

# My Codebook

Felix Huang

September 9, 2022

## Contents

<b>1 Data-structures</b>	<b>1</b>	<b>7 String</b>	<b>23</b>
1.1 DSU.h . . . . .	1	7.1 SuffixArray.h . . . . .	23
1.2 Fenwick.h . . . . .	2	7.2 LCP.h . . . . .	25
1.3 HashMap.h . . . . .	2	7.3 KMP.h . . . . .	25
1.4 Segtree.h . . . . .	2	7.4 DynamicKMP.h . . . . .	25
1.5 LazySegtree.h . . . . .	3	7.5 Zfunc.h . . . . .	26
1.6 OrderStatisticTree.h . . . . .	5	7.6 RollingHash.h . . . . .	26
1.7 SparseTable.h . . . . .	5	7.7 Manacher.h . . . . .	26
1.8 ConvexHullTrick.h . . . . .	6	7.8 Trie.h . . . . .	27
1.9 Treap.h . . . . .	6	7.9 AhoCorasick.h . . . . .	27
<b>2 Combinatorial</b>	<b>7</b>	<b>8 Misc</b>	<b>28</b>
2.1 Combination.h . . . . .	7	8.1 Aliens.h . . . . .	28
2.2 CountInversions.h . . . . .	7	8.2 Timer.h . . . . .	28
<b>3 Number-theory</b>	<b>7</b>	8.3 Random.h . . . . .	29
3.1 ExtendGCD.h . . . . .	7	8.4 Debug.h . . . . .	29
3.2 InvGCD.h . . . . .	8	8.5 Discrete.h . . . . .	30
3.3 StaticModint.h . . . . .	8	8.6 Template.h . . . . .	30
3.4 DynamicModint.h . . . . .	9		
3.5 CRT.h . . . . .	11	<b>1 Data-structures</b>	
3.6 LinearSieve.h . . . . .	11	<b>1.1 DSU.h</b>	
3.7 ModInverses.h . . . . .	11		
3.8 PowMod.h . . . . .	12		
3.9 IsPrime.h . . . . .	12		
3.10 Factorize.h . . . . .	12		
3.11 PrimitiveRoot.h . . . . .	13		
3.12 FloorSum.h . . . . .	13		
<b>4 Numerical</b>	<b>14</b>		
4.1 Barrett.h . . . . .	14		
4.2 BitTransform.h . . . . .	14		
4.3 FFT.h . . . . .	15		
4.4 Poly.h . . . . .	16		
<b>5 Geometry</b>	<b>18</b>		
5.1 Point.h . . . . .	18		
5.2 ConvexHull.h . . . . .	19		
<b>6 Graph</b>	<b>20</b>		
6.1 LCA.h . . . . .	20		
6.2 HLD.h . . . . .	21		
6.3 TwoSat.h . . . . .	21		
6.4 Dinic.h . . . . .	21		
6.5 MCMF.h . . . . .	22		

```
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 }
```

```

26     }
27
28     inline int size(int u) {
29         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 // 0-based index
2 template<class T>
3 class fenwick {
4 public:
5     fenwick() : fenwick(0) {}
6
7     fenwick(int _n) : n(_n), data(_n) {}
8
9     void add(int p, T x) {
10         assert(0 <= p && p < n);
11         while(p < n) {
12             data[p] += x;
13             p |= (p + 1);
14         }
15     }
16
17     T get(int p) {
18         assert(0 <= p && p < n);
19         T res{};
20         while(p >= 0) {
21             res += data[p];
22             p = (p & (p + 1)) - 1;

```

```

23     }
24     return res;
25 }
26
27 T sum(int l, int r) {
28     return get(r) - (l ? get(l - 1) : T{});
29 }
30
31 private:
32     int n;
33     vector<T> data;
34 };
35

```

## 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 =
14        chrono::steady_clock::now().time_since_epoch().count();
15        return splitmix64(x + FIXED_RANDOM);
16    };
17
18    template<class T, class U, class H =
→ splitmix64_hash> using hash_map =
19    gp_hash_table<T, U, H>;
20    template<class T, class H = splitmix64_hash> using
→ hash_set = hash_map<T, null_type, H>;

```

## 1.4 Segtree.h

```

1 template<class T, T (*e)(), T (*op)(T, T)>
2 class segtree {
3 public:
4     segtree() : segtree(0) {}
5
6     segtree(int _n) : segtree(vector<T>(_n, e()))
→ {}
7
8     segtree(const vector<T>& a) : n(int(a.size())) {
9         log = 31 - __builtin_clz(2 * n - 1);
10        size = 1 << log;
11        st.resize(size << 1, e());
12        for(int i = 0; i < n; ++i) {
13            st[size + i] = a[i];
14        }

```

```

15     for(int i = size - 1; i; --i) {
16         update(i);
17     }
18 }
19
20 void set(int p, T val) {
21     assert(0 <= p && p < n);
22     p += size;
23     st[p] = val;
24     for(int i = 1; i <= log; ++i) {
25         update(p >> i);
26     }
27 }
28
29 inline T get(int p) const {
30     assert(0 <= p && p < n);
31     return st[p + size];
32 }
33
34 inline T operator[](int p) const {
35     return get(p);
36 }
37
38 T prod(int l, int r) const {
39     assert(0 <= l && l <= r && r <= n);
40     T sml = e(), smr = e();
41     l += size;
42     r += size;
43     while(l < r) {
44         if(l & 1) {
45             sml = op(sml, st[l++]);
46         }
47         if(r & 1) {
48             smr = op(st[--r], smr);
49         }
50         l >>= 1;
51         r >>= 1;
52     }
53     return op(sml, smr);
54 }
55
56 inline T all_prod() const { return st[1]; }
57
58 template<bool (*f)(T)> int max_right(int l)
59 → const {
60     return max_right(l, [](T x) { return f(x);
61 → });
62 }
63
64 template<class F> int max_right(int l, F f)
65 → const {
66     assert(0 <= l && l <= n);
67     assert(f(e()));
68     if(l == n) {
69         return n;
70     }
71     l += size;
72     T sm = e();
73     do {
74         while(!(l & 1)) {
75             l >>= 1;
76         }
77         if(!f(op(sm, st[l]))) {
78             while(l < size) {
79                 l <<= 1;

```

```

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

---

## 1.5 LazySegtree.h

---

```

1 template<class S,
2         S (*e)(),
3         S (*op)(S, S),
4         class F,
```

```

5         F (*id)(),
6         S (*mapping)(F, S),
7         F (*composition)(F, F)>
8 class lazy_segtree {
9 public:
10     lazy_segtree() : lazy_segtree(0) {}
11
12     explicit lazy_segtree(int _n) :
13     ↪ lazy_segtree(vector<S>(_n, e())) {}
14
15     explicit lazy_segtree(const vector<S>& v) :
16     ↪ n(int(v.size())) {
17         log = 31 - __builtin_clz(2 * n - 1);
18         size = 1 << log;
19         d = vector<S>(size << 1, e());
20         lz = vector<F>(size, id());
21         for(int i = 0; i < n; i++) {
22             d[size + i] = v[i];
23         }
24         for(int i = size - 1; i; --i) {
25             update(i);
26         }
27     }
28
29     void set(int p, S x) {
30         assert(0 <= p && p < n);
31         p += size;
32         for(int i = log; i; --i) {
33             push(p >> i);
34         }
35         d[p] = x;
36         for(int i = 1; i <= log; ++i) {
37             update(p >> i);
38         }
39     }
40
41     S get(int p) {
42         assert(0 <= p && p < n);
43         p += size;
44         for(int i = log; i; i--) {
45             push(p >> i);
46         }
47         return d[p];
48     }
49
50     S operator[](int p) {
51         return get(p);
52     }
53
54     S prod(int l, int r) {
55         assert(0 <= l && l <= r && r <= n);
56         if(l == r) {
57             return e();
58         }
59         l += size;
60         r += size;
61         for(int i = log; i; i--) {
62             if(((l >> i) << i) != 1) {
63                 push(l >> i);
64             }
65             if(((r >> i) << i) != r) {
66                 push(r >> i);
67             }
68         }
69         S sml = e(), smr = e();

```

```

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

```

```

133     template<bool (*g)(S)> int max_right(int l) {
134         return max_right(l, [] (S x) { return g(x);
135     });
136     }
137
138     template<class G> int max_right(int l, G g) {
139         assert(0 <= l && l <= n);
140         assert(g(e()));
141         if(l == n) {
142             return n;
143         }
144         l += size;
145         for(int i = log; i; i--) {
146             push(l >> i);
147         }
148         S sm = e();
149         do {
150             while(!(l & 1)) {
151                 l >>= 1;
152             }
153             if(!g(op(sm, d[l]))) {
154                 while(l < size) {
155                     push(l);
156                     l <<= 1;
157                     if(g(op(sm, d[l]))) {
158                         sm = op(sm, d[l]);
159                         l++;
160                     }
161                 }
162                 return l - size;
163             }
164             sm = op(sm, d[l]);
165             l++;
166         } while((l & -l) != 1);
167         return n;
168     }
169
170     template<bool (*g)(S)> int min_left(int r) {
171         return min_left(r, [] (S x) { return g(x);
172     });
173     }
174
175     template<class G> int min_left(int r, G g) {
176         assert(0 <= r && r <= n);
177         assert(g(e()));
178         if(r == 0) {
179             return 0;
180         }
181         r += size;
182         for(int i = log; i >= 1; i--) {
183             push((r - 1) >> i);
184         }
185         S sm = e();
186         do {
187             r--;
188             while(r > 1 && (r & 1)) {
189                 r >>= 1;
190             }
191             if(!g(op(d[r], sm))) {
192                 while(r < size) {
193                     push(r);
194                     r = r << 1 | 1;
195                     if(g(op(d[r], sm))) {
196                         sm = op(d[r], sm);

```

```

196             r--;
197         }
198     }
199     return r + 1 - size;
200 }
201 sm = op(d[r], sm);
202 } while((r & -r) != r);
203 return 0;
204 }
205
206 private:
207     int n, size, log;
208     vector<S> d;
209     vector<F> lz;
210
211     inline void update(int k) { d[k] = op(d[k <<
212     1], d[k << 1 | 1]); }
213
214     void all_apply(int k, F f) {
215         d[k] = mapping(f, d[k]);
216         if(k < size) {
217             lz[k] = composition(f, lz[k]);
218         }
219
220     void push(int k) {
221         all_apply(k << 1, lz[k]);
222         all_apply(k << 1 | 1, lz[k]);
223         lz[k] = id();
224     }
225 };
226

```

## 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
5     ordered_set = tree<T, null_type, Comp,
6     rb_tree_tag,
7     tree_order_statistics_node_update>;
8 template<class T> using ordered_multiset =
9     ordered_set<T, less_equal<T>>;
10 // Use `s.erase(s.find(x))` when using
11     `ordered_multiset`

```

## 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     ↪ {
15         mat[j][i] = op(mat[j - 1][i], mat[j
16     ↪ - 1][i + (1 << (j - 1))]);
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 inline T operator[](int p) const {
27     assert(0 <= p && p < n);
28     return mat[0][p];
29 }
30 private:
31     int n;
32     vector<vector<T>> mat;
33 };
34

```

## 1.8 ConvexHullTrick.h

```

1 struct Line_t {
2     mutable long long k, m, p;
3
4     inline bool operator<(const Line_t& o) const {
5     ↪ return k < o.k; }
6     inline bool operator<(long long x) const {
7     ↪ return p < x; }
8 };
9
10 // returns maximum (with minimum use negative
11 ↪ coefficient and constant)
12 struct CHT : multiset<Line_t, less<>> {
13     // (for doubles, use INF = 1/.0, div(a,b) =
14     ↪ a/b)
15     static const long long INF = LLONG_MAX;
16     long long div(long long a, long long b) { //
17     ↪ floored division
18         return a / b - ((a ^ b) < 0 && a % b);
19     }
20
21     bool isect(iterator x, iterator y) {
22         if(y == end()) {
23             x->p = INF;
24             return 0;
25         }
26         if(x->k == y->k) {
27             x->p = (x->m > y->m ? INF : -INF);
28         } else {
29             x->p = div(y->m - x->m, x->k - y->k);
30         }
31         return x->p >= y->p;
32     }
33
34     void insert_line(long long k, long long m) {
35

```

```

30         auto z = insert({k, m, 0}), y = z++, x = y;
31         while(isect(y, z)) {
32             z = erase(z);
33         }
34         if(x != begin() && isect(--x, y)) {
35             isect(x, y = erase(y));
36         }
37         while((y = x) != begin() && (--x->p >=
38     ↪ y->p) {
39             isect(x, erase(y));
40         }
41     }
42     long long eval(long long x) {
43         assert(!empty());
44         auto l = *lower_bound(x);
45         return l.k * x + l.m;
46     }
47 };
48

```

## 1.9 Treap.h

```

1 mt19937_64 rng(48763);
2
3 struct Node {
4     long long val;
5     long long sum;
6     bool rev;
7     int size;
8     int pri;
9
10     Node* l;
11     Node* r;
12
13     Node(long long x) : val(x), sum(x), rev(false),
14     ↪ size(1), pri(rng()), l(NULL), r(NULL) {}
15 };
16
17 inline int size(Node*& v) {
18     return (v ? v->size : 0);
19 }
20
21 void pull(Node*& v) {
22     v->size = 1 + size(v->l) + size(v->r);
23     v->sum = v->val + (v->l ? v->l->sum : 0) +
24     ↪ (v->r ? v->r->sum : 0);
25 }
26
27 void push(Node*& v) {
28     if(v->rev) {
29         swap(v->l, v->r);
30         if(v->l) {
31             v->l->rev = !v->l->rev;
32         }
33         if(v->r) {
34             v->r->rev = !v->r->rev;
35         }
36         v->rev = false;
37     }
38 }
39
40 Node* merge(Node* a, Node* b) {
41

```

```

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 // @return the number of inversions s.t  $i < j$ ,
↪  $a_i > a_j$ 
2 template<class T>
3 long long countInversions(const vector<T>& a) {
4     int n = (int) a.size();
5     auto b = a;
6     sort(b.begin(), b.end());
7     b.erase(unique(b.begin(), b.end()), b.end());
8     fenwick<int> fenw((int) b.size() + 1);
9     long long ans = 0;
10    for(int i = 0; i < n; ++i) {
11        int x = lower_bound(b.begin(), b.end(),
↪ a[i]) - b.begin();
12        ans += fenw.sum(x + 1, (int) b.size());
13        fenw.add(x, 1);
14    }
15    return ans;
16 }
17

```

---

## 3 Number-theory

### 3.1 ExtendGCD.h

```

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

## 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    }

```

```

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 // @return
2 //     remainder, modulo
3 //     or
4 //     0,0 if do not exist
5 pair<long long, long long> crt(const vector<long
    ↪ long>& r, const vector<long long>& m) {
6     assert(r.size() == m.size());
7     int n = (int) r.size();
8     // Contracts: 0 <= r0 < m0
9     long long r0 = 0, m0 = 1;
10    for(int i = 0; i < n; i++) {
11        assert(1 <= m[i]);
12        long long r1 = r[i] % m[i];
13        if(r1 < 0) r1 += m[i];
14        long long m1 = m[i];
15        if(m0 < m1) {
16            swap(r0, r1);
17            swap(m0, m1);
18        }
19        if(m0 % m1 == 0) {
20            if(r0 % m1 != r1) return {0, 0};
21            continue;

```

```

22    }
23    long long g, im;
24    tie(g, im) = inv_gcd(m0, m1);
25
26    long long u1 = (m1 / g);
27    if((r1 - r0) % g) return {0, 0};
28
29    long long x = (r1 - r0) / g % u1 * im % u1;
30
31    r0 += x * m0;
32    m0 *= u1;
33    if(r0 < 0) r0 += m0;
34    }
35    return {r0, m0};
36 }
37

```

### 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 }
34

```

### 3.7 ModInverses.h

```

1 // @return array A of length N s.t
2 //      $i \cdot A_i = 1 \pmod{m}$ 
3 vector<int> mod_inverse(int m, int n = -1) {
4     assert(n < m);

```

```

5     if(n == -1) {
6         n = m - 1;
7     }
8     vector<int> inv(n + 1);
9     inv[0] = inv[1] = 1;
10    for(int i = 2; i <= n; ++i) {
11        inv[i] = m - (long long) (m / i) * inv[m %
↪ i] % m;
12    }
13    return inv;
14 }
15

```

### 3.8 PowMod.h

```

1 // @param 0 ≤ n
2 // @param 1 ≤ m
3 // @return xn (mod 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

```

```

22 }
23 return true;
24 }
25 template<int n> constexpr bool is_prime =
↪ is_prime_constexpr(n);
26

```

### 3.10 Factorize.h

```

1 template<class T>
2 vector<pair<T, int>> MergeFactors(const
↪ vector<pair<T, int>>& a, const vector<pair<T,
↪ int>>& b) {
3     vector<pair<T, int>> c;
4     int i = 0, j = 0;
5     while(i < (int) a.size() || j < (int) b.size())
↪ {
6         if(i < (int) a.size() && j < (int) b.size()
↪ && a[i].first == b[j].first) {
7             c.emplace_back(a[i].first, a[i].second
↪ + b[j].second);
8             ++i, ++j;
9             continue;
10        }
11        if(j == (int) b.size() || (i < (int)
↪ a.size() && a[i].first < b[j].first)) {
12            c.push_back(a[i++]);
13        } else {
14            c.push_back(b[j++]);
15        }
16    }
17    return c;
18 }
19
20 template<class T>
21 vector<pair<T, int>> RhoC(const T& n, const T& c) {
22     if(n <= 1) {
23         return {};
24     }
25     if(n % 2 == 0) {
26         return MergeFactors({{2, 1}}, RhoC(n / 2,
↪ c));
27     }
28     if(is_prime_constexpr(n)) {
29         return {{n, 1}};
30     }
31     T x = 2;
32     T saved = 2;
33     T p = 1;
34     T lam = 1;
35     while(true) {
36         x = (x * x % n + c) % n;
37         T g = __gcd(((x - saved) + n) % n, n);
38         if(g != 1) {
39             return MergeFactors(RhoC(g, c + 1),
↪ RhoC(n / g, c + 1));
40         }
41         if(p == lam) {
42             saved = x;
43             p <<= 1;
44             lam = 0;
45         }
46         lam += 1;
47     }
48 }
49

```

```

47     }
48     return {};
49 }
50
51 template<class T>
52 vector<pair<T, int>> Factorize(T n) {
53     if(n <= 1) {
54         return {};
55     }
56     return RhoC(n, T(1));
57 }
58
59 template<class T>
60 vector<T> BuildDivisorsFromFactors(const
↪ vector<pair<T, int>>& factors) {
61     int total = 1;
62     for(int i = 0; i < (int) factors.size(); ++i) {
63         total *= factors[i].second + 1;
64     }
65     vector<T> divisors;
66     divisors.reserve(total);
67     divisors.push_back(1);
68     for(auto& [p, cnt] : factors) {
69         int sz = (int) divisors.size();
70         for(int i = 0; i < sz; ++i) {
71             T cur = divisors[i];
72             for(int j = 0; j < cnt; ++j) {
73                 cur *= p;
74                 divisors.push_back(cur);
75             }
76         }
77     }
78     // sort(divisors.begin(), divisors.end());
79     return divisors;
80 }
81

```

### 3.11 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    return divs[cnt-1];
27 }

```

```

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

```

### 3.12 FloorSum.h

```

1 // @param n < 232
2 // @param 1 ≤ m < 232
3 // @return sum_{i=0}^{n-1} ⌊ floor(frac{ai + b}{m}) ⌋
↪ ⌊ rfloor(pmod{264}}
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    }
41 }

```

```

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:
3 ↪ https://en.wikipedia.org/wiki/Barrett\_reduction
4 class barrett {
5 public:
6     unsigned int m;
7     unsigned long long im;
8
9     explicit barrett(unsigned int _m) : m(_m),
10 ↪ im((unsigned long long)(-1) / _m + 1) {}
11
12     unsigned int umod() const { return m; }
13
14     unsigned int mul(unsigned int a, unsigned int
15 ↪ b) const {
16         unsigned long long z = a;
17         z *= b;
18 #ifdef _MSC_VER
19         unsigned long long x;
20         _umul128(z, im, &x);
21 #else
22         unsigned long long x = (unsigned long
23 ↪ long)(((unsigned __int128)(z) * im) >> 64);
24 #endif
25         unsigned int v = (unsigned int)(z - x * m);
26         if(m <= v) {
27             v += m;
28         }
29         return v;
30     }
31 };
32

```

### 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
61 ↪ + j + k]);
62                 a[j + k] = x + y;
63                 a[i + j + k] = x - y;
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) =
78     ↪ k.
79     // 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) =
93 // k.
94 // Complexity: O(n log n)
95 template<class T>
96 vector<T> AndConvolution(vector<T> a, vector<T> b)
97 {
98     const int n = (int) a.size();
99     assert(n == (int) b.size());
100    AndTransform(a);
101    AndTransform(b);
102    for(int i = 0; i < n; ++i) {
103        a[i] *= b[i];
104    }
105    AndInvTransform(a);
106    return a;
107 }
108
109 // Compute c[k] = sum(a[i] * b[j]) for (i xor j) =
110 // k.
111 // Complexity: O(n log n)
112 template<class T>
113 vector<T> XorConvolution(vector<T> a, vector<T> b)
114 {
115     const int n = (int) a.size();
116     assert(n == (int) b.size());
117     XorTransform(a);
118     XorTransform(b);
119     for (int i = 0; i < n; ++i) {
120         a[i] *= b[i];
121     }
122     XorInvTransform(a);
123     return a;
124 }
125
126 template<class T>
127 void ZetaTransform(vector<T>& a) {
128     OrTransform(a);
129 }
130
131 template<class T>
132 void MobiusTransform(vector<T>& a) {
133     OrInvTransform(a);
134 }
135
136 template<class T>
137 vector<T> SubsetSumConvolution(const vector<T>& f,
138                               const vector<T>& g) {
139     const int n = (int) f.size();
140     assert(n == (int) g.size());
141     assert((n & -n) == n);
142     const int N = __lg(n);
143     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] =
142         f[mask];
143         ghat[__builtin_popcount(mask)][mask] =
144         g[mask];
145     }
146     for(int i = 0; i <= N; ++i) {
147         ZetaTransform(fhat[i]);
148         ZetaTransform(ghat[i]);
149     }
150     vector<vector<T>> h(N + 1, vector<T>(n));
151     for(int mask = 0; mask < n; ++mask) {
152         for(int i = 0; i <= N; ++i) {
153             for(int j = 0; j <= i; ++j) {
154                 h[i][mask] += fhat[j][mask] *
155                 ghat[i - j][mask];
156             }
157         }
158     }
159     for(int i = 0; i <= N; ++i) {
160         MobiusTransform(h[i]);
161     }
162     vector<T> result(n);
163     for(int mask = 0; mask < n; ++mask) {
164         result[mask] =
165         h[__builtin_popcount(mask)][mask];
166     }
167     return result;
168 }

```

### 4.3 FFT.h

```

1 // Fast-Fourier-Transform
2 using cd = complex<double>;
3
4 const double PI = acos(-1);
5
6 void fft(vector<cd>& a, bool inv) {
7     int n = (int) a.size();
8     for(int i = 1, j = 0; i < n; ++i) {
9         int bit = n >> 1;
10        for(; j & bit; bit >>= 1) {
11            j ^= bit;
12        }
13        j ^= bit;
14        if(i < j) {
15            swap(a[i], a[j]);
16        }
17    }
18    for(int len = 2; len <= n; len <= 1) {
19        const double ang = 2 * PI / len * (inv ? -1
20        : +1);
21        cd rot(cos(ang), sin(ang));
22        for(int i = 0; i < n; i += len) {
23            cd w(1);
24            for(int j = 0; j < len / 2; ++j) {
25                cd u = a[i + j], v = a[i + j + len
26                / 2] * w;
27                a[i + j] = u + v;
28                a[i + j + len / 2] = u - v;
29                w *= rot;

```



```

28     }
29 }
30 }
31 if(inv) {
32     for(auto& x : a) {
33         x /= n;
34     }
35 }
36 }
37

```

## 4.4 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 at(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.at(i) + b.at(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.at(i) - b.at(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 Poly sqrt(int m) const {
243     Poly<T> x{1};
244     int k = 1;
245     while(k < m) {
246         k *= 2;
247         x = (x + (modxk(k) *
→ x.inv(k)).modxk(k)) * T((mod() + 1) / 2);
248     }
249     return x.modxk(m);
250 }

251 Poly multT(Poly b) const {
252     if(b.size() == 0) {
253         return Poly<T>();
254     }
255     int n = b.size();
256     reverse(b.coeff.begin(), b.coeff.end());
257     return ((*this) * b).divxk(n - 1);
258 }

259 vector<T> eval(vector<T> x) const {
260     if(size() == 0) {
261         return vector<T>(x.size(), 0);
262     }
263     const int n = max((int) x.size(), size());
264     vector<Poly<T>> q(4 * n);
265     vector<T> ans(x.size());
266     x.resize(n);
267     function<void(int, int, int)> build =
→ [&](int p, int l, int r) {
268         if(r - l == 1) {
269             q[p] = Poly{1, -x[l]};
270         } else {
271             int m = (l + r) / 2;
272             build(2 * p, l, m);
273             build(2 * p + 1, m, r);
274             q[p] = q[2 * p] * q[2 * p + 1];
275         }
276     };
277     build(1, 0, n);
278     function<void(int, int, int, const Poly&)>
→ work = [&](int p, int l, int r, const Poly&
→ num) {
279         if(r - l == 1) {
280             if(l < (int) ans.size()) {
281                 ans[l] = num[0];
282             }
283         } else {
284             int m = (l + r) / 2;
285             work(2 * p, l, m, num.mulT(q[2 * p
→ + 1]).modxk(m - 1));
286             work(2 * p + 1, m, r, num.mulT(q[2
→ * p]).modxk(r - m));

```

```

287         }
288     };
289     work(1, 0, n, mulT(q[1].inv(n)));
290     return ans;
291 }

292 private:
293     vector<T> coeff;
294     static vector<T> roots;
295 };
296
297 template<class T> vector<T> Poly<T>::roots{0, 1};
298
299
300
301
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 // @return the points of the convex hull in
→ clock-wise order
2 template<class T>
3 vector<Point<T>> ConvexHull(vector<Point<T>>
→ points) {
4     const int n = (int) points.size();
5     sort(points.begin(), points.end(), [](const
→ Point<T>& a, const Point<T>& b) {
6         if(a.x == b.x) {
7             return a.y < b.y;
8         }
9         return a.x < b.x;
10    });
11    auto build = [&]() {
12        vector<Point<T>> upper;
13        upper.push_back(points[0]);
14        upper.push_back(points[1]);
15        for(int i = 2; i < n; ++i) {
16            while((int) upper.size() >= 2) {
17                if(cross(upper.end() [-1] -
→ upper.end() [-2], points[i] - upper.end() [-1]) >
→ 0) {
18                    upper.pop_back();
19                } else {
20                    break;
21                }
22            }
23            upper.push_back(points[i]);
24        }
25        return upper;
26    };
27    vector<Point<T>> upper = build();
28    reverse(points.begin(), points.end());
29    vector<Point<T>> lower = build();
30    lower.pop_back();
31    upper.insert(upper.end(), lower.begin() + 1,
→ lower.end());

```

```

32     return upper;
33 }
34

```

## 6 Graph

### 6.1 LCA.h

```

1 class LCA {
2 public:
3     LCA() : LCA(0) {}
4     LCA(int _n) : n(_n), g(_n) {}
5
6     static pair<int, int> __lca_op(pair<int, int>
→ a, pair<int, int> b) {
7         return min(a, b);
8     }
9
10    void add_edge(int u, int v) {
11        assert(0 <= u && u < n);
12        assert(0 <= v && v < n);
13        g[u].push_back(v);
14        g[v].push_back(u);
15    }
16
17    void build(int root = 0) {
18        assert(0 <= root && root < n);
19        depth.assign(n, 0);
20        parent.assign(n, -1);
21        subtree_size.assign(n, 1);
22        euler.reserve(2 * n - 1);
23        first_occurrence.assign(n, 0);
24        tour_list.reserve(n);
25        tour_start.assign(n, 0);
26        function<void(int, int, int)> dfs = [&](int
→ u, int p, int d) {
27            parent[u] = p;
28            depth[u] = d;
29            first_occurrence[u] = (int)
→ euler.size();
30            euler.push_back(u);
31            pair<int, int> heavy = {-1, -1};
32            for(auto& v : g[u]) {
33                if(v == p) {
34                    continue;
35                }
36                dfs(v, u, d + 1);
37                subtree_size[u] += subtree_size[v];
38                if(subtree_size[v] > heavy.first) {
39                    heavy = {subtree_size[v], v};
40                }
41                euler.push_back(u);
42            }
43            sort(g[u].begin(), g[u].end(), [&](int
→ a, int b) {
44                return subtree_size[a] >
→ subtree_size[b];
45            });
46        };
47        dfs(root, -1, 0);
48        heavy_root.assign(n, 0);

```

```

49        function<void(int, bool)> dfs2 = [&](int u,
→ bool is_heavy) {
50            tour_start[u] = (int) tour_list.size();
51            tour_list.push_back(u);
52            heavy_root[u] = (is_heavy ?
→ heavy_root[parent[u]] : u);
53            bool heavy = true;
54            for(auto& v : g[u]) {
55                if(v == parent[u]) {
56                    continue;
57                }
58                dfs2(v, heavy);
59                heavy = false;
60            }
61        };
62        dfs2(root, false);
63        {
64            vector<pair<int, int>> route;
65            route.reserve((int) euler.size());
66            for(auto& u : euler) {
67                route.emplace_back(depth[u], u);
68            }
69            st = sparse_table<pair<int, int>,
→ __lca_op>(route);
70        }
71    }
72
73    inline int dist(int u, int v) const {
74        return depth[u] + depth[v] - 2 *
→ depth[lca(u, v)];
75    }
76
77    pair<int, array<int, 2>> get_diameter() const {
78        pair<int, int> u_max = {-1, -1};
79        pair<int, int> ux_max = {-1, -1};
80        pair<int, array<int, 2>> uxv_max = {-1,
→ {-1, -1}};
81        for(int u : euler) {
82            u_max = max(u_max, {depth[u], u});
83            ux_max = max(ux_max, {u_max.first - 2 *
→ depth[u], u_max.second});
84            uxv_max = max(uxv_max, {ux_max.first +
→ depth[u], {ux_max.second, u}});
85        }
86        return uxv_max;
87    }
88
89    inline int kth_ancestor(int u, int k) const {
90        if(depth[u] < k) {
91            return -1;
92        }
93        while(k > 0) {
94            int root = heavy_root[u];
95            if(depth[root] <= depth[u] - k) {
96                return tour_list[tour_start[u] -
→ k];
97            }
98            k -= depth[u] - depth[root] + 1;
99            u = parent[root];
100        }
101        return u;
102    }
103
104    inline int kth_node_on_path(int a, int b, int
→ k) const {

```

```

105     int z = lca(a, b);
106     int fi = depth[a] - depth[z];
107     int se = depth[b] - depth[z];
108     assert(0 <= k && k <= fi + se);
109     if(k < fi) {
110         return kth_ancestor(a, k);
111     } else {
112         return kth_ancestor(b, fi + se - k);
113     }
114 }
115
116 int lca(int u, int v) const {
117     assert(0 <= u && u < n);
118     assert(0 <= v && v < n);
119     int l = first_occurrence[u];
120     int r = first_occurrence[v];
121     return st.prod(min(l, r), max(l,
↪ r)).second;
122 }
123
124 public:
125     int n;
126     vector<vector<int>> g;
127     vector<int> parent;
128     vector<int> depth;
129     vector<int> subtree_size;
130
131 protected:
132     vector<int> euler;
133     vector<int> first_occurrence;
134     vector<int> tour_list;
135     vector<int> tour_start;
136     vector<int> heavy_root;
137     sparse_table<pair<int, int>, __lca_op> st;
138 };
139

```

## 6.2 HLD.h

```

1 class HLD : LCA {
2 public:
3     using LCA::add_edge;
4     using LCA::build;
5     using LCA::dist;
6     using LCA::get_diameter;
7     using LCA::kth_ancestor;
8     using LCA::kth_node_on_path;
9     using LCA::lca;
10
11     HLD() : HLD(0) {}
12     HLD(int _n) : LCA(_n) {}
13
14     inline int get(int u) const {
15         return tour_start[u];
16     }
17
18     // return path[u,...,p] where p is an ancestor of u
19     vector<pair<int, int>> path_up(int u, int p)
↪ const {
20         vector<pair<int, int>> seg;
21         while(heavy_root[u] != heavy_root[p]) {
22             seg.emplace_back(get(heavy_root[u]),
↪ get(u) + 1);

```

```

23         u = parent[heavy_root[u]];
24     }
25     // idp is smaller than idu but we don't want
↪ idp
26     seg.emplace_back(get(p) + 1, get(u) + 1);
27     return seg;
28 }
29
30 vector<pair<int, int>> path(int u, int v) const
↪ {
31     int z = lca(u, v);
32     auto lhs = path_up(u, z);
33     auto rhs = path_up(v, z);
34     lhs.emplace_back(get(z), get(z) + 1);
35     lhs.insert(lhs.end(), rhs.begin(),
↪ rhs.end());
36     return lhs;
37 }
38 };
39

```

## 6.3 TwoSat.h

```

1 // Under construction

```

## 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 - 1 != end_r - r) {

```

```

149         same = false;
150     } else {
151         while(1 < end_l) {
152             if(s[l] != s[r]) {
153                 break;
154             }
155             l++;
156             r++;
157         }
158         if(1 == 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        void push(char c) {
13            int v = f(c);
14            dp.emplace_back();
15            dp.back()[v] = (int) dp.size();
16            if(p.empty()) {
17                p.push_back(0);
18                return;
19            }
20            int i = (int) p.size();
21            for(int j = 0; j < ALPHABET; ++j) {
22                if(j == v) {
23                    p.push_back(dp[p[i - 1]][j]);
24                } else {
25                    dp.back()[j] = dp[p[i - 1]][j];
26                }
27            }
28        }
29
30        void pop() {
31            p.pop_back();
32            dp.pop_back();
33        }
34
35        int query() const {
36            return p.back();
37        }
38
39        vector<int> query_all() const {
40            return p;
41        }
42
43        void reserve(int sz) {
44            p.reserve(sz);
45

```

```

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]] ==
↪ a[z[i]]) {
10             z[i] += 1;
11         }
12         if(i + z[i] > j + z[j]) {
13             j = i;
14         }
15     }
16     return z;
17 }
18
19 vector<int> z_algorithm(const string& s) {
20     vector<int> s2(s.begin(), s.end());
21     return z_algorithm(s2);
22 }
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
↪ m) {
4      x %= m;
5      if(x < 0) {
6          x += m;
7      }
8      return x;
9  }
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 void insert(const string& s, int p = 0) {
22     for(const char& c : s) {
23         int v = f(c);
24         if(nodes[p].next[v] == -1) {
25             nodes[p].next[v] = newNode();
26         }
27         p = nodes[p].next[v];
28     }
29     nodes[p].answer += 1;
30 }
31
32 int count(const string& s, int p = 0) {
33     for(const char& c : s) {
34         int v = f(c);
35         if(nodes[p].next[v] == -1) {
36             return 0;
37         }
38         p = nodes[p].next[v];
39     }
40     return nodes[p].answer;
41 }
42
43 void clear() {
44     nodes.clear();
45     newNode();
46 }
47
48 void reserve(int n) {
49     nodes.reserve(n);
50 }
51
52 private:
53     vector<Node> nodes;
54
55     inline int newNode() {
56         nodes.emplace_back();
57         return (int) nodes.size() - 1;
58     }
59 }
60 };
61

```

## 7.9 AhoCorasick.h

```

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

```

## 8 Misc

### 8.1 Aliens.h

```

1 // find minimum
2 int aliens(int l, int r, int k) {
3     while(l < r) {
4         int m = l + (r - l) / 2;
5         auto [score, op] = f(m);
6         if(op == k) {
7             return score - m * k;
8         }
9         if(op < k) {
10             r = m;
11         } else {
12             l = m + 1;
13         }
14     }
15     return f(l).first - l * k;
16 }

```

### 8.2 Timer.h

```

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

```

---

```

1 class random_t {
2 public:
3     mt19937_64 rng;
4     unsigned long long seed;
5
6     random_t() :
7     ↪ random_t(chrono::steady_clock::now().time_since_epoch().count())
8     ↪ {}
9
10    random_t(unsigned long long s) : rng(s),
11    ↪ seed(s) {}
12
13    inline void set_seed(unsigned long long s) {
14        seed = s;
15        rng = mt19937_64(s);
16    }
17
18    inline void reset() {
19        set_seed(seed);
20    }
21
22    inline unsigned long long next() {
23        ↪ return uniform_int_distribution<unsigned
24        ↪ long long>(0, ULLONG_MAX)(rng);
25    }
26
27    inline unsigned long long next(unsigned long
28    ↪ long a) {
29        ↪ return uniform_int_distribution<unsigned
30        ↪ long long>(0, a - 1)(rng);
31    }
32
33    inline unsigned long long next(unsigned long
34    ↪ long a, unsigned long long b) {
35        ↪ return uniform_int_distribution<unsigned
36        ↪ long long>(a, b)(rng);
37    }
38
39    inline long double nextDouble() {
40        ↪ return uniform_real_distribution<long
41        ↪ double>(0.0, 1.0)(rng);
42    }
43
44    inline long double nextDouble(long double a) {
45        ↪ return nextDouble() * a;
46    }
47
48    inline long double nextDouble(long double a,
49    ↪ long double b) {
50        ↪ return uniform_real_distribution<long
51        ↪ double>(a, b)(rng);
52    }
53
54    template<class T>
55    void shuffle(vector<T>& a) {
56        ↪ for(int i = (int) a.size() - 1; i >= 0;
57        ↪ --i) {
58            ↪ swap(a[i], a[next(i + 1)]);
59        }
60    }
61 };

```

## 8.4 Debug.h

---

```

1 const string NONE = "\033[m", RED =
2     ↪ "\033[0;32;31m", LIGHT_RED = "\033[1;31m",
3     ↪ GREEN = "\033[0;32;32m", LIGHT_GREEN =
4     ↪ "\033[1;32m", BLUE = "\033[0;32;34m",
5     ↪ LIGHT_BLUE = "\033[1;34m", DARK_GRAY =
6     ↪ "\033[1;30m", CYAN = "\033[0;36m", LIGHT_CYAN =
7     ↪ "\033[1;36m", PURPLE = "\033[0;35m",
8     ↪ LIGHT_PURPLE = "\033[1;35m", BROWN =
9     ↪ "\033[0;33m", YELLOW = "\033[1;33m", LIGHT_GRAY
10    ↪ = "\033[0;37m", WHITE = "\033[1;37m";
11 template<class c> struct rge { c b, e; };
12 template<class c> rge<c> range(c i, c j) { return
13     ↪ rge<c>{i, j}; }
14 template<class c> auto dud(c* x)->decltype(cerr <<
15     ↪ *x, 0);
16 template<class c> char dud(...);
17 struct debug {
18     #ifdef LOCAL
19     ~debug() { cerr << endl; }
20     template<class c> typename enable_if<sizeof
21     ↪ dud<c>(0) != 1, debug&>::type operator<<(c i) {
22     ↪ cerr << boolalpha << i; return *this; }
23     template<class c> typename enable_if<sizeof
24     ↪ dud<c>(0) == 1, debug&>::type operator<<(c i) {
25     ↪ return *this << range(begin(i), end(i)); }
26     template<class c, class b> debug&
27     ↪ operator<<(pair<b, c> d) { return *this << "("
28     ↪ << d.first << ", " << d.second << ")"; }
29     template<class a, class b, class c> debug&
30     ↪ operator<<(tuple<a, b, c> tp) { return *this <<
31     ↪ "(" << get<0>(tp) << ", " << get<1>(tp) << ", "
32     ↪ << get<2>(tp) << ")"; }
33     template<class a, class b, class c, class d>
34     ↪ debug& operator<<(tuple<a, b, c, d> tp) {
35     ↪ return *this << "(" << get<0>(tp) << ", " <<
36     ↪ get<1>(tp) << ", " << get<2>(tp) << ", " <<
37     ↪ get<3>(tp) << ")"; }
38     template<class c> debug& operator<<(rge<c> d) {
39     ↪ *this << "{";
40     ↪ for(auto it = d.b; it != d.e; ++it) {
41     ↪     *this << ", " + 2 * (it == d.b) << *it;
42     }
43     ↪ return *this << "}";
44     }
45     #else
46     template<class c> debug& operator<<(const c&) {
47     ↪ return *this; }
48     #endif
49 };
50 #define show(...) "" << LIGHT_RED << " [" << NONE
51     ↪ << __VA_ARGS__ ": " << (__VA_ARGS__) <<
52     ↪ LIGHT_RED << "]" << NONE << ""
53
54
55
56

```

## 8.5 Discrete.h

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

```
36 template<class T, class H = splitmix64_hash> using
   ↳ hash_set = hash_map<T, null_type, H>;
37
38 template<class T> inline bool chmin(T& a, const T&
   ↳ b) { if(a > b) { a = b; return true; } return
   ↳ false; }
39 template<class T> inline bool chmax(T& a, const T&
   ↳ b) { if(a < b) { a = b; return true; } return
   ↳ false; }
40
41 int main() {
42     ios::sync_with_stdio(false);
43     cin.tie(0);
44     ;
45     return 0;
46 }
47
```

## 8.6 Template.h

```
1 #include <bits/stdc++.h>
2 #include <ext/pb_ds/assoc_container.hpp>
3 using namespace std;
4 using namespace __gnu_pbds;
5
6 using uint = unsigned int;
7 using ll = long long;
8 using ull = unsigned long long;
9 using ld = long double;
10 template<class T> using pair2 = pair<T, T>;
11 using pii = pair2<int>;
12 using pll = pair2<ll>;
13 using pdd = pair2<ld>;
14 using vi = vector<int>;
15 using vl = vector<ll>;
16 template<class T> using PQ = priority_queue<T>;
17 template<class T> using PQG = priority_queue<T,
   ↳ vector<T>, greater<T>>;
18 template<class T, class Comp = less<T>> using
   ↳ ordered_set = tree<T, null_type, Comp,
   ↳ rb_tree_tag,
   ↳ tree_order_statistics_node_update>;
19 template<class T> using ordered_multiset =
   ↳ ordered_set<T, less_equal<T>>;
20
21 struct splitmix64_hash {
22     static ull splitmix64(ull x) {
23         x += 0x9e3779b97f4a7c15;
24         x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
25         x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
26         return x ^ (x >> 31);
27     }
28
29     ull operator()(ull x) const {
30         static const ull FIXED_RANDOM =
   ↳ chrono::steady_clock::now().time_since_epoch().count();
31         return splitmix64(x + FIXED_RANDOM);
32     }
33 };
34
35 template<class T, class U, class H =
   ↳ splitmix64_hash> using hash_map =
   ↳ gp_hash_table<T, U, H>;
```