Esse ai resolve com ccw
PUC-Rio

Thomaz Miranda, Miguel Batista e Joao Arthur Marques

21 de agosto de 2025

Índice

1 structures

1.1 Convex Hull Trick Dinamico

```
// para double, use INF = 1/.0, div(a, b) = a/b
// update(x) atualiza o ponto de intersecao da reta x
// overlap(x) verifica se a reta x sobrepoe a proxima
// add(a, b) adiciona reta da forma ax + b
// query(x) computa maximo de ax + b para entre as retas
//
// O(log(n)) amortizado por insercao
// O(log(n)) por query

72c struct Line {
073     mutable ll a, b, p;
8e3     bool operator<(const Line& o) const { return a < o.a; }
abf     bool operator<(ll x) const { return p < x; }
469 };

1b7 struct CHT : multiset<Line, less<>> {
33a     ll div(ll a, ll b) {
a20         return a / b - ((a ^ b) < 0 and a % b);
a8a     }

bbb     void update(iterator x) {
459         if (next(x) == end()) x->p = INF;
eec         else if (x->a == next(x)->a) x->p = x->b >= next(x)->b ?
    INF : -INF;
424         else x->p = div(next(x)->b - x->b, x->a - next(x)->a);
d37     }

71c     bool overlap(iterator x) {
f18         update(x);
cfa         if (next(x) == end()) return 0;
a4a         if (x->a == next(x)->a) return x->b >= next(x)->b;
d40         return x->p >= next(x)->p;
901     }

176     void add(ll a, ll b) {
1c7         auto x = insert({a, b, 0});
4ab         while (overlap(x)) erase(next(x)), update(x);
dbc         if (x != begin() and !overlap(prev(x))) x = prev(x),
    update(x);
0fc         while (x != begin() and overlap(prev(x)))
4d2             x = prev(x), erase(next(x)), update(x);
48f     }

4ad     ll query(ll x) {
229         assert(!empty());
7d1         auto l = *lower_bound(x);
d41 #warning cuidado com overflow!
aba         return l.a * x + l.b;
3f5     }
f1f };
```

1.2 Custom Hash for hash table

```
// Provides 64-bit hashers for integers and pairs to use with
    __gnu_pbds::gp_hash_table.
//
// complexity: O(1) average, O(n)

c4d #include <bits/extc++.h>
```

```
// for ll
75f struct chash {
5d6     const uint64_t C = ll(4e18 * acos(0)) | 71;
2cf     ll operator()(ll x) const { return __builtin_bswap64(x*C); }
cdd };

// for p64
75f struct chash {
0cf     size_t operator()(const p64& p) const {
cc9         return p.first ^ __builtin_bswap64(p.second);
1ef     }
576 };

b6a __gnu_pbds::gp_hash_table<ll, ll, chash> h({},{},{},{},{1<<16});
121 __gnu_pbds::gp_hash_table<p64, ll, chash> h({},{},{},{},{1<<16});
```

1.3 Disjoint Set Union (Union-Find)

```
// Supports find with path compression and union by size to maintain
   dynamic connectivity of disjoint sets.
//
// complexity: O(alpha(N)) amortized per op, O(N)

8d3 struct dsu {
d64     vector<ll> id, sz;

443     dsu(ll n) : id(n), sz(n, 1) { iota(id.begin(), id.end(), 0); }

f21     ll find(ll a) { return a == id[a] ? a : id[a] = find(id[a]); }

b50     void uni(ll a, ll b) {
605         a = find(a), b = find(b);
d54         if (a == b) return;
956         if (sz[a] < sz[b]) swap(a, b);
6d0         sz[a] += sz[b], id[b] = a;
761     }
7aa };
```

1.4 Fenwick Tree (Binary Indexed Tree)

```
// Supports point updates and prefix/range sum queries in logarithmic
   time using a 1-indexed BIT.
//
// complexity: O(log N) per op, O(N)

8eb struct Bit {
4de     ll n;
06c     v64 bit;
dd0     Bit(ll _n=0) : n(_n), bit(n + 1) {}
328     Bit(v64& v) : n(v.size()), bit(n + 1) {
518         for (ll i = 1; i <= n; i++) {
671             bit[i] += v[i - 1];
c8f             ll j = i + (i & -i);
b8a             if (j <= n) bit[j] += bit[i];
154         }
56d     }
e55     void update(ll i, ll x) { // soma x na posicao i
b64         for (i++; i <= n; i += i & -i) bit[i] += x;
6f4     }
2c0     ll pref(ll i) { // soma [0, i]
b73         ll ret = 0;
4d3         for (i++; i; i -= i & -i) ret += bit[i];
edf         return ret;
af2     }
235     ll query(ll l, ll r) {  // soma [l, r]
89b         return pref(r) - pref(l - 1);
aa0     }
f46     ll upper_bound(ll x) {
62d         ll p = 0;
370         for (ll i = __lg(n); i+1; i--)
6f5             if (p + (1<<i) <= n and bit[p + (1<<i)] <= x)
68e                 x -= bit[p += (1 << i)];
74e         return p;
3d3     }
f26 };
```

1.5 Fenwick Tree with Range Updates

```
// Implements a pair of BITs to support 0-based range add updates and
   range sum queries efficiently.
//
// complexity: O(log N) per op, O(N)

5aa class BIT{
3ba     ll bit[2][MAX+2];
4de     ll n;
673 public:
e33     BIT(ll n2, v64& v) {
1e3         n = n2;
```

```
914         for (ll i = 1; i <= n; i++)
edd             bit[1][min(n+1, i+(i&-i))] += bit[1][i] += v[i-1];
c9d     }
16a     ll get(ll x, ll i) {
b73         ll ret = 0;
360         for (; i; i -= i&-i) ret += bit[x][i];
edf         return ret;
346     }
23b     void add(ll x, ll i, ll val) {
503         for (; i <= n; i += i&-i) bit[x][i] += val;
669     }
f6e     ll get2(ll p) {
c7c         return get(0, p) * p + get(1, p);
006     }
235     ll query(ll l, ll r) {
ff5         return get2(r+1) - get2(l);
e1d     }
ccd     void update(ll l, ll r, ll x) {
e5f         add(0, l+1, x), add(0, r+2, -x);
f58         add(1, l+1, -x*l), add(1, r+2, x*(r+1));
4b5     }
a87 }
```

1.6 Implicit Treap (Sequence Treap)

```
// Maintains a sequence with split and merge operations using
   randomized priorities and subtree sizes.
//
// complexity: O(log N) expected per op, O(N)

125 struct Treap{
348     ll val;
0ce     ll prio, size;
330     vector<Treap*> kids;
b02     Treap(ll c): val(c), prio(rand()), size(1),
680         kids({NULL,NULL}){};
494 };

464 ll size(Treap *me){return me ? me->size : 0;}
e86 void rsz(Treap* me){me -> size =
24d     1 + size(me->kids[0]) + size(me->kids[1]);}

e8f vector<Treap*> split(Treap *me, ll idx){
878     if(!me) return {NULL,NULL};
032     vector<Treap*> out;
52a     if(size(me->kids[0]) < idx){
e1c         auto aux = split(me->kids[1],
312             idx - size(me->kids[0]) -1);
409         me->kids[1] = aux[0];
b14         rsz(me);
abb         out = {me, aux[1]};
aaa     }else{
c8a         auto aux = split(me->kids[0], idx);
c89         me->kids[0] = aux[1];
b14         rsz(me);
3cb         out = {aux[0], me};
d61     }
fe8     return out;
e7d }

b85 Treap* merge(Treap *left, Treap *right){
c10     if(left == NULL) return right;
096     if(right == NULL) return left;

671     Treap* out;

38d     if(left->prio < right->prio){
d90         left->kids[1] = merge(left->kids[1], right);
122         rsz(left);
d7a         out = left;
bbb     }else{
cea         right->kids[0] = merge(left, right->kids[0]);
e85         rsz(right);
015         out = right;
2f1     }
fe8     return out;
499 }
```

1.7 Mo's Algorithm (Offline Range Queries)

```
// Answers offline range queries by ordering them (block or Hilbert
   curve) to get small pointer movement and amortized updates.
//
// complexity: O((N + Q) sqrt N), O(N)

c41 const ll MAX = 2e5+10;
29b const ll SQ = sqrt(MAX);
1b0 ll ans;

fd9 inline void insert(ll p) {
7d3 }
```

```
155 inline void erase(ll p) {
027 }

280 inline ll hilbert(ll x, ll y) {
1ea     static ll N = 1 << (__builtin_clzll(0ll) - __builtin_clzll(MAX));
5bc     ll rx, ry, s;
b72     ll d = 0;
43b     for (s = N/2; s > 0; s /= 2) {
c95         rx = (x & s) > 0, ry = (y & s) > 0;
e3e         d += s * ll(s) * ((3 * rx) ^ ry);
d2e         if (ry == 0) {
5aa             if (rx == 1) x = N-1 - x, y = N-1 - y;
9dd             swap(x, y);
e2d         }
888     }
be2     return d;
95f }

bac #define HILBERT true
6ae vector<ll> MO(vector<pair<ll, ll>> &q) {
c3b     ans = 0;
b6a     ll m = q.size();
7d3     vector<ll> ord(m);
be8     iota(ord.begin(), ord.end(), 0);
6a6 #if HILBERT
8c4     vector<ll> h(m);
f16     for (ll i = 0; i < m; i++) h[i] = hilbert(q[i].first, q[i].second);
e60     sort(ord.begin(), ord.end(), [&](ll l, ll r) { return h[l] < h[r]; });
8c1 #else
0a3     sort(ord.begin(), ord.end(), [&](ll l, ll r) {
9c9         if (q[l].first / SQ != q[r].first / SQ) return q[l].first < q[r].first;
0db         if ((q[l].first / SQ) % 2) return q[l].second > q[r].second;
a66         return q[l].second < q[r].second;
a1d     });
f2e #endif
116     vector<ll> ret(m);
f09     ll l = 0, r = -1;

f99     for (ll i : ord) {
c60         ll ql, qr;
4f5         tie(ql, qr) = q[i];
026         while (r < qr) insert(++r);
232         while (l > ql) insert(--l);
75e         while (l < ql) erase(l++);
fe8         while (r > qr) erase(r--);
381         ret[i] = ans;
c2f     }
edf     return ret;
168 }
```

1.8 Order-Statistic Tree (PBDS)

```
// Wraps __gnu_pbds tree to support order_of_key and find_by_order operations on a sorted set.
//
// complexity: O(log N) per op, O(N)

774 #include <ext/pb_ds/assoc_container.hpp>
30f #include <ext/pb_ds/tree_policy.hpp>

0d7 using namespace __gnu_pbds;

63c #define ordered_set tree<p64, null_type,less<p64>, rb_tree_tag,tree_order_statistics_node_update>

e8d int main() {
7bf     ordered_set s;
d92     s.find_by_order(position);
d91     s.order_of_key(value);
a48 }
```

1.9 Rollback Segment Tree (Min)

```
// Segment tree supporting range min with versioned updates via a change log enabling O(1) rollback per change.
//
// complexity: O(log N) per update/query, O(N + U)

3c9 struct node {
ee4     ll lm, rm;
b7b     ll mn;
ba7     unique_ptr<node> lc, rc;

c0e     node(ll l, ll r, const vector<ll>& a) : lm(l), rm(r) {
d08         if (lm == rm) {
```

```
962             mn = a[lm];
505              return;
be3         }

0a0         ll m = (lm + rm) >> 1;
01e         lc = make_unique<node>(lm, m, a);
026         rc = make_unique<node>(m+1, rm, a);
0ca         pull();
ff1     }

89f     static ll comb(ll a, ll b) {
23a         return min(a, b);
dfe     }

48b     void pull() {
9a4         mn = comb(lc->mn, rc->mn);
3f4     }

bcf     void upd(ll lq, ll rq, ll x, vector<pair<node*,ll>>& log) {
97c         if (lq > rm || lm > rq) return;
9e3         if (lq <= lm && rm <= rq) {
031             if (mn < x) {
e06                 log.emplace_back(this, mn);
795                 mn = x;
ae2             }
505             return;
0b5         }

950         lc->upd(lq, rq, x, log);
710         rc->upd(lq, rq, x, log);

aab         ll nxt = comb(lc->mn, rc->mn);

fe3         if (mn < nxt) {
e06             log.emplace_back(this, mn);
9d8             mn = nxt;
036         }
8be     }

387     ll get(ll lq, ll rq) const {
938         if (lq > rm || lm > rq) return INF;
9af         if (lq <= lm && rm <= rq)  return mn;
002         ll res = min(lc->get(lq, rq), rc->get(lq, rq));
c31         return max(res, mn);
273     }
ed3 };
```

```
07c struct segtree {
2d0     unique_ptr<node> root;
6fa     vector<pair<node*,ll>> log;

0e0     segtree(const v64& a) {
522         root = make_unique<node>(0, (ll)a.size()-1, a);
7d4     }

7f2     void upd(ll l, ll r, ll x){
2ee         root->upd(l, r, x, log);
2d8     }

a47     ll get(ll l, ll r){
3cf         return root->get(l, r);
e85     }

6b2     ll version() const {
7a2         return (ll)log.size();
563     }

061     void rollback(ll ver){
d0f         while ((ll)log.size() > ver){
3ad             auto [p, old] = log.back();
32c             log.pop_back();
6f1             p->mn = old;
2b3         }
ba7     }
469 };
```

1.10 Segment Tree (Range Query + Point Update)

```
// Balanced binary tree for range queries with a customizable combine;
   supports point updates and range queries.
//
// complexity: O(log N) per op, O(N)

67a template<typename T>
3c9 struct node {
ee4     ll lm, rm;
ba7     unique_ptr<node> lc, rc;
f48     T val;

ff1     static constexpr T neutral = T(); // Customize this for
   min/max/gcd/etc.

181     node(ll l_, ll r_, const vector<T>& v) : lm(l_), rm(r_) {
```

```
d08         if (lm == rm) {
f6f             val = v[lm];
dea         } else {
8f6             ll m = (lm + rm) / 2;
c6d             lc = make_unique<node>(lm, m, v);
3d1             rc = make_unique<node>(m + 1, rm, v);
0ca             pull();
959         }
26c     }

592     static T comb(const T& a, const T& b) {
534         return a + b; // Change to min/max/gcd as needed
713     }

48b     void pull() {
b6d         val = comb(lc->val, rc->val);
cb1     }

e58     void point_set(ll idx, T x) {
d08         if (lm == rm) {
c43             val = x;
505             return;
81d         }
12d         if (idx <= lc->rm) lc->point_set(idx, x);
a79         else rc->point_set(idx, x);
0ca         pull();
56d     }

0b7     T query(ll lq, ll rq) {
1c5         if (rq < lm || lq > rm) return neutral;
7ea         if (lq <= lm && rm <= rq) return val;
f73         return comb(lc->query(lq, rq), rc->query(lq, rq));
9c6     }
f3e };
```

1.11 Segment Tree Over Time (Dynamic Connectivity Skeleton)

```
// Stores edge activation intervals in a segment tree over time to
   enable offline dynamic connectivity with rollback DSU.

120 struct time_query{
f88     ll l, r;
60d     time_query(ll l_, ll r_){
ae0         l = l_;
91d         r = r_;
920     }
ef5 };

d13 struct time_node {
ee4     ll lm, rm;
b4f     unique_ptr<time_node> lc, rc;

a22     vector<time_query> op;

da3     time_node(ll lm_, ll rm_){
a44         lm = lm_;
d79         rm = rm_;
be2         if (lm != rm) {
554             ll mid = (lm + rm) / 2;
d44             lc = make_unique<time_node>(lm, mid);
30e             rc = make_unique<time_node>(mid + 1, rm);
746         }
7f2     }

514     void add_query(ll lq, ll rq, time_query x) {
473         if (rq < lm || lq > rm) return;
9e3         if (lq <= lm && rm <= rq) {
488             op.push_back(x);
505             return;
335         }
455         lc->add_query(lq, rq, x);
a82         rc->add_query(lq, rq, x);
3d3     }
127 };
```

1.12 Segment Tree with Lazy Propagation (Add/Set)

```
// Supports range add and range set updates with lazy propagation and
   range queries using a composable lazy state.
//
// complexity: O(log N) per op, O(N)

67a template<typename T>
3c9 struct node {
ee4     ll lm, rm;
ba7     unique_ptr<node> lc, rc;

ff1     static constexpr T neutral = T(); // e.g., 0 for sum, INF for
   min, etc.
e2b     T val = neutral;
3c9     T lazy_add = T();
3e1     optional<T> lazy_set = nullopt;
```

```
c67     node(ll lm_, ll rm_, const vector<T>& v) : lm(lm_), rm(rm_) {
865         if (lm == rm) val = v[lm];
4e6         else {
554             ll mid = (lm + rm) / 2;
44f             lc = make_unique<node>(lm, mid, v);
4f1             rc = make_unique<node>(mid + 1, rm, v);
0ca             pull();
6a6         }
609     }

ecf     void push() {
90c         if (lazy_set.has_value()) {
ba1             val = *lazy_set * (rm - lm + 1);
be2             if (lm != rm) {
8ef                 lc->lazy_set = rc->lazy_set = lazy_set;
fe7                 lc->lazy_add = rc->lazy_add = T();
2c1             }
f46             lazy_set.reset();
0c0         }
3e3         if (lazy_add != T()) {
7aa             val += lazy_add * (rm - lm + 1);
be2             if (lm != rm) {
5ef                 if (lc->lazy_set) *lc->lazy_set += lazy_add;
57b                 else lc->lazy_add += lazy_add;

030                 if (rc->lazy_set) *rc->lazy_set += lazy_add;
5f1                 else rc->lazy_add += lazy_add;
e84             }
90d             lazy_add = T();
cf1         }
aa4     }

48b     void pull() {
b6d         val = comb(lc->val, rc->val);
cb1     }

c8e     static T comb(T a, T b) {
534         return a + b; // change for min/max/gcd/etc.
e79     }

3e2     void range_add(ll lq, ll rq, T x) {
215         push();
473         if (rq < lm || lq > rm) return;
9e3         if (lq <= lm && rm <= rq) {
4d6             lazy_add += x;
215             push();
505             return;
16c         }
5a2         lc->range_add(lq, rq, x);
903         rc->range_add(lq, rq, x);
0ca         pull();
7af     }

bac     void range_set(ll lq, ll rq, T x) {
215         push();
473         if (rq < lm || lq > rm) return;
9e3         if (lq <= lm && rm <= rq) {
111             lazy_set = x;
90d             lazy_add = T();
215             push();
505             return;
748         }
6bd         lc->range_set(lq, rq, x);
15a         rc->range_set(lq, rq, x);
0ca         pull();
b8a     }

0b7     T query(ll lq, ll rq) {
215         push();
1c5         if (rq < lm || lq > rm) return neutral;
7ea         if (lq <= lm && rm <= rq) return val;
f73         return comb(lc->query(lq, rq), rc->query(lq, rq));
065     }

e58     void point_set(ll idx, T x) {
215         push();
d08         if (lm == rm) {
c43             val = x;
505             return;
81d         }
12d         if (idx <= lc->rm) lc->point_set(idx, x);
a79         else rc->point_set(idx, x);
0ca         pull();
048     }
7d7 };
```

1.13 Sparse Table (Idempotent Range Query)

```
// Preprocesses static array to answer idempotent range queries (e.g.,
   min/max) in O(1) after O(N log N) build.
//
// complexity: O(N log N) build, O(1) query; O(N log N) space
```

```
a08 ll m[MAXN][MAXLOGN];

9ab void build(v64& v) {
90e     ll sz = v.size();

46d     forn(i, 0, sz) {
f77         m[i][0] = v[i];
313     }

27b     for (ll j = 1; (1 << j) <= sz; j++) {
edd         for (ll i = 0; i + (1 << j) <= sz; i++) {
fc8             m[i][j] = max(m[i][j-1], m[i + (1 << (j-1))][j-1]);
967         }
6f9     }
69f }

4de ll query(ll a, ll b) {
b44     ll j = __builtin_clzll(1) - __builtin_clzll(b - a + 1);
7a5     return max(m[a][j], m[b - (1 << j) + 1][j]);
168 }
```

2 math

2.1 Euler Totient Linear Sieve

```
// Computes Euler's totient for all numbers up to n using a linear
   sieve and collects primes.
//
// complexity: O(n), O(n)

558 v64 primes;
b1a vector<bool> is_comp(MAXN,false);
6d1 ll phi[MAXN];
433 ll cum_sum[MAXN];

d03 void sieve(ll n){
678   phi[1] = 1;
aff   forn(i,2,n){
850     if(!is_comp[i]){
abd       phi[i] = i-1;
e74       primes.push_back(i);
405     }
5ec     forn(j,0,primes.size()){
65d       if(i*primes[j] > n) break;
189       is_comp[i*primes[j]] = true;

01e       if(i % primes[j] == 0){
aa6         phi[i*primes[j]] = phi[i]*primes[j];
c2b         break;
522       }
10c       phi[i*primes[j]] = phi[i]*phi[primes[j]];
fef     }
295   }
829 }
```

2.2 FFT/NTT Convolution

```
// Implements iterative FFT over complex numbers and NTT over
   supported primes; provides convolution utility.
//
// complexity: O(N log N), O(N)

// Para FFT
488 void get_roots(bool f, int n, vector<complex<double>>& roots) {
f26     const static double PI = acosl(-1);
71a     for (int i = 0; i < n/2; i++) {
b1e         double alpha = i*((2*PI)/n);
1a1         if (f) alpha = -alpha;
069         roots[i] = {cos(alpha), sin(alpha)};
804     }
de5 }

// Para NTT
9f7 template<int p>
97b void get_roots(bool f, int n, vector<mod_int<p>>& roots) {
1e6     mod_int<p> r;
de9     int ord;
57a     if (p == 998244353) {
9b6         r = 102292;
81b         ord = (1 << 23);
121     } else if (p == 754974721) {
43a         r = 739831874;
f0a         ord = (1 << 24);
d48     } else if (p == 167772161) {
a2a         r = 243;
033         ord = (1 << 25);
5a4     } else assert(false);
```

```
547     if (f) r = r^(p - 1 -ord/n);
ee2     else r = r^(ord/n);
be4     roots[0] = 1;
078     for (int i = 1; i < n/2; i++) roots[i] = roots[i-1]*r;
63f }

8a2 template<typename T> void fft(vector<T>& a, bool f, int N,
    vector<int>& rev) {
bc7     for (int i = 0; i < N; i++) if (i < rev[i]) swap(a[i],
    a[rev[i]]);
12b     int l, r, m;
cb4     vector<T> roots(N);
192     for (int n = 2; n <= N; n *= 2) {
0f4         get_roots(f, n, roots);
5dc         for (int pos = 0; pos < N; pos += n) {
432             l = pos + 0, r = pos + n/2, m = 0;
a88             while (m < n/2) {
297                 auto t = roots[m] * a[r];
254                 a[r] = a[l] - t;
b8f                 a[l] = a[l] + t;
2c9                 l++, r++, m++;
d89             }
1fd         }
185     }
235     if (f) {
1c5         auto invN = T(1) / T(N);
557         for (int i = 0; i < N; i++) a[i] = a[i] * invN;
256     }
1b1 }

bf5 template<typename T> vector<T> convolution(vector<T>& a,
    vector<T>& b) {
87a     vector<T> l(a.begin(), a.end()), r(b.begin(), b.end());
e0a     int N = l.size()+r.size()-1;
f03     int n = 1, log_n = 0;
0a4     while (n <= N) n *= 2, log_n++;
808     vector<int> rev(n);
603     for (int i = 0; i < n; i++) {
434         rev[i] = 0;
f44         for (int j = 0; j < log_n; j++) if (i>>j&1)
4ff             rev[i] |= 1 << (log_n-1-j);
256     }

143     assert(N <= n);
fa4     l.resize(n);
7e4     r.resize(n);
56e     fft(l, false, n, rev);
fcf     fft(r, false, n, rev);
917     for (int i = 0; i < n; i++) l[i] *= r[i];
88b     fft(l, true, n, rev);
5e1     l.resize(N);
792     return l;
bd6 }

// NTT
74c template<int p, typename T>
b74 vector<mod_int<p>> ntt(vector<T>& a, vector<T>& b) {
d52     vector<mod_int<p>> A(a.begin(), a.end()), B(b.begin(),
    b.end());
d29     return convolution(A, B);
543 }
```

2.3 Modular Arithmetic Helpers

```
// Provides modular add/sub/mul, fast exponentiation, and modular
    inverse under fixed MOD.
//
// complexity: O(log E) for power/inverse, O(1)

0f6 const ll MOD = 1_000_000_007;

d7e inline ll sum(ll a, ll b) { a += b; if (a >= MOD) a -= MOD; return
    a; }
e0b inline ll sub(ll a, ll b) { a -= b; if (a < 0)  a += MOD; return
    a; }
d06 inline ll mult(ll a, ll b) { return (a * b) % MOD; }

f15 inline ll pot(ll base, ll exp) {
ce0     ll res = 1;
fb9     while (exp) {
3c3         if (exp & 1) res = mult(res, base);
ee9         base = mult(base, base);
ef0         exp >>= 1;
dcf     }
b50     return res;
24d }

840 inline ll inv_mod(ll a) {return pot(a, MOD-2);}
```

3 geometry

3.1 Convex Hull (Monotone Chain)

```
// Computes the convex hull of a set of points using Andrew's monotone chain; handles collinear points based on ccw condition.
//
// complexity: O(N log N), O(N)

// se contar pontos colineares, faz o ccw com >=
0cd bool ccw(pt p, pt q, pt r) { // se p, q, r sao ccw
276     return sarea2(p, q, r) > 0;
42b }

eb2 vector<pt> convex_hull(vector<pt>& v) { // convex hull - O(n log(n))
fca     sort(v.begin(), v.end());
d76     v.erase(unique(v.begin(), v.end()), v.end());
52d     if (v.size() <= 1) return v;
526     vector<pt> l, u;
f14     for (int i = 0; i < v.size(); i++) {
fb2         while (l.size() > 1 and !ccw(l.end()[-2], l.end()[-1], v[i]))
364             l.pop_back();
c35         l.push_back(v[i]);
58e     }
3e9     for (int i = v.size() - 1; i >= 0; i--) {
f19         while (u.size() > 1 and !ccw(u.end()[-2], u.end()[-1], v[i]))
7a8             u.pop_back();
a95         u.push_back(v[i]);
0b8     }
cfc     l.pop_back(); u.pop_back();
82b     for (pt i : u) l.push_back(i);
792     return l;
548 }
```

3.2 Integer Geometry Primitives

```
// Defines 2D point and line structures with orientation, area, and angle comparisons plus a sweep-line comparator.

b2a struct pt { // ponto
0be     ll x, y;
f6f     pt(ll x_ = 0, ll y_ = 0) : x(x_), y(y_) {}
5bc     bool operator < (const pt p) const {
95a         if (x != p.x) return x < p.x;
89c         return y < p.y;
dcd     }
a83     bool operator == (const pt p) const {
d74         return x == p.x and y == p.y;
7b4     }
cb9     pt operator + (const pt p) const { return pt(x+p.x, y+p.y); }
a24     pt operator - (const pt p) const { return pt(x-p.x, y-p.y); }
8f0     pt operator * (const ll c) const { return pt(x*c, y*c); }
60d     ll operator * (const pt p) const { return x*(ll)p.x + y*(ll)p.y; }
d86     ll operator ^ (const pt p) const { return x*(ll)p.y - y*(ll)p.x; }
5ed     friend istream& operator >> (istream& in, pt& p) {
e37         return in >> p.x >> p.y;
e45     }
f3f };

b3a struct line { // reta
730     pt p, q;

0d6     line() {}
4b8     line(pt p_, pt q_) : p(p_), q(q_) {}
7f9     bool operator < (const line l) const {
d1d         if (!(p == l.p)) return p < l.p;
d4a         return q < l.q;
2ca     }
e1c     bool operator == (const line l) const {
689         return p == l.p and q == l.q;
030     }
8d7     friend istream& operator >> (istream& in, line& r) {
4cb         return in >> r.p >> r.q;
858     }
c29 };

5a2 ll sarea2(pt p, pt q, pt r) { // 2 * area com sinal
586     return (q-p)^(r-q);
bf4 }

0cd bool ccw(pt p, pt q, pt r) { // se p, q, r sao ccw
276     return sarea2(p, q, r) > 0;
42b }

c31 int quad(pt p) { // quadrante de um ponto
dbb     return (p.x<0)^3*(p.y<0);
fcf }
```

```
2df bool compare_angle(pt p, pt q) { // retorna se ang(p) < ang(q)
9fc     if (quad(p) != quad(q)) return quad(p) < quad(q);
ea1     return ccw(q, pt(0, 0), p);
771 }

// comparador pro set pra fazer sweep line com segmentos
2c4 struct cmp_sweepline {
d80     bool operator () (const line& a, const line& b) const {
            // assume que os segmentos tem p < q
191         if (a.p == b.p) return ccw(a.p, a.q, b.q);
614         if (a.p.x != a.q.x and (b.p.x == b.q.x or a.p.x < b.p.x))
780             return ccw(a.p, a.q, b.p);
dc0         return ccw(a.p, b.q, b.p);
baf     }
677 };
```

3.3 Segment Sweep Line Skeleton

```
// Maintains an active set of segments ordered for sweep-line processing over x; insertion and removal are typically logarithmic.

// observacoes sobre sweepline em segmentos:
// tomar cuidado com segmentos verticais se a sweepline e em x, nesse caso devemos ignorar esses casos sera que podemos fazer isso em outros problemas
// tomar cuidado para nao usar funcoes da biblioteca em lugares errados...
// a partir de agora, usar a funcao de comparacao de linhas como nesse arquivo
// colocar informacoes na struc de linha para retirar mapas

719 map<ll, set<line, cmp_sweepline>> sweepline_begin; // dado um x, diz quais linhas comecam naquele x
5a6 map<ll, set<line, cmp_sweepline>> sweepline_end;   // dado um x, diz quais linhas terminam naquele x

972 void process_beg(set<line, cmp_sweepline>& v, set<line, cmp_sweepline>& active_line, vector<ll>& parent){
47d     for(auto x : v){
380         active_line.insert(x);
            // processar uma linha que esta sendo adicionada
9c5     }
6b3 }
923 void process_end(set<line, cmp_sweepline>& v, set<line, cmp_sweepline>& active_line){
47d     for(auto x : v){
d76         active_line.erase(x);
a1d     }
68a }

ec5 void sweepline(ll n){
23e     set<line, cmp_sweepline> active_line;

967     while(!sweepline_begin.empty() or !sweepline_end.empty()){
c58         auto it_beg = sweepline_begin.begin();
aa0         auto it_end = sweepline_end.begin();

385         if(sweepline_end.empty()){
570             process_beg(it_beg->second, active_line, parent);
7ae             sweepline_begin.erase(it_beg);
5e2             continue;
a8b         }

32a         if(sweepline_begin.empty() or it_end->first <= it_beg->first){
2a4             process_end(it_end->second, active_line);
61a             sweepline_end.erase(it_end);
5e2             continue;
ddb         }

570         process_beg(it_beg->second, active_line, parent);
7ae         sweepline_begin.erase(it_beg);
c12     }
9c2 }
```

4 graphs

4.1 Bridge Detection (Tarjan)

```
// Finds all bridges in an undirected graph via DFS timestamps and low-link values.
//
// complexity: O(N + M), O(N + M)

a64 vector<v64> g;
591 vector<bool> visited;
023 vector<ll> tin, low;
```

```
ddf ll timer = 0;

081 void dfs(ll u, ll p = -1) {
2a9     visited[u] = true;
ae3     tin[u] = low[u] = timer++;
cd0     for (ll v : g[u]) {
730         if (v == p) continue;

d53         if (visited[v]) {
34f             low[u] = min(low[u], tin[v]);
caf         } else {
95e             dfs(v, u);
ab6             low[u] = min(low[u], low[v]);
975             if (low[v] > tin[u]) {
                    // THIS IS A BRIDGE
4b8             }
450         }
e83     }
7a4 }

822 void find_bridges() {
451     timer = 0;
411     visited.assign(n, false);
cfd     tin.assign(n, -1);
dc4     low.assign(n, -1);
522     forn(i, 0, n) {
b1c         if (!visited[i])
1e5             dfs(i);
bf3     }
bf3 }
```

4.2 Centroid Decomposition

```
// Decompose Centroid

84c vector<ll> g[MAX];
d7c ll sz[MAX], rem[MAX];

b87 void dfs(v64& path, ll i, ll l=-1, ll d=0) {
547     path.push_back(d);
3d0     for (ll j : g[i]) if (j != l and !rem[j]) dfs(path, j, i, d+1);
3e1 }

499 ll dfs_sz(ll i, ll l=-1) {
02c     sz[i] = 1;
05b     for (ll j : g[i]) if (j != l and !rem[j]) sz[i] += dfs_sz(j,
    i);
191     return sz[i];
329 }

c46 ll centroid(ll i, ll l, ll size) {
51f     for (ll j : g[i]) if (j != l and !rem[j] and sz[j] > size / 2)
735         return centroid(j, i, size);
d9a     return i;
c6f }

27a ll decomp(ll i, ll k) {
79c     ll c = centroid(i, i, dfs_sz(i));
a67     rem[c] = 1;

        // gasta O(n) aqui - dfs sem ir pros caras removidos
04b     ll ans = 0;
4eb     vector<ll> cnt(sz[i]);
878     cnt[0] = 1;
e65     for (ll j : g[c]) if (!rem[j]) {
04c         vector<ll> path;
baf         dfs(path, j);
392         for (ll d : path) if (0 <= k-d-1 and k-d-1 < sz[i])
285             ans += cnt[k-d-1];
477         for (ll d : path) cnt[d+1]++;
4d9     }

ffb     for (ll j : g[c]) if (!rem[j]) ans += decomp(j, k);
3f1     rem[c] = 0;
ba7     return ans;
595 }
```

4.3 Centroid Tree

```
// Constroi a centroid tree
// p[i] eh o pai de i na centroid-tree
// dist[i][k] = distancia na arvore original entre i
// e o k-esimo ancestral na arvore da centroid
//
// O(n log(n)) de tempo e memoria

7d6 vector<v64> g(MAX), dist(MAX);
20d vector<ll> sz(MAX), rem(MAX), p(MAX);

499 ll dfs_sz(ll i, ll l=-1) {
02c     sz[i] = 1;
```

```
05b     for (ll j : g[i]) if (j != l and !rem[j]) sz[i] += dfs_sz(j, i);
191     return sz[i];
329 }

c46 ll centroid(ll i, ll l, ll size) {
51f     for (ll j : g[i]) if (j != l and !rem[j] and sz[j] > size / 2)
735         return centroid(j, i, size);
d9a     return i;
c6f }

3de void dfs_dist(ll i, ll l, ll d=0) {
541     dist[i].push_back(d);
a75     for (ll j : g[i]) if (j != l and !rem[j])
82a         dfs_dist(j, i, d+1);
fea }

457 void decomp(ll i, ll l = -1) {
79c     ll c = centroid(i, i, dfs_sz(i));
1b9     rem[c] = 1, p[c] = l;
534     dfs_dist(c, c);
1ef     for (ll j : g[c]) if (!rem[j]) decomp(j, c);
f75 }

145 void build(ll n) {
b26     forn(i,0,n) rem[i] = 0, dist[i].clear();
867     decomp(0);
40c     forn(i,0,n) reverse(dist[i].begin(), dist[i].end());
9d9 }
```

4.4 Dijkstra's Shortest Paths

```
// Computes single-source shortest paths on non-negative weighted
   graphs using a priority queue.
//
// complexity: O((N + M) log N), O(N + M)

c6d vector<vector<p64>> g;

// d = distance | p = from/path
ff3 void dijkstra(ll s, v64 &d, v64& p) {
846     ll n = g.size();
355     d.assign(n, INF);
d8d     p.assign(n, -1);

d66     d[s] = 0;
930     priority_queue<p64> pq;
7ba     pq.push({0, s});
502     while (!pq.empty()) {
5cd         ll u = pq.top().second;
6fd         ll d_u = -pq.top().first;
716         pq.pop();

211         if (d_u != d[u]) continue;

bf7         for (auto edge : g[u]) {
615             ll v = edge.first;
61d             ll w_v = edge.second;

f35             if (d[u] + w_v < d[v]) {
7ca                 d[v] = d[u] + w_v;
e42                 p[v] = u;
2a6                 pq.push({-d[v], v});
e72             }
138         }
461     }
a63 }
```

4.5 Dinic's Maximum Flow (with Scaling)

```
// Computes max flow using Dinic's algorithm with optional capacity
   scaling to speed up BFS levels.
//
// complexity: O(E V^2) worst-case, O(E)

472 struct dinitz {
d76     const bool scaling = true;
d74     ll lim;
670     struct edge {
283         ll to, cap, rev, flow;
7f9         bool res;
764         edge(ll to_, ll cap_, ll rev_, bool res_)
a94             : to(to_), cap(cap_), rev(rev_), flow(0), res(res_) {}
eb4     };

002     vector<vector<edge>> g;
d6c     vector<ll> lev, beg;
a71     ll F;
17f     dinitz(ll n) : g(n), F(0) {}

f3e     void add(ll a, ll b, ll c) {
bae         g[a].emplace_back(b, c, g[b].size(), false);
```

```
4c6         g[b].emplace_back(a, 0, g[a].size()-1, true);
abb     }

6d8     bool bfs(ll s, ll t) {
c8a         lev = vector<ll>(g.size(), -1); lev[s] = 0;
0a3         beg = vector<ll>(g.size(), 0);
7a6         queue<ll> q; q.push(s);
402         while (q.size()) {
c79             ll u = q.front(); q.pop();
bd9             for (auto& i : g[u]) {
dbc                 if (lev[i.to] != -1 or (i.flow == i.cap)) continue;
b4f                 if (scaling and i.cap - i.flow < lim) continue;
185                 lev[i.to] = lev[u] + 1;
8ca                 q.push(i.to);
f97             }
cab         }
0de         return lev[t] != -1;
0db     }

bae     ll dfs(ll v, ll s, ll f = INF) {
50b         if (!f or v == s) return f;
678         for (ll& i = beg[v]; i < g[v].size(); i++) {
027             auto& e = g[v][i];
206             if (lev[e.to] != lev[v] + 1) continue;
a30             ll foi = dfs(e.to, s, min(f, e.cap - e.flow));
749             if (!foi) continue;
3c5             e.flow += foi, g[e.to][e.rev].flow -= foi;
45c             return foi;
7bf         }
bb3         return 0;
d2a     }

074     ll max_flow(ll s, ll t) {
a86         for (lim = scaling ? (1<<30) : 1; lim; lim /= 2)
69c             while (bfs(s, t)) while (ll ff = dfs(s, t)) F += ff;
4ff         return F;
370     }

e30     void reset() {
59f         F = 0;
843         for (auto& edges : g) for (auto& e : edges) e.flow = 0;
5d0     }
575 };
```

4.6 Floyd-Warshall Algorithm

```
// Computes all-pairs shortest paths and detects negative cycles using
    dynamic programming over path lengths.
//
// complexity: O(N^3), O(N^2)

4de ll n;
1a5 ll d[MAX][MAX];

73c bool floyd_warshall() {
e22     for (int k = 0; k < n; k++)
830     for (int i = 0; i < n; i++)
f90     for (int j = 0; j < n; j++)
0ab         d[i][j] = min(d[i][j], d[i][k] + d[k][j]);

830     for (int i = 0; i < n; i++)
753         if (d[i][i] < 0) return 1;

bb3     return 0;
192 }
```

4.7 Strongly Connected Components (Kosaraju)

```
// Computes SCCs using two DFS passes and builds the condensation
    graph.
//
// complexity: O(N + M), O(N + M)

591 vector<bool> visited;

297 void dfs(ll v, vector<v64>& g, vector<ll> &out) {
e75     visited[v] = true;
819     for(auto u : g[v]) if(!visited[u]) dfs(u, g, out);
3ad     out.push_back(v);
b7f }

64d vector<v64> scc(vector<v64>& g) {
af1     int n = g.size();
cb9     v64 order, roots(n, 0);

c44     vector<v64> adj_rev(n);
0c2     forn(u, 0, n) for (ll v : g[u]) adj_rev[v].push_back(u);

411     visited.assign(n, false);
2b4     forn(i, 0, n) if (!visited[i]) dfs(i, g, order);
b3a     reverse(order.begin(), order.end());
```

```
411     visited.assign(n, false);
de0     ll curr_comp = 0;
0ee     for (auto v : order) {
451         if (!visited[v]) {
a76             v64 component; dfs(v, adj_rev, component);
fe3             for (auto u : component) roots[u] = curr_comp;
5f2             curr_comp++;
19c         }
74d     }

e7f     set<p64> edges;
556     vector<v64> cond_g(curr_comp);
c6b     forn(u, 0, n) {
7b9         for (auto v : g[u]) {
2dd             if (roots[u] != roots[v] && !edges.count({roots[u], roots[v]})) {
2b9                 cond_g[roots[u]].push_back(roots[v]);
893                 edges.emplace(roots[u], roots[v]);
fbe             }
f76         }
3b9     }
594     return cond_g;
afd }
```

4.8 Topological Sort (Kahn's Algorithm)

```
// Produces a topological ordering of a DAG using indegree counting
   and a queue-like frontier.
//
// complexity: O(N + M), O(N)

1f7 v64 topo_sort(const vector<v64>& g) {
94c     v64 indeg(g.size()), q;
edb     for (auto& li : g) for (int x : li) indeg[x]++;
6bc     forn(i, 0, g.size()) if (indeg[i] == 0) q.push_back(i);
ff1     forn(j, 0, q.size()) for(int x : g[q[j]]) if(--indeg[x] == 0) q.push_back(x);
bef     return q;
ebe }
```

5 strings

5.1 Aho-Corasick Automaton

```
// Builds a trie with failure links for multi-pattern matching; insert
   is O(|s|), build is linear in total length, and queries run in
   linear time in the text.
//
// complexity: varies, O(total patterns length)

ea1 namespace aho {
05b     map<char, ll> to[MAX];
b0a     ll link[MAX], idx, term[MAX], exit[MAX], sobe[MAX];
5e1     vector<ll> max_match(MAX, 0);

bfc     void insert(string& s) {
4eb         ll at = 0;
b4f         for (char c : s) {
b68             auto it = to[at].find(c);
1c9             if (it == to[at].end()) at = to[at][c] = ++idx;
361             else at = it->second;
ff4         }
142         term[at]++, sobe[at]++;
8f6         max_match[at] = s.size();
d0b     }

0a8     void build() {
848         queue<ll> q;
537         q.push(0);
dff         link[0] = exit[0] = -1;
402         while (q.size()) {
aa7             ll i = q.front(); q.pop();
3c4             for (auto [c, j] : to[i]) {
7d8                 ll l = link[i];
102                 while (l != -1 and !to[l].count(c)) l = link[l];
7a5                 link[j] = l == -1 ? 0 : to[l][c];
3ab                 exit[j] = term[link[j]] ? link[j] : exit[link[j]];

058                 max_match[j] = max(max_match[link[j]], max_match[j]);
6f2                 if (exit[j]+1) sobe[j] += sobe[exit[j]];
113                 q.push(j);
ed4             }
c9f         }
138     }
5e1     ll query(string& s) {
```

```
0db         ll at = 0, ans = 0;
b4f         for (char c : s){
1ca             while (at != -1 and !to[at].count(c)) at = link[at];
5b9             at = at == -1 ? 0 : to[at][c];
2b1             ans += sobe[at];
b85         }
ba7         return ans;
0bf     }
028     vector<ll> match_vec(string& s) {
5bf         ll at = 0, n = s.size();
d93         vector<ll> v(n, 0);
522         forn(i, 0, n){
827             char c = s[i];
1ca             while (at != -1 and !to[at].count(c)) at = link[at];
5b9             at = at == -1 ? 0 : to[at][c];

c84             v[i] = max_match[at]; // quero isso
5eb         }
6dc         return v;
9db     }
16d }
```

5.2 Knuth-Morris-Pratt (KMP)

```
// Computes prefix function and performs linear-time substring search with optional automaton construction.
//
// complexity: O(n + m), O(n)

a15 v64 pi(string& s) {
125     v64 p(s.size());
030     for (ll i = 1, j = 0; i < (ll) s.size(); i++) {
a51         while (j and s[j] != s[i]) j = p[j-1];
973         if (s[j] == s[i]) j++;
f8c         p[i] = j;
e98     }
74e     return p;
c1a }

e89 v64 match(string& pat, string& s) {
cce     v64 p = pi(pat), match;
dde     for (ll i = 0, j = 0; i < (ll) s.size(); i++) {
a5a         while (j and pat[j] != s[i]) j = p[j-1];
3e3         if (pat[j] == s[i]) j++;
64d         if (j == pat.size()) match.push_back(i-j+1), j = p[j-1];
d07     }
ed8     return match;
3c7 }

4a5 struct KMPaut : vector<v64> {
47c     KMPaut(){}
501     KMPaut (string& s) : vector<v64>(26, v64(s.size()+1)) {
bb1         v64 p = pi(s);
04b         auto& aut = *this;
4fa         aut[s[0]-'a'][0] = 1;
19a         for (char c = 0; c < 26; c++)
5d3             for (int i = 1; i <= s.size(); i++)
42b                 aut[c][i] = s[i]-'a' == c ? i+1 : aut[c][p[i-1]];
86c     }
af1 };
```

5.3 Suffix Array - O(n log n)

```
// kasai recebe o suffix array e calcula lcp[i],
// o lcp entre s[sa[i],...,n-1] e s[sa[i+1],..,n-1]
//
// Complexidades:
// suffix_array - O(n log(n))
// kasai - O(n)

ad7 v64 suffix_array(string s) {
59b     s.push_back('$'); // 0 caso v64 (CHECAR SE PODE)
e1f     ll n = s.size(), N = max(n, 260ll);
b3e     v64 sa(n), ra(n);
828     forn(i, 0, n) sa[i] = i, ra[i] = s[i];

2e6     for(ll k = 0; k < n; k ? k *= 2 : k++) {
1db         v64 nsa(sa), nra(n), cnt(N);

fae         for(int i = 0; i < n; i++) nsa[i] = (nsa[i]-k+n)%n, cnt[ra[i]]++;

e18         forn(i, 1, N) cnt[i] += cnt[i-1];
f62         for(ll i = n-1; i+1; i--) sa[--cnt[ra[nsa[i]]]] = nsa[i];

aec         for(ll i = 1, r = 0; i < n; i++) nra[sa[i]] = r += ra[sa[i]] !=
f86             ra[sa[i-1]] or ra[(sa[i]+k)%n] != ra[(sa[i-1]+k)%n];
26b         ra = nra;
d5e         if (ra[sa[n-1]] == n-1) break;
02e     }
9f9     return v64(sa.begin()+1, sa.end());
```

```
2ea }

c46 v64 kasai(string s, v64 sa) {
381     ll n = s.size(), k = 0;
f7c     v64 ra(n), lcp(n);
540     forn(i, 0, n) ra[sa[i]] = i;

514     for (ll i = 0; i < n; i++, k -= !!k) {
199         if (ra[i] == n-1) { k = 0; continue; }
674         ll j = sa[ra[i]+1];
891         while (i+k < n and j+k < n and s[i+k] == s[j+k]) k++;
d98         lcp[ra[i]] = k;
b37     }
5ed     return lcp;
8b5 }
```

5.4 Trie (Prefix Tree)

```
// Stores strings over a fixed alphabet to support insert, erase, and prefix counting in linear time.
//
// complexity: O(|s|) per op, O(total keys)

ab5 struct trie {
99e     vector<v64> to;
82f     v64 end, pref;
1c5     ll sigma; char norm;

a5e     trie(ll sigma_=26, char norm_='a') : sigma(sigma_), norm(norm_) {
108         to = {v64(sigma)};
86e         end = {0}, pref = {0};
d3f     }

64e     void insert(string s) {
00d         ll x = 0;
7e7         for (auto c : s) {
00c             ll &nxt = to[x][c-norm];
dd7             if (!nxt) {
0aa                 nxt = to.size();
821                 to.push_back(v64(sigma));
770                 end.push_back(0), pref.push_back(0);
0bd             }
827             x = nxt, pref[x]++;
b7d         }
421         end[x]++, pref[0]++;
dcd     }

6b2     void erase(string s) {
00d         ll x = 0;
b4f         for (char c : s) {
00c             ll &nxt = to[x][c-norm];
10c             x = nxt, pref[x]--;
d8e             if (!pref[x]) nxt = 0;
e3d         }
104         end[x]--, pref[0]--;
b69     }

680     ll find(string s) {
00d         ll x = 0;
7e7         for (auto c : s) {
2ec             x = to[x][c-norm];
59b             if (!x) return -1;
42d         }
ea5         return x;
63a     }

fde     ll count_pref(string s) {
b09         ll id = find(s);
fc1         return id >= 0 ? pref[id] : 0;
d11     }
17b };
```

6 misc

6.1 Binary Search Helpers

```
// Template functions to find first or last index satisfying a monotonic predicate over a sorted search space.
//
// complexity: O(log N), O(1)

8e2 ll find_last_valid(ll val) {
70d     ll left = 0;
cac     ll right = n - 1;
263     ll result = -1;

d08     while (left <= right) {
184         ll mid = left + (right - left) / 2;
de0         if (condition) {
```

```
294             result = mid;
3f4             left = mid + 1;
120         } else {
75e             right = mid - 1;
9f9         }
c4a     }
dc8     return result;
33d }

77b ll find_first_valid(ll val) {
70d     ll left = 0;
cac     ll right = n - 1;
c66     ll result = n;

d08     while (left <= right) {
184         ll mid = left + (right - left) / 2;
de0         if (condition) {
294             result = mid;
75e             right = mid - 1;
a0d         } else {
3f4             left = mid + 1;
113         }
2fc     }
dc8     return result;
d96 }
```

6.2 Divide and Conquer DP Optimization

```
// Optimizes DP transitions with quadrangle inequality/monge-like
   structure using divide-and-conquer over optimal decision points.
//
// complexity: O(K N log N) with O(1) cost, O(N)

f6c vector<v64> dp; // dp[n+1][2]

b8e void solve(ll k, ll l, ll r, ll lk, ll rk) {
de6     if (l > r) return;
f20     ll m = (l+r)/2, p = -1;
bff     auto& ans = dp[m][k&1] = INF;
f08     for (ll i = max(m, lk); i <= rk; i++) {
d73         ll at = dp[i+1][~k&1] + cost(m, i);
57d         if (at < ans) ans = at, p = i;
d63     }
1ee     solve(k, l, m-1, lk, p), solve(k, m+1, r, p, rk);
35b }

c5c ll dnc(ll n, ll k) {
321     dp[n][0] = dp[n][1] = 0;
390     forn(i,0,n) dp[i][0] = INF;
050     forn(i,1,k+1) solve(i, 0, n-i, 0, n-i);
8e7     return dp[0][k&1];
40f }
```

6.3 Modular Integer

```
// Fixed-modulus integer type with +, -, *, /, and exponentiation;
   modulo should be prime for division via Fermat.
//
// complexity: O(1) per arithmetic op (O(log E) for exponentiation),
   O(1)

a2f const ll MOD = 998244353;

429 template<int p> struct mod_int {
c68     ll expo(ll b, ll e) {
c85         ll ret = 1;
c87         while (e) {
cad             if (e % 2) ret = ret * b % p;
9d2             e /= 2, b = b * b % p;
c42         }
edf         return ret;
734     }
1f6     ll inv(ll b) { return expo(b, p-2); }

4d7     using m = mod_int;
aa3     ll v;
fe0     mod_int() : v(0) {}
e12     mod_int(ll v_) {
019         if (v_ >= p or v_ <= -p) v_ %= p;
bc6         if (v_ < 0) v_ += p;
2e7         v = v_;
7f3     }
74d     m& operator +=(const m& a) {
2fd         v += a.v;
ba5         if (v >= p) v -= p;
357         return *this;
c8b     }
eff     m& operator -=(const m& a) {
8b4         v -= a.v;
cc8         if (v < 0) v += p;
357         return *this;
f8d     }
```

```
4c4     m& operator *=(const m& a) {
8a5         v = v * ll(a.v) % p;
357         return *this;
d4c     }
3f9     m& operator /=(const m& a) {
5d6         v = v * inv(a.v) % p;
357         return *this;
62d     }
d65     m operator -(){ return m(-v); }
b3e     m& operator ^=(ll e) {
06d         if (e < 0) {
6e2             v = inv(v);
00c             e = -e;
275         }
284         v = expo(v, e);
            // possivel otimizacao:
            // cuidado com 0^0
            // v = expo(v, e%(p-1));
357         return *this;
6ed     }
423     bool operator ==(const m& a) { return v == a.v; }
69f     bool operator !=(const m& a) { return v != a.v; }

1c6     friend istream& operator >>(istream& in, m& a) {
d1c         ll val; in >> val;
d48         a = m(val);
091         return in;
870     }
44f     friend ostream& operator <<(ostream& out, m a) {
5a0         return out << a.v;
214     }
399     friend m operator +(m a, m b) { return a += b; }
f9e     friend m operator -(m a, m b) { return a -= b; }
9c1     friend m operator *(m a, m b) { return a *= b; }
51b     friend m operator /(m a, m b) { return a /= b; }
08f     friend m operator ^(m a, ll e) { return a ^= e; }
424 };
0f4 typedef mod_int<MOD> mint;
```

7 extra

7.1 stress.sh

```
P=a
make ${P} ${P}2 gen || exit 1
for ((i = 1; ; i++)) do
    ./gen $i > in
    ./${P} < in > out
    ./${P}2 < in > out2
    if (! cmp -s out out2) then
        echo "--> entrada:"
        cat in
        echo "--> saida1:"
        cat out
        echo "--> saida2:"
        cat out2
        break;
    fi
    echo $i
done
```

7.2 pragmas.cpp

```
// Perfomance geral (seguro p/ CP)
 #pragma GCC optimize("O3,unroll-loops,fast-math")

// Maximo vetor + FP agressivo (pode quebrar precisao)
 #pragma GCC optimize("Ofast,fast-math,unroll-loops,inline")

// Foco em binario pequeno
 #pragma GCC optimize("Os")
```

7.3 hash.sh

```
sed -n $2','$3' p' $1 | sed '/^#w/d' | cpp -dD -P -fpreprocessed | tr
    -d '[:space:]' | md5sum | cut -c-6
```

7.4 makefile

```
CXX = g++
CXXFLAGS = -fsanitize=address,undefined -fno-omit-frame-pointer -g
    -Wall -Wshadow -std=c++17 -Wno-unused-result -Wno-sign-compare
    -Wno-char-subscripts
```

7.5 template.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef pair<ll, ll> p64;
typedef vector<ll> v64;

#define forn(i, s, e) for(ll i = (s); i < (e); i++)
#define ln "\n"

#if defined(DEBUG)
    #define _ (void)0
    #define debug(x) cout << __LINE__ << ": " << #x << " = " << x << ln
#else
    #define _ ios_base::sync_with_stdio(false), cin.tie(NULL)
    #define debug(x) (void)0
#endif

const ll INF = 0x3f3f3f3f3f3f3f3fll;

int main(){
    _;
    return 0;
}
```

7.6 random.cpp

```cpp
mt19937_64 rng((ll)
    chrono::steady_clock::now().time_since_epoch().count());

ll uniform(ll l, ll r){
    uniform_int_distribution<ll> uid(l, r);
    return uid(rng);
}
```