
→ rhs) {
131     return static_modint<m>(lhs) += rhs;
132 }
133
134 template<int m, class T> static_modint<m>
→ operator-(const T& lhs, const static_modint<m>&
→ rhs) {
135     return static_modint<m>(lhs) -= rhs;
136 }
137
138 template<int m, class T> static_modint<m>
→ operator*(const T& lhs, const static_modint<m>&
→ rhs) {
139     return static_modint<m>(lhs) *= rhs;
140 }
141
142 template<int m, class T> static_modint<m>
→ operator/(const T& lhs, const static_modint<m>&
→ rhs) {
143     return static_modint<m>(lhs) /= rhs;
144 }
145
146 template<int m>
147 istream& operator>>(istream& in, static_modint<m>&
→ num) {
148     long long x;
149     in >> x;
150     num = static_modint<m>(x);
151     return in;
152 }
153
154 template<int m>
155 ostream& operator<<(ostream& out, const
→ static_modint<m>& num) {
156     return out << num();
157 }
158
159 using modint998244353 = static_modint<998244353>;
160 using modint1000000007 = static_modint<1000000007>;
161

```

3.4 DynamicModint.h

```

1 template<int id>
2 class dynamic_modint {
3 public:
4     static int mod() {
5         return int(bt.umod());
6     }
7
8     static void set_mod(int m) {
9         assert(1 <= m);
10        bt = barrett(m);
11    }
12
13    dynamic_modint() : value(0) {}
14
15    dynamic_modint(long long v) {
16        v %= mod();
17        if(v < 0) {

```

```

18         v += mod();
19     }
20     value = v;
21 }
22
23 const unsigned int& operator()() const {
24     return value;
25 }
26
27 template<class T>
28 explicit operator T() const {
29     return static_cast<T>(value);
30 }
31
32 dynamic_modint& operator+=(const
↪ dynamic_modint& rhs) {
33     value += rhs.value;
34     if(value >= umod()) {
35         value -= umod();
36     }
37     return *this;
38 }
39
40 template<class T>
41 dynamic_modint& operator+=(const T& rhs) {
42     return *this += dynamic_modint(rhs);
43 }
44
45 dynamic_modint& operator-=(const
↪ dynamic_modint& rhs) {
46     value += mod() - rhs.value;
47     if(value >= umod()) {
48         value -= umod();
49     }
50     return *this;
51 }
52
53 template<class T>
54 dynamic_modint& operator-=(const T& rhs) {
55     return *this -= dynamic_modint(rhs);
56 }
57
58 dynamic_modint& operator*=(const
↪ dynamic_modint& rhs) {
59     value = bt.mul(value, rhs.value);
60     return *this;
61 }
62
63 template<class T>
64 dynamic_modint& operator*=(const T& rhs) {
65     return *this *= dynamic_modint(rhs);
66 }
67
68 dynamic_modint& operator/=(const
↪ dynamic_modint& rhs) {
69     auto eg = inv_gcd(rhs.value, mod());
70     assert(eg.first == 1);
71     return *this *= eg.second;
72 }
73
74 template<class T>
75 dynamic_modint& operator/=(const T& rhs) {
76     return *this /= dynamic_modint(rhs);
77 }
78

```

```

79 dynamic_modint operator+() const {
80     return *this;
81 }
82
83 dynamic_modint operator-() const {
84     return dynamic_modint() - *this;
85 }
86
87 dynamic_modint& operator++() {
88     ++value;
89     if(value == umod()) {
90         value = 0;
91     }
92     return *this;
93 }
94
95 dynamic_modint& operator--() {
96     if(value == 0) {
97         value = umod();
98     }
99     --value;
100     return *this;
101 }
102
103 dynamic_modint operator++(int) {
104     dynamic_modint res(*this);
105     ++*this;
106     return res;
107 }
108
109 dynamic_modint operator--(int) {
110     dynamic_modint res(*this);
111     --*this;
112     return res;
113 }
114
115 dynamic_modint operator+(const dynamic_modint&
↪ rhs) {
116     return dynamic_modint(*this) += rhs;
117 }
118
119 dynamic_modint operator-(const dynamic_modint&
↪ rhs) {
120     return dynamic_modint(*this) -= rhs;
121 }
122
123 dynamic_modint operator*(const dynamic_modint&
↪ rhs) {
124     return dynamic_modint(*this) *= rhs;
125 }
126
127 dynamic_modint operator/(const dynamic_modint&
↪ rhs) {
128     return dynamic_modint(*this) /= rhs;
129 }
130
131 inline bool operator==(const dynamic_modint&
↪ rhs) const {
132     return value == rhs();
133 }
134
135 inline bool operator!=(const dynamic_modint&
↪ rhs) const {
136     return !(*this == rhs);
137 }

```

```

138
139 private:
140     unsigned int value;
141     static barrett bt;
142     static unsigned int umod() { return bt.umod();
    ↪ }
143 };
144
145 template<int id, class T> dynamic_modint<id>
    ↪ operator+(const T& lhs, const
    ↪ dynamic_modint<id>& rhs) {
146     return dynamic_modint<id>(lhs) += rhs;
147 }
148
149 template<int id, class T> dynamic_modint<id>
    ↪ operator-(const T& lhs, const
    ↪ dynamic_modint<id>& rhs) {
150     return dynamic_modint<id>(lhs) -= rhs;
151 }
152
153 template<int id, class T> dynamic_modint<id>
    ↪ operator*(const T& lhs, const
    ↪ dynamic_modint<id>& rhs) {
154     return dynamic_modint<id>(lhs) *= rhs;
155 }
156
157 template<int id, class T> dynamic_modint<id>
    ↪ operator/(const T& lhs, const
    ↪ dynamic_modint<id>& rhs) {
158     return dynamic_modint<id>(lhs) /= rhs;
159 }
160
161 template<int id> barrett
    ↪ dynamic_modint<id>::bt(998244353);
162
163 template<int id>
164 istream& operator>>(istream& in,
    ↪ dynamic_modint<id>& num) {
165     long long x;
166     in >> x;
167     num = dynamic_modint<id>(x);
168     return in;
169 }
170
171 template<int id>
172 ostream& operator<<(ostream& out, const
    ↪ dynamic_modint<id>& num) {
173     return out << num();
174 }
175

```

3.5 CRT.h

```

1 // (rem, mod)
2 pair<long long, long long> crt(const vector<long
    ↪ long>& r, const vector<long long>& m) {
3     assert(r.size() == m.size());
4     int n = (int) r.size();
5     // Contracts: 0 <= r0 < m0
6     long long r0 = 0, m0 = 1;
7     for(int i = 0; i < n; i++) {
8         assert(1 <= m[i]);

```

```

9         long long r1 = safe_mod(r[i], m[i]), m1 =
    ↪ m[i];
10         if(m0 < m1) {
11             swap(r0, r1);
12             swap(m0, m1);
13         }
14         if(m0 % m1 == 0) {
15             if(r0 % m1 != r1) return {0, 0};
16             continue;
17         }
18         long long g, im;
19         tie(g, im) = inv_gcd(m0, m1);
20
21         long long u1 = (m1 / g);
22         if((r1 - r0) % g) return {0, 0};
23
24         long long x = (r1 - r0) / g % u1 * im % u1;
25
26         r0 += x * m0;
27         m0 *= u1;
28         if(r0 < 0) r0 += m0;
29     }
30     return {r0, m0};
31 }
32

```

3.6 LinearSieve.h

```

1 vector<bool> isprime;
2 vector<int> primes;
3 vector<int> phi;
4 vector<int> mobius;
5 void linear_sieve(int n) {
6     n += 1;
7     isprime.resize(n);
8     fill(isprime.begin() + 2, isprime.end(), true);
9     phi.resize(n);
10    mobius.resize(n);
11    phi[1] = mobius[1] = 1;
12    for(int i = 2; i < n; ++i) {
13        if(isprime[i]) {
14            primes.push_back(i);
15            phi[i] = i - 1;
16            mobius[i] = -1;
17        }
18        for(auto& j : primes) {
19            if(i * j >= n) {
20                break;
21            }
22            isprime[i * j] = false;
23            if(i % j == 0) {
24                mobius[i * j] = 0;
25                phi[i * j] = phi[i] * j;
26                break;
27            } else {
28                mobius[i * j] = mobius[i] *
    ↪ mobius[j];
29                phi[i * j] = phi[i] * phi[j];
30            }
31        }
32    }
33 }

```

3.7 ModInverses.h

```
1 // Calculate modular inverse for mod m up to n in
  ↳ O(n)
2 vector<int> mod_inverse(int m, int n = -1) {
3     assert(n < m);
4     if(n == -1) {
5         n = m - 1;
6     }
7     vector<int> inv(n + 1);
8     inv[0] = inv[1] = 1;
9     for(int i = 2; i <= n; ++i) {
10         inv[i] = m - (long long) (m / i) * inv[m %
  ↳ i] % m;
11     }
12     return inv;
13 }
14
```

3.8 ModPow.h

```
1 // @param n `0 <= n`
2 // @param m `1 <= m`
3 // @return `(x ** n) % m`
4 constexpr long long pow_mod_constexpr(long long x,
  ↳ long long n, int m) {
5     if(m == 1) return 0;
6     unsigned int _m = (unsigned int)(m);
7     unsigned long long r = 1;
8     x %= m;
9     if(x < 0) {
10         x += m;
11     }
12     unsigned long long y = x;
13     while(n) {
14         if(n & 1) r = (r * y) % _m;
15         y = (y * y) % _m;
16         n >>= 1;
17     }
18     return r;
19 }
20
```

3.9 IsPrime.h

```
1 // Reference:
2 // M. Forisek and J. Jancina,
3 // Fast Primality Testing for Integers That Fit into
  ↳ a Machine Word
4 // @param n `0 <= n`
5 constexpr bool is_prime_constexpr(int n) {
6     if(n <= 1) return false;
7     if(n == 2 || n == 7 || n == 61) return true;
8     if(n % 2 == 0) return false;
9     long long d = n - 1;
10     while(d % 2 == 0) d /= 2;
11     constexpr long long bases[3] = {2, 7, 61};
```

```
12     for(long long a : bases) {
13         long long t = d;
14         long long y = pow_mod_constexpr(a, t, n);
15         while(t != n - 1 && y != 1 && y != n - 1) {
16             y = y * y % n;
17             t <<= 1;
18         }
19         if(y != n - 1 && t % 2 == 0) {
20             return false;
21         }
22     }
23     return true;
24 }
25 template<int n> constexpr bool is_prime =
  ↳ is_prime_constexpr(n);
26
```

3.10 PrimitiveRoot.h

```
1 // Compile time primitive root
2 // @param m must be prime
3 // @return primitive root (and minimum in now)
4 constexpr int primitive_root_constexpr(int m) {
5     if(m == 2) return 1;
6     if(m == 167772161) return 3;
7     if(m == 469762049) return 3;
8     if(m == 754974721) return 11;
9     if(m == 998244353) return 3;
10     int divs[20] = {};
11     divs[0] = 2;
12     int cnt = 1;
13     int x = (m - 1) / 2;
14     while(x % 2 == 0) x /= 2;
15     for(int i = 3; (long long)(i)*i <= x; i += 2) {
16         if(x % i == 0) {
17             divs[cnt++] = i;
18             while(x % i == 0) {
19                 x /= i;
20             }
21         }
22     }
23     if(x > 1) {
24         divs[cnt++] = x;
25     }
26     for(int g = 2;; g++) {
27         bool ok = true;
28         for(int i = 0; i < cnt; i++) {
29             if(pow_mod_constexpr(g, (m - 1) /
  ↳ divs[i], m) == 1) {
30                 ok = false;
31                 break;
32             }
33         }
34         if(ok) return g;
35     }
36 }
37 template<int m> constexpr int primitive_root =
  ↳ primitive_root_constexpr(m);
38
```

3.11 FloorSum.h

```
1 // @param n `n < 2^32`
2 // @param m `1 <= m < 2^32`
3 // @return sum_{i=0}^{n-1} floor((ai + b) / m) (mod
  ↪ 2^64)
4 unsigned long long floor_sum_unsigned(unsigned long
  ↪ long n, unsigned long long m, unsigned long
  ↪ long a, unsigned long long b) {
5     unsigned long long ans = 0;
6     while(true) {
7         if(a >= m) {
8             ans += n * (n - 1) / 2 * (a / m);
9             a %= m;
10        }
11        if(b >= m) {
12            ans += n * (b / m);
13            b %= m;
14        }
15        unsigned long long y_max = a * n + b;
16        if(y_max < m) {
17            break;
18        }
19        // y_max < m * (n + 1)
20        // floor(y_max / m) <= n
21        n = (unsigned long long)(y_max / m);
22        b = (unsigned long long)(y_max % m);
23        swap(m, a);
24    }
25    return ans;
26 }
27
28 long long floor_sum(long long n, long long m, long
  ↪ long a, long long b) {
29     assert(0 <= n && n < (1LL << 32));
30     assert(1 <= m && m < (1LL << 32));
31     unsigned long long ans = 0;
32     if(a < 0) {
33         unsigned long long a2 = safe_mod(a, m);
34         ans -= 1ULL * n * (n - 1) / 2 * ((a2 - a) /
  ↪ m);
35         a = a2;
36     }
37     if(b < 0) {
38         unsigned long long b2 = safe_mod(b, m);
39         ans -= 1ULL * n * ((b2 - b) / m);
40         b = b2;
41     }
42     return ans + floor_sum_unsigned(n, m, a, b);
43 }
44
```

4 Numerical

4.1 Barrett.h

```
1 // Fast modular multiplication by barrett reduction
2 // Reference:
  ↪ https://en.wikipedia.org/wiki/Barrett\_reduction
3 class barrett {
4 public:
```

```
5     unsigned int m;
6     unsigned long long im;
7
8     explicit barrett(unsigned int _m) : m(_m),
  ↪ im((unsigned long long)(-1) / _m + 1) {}
9
10    unsigned int umod() const { return m; }
11
12    unsigned int mul(unsigned int a, unsigned int
  ↪ b) const {
13        unsigned long long z = a;
14        z *= b;
15        #ifdef _MSC_VER
16            unsigned long long x;
17            _umul128(z, im, &x);
18        #else
19            unsigned long long x = (unsigned long
  ↪ long)((unsigned __int128)(z) * im) >> 64);
20        #endif
21        unsigned int v = (unsigned int)(z - x * m);
22        if(m <= v) {
23            v += m;
24        }
25        return v;
26    }
27 };
28
```

4.2 BitTransform.h

```
1 template<class T>
2 void OrTransform(vector<T>& a) {
3     const int n = (int) a.size();
4     assert((n & -n) == n);
5     for(int i = 1; i < n; i <= 1) {
6         for(int j = 0; j < n; j += i << 1) {
7             for(int k = 0; k < i; ++k) {
8                 a[i + j + k] += a[j + k];
9             }
10        }
11    }
12 }
13
14 template<class T>
15 void OrInvTransform(vector<T>& a) {
16     const int n = (int) a.size();
17     assert((n & -n) == n);
18     for(int i = 1; i < n; i <= 1) {
19         for(int j = 0; j < n; j += i << 1) {
20             for(int k = 0; k < i; ++k) {
21                 a[i + j + k] -= a[j + k];
22             }
23        }
24    }
25 }
26
27 template<class T>
28 void AndTransform(vector<T>& a) {
29     const int n = (int) a.size();
30     assert((n & -n) == n);
31     for(int i = 1; i < n; i <= 1) {
32         for(int j = 0; j < n; j += i << 1) {
33             for(int k = 0; k < i; ++k) {
```

```

34         a[j + k] += a[i + j + k];
35     }
36 }
37 }
38 }
39
40 template<class T>
41 void AndInvTransform(vector<T>& a) {
42     const int n = (int) a.size();
43     assert((n & -n) == n);
44     for(int i = 1; i < n; i <= 1) {
45         for(int j = 0; j < n; j += i << 1) {
46             for(int k = 0; k < i; ++k) {
47                 a[j + k] -= a[i + j + k];
48             }
49         }
50     }
51 }
52
53 template<class T>
54 void XorTransform(vector<T>& a) {
55     const int n = (int) a.size();
56     assert((n & -n) == n);
57     for(int i = 1; i < n; i <= 1) {
58         for(int j = 0; j < n; j += i << 1) {
59             for(int k = 0; k < i; ++k) {
60                 T x = move(a[j + k]), y = move(a[i
↪ + j + k]);
61                 a[j + k] = x + y;
62                 a[i + j + k] = x - y;
63             }
64         }
65     }
66 }
67
68 template<class T>
69 void XorInvTransform(vector<T>& a) {
70     XorTransform(a);
71     T inv2 = T(1) / T((int) a.size());
72     for(auto& x : a) {
73         x *= inv2;
74     }
75 }
76
77 // Compute c[k] = sum(a[i] * b[j]) for (i or j) =
↪ k.
78 // Complexity: O(n log n)
79 template<class T>
80 vector<T> OrConvolution(vector<T> a, vector<T> b) {
81     const int n = (int) a.size();
82     assert(n == (int) b.size());
83     OrTransform(a);
84     OrTransform(b);
85     for(int i = 0; i < n; ++i) {
86         a[i] *= b[i];
87     }
88     OrInvTransform(a);
89     return a;
90 }
91
92 // Compute c[k] = sum(a[i] * b[j]) for (i and j) =
↪ k.
93 // Complexity: O(n log n)
94 template<class T>

```

```

95 vector<T> AndConvolution(vector<T> a, vector<T> b)
↪ {
96     const int n = (int) a.size();
97     assert(n == (int) b.size());
98     AndTransform(a);
99     AndTransform(b);
100     for(int i = 0; i < n; ++i) {
101         a[i] *= b[i];
102     }
103     AndInvTransform(a);
104     return a;
105 }
106
107 // Compute c[k] = sum(a[i] * b[j]) for (i xor j) =
↪ k.
108 // Complexity: O(n log n)
109 template<class T>
110 vector<T> XorConvolution(vector<T> a, vector<T> b)
↪ {
111     const int n = (int) a.size();
112     assert(n == (int) b.size());
113     XorTransform(a);
114     XorTransform(b);
115     for (int i = 0; i < n; ++i) {
116         a[i] *= b[i];
117     }
118     XorInvTransform(a);
119     return a;
120 }
121
122 template<class T>
123 void ZetaTransform(vector<T>& a) {
124     OrTransform(a);
125 }
126
127 template<class T>
128 void MobiusTransform(vector<T>& a) {
129     OrInvTransform(a);
130 }
131
132 template<class T>
133 vector<T> SubsetSumConvolution(const vector<T>& f,
↪ const vector<T>& g) {
134     const int n = (int) f.size();
135     assert(n == (int) g.size());
136     assert((n & -n) == n);
137     const int N = __lg(n);
138     vector<vector<T>> fhat(N + 1, vector<T>(n));
139     vector<vector<T>> ghat(N + 1, vector<T>(n));
140     for(int mask = 0; mask < n; ++mask) {
141         fhat[__builtin_popcount(mask)][mask] =
↪ f[mask];
142         ghat[__builtin_popcount(mask)][mask] =
↪ g[mask];
143     }
144     for(int i = 0; i <= N; ++i) {
145         ZetaTransform(fhat[i]);
146         ZetaTransform(ghat[i]);
147     }
148     vector<vector<T>> h(N + 1, vector<T>(n));
149     for(int mask = 0; mask < n; ++mask) {
150         for(int i = 0; i <= N; ++i) {
151             for(int j = 0; j <= i; ++j) {
152                 h[i][mask] += fhat[j][mask] *
↪ ghat[i - j][mask];

```

```

153     }
154 }
155 }
156 for(int i = 0; i <= N; ++i) {
157     MobiusTransform(h[i]);
158 }
159 vector<T> result(n);
160 for(int mask = 0; mask < n; ++mask) {
161     result[mask] =
→ h[__builtin_popcount(mask)][mask];
162 }
163 return result;
164 }
165

```

4.3 Poly.h

```

1 vector<int> __bit_reorder;
2
3 template<class T>
4 class Poly {
5 public:
6     static constexpr int R =
→ primitive_root<T::mod()>;
7
8     Poly() {}
9
10    Poly(int n) : coeff(n) {}
11
12    Poly(const vector<T>& a) : coeff(a) {}
13
14    Poly(const initializer_list<T>& a) : coeff(a)
→ {}
15
16    static constexpr int mod() {
17        return (int) T::mod();
18    }
19
20    inline int size() const {
21        return (int) coeff.size();
22    }
23
24    void resize(int n) {
25        coeff.resize(n);
26    }
27
28    T operator[](int idx) const {
29        if(idx < 0 || idx >= size()) {
30            return 0;
31        }
32        return coeff[idx];
33    }
34
35    T& operator[](int idx) {
36        return coeff[idx];
37    }
38
39    Poly mulxk(int k) const {
40        auto b = coeff;
41        b.insert(b.begin(), k, T(0));
42        return Poly(b);
43    }
44

```

```

45    Poly modxk(int k) const {
46        k = min(k, size());
47        return Poly(vector<T>(coeff.begin(),
→ coeff.begin() + k));
48    }
49
50    Poly divxk(int k) const {
51        if(size() <= k) {
52            return Poly<T>();
53        }
54        return Poly(vector<T>(coeff.begin() + k,
→ coeff.end()));
55    }
56
57    friend Poly operator+(const Poly& a, const
→ Poly& b) {
58        vector<T> res(max(a.size(), b.size()));
59        for(int i = 0; i < (int) res.size(); ++i) {
60            res[i] = a[i] + b[i];
61        }
62        return Poly(res);
63    }
64
65    friend Poly operator-(const Poly& a, const
→ Poly& b) {
66        vector<T> res(max(a.size(), b.size()));
67        for(int i = 0; i < (int) res.size(); ++i) {
68            res[i] = a[i] - b[i];
69        }
70        return Poly(res);
71    }
72
73    static void ensure_base(int n) {
74        if((int) __bit_reorder.size() != n) {
75            int k = __builtin_ctz(n) - 1;
76            __bit_reorder.resize(n);
77            for(int i = 0; i < n; ++i) {
78                __bit_reorder[i] = __bit_reorder[i
→ >> 1] >> 1 | (i & 1) << k;
79            }
80        }
81        if((int) roots.size() < n) {
82            int k = __builtin_ctz(roots.size());
83            roots.resize(n);
84            while((1 << k) < n) {
85                T e = pow_mod_constexpr(R,
→ (T::mod() - 1) >> (k + 1), T::mod());
86                for(int i = 1 << (k - 1); i < (1 <<
→ k); ++i) {
87                    roots[2 * i] = roots[i];
88                    roots[2 * i + 1] = roots[i] *
→ e;
89                }
90                k += 1;
91            }
92        }
93    }
94
95    static void dft(vector<T>& a) {
96        const int n = (int) a.size();
97        assert((n & -n) == n);
98        ensure_base(n);
99        for(int i = 0; i < n; ++i) {
100            if(__bit_reorder[i] < i) {
101                swap(a[i], a[__bit_reorder[i]]);

```



```

102     }
103 }
104 for(int k = 1; k < n; k *= 2) {
105     for(int i = 0; i < n; i += 2 * k) {
106         for(int j = 0; j < k; ++j) {
107             T u = a[i + j];
108             T v = a[i + j + k] * roots[k +
→ j];
109             a[i + j] = u + v;
110             a[i + j + k] = u - v;
111         }
112     }
113 }
114 }
115
116 static void idft(vector<T>& a) {
117     const int n = (int) a.size();
118     reverse(a.begin() + 1, a.end());
119     dft(a);
120     T inv = (1 - T::mod()) / n;
121     for(int i = 0; i < n; ++i) {
122         a[i] *= inv;
123     }
124 }
125
126 friend Poly operator*(Poly a, Poly b) {
127     if(a.size() == 0 || b.size() == 0) {
128         return Poly();
129     }
130     if(min(a.size(), b.size()) < 250) {
131         vector<T> c(a.size() + b.size() - 1);
132         for(int i = 0; i < a.size(); ++i) {
133             for(int j = 0; j < b.size(); ++j) {
134                 c[i + j] += a[i] * b[j];
135             }
136         }
137         return Poly(c);
138     }
139     int tot = a.size() + b.size() - 1;
140     int sz = 1;
141     while(sz < tot) {
142         sz <= 1;
143     }
144     a.coeff.resize(sz);
145     b.coeff.resize(sz);
146     dft(a.coeff);
147     dft(b.coeff);
148     for(int i = 0; i < sz; ++i) {
149         a.coeff[i] = a[i] * b[i];
150     }
151     idft(a.coeff);
152     a.resize(tot);
153     return a;
154 }
155
156 friend Poly operator*(T a, Poly b) {
157     for(int i = 0; i < b.size(); ++i) {
158         b[i] *= a;
159     }
160     return b;
161 }
162
163 friend Poly operator*(Poly a, T b) {
164     for(int i = 0; i < a.size(); ++i) {
165         a[i] *= b;
166     }
167     return a;
168 }
169
170 Poly& operator+=(Poly b) {
171     return *this = *this + b;
172 }
173
174 Poly& operator-=(Poly b) {
175     return *this = *this - b;
176 }
177
178 Poly& operator*=(Poly b) {
179     return *this = *this * b;
180 }
181
182 Poly deriv() const {
183     if(coeff.empty()) {
184         return Poly<T>();
185     }
186     vector<T> res(size() - 1);
187     for(int i = 0; i < size() - 1; ++i) {
188         res[i] = (i + 1) * coeff[i + 1];
189     }
190     return Poly(res);
191 }
192
193 Poly integr() const {
194     vector<T> res(size() + 1);
195     for(int i = 0; i < size(); ++i) {
196         res[i + 1] = coeff[i] / T(i + 1);
197     }
198     return Poly(res);
199 }
200
201 Poly inv(int m) const {
202     Poly x{T(1) / coeff[0]};
203     int k = 1;
204     while(k < m) {
205         k *= 2;
206         x = (x * (Poly{T(2)} - modxk(k) *
→ x)).modxk(k);
207     }
208     return x.modxk(m);
209 }
210
211 Poly log(int m) const {
212     return (deriv() *
→ inv(m)).integr().modxk(m);
213 }
214
215 Poly exp(int m) const {
216     Poly x{T(1)};
217     int k = 1;
218     while(k < m) {
219         k *= 2;
220         x = (x * (Poly{T(1)} - x.log(k) +
→ modxk(k))).modxk(k);
221     }
222     return x.modxk(m);
223 }
224
225 Poly pow(int k, int m) const {
226     if(k == 0) {
227         vector<T> a(m);

```



```

228     a[0] = 1;
229     return Poly(a);
230 }
231 int i = 0;
232 while(i < size() && coeff[i]() == 0) {
233     i++;
234 }
235 if(i == size() || 1LL * i * k >= m) {
236     return Poly(vector<T>(m));
237 }
238 T v = coeff[i];
239 auto f = divxk(i) * (1 / v);
240 return (f.log(m - i * k) * T(k)).exp(m - i
→ * k).mulxk(i * k) * power(v, k);
241 }
242
243 Poly sqrt(int m) const {
244     Poly<T> x{1};
245     int k = 1;
246     while(k < m) {
247         k *= 2;
248         x = (x + (modxk(k) *
→ x.inv(k)).modxk(k)) * T((mod() + 1) / 2);
249     }
250     return x.modxk(m);
251 }
252
253 Poly mulT(Poly b) const {
254     if(b.size() == 0) {
255         return Poly<T>();
256     }
257     int n = b.size();
258     reverse(b.coeff.begin(), b.coeff.end());
259     return ((*this) * b).divxk(n - 1);
260 }
261
262 vector<T> eval(vector<T> x) const {
263     if(size() == 0) {
264         return vector<T>(x.size(), 0);
265     }
266     const int n = max((int) x.size(), size());
267     vector<Poly<T>> q(4 * n);
268     vector<T> ans(x.size());
269     x.resize(n);
270     function<void(int, int, int)> build =
→ [&](int p, int l, int r) {
271         if(r - l == 1) {
272             q[p] = Poly{1, -x[l]};
273         } else {
274             int m = (l + r) / 2;
275             build(2 * p, l, m);
276             build(2 * p + 1, m, r);
277             q[p] = q[2 * p] * q[2 * p + 1];
278         }
279     };
280     build(1, 0, n);
281     function<void(int, int, int, const Poly&)>
→ work = [&](int p, int l, int r, const Poly&
→ num) {
282         if(r - l == 1) {
283             if(l < (int) ans.size()) {
284                 ans[l] = num[0];
285             }
286         } else {
287             int m = (l + r) / 2;

```

```

288         work(2 * p, l, m, num.mulT(q[2 * p
→ + 1])).modxk(m - 1));
289         work(2 * p + 1, m, r, num.mulT(q[2
→ * p])).modxk(r - m));
290     }
291 };
292 work(1, 0, n, mulT(q[1].inv(n)));
293 return ans;
294 }
295
296 private:
297     vector<T> coeff;
298     static vector<T> roots;
299 };
300
301 template<class T> vector<T> Poly<T>::roots{0, 1};
302

```

5 Geometry

5.1 Point.h

```

1  template<class T>
2  class Point {
3  public:
4      T x, y;
5
6      Point() : x(0), y(0) {}
7
8      Point(const T& a, const T& b) : x(a), y(b) {}
9
10     template<class U>
11     explicit Point(const Point<U>& p) :
→ x(static_cast<T>(p.x)), y(static_cast<T>(p.y))
→ {}
12
13     Point(const pair<T, T>& p) : x(p.first),
→ y(p.second) {}
14
15     Point(const complex<T>& p) : x(real(p)),
→ y(imag(p)) {}
16
17     explicit operator pair<T, T>() const {
18         return pair<T, T>(x, y);
19     }
20
21     explicit operator complex<T>() const {
22         return complex<T>(x, y);
23     }
24
25     inline Point& operator+=(const Point& rhs) {
26         x += rhs.x;
27         y += rhs.y;
28         return *this;
29     }
30
31     inline Point& operator-=(const Point& rhs) {
32         x -= rhs.x;
33         y -= rhs.y;
34         return *this;
35     }
36

```

```

37 inline Point& operator*=(const T& rhs) {
38     x *= rhs;
39     y *= rhs;
40     return *this;
41 }
42
43 inline Point& operator/=(const T& rhs) {
44     x /= rhs;
45     y /= rhs;
46     return *this;
47 }
48
49 template<class U>
50 inline Point& operator+=(const Point<U>& rhs) {
51     return *this += Point<T>(rhs);
52 }
53
54 template<class U>
55 inline Point& operator-=(const Point<U>& rhs) {
56     return *this -= Point<T>(rhs);
57 }
58
59 inline Point operator+() const {
60     return *this;
61 }
62
63 inline Point operator-() const {
64     return Point(-x, -y);
65 }
66
67 inline Point operator+(const Point& rhs) {
68     return Point(*this) += rhs;
69 }
70
71 inline Point operator-(const Point& rhs) {
72     return Point(*this) -= rhs;
73 }
74
75 inline Point operator*(const T& rhs) {
76     return Point(*this) *= rhs;
77 }
78
79 inline Point operator/(const T& rhs) {
80     return Point(*this) /= rhs;
81 }
82
83 inline bool operator==(const Point& rhs) {
84     return x == rhs.x && y == rhs.y;
85 }
86
87 inline bool operator!=(const Point& rhs) {
88     return !(*this == rhs);
89 }
90
91 inline T dist2() const {
92     return x * x + y * y;
93 }
94
95 inline long double dist() const {
96     return sqrt(dist2());
97 }
98
99 inline Point unit() const {
100     return *this / this->dist();
101 }

```

```

102
103 inline long double angle() const {
104     return atan2(y, x);
105 }
106
107 inline friend T dot(const Point& lhs, const
→ Point& rhs) {
108     return lhs.x * rhs.x + lhs.y * rhs.y;
109 }
110
111 inline friend T cross(const Point& lhs, const
→ Point& rhs) {
112     return lhs.x * rhs.y - lhs.y * rhs.x;
113 }
114
115 inline friend Point dot_cross(const Point& lhs,
→ const Point& rhs) {
116     return Point(dot(lhs, rhs), cross(lhs,
→ rhs));
117 }
118 };
119
120 template<class T>
121 istream& operator>>(istream& in, Point<T>& p) {
122     return in >> p.x >> p.y;
123 }
124

```

5.2 ConvexHull.h

```

1 template<class T>
2 vector<Point<T>> ConvexHull(vector<Point<T>>
→ points) {
3     const int n = (int) points.size();
4     sort(points.begin(), points.end(), [](const
→ Point<T>& a, const Point<T>& b) {
5         if(a.x == b.x) {
6             return a.y < b.y;
7         }
8         return a.x < b.x;
9     });
10    auto build = [&]() {
11        vector<Point<T>> upper;
12        upper.push_back(points[0]);
13        upper.push_back(points[1]);
14        for(int i = 2; i < n; ++i) {
15            while((int) upper.size() >= 2) {
16                if(cross(upper.end()[-1] -
→ upper.end()[-2], points[i] - upper.end()[-1]) >
→ 0) {
17                    upper.pop_back();
18                } else {
19                    break;
20                }
21            }
22            upper.push_back(points[i]);
23        }
24        return upper;
25    };
26    vector<Point<T>> upper = build();
27    reverse(points.begin(), points.end());
28    vector<Point<T>> lower = build();
29    lower.pop_back();

```

```

30     upper.insert(upper.end(), lower.begin() + 1,
    ↪     lower.end());
31     return upper;
32 }
33

```

6 Graph

6.1 LCA.h

```

1 template<class T>
2 class LCA {
3 public:
4     LCA() : LCA(0) {}
5     LCA(int _n) : n(_n), g(_n) {}
6
7     static pair<int, int> __lca_op(pair<int, int>
    ↪     a, pair<int, int> b) {
8         return min(a, b);
9     }
10
11     struct Edge {
12         int u, v;
13         T cost;
14
15         Edge(int a, int b, T c) : u(a), v(b),
    ↪         cost(c) {}
16     };
17
18     void add_edge(int u, int v, T cost = 1) {
19         assert(0 <= u && u < n);
20         assert(0 <= v && v < n);
21
22         g[u].push_back((int) edges.size());
23         g[v].push_back((int) edges.size());
24         edges.emplace_back(u, v, cost);
25     }
26
27     void build(int root) {
28         assert(0 <= root && root < n);
29         assert((int) edges.size() == n - 1);
30
31         _depth.assign(n, 0);
32         _dist.assign(n, 0);
33
34         euler_tour.reserve(2 * n - 1);
35         first_occurrence.assign(n, 0);
36
37         function<void(int, int, int)> dfs = [&](int
    ↪         u, int p, int d) {
38             _depth[u] = d;
39             first_occurrence[u] = (int)
    ↪         euler_tour.size();
40             euler_tour.push_back(u);
41
42             for(auto& id : g[u]) {
43                 int x = edges[id].u;
44                 int y = edges[id].v;
45                 T c = edges[id].cost;
46                 int v = u ^ x ^ y;
47
48                 if(v == p) {

```

```

49                     continue;
50                 }
51
52                 _depth[v] = _depth[u] + 1;
53                 _dist[v] = _dist[u] + c;
54
55                 dfs(v, u, d + 1);
56
57                 euler_tour.push_back(u);
58             }
59         };
60
61         dfs(root, -1, 0);
62
63         vector<pair<int, int>> route;
64         route.reserve((int) euler_tour.size());
65
66         for(auto& u : euler_tour) {
67             route.emplace_back(_depth[u], u);
68         }
69
70         st = sparse_table<pair<int, int>,
    ↪         __lca_op>(route);
71     }
72
73     inline int depth(int u) const {
74         assert(0 <= u && u < n);
75         return _depth[u];
76     }
77
78     inline int dist(int u) const {
79         assert(0 <= u && u < n);
80         return _dist[u];
81     }
82
83     int lca(int u, int v) const {
84         assert(0 <= u && u < n);
85         assert(0 <= v && v < n);
86
87         int l = first_occurrence[u];
88         int r = first_occurrence[v];
89
90         return st.prod(min(l, r), max(l,
    ↪         r)).second;
91     }
92
93     inline int dist(int u, int v) const {
94         assert(0 <= u && u < n);
95         assert(0 <= v && v < n);
96
97         return dist(u) + dist(v) - 2 * dist(lca(u,
    ↪         v));
98     }
99
100 protected:
101     int n;
102     vector<Edge> edges;
103     vector<vector<int>> g;
104
105     vector<int> _depth;
106     vector<T> _dist;
107
108     vector<int> euler_tour;
109     vector<int> first_occurrence;
110

```

```

111     sparse_table<pair<int, int>, __lca_op> st;
112 };
113

```

6.2 HLD.h

```

1  template<class T>
2  class HLD : LCA<T> {
3      using LCA<T>::n;
4      using LCA<T>::edges;
5      using LCA<T>::g;
6      using LCA<T>::build;
7
8  public:
9      using LCA<T>::add_edge;
10     using LCA<T>::parent;
11     using LCA<T>::lca;
12
13     HLD() : HLD(0) {}
14     HLD(int _n) : LCA<T>(_n) {}
15
16     void add_edge(int u, int v, T cost = 1) {
17         assert(0 <= u && u < n);
18         assert(0 <= v && v < n);
19
20         g[u].push_back((int) edges.size());
21         g[v].push_back((int) edges.size());
22         edges.emplace_back(u, v, cost);
23     }
24
25     void build_hld(int root = 0) {
26         build(root);
27
28         heavy_node.assign(n, -1);
29
30         function<int(int)> dfs = [&](int u) {
31             int sz = 1;
32             int max_sz = 0;
33
34             int p = parent(u);
35
36             for(auto& i : g[u]) {
37                 int x = edges[i].u;
38                 int y = edges[i].v;
39                 T c = edges[i].cost;
40
41                 int v = u ^ x ^ y;
42                 if(v == p) {
43                     continue;
44                 }
45
46                 int sub_sz = dfs(v);
47
48                 sz += sub_sz;
49
50                 if(sub_sz > max_sz) {
51                     max_sz = sub_sz;
52                     heavy_node[u] = v;
53                 }
54             }
55             return sz;
56         };
57

```

```

58     dfs(root);
59
60     id.assign(n, -1);
61
62     function<void(int)> dfs2 = [&](int u) {
63         static int counter = 0;
64         id[u] = counter++;
65
66         int p = parent(u);
67
68         if(heavy_node[u] != -1) {
69             dfs2(heavy_node[u]);
70         }
71
72         for(auto& i : g[u]) {
73             int x = edges[i].u;
74             int y = edges[i].v;
75             T c = edges[i].cost;
76
77             int v = u ^ x ^ y;
78             if(v == p || v == heavy_node[u]) {
79                 continue;
80             }
81
82             dfs2(v);
83         }
84     };
85
86     dfs2(root);
87
88     chain.resize(n);
89     iota(chain.begin(), chain.end(), 0);
90
91     function<void(int)> dfs3 = [&](int u) {
92         int p = parent(u);
93
94         if(heavy_node[u] != -1) {
95             chain[heavy_node[u]] = chain[u];
96         }
97
98         for(auto& i : g[u]) {
99             int x = edges[i].u;
100            int y = edges[i].v;
101            T c = edges[i].cost;
102
103            int v = u ^ x ^ y;
104            if(v == p) {
105                continue;
106            }
107
108            dfs3(v);
109        }
110    };
111
112     dfs3(root);
113
114     inline int get(int u) const {
115         return id[u];
116     }
117
118     // path[u, ..., p) where p is an ancestor of u
119     vector<pair<int, int>> path_up(int u, int p)
120     const {
121         vector<pair<int, int>> seg;

```

```

122         while(chain[u] != chain[p]) {
123             seg.emplace_back(id[chain[u]], id[u] +
↪ 1);
124             u = parent(chain[u]);
125         }
126
127         // id[p] is smaller than id[u] but we don't
↪ want id[p]
128         seg.emplace_back(id[p] + 1, id[u] + 1);
129
130         return seg;
131     }
132
133     vector<pair<int, int>> path(int u, int v) const
↪ {
134         int z = lca(u, v);
135
136         auto lhs = path_up(u, z);
137         auto rhs = path_up(v, z);
138
139         lhs.emplace_back(id[z], id[z] + 1);
140         lhs.insert(lhs.end(), rhs.begin(),
↪ rhs.end());
141
142         return lhs;
143     }
144
145 private:
146     vector<int> heavy_node;
147     vector<int> id;
148     vector<int> chain;
149 };
150
151

```

6.3 TwoSat.h

6.4 Dinic.h

```

1 template<class T>
2 class Dinic {
3 public:
4     struct Edge {
5         int to;
6         T cap;
7         Edge(int _to, T _cap) : to(_to), cap(_cap)
↪ {}
8     };
9
10     static constexpr T INF =
↪ numeric_limits<T>::max() / 2;
11
12     int n;
13     vector<Edge> e;
14     vector<vector<int>> g;
15     vector<int> cur, h;
16
17     Dinic() {}
18     Dinic(int _n) : n(_n), g(_n) {}

```

```

19
20     void add_edge(int u, int v, T c) {
21         assert(0 <= u && u < n);
22         assert(0 <= v && v < n);
23         g[u].push_back(e.size());
24         e.emplace_back(v, c);
25         g[v].push_back(e.size());
26         e.emplace_back(u, 0);
27     }
28
29     bool bfs(int s, int t) {
30         h.assign(n, -1);
31         queue<int> que;
32         h[s] = 0;
33         que.push(s);
34         while(!que.empty()) {
35             int u = que.front();
36             que.pop();
37             for(int i : g[u]) {
38                 int v = e[i].to;
39                 T c = e[i].cap;
40                 if(c > 0 && h[v] == -1) {
41                     h[v] = h[u] + 1;
42                     if(v == t) {
43                         return true;
44                     }
45                     que.push(v);
46                 }
47             }
48         }
49         return false;
50     }
51
52     T dfs(int u, int t, T f) {
53         if(u == t) {
54             return f;
55         }
56         T r = f;
57         for(int &i = cur[u]; i < int(g[u].size());
↪ ++i) {
58             int j = g[u][i];
59             int v = e[j].to;
60             T c = e[j].cap;
61             if(c > 0 && h[v] == h[u] + 1) {
62                 T a = dfs(v, t, min(r, c));
63                 e[j].cap -= a;
64                 e[j ^ 1].cap += a;
65                 r -= a;
66                 if (r == 0) {
67                     return f;
68                 }
69             }
70         }
71         return f - r;
72     }
73
74     T flow(int s, int t) {
75         assert(0 <= s && s < n);
76         assert(0 <= t && t < n);
77         T ans = 0;
78         while(bfs(s, t)) {
79             cur.assign(n, 0);
80             ans += dfs(s, t, INF);
81         }
82         return ans;

```

```

83     }
84 };
85


---


6.5 MCMF.h


---


1  template<class Cap_t, class Cost_t>
2  class MCMF {
3  public:
4      struct Edge {
5          int from;
6          int to;
7          Cap_t cap;
8          Cost_t cost;
9          Edge(int u, int v, Cap_t _cap, Cost_t
→ _cost) : from(u), to(v), cap(_cap), cost(_cost)
→ {}
10     };
11
12     static constexpr Cap_t EPS =
→ static_cast<Cap_t>(1e-9);
13
14     int n;
15     vector<Edge> edges;
16     vector<vector<int>>> g;
17     vector<Cost_t> d;
18     vector<bool> in_queue;
19     vector<int> previous_edge;
20
21     MCMF(int _n) : n(_n), g(_n), d(_n),
→ in_queue(_n), previous_edge(_n) {}
22
23     void add_edge(int u, int v, Cap_t cap, Cost_t
→ cost) {
24         assert(0 <= u && u < n);
25         assert(0 <= v && v < n);
26         g[u].push_back(edges.size());
27         edges.emplace_back(u, v, cap, cost);
28         g[v].push_back(edges.size());
29         edges.emplace_back(v, u, 0, -cost);
30     }
31
32     bool bfs(int s, int t) {
33         bool found = false;
34         fill(d.begin(), d.end(),
→ numeric_limits<Cost_t>::max());
35         d[s] = 0;
36         in_queue[s] = true;
37         queue<int> que;
38         que.push(s);
39         while(!que.empty()) {
40             int u = que.front();
41             que.pop();
42             if(u == t) {
43                 found = true;
44             }
45             in_queue[u] = false;
46             for(auto& id : g[u]) {
47                 const Edge& e = edges[id];
48                 if(e.cap > EPS && d[u] + e.cost <
→ d[e.to]) {
49                     d[e.to] = d[u] + e.cost;
50                     previous_edge[e.to] = id;

```

```

51         if(!in_queue[e.to]) {
52             que.push(e.to);
53             in_queue[e.to] = true;
54         }
55     }
56 }
57
58     return found;
59 }
60
61     pair<Cap_t, Cost_t> flow(int s, int t) {
62         assert(0 <= s && s < n);
63         assert(0 <= t && t < n);
64         Cap_t cap = 0;
65         Cost_t cost = 0;
66         while(bfs(s, t)) {
67             Cap_t send =
→ numeric_limits<Cap_t>::max();
68             int u = t;
69             while(u != s) {
70                 const Edge& e =
→ edges[previous_edge[u]];
71                 send = min(send, e.cap);
72                 u = e.from;
73             }
74             u = t;
75             while(u != s) {
76                 Edge& e = edges[previous_edge[u]];
77                 e.cap -= send;
78                 Edge& b = edges[previous_edge[u] ^
→ 1];
79                 b.cap += send;
80                 u = e.from;
81             }
82             cap += send;
83             cost += send * d[t];
84         }
85         return make_pair(cap, cost);
86     }
87 };
88


---



```

7 String

7.1 SuffixArray.h

```

1  vector<int> sa_naive(const vector<int>& s) {
2      int n = int(s.size());
3      vector<int> sa(n);
4      iota(sa.begin(), sa.end(), 0);
5      sort(sa.begin(), sa.end(), [&](int l, int r) {
6          if(l == r) {
7              return false;
8          }
9          while(l < n && r < n) {
10             if(s[l] != s[r]) {
11                 return s[l] < s[r];
12             }
13             l++;
14             r++;
15         }
16         return l == n;

```

```

17     });
18     return sa;
19 }
20
21 vector<int> sa_doubling(const vector<int>& s) {
22     int n = int(s.size());
23     vector<int> sa(n), rnk = s, tmp(n);
24     iota(sa.begin(), sa.end(), 0);
25     for(int k = 1; k < n; k *= 2) {
26         auto cmp = [&](int x, int y) {
27             if(rnk[x] != rnk[y]) return rnk[x] <
→ rnk[y];
28             int rx = x + k < n ? rnk[x + k] : -1;
29             int ry = y + k < n ? rnk[y + k] : -1;
30             return rx < ry;
31         };
32         sort(sa.begin(), sa.end(), cmp);
33         tmp[sa[0]] = 0;
34         for(int i = 1; i < n; i++) {
35             tmp[sa[i]] = tmp[sa[i - 1]] + (cmp(sa[i
→ - 1], sa[i]) ? 1 : 0);
36         }
37         swap(tmp, rnk);
38     }
39     return sa;
40 }
41
42 // SA-IS, linear-time suffix array construction
43 // Reference:
44 // G. Nong, S. Zhang, and W. H. Chan,
45 // Two Efficient Algorithms for Linear Time Suffix
→ Array Construction
46 template<int THRESHOLD_NAIVE = 10, int
→ THRESHOLD_DOUBLING = 40>
47 vector<int> sa_is(const vector<int>& s, int upper)
→ {
48     int n = int(s.size());
49     if(n == 0) {
50         return {};
51     }
52     if(n == 1) {
53         return {0};
54     }
55     if(n == 2) {
56         if(s[0] < s[1]) {
57             return {0, 1};
58         } else {
59             return {1, 0};
60         }
61     }
62     if(n < THRESHOLD_NAIVE) {
63         return sa_naive(s);
64     }
65     if(n < THRESHOLD_DOUBLING) {
66         return sa_doubling(s);
67     }
68     vector<int> sa(n);
69     vector<bool> ls(n);
70     for(int i = n - 2; i >= 0; i--) {
71         ls[i] = (s[i] == s[i + 1]) ? ls[i + 1] :
→ (s[i] < s[i + 1]);
72     }
73     vector<int> sum_l(upper + 1), sum_s(upper + 1);
74     for(int i = 0; i < n; i++) {
75         if(!ls[i]) {
76             sum_s[s[i]]++;
77         } else {
78             sum_l[s[i] + 1]++;
79         }
80     }
81     for(int i = 0; i <= upper; i++) {
82         sum_s[i] += sum_l[i];
83         if(i < upper) {
84             sum_l[i + 1] += sum_s[i];
85         }
86     }
87
88     auto induce = [&](const vector<int>& lms) {
89         fill(sa.begin(), sa.end(), -1);
90         vector<int> buf(upper + 1);
91         copy(sum_s.begin(), sum_s.end(),
→ buf.begin());
92         for(auto d : lms) {
93             if(d == n) {
94                 continue;
95             }
96             sa[buf[s[d]]++] = d;
97         }
98         copy(sum_l.begin(), sum_l.end(),
→ buf.begin());
99         sa[buf[s[n - 1]]++] = n - 1;
100         for(int i = 0; i < n; i++) {
101             int v = sa[i];
102             if(v >= 1 && !ls[v - 1]) {
103                 sa[buf[s[v - 1]]++] = v - 1;
104             }
105         }
106         copy(sum_l.begin(), sum_l.end(),
→ buf.begin());
107         for(int i = n - 1; i >= 0; i--) {
108             int v = sa[i];
109             if(v >= 1 && ls[v - 1]) {
110                 sa[--buf[s[v - 1] + 1]] = v - 1;
111             }
112         }
113     };
114
115     vector<int> lms_map(n + 1, -1);
116     int m = 0;
117     for(int i = 1; i < n; i++) {
118         if(!ls[i - 1] && ls[i]) {
119             lms_map[i] = m++;
120         }
121     }
122     vector<int> lms;
123     lms.reserve(m);
124     for(int i = 1; i < n; i++) {
125         if(!ls[i - 1] && ls[i]) {
126             lms.push_back(i);
127         }
128     }
129
130     induce(lms);
131
132     if(m) {
133         vector<int> sorted_lms;
134         sorted_lms.reserve(m);
135         for(int v : sa) {
136             if(lms_map[v] != -1) {
137                 sorted_lms.push_back(v);

```



```

138     }
139 }
140 vector<int> rec_s(m);
141 int rec_upper = 0;
142 rec_s[lms_map[sorted_lms[0]]] = 0;
143 for(int i = 1; i < m; i++) {
144     int l = sorted_lms[i - 1], r =
↪ sorted_lms[i];
145     int end_l = (lms_map[l] + 1 < m) ?
↪ lms[lms_map[l] + 1] : n;
146     int end_r = (lms_map[r] + 1 < m) ?
↪ lms[lms_map[r] + 1] : n;
147     bool same = true;
148     if(end_l - l != end_r - r) {
149         same = false;
150     } else {
151         while(l < end_l) {
152             if(s[l] != s[r]) {
153                 break;
154             }
155             l++;
156             r++;
157         }
158         if(l == n || s[l] != s[r]) {
159             same = false;
160         }
161     }
162     if(!same) {
163         rec_upper++;
164     }
165     rec_s[lms_map[sorted_lms[i]]] =
↪ rec_upper;
166 }
167
168 auto rec_sa = sa_is<THRESHOLD_NAIVE,
↪ THRESHOLD_DOUBLING>(rec_s, rec_upper);
169
170 for(int i = 0; i < m; i++) {
171     sorted_lms[i] = lms[rec_sa[i]];
172 }
173 induce(sorted_lms);
174 }
175 return sa;
176 }
177
178 vector<int> suffix_array(const vector<int>& s, int
↪ upper) {
179     assert(0 <= upper);
180     for(int d : s) {
181         assert(0 <= d && d <= upper);
182     }
183     auto sa = sa_is(s, upper);
184     return sa;
185 }
186
187 template<class T>
188 vector<int> suffix_array(const vector<T>& s) {
189     int n = int(s.size());
190     vector<int> idx(n);
191     iota(idx.begin(), idx.end(), 0);
192     sort(idx.begin(), idx.end(), [&](int l, int r)
↪ { return s[l] < s[r]; });
193     vector<int> s2(n);
194     int now = 0;
195     for(int i = 0; i < n; i++) {

```

```

196         if(i && s[idx[i - 1]] != s[idx[i]]) {
197             now++;
198         }
199         s2[idx[i]] = now;
200     }
201     return sa_is(s2, now);
202 }
203
204 vector<int> suffix_array(const string& s) {
205     int n = int(s.size());
206     vector<int> s2(n);
207     for(int i = 0; i < n; i++) {
208         s2[i] = s[i];
209     }
210     return sa_is(s2, 255);
211 }
212

```

7.2 LCP.h

```

1 // Reference:
2 // T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K.
↪ Park,
3 // Linear-Time Longest-Common-Prefix Computation in
↪ Suffix Arrays and Its
4 // Applications
5 template<class T>
6 vector<int> lcp_array(const vector<T>& s, const
↪ vector<int>& sa) {
7     int n = int(s.size());
8     assert(n >= 1);
9     vector<int> rnk(n);
10    for(int i = 0; i < n; i++) {
11        rnk[sa[i]] = i;
12    }
13    vector<int> lcp(n - 1);
14    int h = 0;
15    for(int i = 0; i < n; i++) {
16        if(h > 0) {
17            h--;
18        }
19        if(rnk[i] == 0) {
20            continue;
21        }
22        int j = sa[rnk[i] - 1];
23        for(; j + h < n && i + h < n; h++) {
24            if(s[j + h] != s[i + h]) {
25                break;
26            }
27        }
28        lcp[rnk[i] - 1] = h;
29    }
30    return lcp;
31 }
32
33 vector<int> lcp_array(const string& s, const
↪ vector<int>& sa) {
34     int n = int(s.size());
35     vector<int> s2(n);
36     for(int i = 0; i < n; i++) {
37         s2[i] = s[i];
38     }
39     return lcp_array(s2, sa);

```


40 }

41

7.3 KMP.h

```
1 template<class T>
2 vector<int> KMP(const vector<T>& a) {
3     int n = (int) a.size();
4     vector<int> k(n);
5     for(int i = 1; i < n; ++i) {
6         int j = k[i - 1];
7         while(j > 0 && a[i] != a[j]) {
8             j = k[j - 1];
9         }
10        if(a[i] == a[j]) {
11            j += 1;
12        }
13        k[i] = j;
14    }
15    return k;
16 }
17
18 vector<int> KMP(const string& s) {
19     vector<int> s2(s.begin(), s.end());
20     return KMP(s2);
21 }
22
```

7.4 DynamicKMP.h

```
1 template<int ALPHABET, int (*f)(char)>
2 class DynamicKMP {
3 public:
4     DynamicKMP() {}
5
6     DynamicKMP(const string& s) {
7         reserve(s.size());
8         for(const char& c : s) {
9             push(c);
10        }
11    }
12
13    void push(char c) {
14        int v = f(c);
15        dp.emplace_back();
16        dp.back()[v] = (int) dp.size();
17        if(p.empty()) {
18            p.push_back(0);
19            return;
20        }
21        int i = (int) p.size();
22        for(int j = 0; j < ALPHABET; ++j) {
23            if(j == v) {
24                p.push_back(dp[p[i - 1]][j]);
25            } else {
26                dp.back()[j] = dp[p[i - 1]][j];
27            }
28        }
29    }
30
31    void pop() {
```

```
32        p.pop_back();
33        dp.pop_back();
34    }
35
36    int query() const {
37        return p.back();
38    }
39
40    vector<int> query_all() const {
41        return p;
42    }
43
44    void reserve(int sz) {
45        p.reserve(sz);
46        dp.reserve(sz);
47    }
48
49 private:
50     vector<int> p;
51     vector<array<int, ALPHABET>> dp;
52 }
```

7.5 Zfunc.h

```
1 template<class T>
2 vector<int> z_algorithm(const vector<T>& a) {
3     int n = (int) a.size();
4     vector<int> z(n);
5     for(int i = 1, j = 0; i < n; ++i) {
6         if(i <= j + z[j]) {
7             z[i] = min(z[i - j], j + z[j] - i);
8         }
9         while(i + z[i] < n && a[i + z[i]] ==
10 → a[z[i]]) {
11             z[i] += 1;
12         }
13         if(i + z[i] > j + z[j]) {
14             j = i;
15         }
16     }
17     return z;
18 }
19
20 vector<int> z_algorithm(const string& s) {
21     vector<int> s2(s.begin(), s.end());
22     return z_algorithm(s2);
23 }
```

7.6 RollingHash.h

```
1 // @param m `1 <= m`
2 // @return x mod m
3 constexpr long long safe_mod(long long x, long long
4 → m) {
5     x %= m;
6     if(x < 0) {
7         x += m;
8     }
9     return x;
10 }
```

```

10
11 // @param n `0 <= n`
12 // @param m `1 <= m`
13 // @return `(x ** n) % m`
14 constexpr long long pow_mod_constexpr(long long x,
    ↪ long long n, int m) {
15     if(m == 1) return 0;
16     unsigned int _m = (unsigned int)(m);
17     unsigned long long r = 1;
18     unsigned long long y = safe_mod(x, m);
19     while(n) {
20         if(n & 1) r = (r * y) % _m;
21         y = (y * y) % _m;
22         n >>= 1;
23     }
24     return r;
25 }
26
27 template<class T>
28 class Rolling_Hash {
29 public:
30     Rolling_Hash() {}
31
32     Rolling_Hash(int _A, string _s): A(_A), n((int)
    ↪ _s.size()), s(_s), pref(n) {
33         pref[0] = s[0];
34         for(int i = 1; i < n; ++i) {
35             pref[i] = pref[i - 1] * A + s[i];
36         }
37     }
38
39     inline int size() const {
40         return n;
41     }
42
43     inline T get(int l, int r) const {
44         assert(0 <= l && l <= r && r < n);
45         if(l == 0) {
46             return pref[r];
47         }
48         return pref[r] - pref[l - 1] *
    ↪ pow_mod_constexpr(A, r - l + 1, T::mod());
49     }
50
51     inline T id() const {
52         return pref.back();
53     }
54
55 private:
56     int A;
57     int n;
58     string s;
59     vector<T> pref;
60 };
61

```

7.7 Manacher.h

```

1 template<class T>
2 vector<int> manacher_odd(const vector<T>& a) {
3     vector<T> b(1, -87);
4     b.insert(b.end(), a.begin(), a.end());
5     b.push_back(-69);

```

```

6     int n = (int) b.size();
7     vector<int> z(n);
8     z[0] = 1;
9     for(int i = 1, l = -1, r = 1; i <= n; ++i) {
10         if(i < r) {
11             z[i] = min(z[l + r - i], r - i);
12         }
13         while(b[i - z[i]] == b[i + z[i]]) {
14             z[i] += 1;
15         }
16         if(i + z[i] - 1 > r) {
17             l = i - z[i] + 1;
18             r = i + z[i] - 1;
19         }
20     }
21     return vector<int>(z.begin() + 1, z.end() - 1);
22 }
23
24 template<class T>
25 vector<int> manacher(const vector<T>& a) {
26     int n = (int) a.size();
27     vector<int> idx(n);
28     iota(idx.begin(), idx.end(), 0);
29     sort(idx.begin(), idx.end(), [&](int l, int r)
    ↪ { return s[l] < s[r]; });
30     vector<int> b(n);
31     int now = 0;
32     for(int i = 0; i < n; i++) {
33         if(i && s[idx[i - 1]] != s[idx[i]]) {
34             now++;
35         }
36         b[idx[i]] = now;
37     }
38     vector<int> s2;
39     s2.reserve((int) b.size() * 2);
40     for(auto& x : b) {
41         s2.push_back(x);
42         s2.push_back(-1);
43     }
44     s2.pop_back();
45     return manacher_odd(s2);
46 }
47
48 vector<int> manacher(const string& s) {
49     vector<int> s2;
50     s2.reserve((int) s.size() * 2);
51     for(const auto& c : s) {
52         s2.push_back(c);
53         s2.push_back(-1);
54     }
55     s2.pop_back();
56     return manacher_odd(s2);
57 }
58

```

7.8 Trie.h

```

1 template<int ALPHABET, int (*f)(char)>
2 class Trie {
3 public:
4     struct Node {
5         int answer = 0;
6         int next[ALPHABET];

```

```

7
8     Node() {
9         memset(next, -1, sizeof(next));
10    }
11 };
12
13 Trie() : Trie(vector<string>()) {}
14
15 Trie(const vector<string>& strs) {
16     clear();
17     for(const string& s : strs) {
18         insert(s);
19     }
20 }
21
22 void insert(const string& s, int p = 0) {
23     for(const char& c : s) {
24         int v = f(c);
25         if(nodes[p].next[v] == -1) {
26             nodes[p].next[v] = newNode();
27         }
28         p = nodes[p].next[v];
29     }
30     nodes[p].answer += 1;
31 }
32
33 int count(const string& s, int p = 0) {
34     for(const char& c : s) {
35         int v = f(c);
36         if(nodes[p].next[v] == -1) {
37             return 0;
38         }
39         p = nodes[p].next[v];
40     }
41     return nodes[p].answer;
42 }
43
44 void clear() {
45     nodes.clear();
46     newNode();
47 }
48
49 void reserve(int n) {
50     nodes.reserve(n);
51 }
52
53 private:
54     vector<Node> nodes;
55
56     inline int newNode() {
57         nodes.emplace_back();
58         return (int) nodes.size() - 1;
59     }
60 };
61

```

7.9 AhoCorasick.h

```

1 template<int ALPHABET, int (*)(char)>
2 class AhoCorasick {
3 public:
4     struct Node {
5         int fail = -1;

```

```

6         int answer = 0;
7         int next[ALPHABET];
8
9     Node() {
10         memset(next, -1, sizeof(next));
11     }
12 };
13
14 AhoCorasick() : AhoCorasick(vector<string>()) {}
15
16 AhoCorasick(const vector<string>& strs) {
17     clear();
18     for(const string& s : strs) {
19         query_index.push_back(insert(s));
20     }
21 }
22
23 int insert(const string& s) {
24     int p = 0;
25     for(int i = 0; i < (int) s.size(); ++i) {
26         int v = f(s[i]);
27         if(nodes[p].next[v] == -1) {
28             nodes[p].next[v] = newNode();
29         }
30         p = nodes[p].next[v];
31     }
32     return p;
33 }
34
35 vector<int> solve(const string& s) {
36     build_failure_all();
37     int p = 0;
38     for(int i = 0; i < (int) s.size(); ++i) {
39         int v = f(s[i]);
40         while(p > 0 && nodes[p].next[v] == -1)
41             p = nodes[p].fail;
42         if(nodes[p].next[v] != -1) {
43             p = nodes[p].next[v];
44             nodes[p].answer += 1;
45         }
46     }
47     for(int i = (int) que.size() - 1; i >= 0;
48     --i) {
49         nodes[nodes[que[i]].fail].answer +=
50         nodes[que[i]].answer;
51     }
52     vector<int> res(query_index.size());
53     for(int i = 0; i < (int) res.size(); ++i) {
54         res[i] = nodes[query_index[i]].answer;
55     }
56     return res;
57 }
58
59 void clear() {
60     nodes.clear();
61     que.clear();
62     query_index.clear();
63     newNode();
64     nodes[0].fail = 0;
65 }
66
67 void reserve(int n) {

```

```

67     nodes.reserve(n);
68 }
69
70 private:
71     vector<Node> nodes;
72     vector<int> que;
73     vector<int> query_index;
74
75     inline int newNode() {
76         nodes.emplace_back();
77         return (int) nodes.size() - 1;
78     }
79
80     void build_failure(int p) {
81         for(int i = 0; i < ALPHABET; ++i) {
82             if(nodes[p].next[i] != -1) {
83                 int tmp = nodes[p].fail;
84                 while(tmp > 0 && nodes[tmp].next[i]
↪ == -1) {
85                     tmp = nodes[tmp].fail;
86                 }
87                 if(nodes[tmp].next[i] !=
↪ nodes[p].next[i] && nodes[tmp].next[i] != -1) {
88                     tmp = nodes[tmp].next[i];
89                 }
90                 nodes[nodes[p].next[i]].fail = tmp;
91                 que.push_back(nodes[p].next[i]);
92             }
93         }
94     }
95
96     void build_failure_all() {
97         que.clear();
98         que.reserve(nodes.size());
99         que.push_back(0);
100         for(int i = 0; i < (int) que.size(); ++i) {
101             build_failure(que[i]);
102         }
103     }
104 };
105

```

8 Misc

8.1 Timer.h

```

1 const clock_t startTime = clock();
2 double getCurrentTime() {
3     return (double) (clock() - startTime) /
↪     CLOCKS_PER_SEC;
4 }
5

```

8.2 Random.h

```

1 class random_t {
2 public:
3     mt19937_64 rng;
4     unsigned long long seed;
5

```

```

6     random_t() :
↪     random_t(chrono::steady_clock::now().time_since_epoch().count()) {}
7
8     random_t(unsigned long long s) : rng(s),
↪     seed(s) {}
9
10    inline void set_seed(unsigned long long s) {
11        seed = s;
12        rng = mt19937_64(s);
13    }
14
15    inline void reset() {
16        set_seed(seed);
17    }
18
19    inline unsigned long long next() {
20        return uniform_int_distribution<unsigned
↪ long long>(0, ULLONG_MAX)(rng);
21    }
22
23    inline unsigned long long next(unsigned long
↪ long a) {
24        return next() % a;
25    }
26
27    inline unsigned long long next(unsigned long
↪ long a, unsigned long long b) {
28        return a + next(b - a + 1);
29    }
30
31    inline long double nextDouble() {
32        return ((unsigned int) next()) /
↪ 4294967295.0;
33    }
34
35    inline long double nextDouble(long double a) {
36        return nextDouble() * a;
37    }
38
39    inline long double nextDouble(long double a,
↪ long double b) {
40        return a + nextDouble() * (b - a);
41    }
42
43    template<class T>
44    void shuffle(vector<T>& a) {
45        for(int i = (int) a.size() - 1; i >= 0;
↪ --i) {
46            swap(a[i], a[next(i + 1)]);
47        }
48    }
49 };
50
51 random_t rnd;
52

```

8.3 Debug.h

```
1 const string NONE = "\033[m", RED =  
  ↪ "\033[0;32;31m", LIGHT_RED = "\033[1;31m",  
  ↪ GREEN = "\033[0;32;32m", LIGHT_GREEN =  
  ↪ "\033[1;32m", BLUE = "\033[0;32;34m",  
  ↪ LIGHT_BLUE = "\033[1;34m", DARK_GRAY =  
  ↪ "\033[1;30m", CYAN = "\033[0;36m", LIGHT_CYAN =  
  ↪ "\033[1;36m", PURPLE = "\033[0;35m",  
  ↪ LIGHT_PURPLE = "\033[1;35m", BROWN =  
  ↪ "\033[0;33m", YELLOW = "\033[1;33m", LIGHT_GRAY  
  ↪ = "\033[0;37m", WHITE = "\033[1;37m";  
2 template<class c> struct rge { c b, e; };  
3 template<class c> rge<c> range(c i, c j) { return  
  ↪ rge<c>{i, j}; }  
4 template<class c> auto dud(c* x)->decltype(cerr <<  
  ↪ *x, 0);  
5 template<class c> char dud(...);  
6 struct debug {  
7   #ifdef LOCAL  
8     ~debug() { cerr << endl; }  
9     template<class c> typename enable_if<sizeof  
  ↪ dud<c>(0) != 1, debug&::type operator<<(c i) {  
  ↪ cerr << boolalpha << i; return *this; }  
10    template<class c> typename enable_if<sizeof  
  ↪ dud<c>(0) == 1, debug&::type operator<<(c i) {  
  ↪ return *this << range(begin(i), end(i)); }  
11    template<class c, class b> debug&  
  ↪ operator<<(pair<b, c> d) { return *this << "("  
  ↪ << d.first << ", " << d.second << ")"; }  
12    template<class a, class b, class c> debug&  
  ↪ operator<<(tuple<a, b, c> tp) { return *this <<  
  ↪ "(" << get<0>(tp) << ", " << get<1>(tp) << ", "  
  ↪ << get<2>(tp) << ")"; };  
13    template<class a, class b, class c, class d>  
  ↪ debug& operator<<(tuple<a, b, c, d> tp) {  
  ↪ return *this << "(" << get<0>(tp) << ", " <<  
  ↪ get<1>(tp) << ", " << get<2>(tp) << ", " <<  
  ↪ get<3>(tp) << ")"; };  
14    template<class c> debug& operator<<(rge<c> d) {  
15      *this << "{";  
16      for(auto it = d.b; it != d.e; ++it) {  
17        *this << ", " + 2 * (it == d.b) << *it;  
18      }  
19      return *this << "}";  
20    }  
21    #else  
22    template<class c> debug& operator<<(const c&) {  
  ↪ return *this; }  
23    #endif  
24  };  
25 #define show(...) "" << LIGHT_RED << " [" << NONE  
  ↪ << #__VA_ARGS__ ": " << (__VA_ARGS__) <<  
  ↪ LIGHT_RED << "]" << NONE << ""  
26
```

```
4    sort(b.begin(), b.end());  
5    b.erase(unique(b.begin(), b.end()), b.end());  
6    vector<int> c(a.size());  
7    for(int i = 0; i < (int) a.size(); ++i) {  
8      c[i] = int(lower_bound(b.begin(), b.end(),  
  ↪ a[i]) - b.begin()) + OFFSET;  
9    }  
10   return c;  
11 }  
12  
13 template<class T>  
14 vector<int> unordered_compress(const vector<T>& a,  
  ↪ int OFFSET = 0) {  
15   int n = (int) a.size();  
16   hash_map<T, int> mapping;  
17   vector<int> b(n);  
18   for(int i = 0; i < n; ++i) {  
19     auto it = mapping.find(a[i]);  
20     if(it == mapping.end()) {  
21       b[i] = mapping[a[i]] = OFFSET;  
22       OFFSET += 1;  
23     } else {  
24       b[i] = it->second;  
25     }  
26   }  
27   return b;  
28 }  
29
```

8.4 Discrete.h

```
1 template<class T>  
2 vector<int> ordered_compress(const vector<T>& a,  
  ↪ int OFFSET = 0) {  
3   vector<T> b(a);
```