template

```cpp
#pragma GCC optimize("Ofast")
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

#define repx(i, a, b) for (int i = a; i < b; i++)
#define rep(i, n) repx(i, 0, n)
#define invrepx(i, a, b) for (int i = b - 1; i >= a; i--)
#define invrep(i, n) invrepx(i, 0, n)

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
}
```

ds

dsu

```cpp
struct Dsu {
  vector<int> p, r;

  // initialize the disjoint-set-union to all unitary sets
  void reset(int N) {
    p.resize(N), r.assign(N, 0);
    rep(i, N) p[i] = i;
  }

  // find the leader node corresponding to node `i`
  int find(int i) {
    if (p[i] != i) p[i] = find(p[i]);
    return p[i];
  }

  // perform union on the two sets with leaders `i` and `j`
  // note: `i` and `j` must be GROUP LEADERS
  void unite(int i, int j) {
    if (i == j) return;
    if (r[i] > r[j]) swap(i, j);
    if (r[i] == r[j]) r[j] += 1;
    p[i] = j;
  }
};
```

segtree

```cpp
template <class T>
struct StSum {
  vector<T> node;

  void reset(int N) { node.assign(4 * N, 0); }

  void build(const vector<T>& a, int v = 1, int vl = 0, int vr = -1) {
    node.resize(4 * a.size());
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v] = a[vl]; // construction
      return;
    }
    int vm = (vl + vr) / 2;
    build(a, 2 * v, vl, vm);
    build(a, 2 * v + 1, vm, vr);
    node[v] = node[2 * v] + node[2 * v + 1]; // query op
  }

  // query for range [l, r)
  T query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l == vl && r == vr) return node[v];
    int vm = (vl + vr) / 2;
    T val = 0; // neutral element
    if (l >= vr || r <= vl) return val;
    val += query(l, min(r, vm), 2 * v, vl, vm);   // query op
    val += query(max(l, vm), r, 2 * v + 1, vm, vr); // query op
    return val;
  }

  // set element i to val
  void update(int i, T val, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v] = val;
      return;
    }
    int vm = (vl + vr) / 2;
    if (i < vm) {
      update(i, val, 2 * v, vl, vm);
    } else {
      update(i, val, 2 * v + 1, vm, vr);
    }
    node[v] = node[2 * v] + node[2 * v + 1]; // query op
  }
};

const ll INF = 1e18;

template <class T>
struct StMax {
  vector<T> node;

  void reset(int N) { node.assign(4 * N, -INF); }

  void build(const vector<T>& a, int v = 1, int vl = 0, int vr = -1) {
    node.resize(4 * a.size());
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v] = a[vl]; // construction
      return;
    }
    int vm = (vl + vr) / 2;
    build(a, 2 * v, vl, vm);
    build(a, 2 * v + 1, vm, vr);
    node[v] = max(node[2 * v], node[2 * v + 1]); // query op
  }

  // query for range [l, r)
  T query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l == vl && r == vr) return node[v];
    int vm = (vl + vr) / 2;
    T val = -INF; // neutral element
    if (l >= vr || r <= vl) return val;
    val = max(val, query(l, min(r, vm), 2 * v, vl, vm));   // query op
    val = max(val, query(max(l, vm), r, 2 * v + 1, vm, vr)); // query op
    return val;
  }

  // set element i to val
  void update(int i, T val, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v] = val;
      return;
    }
    int vm = (vl + vr) / 2;
    if (i < vm) {
      update(i, val, 2 * v, vl, vm);
    } else {
      update(i, val, 2 * v + 1, vm, vr);
    }
    node[v] = max(node[2 * v], node[2 * v + 1]); // query op
  }
};

template <class T>
struct StMin {
  vector<T> node;

  void reset(int N) { node.assign(4 * N, INF); }

  void build(const vector<T>& a, int v = 1, int vl = 0, int vr = -1) {
    node.resize(4 * a.size());
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v] = a[vl]; // construction
      return;
    }
    int vm = (vl + vr) / 2;
    build(a, 2 * v, vl, vm);
```

```
    build(a, 2 * v + 1, vm, vr);
    node[v] = min(node[2 * v], node[2 * v + 1]); // query op
  }

  // query for range [l, r)
  T query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l == vl && r == vr) return node[v];
    int vm = (vl + vr) / 2;
    T val = INF; // neutral element
    if (l >= vr || r <= vl) return val;
    val = min(val, query(l, min(r, vm), 2 * v, vl, vm));    // query op
    val = min(val, query(max(l, vm), r, 2 * v + 1, vm, vr)); // query op
    return val;
  }

  // set element i to val
  void update(int i, T val, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v] = val;
      return;
    }
    int vm = (vl + vr) / 2;
    if (i < vm) {
      update(i, val, 2 * v, vl, vm);
    } else {
      update(i, val, 2 * v + 1, vm, vr);
    }
    node[v] = min(node[2 * v], node[2 * v + 1]); // query op
  }
};
```

segtreelazy

```
template <class T>
struct StlSumSum {
  // immediate (result of querying in the segment)
  // lazy (value that has not been pushed to the children)
  vector<pair<T, T>> node;

  void reset(int N) { node.assign(4 * N, {0, 0}); }

  void build(const vector<T>& a, int v = 1, int vl = 0, int vr = -1) {
    node.resize(4 * a.size());
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v].first = a[vl]; // construction
      return;
    }
    int vm = (vl + vr) / 2;
    build(a, 2 * v, vl, vm);
    build(a, 2 * v + 1, vm, vr);
    node[v].first = node[2 * v].first + node[2 * v + 1].first; // query op
    node[v].second = 0; // update-zero
  }

  // helper: propagate lazy values in vertex `v` to both of its children
  void push(int v, int vl, int vr) {
    int vm = (vl + vr) / 2;
    T& lazy = node[v].second;
    node[2 * v].first += lazy * (vm - vl);    // update-op & query-op mix
    node[2 * v].second += lazy;             // update-op
    node[2 * v + 1].first += lazy * (vr - vm); // update-op & query-op mix
    node[2 * v + 1].second += lazy;          // update-op
    lazy = 0;                            // update-zero
  }

  // update range [l, r) using val
  void update(int l, int r, T val, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l >= vr || r <= vl || r <= l) return;
    if (l == vl && r == vr) {
      node[v].first += val * (vr - vl); // update-op & query-op mix
      node[v].second += val;         // update-op
      return;
    }
    push(v, vl, vr);
    int vm = (vl + vr) / 2;
    update(l, min(r, vm), val, 2 * v, vl, vm);
    update(max(l, vm), r, val, 2 * v + 1, vm, vr);
    node[v].first = node[2 * v].first + node[2 * v + 1].first; // query-op
  }

  // query range [l, r)
  T query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l <= vl && r >= vr) return node[v].first;
    int vm = (vl + vr) / 2;
    T val = 0; // query-zero
    if (l >= vr || r <= vl || r <= l) return val;
    push(v, vl, vr);
    val += query(l, min(r, vm), 2 * v, vl, vm);    // query-op
    val += query(max(l, vm), r, 2 * v + 1, vm, vr); // query-op
    return val;
  }
};

const ll INF = 1e18;

template <class T>
struct StlSetSum {
  // immediate (result of querying in the segment)
  // lazy (value that has not been pushed to the children)
  vector<pair<T, T>> node;

  void reset(int N) { node.assign(4 * N, {0, INF}); }

  void build(const vector<T>& a, int v = 1, int vl = 0, int vr = -1) {
    node.resize(4 * a.size());
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v] = {a[vl], INF}; // construction
      return;
    }
    int vm = (vl + vr) / 2;
    build(a, 2 * v, vl, vm);
    build(a, 2 * v + 1, vm, vr);
    node[v].first = node[2 * v].first + node[2 * v + 1].first; // query op
    node[v].second = INF; // update-zero
  }

  // helper: propagate lazy values in vertex `v` to both of its children
  void push(int v, int vl, int vr) {
    int vm = (vl + vr) / 2;
    T& lazy = node[v].second;
    if (lazy != INF) {
      node[2 * v].first = lazy * (vm - vl); // update-op & query-op mix
      node[2 * v].second = lazy;         // update-op
      node[2 * v + 1].first =
        lazy * (vr - vm);       // update-op & query-op mix
      node[2 * v + 1].second = lazy; // update-op
    }
    lazy = INF; // update-zero
  }

  // update range [l, r) using val
  void update(int l, int r, T val, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l >= vr || r <= vl || r <= l) return;
    if (l == vl && r == vr) {
      node[v].first = val * (vr - vl); // update-op & query-op mix
      node[v].second = val;         // update-op
      return;
    }
    push(v, vl, vr);
    int vm = (vl + vr) / 2;
    update(l, min(r, vm), val, 2 * v, vl, vm);
    update(max(l, vm), r, val, 2 * v + 1, vm, vr);
    node[v].first = node[2 * v].first + node[2 * v + 1].first; // query-op
  }

  // query range [l, r)
  T query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l <= vl && r >= vr) return node[v].first;
    int vm = (vl + vr) / 2;
    T val = 0; // query-zero
    if (l >= vr || r <= vl || r <= l) return val;
    push(v, vl, vr);
    val += query(l, min(r, vm), 2 * v, vl, vm);    // query-op
    val += query(max(l, vm), r, 2 * v + 1, vm, vr); // query-op
    return val;
  }
};

template <class T>
struct StlSumMin {
  // immediate (result of querying in the segment)
  // lazy (value that has not been pushed to the children)
  vector<pair<T, T>> node;

  void reset(int N) { node.assign(4 * N, {0, 0}); }

  void build(const vector<T>& a, int v = 1, int vl = 0, int vr = -1) {
```

```
    node.resize(4 * a.size());
    if (vr == -1) vr = node.size() / 4;
    if (vr - vl == 1) {
      node[v] = {a[vl], 0}; // construction
      return;
    }
    int vm = (vl + vr) / 2;
    build(a, 2 * v, vl, vm);
    build(a, 2 * v + 1, vm, vr);
    node[v].first =
      min(node[2 * v].first, node[2 * v + 1].first); // query op
    node[v].second = 0;                     // update-zero
  }

  // helper: propagate lazy values in vertex `v` to both of its children
  void push(int v, int vl, int vr) {
    int vm = (vl + vr) / 2;
    T& lazy = node[v].second;
    node[2 * v].first += lazy;     // update-op & query-op mix
    node[2 * v].second += lazy;    // update-op
    node[2 * v + 1].first += lazy;  // update-op & query-op mix
    node[2 * v + 1].second += lazy; // update-op
    lazy = 0;               // update-zero
  }

  // update range [l, r) using val
  void update(int l, int r, T val, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l >= vr || r <= vl || r <= l) return;
    if (l == vl && r == vr) {
      node[v].first += val;  // update-op & query-op mix
      node[v].second += val; // update-op
      return;
    }
    push(v, vl, vr);
    int vm = (vl + vr) / 2;
    update(l, min(r, vm), val, 2 * v, vl, vm);
    update(max(l, vm), r, val, 2 * v + 1, vm, vr);
    node[v].first =
      min(node[2 * v].first, node[2 * v + 1].first); // query-op
  }

  // query range [l, r)
  T query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l <= vl && r >= vr) return node[v].first;
    int vm = (vl + vr) / 2;
    T val = INF; // query-zero
    if (l >= vr || r <= vl || r <= l) return val;
    push(v, vl, vr);
    val = min(val, query(l, min(r, vm), 2 * v, vl, vm));    // query-op
    val = min(val, query(max(l, vm), r, 2 * v + 1, vm, vr)); // query-op
    return val;
  }
};
```

sparse

```
// handle immutable range maximum queries (or any idempotent query) in O(1)
template <class T>
struct Sparse {
  vector<vector<T>> st;

  Sparse() {}

  void reset(int N) { st = {vector<T>(N)}; }
  void set(int i, T val) { st[0][i] = val; }

  // O(N log N) time
  // O(N log N) memory
  void init() {
    int N = st[0].size();
    int npot = N <= 1 ? 1 : 32 - __builtin_clz(N);
    st.resize(npot);
    repx(i, 1, npot) rep(j, N + 1 - (1 << i)) st[i].push_back(
      max(st[i - 1][j], st[i - 1][j + (1 << (i - 1))])); // query op
  }

  // query maximum in the range [l, r) in O(1) time
  T query(int l, int r) {
    int i = 31 - __builtin_clz(r - l);
    return max(st[i][l], st[i][r - (1 << i)]); // query op
  }
};
```

umap

```
// hackproof rng
static mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

// deterministic rng
uint64_t splitmix64(uint64_t* x) {
  uint64_t z = (*x += 0x9e3779b97f4a7c15);
  z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
  z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
  return z ^ (z >> 31);
}

// hackproof unordered map hash
struct Hash {
  size_t operator()(const ll& x) const {
    static const uint64_t RAND =
      chrono::steady_clock::now().time_since_epoch().count();
    uint64_t z = x + RAND + 0x9e3779b97f4a7c15;
    z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
    z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
    return z ^ (z >> 31);
  }
};

// hackproof unordered_map
template <class T, class U>
using umap = unordered_map<T, U, Hash>;

// hackproof unordered_set
template <class T>
using uset = unordered_set<T, Hash>;

// an unordered map with small integer keys that avoids hashing, but allows
↪  O(N)
// iteration and clearing, with N being the amount of items (not the maximum
// key).
template <class T>
struct Map {
  int N;
  vector<bool> used;
  vector<int> keys;
  vector<T> vals;

  Map() {}
  // O(C)
  void resize(int C) {
    C += 1, used.resize(C), keys.resize(C), vals.resize(C);
  }

  // O(1)
  T& operator[](int k) {
    if (!used[k]) used[k] = true, keys[N++] = k, vals[k] = T();
    return vals[k];
  }

  // O(N)
  void clear() {
    while (N) used[keys[--N]] = false;
  }

  // O(N)
  template <class OP>
  void iterate(OP op) {
    rep(i, N) op(keys[i], vals[keys[i]]);
  }
};
```

mo

```
struct Query {
  int l, r, idx;
};

// answer segment queries using only `add(i)`, `remove(i)` and `get()`
// functions.
//
// complexity: O((N + Q) * sqrt(N) * F)
//   N = length of the full segment
//   Q = amount of queries
//   F = complexity of the `add`, `remove` functions
template <class A, class R, class G, class T>
void mo(vector<Query>& queries, vector<T>& ans, A add, R remove, G get) {
  int Q = queries.size(), B = (int)sqrt(Q);
```

```
  sort(queries.begin(), queries.end(), [&](Query& a, Query& b) {
    return make_pair(a.l / B, a.r) < make_pair(b.l / B, b.r);
  });
  ans.resize(Q);

  int l = 0, r = 0;
  for (auto& q : queries) {
    while (r < q.r) add(r), r++;
    while (l > q.l) l--, add(l);
    while (r > q.r) r--, remove(r);
    while (l < q.l) remove(l), l++;
    ans[q.idx] = get();
  }
}
```

math

arith

```
// floor(log2(n)) without precision loss
inline int floor_log2(int n) { return n <= 1 ? 0 : 31 - __builtin_clz(n); }
// ceil(log2(n)) without precision loss
inline int ceil_log2(int n) { return n <= 1 ? 0 : 32 - __builtin_clz(n - 1); }

inline ll floordiv(ll a, ll b) {
  ll d = a / b;
  return d * b == a ? d : d - ((a < 0) ^ (b < 0));
}

inline ll ceildiv(ll a, ll b) {
  ll d = a / b;
  return d * b == a ? d : d - ((a < 0) ^ (b < 0)) + 1;
}

// binary exponentiation
ll binexp(ll a, ll m) {
  ll res = 1; // neutral element
  while (m) {
    if (m & 1) res = res * a; // multiplication
    a = a * a;          // multiplication
    m >>= 1;
  }
  return res;
}

// counts the divisors of a positive integer in O(sqrt(n))
ll count_divisors(ll x) {
  ll divs = 1;
  ll i = 2;
  while (x > 1) {
    if (i * i > x) {
      divs *= 2;
      break;
    }
    int n = 1;
    while (x % i == 0) {
      x /= i;
      n += 1;
    }
    divs *= n;
    i += 1;
  }
  return divs;
}

// gets the prime factorization of a number in O(sqrt(n))
void factorize(vector<pair<ll, int>>& facts, ll x) {
  ll k = 2;
  while (x > 1) {
    if (k * k > x) {
      facts.push_back({x, 1});
      break;
    }
    int n = 0;
    while (x % k == 0) x /= k, n++;
    if (n > 0) facts.push_back({k, n});
    k += 1;
  }
}

// iterate over all divisors of a number.
//
// divisor count upper bound: n^(1.07 / ln ln n)
template <class OP>
void divisors(ll x, OP op) {
  vector<pair<ll, int>> facts;
  factorize(facts, x);
  vector<int> f(facts.size());
  while (true) {
    ll y = 1;
    rep(i, f.size()) rep(j, f[i]) y *= facts[i].first;
    op(y);

    int i = 0;
    while (i < f.size()) {
      f[i] += 1;
      if (f[i] > facts[i].second)
        f[i++] = 0;
      else
        break;
    }
    if (i == f.size()) break;
  }
}
```

mod

```
// take the modulo for possibly negative numbers.
ll mod(ll a, ll M) { return (a % M + M) % M; }

// binary exponentiation modulo M.
ll binexp(ll a, ll m, ll M) {
  assert(m >= 0);
  ll res = 1 % M;
  while (m) {
    if (m & 1) res = (res * a) % M;
    a = (a * a) % M;
    m >>= 1;
  }
  return res;
}

// compute the modular multiplicative inverse, assuming M is prime.
ll multinv(ll a, ll M) { return binexp(a, M - 2, M); }

// calculate gcd(a, b).
// also, calculate x and y such that:
// a * x + b * y == gcd(a, b)
//
// time: O(log min(a, b))
// (ignoring complexity of arithmetic)
ll ext_gcd(ll a, ll b, ll& x, ll& y) {
  if (b == 0) {
    x = 1, y = 0;
    return a;
  }
  ll d = ext_gcd(b, a % b, y, x);
  y -= a / b * x;
  return d;
}

// compute the modular multiplicative inverse, assuming a and MOD are
↪  coprime.
// MOD may not be prime.
ll multinv_euc(ll a, ll M) {
  ll x, y;
  ll g = ext_gcd(a, M, x, y);
  assert(g == 1);
  return (x % M + M) % M;
}

// computes euler totative function phi(x), counting the amount of integers
↪  in
// [1, x] that are relatively prime to x.
//
// time: O(sqrt(x))
ll eulerphi(ll x) {
  ll phi = 1;
  ll k = 2;
  while (x > 1) {
    if (k * k > x) {
      phi *= x - 1;
```

```
      break;
    }
    ll k1 = 1, k0 = 0;
    while (x % k == 0) x /= k, k0 = k1, k1 *= k;
    phi *= k1 - k0;
    k += 1;
  }
  return phi;
}

// N choose K but modular, using a precomputed factorial table.
ll choose(ll* fact, ll n, ll k, ll M) {
  return fact[n] * multinv(fact[k] * fact[n - k] % M, M) % M;
}

// multiply two big numbers (~10^18) under a large modulo, without resorting
↪  to
// bigints.
ll bigmul(ll x, ll y, ll M) {
  ll z = 0;
  while (y) {
    if (y & 1) z = (z + x) % M;
    x = (x << 1) % M, y >>= 1;
  }
  return z;
}

// discrete logarithm.
// solve a ^ x = b (mod M) for the smallest x.
// returns -1 if no solution is found.
//
// time: O(sqrt(M))
ll dlog(ll a, ll b, ll M) {
  ll k = 1 % M, s = 0;
  while (true) {
    ll g = __gcd(a, M);
    if (g <= 1) break;
    if (b == k) return s;
    if (b % g != 0) return -1;
    b /= g, M /= g, s += 1, k = a / g * k % M;
  }
  ll N = sqrt(M) + 1;

  static umap<ll, ll> r;
  r.clear();
  ll baq = b;
  rep(q, N + 1) {
    r[baq] = q;
    baq = baq * a % M;
  }

  ll aN = binexp(a, N, M), aNp = k;
  repx(p, 1, N + 1) {
    aNp = aNp * aN % M;
    if (r.count(aNp)) return N * p - r[aNp] + s;
  }
  return -1;
}
```

crt

```
// given a set of modular equations, each of the form `x = Ai (mod Pi)`,
↪  compute
// `x mod LCM(P0, P1, ..., Pn)`.
// returns a pair of `x` and `LCM(P0, P1, ..., Pn)`.
//
// if the equations cannot be satisfied, {-1, -1} is returned.
pair<bigint, bigint> solve_crt_big(const vector<pair<ll, ll>>& eqs) {
  bigint a0 = eqs[0].first, p0 = eqs[0].second;
  repx(i, 1, eqs.size()) {
    bigint a1 = eqs[i].first, p1 = eqs[i].second;
    bigint k1, k0;
    bigint d = ext_gcd(p1, p0, k1, k0);
    a0 -= a1;
    if (a0.divmod(d) != 0) return {-1, -1};
    p0 /= d, p0 *= p1;
    a0 *= k1, a0 *= p1, a0 += a1;
    a0 %= p0, a0 += p0, a0 %= p0;
  }
  return {a0, p0};
}

ll ext_gcd_small(ll a, ll b, ll& x, ll& y) {
  if (b == 0) {
    x = 1, y = 0;
    return a;
  }
  ll d = ext_gcd_small(b, a % b, y, x);
  y -= a / b * x;
  return d;
}

pair<ll, ll> solve_crt(const vector<pair<ll, ll>>& eqs) {
  ll a0 = eqs[0].first, p0 = eqs[0].second;
  repx(i, 1, eqs.size()) {
    ll a1 = eqs[i].first, p1 = eqs[i].second;
    ll k1, k0;
    ll d = ext_gcd(p1, p0, k1, k0);
    a0 -= a1;
    if (a0 % d != 0) return {-1, -1};
    p0 = p0 / d * p1;
    a0 = a0 / d * k1 % p0 * p1 % p0 + a1;
    a0 = (a0 % p0 + p0) % p0;
  }
  return {a0, p0};
}
```

gauss

```
const double EPS = 1e-9;

// complexity: O(min(N, M) * N * M)
// `a` is a list of rows
// the last value in each row is the result of the equation
// return values:
//  0 -> no solutions
//  1 -> unique solution, stored in `ans`
// -1 -> infinitely many solutions, one of which is stored in `ans`
int gauss(vector<vector<double>> a, vector<double>& ans) {
  int N = a.size(), M = a[0].size() - 1;

  vector<int> where(M, -1);
  for (int col = 0, row = 0; col < M && row < N; col++, row++) {
    int sel = row;
    repx(i, row, N) if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
    if (abs(a[sel][col]) < EPS) continue;
    repx(i, col, M + 1) swap(a[sel][i], a[row][i]);
    where[col] = row;

    rep(i, N) if (i != row) {
      double c = a[i][col] / a[row][col];
      repx(j, col, M + 1) a[i][j] -= a[row][j] * c;
    }
  }

  ans.assign(M, 0);
  rep(i, M) if (where[i] != -1) ans[i] = a[where[i]][M] / a[where[i]][i];
  rep(i, N) {
    double sum = 0;
    rep(j, M) sum += ans[j] * a[i][j];
    if (abs(sum - a[i][M]) > EPS) return 0;
  }

  rep(i, M) if (where[i] == -1) return -1;
  return 1;
}
```

poly

```
using cd = complex<double>;
const double PI = acos(-1);

// compute the DFT of a power-of-two-length sequence.
// if `inv` is true, computes the inverse DFT.
//
// the DFT of a polynomial A(x) = A0 + A1*x + A2*x^2 + ... + An*x^n is the
↪  array
// of the polynomial A evaluated in all nths roots of unity: [A(w0), A(w1),
// A(w2), ..., A(wn-1)], where w0 = 1 and w1 is the nth principal root of
↪  unity.
void fft(vector<cd>& a, bool inv) {
  int N = a.size();
  assert(N == 1 << __builtin_ctz(N));

  int k = 0;
  rep(i, N) {
    int bit = N >> 1;
    while (k & bit) k ^= bit, bit >>= 1;
    k ^= bit;
```

```cpp
    if (i < k) swap(a[i], a[k]);
  }

  for (int len = 2; len <= N; len <<= 1) {
    double ang = 2 * PI / len * (inv ? -1 : 1);
    cd wlen(cos(ang), sin(ang));
    for (int i = 0; i < N; i += len) {
      cd w(1);
      repx(j, 0, len / 2) {
        cd u = a[i + j], v = a[i + j + len / 2] * w;
        a[i + j] = u + v;
        a[i + j + len / 2] = u - v;
        w *= wlen;
      }
      i += len;
    }
    len <<= 1;
  }

  if (inv)
    for (cd& x : a) x /= N;
}

const ll MOD = 7340033, ROOT = 5, ROOTPOW = 1 << 20;

// compute the DFT of a power-of-two-length sequence, modulo a special prime
// number with principal root.
//
// the modulus _must_ be a prime number with an Nth root of unity, where N
↪  is a
// power of two. the FFT can only be performed on arrays of size <= N.
void modfft(vector<ll>& a, bool inv) {
  int N = a.size();
  assert(N == 1 << __builtin_ctz(N) && N <= ROOTPOW);
  rep(i, N) a[i] = (a[i] % MOD + MOD) % MOD;

  int k = 0;
  repx(i, 1, N) {
    int bit = N >> 1;
    while (k & bit) k ^= bit, bit >>= 1;
    k ^= bit;
    if (i < k) swap(a[i], a[k]);
  }

  for (int len = 2; len <= N; len <<= 1) {
    ll wlen = inv ? multinv(ROOT, MOD) : ROOT;
    for (ll i = ROOTPOW; i > len; i >>= 1) wlen = wlen * wlen % MOD;
    for (int i = 0; i < N; i += len) {
      ll w = 1;
      repx(j, 0, len / 2) {
        ll u = a[i + j], v = a[i + j + len / 2] * w % MOD;
        a[i + j] = (u + v) % MOD;
        a[i + j + len / 2] = (u - v + MOD) % MOD;
        w = w * wlen % MOD;
      }
    }
  }

  if (inv) {
    ll ninv = multinv(N, MOD);
    for (ll& x : a) x = x * ninv % MOD;
  }
}

using T = ll;
T pmul(T a, T b) { return a * b % MOD; }
T padd(T a, T b) { return (a + b) % MOD; }
T psub(T a, T b) { return (a - b + MOD) % MOD; }
T pinv(T a) { return multinv(a, MOD); }

struct Poly {
  vector<T> a;

  Poly() {}
  Poly(T c) : a(c) { trim(); }
  Poly(vector<T> c) : a(c) { trim(); }

  void trim() {
    while (!a.empty() && a.back() == 0) a.pop_back();
  }
  int deg() const { return a.empty() ? -1000000 : a.size() - 1; }
  Poly sub(int l, int r) const {
    r = min(r, (int)a.size()), l = min(l, r);
    return vector<T>(a.begin() + l, a.begin() + r);
  }
  Poly trunc(int n) const { return sub(0, n); }
  Poly shl(int n) const {
    Poly out = *this;
    out.a.insert(out.a.begin(), n, 0);
    return out;
  }
  Poly rev(int n, bool r = false) const {
    Poly out(*this);
    if (r) out.a.resize(max(n, (int)a.size()));
    reverse(out.a.begin(), out.a.end());
    return out.trunc(n);
  }

  Poly& operator+=(const Poly& rhs) {
    auto& b = rhs.a;
    a.resize(max(a.size(), b.size()));
    rep(i, b.size()) a[i] = padd(a[i], b[i]); // add
    trim();
    return *this;
  }
  Poly& operator-=(const Poly& rhs) {
    auto& b = rhs.a;
    a.resize(max(a.size(), b.size()));
    rep(i, b.size()) a[i] = psub(a[i], b[i]); // sub
    trim();
    return *this;
  }
  Poly& operator*=(const Poly& rhs) {
    int n = deg() + rhs.deg() + 1;
    if (n <= 0) return *this = Poly();
    n = 1 << (n <= 1 ? 0 : 32 - __builtin_clz(n - 1));
    vector<T> b = rhs.a;
    a.resize(n), b.resize(n);
    modfft(a, false), modfft(b, false);     // fft
    rep(i, a.size()) a[i] = pmul(a[i], b[i]); // mul
    modfft(a, true), trim();              // invfft
    return *this;
  }
  Poly inv(int n) const {
    assert(deg() >= 0);
    Poly ans = pinv(a[0]); // inverse
    int b = 1;
    while (b < n) {
      Poly C = (ans * trunc(2 * b)).sub(b, 2 * b);
      ans -= (ans * C).trunc(b).shl(b);
      b *= 2;
    }
    return ans.trunc(n);
  }

  Poly operator+(const Poly& rhs) const { return Poly(*this) += rhs; }
  Poly operator-(const Poly& rhs) const { return Poly(*this) -= rhs; }
  Poly operator*(const Poly& rhs) const { return Poly(*this) *= rhs; }

  pair<Poly, Poly> divmod(const Poly& b) const {
    if (deg() < b.deg()) return {Poly(), *this};
    int d = deg() - b.deg() + 1;
    Poly D = (rev(d) * b.rev(d).inv(d)).trunc(d).rev(d, true);
    return {D, *this - D * b};
  }
  Poly operator/(const Poly& b) const { return divmod(b).first; }
  Poly operator%(const Poly& b) const { return divmod(b).second; }
  Poly& operator/=(const Poly& b) { return *this = divmod(b).first; }
  Poly& operator%=(const Poly& b) { return *this = divmod(b).second; }

  T eval(T x) {
    T y = 0;
    invrep(i, a.size()) y = padd(pmul(y, x), a[i]); // add, mul
    return y;
  }
  Poly& build(vector<Poly>& tree, vector<T>& x, int v, int l, int r) {
    if (l == r) return tree[v] = vector<T>{-x[l], 1};
    int m = (l + r) / 2;
    return tree[v] = build(tree, x, 2 * v, l, m) *
             build(tree, x, 2 * v + 1, m + 1, r);
  }
  void subeval(vector<Poly>& tree, vector<T>& x, vector<T>& y, int v, int l,
        int r) {
    if (l == r) {
      y[l] = eval(x[l]);
      return;
    }
    int m = (l + r) / 2;
    (*this % tree[2 * v]).subeval(tree, x, y, 2 * v, l, m);
    (*this % tree[2 * v + 1]).subeval(tree, x, y, 2 * v + 1, m + 1, r);
  }
  // evaluate m points in O(k (log k)^2) with k = max(n, m).
  vector<T> multieval(vector<T>& x) {
    int N = x.size();
    if (deg() < 0) return vector<T>(N, 0);
    vector<Poly> tree(4 * N);
    build(tree, x, 1, 0, N - 1);
```

```
    vector<T> y(N);
    subeval(tree, x, y, 1, 0, N - 1);
    return y;
  }

  friend ostream& operator<<(ostream& s, const Poly& p) {
    s << "(";
    bool first = true;
    rep(i, p.a.size()) {
      if (p.a[i] == 0) continue;
      if (!first) s << " + ";
      s << p.a[i];
      if (i > 0) s << " x";
      if (i > 1) s << "^" << i;
      first = false;
    }
    s << ")";
    return s;
  }
};
```

bigint

```
using u32 = uint32_t;
using u64 = uint64_t;

// signed bigint
struct bigint {
  vector<u32> digits;
  u32 neg;

  bigint() : neg(0) {}
  bigint(ll x) : digits{lo(x), hi(x)}, neg(x < 0 ? ~0 : 0) { this->trim(); }
  bigint(vector<u32> d) : digits(d), neg(0) {}

  static u32 lo(u64 dw) { return (u32)dw; }
  static u32 hi(u64 dw) { return (u32)(dw >> 32); }

  // remove leading zeros from representation
  void trim() {
    while (digits.size() && digits.back() == neg) digits.pop_back();
  }

  void add(const bigint& rhs, u32 c = 0) {
    int ls = digits.size();
    int rs = rhs.digits.size();
    rep(i, max(ls, rs)) {
      if (i >= ls) digits.push_back(neg);
      u64 r = (u64)digits[i] + (i < rs ? rhs.digits[i] : rhs.neg) + c;
      digits[i] = lo(r), c = hi(r);
    }
    u64 ec = (u64)c + neg + rhs.neg;
    neg = ((hi(ec) ^ neg ^ rhs.neg) & 1 ? ~0 : 0);
    if (lo(ec) != neg) digits.push_back(lo(ec));
  }
  bigint& operator+=(const bigint& rhs) {
    this->add(rhs);
    return *this;
  }
  bigint& operator+=(u32 rhs) {
    this->add({}, rhs);
    return *this;
  }

  void negate() {
    rep(i, digits.size()) digits[i] = ~digits[i];
    neg = ~neg;
    this->add({}, 1);
  }

  bigint negated() const {
    bigint out = *this;
    out.negate();
    return out;
  }

  bigint& operator-=(const bigint& rhs) {
    this->negate();
    *this += rhs;
    this->negate();
    return *this;
  }

  bigint& operator*=(bigint& rhs) {
    static bigint lhs;
    swap(*this, lhs), digits.clear(), neg = 0;
    u32 r = rhs.neg, s = 0;
    if (lhs.neg) s ^= lhs.neg, lhs.negate();
    if (rhs.neg) s ^= rhs.neg, rhs.negate();
    rep(j, rhs.digits.size()) {
      u64 c = 0;
      int ls = digits.size();
      int rs = lhs.digits.size();
      repx(i, j, max(ls, rs + j)) {
        if (i >= ls) digits.push_back(0);
        u64 r =
          (u64)digits[i] +
          (u64)(i - j < rs ? lhs.digits[i - j] : 0) * rhs.digits[j] +
          c;
        digits[i] = lo(r), c = hi(r);
      }
      if (c != 0) digits.push_back(c);
    }
    if (r) rhs.negate();
    if (s) negate();
    return *this;
  }

  bigint& operator/=(bigint& rhs) {
    divmod(rhs);
    return *this;
  }
  bigint& operator%=(bigint& rhs) {
    *this = divmod(rhs);
    return *this;
  }

  int divmod_trunc(int rhs) {
    u32 s = (rhs < 0 ? ~0 : 0) ^ this->neg, q = abs(rhs);
    u64 r = 0;
    if (this->neg) this->negate();
    invrep(i, digits.size()) {
      r = (r << 32) | digits[i];
      digits[i] = r / q, r %= q;
    }
    if (s) {
      this->negate();
      return -(int)r;
    }
    return (int)r;
  }

  // compares `this` with `rhs`
  //  `this < rhs`: -1
  //  `this == rhs`: 0
  //  `this > rhs`: 1
  int cmp(const bigint& rhs) const {
    if (neg && !rhs.neg) return -1;
    if (!neg && rhs.neg) return 1;
    int ls = digits.size(), rs = rhs.digits.size();
    invrep(i, max(ls, rs)) {
      u32 l = i < ls ? digits[i] : neg;
      u32 r = i < rs ? rhs.digits[i] : rhs.neg;
      if (l < r) return -1;
      if (l > r) return 1;
    }
    return 0;
  }

  bool operator==(const bigint& rhs) const { return cmp(rhs) == 0; }
  bool operator!=(const bigint& rhs) const { return cmp(rhs) != 0; }
  bool operator<(const bigint& rhs) const { return cmp(rhs) == -1; }
  bool operator>=(const bigint& rhs) const { return cmp(rhs) != -1; }
  bool operator>(const bigint& rhs) const { return cmp(rhs) == 1; }
  bool operator<=(const bigint& rhs) const { return cmp(rhs) != 1; }

  friend ostream& operator<<(ostream& s, const bigint& self) {
    if (self == bigint()) return s << "0";
    bigint x = self;
    if (x.neg) {
      x.negate();
      s << "-";
    }
    vector<int> digs;
    while (x != bigint()) digs.push_back(x.divmod_trunc(10));
    invrep(i, digs.size()) s << digs[i];
    return s;
  }

  // truncating division and modulo
  bigint divmod(bigint& rhs) {
    assert(rhs != bigint());
    u32 sr = rhs.neg, s = neg ^ rhs.neg;
    if (neg) negate();
    if (sr) rhs.negate();
```

```
    bigint l = 0, r = *this, x;
    r += 1u;
    while (l != r) {
      bigint m = l;
      m += r;
      rep(i, m.digits.size()) m.digits[i] =
        (m.digits[i] >> 1) |
        (i + 1 < m.digits.size() ? m.digits[i + 1] << 31 : 0);
      x = m, x *= rhs;
      if (x <= *this) {
        l = (m += 1);
      } else {
        r = m;
      }
    }
    l -= 1, swap(l, *this);
    r = *this, r *= rhs, l -= r;
    trim(), l.trim();
    if (sr) rhs.negate();
    if (s) negate(), l.negate();
    return l;
  }
};

// calculate gcd(a, b).
// also, calculate x and y such that:
// a * x + b * y == gcd(a, b)
bigint ext_gcd(bigint a, bigint b, bigint& x, bigint& y) {
  if (b == bigint()) {
    x = 1, y = 0;
    return a;
  }
  bigint c = a.divmod(b), d = ext_gcd(b, c, y, x);
  a *= x, y -= a;
  return d;
}
```

segment

```
// in-place segment intersection.
void intersect(pair<int, int>& a, pair<int, int> b) {
  a = {max(a.first, b.first), min(a.second, b.second)};
}

// in-place segment "union".
// finds the shortest segment that contains both `a` and `b`.
//
// for [a, b) segments: change > to >= and <= to <
void unite(pair<int, int>& a, pair<int, int> b) {
  if (a.first > a.second)
    a = b;
  else if (b.first <= b.second)
    a = {min(a.first, b.first), max(a.second, b.second)};
}

// segment containment.
//
// [a, b] in [c, d]
// subset or equal: a >= c && b <= d || a >  b
// proper subset:   a >  c && b <  d || a >  b && c <= d
//
// [a, b) in [c, d)
// subset or equal: a >= c && b <= d || a >= b
// proper subset:   a >  c && b <  d || a >= b && c <  d
bool is_subset(pair<int, int> sub, pair<int, int> sup) {
  return sub.first >= sup.first && sub.second <= sup.second ||
     sub.second < sub.first;
}
bool is_subset_proper(pair<int, int> sub, pair<int, int> sup) {
  return sub.first > sup.first && sub.second < sup.second ||
     sub.second < sub.first && sup.first <= sup.second;
}
```

theorems

```
Burnside lemma

  For a set X, with members x in X, and a group G, with operations g in G, where g(x) : X -> X.
  F_g is the set of x which are fixed points of g (ie. { x in X / g(x) = x }).
  The number of orbits (connected components in the graph formed by assigning each x a node and
  a directed edge between x and g(x) for every g) is called M.
  M = the average of the fixed points of all g = (|F_g1| + |F_g2| + ... + |F_gn|) / |G|

  If x are images and g are simmetries, then M corresponds to the amount of objects , |G|
  corresponds to the amount of simmetries, and F_g corresponds to the amount of simmetrical
  images under the simmetry g.

Rational root theorem

  All rational roots of the polynomials with integer coefficients:

  a0 * x^0 + a1 * x^1 + a2 * x^2 + ... + an * x^n = 0

  If these roots are represented as p / q, with p and q coprime,
  - p is an integer factor of a0
  - q is an integer factor of an

  Note that if a0 = 0, then x = 0 is a root, the polynomial can be divided by x and the theorem
  applies once again.

Legendres formula

  Considering a prime p, the largest power p^k that divides n! is given by:

  k = floor(n/p) + floor(n/p^2) + floor(n/p^3) + ...

  O(log n / log p)
```

graph

dist

```
const ll INF = 1e18;

// calculate distances between every pair of nodes in O(n^3) time and O(n^2)
// memory.
// requires an NxN array to store results.
void floyd(const vector<vector<pair<int, ll>>>& G, vector<vector<ll>>& dists) {
  int N = G.size();
  rep(i, N) rep(j, N) dists[i][j] = i == j ? 0 : INF;
  rep(i, N) for (auto edge : G[i]) dists[i][edge.first] = edge.second;
  rep(k, N) rep(i, N) rep(j, N) {
    dists[i][j] = min(dists[i][j], dists[i][k] + dists[k][j]);
  }
}

// calculate shortest distances from a source node to every other node in
// O(m log n). requires an array of size N to store results.
void dijkstra(const vector<vector<pair<int, ll>>>& G, vector<ll>& dists,
       int src) {
  dists.assign(G.size(), INF);
  dists[src] = 0;
  priority_queue<pair<ll, int>> q;
  q.push({0, src});
  while (!q.empty()) {
    ll d = q.top().first;
    int v = q.top().second;
    q.pop();
    if (d > dists[v]) continue;
    for (auto edge : G[v]) {
      int to = edge.first;
      ll w = edge.second;
      if (d + w < dists[to]) {
        dists[to] = d + w;
        q.push({dists[to], to});
      }
    }
  }
}
```

lca

```
// calculates the lowest common ancestor for any two nodes in O(log N) time,
// with O(N log N) preprocessing
struct Lca {
  int L;
  vector<vector<int>> up;
  vector<pair<int, int>> time;

  Lca() {}
  void init(const vector<vector<int>>& G) {
    int N = G.size();
    L = N <= 1 ? 0 : 32 - __builtin_clz(N - 1);
    up.resize(L + 1);
    rep(l, L + 1) up[l].resize(N);
    time.resize(N);
    int t = 0;
    visit(G, 0, 0, t);
    rep(l, L) rep(i, N) up[l + 1][i] = up[l][up[l][i]];
  }

  void visit(const vector<vector<int>>& G, int i, int p, int& t) {
    up[0][i] = p;
    time[i].first = t++;
    for (int edge : G[i]) {
      if (edge == p) continue;
      visit(G, edge, i, t);
    }
    time[i].second = t++;
  }

  bool is_anc(int up, int dn) {
    return time[up].first <= time[dn].first &&
        time[dn].second <= time[up].second;
  }

  int get(int i, int j) {
    if (is_anc(i, j)) return i;
    if (is_anc(j, i)) return j;
    int l = L;
    while (l >= 0) {
      if (is_anc(up[l][i], j))
        l--;
      else
        i = up[l][i];
    }
    return up[0][i];
  }
};
```

matching

```
vector<bool> seen;
vector<int> mt;

bool subkuhn(vector<vector<int>>& adj, int i) {
  if (seen[i]) return false;
  seen[i] = true;
  for (int to : adj[i])
    if (mt[to] == -1 || subkuhn(adj, mt[to])) {
      mt[to] = i;
      return true;
    }
  return false;
}

// get a maximum matching out of a **bipartite** graph.
// returns the size of the matching (ie. the covered vertices).
// runs in O(n * m) time.
int kuhn(vector<vector<int>>& adj) {
  int N = adj.size(), total = 0;
  mt.assign(N, -1);
  rep(i, N) {
    seen.assign(N, -1);
    total += subkuhn(adj, i);
  }
  return total;
}
```

flow

```
const ll INF = 1e18;

struct Flow {
  vector<vector<ll>> cap, flow;
  vector<ll> excess;
  vector<int> height;

  Flow() {}
  void resize(int N) { cap.assign(N, vector<ll>(N)); }

  // push as much excess flow as possible from u to v.
  void push(int u, int v) {
    ll f = min(excess[u], cap[u][v] - flow[u][v]);
    flow[u][v] += f;
    flow[v][u] -= f;
    excess[v] += f;
    excess[u] -= f;
  }

  // relabel the height of a vertex so that excess flow may be pushed.
  void relabel(int u) {
    int d = INT32_MAX;
    rep(v, cap.size()) if (cap[u][v] - flow[u][v] > 0) d =
      min(d, height[v]);
    if (d < INF) height[u] = d + 1;
  }

  // get the maximum flow on the network specified by `cap` with source `s`
  // and sink `t`.
  // node-to-node flows are output to the `flow` member.
  //
  // time: O(V^2 sqrt(E)) <= O(V^3)
  // memory: O(V^2)
  ll maxflow(int s, int t) {
    int N = cap.size(), M;
    flow.assign(N, vector<ll>(N));
    height.assign(N, 0), height[s] = N;
    excess.assign(N, 0), excess[s] = INF;
    rep(i, N) if (i != s) push(s, i);

    vector<int> q;
    while (true) {
      // find the highest vertices with excess
      q.clear(), M = 0;
      rep(i, N) {
        if (excess[i] <= 0 || i == s || i == t) continue;
        if (height[i] > M) q.clear(), M = height[i];
        if (height[i] >= M) q.push_back(i);
      }
      if (q.empty()) break;
      // process vertices
      for (int u : q) {
        bool relab = true;
        rep(v, N) {
          if (excess[u] <= 0) break;
          if (cap[u][v] - flow[u][v] > 0 && height[u] > height[v])
            push(u, v), relab = false;
        }
        if (relab) {
          relabel(u);
          break;
        }
      }
    }

    ll f = 0;
    rep(i, N) f += flow[i][t];
    return f;
  }
};
```

hld

```
struct Hld {
  vector<int> parent, heavy, depth, pos, top;

  Hld() {}
  void init(vector<vector<int>>& G) {
    int N = G.size();
    parent.resize(N), heavy.resize(N), depth.resize(N), pos.resize(N),
      top.resize(N);
    depth[0] = -1, dfs(G, 0);
    int t = 0;
    rep(i, N) if (heavy[parent[i]] != i) {
      int j = i;
      while (j != -1) {
        top[j] = i, pos[j] = t++;
```

```
        j = heavy[j];
      }
    }
  }

  int dfs(vector<vector<int>>& G, int i) {
    int w = 1, mw = 0;
    depth[i] = depth[parent[i]] + 1, heavy[i] = -1;
    for (int c : G[i]) {
      if (c == parent[i]) continue;
      parent[c] = i;
      int sw = dfs(G, c);
      if (sw > mw) heavy[i] = c, mw = sw;
      w += sw;
    }
    return w;
  }

  template <class OP>
  void path(int u, int v, OP op) {
    while (top[u] != top[v]) {
      if (depth[top[u]] > depth[top[v]]) swap(u, v);
      op(pos[top[v]], pos[v]);
      v = parent[top[v]];
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(pos[u], pos[v]); // value on vertex
    // op(pos[u]+1, pos[v]); // value on path
  }

  // segment tree
  template <class T, class S>
  void update(S& seg, int i, T val) {
    seg.update(pos[i], val);
  }

  // segment tree lazy
  template <class T, class S>
  void update(S& seg, int u, int v, T val) {
    path(u, v, [&](int l, int r) { seg.update(l, r, val); });
  }

  template <class T, class S>
  T query(S& seg, int u, int v) {
    T ans = 0;
    path(u, v, [&](int l, int r) { ans += seg.query(l, r); }); // query op
    return ans;
  }
};
```

strings

sufarr

```
// build the suffix array
// suffixes are sorted, with each suffix represented by its starting position
vector<int> suffixarray(const string& s) {
  int N = s.size() + 1; // optional: include terminating NUL
  vector<int> p(N), p2(N), c(N), c2(N), cnt(256);
  rep(i, N) cnt[s[i]] += 1;
  repx(b, 1, 256) cnt[b] += cnt[b - 1];
  rep(i, N) p[--cnt[s[i]]] = i;
  repx(i, 1, N) c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
  for (int k = 1; k < N; k <<= 1) {
    int C = c[p[N - 1]] + 1;
    cnt.assign(C + 1, 0);
    for (int& pi : p) pi = (pi - k + N) % N;
    for (int cl : c) cnt[cl + 1] += 1;
    rep(i, C) cnt[i + 1] += cnt[i];
    rep(i, N) p2[cnt[c[p[i]]]++] = p[i];
    c2[p2[0]] = 0;
    repx(i, 1, N) c2[p2[i]] =
      c2[p2[i - 1]] + (c[p2[i]] != c[p2[i - 1]] ||
              c[(p2[i] + k) % N] != c[(p2[i - 1] + k) % N]);
    swap(c, c2), swap(p, p2);
  }
  p.erase(p.begin()); // optional: erase terminating NUL
  return p;
}

// build the lcp
// `lcp[i]` represents the length of the longest common prefix between
↪  suffix i
// and suffix i+1 in the suffix array `p`. the last element of `lcp` is zero
↪  by
// convention
vector<int> makelcp(const string& s, const vector<int>& p) {
  int N = p.size(), k = 0;
  vector<int> r(N), lcp(N);
  rep(i, N) r[p[i]] = i;
  rep(i, N) {
    if (r[i] + 1 >= N) {
      k = 0;
      continue;
    }
    int j = p[r[i] + 1];
    while (i + k < N && j + k < N && s[i + k] == s[j + k]) k += 1;
    lcp[r[i]] = k;
    if (k) k -= 1;
  }
  return lcp;
}
```

kmp

```
// compute the prefix function for string `s`:
// for every character substring [0 : i], compute the longest proper prefix
↪  that
// is also a suffix.
// O(N)
//
// computing `prefunc` on a string of the type `wwww#ttttttt` will give for
// every `t` the amount of characters from `w` that match (ie. search for the
// string `w` inside the string `t`).
void prefunc(const string& s, vector<int>& p) {
  int N = s.size(), j;
  p.resize(N), p[0] = 0;
  repx(i, 1, N) {
    for (j = p[i - 1]; j > 0 && s[j] != s[i];) j = p[j - 1];
    p[i] = j + (s[j] == s[i]);
  }
}
```

hash

```
// compute substring hashes in O(1).
// hashes are compatible between different strings.
struct Hash {
  ll HMOD;
  int N;
  vector<int> h;
  vector<int> p;

  Hash() {}
  // O(N)
  Hash(const string& s, ll HMOD_ = 1000003931)
    : N(s.size() + 1), HMOD(HMOD_), p(N), h(N) {
    static const ll P =
      chrono::steady_clock::now().time_since_epoch().count() % (1 << 29);
    p[0] = 1;
    rep(i, N - 1) p[i + 1] = p[i] * P % HMOD;
    rep(i, N - 1) h[i + 1] = (h[i] + (ll)s[i] * p[i]) % HMOD;
  }

  // O(1)
  pair<ll, int> get(int i, int j) { return {(h[j] - h[i] + HMOD) % HMOD, i}; }

  bool cmp(pair<ll, int> x0, pair<ll, int> x1) {
    int d = x0.second - x1.second;
    ll& lo = d < 0 ? x0.first : x1.first;
    lo = lo * p[abs(d)] % HMOD;
    return x0.first == x1.first;
  }
};

// compute hashes in multiple prime modulos simultaneously, to reduce the
↪  chance
// of collisions.
```

```
struct HashM {
  int N;
  vector<Hash> sub;

  HashM() {}
  // O(K N)
  HashM(const string& s, const vector<ll>& mods) : N(mods.size()), sub(N) {
    rep(i, N) sub[i] = Hash(s, mods[i]);
  }

  // O(K)
  vector<pair<ll, int>> get(int i, int j) {
    vector<pair<ll, int>> hs(N);
    rep(k, N) hs[k] = sub[k].get(i, j);
    return hs;
  }

  bool cmp(const vector<pair<ll, int>>& x0, const vector<pair<ll, int>>& x1) {
    rep(i, N) if (!sub[i].cmp(x0[i], x1[i])) return false;
    return true;
  }

  bool cmp(int i0, int j0, int i1, int j1) {
    rep(i, N) if (!sub[i].cmp(sub[i].get(i0, j0),
                  sub[i].get(i1, j1))) return false;
    return true;
  }
};
```

hash2d

```
using Hash = pair<ll, int>;

struct Block {
  int x0, y0, x1, y1;
};

struct Hash2d {
  ll HMOD;
  int W, H;
  vector<int> h;
  vector<int> p;

  Hash2d() {}
  Hash2d(const string& s, int W_, int H_, ll HMOD_ = 1000003931)
    : W(W_ + 1), H(H_ + 1), HMOD(HMOD_) {
    static const ll P =
      chrono::steady_clock::now().time_since_epoch().count() % (1 << 29);
    p.resize(W * H);
    p[0] = 1;
    rep(i, W * H - 1) p[i + 1] = p[i] * P % HMOD;
    h.assign(W * H, 0);
    repx(y, 1, H) repx(x, 1, W) {
      ll c = (ll)s[(y - 1) * (W - 1) + x - 1] * p[y * W + x] % HMOD;
      h[y * W + x] = (HMOD + h[y * W + x - 1] + h[(y - 1) * W + x] -
              h[(y - 1) * W + x - 1] + c) %
             HMOD;
    }
  }

  bool isout(Block s) {
    return s.x0 < 0 || s.x0 >= W || s.x1 < 0 || s.x1 >= W || s.y0 < 0 ||
        s.y0 >= H || s.y1 < 0 || s.y1 >= H;
  }

  Hash get(Block s) {
    return {(2 * HMOD + h[s.y1 * W + s.x1] - h[s.y1 * W + s.x0] -
         h[s.y0 * W + s.x1] + h[s.y0 * W + s.x0]) %
          HMOD,
        s.y0 * W + s.x0};
  }

  bool cmp(Hash x0, Hash x1) {
    int d = x0.second - x1.second;
    ll& lo = d < 0 ? x0.first : x1.first;
    lo = lo * p[abs(d)] % HMOD;
    return x0.first == x1.first;
  }
};

struct Hash2dM {
  int N;
  vector<Hash2d> sub;

  Hash2dM() {}
  Hash2dM(const string& s, int W, int H, const vector<ll>& mods)
    : N(mods.size()), sub(N) {
    rep(i, N) sub[i] = Hash2d(s, W, H, mods[i]);
  }

  bool isout(Block s) { return sub[0].isout(s); }

  vector<Hash> get(Block s) {
    vector<Hash> hs(N);
    rep(i, N) hs[i] = sub[i].get(s);
    return hs;
  }

  bool cmp(const vector<Hash>& x0, const vector<Hash>& x1) {
    rep(i, N) if (!sub[i].cmp(x0[i], x1[i])) return false;
    return true;
  }

  bool cmp(Block s0, Block s1) {
    rep(i, N) if (!sub[i].cmp(sub[i].get(s0), sub[i].get(s1))) return false;
    return true;
  }
};
```

palin

```
// find maximal palindromes (and therefore all palindromes) in O(n).
// returns a vector of positions, with one position for every character and
↪  in
// between characters.
//
// a   b   c   c   c
// 0 1 2 3 4 5 6 7 8
// 1 0 1 0 1 2 3 2 1
void manacher(const string& s, vector<int>& p) {
  int N = s.size(), P = 2 * N - 1;
  p.assign(P, 0);
  int l = 0, r = -1;
  rep(i, P) {
    int d = (r >= i ? min(p[l + r - i], r - i + 2) : i % 2);
    while (i - d >= 0 && i + d < P && s[(i - d) / 2] == s[(i + d) / 2])
      d += 2;
    p[i] = d;
    if (i + d - 2 > r) l = i - d + 2, r = i + d - 2;
  }
  rep(i, P) p[i] -= 1;
}
```

geo

2d

```
typedef double T;

struct P {
  T x;
  T y;

  P(T x_, T y_) : x{x_}, y{y_} {}
  P() : x{0}, y{0} {}

  friend ostream& operator<<(ostream& s, const P& self) {
    s << self.x << " " << self.y;
    return s;
  }
  friend istream& operator>>(istream& s, P& self) {
    s >> self.x;
    s >> self.y;
    return s;
  }

  P& operator+=(const P& r) {
    this->x += r.x;
    this->y += r.y;
```

```
    return *this;
  }
  friend P operator+(P l, const P& r) { return {l.x + r.x, l.y + r.y}; }

  P& operator-=(const P& r) {
    this->x -= r.x;
    this->y -= r.y;
    return *this;
  }
  friend P operator-(P l, const P& r) { return {l.x - r.x, l.y - r.y}; }
  P operator-() { return {-this->x, -this->y}; }

  P& operator*=(const T& r) {
    this->x *= r;
    this->y *= r;
    return *this;
  }
  friend P operator*(P l, const T& r) { return {l.x * r, l.y * r}; }
  friend P operator*(const T& l, P r) { return {l * r.x, l * r.y}; }

  P& operator/=(const T& r) {
    this->x /= r;
    this->y /= r;
    return *this;
  }
  friend P operator/(P l, const T& r) { return {l.x / r, l.y / r}; }

  // Dot product
  friend T operator*(P l, const P& r) { return l.x * r.x + l.y * r.y; }

  // Cross product (equiv to l.rotated() * r in 2D)
  friend T operator^(P l, const P& r) { return l.x * r.y - l.y * r.x; }

  T magsq() { return this->x * this->x + this->y * this->y; }
  T mag() { return sqrt(this->magsq()); }
  P unit() { return (1. / this->mag()) * (*this); }

  P rotated() { return {-this->y, this->x}; }

  double angle() { return atan2((double)this->y, (double)this->x); }
  float angle_float() { return atan2((float)this->y, (float)this->x); }
  static P from_angle(T angle) { return {(T)cos(angle), (T)sin(angle)}; }
};

// receives two segments in origin/distance format.
// the second segment is extended to infinity
// returns a weight, representing how much along the first line segment do
↪   the
// lines intersect.
// if it is in the [0, 1] range, they intersect.
T seg_line(P o0, P d0, P o1, P d1) {
  T d = d0 ^ d1;
  if (d == 0) return 0;
  return ((o1 - o0) ^ d0) / d;
}

// returns true if two segments intersect.
// handles all corner cases correctly, including parallel and zero
// sized segments.
// in the non-parallel case, the intersection point can be retrieved
// as `o0 + n0 / d * d0` or `o1 + n1 / d * d1`.
bool seg_seg(P o0, P d0, P o1, P d1) {
  P o = o1 - o0;
  T d = d0 ^ d1;
  if (d == 0) {
    if ((o ^ d0) != 0) return false;
    T e0 = o * d0, e1 = (o + d1) * d0;
    return (0 <= e1 && e0 <= d0 * d0) || (0 <= e0 && e1 <= d0 * d0);
  }
  T n0 = o ^ d0, n1 = o ^ d1;
  if (d < 0) n0 = -n0, n1 = -n1, d = -d;
  return 0 <= n0 && n0 <= d && 0 <= n1 && n1 <= d;
}

// iterate over all slopes, keeping points sorted with respect to the signed
// distance to the slope.
template <class OP>
void iter_slopes(vector<P>& points, OP op) {
  int N = points.size();
  vector<pair<int, int>> slopes;
  rep(i, N) rep(j, N) {
    if (i == j) continue;
    slopes.push_back({i, j});
  }
  vector<int> perms(N);
  rep(i, N) perms[i] = i;
  sort(slopes.begin(), slopes.end(), [&](pair<int, int> i, pair<int, int> j) {
    P d1 = points[i.second] - points[i.first];
    P d2 = points[j.second] - points[j.first];
    return (d1 ^ d2) > 0;
  });
  for (auto& s : slopes) {
    int i = perms[s.first], j = perms[s.second];
    op(i, j);
    swap(points[perms[i]], points[perms[j]]);
    swap(perms[i], perms[j]);
  }
}
```

3d

```
typedef double T;

struct Vec3 {
  T x;
  T y;
  T z;

  Vec3(T x_, T y_, T z_) : x{x_}, y{y_}, z{z_} {}
  Vec3() : x{0}, y{0}, z{0} {}

  friend ostream& operator<<(ostream& out, const Vec3& self) {
    out << "(" << self.x << ", " << self.y << ", " << self.x << ")";
    return out;
  }
  friend istream& operator>>(istream& in, Vec3& self) {
    in >> self.x;
    in >> self.y;
    in >> self.z;
    return in;
  }

  Vec3& operator+=(const Vec3& r) {
    this->x += r.x;
    this->y += r.y;
    this->z += r.z;
    return *this;
  }
  friend Vec3 operator+(Vec3 l, const Vec3& r) {
    return {l.x + r.x, l.y + r.y, l.z + r.z};
  }

  Vec3& operator-=(const Vec3& r) {
    this->x -= r.x;
    this->y -= r.y;
    this->z -= r.z;
    return *this;
  }
  friend Vec3 operator-(Vec3 l, const Vec3& r) {
    return {l.x - r.x, l.y - r.y, l.z - r.z};
  }
  Vec3 operator-() { return {-this->x, -this->y, -this->z}; }

  Vec3& operator*=(const T& r) {
    this->x *= r;
    this->y *= r;
    this->z *= r;
    return *this;
  }
  friend Vec3 operator*(Vec3 l, const T& r) {
    return {l.x * r, l.y * r, l.z * r};
  }
  friend Vec3 operator*(const T& l, Vec3 r) {
    return {l * r.x, l * r.y, l * r.z};
  }

  Vec3& operator/=(const T& r) {
    this->x /= r;
    this->y /= r;
    this->z /= r;
    return *this;
  }
  friend Vec3 operator/(Vec3 l, const T& r) {
    return {l.x / r, l.y / r, l.z / r};
  }

  // Dot product
  friend T operator*(Vec3 l, const Vec3& r) {
    return l.x * r.x + l.y * r.y + l.z * r.z;
  }

  // Cross product
  friend Vec3 operator^(Vec3 l, const Vec3& r) {
    return {l.y * r.z - r.y * l.z, l.x * r.z - r.x * l.z,
        l.x * r.y - r.x * l.y};
```

```
  }

  T magsq() {
    return this->x * this->x + this->y * this->y + this->z * this->z;
  }
  T mag() { return sqrt(this->magsq()); }
  Vec2 unit() { return (1. / this->mag()) * (*this); }
};
```

circle

```
struct Circle {
  Vec2 o;
  T r;
};

/// Find the pair of tangent points on circumference `c` to point `p`.
/// That is, find the tangent lines that cross point `p` and intersect the
/// circumference `c`, and return the points where these lines intersect `c`.
///
/// The first point returned is the counterclockwise tangent, followed by the
/// clockwise tangent.
///
/// If the point is inside the circle, NaN is returned.
pair<Vec2, Vec2> tangents(Circle c, Vec2 p) {
  Vec2 d = p - c.o;
  T r2d2 = c.r * c.r / d.magsq();
  Vec2 mid = c.o + r2d2 * d;
  Vec2 dif = sqrt(r2d2 * (1. - r2d2)) * d.rotated();
  return {mid + dif, mid - dif};
}
```

search

```
// searches for a value in an [l, r] range (both inclusive).
//
// the `isleft(m)` function evaluates whether `m` is strictly to the left of
↪   the
// target value.
int binsearch_left(int l, int r, bool isleft(int)) {
  while (l != r) {
    int m = (l + r) / 2;
    if (isleft(m)) {
      l = m + 1;
    } else {
      r = m;
    }
  }
  return l;
}

// searches for a value in an [l, r] range (both inclusive).
//
// the `isright(m)` function evaluates whether `m` is strictly to the right
↪   of
// the target value.
//
// note the `+1` when computing `m`, which avoids infinite loops.
// the only difference with `binsearch_left` is how the evaluation function
↪   is
// specified. both are functionally identical.
int binsearch_right(int l, int r, bool isright(int)) {
  while (l != r) {
    int m = (l + r + 1) / 2;
    if (isright(m)) {
      r = m - 1;
    } else {
      l = m;
    }
  }
  return l;
}

// continuous ternary (golden section) search.
//
// searches for a minimum value of the given unimodal function (monotonic
// positive derivative).
template <typename T, typename U>
pair<T, U> ctersearch(int iter, T a, T b, U f(T)) {
  const T INVG = 0.61803398874989484820;

  U av = f(a);
  U bv = f(b);
  T mid = a + (b - a) * INVG;
  U midv = f(mid);

  for (int i = 0; i < iter; i++) {
    T new_mid = a + (mid - a) * INVG;
    U new_midv = f(new_mid);
    if (new_midv > midv) {
      // Search the right interval
      a = b;
      av = bv;
      b = new_mid;
      bv = new_midv;
    } else {
      // Search the left interval
      b = mid;
      bv = midv;
      mid = new_mid;
      midv = new_midv;
    }
  }
  return {mid, midv};
}
```