.bashrc d41d

```
build() (
  g++ $@ -o $1.e -DLOC -std=c++11               \
      -Wall -Wextra -Wfatal-errors -Wshadow \
      -Wlogical-op -Wconversion -Wfloat-equal
)

b() ( build $@ -O2 )

d() ( build $@ -fsanitize=address,undefined \
              -D_GLIBCXX_DEBUG -g )

run() ( $1 $2 && echo start >&2 && time ./$2.e )

loo() (
  set -e; $1 $2; $1 $3
  for ((;;)) {
    ./$3.e > gen.in
    time ./$2.e < gen.in > gen.out
  }
)

cmp() (
  set -e; $1 $2; $1 $3; $1 $4
  for ((;;)) {
    ./$4.e > gen.in;            echo -n 0
    ./$2.e < gen.in > p1.out; echo -n 1
    ./$3.e < gen.in > p2.out; echo -n 2
    diff p1.out p2.out;         echo -n Y
  }
)

# Other flags:
# -Wformat=2 -Wshift-overflow=2 -Wcast-qual
# -Wcast-align -Wduplicated-cond
# -D_GLIBCXX_DEBUG_PEDANTIC -D_FORTIFY_SOURCE=2
# -fno-sanitize-recover -fstack-protector
```

.vimrc d41d

```
se ai aw cin cul ic is nocp nohls nu sc scs
se bg=dark sw=4 ts=4 so=7 ttm=9
sy on
ca hash w !cpp -dD -P -fpreprocessed \|        \
  tr -d '[:space:]' \| md5sum \| cut -c-4
```

template.cpp d41d

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

using ll  = long long;
using Vi  = vector<int>;
using Pii = pair<int,int>;

#define mp make_pair
#define pb push_back
#define x  first
#define y  second

#define rep(i,b,e)  for(int i=(b); i<(e); i++)
#define each(a,x)   for(auto& a : (x))
#define all(x)      (x).begin(),(x).end()
#define sz(x)       int((x).size())

int main() {
  cin.sync_with_stdio(0); cin.tie(0);
  cout << fixed << setprecision(18);

  // Don't call destructors:
  cout << flush; _Exit(0);
} // d41d

// > Debug printer

#define tem template<class t,class u,class...w>
#define pri(x,y,z)tem auto operator<<(t&o,u a)\
  ->decltype(x,o) { o << z; return o << y; }

pri(a.print(), '}', '{'; a.print())
pri(a.y, ')', '(' << a.x << ", " << a.y)

pri(all(a), ']', '['; auto d="";
  for (auto i : a) (o << d << i, d = ", "))

void DD(...) {}
tem void DD(t s, u a, w... k) {
  for (int b=1; *s && *s - b*44; cerr << *s++)
    b += 2 / (*s*2 - 81);
  cerr << ": " << a << *s++; DD(s, k...);
} // d41d

#ifdef LOC
#define deb(...) (DD("[,\b :] "#__VA_ARGS__, \
  __LINE__, __VA_ARGS__), cerr << endl)
#else
#define deb(...)
#endif

#define DBP(...) void print() { \
  DD(#__VA_ARGS__, __VA_ARGS__); } // d41d

// > Utils

// Return smallest k such that 2^k > n
// Undefined for n = 0!
int uplg(int n) { return 32-__builtin_clz(n); }
int uplg(ll n){ return 64-__builtin_clzll(n); }

// Compare with certain epsilon (branchless)
// Returns -1 if a < b; 1 if a > b; 0 if equal
// a and b are assumed equal if |a-b| <= eps
int cmp(double a, double b, double eps=1e-10) {
  return (a > b+eps) - (a+eps < b);
} // d41d
```

various.h d41d

```
// If math constants like M_PI are not found
// add this at the beginning of file
#define _USE_MATH_DEFINES

// Pragmas
#pragma GCC optimize("Ofast,unroll-loops,      \
                      no-stack-protector")
#pragma GCC target("popcnt,avx,tune=native")

// Clock
while (clock() < duration*CLOCKS_PER_SEC)

// Automatically implement operators:
// 1. != if == is defined
// 2. >, <= and >= if < is defined
using namespace rel_ops;

// Mersenne twister for randomization.
mt19937_64 rnd(chrono::steady_clock::now()
  .time_since_epoch().count());

// To shuffle randomly use:
shuffle(all(vec), rnd)

// To pick random integer from [A;B] use:
uniform_int_distribution<> dist(A, B);
int value = dist(rnd);

// To pick random real number from [A;B] use:
uniform_real_distribution<> dist(A, B);
double value = dist(rnd);
```

geometry/convex_hull.h d41d

```
#include "vec2.h"

// Translate points such that lower-left point
// is (0, 0). Returns old point location; O(n)
vec2 normPos(vector<vec2>& points) {
  auto q = points[0].yxPair();
  each(p, points) q = min(q, p.yxPair());
  vec2 ret{q.y, q.x};
  each(p, points) p = p-ret;
  return ret;
} // d41d

// Find convex hull of points; time: O(n lg n)
// Points are returned counter-clockwise,
// first point is the lowest-left.
vector<vec2> convexHull(vector<vec2> points) {
  vec2 pivot = normPos(points);
  sort(all(points));
  vector<vec2> hull;

  each(p, points) {
    while (sz(hull) >= 2) {
      vec2 a = hull.back() - hull[sz(hull)-2];
      vec2 b = p - hull.back();
      if (a.cross(b) > 0) break;
      hull.pop_back();
    } // d41d
    hull.pb(p);
  } // d41d

  // Translate back, optional
  each(p, hull) p = p+pivot;
  return hull;
} // d41d

// Find point p that minimizes dot product p*q.
// Returns point index in hull; time: O(lg n)
// If multiple points have same dot product
// one with smallest index is returned.
// Points are expected to be in the same order
// as output from convexHull function.
int minDot(const vector<vec2>& hull, vec2 q) {
  auto search = [&](int b, int e, vec2 p) {
    while (b+1 < e) {
      int m = (b+e) / 2;
      (p.dot(hull[m-1]) > p.dot(hull[m])
        ? b : e) = m;
    } // d41d
    return b;
  }; // d41d

  int m = search(0, sz(hull), {0, -1});
  int i = search(0, m, q);
  int j = search(m, sz(hull), q);
  return q.dot(hull[i]) > q.dot(hull[j])
    ? j : i;
} // d41d
```

geometry/convex_hull_dist.h d41d

```
#include "vec2.h"

// Check if p is inside convex polygon. Hull
// must be given in counter-clockwise order.
// Returns 2 if inside, 1 if on border,
// 0 if outside; time: O(n)
int insideHull(vector<vec2>& hull, vec2 p) {
  int ret = 1;
  rep(i, 0, sz(hull)) {
    auto v = hull[(i+1)%sz(hull)] - hull[i];
    auto t = v.cross(p-hull[i]);
    ret = min(ret, cmp(t, 0)); // For doubles
    //ret = min(ret, (t>0) - (t<0)); // Ints
  } // d41d
  return int(max(ret+1, 0));
} // d41d

#include "segment2.h"

// Get distance from point to hull; time: O(n)
double hullDist(vector<vec2>& hull, vec2 p) {
  if (insideHull(hull, p)) return 0;
  double ret = 1e30;
  rep(i, 0, sz(hull)) {
    seg2 seg{hull[(i+1)%sz(hull)], hull[i]};
    ret = min(ret, seg.distTo(p));
  } // d41d
  return ret;
} // d41d

// Compare distance from point to hull
// with sqrt(d2); time: O(n)
// -1 if smaller, 0 if equal, 1 if greater
int cmpHullDist(vector<vec2>& hull,
                vec2 p, ll d2) {
  if (insideHull(hull,p)) return (d2<0)-(d2>0);
  int ret = 1;
  rep(i, 0, sz(hull)) {
    seg2 seg{hull[(i+1)%sz(hull)], hull[i]};
    ret = min(ret, seg.cmpDistTo(p, d2));
  } // d41d
  return ret;
} // d41d
```

geometry/convex_hull_sum.h d41d

```
#include "vec2.h"

// Get edge sequence for given polygon
// starting from lower-left vertex; time: O(n)
// Returns start position.
vec2 edgeSeq(vector<vec2> points,
             vector<vec2>& edges) {
  int i = 0, n = sz(points);
  rep(j, 0, n) {
    if (points[i].yxPair()>points[j].yxPair())
```

```
      i = j;
  } // d41d
  rep(j, 0, n) edges.pb(points[(i+j+1)%n] -
                        points[(i+j)%n]);
  return points[i];
} // d41d

// Minkowski sum of given convex polygons.
// Vertices are required to be in
// counter-clockwise order; time: O(n+m)
vector<vec2> hullSum(vector<vec2> A,
                     vector<vec2> B) {
  vector<vec2> sum, e1, e2, es(sz(A) + sz(B));
  vec2 pivot = edgeSeq(A, e1) + edgeSeq(B, e2);
  merge(all(e1), all(e2), es.begin());

  sum.pb(pivot);
  each(e, es) sum.pb(sum.back() + e);
  sum.pop_back();
  return sum;
} // d41d
```

geometry/halfplanes.h d41d

```
#include "vec2.h"
#include "line2.h"

// Intersect given halfplanes and output
// hull vertices to out.
// Returns 0 if intersection is empty,
// 1 if intersection is non-empty and bounded,
// 2 if intersection is unbounded.
// Output vertices are valid ONLY IF
// intersection is non-empty and bounded.
// Works only with floating point vec2/line2.
// CURRENTLY DOESN'T WORK FOR NON-EMPTY
// AND UNBOUNDED CASES!
int intersectHalfPlanes(vector<line2> lines,
                        vector<vec2>& out) {
  deque<line2> H;
  out.clear();
  if (sz(lines) <= 1) return 2;

  sort(all(lines), [](line2 a, line2 b) {
    int t = cmp(a.norm.angle(),b.norm.angle());
    return t ? t < 0 : cmp(a.off*b.norm.len(),
      b.off*a.norm.len()) < 0;
  }); // d41d

  auto bad = [](line2 a, line2 b, line2 c) {
    if (cmp(a.norm.cross(c.norm), 0) <= 0)
      return false;
    vec2 p; assert(a.intersect(c, p));
    return b.side(p) <= 0;
  }; // d41d

  each(e, lines) {
    if (!H.empty() &&
      !cmp(H.back().norm.angle(),
      e.norm.angle())) continue;
    while (sz(H) > 1 && bad(H[sz(H)-2],
      H.back(), e)) H.pop_back();
    while (sz(H) > 1 && bad(e, H[0], H[1]))
      H.pop_front();
    H.pb(e);
  } // d41d

  while (sz(H) > 2 && bad(H[sz(H)-2],
    H.back(), H[0])) H.pop_back();
  while (sz(H) > 2 && bad(H.back(),
    H[0], H[1])) H.pop_front();

  out.resize(sz(H));

  rep(i, 0, sz(H)) {
    auto a = H[i], b = H[(i+1)%sz(H)];
    if (a.norm.cross(b.norm) <= 0)
      return cmp(a.off*b.norm.len(),
        -b.off*a.norm.len()) <= 0 ? 0 : 2;
    assert(a.intersect(b, out[i]));
  } // d41d

  rep(i, 0, sz(H)) {
    auto a = out[i], b = out[(i+1)%sz(H)];
    if (H[i].norm.perp().cross(b-a) <= 0)
      return 0;
  } // d41d
  return 1;
} // d41d
```

geometry/line2.h d41d

```
#include "vec2.h"

// 2D line/halfplane structure
// PARTIALLY TESTED

// Base class of versions for ints and doubles
template<class T, class P, class S>
struct bline2 {
  // For lines: norm*point == off
  // For halfplanes: norm*point <= off
  // (i.e. normal vector points outside)
  P norm; // Normal vector [A; B]
  T off;  // Offset (C parameter of equation)
  DBP(norm, off);

  // Line through 2 points; normal vector
  // points to the right of ab vector
  static S through(P a, P b) {
    return { (a-b).perp(), a.cross(b) };
  } // d41d

  // Parallel line through point
  static S parallel(P a, S b) {
    return { b.norm, b.norm.dot(a) };
  } // d41d

  // Perpendicular line through point
  static S perp(P a, S b) {
    return { b.norm.perp(), b.norm.cross(a) };
  } // d41d

  // Distance from point to line
  double distTo(P a) {
    return fabs(norm.dot(a)-off) / norm.len();
  } // d41d
}; // d41d

// Version for integer coordinates (long long)
struct line2i : bline2<ll, vec2i, line2i> {
  line2i() : bline2{{}, 0} {}
  line2i(vec2i n, ll c) : bline2{n, c} {}

  // Returns 0 if point a lies on the line,
  // 1 if on side where normal vector points,
  // -1 if on the other side.
  int side(vec2i a) {
    ll d = norm.dot(a);
    return (d > off) - (d < off);
  } // d41d
}; // d41d

// Version for double coordinates
// Requires cmp() from template
struct line2d : bline2<double, vec2d, line2d> {
  line2d() : bline2{{}, 0} {}
  line2d(vec2d n, double c) : bline2{n, c} {}

  // Returns 0 if point a lies on the line,
  // 1 if on side where normal vector points,
  // -1 if on the other side.
  int side(vec2d a) {
    return cmp(norm.dot(a), off);
  } // d41d

  // Intersect this line with line a, returns
  // true on success (false if parallel).
  // Intersection point is saved to `out`.
  bool intersect(line2d a, vec2d& out) {
    double d = norm.cross(a.norm);
    if (cmp(d, 0) == 0) return false;
    out = (norm*a.off-a.norm*off).perp() / d;
    return true;
  } // d41d
}; // d41d

using line2 = line2d;
```

geometry/rmst.h d41d

```
#include "../structures/find_union.h"

// Rectilinear Minimum Spanning Tree
// (MST in Manhattan metric); time: O(n lg n)
// Returns MST weight. Outputs spanning tree
// to G, vertex indices match point indices.
// Edge in G is pair (target, weight).
ll rmst(vector<Pii>& points,
        vector<vector<Pii>>& G) {
  int n = sz(points);
  vector<pair<int, Pii>> edges;
  vector<Pii> close;
  Vi ord(n), merged(n);
  iota(all(ord), 0);

  function<void(int,int)> octant =
      [&](int begin, int end) {
    if (begin+1 >= end) return;

    int mid = (begin+end) / 2;
    octant(begin, mid);
    octant(mid, end);

    int j = mid;
    Pii best = {INT_MAX, -1};
    merged.clear();

    rep(i, begin, mid) {
      int v = ord[i];
      Pii p = points[v];

      while (j < end) {
        int e = ord[j];
        Pii q = points[e];
        if (q.x-q.y > p.x-p.y) break;
        best = min(best,make_pair(q.x+q.y, e));
        merged.pb(e);
        j++;
      } // d41d

      if (best.y != -1) {
        int alt = best.x-p.x-p.y;
        if (alt < close[v].x)
          close[v] = {alt, best.y};
      } // d41d
      merged.pb(v);
    } // d41d

    while (j < end) merged.pb(ord[j++]);
    copy(all(merged), ord.begin()+begin);
  }; // d41d

  rep(i, 0, 4) {
    rep(j, 0, 2) {
      sort(all(ord), [&](int l, int r) {
        return points[l] < points[r];
      }); // d41d
      close.assign(n, {INT_MAX, -1});
      octant(0, n);
      rep(k, 0, n) {
        Pii p = close[k];
        if (p.y != -1) edges.pb({p.x,{k,p.y}});
        points[k].x *= -1;
      } // d41d
    } // d41d
    each(p, points) p = {p.y, -p.x};
  } // d41d

  ll sum = 0;
  FAU fau(n);
  sort(all(edges));
  G.assign(n, {});

  each(e, edges) if (fau.join(e.y.x, e.y.y)) {
    sum += e.x;
    G[e.y.x].pb({e.y.y, e.x});
    G[e.y.y].pb({e.y.x, e.x});
  } // d41d
  return sum;
} // d41d
```

geometry/segment2.h d41d

```
#include "vec2.h"

// 2D segment structure; NOT HEAVILY TESTED

// Base class of versions for ints and doubles
template<class P, class S> struct bseg2 {
  P a, b; // Endpoints

  // Distance from segment to point
  double distTo(P p) const {
    if ((p-a).dot(b-a) < 0) return (p-a).len();
    if ((p-b).dot(a-b) < 0) return (p-b).len();
    return double(abs((p-a).cross(b-a)))
                  / (b-a).len();
```

```
  } // d41d
}; // d41d

// Version for integer coordinates (long long)
struct seg2i : bseg2<vec2i, seg2i> {
  seg2i() {}
  seg2i(vec2i c, vec2i d) : bseg2{c, d} {}

  // Check if segment contains point p
  bool contains(vec2i p) {
    return (a-p).dot(b-p) <= 0 &&
           (a-p).cross(b-p) == 0;
  } // d41d

  // Compare distance to p with sqrt(d2)
  // -1 if smaller, 0 if equal, 1 if greater
  int cmpDistTo(vec2i p, ll d2) const {
    if ((p-a).dot(b-a) < 0) {
      ll l = (p-a).len2();
      return (l > d2) - (l < d2);
    } // d41d
    if ((p-b).dot(a-b) < 0) {
      ll l = (p-b).len2();
      return (l > d2) - (l < d2);
    } // d41d

    ll c = abs((p-a).cross(b-a));
    d2 *= (b-a).len2();
    return (c*c > d2) - (c*c < d2);
  } // d41d
}; // d41d

// Version for double coordinates
// Requires cmp() from template
struct seg2d : bseg2<vec2d, seg2d> {
  seg2d() {}
  seg2d(vec2d c, vec2d d) : bseg2{c, d} {}

  bool contains(vec2d p) {
    return cmp((a-p).dot(b-p), 0) <= 0 &&
           cmp((a-p).cross(b-p), 0) == 0;
  } // d41d
}; // d41d

using seg2 = seg2d;
```

geometry/vec2.h d41d

```
// 2D point/vector structure; PARTIALLY TESTED

// Base class of versions for ints and doubles
template<class T, class S> struct bvec2 {
  T x, y;
  S operator+(S r) const {return{x+r.x,y+r.y};}
  S operator-(S r) const {return{x-r.x,y-r.y};}
  S operator*(T r) const { return {x*r, y*r}; }
  S operator/(T r) const { return {x/r, y/r}; }

  T dot(S r)   const { return x*r.x+y*r.y; }
  T cross(S r) const { return x*r.y-y*r.x; }
  T len2()     const { return x*x + y*y; }
  double len() const { return sqrt(len2()); }
  S perp()     const { return {-y,x}; } //90deg

  pair<T, T> yxPair() const { return {y,x}; }

  double angle() const { //[0;2*PI] CCW from OX
    double a = atan2(y, x);
    return (a < 0 ? a+2*M_PI : a);
  } // d41d
}; // d41d

// Version for integer coordinates (long long)
struct vec2i : bvec2<ll, vec2i> {
  vec2i() : bvec2{0, 0} {}
  vec2i(ll a, ll b) : bvec2{a, b} {}

  bool operator==(vec2i r) const {
    return x == r.x && y == r.y;
  } // d41d

  // Compare by angle, length if angles equal
  bool operator<(vec2i r) const {
    if (upper() != r.upper()) return upper();
    auto t = cross(r);
    return t > 0 || (!t && len2() < r.len2());
  } // d41d

  bool upper() const {
    return y > 0 || (y == 0 && x >= 0);
  } // d41d
}; // d41d

// Version for double coordinates
// Requires cmp() from template
struct vec2d : bvec2<double, vec2d> {
  vec2d() : bvec2{0, 0} {}
  vec2d(double a, double b) : bvec2{a, b} {}

  vec2d unit() const { return *this/len(); }
  vec2d rotate(double a) const { // CCW
    return {x*cos(a) - y*sin(a),
            x*sin(a) + y*cos(a)}; // d41d
  } // d41d

  bool operator==(vec2d r) const {
    return !cmp(x, r.x) && !cmp(y, r.y);
  } // d41d

  // Compare by angle, length if angles equal
  bool operator<(vec2d r) const {
    int t = cmp(angle(), r.angle());
    return t < 0 || (!t && len2()<r.len2());
  } // d41d
}; // d41d

using vec2 = vec2d;
```

graphs/2sat.h d41d

```
// 2-SAT solver; time: O(n+m), space: O(n+m)
// Variables are indexed from 1 and
// negative indices represent negations!
// Usage: SAT2 sat(variable_count);
// (add constraints...)
// bool solution_found = sat.solve();
// sat[i] = value of i-th variable, 0 or 1
//          (also indexed from 1!)
// (internally: positive = i*2-1, neg. = i*2-2)
struct SAT2 : Vi {
  vector<Vi> G;
  Vi order, flags;

  // Init n variables, you can add more later
  SAT2(int n = 0) : G(n*2) {}

  // Add new var and return its index
  int addVar() {
    G.resize(sz(G)+2); return sz(G)/2;
  } // d41d

  // Add (i => j) constraint
  void imply(int i, int j) {
    i = max(i*2-1, -i*2-2);
    j = max(j*2-1, -j*2-2);
    G[i].pb(j); G[j^1].pb(i^1);
  } // d41d

  // Add (i v j) constraint
  void either(int i, int j) { imply(-i, j); }

  // Constraint at most one true variable
  void atMostOne(Vi& vars) {
    int x = addVar();
    each(i, vars) {
      int y = addVar();
      imply(x, y); imply(i, -x); imply(i, y);
      x = y;
    } // d41d
  } // d41d

  // Solve and save assignments in `values`
  bool solve() { // O(n+m), Kosaraju is used
    assign(sz(G)/2+1, -1);
    flags.assign(sz(G), 0);
    rep(i, 0, sz(G)) dfs(i);
    while (!order.empty()) {
      if (!propag(order.back()^1, 1)) return 0;
      order.pop_back();
    } // d41d
    return 1;
  } // d41d

  void dfs(int i) {
    if (flags[i]) return;
    flags[i] = 1;
    each(e, G[i]) dfs(e);
    order.pb(i);
  } // d41d

  bool propag(int i, bool first) {
    if (!flags[i]) return 1;
    flags[i] = 0;
    if (at(i/2+1) >= 0) return first;
    at(i/2+1) = i&1;
    each(e, G[i]) if (!propag(e, 0)) return 0;
    return 1;
  } // d41d
}; // d41d
```

graphs/bellman_ineq.h d41d

```
struct Ineq {
  ll a, b, c; // a - b >= c
}; // d41d

// Solve system of inequalities of form a-b>=c
// using Bellman-Ford; time: O(n*m)
bool solveIneq(vector<Ineq>& edges,
               vector<ll>& vars) {
  rep(i, 0, sz(vars)) each(e, edges)
    vars[e.b] = min(vars[e.b], vars[e.a]-e.c);
  each(e, edges)
    if (vars[e.a]-e.c < vars[e.b]) return 0;
  return 1;
} // d41d
```

graphs/biconnected.h d41d

```
// Biconnected components; time: O(n+m)
// Usage: Biconnected bi(graph);
// bi[v] = indices of components containing v
// bi.verts[i] = vertices of i-th component
// bit.edges[i] = edges of i-th component
// Bridges <=> components with 2 vertices
// Articulation points <=> vertices that belong
//                         to > 1 component
// Isolated vertex <=> empty component list
struct Biconnected : vector<Vi> {
  vector<Vi> verts;
  vector<vector<Pii>> edges;
  vector<Pii> S;

  Biconnected() {}

  Biconnected(vector<Vi>& G) : S(sz(G)) {
    resize(sz(G));
    rep(i, 0, sz(G)) if (!S[i].x) dfs(G,i,-1);
    rep(c, 0, sz(verts)) each(v, verts[c])
      at(v).pb(c);
  } // d41d

  int dfs(vector<Vi>& G, int v, int p) {
    int low = S[v].x = sz(S)-1;
    S.pb({v, -1});

    each(e, G[v]) if (e != p) {
      if (S[e].x < S[v].x) S.pb({v, e});
      low = min(low, S[e].x ?: dfs(G, e, v));
    } // d41d

    if (p != -1 && low >= S[p].x) {
      verts.pb({p}); edges.pb({});
      rep(i, S[v].x, sz(S)) {
        if (S[i].y == -1)
          verts.back().pb(S[i].x);
        else
          edges.back().pb(S[i]);
      } // d41d
      S.resize(S[v].x);
    } // d41d

    return low;
  } // d41d
}; // d41d
```

graphs/bridges_online.h d41d

```
// Dynamic 2-edge connectivity queries
// Usage: Bridges bridges(vertex_count);
// - bridges.addEdge(u, v); - add edge (u, v)
// - bridges.cc[v] = connected component ID
// - bridges.bi(v) = 2-edge connected comp ID
struct Bridges {
  vector<Vi> G; // Spanning forest
  Vi cc, size, par, bp, seen;
  int cnt{0};
```

```
  // Initialize structure for n vertices; O(n)
  Bridges(int n = 0) : G(n), cc(n), size(n, 1),
                       par(n, -1), bp(n, -1),
                       seen(n) {
    iota(all(cc), 0);
  } // d41d

  // Add edge (u, v); time: amortized O(lg n)
  void addEdge(int u, int v) {
    if (cc[u] == cc[v]) {
      int r = lca(u, v);
      while ((v = root(v)) != r)
        v = bp[bi(v)] = par[v];
      while ((u = root(u)) != r)
        u = bp[bi(u)] = par[u];
    } else {
      G[u].pb(v); G[v].pb(u);
      if (size[cc[u]] > size[cc[v]]) swap(u,v);
      size[cc[v]] += size[cc[u]];
      dfs(u, v);
    } // d41d
  } // d41d

  // Get 2-edge connected component ID
  int bi(int v) { // amortized time: < O(lg n)
    return bp[v] == -1 ? v : bp[v] = bi(bp[v]);
  } // d41d

  int root(int v) {
    return par[v] == -1 || bi(par[v]) != bi(v)
      ? v : par[v] = root(par[v]);
  } // d41d

  void dfs(int v, int p) {
    par[v] = p; cc[v] = cc[p];
    each(e, G[v]) if (e != p) dfs(e, v);
  } // d41d

  int lca(int u, int v) { // Don't use this!
    for (cnt++;; swap(u, v)) if (u != -1) {
      if (seen[u = root(u)] == cnt) return u;
      seen[u] = cnt; u = par[u];
    } // d41d
  } // d41d
}; // d41d
```

graphs/dense_dfs.h d41d

```
#include "../math/bit_matrix.h"

// DFS over bit-packed adjacency matrix
// G = NxN adjacency matrix of graph
//     G(i,j) <=> (i,j) is edge
// V = 1xN matrix containing unvisited vertices
//     V(0,i) <=> i-th vertex is not visited
// Total DFS time: O(n^2/64)
struct DenseDFS {
  BitMatrix G, V; // space: O(n^2/64)

  // Initialize structure for n vertices
  DenseDFS(int n = 0) : G(n, n), V(1, n) {
    reset();
  } // d41d

  // Mark all vertices as unvisited
  void reset() { each(x, V.M) x = -1; }

  // Get/set visited flag for i-th vertex
  void setVisited(int i) { V.set(0, i, 0); }
  bool isVisited(int i)  { return !V(0, i); }

  // DFS step: func is called on each unvisited
  // neighbour of i. You need to manually call
  // setVisited(child) to mark it visited
  // or this function will call the callback
  // with the same vertex again.
  template<class T>
  void step(int i, T func) {
    ull* E = G.row(i);
    for (int w = 0; w < G.stride;) {
      ull x = E[w] & V.row(0)[w];
      if (x) func((w<<6) | __builtin_ctzll(x));
      else   w++;
    } // d41d
  } // d41d
}; // d41d
```

graphs/edmonds_karp.h d41d

```
using flow_t = int;
constexpr flow_t INF = 1e9+10;

// Edmonds-Karp algorithm for finding
// maximum flow in graph; time: O(V*E^2)
// NOT HEAVILY TESTED
struct MaxFlow {
  struct Edge {
    int dst, inv;
    flow_t flow, cap;
  }; // d41d

  vector<vector<Edge>> G;
  vector<flow_t> add;
  Vi prev;

  // Initialize for n vertices
  MaxFlow(int n = 0) : G(n) {}

  // Add new vertex
  int addVert() {
    G.emplace_back(); return sz(G)-1;
  } // d41d

  // Add edge between u and v with capacity cap
  // and reverse capacity rcap
  void addEdge(int u, int v,
               flow_t cap, flow_t rcap = 0) {
    G[u].pb({ v, sz(G[v]), 0, cap });
    G[v].pb({ u, sz(G[u])-1, 0, rcap });
  } // d41d

  // Compute maximum flow from src to dst.
  // Flow values can be found in edges,
  // vertices with `add` >= 0 belong to
  // cut component containing `s`.
  flow_t maxFlow(int src, int dst) {
    flow_t f = 0;
    each(v, G) each(e, v) e.flow = 0;

    do {
      queue<int> Q;
      Q.push(src);
      prev.assign(sz(G), -1);
      add.assign(sz(G), -1);
      add[src] = INF;

      while (!Q.empty()) {
        int i = Q.front();
        flow_t m = add[i];
        Q.pop();

        if (i == dst) {
          while (i != src) {
            auto& e = G[i][prev[i]];
            e.flow -= m;
            G[e.dst][e.inv].flow += m;
            i = e.dst;
          } // d41d
          f += m;
          break;
        } // d41d

        each(e, G[i]) if (add[e.dst] < 0) {
          if (e.flow < e.cap) {
            Q.push(e.dst);
            prev[e.dst] = e.inv;
            add[e.dst] = min(m, e.cap-e.flow);
          } // d41d
        } // d41d
      } // d41d
    } while (prev[dst] != -1);

    return f;
  } // d41d

  // Get if v belongs to cut component with src
  bool cutSide(int v) {
    return add[v] >= 0;
  } // d41d
}; // d41d
```

graphs/gomory_hu.h d41d

```
#include "edmonds_karp.h"
//#include "push_relabel.h" // if you need

struct Edge {
  int a, b; // vertices
  flow_t w; // weight
}; // d41d

// Build Gomory-Hu tree; time: O(n*maxflow)
// Gomory-Hu tree encodes minimum cuts between
// all pairs of vertices: mincut for u and v
// is equal to minimum on path from u and v
// in Gomory-Hu tree. n is vertex count.
// Returns vector of Gomory-Hu tree edges.
vector<Edge> gomoryHu(vector<Edge>& edges,
                      int n) {
  MaxFlow flow(n);
  each(e, edges) flow.addEdge(e.a,e.b,e.w,e.w);

  vector<Edge> ret(n-1);
  rep(i, 1, n) ret[i-1] = {i, 0, 0};

  rep(i, 1, n) {
    ret[i-1].w = flow.maxFlow(i, ret[i-1].b);
    rep(j, i+1, n)
      if (ret[j-1].b == ret[i-1].b &&
          flow.cutSide(j)) ret[j-1].b = i;
  } // d41d
  return ret;
} // d41d
```

graphs/matroids.h d41d

```
// Find largest subset S of [n] such that
// S is independent in both matroid A and B.
// A and B are given by their oracles,
// see example implementations below.
// Returns vector V such that V[i] = 1 iff
// i-th element is included in found set;
// time: O(r^2*init + r^2*n*add),
// where r is max independent set,
// `init` is max time of oracles init
// and `add` is max time of oracles canAdd.
template<class T, class U>
vector<bool> intersectMatroids(T& A, U& B,
                               int n) {
  vector<bool> ans(n);
  bool ok = 1;

  A.init(ans);
  B.init(ans);
  rep(i, 0, n) if (A.canAdd(i) && B.canAdd(i))
    ans[i] = 1, A.init(ans), B.init(ans);

  while (ok) {
    vector<Vi> G(n);
    vector<bool> good(n);
    queue<int> que;
    Vi prev(n, -1);

    A.init(ans);
    B.init(ans);
    ok = 0;

    rep(i, 0, n) if (!ans[i]) {
      if (A.canAdd(i)) que.push(i), prev[i]=-2;
      good[i] = B.canAdd(i);
    } // d41d

    rep(i, 0, n) if (ans[i]) {
      ans[i] = 0;
      A.init(ans);
      B.init(ans);
      rep(j, 0, n) if (i != j && !ans[j]) {
        if (A.canAdd(j)) G[i].pb(j);
        if (B.canAdd(j)) G[j].pb(i);
      } // d41d
      ans[i] = 1;
    } // d41d

    while (!que.empty()) {
      int i = que.front();
      que.pop();

      if (good[i]) {
        ans[i] = 1;
        while (prev[i] >= 0) {
          ans[i = prev[i]] = 0;
          ans[i = prev[i]] = 1;
        } // d41d
        ok = 1;
        break;
      } // d41d
```

```
      each(j, G[i]) if (prev[j] == -1)
        que.push(j), prev[j] = i;
    } // d41d
  } // d41d

  return ans;
} // d41d

// Matroid where each element has color
// and set is independent iff for each color c
// #{elements of color c} <= maxAllowed[c].
struct LimOracle {
  Vi color; // color[i] = color of i-th element
  Vi maxAllowed; // Limits for colors
  Vi tmp;

  // Init oracle for independent set S; O(n)
  void init(vector<bool>& S) {
    tmp = maxAllowed;
    rep(i, 0, sz(S)) tmp[color[i]] -= S[i];
  } // d41d

  // Check if S+{k} is independent; time: O(1)
  bool canAdd(int k) {
    return tmp[color[k]] > 0;
  } // d41d
}; // d41d

// Graphic matroid - each element is edge,
// set is independent iff subgraph is acyclic.
struct GraphOracle {
  vector<Pii> elems; // Ground set: graph edges
  int n; // Number of vertices, indexed [0;n-1]
  Vi par;

  int find(int i) {
    return par[i] == -1 ? i
      : par[i] = find(par[i]);
  } // d41d

  // Init oracle for independent set S; ~O(n)
  void init(vector<bool>& S) {
    par.assign(n, -1);
    rep(i, 0, sz(S)) if (S[i])
      par[find(elems[i].x)] = find(elems[i].y);
  } // d41d

  // Check if S+{k} is independent; time: ~O(1)
  bool canAdd(int k) {
    return
      find(elems[k].x) != find(elems[k].y);
  } // d41d
}; // d41d

// Co-graphic matroid - each element is edge,
// set is independent iff after removing edges
// from graph number of connected components
// doesn't change.
struct CographOracle {
  vector<Pii> elems; // Ground set: graph edges
  int n; // Number of vertices, indexed [0;n-1]
  vector<Vi> G;
  Vi pre, low;
  int cnt;

  int dfs(int v, int p) {
    pre[v] = low[v] = ++cnt;
    each(e, G[v]) if (e != p)
      low[v] = min(low[v], pre[e] ?: dfs(e,v));
    return low[v];
  } // d41d

  // Init oracle for independent set S; O(n)
  void init(vector<bool>& S) {
    G.assign(n, {});
    pre.assign(n, 0);
    low.resize(n);
    cnt = 0;
    rep(i, 0, sz(S)) if (!S[i]) {
      Pii e = elems[i];
      G[e.x].pb(e.y);
      G[e.y].pb(e.x);
    } // d41d
    rep(v, 0, n) if (!pre[v]) dfs(v, -1);
  } // d41d

  // Check if S+{k} is independent; time: O(1)
  bool canAdd(int k) {
    Pii e = elems[k];
    return max(pre[e.x], pre[e.y])
      != max(low[e.x], low[e.y]);
  } // d41d
}; // d41d

// Matroid equivalent to linear space with XOR
struct XorOracle {
  vector<ll> elems; // Ground set: numbers
  vector<ll> base;

  // Init for independent set S; O(n+r^2)
  void init(vector<bool>& S) {
    base.assign(63, 0);
    rep(i, 0, sz(S)) if (S[i]) {
      ll e = elems[i];
      rep(j, 0, sz(base)) if ((e >> j) & 1) {
        if (!base[j]) {
          base[j] = e;
          break;
        } // d41d
        e ^= base[j];
      } // d41d
    } // d41d
  } // d41d

  // Check if S+{k} is independent; time: O(r)
  bool canAdd(int k) {
    ll e = elems[k];
    rep(i, 0, sz(base)) if ((e >> i) & 1) {
      if (!base[i]) return 1;
      e ^= base[i];
    } // d41d
    return 0;
  } // d41d
}; // d41d
```

graphs/push_relabel.h d41d

```
using flow_t = int;
constexpr flow_t INF = 1e9+10;

// Push-relabel algorithm for maximum flow;
// O(V^2*sqrt(E)), but very fast in practice.
// Preflow is not converted to flow!

// UNTESTED
struct MaxFlow {
  struct Edge {
    int to, inv;
    flow_t rem, cap;
  }; // d41d

  vector<basic_string<Edge>> G;
  vector<flow_t> extra;
  Vi hei, arc, prv, nxt, act, bot;
  queue<int> Q;
  int n, high, cut, work;

  // Initialize for n vertices
  MaxFlow(int k = 0) : G(k) {}

  // Add new vertex
  int addVert() {
    G.emplace_back();
    return sz(G)-1;
  } // d41d

  // Add edge between u and v with
  // capacity cap and reverse capacity rcap
  void addEdge(int u, int v,
               flow_t cap, flow_t rcap = 0) {
    G[u].pb({ v, sz(G[v]), 0, cap });
    G[v].pb({ u, sz(G[u])-1, 0, rcap });
  } // d41d

  void raise(int v, int h) {
    if (hei[v] < n)
      prv[nxt[prv[v]] = nxt[v]] = prv[v];
    if ((hei[v] = h) < n) {
      if (extra[v] > 0) {
        bot[v] = act[h], act[h] = v;
        high = max(high, h);
      } // d41d
      cut = max(cut, h);
      nxt[v] = nxt[prv[v] = h += n];
      prv[nxt[nxt[h] = v]] = v;
    } // d41d
  } // d41d

  void global(int t) {
    hei.assign(n, n);
    act.assign(n, -1);
    iota(all(prv), 0);
    iota(all(nxt), 0);
    hei[t] = high = cut = work = 0;
    for (Q.push(t); !Q.empty(); Q.pop()) {
      int v = Q.front();
      each(e, G[v])
        if (hei[e.to]==n && G[e.to][e.inv].rem)
          Q.push(e.to), raise(e.to, hei[v]+1);
    } // d41d
  } // d41d

  void push(int v, Edge& e) {
    auto f = min(extra[v], e.rem);
    if (f > 0) {
      if (!extra[e.to]) {
        bot[e.to] = act[hei[e.to]];
        act[hei[e.to]] = e.to;
      } // d41d
      e.rem -= f; G[e.to][e.inv].rem += f;
      extra[v] -= f; extra[e.to] += f;
    } // d41d
  } // d41d

  void discharge(int v) {
    int h = n;

    auto go = [&](int a, int b) {
      rep(i, a, b) {
        auto& e = G[v][i];
        if (e.rem) {
          if (hei[v] == hei[e.to]+1) {
            push(v, e);
            if (extra[v] <= 0)
              return arc[v] = i, 0;
          } else h = min(h, hei[e.to]+1);
        } // d41d
      } // d41d
      return 1;
    }; // d41d

    if (go(arc[v], sz(G[v])) && go(0, arc[v])){
      int k = hei[v] + n;
      if (nxt[k] == prv[k]) {
        rep(j, k, cut+n+1)
          while (nxt[j] < n) raise(nxt[j], n);
        cut = k-n-1;
      } else raise(v, h), work++;
    } // d41d
  } // d41d

  // Compute maximum flow from src to dst
  flow_t maxFlow(int src, int dst) {
    extra.assign(n = sz(G), 0);
    hei.assign(n, 0);
    arc.assign(n, 0);
    prv.resize(n*2);
    nxt.resize(n*2);
    act.resize(n);
    bot.resize(n);
    each(v, G) each(e, v) e.rem = e.cap;

    extra[dst] = -(extra[src] = INF);
    each(e, G[src]) push(src, e);
    global(dst);

    for (; high; high--)
      while (act[high] != -1) {
        int v = act[high];
        act[high] = bot[v];
        if (hei[v] == high) {
          discharge(v);
          if (work > 4*n) global(dst);
        } // d41d
      } // d41d

    return extra[dst] + INF;
  } // d41d

  // Get if v belongs to cut component with src
  bool cutSide(int v) { return hei[v] >= n; }
}; // d41d
```

graphs/scc.h d41d

```
// Tarjan's SCC algorithm; time: O(n+m)
// Usage: SCC scc(graph);
```

```
// scc[v] = index of SCC for vertex v
// scc.comps[i] = vertices of i-th SCC
// Components are in reversed topological order
struct SCC : Vi {
  vector<Vi> comps;
  Vi S;

  SCC() {}

  SCC(vector<Vi>& G) : Vi(sz(G),-1), S(sz(G)) {
    rep(i, 0, sz(G)) if (!S[i]) dfs(G, i);
  } // d41d

  int dfs(vector<Vi>& G, int v) {
    int low = S[v] = sz(S);
    S.pb(v);

    each(e, G[v]) if (at(e) < 0)
      low = min(low, S[e] ?: dfs(G, e));

    if (low == S[v]) {
      comps.pb({});
      rep(i, S[v], sz(S)) {
        at(S[i]) = sz(comps)-1;
        comps.back().pb(S[i]);
      } // d41d
      S.resize(S[v]);
    } // d41d

    return low;
  } // d41d
}; // d41d
```

graphs/turbo_matching.h d41d

```
// Find maximum bipartite matching; time: ?
// G must be bipartite graph!
// Returns matching size (edge count).
// match[v] = vert matched to v or -1
int matching(vector<Vi>& G, Vi& match) {
  vector<bool> seen;
  int n = 0, k = 1;
  match.assign(sz(G), -1);

  function<int(int)> dfs = [&](int i) {
    if (seen[i]) return 0;
    seen[i] = 1;
    each(e, G[i]) {
      if (match[e] < 0 || dfs(match[e])) {
        match[i] = e; match[e] = i;
        return 1;
      } // d41d
    } // d41d
    return 0;
  }; // d41d

  while (k) {
    seen.assign(sz(G), 0);
    k = 0;
    rep(i, 0, sz(G)) if (match[i] < 0)
      k += dfs(i);
    n += k;
  } // d41d
  return n;
} // d41d

// Convert maximum matching to vertex cover
// time: O(n+m)
Vi vertexCover(vector<Vi>& G, Vi& match) {
  Vi ret, col(sz(G)), seen(sz(G));

  function<void(int, int)> dfs =
      [&](int i, int c) {
    if (col[i]) return;
    col[i] = c+1;
    each(e, G[i]) dfs(e, !c);
  }; // d41d

  function<void(int)> aug = [&](int i) {
    if (seen[i] || col[i] != 1) return;
    seen[i] = 1;
    each(e, G[i]) seen[e] = 1, aug(match[e]);
  }; // d41d

  rep(i, 0, sz(G)) dfs(i, 0);
  rep(i, 0, sz(G)) if (match[i] < 0) aug(i);
  rep(i, 0, sz(G))
    if (seen[i] == col[i]-1) ret.pb(i);
  return ret;
} // d41d
```

math/berlekamp_massey.h d41d

```
constexpr int MOD = 1e9+7;

ll modInv(ll a, ll m) { // a^(-1) mod m
  if (a == 1) return 1;
  return ((a - modInv(m%a, a))*m + 1) / a;
} // d41d

// Find shortest linear recurrence that matches
// given starting terms of recurrence; O(n^2)
// Returns vector C such that for each i >= |C|
// A[i] = sum A[i-j-1]*C[j] for j = 0..|C|-1
// UNTESTED
vector<ll> massey(vector<ll>& A) {
  int n = sz(A), len = 0, k = 0;
  ll s = 1;
  vector<ll> B(n), C(n), tmp;
  B[0] = C[0] = 1;

  rep(i, 0, n) {
    ll d = 0;
    k++;
    rep(j, 0, len+1)
      d = (d + C[j] * A[i-j]) % MOD;

    if (d) {
      ll q = d * modInv(s, MOD) % MOD;
      tmp = C;

      rep(j, k, n)
        C[j] = (C[j] - q * B[j-k]) % MOD;

      if (len*2 <= i) {
        B.swap(tmp);
        len = i-len+1;
        s = d + (d < 0) * MOD;
        k = 0;
      } // d41d
    } // d41d
  } // d41d

  C.resize(len+1);
  C.erase(C.begin());
  each(x, C) x = (MOD - x) % MOD;
  return C;
} // d41d
```

math/bit_gauss.h d41d

```
constexpr int MAX_COLS = 2048;

// Solve system of linear equations over Z_2
// time: O(n^2*m/W), where W is word size
// - A - extended matrix, rows are equations,
//       columns are variables,
//       m-th column is equation result
//       (A[i][j] - i-th row and j-th column)
// - ans - output for variables values
// - m - variable count
// Returns 0 if no solutions found, 1 if one,
// 2 if more than 1 solution exist.
int bitGauss(vector<bitset<MAX_COLS>>& A,
             vector<bool>& ans, int m) {
  Vi col;
  ans.assign(m, 0);

  rep(i, 0, sz(A)) {
    int c = int(A[i]._Find_first());
    if (c >= m) {
      if (c == m) return 0;
      continue;
    } // d41d

    rep(k, i+1, sz(A)) if (A[k][c]) A[k]^=A[i];
    swap(A[i], A[sz(col)]);
    col.pb(c);
  } // d41d

  for (int i = sz(col); i--;) if (A[i][m]) {
    ans[col[i]] = 1;
    rep(k,0,i) if(A[k][col[i]]) A[k][m].flip();
  } // d41d

  return sz(col) < m ? 2 : 1;
} // d41d
```

math/bit_matrix.h d41d

```
using ull = uint64_t;

// Matrix over Z_2 (bits and xor)
// TODO: arithmetic operations
struct BitMatrix {
  vector<ull> M;
  int rows, cols, stride;

  // Create matrix with n rows and m columns
  BitMatrix(int n = 0, int m = 0) {
    rows = n; cols = m;
    stride = (m+63)/64;
    M.resize(n*stride);
  } // d41d

  // Get pointer to bit-packed data of i-th row
  ull* row(int i) { return &M[i*stride]; }

  // Get value in i-th row and j-th column
  bool operator()(int i, int j) {
    return (row(i)[j/64] >> (j%64)) & 1;
  } // d41d

  // Set value in i-th row and j-th column
  void set(int i, int j, bool val) {
    ull &w = row(i)[j/64], m = 1ull << (j%64);
    if (val) w |= m;
    else w &= ~m;
  } // d41d
}; // d41d
```

math/crt.h d41d

```
using Pll = pair<ll, ll>;

ll egcd(ll a, ll b, ll& x, ll& y) {
  if (!a) return x=0, y=1, b;
  ll d = egcd(b%a, a, y, x);
  x -= b/a*y;
  return d;
} // d41d

// Chinese Remainder Theoerem; time: O(lg lcm)
// Solves x = a.x (mod a.y), x = b.x (mod b.y)
// Returns pair (x mod lcm, lcm(a.y, b.y))
// or (-1, -1) if there's no solution.
// WARNING: a.x and b.x are assumed to be
// in [0;a.y) and [0;b.y) respectively.
// Works properly if lcm(a.y, b.y) < 2^63.
Pll crt(Pll a, Pll b) {
  if (a.y < b.y) swap(a, b);
  ll x, y, g = egcd(a.y, b.y, x, y);
  ll c = b.x-a.x, d = b.y/g, p = a.y*d;
  if (c % g) return {-1, -1};
  ll s = (a.x + c/g*x % d * a.y) % p;
  return {s < 0 ? s+p : s, p};
} // d41d
```

math/fft_complex.h d41d

```
using dbl = double;
using cmpl = complex<dbl>;

// Default std::complex multiplication is slow.
// You can use this to achieve small speedup.
cmpl operator*(cmpl a, cmpl b) {
  dbl ax = real(a), ay = imag(a);
  dbl bx = real(b), by = imag(b);
  return {ax*bx-ay*by, ax*by+ay*bx};
} // d41d

cmpl operator*=(cmpl& a,cmpl b) {return a=a*b;}

// Compute DFT over complex numbers; O(n lg n)
// Input size must be power of 2!
void fft(vector<cmpl>& a) {
  static vector<cmpl> w(2, 1);
  int n = sz(a);

  for (int k = sz(w); k < n; k *= 2) {
    w.resize(n);
    rep(i,0,k) w[k+i] = exp(cmpl(0, M_PI*i/k));
  } // d41d

  Vi rev(n);
  rep(i,0,n) rev[i] = (rev[i/2] | i%2*n) / 2;
  rep(i,0,n) if(i<rev[i]) swap(a[i],a[rev[i]]);
```

```
  for (int k = 1; k < n; k *= 2) {
    for (int i=0; i < n; i += k*2) rep(j,0,k) {
      auto d = a[i+j+k] * w[j+k];
      a[i+j+k] = a[i+j] - d;
      a[i+j] += d;
    } // d41d
  } // d41d
} // d41d

// Convolve complex-valued a and b,
// store result in a; time: O(n lg n), 3x FFT
void convolve(vector<cmpl>& a, vector<cmpl> b){
  int len = max(sz(a) + sz(b) - 1, 0);
  int n = 1 << (32 - __builtin_clz(len));
  a.resize(n); b.resize(n);
  fft(a); fft(b);
  rep(i, 0, n) a[i] *= b[i] / dbl(n);
  reverse(a.begin()+1, a.end());
  fft(a);
  a.resize(len);
} // d41d

// Convolve real-valued a and b, returns result
// time: O(n lg n), 2x FFT
// Rounding to integers is safe as long as
// (max_coeff^2)*n*log_2(n) < 9*10^14
// (in practice 10^16 or higher).
vector<dbl> convolve(vector<dbl>& a,
                     vector<dbl>& b) {
  int len = max(sz(a) + sz(b) - 1, 0);
  int n = 1 << (32 - __builtin_clz(len));

  vector<cmpl> in(n), out(n);
  rep(i, 0, sz(a)) in[i].real(a[i]);
  rep(i, 0, sz(b)) in[i].imag(b[i]);

  fft(in);
  each(x, in) x *= x;
  rep(i,0,n) out[i] = in[-i&(n-1)]-conj(in[i]);
  fft(out);

  vector<dbl> ret(len);
  rep(i, 0, len) ret[i] = imag(out[i]) / (n*4);
  return ret;
} // d41d

constexpr ll MOD = 1e9+7;

// High precision convolution of integer-valued
// a and b mod MOD; time: O(n lg n), 4x FFT
// Input is expected to be in range [0;MOD)!
// Rounding is safe if MOD*n*log_2(n) < 9*10^14
// (in practice 10^16 or higher).
vector<ll> convMod(vector<ll>& a,
                   vector<ll>& b) {
  vector<ll> ret(sz(a) + sz(b) - 1);
  int n = 1 << (32 - __builtin_clz(sz(ret)));
  ll cut = ll(sqrt(MOD))+1;

  vector<cmpl> c(n), d(n), g(n), f(n);

  rep(i, 0, sz(a))
    c[i] = {dbl(a[i]/cut), dbl(a[i]%cut)};
  rep(i, 0, sz(b))
    d[i] = {dbl(b[i]/cut), dbl(b[i]%cut)};
  fft(c); fft(d);

  rep(i, 0, n) {
    int j = -i & (n-1);
    f[j] = (c[i]+conj(c[j])) * d[i] / (n*2.0);
    g[j] =
      (c[i]-conj(c[j])) * d[i] / cmpl(0, n*2);
  } // d41d

  fft(f); fft(g);

  rep(i, 0, sz(ret)) {
    ll t = llround(real(f[i])) % MOD * cut;
    t += llround(imag(f[i]));
    t = (t + llround(real(g[i]))) % MOD * cut;
    t = (t + llround(imag(g[i]))) % MOD;
    ret[i] = (t < 0 ? t+MOD : t);
  } // d41d

  return ret;
} // d41d
```

math/fft_mod.h d41d

```
// Number Theoretic Tranform (NTT)
// For functions below you can choose 2 params:
// 1. M - prime modulus that MUST BE of form
//        a*2^k+1, computation is done in Z_M
// 2. R - generator of Z_M

// Modulus often seen on Codeforces:
// M = (119<<23)+1, R = 62; M is 998244353

// Parameters for ll computation with CRT:
// M = (479<<21)+1, R = 62; M is > 10^9
// M = (483<<21)+1, R = 62; M is > 10^9

ll modPow(ll a, ll e, ll m) {
  ll t = 1 % m;
  while (e) {
    if (e % 2) t = t*a % m;
    e /= 2; a = a*a % m;
  } // d41d
  return t;
} // d41d

// Compute DFT over Z_M with generator R.
// Input size must be power of 2; O(n lg n)
// Input is expected to be in range [0;MOD)!
// dit == true <=> inverse transform * 2^n
//                 (without normalization)
template<ll M, ll R, bool dit>
void ntt(vector<ll>& a) {
  static vector<ll> w(2, 1);
  int n = sz(a);

  for (int k = sz(w); k < n; k *= 2) {
    w.resize(n, 1);
    ll c = modPow(R, M/2/k, M);
    if (dit) c = modPow(c, M-2, M);
    rep(i, k+1, k*2) w[i] = w[i-1]*c % M;
  } // d41d

  for (int t = 1; t < n; t *= 2) {
    int k = (dit ? t : n/t/2);
    for (int i=0; i < n; i += k*2) rep(j,0,k) {
      ll &c = a[i+j], &d = a[i+j+k];
      ll e = w[j+k], f = d;
      d = (dit ? c - (f=f*e%M) : (c-f)*e % M);
      if (d < 0) d += M;
      if ((c += f) >= M) c -= M;
    } // d41d
  } // d41d
} // d41d

// Convolve a and b mod M (R is generator),
// store result in a; time: O(n lg n), 3x NTT
// Input is expected to be in range [0;MOD)!
template<ll M = (119<<23)+1, ll R = 62>
void convolve(vector<ll>& a, vector<ll> b) {
  int len = max(sz(a) + sz(b) - 1, 0);
  int n = 1 << (32 - __builtin_clz(len));
  ll t = modPow(n, M-2, M);
  a.resize(n); b.resize(n);
  ntt<M,R,0>(a); ntt<M,R,0>(b);
  rep(i, 0, n) a[i] = a[i]*b[i] % M * t % M;
  ntt<M,R,1>(a);
  a.resize(len);
} // d41d

ll egcd(ll a, ll b, ll& x, ll& y) {
  if (!a) return x=0, y=1, b;
  ll d = egcd(b%a, a, y, x);
  x -= b/a*y;
  return d;
} // d41d

// Convolve a and b with 64-bit output,
// store result in a; time: O(n lg n), 6x NTT
// Input is expected to be non-negative!
void convLong(vector<ll>& a, vector<ll> b) {
  const ll M1 = (479<<21)+1, M2 = (483<<21)+1;
  const ll MX = M1*M2, R = 62;

  vector<ll> c = a, d = b;
  each(k, a) k %= M1; each(k, b) k %= M1;
  each(k, c) k %= M2; each(k, d) k %= M2;

  convolve<M1, R>(a, b);
  convolve<M2, R>(c, d);

  ll x, y; egcd(M1, M2, x, y);

  rep(i, 0, sz(a)) {
    a[i] += (c[i]-a[i])*x % M2 * M1;
    if ((a[i] %= MX) < 0) a[i] += MX;
  } // d41d
} // d41d
```

math/fwht.h d41d

```
// Fast Walsh-Hadamard Transform; O(n lg n)
// Input must be power of 2!
// Uncommented version is for XOR.
// OR version is equivalent to sum-over-subsets
// (Zeta transform, inverse is Moebius).
// AND version is same as sum-over-supersets.
// TESTED ON RANDS
template<bool inv, class T>
void fwht(vector<T>& b) {
  for (int s = 1; s < sz(b); s *= 2) {
    for (int i = 0; i < sz(b); i += s*2) {
      rep(j, i, i+s) {
        auto &x = b[j], &y = b[j+s];
        tie(x, y) =
          mp(x+y, x-y);                  //XOR
        // inv ? mp(x-y, y) : mp(x+y, y); //AND
        // inv ? mp(x, y-x) : mp(x, x+y); //OR
      } // d41d
    } // d41d
  } // d41d

  // ONLY FOR XOR:
  if (inv) each(e, b) e /= sz(b);
} // d41d

// Compute convolution of a and b such that
// ans[i#j] += a[i]*b[j], where # is OR, AND
// or XOR, depending on FWHT version.
// Stores result in a; time: O(n lg n)
// Both arrays must be of same size = 2^n!
template<class T>
void bitConv(vector<T>& a, vector<T> b) {
  fwht<0>(a);
  fwht<0>(b);
  rep(i, 0, sz(a)) a[i] *= b[i];
  fwht<1>(a);
} // d41d
```

math/gauss.h d41d

```
// Solve system of linear equations; O(n^2*m)
// - A - extended matrix, rows are equations,
//       columns are variables,
//       m-th column is equation result
//       (A[i][j] - i-th row and j-th column)
// - ans - output for variables values
// - m - variable count
// Returns 0 if no solutions found, 1 if one,
// 2 if more than 1 solution exist.
int gauss(vector<vector<double>>& A,
          vector<double>& ans, int m) {
  Vi col;
  ans.assign(m, 0);

  rep(i, 0, sz(A)) {
    int c = 0;
    while (c <= m && !cmp(A[i][c], 0)) c++;
    // For Zp:
    //while (c <= m && !A[i][c].x) c++;

    if (c >= m) {
      if (c == m) return 0;
      continue;
    } // d41d

    rep(k, i+1, sz(A)) {
      auto mult = A[k][c] / A[i][c];
      rep(j, 0, m+1) A[k][j] -= A[i][j]*mult;
    } // d41d

    swap(A[i], A[sz(col)]);
    col.pb(c);
  } // d41d

  for (int i = sz(col); i--;) {
    ans[col[i]] = A[i][m] / A[i][col[i]];
    rep(k, 0, i)
      A[k][m] -= ans[col[i]] * A[k][col[i]];
  } // d41d
```

```
  return sz(col) < m ? 2 : 1;
} // d41d
```

math/matrix.h d41d

```
#include "modular.h"

// UNTESTED
using Row = vector<Zp>;
using Matrix = vector<Row>;

// Create n x n identity matrix
Matrix ident(int n) {
  Matrix ret(n, Row(n));
  rep(i, 0, n) ret[i][i] = 1;
  return ret;
} // d41d

// Add matrices
Matrix& operator+=(Matrix& l, const Matrix& r){
  rep(i, 0, sz(l)) rep(k, 0, sz(l[0]))
    l[i][k] += r[i][k];
  return l;
} // d41d
Matrix operator+(Matrix l, const Matrix& r) {
  return l += r;
} // d41d

// Subtract matrices
Matrix& operator-=(Matrix& l, const Matrix& r){
  rep(i, 0, sz(l)) rep(k, 0, sz(l[0]))
    l[i][k] -= r[i][k];
  return l;
} // d41d
Matrix operator-(Matrix l, const Matrix& r) {
  return l -= r;
} // d41d

// Multiply matrices
Matrix operator*(const Matrix& l,
                 const Matrix& r) {
  Matrix ret(sz(l), Row(sz(r[0])));
  rep(i, 0, sz(l)) rep(j, 0, sz(r[0]))
    rep(k, 0, sz(r))
      ret[i][j] += l[i][k] * r[k][j];
  return ret;
} // d41d
Matrix& operator*=(Matrix& l, const Matrix& r){
  return l = l*r;
} // d41d

// Square matrix power; time: O(n^3 * lg e)
Matrix pow(Matrix a, ll e) {
  Matrix t = ident(sz(a));
  while (e) {
    if (e % 2) t *= a;
    e /= 2; a *= a;
  } // d41d
  return t;
} // d41d

// Transpose matrix
Matrix transpose(const Matrix& m) {
  Matrix ret(sz(m[0]), Row(sz(m)));
  rep(i, 0, sz(m)) rep(j, 0, sz(m[0]))
    ret[j][i] = m[i][j];
  return ret;
} // d41d

// Transform matrix to echelon form
// and compute its determinant sign and rank.
int echelon(Matrix& A, int& sign) { // O(n^3)
  int rank = 0;
  sign = 1;
  rep(c, 0, sz(A[0])) {
    if (rank >= sz(A)) break;
    rep(i, rank+1, sz(A)) if (A[i][c].x) {
      swap(A[i], A[rank]);
      sign *= -1;
      break;
    } // d41d
    if (A[rank][c].x) {
      rep(i, rank+1, sz(A)) {
        auto mult = A[i][c] / A[rank][c];
        rep(j, 0, sz(A[0]))
          A[i][j] -= A[rank][j]*mult;
      } // d41d
      rank++;
    } // d41d
  } // d41d
  return rank;
} // d41d

// Compute matrix rank; time: O(n^3)
#define rank rank_
int rank(Matrix A) {
  int s; return echelon(A, s);
} // d41d

// Compute square matrix determinant; O(n^3)
Zp det(Matrix A) {
  int s; echelon(A, s);
  Zp ret = s;
  rep(i, 0, sz(A)) ret *= A[i][i];
  return ret;
} // d41d

// Invert square matrix if possible; O(n^3)
// Returns true if matrix is invertible.
bool invert(Matrix& A) {
  int s, n = sz(A);
  rep(i, 0, n) A[i].resize(n*2), A[i][n+i] = 1;
  echelon(A, s);
  for (int i = n; i--;) {
    if (!A[i][i].x) return 0;
    auto mult = A[i][i].inv();
    each(k, A[i]) k *= mult;
    rep(k, 0, i) rep(j, 0, n)
      A[k][n+j] -= A[i][n+j]*A[k][i];
  } // d41d
  each(r, A) r.erase(r.begin(), r.begin()+n);
  return 1;
} // d41d
```

math/miller_rabin.h d41d

```
#include "modular64.h"

// Miller-Rabin primality test
// time O(k*lg^2 n), where k = number of bases

// Deterministic for p <= 10^9
// constexpr ll BASES[] = {
//   336781006125, 9639812373923155
// }; // d41d

// Deterministic for p <= 2^64
constexpr ll BASES[] = {
  2,325,9375,28178,450775,9780504,1795265022
}; // d41d

bool isPrime(ll p) {
  if (p <= 2) return p == 2;
  if (p%2 == 0) return 0;

  ll d = p-1, t = 0;
  while (d%2 == 0) d /= 2, t++;

  each(a, BASES) if (a%p) {
    // ll a = rand() % (p-1) + 1;
    ll b = modPow(a%p, d, p);
    if (b == 1 || b == p-1) continue;
    rep(i, 1, t)
      if ((b = modMul(b, b, p)) == p-1) break;
    if (b != p-1) return 0;
  } // d41d

  return 1;
} // d41d
```

math/modinv_precompute.h d41d

```
constexpr ll MOD = 234567899;
vector<ll> modInv(MOD); // You can lower size

// Precompute modular inverses; time: O(MOD)
void initModInv() {
  modInv[1] = 1;
  rep(i, 2, sz(modInv)) modInv[i] =
    (MOD - (MOD/i) * modInv[MOD%i]) % MOD;
} // d41d
```

math/modular.h d41d

```
// Modulus often seen on Codeforces:
constexpr int MOD = 998244353;
// Some big prime: 15*(1<<27)+1 ~ 2*10^9

ll modInv(ll a, ll m) { // a^(-1) mod m
  if (a == 1) return 1;
  return ((a - modInv(m%a, a))*m + 1) / a;
} // d41d

ll modPow(ll a, ll e, ll m) { // a^e mod m
  ll t = 1 % m;
  while (e) {
    if (e % 2) t = t*a % m;
    e /= 2; a = a*a % m;
  } // d41d
  return t;
} // d41d

// Wrapper for modular arithmetic
struct Zp {
  ll x; // Contained value, in range [0;MOD-1]
  Zp() : x(0) {}
  Zp(ll a) : x(a%MOD) { if (x < 0) x += MOD; }

  #define OP(c,d) Zp& operator c##=(Zp r) { \
      x = x d; return *this; } \
    Zp operator c(Zp r) const { \
      Zp t = *this; return t c##= r; } // d41d

  OP(+, +r.x - MOD*(x+r.x >= MOD));
  OP(-, -r.x + MOD*(x-r.x < 0));
  OP(*, *r.x % MOD);
  OP(/, *r.inv().x % MOD);
  Zp operator-()
    const { Zp t; t.x = MOD-x; return t; }

  // For composite modulus use modInv, not pow
  Zp inv() const { return pow(MOD-2); }
  Zp pow(ll e) const{ return modPow(x,e,MOD); }
  void print() { cerr << x; } // For deb()
}; // d41d

// Extended Euclidean Algorithm
ll egcd(ll a, ll b, ll& x, ll& y) {
  if (!a) return x=0, y=1, b;
  ll d = egcd(b%a, a, y, x);
  x -= b/a*y;
  return d;
} // d41d
```

math/modular64.h d41d

```
// Modular arithmetic for modulus < 2^62

ll modAdd(ll x, ll y, ll m) {
  x += y;
  return x < m ? x : x-m;
} // d41d

ll modSub(ll x, ll y, ll m) {
  x -= y;
  return x >= 0 ? x : x+m;
} // d41d

// About 4x slower than normal modulo
ll modMul(ll a, ll b, ll m) {
  ll c = ll((long double)a * b / m);
  ll r = (a*b - c*m) % m;
  return r < 0 ? r+m : r;
} // d41d

ll modPow(ll x, ll e, ll m) {
  ll t = 1;
  while (e) {
    if (e & 1) t = modMul(t, x, m);
    e >>= 1;
    x = modMul(x, x, m);
  } // d41d
  return t;
} // d41d
```

math/modular_generator.h d41d

```
#include "modular.h" // modPow

// Get unique prime factors of n; O(sqrt n)
vector<ll> factorize(ll n) {
  vector<ll> fac;
  for (ll i = 2; i*i <= n; i++) {
    if (n%i == 0) {
      while (n%i == 0) n /= i;
      fac.pb(i);
    } // d41d
  } // d41d
```

```
  if (n > 1) fac.pb(n);
  return fac;
} // d41d

// Find smallest primitive root mod n;
// time: O(sqrt(n) + g*log^2 n)
// Returns -1 if generator doesn't exist.
// For n <= 10^7 smallest generator is <= 115.
// You can use faster factorization algorithm
// to get rid of sqrt(n). [UNTESTED]
ll generator(ll n) {
  if (n <= 1 || (n > 4 && n%4 == 0)) return -1;

  vector<ll> fac = factorize(n);
  if (sz(fac) > (fac[0] == 2)+1) return -1;

  ll phi = n;
  each(p, fac) phi = phi / p * (p-1);
  fac = factorize(phi);

  for (ll g = 1;; g++) if (__gcd(g, n) == 1) {
    each(f, fac) if (modPow(g, phi/f, n) == 1)
      goto nxt;
    return g;
    nxt:;
  } // d41d
} // d41d
```

math/modular_log.h d41d

```
#include "modular.h" // modInv

// Baby-step giant-step algorithm; O(sqrt(p))
// Finds smallest x such that a^x = b (mod p)
// or returns -1 if there's no solution.
ll dlog(ll a, ll b, ll p) {
  int m = int(min(ll(sqrt(p))+2, p-1));
  unordered_map<ll, int> small;
  ll t = 1;

  rep(i, 0, m) {
    int& k = small[t];
    if (!k) k = i+1;
    t = t*a % p;
  } // d41d

  t = modInv(t, p);

  rep(i, 0, m) {
    int j = small[b];
    if (j) return i*ll(m) + j - 1;
    b = b*t % p;
  } // d41d

  return -1;
} // d41d
```

math/modular_sqrt.h d41d

```
#include "modular.h" // modPow

// Tonelli-Shanks algorithm for modular sqrt
// modulo prime; O(lg^2 p), O(lg p) for most p
// Returns -1 if root doesn't exists or else
// returns square root x (the other one is -x).
// UNTESTED
ll modSqrt(ll a, ll p) {
  a %= p;
  if (a < 0) a += p;
  if (a <= 1) return a;
  if (modPow(a, p/2, p) != 1) return -1;
  if (p%4 == 3) return modPow(a, p/4+1, p);

  ll s = p-1, n = 2;
  int r = 0, j;
  while (s%2 == 0) s /= 2, r++;
  while (modPow(n, p/2, p) != p-1) n++;

  ll x = modPow(a, (s+1)/2, p);
  ll b = modPow(a, s, p), g = modPow(n, s, p);

  for (;; r = j) {
    ll t = b;
    for (j = 0; j < r && t != 1; j++)
      t = t*t % p;
    if (!j) return x;
    ll gs = modPow(g, 1LL << (r-j-1), p);
    g = gs*gs % p;
    x = x*gs % p;
    b = b*g % p;
  } // d41d
} // d41d
```

math/montgomery.h d41d

```
#include "modular.h" // modInv

// Montgomery modular multiplication
// MOD < MG_MULT, gcd(MG_MULT, MOD) must be 1
// Don't use if modulo is constexpr; UNTESTED

constexpr ll MG_SHIFT = 32;
constexpr ll MG_MULT  = 1LL << MG_SHIFT;
constexpr ll MG_MASK  = MG_MULT - 1;
const ll MG_INV = MG_MULT-modInv(MOD, MG_MULT);

// Convert to Montgomery form
ll MG(ll x) { return (x*MG_MULT) % MOD; }

// Montgomery reduction
// redc(mg * mg) = Montgomery-form product
ll redc(ll x) {
  ll q = (x * MG_INV) & MG_MASK;
  x = (x + q*MOD) >> MG_SHIFT;
  return (x >= MOD ? x-MOD : x);
} // d41d
```

math/nimber.h d41d

```
// Nimbers are defined as sizes of Nim heaps.
// Operations on nimbers are defined as:
// a+b = mex({a'+b : a' < a} u {a+b' : b' < b})
// ab  = mex({a'b+ab'+a'b' : a' < a, b' < b})
// Nimbers smaller than M = 2^2^k form a field.
// Addition is equivalent to xor, meanwhile
// multiplication can be evaluated
// in O(lg^2 M) after precomputing.

using ull = uint64_t;
ull nbuf[64][64]; // Nim-products for 2^i * 2^j

// Multiply nimbers; time: O(lg^2 M)
// WARNING: Call initNimMul() before using.
ull nimMul(ull a, ull b) {
  ull ret = 0;
  for (ull s = a; s; s &= (s-1))
    for (ull t = b; t; t &= (t-1))
      ret ^= nbuf[__builtin_ctzll(s)]
                 [__builtin_ctzll(t)];
  return ret;
} // d41d

// Initialize nim-products lookup table
void initNimMul() {
  rep(i, 0, 64)
    nbuf[i][0] = nbuf[0][i] = 1ull << i;
  rep(b, 1, 64) rep(a, 1, b+1) {
    int i = 1 << (63 - __builtin_clzll(a));
    int j = 1 << (63 - __builtin_clzll(b));
    ull t = nbuf[a-i][b-j];
    if (i < j)
      t = nimMul(t, 1ull << i) << j;
    else
      t = nimMul(t, 1ull << (i-1)) ^ (t << i);
    nbuf[a][b] = nbuf[b][a] = t;
  } // d41d
} // d41d

// Compute a^e under nim arithmetic; O(lg^3 M)
// WARNING: Call initNimMul() before using.
ull nimPow(ull a, ull e) {
  ull t = 1;
  while (e) {
    if (e % 2) t = nimMul(t, a);
    e /= 2; a = nimMul(a, a);
  } // d41d
  return t;
} // d41d

// Compute inverse of a in 2^64 nim-field;
// time: O(lg^3 M)
// WARNING: Call initNimMul() before using.
ull nimInv(ull a) {
  return nimPow(a, ull(-2));
} // d41d

// If you need to multiply many nimbers by
// the same value you can use this to speedup.
// WARNING: Call initNimMul() before using.
struct NimMult {
  ull M[64] = {0};

  // Initialize lookup; time: O(lg^2 M)
  NimMult(ull a) {
    for (ull t=a; t; t &= (t-1)) rep(i, 0, 64)
      M[i] ^= nbuf[__builtin_ctzll(t)][i];
  } // d41d

  // Multiply by b; time: O(lg M)
  ull operator()(ull b) {
    ull ret = 0;
    for (ull t = b; t; t &= (t-1))
      ret ^= M[__builtin_ctzll(t)];
    return ret;
  } // d41d
}; // d41d
```

math/phi_large.h d41d

```
#include "pollard_rho.h"

// Compute Euler's totient of large numbers
// time: O(n^(1/4)) <- factorization
ll phi(ll n) {
  each(p, factorize(n)) n = n / p.x * (p.x-1);
  return n;
} // d41d
```

math/phi_precompute.h d41d

```
Vi phi(1e7+1);

// Precompute Euler's totients; time: O(n lg n)
void calcPhi() {
  iota(all(phi), 0);
  rep(i, 2, sz(phi)) if (phi[i] == i)
    for (int j = i; j < sz(phi); j += i)
      phi[j] = phi[j] / i * (i-1);
} // d41d
```

math/phi_prefix_sum.h d41d

```
#include "phi_precompute.h"

vector<ll> phiSum; // [k] = sum from 0 to k-1

// Precompute Euler's totient prefix sums
// for small values; time: O(n lg n)
void calcPhiSum() {
  calcPhi();
  phiSum.resize(sz(phi)+1);
  rep(i, 0, sz(phi))
    phiSum[i+1] = phiSum[i] + phi[i];
} // d41d

// Get prefix sum of phi(0) + ... + phi(n-1).
// WARNING: Call calcPhiSum first!
// For n > 4*10^9, answer will overflow.
// If you wish to get answer mod M use
// commented lines.
ll getPhiSum(ll n) { // time: O(n^(2/3))
  static unordered_map<ll, ll> big;
  if (n < sz(phiSum)) return phiSum[n];
  if (big.count(--n)) return big[n];

  ll ret = n*(n+1)/2;
  // ll ret = (n%2 ? n%M * ((n+1)/2 % M)
  //               : n/2%M * (n%M+1)) % M;

  for (ll s, i = 2; i <= n; i = s+1) {
    s = n / (n/i);
    ret -= (s-i+1) * getPhiSum(n/i+1);
    // ret -= (s-i+1)%M * getPhiSum(n/i+1) % M;
  } // d41d

  // ret = (ret%M + M) % M;
  return big[n] = ret;
} // d41d
```

math/pi_large.h d41d

```
constexpr int MAX_P = 1e7;
vector<ll> pis, prl;

// Precompute prime counting function
// for small values; time: O(n lg lg n)
void initPi() {
  pis.assign(MAX_P+1, 1);
```

```
  pis[0] = pis[1] = 0;

  for (int i = 2; i*i <= MAX_P; i++)
    if (pis[i])
      for (int j = i*i; j <= MAX_P; j += i)
        pis[j] = 0;

  rep(i, 1, sz(pis)) {
    if (pis[i]) prl.pb(i);
    pis[i] += pis[i-1];
  } // d41d
} // d41d

ll partial(ll x, ll a) {
  static vector<unordered_map<ll, ll>> big;
  big.resize(sz(prl));
  if (!a) return (x+1) / 2;
  if (big[a].count(x)) return big[a][x];
  ll ret = partial(x, a-1)
    - partial(x / prl[a], a-1);
  return big[a][x] = ret;
} // d41d

// Count number of primes <= x;
// time: O(n^(2/3) * log(n)^(1/3))
// Set MAX_P to be > sqrt(x) and call initPi
// before using!
ll pi(ll x) {
  static unordered_map<ll, ll> big;
  if (x < sz(pis)) return pis[x];
  if (big.count(x)) return big[x];

  ll a = 0;
  while (prl[a]*prl[a]*prl[a]*prl[a] < x) a++;

  ll ret = 0, b = --a;

  while (++b < sz(prl) && prl[b]*prl[b] < x) {
    ll w = x / prl[b];
    ret -= pi(w);
    for (ll j = b; prl[j]*prl[j] <= w; j++)
      ret -= pi(w / prl[j]) - j;
  } // d41d

  ret += partial(x, a) + (b+a-1)*(b-a)/2;
  return big[x] = ret;
} // d41d
```

math/pi_large_precomp.h d41d

```
#include "sieve.h"

// Count primes in given interval
// using precomputed table.
// Set MAX_P to sqrt(MAX_N) and run sieve()!
// Precomputed table will contain N_BUCKETS
// elements - check source size limit.

constexpr ll MAX_N = 1e11+1;
constexpr ll N_BUCKETS = 10000;
constexpr ll BUCKET_SIZE = (MAX_N/N_BUCKETS)+1;
constexpr ll precomputed[] = {/* ... */};

ll sieveRange(ll from, ll to) {
  bitset<BUCKET_SIZE> elems;
  from = max(from, 2LL);
  to = max(from, to);
  each(p, primesList) {
    ll c = max((from+p-1) / p, 2LL);
    for (ll i = c*p; i < to; i += p)
      elems.set(i-from);
  } // d41d
  return to-from-elems.count();
} // d41d

// Run once on local computer to precompute
// table. Takes about 10 minutes for n = 1e11.
// Sanity check (for default params):
// 664579, 606028, 587253, 575795, ...
void localPrecompute() {
  for (ll i = 0; i < MAX_N; i += BUCKET_SIZE) {
    ll to = min(i+BUCKET_SIZE, MAX_N);
    cout << sieveRange(i, to) << ',' << flush;
  } // d41d
  cout << endl;
} // d41d

// Count primes in [from;to) using table.
// O(N_BUCKETS + BUCKET_SIZE*lg lg n + sqrt(n))
ll countPrimes(ll from, ll to) {
  ll bFrom = from/BUCKET_SIZE+1,
     bTo = to/BUCKET_SIZE;
  if (bFrom > bTo) return sieveRange(from, to);
  ll ret = accumulate(precomputed+bFrom,
                      precomputed+bTo, 0);
  ret += sieveRange(from, bFrom*BUCKET_SIZE);
  ret += sieveRange(bTo*BUCKET_SIZE, to);
  return ret;
} // d41d
```

math/pollard_rho.h d41d

```
#include "modular64.h"
#include "miller_rabin.h"

using Factor = pair<ll, int>;

void rho(vector<ll>& out, ll n) {
  if (n <= 1) return;
  if (isPrime(n)) out.pb(n);
  else if (n%2 == 0) rho(out,2), rho(out,n/2);
  else for (ll a = 2;; a++) {
    ll x = 2, y = 2, d = 1;
    while (d == 1) {
      x = modAdd(modMul(x, x, n), a, n);
      y = modAdd(modMul(y, y, n), a, n);
      y = modAdd(modMul(y, y, n), a, n);
      d = __gcd(abs(x-y), n);
    } // d41d
    if (d != n) return rho(out,d),rho(out,n/d);
  } // d41d
} // d41d

// Pollard's rho factorization algorithm
// Las Vegas version; time: n^(1/4)
// Returns pairs (prime, power), sorted
vector<Factor> factorize(ll n) {
  vector<Factor> ret;
  vector<ll> raw;
  rho(raw, n);
  sort(all(raw));
  each(f, raw) {
    if (ret.empty() || ret.back().x != f)
      ret.pb({ f, 1 });
    else
      ret.back().y++;
  } // d41d
  return ret;
} // d41d
```

math/polynomial.h d41d

```
#include "modular.h"
#include "fft_mod.h"

// UNTESTED
using Poly = vector<Zp>;

// Cut off trailing zeroes; time: O(n)
void norm(Poly& P) {
  while (!P.empty() && !P.back().x)
    P.pop_back();
} // d41d

// Evaluate polynomial at x; time: O(n)
Zp eval(const Poly& P, Zp x) {
  Zp n = 0, y = 1;
  each(a, P) n += a*y, y *= x;
  return n;
} // d41d

// Add polynomial; time: O(n)
Poly& operator+=(Poly& l, const Poly& r) {
  l.resize(max(sz(l), sz(r)));
  rep(i, 0, sz(r)) l[i] += r[i];
  norm(l);
  return l;
} // d41d
Poly operator+(Poly l, const Poly& r) {
  return l += r;
} // d41d

// Subtract polynomial; time: O(n)
Poly& operator-=(Poly& l, const Poly& r) {
  l.resize(max(sz(l), sz(r)));
  rep(i, 0, sz(r)) l[i] -= r[i];
  norm(l);
  return l;
} // d41d
Poly operator-(Poly l, const Poly& r) {
  return l -= r;
} // d41d

// Multiply by polynomial; time: O(n lg n)
Poly& operator*=(Poly& l, const Poly& r) {
  if (min(sz(l), sz(r)) < 50) {
    // Naive multiplication
    Poly P(sz(l)+sz(r));
    rep(i, 0, sz(l)) rep(j, 0, sz(r))
      P[i+j] += l[i]*r[j];
    l.swap(P);
  } else {
    // FFT multiplication
    vector<ll> a, b;
    each(k, l) a.pb(k.x);
    each(k, r) b.pb(k.x);
    // Choose appropriate convolution method,
    // see fft_mod.h and fft_complex.h
    convolve<MOD, 62>(a, b);
    l.assign(all(a));
  } // d41d
  norm(l);
  return l;
} // d41d
Poly operator*(Poly l, const Poly& r) {
  return l *= r;
} // d41d

// Derivate polynomial; time: O(n)
Poly derivate(Poly P) {
  if (!P.empty()) {
    rep(i, 1, sz(P)) P[i-1] = P[i]*i;
    P.pop_back();
  } // d41d
  return P;
} // d41d

// Integrate polynomial; time: O(n)
Poly integrate(Poly P) {
  if (!P.empty()) {
    P.pb(0);
    for (int i = sz(P); --i;) P[i] = P[i-1]/i;
    P[0] = 0;
  } // d41d
  return P;
} // d41d

// Compute inverse series mod x^n; O(n lg n)
Poly invert(const Poly& P, int n) {
  assert(!P.empty() && P[0].x != 0);
  Poly tmp, ret = {P[0].inv()};

  for (int i = 1; i < n; i *= 2) {
    tmp.clear();
    rep(j, 0, min(i*2, sz(P))) tmp.pb(-P[j]);
    tmp *= ret;
    tmp[0] += 2;
    ret *= tmp;
    ret.resize(i*2);
  } // d41d

  ret.resize(n);
  return ret;
} // d41d

// Floor division by polynomial; O(n lg n)
Poly operator/(Poly l, Poly r) {
  norm(l); norm(r);
  int d = sz(l)-sz(r)+1;
  if (d <= 0) return {};
  reverse(all(l));
  reverse(all(r));
  l.resize(d);
  l *= invert(r, d);
  l.resize(d);
  reverse(all(l));
  return l;
} // d41d
Poly& operator/=(Poly& l, const Poly& r) {
  return l = l/r;
} // d41d

// Compute modulo by polynomial; O(n lg n)
Poly operator%(const Poly& l, const Poly& r) {
  return l - r*(l/r);
} // d41d
Poly& operator%=(Poly& l, const Poly& r) {
```

```
  return l -= r*(l/r);
} // d41d

// Evaluate polynomial P in given points;
// time: O(n lg^2 n)
Poly eval(const Poly& P, Poly points) {
  int len = 1;
  while (len < sz(points)) len *= 2;

  vector<Poly> tree(len*2, {1});
  rep(i, 0, sz(points))
    tree[len+i] = {-points[i], 1};

  for (int i = len; --i;)
    tree[i] = tree[i*2] * tree[i*2+1];

  tree[0] = P;
  rep(i, 1, len*2)
    tree[i] = tree[i/2] % tree[i];

  rep(i, 0, sz(points)) {
    auto& vec = tree[len+i];
    points[i] = vec.empty() ? 0 : vec[0];
  } // d41d
  return points;
} // d41d

// Given n points (x, f(x)) compute n-1-degree
// polynomial f that passes through them;
// time: O(n lg^2 n)
// For O(n^2) version see polynomial_interp.h.
Poly interpolate(const vector<pair<Zp,Zp>>& P){
  int len = 1;
  while (len < sz(P)) len *= 2;

  vector<Poly> mult(len*2, {1}), tree(len*2);
  rep(i, 0, sz(P))
    mult[len+i] = {-P[i].x, 1};

  for (int i = len; --i;)
    mult[i] = mult[i*2] * mult[i*2+1];

  tree[0] = derivate(mult[1]);
  rep(i, 1, len*2)
    tree[i] = tree[i/2] % mult[i];

  rep(i, 0, sz(P))
    tree[len+i][0] = P[i].y / tree[len+i][0];

  for (int i = len; --i;)
    tree[i] = tree[i*2]*mult[i*2+1]
            + mult[i*2]*tree[i*2+1];
  return tree[1];
} // d41d
```

math/polynomial_interp.h d41d

```
// Interpolate set of points (i, vec[i])
// and return it evaluated at x; time: O(n)
template<class T>
T polyExtend(vector<T>& vec, T x) {
  int n = sz(vec);
  vector<T> fac(n, 1), suf(n, 1);

  rep(i, 1, n) fac[i] = fac[i-1] * i;
  for (int i=n; --i;) suf[i-1] = suf[i]*(x-i);

  T pref = 1, ret = 0;
  rep(i, 0, n) {
    T d = fac[i] * fac[n-i-1] * ((n-i)%2*2-1);
    ret += vec[i] * suf[i] * pref / d;
    pref *= x-i;
  } // d41d
  return ret;
} // d41d

// Given n points (x, f(x)) compute n-1-degree
// polynomial f that passes through them;
// time: O(n^2)
// For O(n lg^2 n) version see polynomial.h
template<class T>
vector<T> polyInterp(vector<pair<T, T>> P) {
  int n = sz(P);
  vector<T> ret(n), tmp(n);
  T last = 0;
  tmp[0] = 1;

  rep(k, 0, n-1) rep(i, k+1, n)
    P[i].y = (P[i].y-P[k].y) / (P[i].x-P[k].x);

  rep(k, 0, n) rep(i, 0, n) {
    ret[i] += P[k].y * tmp[i];
    swap(last, tmp[i]);
    tmp[i] -= last * P[k].x;
  } // d41d
  return ret;
} // d41d
```

math/sieve.h d41d

```
constexpr int MAX_P = 1e6;
bitset<MAX_P+1> primes;
Vi primesList;

// Erathostenes sieve; time: O(n lg lg n)
void sieve() {
  primes.set();
  primes.reset(0);
  primes.reset(1);

  for (int i = 2; i*i <= MAX_P; i++)
    if (primes[i])
      for (int j = i*i; j <= MAX_P; j += i)
        primes.reset(j);

  rep(i, 0, MAX_P+1) if (primes[i])
    primesList.pb(i);
} // d41d
```

math/sieve_factors.h d41d

```
constexpr int MAX_P = 1e6;
Vi factor(MAX_P+1);

// Erathostenes sieve with saving smallest
// factor for each number; time: O(n lg lg n)
void sieve() {
  for (int i = 2; i*i <= MAX_P; i++)
    if (!factor[i])
      for (int j = i*i; j <= MAX_P; j += i)
        if (!factor[j])
          factor[j] = i;

  rep(i,0,MAX_P+1) if (!factor[i]) factor[i]=i;
} // d41d

// Factorize n <= MAX_P; time: O(lg n)
// Returns pairs (prime, power), sorted
vector<Pii> factorize(ll n) {
  vector<Pii> ret;
  while (n > 1) {
    int f = factor[n];
    if (ret.empty() || ret.back().x != f)
      ret.pb({ f, 1 });
    else
      ret.back().y++;
    n /= f;
  } // d41d
  return ret;
} // d41d
```

math/sieve_segmented.h d41d

```
constexpr int MAX_P = 1e9;
bitset<MAX_P/2+1> primes; // Only odd numbers

// Cache-friendly Erathostenes sieve
// ~1.5s on Intel Core i5 for MAX_P = 10^9
// Memory usage: MAX_P/16 bytes
void sieve() {
  constexpr int SEG_SIZE = 1<<18;
  int pSqrt = int(sqrt(MAX_P)+0.5);
  vector<Pii> dels;
  primes.set();
  primes.reset(0);

  for (int i = 3; i <= pSqrt; i += 2) {
    if (primes[i/2]) {
      int j;
      for (j = i*i; j <= pSqrt; j += i*2)
        primes.reset(j/2);
      dels.pb({ i, j/2 });
    } // d41d
  } // d41d

  for (int seg = pSqrt/2;
       seg <= sz(primes); seg += SEG_SIZE) {
    int lim = min(seg+SEG_SIZE, sz(primes));
    each(d, dels) for (;d.y < lim; d.y += d.x)
      primes.reset(d.y);
  } // d41d
} // d41d

bool isPrime(int x) {
  return x == 2 || (x%2 && primes[x/2]);
} // d41d
```

structures/bitset_plus.h d41d

```
// Undocumented std::bitset features:
// - _Find_first() - returns first bit = 1 or N
// - _Find_next(i) - returns first bit = 1
//                   after i-th bit
//                   or N if not found

// Bitwise operations for vector<bool>
// UNTESTED

#define OP(x) vector<bool>& operator x##=(     \
    vector<bool>& l, const vector<bool>& r) { \
  assert(sz(l) == sz(r));                     \
  auto a = l.begin(); auto b = r.begin();     \
  while (a<l.end()) *a._M_p++ x##= *b._M_p++; \
  return l; } // d41d
OP(&)OP(|)OP(^)
```

structures/fenwick_tree.h d41d

```
// Fenwick tree (BIT tree); space: O(n)
// Default version: prefix sums
struct Fenwick {
  using T = int;
  static const T ID = 0;
  T f(T a, T b)  { return a+b; }

  vector<T> s;
  Fenwick(int n = 0) : s(n, ID) {}

  // A[i] = f(A[i], v); time: O(lg n)
  void modify(int i, T v) {
    for (; i < sz(s); i |= i+1) s[i]=f(s[i],v);
  } // d41d

  // Get f(A[0], .., A[i-1]); time: O(lg n)
  T query(int i) {
    T v = ID;
    for (; i > 0; i &= i-1) v = f(v, s[i-1]);
    return v;
  } // d41d

  // Find smallest i such that
  // f(A[0],...,A[i-1]) >= val; time: O(lg n)
  // Prefixes must have non-descreasing values.
  int lowerBound(T val) {
    if (val <= ID) return 0;
    int i = -1, mask = 1;
    while (mask <= sz(s)) mask *= 2;
    T off = ID;

    while (mask /= 2) {
      int k = mask+i;
      if (k < sz(s)) {
        T x = f(off, s[k]);
        if (val > x) i=k, off=x;
      } // d41d
    } // d41d
    return i+2;
  } // d41d
}; // d41d
```

structures/fenwick_tree_2d.h d41d

```
// Fenwick tree 2D (BIT tree 2D); space: O(n*m)
// Default version: prefix sums 2D
// Change s to hashmap for O(q lg^2 n) memory
struct Fenwick2D {
  using T = int;
  static constexpr T ID = 0;
  T f(T a, T b) { return a+b; }

  vector<T> s;
  int w, h;

  Fenwick2D(int n = 0, int m = 0)
    : s(n*m, ID), w(n), h(m) {}

  // A[i,j] = f(A[i,j], v); time: O(lg^2 n)
  void modify(int i, int j, T v) {
```

```
    for (; i < w; i |= i+1)
      for (int k = j; k < h; k |= k+1)
        s[i*h+k] = f(s[i*h+k], v);
  } // d41d

  // Query prefix; time: O(lg^2 n)
  T query(int i, int j) {
    T v = ID;
    for (; i>0; i&=i-1)
      for (int k = j; k > 0; k &= k-1)
        v = f(v, s[i*h+k-h-1]);
    return v;
  } // d41d
}; // d41d
```

structures/find_union.h d41d

```
// Disjoint set data structure; space: O(n)
// Operations work in amortized O(alfa(n))
struct FAU {
  Vi G;
  FAU(int n = 0) : G(n, -1) {}

  // Get size of set containing i
  int size(int i) { return -G[find(i)]; }

  // Find representative of set containing i
  int find(int i) {
    return G[i] < 0 ? i : G[i] = find(G[i]);
  } // d41d

  // Union sets containing i and j
  bool join(int i, int j) {
    i = find(i); j = find(j);
    if (i == j) return 0;
    if (G[i] > G[j]) swap(i, j);
    G[i] += G[j]; G[j] = i;
    return 1;
  } // d41d
}; // d41d
```

structures/hull_offline.h d41d

```
constexpr ll INF = 2e18;
// constexpr double INF = 1e30;
// constexpr double EPS = 1e-9;

// MAX of linear functions; space: O(n)
// Use if you add lines in increasing `a` order
// Default uncommented version is for int64
struct Hull {
  using T = ll; // Or change to double

  struct Line {
    T a, b, end;
    T intersect(const Line& r) const {
      // Version for double:
      //if (r.a-a < EPS) return b>r.b?INF:-INF;
      //return (b-r.b) / (r.a-a);
      if (a==r.a) return b > r.b ? INF : -INF;
      ll u = b-r.b, d = r.a-a;
      return u/d + ((u^d) >= 0 || !(u%d));
    } // d41d
  }; // d41d

  vector<Line> S;
  Hull() { S.pb({ 0, -INF, INF }); }

  // Insert f(x) = ax+b; time: amortized O(1)
  void push(T a, T b) {
    Line l{a, b, INF};
    while (true) {
      T e = S.back().end=S.back().intersect(l);
      if (sz(S) < 2 || S[sz(S)-2].end < e)
        break;
      S.pop_back();
    } // d41d
    S.pb(l);
  } // d41d

  // Query max(f(x) for each f): time: O(lg n)
  T query(T x) {
    auto t = *upper_bound(all(S), x,
      [](int l, const Line& r) {
        return l < r.end;
      }); // d41d
    return t.a*x + t.b;
  } // d41d
}; // d41d
```

structures/hull_online.h d41d

```
constexpr ll INF = 2e18;

// MAX of linear functions online; space: O(n)
struct Hull {
  static bool modeQ; // Toggles operator< mode

  struct Line {
    mutable ll a, b, end;

    ll intersect(const Line& r) const {
      if (a==r.a) return b > r.b ? INF : -INF;
      ll u = b-r.b, d = r.a-a;
      return u/d + ((u^d) >= 0 || !(u%d));
    } // d41d

    bool operator<(const Line& r) const {
      return modeQ ? end < r.end : a < r.a;
    } // d41d
  }; // d41d

  multiset<Line> S;
  Hull() { S.insert({ 0, -INF, INF }); }

  // Updates segment end
  bool update(multiset<Line>::iterator it) {
    auto cur = it++; cur->end = INF;
    if (it == S.end()) return false;
    cur->end = cur->intersect(*it);
    return cur->end >= it->end;
  } // d41d

  // Insert f(x) = ax+b; time: O(lg n)
  void insert(ll a, ll b) {
    auto it = S.insert({ a, b, INF });
    while (update(it)) it = --S.erase(++it);
    rep(i, 0, 2)
      while (it != S.begin() && update(--it))
        update(it = --S.erase(++it));
  } // d41d

  // Query max(f(x) for each f): time: O(lg n)
  ll query(ll x) {
    modeQ = 1;
    auto l = *S.upper_bound({ 0, 0, x });
    modeQ = 0;
    return l.a*x + l.b;
  } // d41d
}; // d41d

bool Hull::modeQ = false;
```

structures/max_queue.h d41d

```
// Queue with max query on contained elements
struct MaxQueue {
  using T = int;
  deque<T> Q, M;

  // Add v to the back; time: amortized O(1)
  void push(T v) {
    while (!M.empty() && M.back() < v)
      M.pop_back();
    M.pb(v); Q.pb(v);
  } // d41d

  // Pop from the front; time: O(1)
  void pop() {
    if (M.front() == Q.front()) M.pop_front();
    Q.pop_front();
  } // d41d

  // Get max element value; time: O(1)
  T max() const { return M.front(); }
}; // d41d
```

structures/pairing_heap.h d41d

```
// Pairing heap implementation; space O(n)
// Elements are stored in vector for faster
// allocation. It's MINIMUM queue.
// Allows to merge heaps in O(1)
template<class T, class Cmp = less<T>>
struct PHeap {
  struct Node {
    T val;
    int child{-1}, next{-1}, prev{-1};

    Node(T x = T()) : val(x) {}
  }; // d41d

  using Vnode = vector<Node>;
  Vnode& M;
  int root{-1};

  int unlink(int& i) {
    if (i >= 0) M[i].prev = -1;
    int x = i; i = -1;
    return x;
  } // d41d

  void link(int host, int& i, int val) {
    if (i >= 0) M[i].prev = -1;
    i = val;
    if (i >= 0) M[i].prev = host;
  } // d41d

  int merge(int l, int r) {
    if (l < 0) return r;
    if (r < 0) return l;
    if (Cmp()(M[l].val, M[r].val)) swap(l, r);

    link(l, M[l].next, unlink(M[r].child));
    link(r, M[r].child, l);
    return r;
  } // d41d

  int mergePairs(int v) {
    if (v < 0 || M[v].next < 0) return v;
    int v2 = unlink(M[v].next);
    int v3 = unlink(M[v2].next);
    return merge(merge(v, v2), mergePairs(v3));
  } // d41d

  // ---

  // Initialize heap with given node storage
  // Just declare 1 Vnode and pass it to heaps
  PHeap(Vnode& mem) : M(mem) {}

  // Add given key to heap, returns index; O(1)
  int push(const T& x) {
    int index = sz(M);
    M.emplace_back(x);
    root = merge(root, index);
    return index;
  } // d41d

  // Change key of i to smaller value; O(1)
  void decrease(int i, T val) {
    assert(!Cmp()(M[i].val, val));
    M[i].val = val;

    int prev = M[i].prev;
    if (prev < 0) return;

    auto& p = M[prev];
    link(prev, (p.child == i ? p.child
        : p.next), unlink(M[i].next));

    root = merge(root, i);
  } // d41d

  bool empty() { return root < 0; }
  const T& top() { return M[root].val; }

  // Merge with other heap. Must use same vec.
  void merge(PHeap& r) { // time: O(1)
    assert(&M == &r.M);
    root = merge(root, r.root); r.root = -1;
  } // d41d

  // Remove min element; time: O(lg n)
  void pop() {
    root = mergePairs(unlink(M[root].child));
  } // d41d
}; // d41d
```

structures/rmq.h d41d

```
// Range Minimum Query; space: O(n lg n)
struct RMQ {
  using T = int;
  static constexpr T ID = INT_MAX;
  T f(T a, T b)  { return min(a, b); }

  vector<vector<T>> s;
```

```
  // Initialize RMQ structure; time: O(n lg n)
  RMQ(const vector<T>& vec = {}) {
    s = {vec};
    for (int h = 1; h <= sz(vec); h *= 2) {
      s.emplace_back();
      auto& prev = s[sz(s)-2];
      rep(i, 0, sz(vec)-h*2+1)
        s.back().pb(f(prev[i], prev[i+h]));
    } // d41d
  } // d41d

  // Query f(s[b], ... ,s[e-1]); time: O(1)
  T query(int b, int e) {
    if (b >= e) return ID;
    int k = 31 - __builtin_clz(e-b);
    return f(s[k][b], s[k][e - (1<<k)]);
  } // d41d
}; // d41d
```

structures/segtree_config.h d41d

```
// Segment tree configurations to be used
// in segtree_general and segtree_persistent.
// See comments in TREE_PLUS version
// to understand how to create custom ones.
// Capabilities notation: (update; query)

#if TREE_PLUS // (+; sum, max, max count)
  // time: O(lg n) [UNTESTED]
  using T = int;  // Data type for update
                  // operations (lazy tag)
  const T ID = 0; // Neutral value for
                  // updates and lazy tags

  // This structure keeps aggregated data
  struct Agg {
    // Aggregated data: sum, max, max count
    // Default values should be neutral
    // values, i.e. "aggregate over empty set"
    T sum{0}, vMax{INT_MIN}, nMax{0};

    // Initialize as leaf (single value)
    void leaf() { sum = vMax = 0; nMax = 1; }

    // Combine data with aggregated data
    // from node to the right
    void merge(const Agg& r) {
      if (vMax < r.vMax) nMax = r.nMax;
      else if (vMax == r.vMax) nMax += r.nMax;
      vMax = max(vMax, r.vMax);
      sum += r.sum;
    } // d41d

    // Apply update provided in `x`:
    // - update aggregated data
    // - update lazy tag `lazy`
    // - `size` is amount of elements
    // - return 0 if update should branch
    //   (to be used in "segement tree beats")
    // - if you change value of `x` changed
    //   value will be passed to next node
    //   to the right during updates
    bool apply(T& lazy, T& x, int size) {
      lazy += x;
      sum += x*size;
      vMax += x;
      return 1;
    } // d41d
  }; // d41d
#elif TREE_MAX // (max; max, max count)
  // time: O(lg n) [UNTESTED]
  using T = int;
  const T ID = INT_MIN;

  struct Agg {
    // Aggregated data: max value, max count
    T vMax{INT_MIN}, nMax{0};
    void leaf() { vMax = 0; nMax = 1; }

    void merge(const Agg& r) {
      if (vMax < r.vMax) nMax = r.nMax;
      else if (vMax == r.vMax) nMax += r.nMax;
      vMax = max(vMax, r.vMax);
    } // d41d

    bool apply(T& lazy, T& x, int size) {
      if (vMax <= x) nMax = size;
      lazy = max(lazy, x);
      vMax = max(vMax, x);
      return 1;
    } // d41d
  }; // d41d
#elif TREE_SET // (=; sum, max, max count)
  // time: O(lg n) [UNTESTED]
  // Set ID to some unused value.
  using T = int;
  const T ID = INT_MIN;

  struct Agg {
    // Aggregated data: sum, max, max count
    T sum{0}, vMax{INT_MIN}, nMax{0};
    void leaf() { sum = vMax = 0; nMax = 1; }

    void merge(const Agg& r) {
      if (vMax < r.vMax) nMax = r.nMax;
      else if (vMax == r.vMax) nMax += r.nMax;
      vMax = max(vMax, r.vMax);
      sum += r.sum;
    } // d41d

    bool apply(T& lazy, T& x, int size) {
      lazy = x;
      sum = x*size;
      vMax = x;
      nMax = size;
      return 1;
    } // d41d
  }; // d41d
#elif TREE_BEATS // (+, min; sum, max)
  // time: amortized O(lg n) if not using +
  //       amortized O(lg^2 n) if using +
  // Lazy tag is pair (add, min).
  // To add x: run update with {x, INT_MAX},
  // to min x: run update with {0, x}.
  // When both parts are provided addition
  // is applied first, then minimum.
  using T = Pii;
  const T ID = {0, INT_MAX};

  struct Agg {
    // Aggregated data: max value, max count,
    //                  second max value, sum
    int vMax{INT_MIN}, nMax{0}, max2{INT_MIN};
    int sum{0};
    void leaf() { sum = vMax = 0; nMax = 1; }

    void merge(const Agg& r) {
      if (r.vMax > vMax) {
        max2 = vMax;
        vMax = r.vMax;
        nMax = r.nMax;
      } else if (r.vMax == vMax) {
        nMax += r.nMax;
      } else if (r.vMax > max2) {
        max2 = r.vMax;
      } // d41d
      max2 = max(max2, r.max2);
      sum += r.sum;
    } // d41d

    bool apply(T& lazy, T& x, int size) {
      if (max2 != INT_MIN && max2+x.x >= x.y)
        return 0;

      lazy.x += x.x;
      sum += x.x*size;
      vMax += x.x;
      if (max2 != INT_MIN) max2 += x.x;

      if (x.y < vMax) {
        sum -= (vMax-x.y) * nMax;
        vMax = x.y;
      } // d41d
      lazy.y = vMax;
      return 1;
    } // d41d
  }; // d41d
#endif
```

structures/segtree_general.h d41d

```
// Highly configurable statically allocated
// (interval; interval) segment tree;
// space: O(n) [UNTESTED]
struct SegTree {
  // Choose/write configuration
  #include "segtree_config.h"

  // Root node is 1, left is i*2, right i*2+1
  vector<Agg> agg; // Aggregated data for nodes
  vector<T> lazy;  // Lazy tags for nodes
  int len{1};      // Number of leaves

  // Initialize tree for n elements; time: O(n)
  SegTree(int n = 0) {
    while (len < n) len *= 2;
    agg.resize(len*2);
    lazy.resize(len*2, ID);
    rep(i, 0, n) agg[len+i].leaf();
    for (int i = len; --i;)
      (agg[i] = agg[i*2]).merge(agg[i*2+1]);
  } // d41d

  void push(int i, int s) {
    if (lazy[i] != ID) {
      agg[i*2].apply(lazy[i*2], lazy[i], s/2);
      agg[i*2+1].apply(lazy[i*2+1],
                       lazy[i], s/2);
      lazy[i] = ID;
    } // d41d
  } // d41d

  // Modify interval [vb;ve) with val; O(lg n)
  T update(int vb, int ve, T val, int i = 1,
           int b = 0, int e = -1) {
    if (e < 0) e = len;
    if (vb >= e || b >= ve) return val;

    if (b >= vb && e <= ve &&
        agg[i].apply(lazy[i], val, e-b))
      return val;

    int m = (b+e) / 2;
    push(i, e-b);
    val = update(vb, ve, val, i*2, b, m);
    val = update(vb, ve, val, i*2+1, m, e);
    (agg[i] = agg[i*2]).merge(agg[i*2+1]);
    return val;
  } // d41d

  // Query interval [vb;ve); time: O(lg n)
  Agg query(int vb, int ve, int i = 1,
            int b = 0, int e = -1) {
    if (e < 0) e = len;
    if (vb >= e || b >= ve) return {};
    if (b >= vb && e <= ve) return agg[i];

    int m = (b+e) / 2;
    push(i, e-b);
    Agg t = query(vb, ve, i*2, b, m);
    t.merge(query(vb, ve, i*2+1, m, e));
    return t;
  } // d41d
}; // d41d
```

structures/segtree_persist.h d41d

```
// Highly configurable (interval; interval)
// persistent segment tree;
// space: O(queries lg n) [UNTESTED]
// First tree version number is 0.
struct SegTree {
  // Choose/write configuration
  #include "segtree_config.h"

  vector<Agg> agg;  // Aggregated data for nodes
  vector<T> lazy;   // Lazy tags for nodes
  vector<bool> cow; // Copy children on push?
  Vi L, R;          // Children links
  int len{1};       // Number of leaves

  // Initialize tree for n elements; O(lg n)
  SegTree(int n = 0) {
    int k = 1;
    while (len < n) len *= 2, k++;

    agg.resize(k);
    lazy.resize(k, ID);
    cow.resize(k, 1);
    L.resize(k);
    R.resize(k);
    agg[--k].leaf();

    while (k--) {
      (agg[k] = agg[k+1]).merge(agg[k+1]);
      L[k] = R[k] = k+1;
```

```
  } // d41d
} // d41d

// New version from version `i`; time: O(1)
// First version number is 0.
int fork(int i) {
  L.pb(L[i]); R.pb(R[i]); cow.pb(cow[i] = 1);
  agg.pb(agg[i]); lazy.pb(lazy[i]);
  return sz(L)-1;
} // d41d

void push(int i, int s, bool w) {
  bool has = (lazy[i] != ID);
  if ((has || w) && cow[i]) {
    int a = fork(L[i]), b = fork(R[i]);
    L[i] = a; R[i] = b; cow[i] = 0;
  } // d41d
  if (has) {
    agg[L[i]].apply(lazy[L[i]],lazy[i],s/2);
    agg[R[i]].apply(lazy[R[i]],lazy[i],s/2);
    lazy[i] = ID;
  } // d41d
} // d41d

// Modify interval [vb;ve) with val
// in tree version `i`; time: O(lg n)
T update(int i, int vb, int ve, T val,
         int b = 0, int e = -1) {
  if (e < 0) e = len;
  if (vb >= e || b >= ve) return val;

  if (b >= vb && e <= ve &&
      agg[i].apply(lazy[i], val, e-b))
    return val;

  int m = (b+e) / 2;
  push(i, e-b, 1);
  val = update(L[i], vb, ve, val, b, m);
  val = update(R[i], vb, ve, val, m, e);
  (agg[i] = agg[L[i]]).merge(agg[R[i]]);
  return val;
} // d41d

// Query interval [vb;ve)
// in tree version `i`; time: O(lg n)
Agg query(int i, int vb, int ve,
          int b = 0, int e = -1) {
  if (e < 0) e = len;
  if (vb >= e || b >= ve) return {};
  if (b >= vb && e <= ve) return agg[i];

  int m = (b+e) / 2;
  push(i, e-b, 0);
  Agg t = query(L[i], vb, ve, b, m);
  t.merge(query(R[i], vb, ve, m, e));
  return t;
} // d41d
}; // d41d
```

structures/segtree_point.h d41d

```
// Segment tree (point, interval)
// Configure by modifying:
// - T - stored data type
// - ID - neutral element for query operation
// - merge(a, b) - combine results
struct SegTree {
  using T = int;
  static constexpr T ID = INT_MIN;
  static T merge(T a, T b) { return max(a,b); }

  vector<T> V;
  int len;

  // Initialize tree for n elements; time: O(n)
  SegTree(int n = 0, T def = ID) {
    for (len = 1; len < n; len *= 2);
    V.resize(len*2, ID);
    rep(i, 0, n) V[len+i] = def;
    for (int i = len; --i;)
      V[i] = merge(V[i*2], V[i*2+1]);
  } // d41d

  // Set element `i` to `val`; time: O(lg n)
  void set(int i, T val) {
    V[i += len] = val;
    while (i /= 2)
      V[i] = merge(V[i*2], V[i*2+1]);
  } // d41d

  // Query interval [b;e); time: O(lg n)
  T query(int b, int e) {
    b += len; e += len-1;
    if (b > e)  return ID;
    if (b == e) return V[b];
    T x = merge(V[b], V[e]);

    while (b/2 < e/2) {
      if (~b&1) x = merge(x, V[b^1]);
      if (e&1)  x = merge(x, V[e^1]);
      b /= 2; e /= 2;
    } // d41d
    return x;
  } // d41d
}; // d41d

constexpr SegTree::T SegTree::ID;
```

structures/treap.h d41d

```
// "Set" of implicit keyed treaps; space: O(n)
// Nodes are keyed by their indices in array
// of all nodes. Treap key is key of its root.
// "Node x" means "node with key x".
// "Treap x" means "treap with key x".
// Key -1 is "null".
// Put any additional data in Node struct.
struct Treap {
  struct Node {
    // E[0] = left child, E[1] = right child
    // weight = node random weight (for treap)
    // size = subtree size, par = parent node
    int E[2] = {-1, -1}, weight{rand()};
    int size{1}, par{-1};
    bool flip{0}; // Is interval reversed?
  }; // d41d

  vector<Node> G; // Array of all nodes

  // Initialize structure for n nodes
  // with keys 0, ..., n-1; time: O(n)
  // Each node is separate treap,
  // use join() to make sequence.
  Treap(int n = 0) : G(n) {}

  // Create new treap (a single node),
  // returns its key; time: O(1)
  int make() {
    G.emplace_back(); return sz(G)-1;
  } // d41d

  // Get size of node x subtree. x can be -1.
  int size(int x) { // time: O(1)
    return (x >= 0 ? G[x].size : 0);
  } // d41d

  // Propagate down data (flip flag etc).
  // x can be -1; time: O(1)
  void push(int x) {
    if (x >= 0 && G[x].flip) {
      G[x].flip = 0;
      swap(G[x].E[0], G[x].E[1]);
      each(e, G[x].E) if (e>=0) G[e].flip ^= 1;
    } // + any other lazy operations
  } // d41d

  // Update aggregates of node x.
  // x can be -1; time: O(1)
  void update(int x) {
    if (x >= 0) {
      int& s = G[x].size = 1;
      G[x].par = -1;
      each(e, G[x].E) if (e >= 0) {
        s += G[e].size;
        G[e].par = x;
      } // d41d
    } // + any other aggregates
  } // d41d

  // Split treap x into treaps l and r
  // such that l contains first i elements
  // and r the remaining ones.
  // x, l, r can be -1; time: ~O(lg n)
  void split(int x, int& l, int& r, int i) {
    push(x); l = r = -1;
    if (x < 0) return;
    int key = size(G[x].E[0]);
    if (i <= key) {
      split(G[x].E[0], l, G[x].E[0], i);
      r = x;
    } else {
      split(G[x].E[1], G[x].E[1], r, i-key-1);
      l = x;
    } // d41d
    update(x);
  } // d41d

  // Join treaps l and r into one treap
  // such that elements of l are before
  // elements of r. Returns new treap.
  // l, r and returned value can be -1.
  int join(int l, int r) { // time: ~O(lg n)
    push(l); push(r);
    if (l < 0 || r < 0) return max(l, r);

    if (G[l].weight < G[r].weight) {
      G[l].E[1] = join(G[l].E[1], r);
      update(l);
      return l;
    } // d41d
    G[r].E[0] = join(l, G[r].E[0]);
    update(r);
    return r;
  } // d41d

  // Find i-th node in treap x.
  // Returns its key or -1 if not found.
  // x can be -1; time: ~O(lg n)
  int find(int x, int i) {
    while (x >= 0) {
      push(x);
      int key = size(G[x].E[0]);
      if (key == i) return x;
      x = G[x].E[key < i];
      if (key < i) i -= key+1;
    } // d41d
    return -1;
  } // d41d

  // Get key of treap containing node x
  // (key of treap root). x can be -1.
  int root(int x) { // time: ~O(lg n)
    while (G[x].par >= 0) x = G[x].par;
    return x;
  } // d41d

  // Get position of node x in its treap.
  // x is assumed to NOT be -1; time: ~O(lg n)
  int index(int x) {
    int p, i = size(G[x].E[G[x].flip]);
    while ((p = G[x].par) >= 0) {
      if (G[p].E[1] == x) i+=size(G[p].E[0])+1;
      if (G[p].flip) i = G[p].size-i-1;
      x = p;
    } // d41d
    return i;
  } // d41d

  // Reverse interval [l;r) in treap x.
  // Returns new key of treap; time: ~O(lg n)
  int reverse(int x, int l, int r) {
    int a, b, c;
    split(x, b, c, r);
    split(b, a, b, l);
    if (b >= 0) G[b].flip ^= 1;
    return join(join(a, b), c);
  } // d41d
}; // d41d
```

structures/wavelet_tree.h d41d

```
// Wavelet tree ("merge-sort tree over values")
// Each node represent interval of values.
// seq[1]     = original sequence
// seq[i]     = subsequence with values
//              represented by i-th node
// left[i][j] = how many values in seq[0:j)
//              go to left subtree
struct WaveletTree {
  vector<Vi> seq, left;
  int len;

  // Build wavelet tree for sequence `elems`;
  // time and space: O((n+maxVal) log maxVal)
  // Values are expected to be in [0;maxVal).
  WaveletTree(const Vi& elems, int maxVal) {
```

```
    for (len = 1; len < maxVal; len *= 2);
    seq.resize(len*2);
    left.resize(len*2);
    seq[1] = elems;
    build(1, 0, len);
  } // d41d

  void build(int i, int b, int e) {
    if (i >= len) return;
    int m = (b+e) / 2;
    left[i].pb(0);
    each(x, seq[i]) {
      left[i].pb(left[i].back() + (x < m));
      seq[i*2 + (x >= m)].pb(x);
    } // d41d
    build(i*2, b, m);
    build(i*2+1, m, e);
  } // d41d

  // Find k-th smallest element in [begin;end)
  // [begin;end); time: O(log maxVal)
  int kth(int begin, int end, int k, int i=1) {
    if (i >= len) return seq[i][0];
    int x = left[i][begin], y = left[i][end];
    if (k < y-x) return kth(x, y, k, i*2);
    return kth(begin-x, end-y, k-y+x, i*2+1);
  } // d41d

  // Count number of elements >= vb and < ve
  // in [begin;end); time: O(log maxVal)
  int count(int begin, int end, int vb, int ve,
            int i = 1, int b = 0, int e = -1) {
    if (e < 0) e = len;
    if (b >= ve || vb >= e) return 0;
    if (b >= vb && e <= ve) return end-begin;
    int m = (b+e) / 2;
    int x = left[i][begin], y = left[i][end];
    return count(x, y, vb, ve, i*2, b, m) +
      count(begin-x, end-y, vb, ve, i*2+1,m,e);
  } // d41d
}; // d41d
```

structures/ext/hash_table.h d41d

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
// gp_hash_table<K, V> = faster unordered_set

// Anti-anti-hash
const size_t HXOR = mt19937_64(time(0))();
template<class T> struct SafeHash {
  size_t operator()(const T& x) const {
    return hash<T>()(x ^ T(HXOR));
  } // d41d
}; // d41d
```

structures/ext/rope.h d41d

```
#include <ext/rope>
using namespace __gnu_cxx;
// rope<T> = implicit cartesian tree
```

structures/ext/tree.h d41d

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template<class T, class Cmp = less<T>>
using ordered_set = tree<
  T, null_type, Cmp, rb_tree_tag,
  tree_order_statistics_node_update
>;

// Standard set functions and:
// t.order_of_key(key) - index of first >= key
// t.find_by_order(i) - find i-th element
// t1.join(t2) - assuming t1<>t2 merge t2 to t1
```

structures/ext/trie.h d41d

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/trie_policy.hpp>
using namespace __gnu_pbds;

using pref_trie = trie<
  string, null_type,
  trie_string_access_traits<>, pat_trie_tag,
  trie_prefix_search_node_update
>;
```

text/aho_corasick.h d41d

```
constexpr char AMIN = 'a'; // Smallest letter
constexpr int ALPHA = 26;  // Alphabet size

// Aho-Corasick algorithm for linear-time
// multiple pattern matching.
// Add patterns using add(), then call build().
struct Aho {
  vector<array<int, ALPHA>> nxt{1};
  Vi suf = {-1}, accLink = {-1};
  vector<Vi> accept{1};

  // Add string with given ID to structure
  // Returns index of accepting node
  int add(const string& str, int id) {
    int i = 0;
    each(c, str) {
      if (!nxt[i][c-AMIN]) {
        nxt[i][c-AMIN] = sz(nxt);
        nxt.pb({}); suf.pb(-1);
        accLink.pb(1); accept.pb({});
      } // d41d
      i = nxt[i][c-AMIN];
    } // d41d
    accept[i].pb(id);
    return i;
  } // d41d

  // Build automata; time: O(V*ALPHA)
  void build() {
    queue<int> que;
    each(e, nxt[0]) if (e) {
      suf[e] = 0; que.push(e);
    } // d41d
    while (!que.empty()) {
      int i = que.front(), s = suf[i], j = 0;
      que.pop();
      each(e, nxt[i]) {
        if (e) que.push(e);
        (e ? suf[e] : e) = nxt[s][j++];
      } // d41d
      accLink[i] = (accept[s].empty() ?
          accLink[s] : s);
    } // d41d
  } // d41d

  // Append `c` to state `i`
  int next(int i, char c) {
    return nxt[i][c-AMIN];
  } // d41d

  // Call `f` for each pattern accepted
  // when in state `i` with its ID as argument.
  // Return true from `f` to terminate early.
  // Calls are in descreasing length order.
  template<class F> void accepted(int i, F f) {
    while (i != -1) {
      each(a, accept[i]) if (f(a)) return;
      i = accLink[i];
    } // d41d
  } // d41d
}; // d41d
```

text/kmp.h d41d

```
// Computes prefsuf array; time: O(n)
// ps[i] = max prefsuf of [0;i); ps[0] := -1
template<class T> Vi kmp(const T& str) {
  Vi ps; ps.pb(-1);
  each(x, str) {
    int k = ps.back();
    while (k >= 0 && str[k] != x) k = ps[k];
    ps.pb(k+1);
  } // d41d
  return ps;
} // d41d

// Finds occurences of pat in vec; time: O(n)
// Returns starting indices of matches.
template<class T>
Vi match(const T& str, T pat) {
  int n = sz(pat);
  pat.pb(-1); // SET TO SOME UNUSED CHARACTER
  pat.insert(pat.end(), all(str));
  Vi ret, ps = kmp(pat);
  rep(i, 0, sz(ps)) {
    if (ps[i] == n) ret.pb(i-2*n-1);
  } // d41d
  return ret;
} // d41d
```

text/kmr.h d41d

```
// KMR algorithm for O(1) lexicographical
// comparison of substrings.
struct KMR {
  vector<Vi> ids;

  KMR() {}

  // Initialize structure; time: O(n lg^2 n)
  // You can change str type to Vi freely.
  KMR(const string& str) {
    ids.clear();
    ids.pb(Vi(all(str)));

    for (int h = 1; h <= sz(str); h *= 2) {
      vector<pair<Pii, int>> tmp;

      rep(j, 0, sz(str)) {
        int a = ids.back()[j], b = -1;
        if (j+h < sz(str)) b = ids.back()[j+h];
        tmp.pb({ {a, b}, j });
      } // d41d

      sort(all(tmp));
      ids.emplace_back(sz(tmp));

      int n = 0;
      rep(j, 0, sz(tmp)) {
        if (j > 0 && tmp[j-1].x != tmp[j].x)
          n++;
        ids.back()[tmp[j].y] = n;
      } // d41d
    } // d41d
  } // d41d

  // Get representative of [begin;end); O(1)
  Pii get(int begin, int end) {
    if (begin >= end) return {0, 0};
    int k = 31 - __builtin_clz(end-begin);
    return {ids[k][begin], ids[k][end-(1<<k)]};
  } // d41d

  // Compare [b1;e1) with [b2;e2); O(1)
  // Returns -1 if <, 0 if ==, 1 if >
  int cmp(int b1, int e1, int b2, int e2) {
    int l1 = e1-b1, l2 = e2-b2;
    int l = min(l1, l2);
    Pii x = get(b1, b1+l), y = get(b2, b2+l);

    if (x == y) return (l1 > l2) - (l1 < l2);
    return (x > y) - (x < y);
  } // d41d

  // Compute suffix array of string; O(n)
  Vi sufArray() {
    Vi sufs(sz(ids.back()));
    rep(i, 0, sz(ids.back()))
      sufs[ids.back()[i]] = i;
    return sufs;
  } // d41d
}; // d41d
```

text/lcp.h d41d

```
// Compute Longest Common Prefix array for
// given string and it's suffix array; O(n)
// In order to compute suffix array use kmr.h
// or suffix_array_linear.h
template<class T>
Vi lcpArray(const T& str, const Vi& sufs) {
  int n = sz(str), k = 0;
  Vi pos(n), lcp(n-1);
  rep(i, 0, n) pos[sufs[i]] = i;
  rep(i, 0, n) {
    if (pos[i] < n-1) {
      int j = sufs[pos[i]+1];
      while (i+k < n && j+k < n &&
          str[i+k] == str[j+k]) k++;
      lcp[pos[i]] = k;
    } // d41d
    if (k > 0) k--;
  } // d41d
  return lcp;
} // d41d
```

text/lyndon_factorization.h d41d

```
// Compute Lyndon factorization for s; O(n)
// Word is simple iff it's stricly smaller
// than any of it's nontrivial suffixes.
// Lyndon factorization is division of string
// into non-increasing simple words.
// It is unique.
vector<string> duval(const string& s) {
  int n = sz(s), i = 0;
  vector<string> ret;
  while (i < n) {
    int j = i+1, k = i;
    while (j < n && s[k] <= s[j])
      k = (s[k] < s[j] ? i : k+1), j++;
    while (i <= k)
      ret.pb(s.substr(i, j-k)), i += j-k;
  } // d41d
  return ret;
} // d41d
```

text/main_lorentz.h d41d

```
#include "z_function.h"

struct Sqr {
  int begin, end, len;
}; // d41d

// Main-Lorentz algorithm for finding
// all squares in given word; time: O(n lg n)
// Results are in compressed form:
// (b, e, l) means that for each b <= i < e
// there is square at position i of size 2l.
// Each square is present in only one interval.
vector<Sqr> lorentz(const string& s) {
  int n = sz(s);
  if (n <= 1) return {};
  auto a = s.substr(0, n/2), b = s.substr(n/2);

  auto ans = lorentz(a);
  each(p, lorentz(b))
    ans.pb({p.begin+n/2, p.end+n/2, p.len});

  string ra(a.rbegin(), a.rend());
  string rb(b.rbegin(), b.rend());

  rep(j, 0, 2) {
    Vi z1 = prefPref(ra), z2 = prefPref(b+a);
    z1.pb(0); z2.pb(0);

    rep(c, 0, sz(a)) {
      int l = sz(a)-c;
      int x = c - min(l-1, z1[l]);
      int y = c - max(l-z2[sz(b)+c], j);
      if (x > y) continue;

      if (j)
        ans.pb({n-y-l*2, n-x-l*2+1, l});
      else
        ans.pb({x, y+1, l});
    } // d41d

    a.swap(rb);
    b.swap(ra);
  } // d41d
```

```
  return ans;
} // d41d
```

text/manacher.h d41d

```
// Manacher algorithm; time: O(n)
// Finds largest radiuses for palindromes:
// r[2*i] = for center at i (single letter = 1)
// r[2*i+1] = for center between i and i+1
template<class T> Vi manacher(const T& str) {
  int n = sz(str)*2, c = 0, e = 1;
  Vi r(n, 1);
  auto get = [&](int i) { return i%2 ? 0 :
    (i >= 0 && i < n ? str[i/2] : i); }; // d41d

  rep(i, 0, n) {
    if (i < e) r[i] = min(r[c*2-i], e-i);
    while (get(i-r[i]) == get(i+r[i])) r[i]++;
    if (i+r[i] > e) c = i, e = i+r[i]-1;
  } // d41d

  rep(i, 0, n) r[i] /= 2;
  return r;
} // d41d
```

text/min_rotation.h d41d

```
// Find lexicographically smallest
// rotation of s; time: O(n)
// Returns index where shifted word starts.
// You can use std::rotate to get the word:
// rotate(s.begin(), s.begin()+minRotation(s),
//        s.end());
int minRotation(string s) {
  int a = 0, n = sz(s); s += s;
  rep(b, 0, n) rep(i, 0, n) {
    if (a+i == b || s[a+i] < s[b+i]) {
      b += max(0, i-1); break;
    } // d41d
    if (s[a+i] > s[b+i]) {
      a = b; break;
    } // d41d
  } // d41d
  return a;
} // d41d
```

text/palindromic_tree.h d41d

```
constexpr int ALPHA = 26; // Set alphabet size

// Tree of all palindromes in string,
// constructed online by appending letters.
// space: O(n*ALPHA); time: O(n)
// Code marked with [EXT] is extension for
// calculating minimal palindrome partition
// in O(n lg n). Can also be modified for
// similar dynamic programmings.
struct PalTree {
  Vi txt; // Text for which tree is built

  // Node 0 = empty palindrome (root of even)
  // Node 1 = "-1" palindrome (root of odd)
  Vi len{0, -1}; // Lengths of palindromes
  Vi link{1, 0}; // Suffix palindrome links
  // Edges to next palindromes
  vector<array<int, ALPHA>> to{ {}, {} };
  int last{0}; // Current node (max suffix pal)

  Vi diff{0, 0};   // len[i]-len[link[i]] [EXT]
  Vi slink{0, 0};  // Serial links        [EXT]
  Vi series{0, 0}; // Series DP answer    [EXT]
  Vi ans{0};       // DP answer for prefix[EXT]

  int ext(int i) {
    while (len[i]+2 > sz(txt) ||
        txt[sz(txt)-len[i]-2] != txt.back())
      i = link[i];
    return i;
  } // d41d

  // Append letter from [0;ALPHA); time: O(1)
  // (or O(lg n) if [EXT] is enabled)
  void add(int x) {
    txt.pb(x);
    last = ext(last);

    if (!to[last][x]) {
      len.pb(len[last]+2);
      link.pb(to[ext(link[last])][x]);
      to[last][x] = sz(to);
      to.emplace_back();

      // [EXT]
      diff.pb(len.back() - len[link.back()]);
      slink.pb(diff.back() == diff[link.back()]
        ? slink[link.back()] : link.back());
      series.pb(0);
      // [/EXT]
    } // d41d
    last = to[last][x];

    // [EXT]
    ans.pb(INT_MAX);
    for (int i=last; len[i] > 0; i=slink[i]) {
      series[i] = ans[sz(ans) - len[slink[i]]
                  - diff[i] - 1];
      if (diff[i] == diff[link[i]])
        series[i] = min(series[i],
                        series[link[i]]);
      // If you want only even palindromes
      // set ans only for sz(txt)%2 == 0
      ans.back() = min(ans.back(),series[i]+1);
    } // d41d
    // [/EXT]
  } // d41d
}; // d41d
```

text/suffix_array_linear.h d41d

```
#include "../util/radix_sort.h"

// KS algorithm for suffix array; time: O(n)
// Input values are assumed to be in [1;k]
Vi sufArray(Vi str, int k) {
  int n = sz(str);
  Vi suf(n);
  str.resize(n+15);

  if (n < 15) {
    iota(all(suf), 0);
    rep(j, 0, n) countSort(suf,
      [&](int i) { return str[i+n-j-1]; }, k);
    return suf;
  } // d41d

  // Compute triples codes
  Vi tmp, code(n+2);
  rep(i, 0, n) if (i % 3) tmp.pb(i);

  rep(j, 0, 3) countSort(tmp,
    [&](int i) { return str[i-j+2]; }, k);

  int mc = 0, j = -1;

  each(i, tmp) {
    code[i] = mc += (j == -1 ||
        str[i] != str[j] ||
        str[i+1] != str[j+1] ||
        str[i+2] != str[j+2]);
    j = i;
  } // d41d

  // Compute suffix array of 2/3
  tmp.clear();
  for (int i=1; i < n; i += 3) tmp.pb(code[i]);
  tmp.pb(0);
  for (int i=2; i < n; i += 3) tmp.pb(code[i]);
  tmp = sufArray(move(tmp), mc);

  // Compute partial suffix arrays
  Vi third;
  int th = (n+4) / 3;
  if (n%3 == 1) third.pb(n-1);

  rep(i, 1, sz(tmp)) {
    int e = tmp[i];
    tmp[i-1] = (e < th ? e*3+1 : (e-th)*3+2);
    code[tmp[i-1]] = i;
    if (e < th) third.pb(e*3);
  } // d41d

  tmp.pop_back();
  countSort(third,
    [&](int i) { return str[i]; }, k);

  // Merge suffix arrays
  merge(all(third), all(tmp), suf.begin(),
    [&](int l, int r) {
      while (l%3 == 0 || r%3 == 0) {
        if (str[l] != str[r])
          return str[l] < str[r];
        l++; r++;
      } // d41d
      return code[l] < code[r];
    }); // d41d

  return suf;
} // d41d

// KS algorithm for suffix array; time: O(n)
Vi sufArray(const string& str) {
  return sufArray(Vi(all(str)), 255);
} // d41d
```

text/suffix_automaton.h d41d

```
constexpr char AMIN = 'a'; // Smallest letter
constexpr int ALPHA = 26; // Set alphabet size

// Suffix automaton - minimal DFA that
// recognizes all suffixes of given string
```

```cpp
// (and encodes all substrings);
// space: O(n*ALPHA); time: O(n)
// Paths from root are equivalent to substrings
// Extensions:
// - [OCC] - count occurences of substrings
// - [PATHS] - count paths from node
struct SufDFA {
  // State v represents endpos-equivalence
  // class that contains words of all lengths
  // between link[len[v]]+1 and len[v].
  // len[v] = longest word of equivalence class
  // link[v] = link to state of longest suffix
  //           in other equivalence class
  // to[v][c] = automaton edge c from v
  Vi len{0}, link{-1};
  vector<array<int, ALPHA>> to{ {} };
  int last{0}; // Current node (whole word)

  vector<Vi> insSufs; // [OCC] Suffix-link tree
  Vi cnt{0};          // [OCC] Occurence count
  vector<ll> paths;   // [PATHS] Out-path count

  SufDFA() {}

  // Build suffix automaton for given string
  // and compute extended stuff; time: O(n)
  SufDFA(const string& s) {
    each(c, s) add(c);
    finish();
  } // d41d

  // Append letter to the back
  void add(char c) {
    int v = last, x = c-AMIN;
    last = sz(len);
    len.pb(len[v]+1);
    link.pb(0);
    to.pb({});
    cnt.pb(1); // [OCC]

    while (v != -1 && !to[v][x]) {
      to[v][x] = last;
      v = link[v];
    } // d41d

    if (v != -1) {
      int q = to[v][x];
      if (len[v]+1 == len[q]) {
        link[last] = q;
      } else {
        len.pb(len[v]+1);
        link.pb(link[q]);
        to.pb(to[q]);
        cnt.pb(0); // [OCC]
        link[last] = link[q] = sz(len)-1;
        while (v != -1 && to[v][x] == q) {
          to[v][x] = link[q];
          v = link[v];
        } // d41d
      } // d41d
    } // d41d
  } // d41d

  // Compute some additional stuff (offline)
  void finish() {
    // [OCC]
    inSufs.resize(sz(len));
    rep(i, 1, sz(link)) inSufs[link[i]].pb(i);
    dfsSufs(0);
    // [/OCC]

    // [PATHS]
    paths.assign(sz(len), 0);
    dfs(0);
    // [/PATHS]
  } // d41d

  // Only for [OCC]
  void dfsSufs(int v) {
    each(e, inSufs[v]) {
      dfsSufs(e);
      cnt[v] += cnt[e];
    } // d41d
  } // d41d

  // Only for [PATHS]
  void dfs(int v) {
    if (paths[v]) return;
    paths[v] = 1;
    each(e, to[v]) if (e) {
      dfs(e);
      paths[v] += paths[e];
    } // d41d
  } // d41d

  // Go using edge `c` from state `i`.
  // Returns 0 if edge doesn't exist.
  int next(int i, char c) {
    return to[i][c-AMIN];
  } // d41d

  // Get lexicographically k-th substring
  // of represented string; time: O(|substr|)
  // Empty string has index 0.
  // Requires [PATHS] extension.
  string lex(ll k) {
    string s;
    int v = 0;
    while (k--) rep(i, 0, ALPHA) {
      int e = to[v][i];
      if (e) {
        if (k < paths[e]) {
          s.pb(char(AMIN+i));
          v = e;
          break;
        } // d41d
        k -= paths[e];
      } // d41d
    } // d41d
    return s;
  } // d41d
}; // d41d
```

text/suffix_tree.h d41d

```cpp
constexpr int ALPHA = 26;

// Ukkonen's algorithm for online suffix tree
// construction; space: O(n*ALPHA); time: O(n)
// Real tree nodes are called dedicated nodes.
// "Nodes" lying on compressed edges are called
// implicit nodes and are represented
// as pairs (lower node, label index).
// Labels are represented as intervals [L;R)
// which refer to substrings [L;R) of txt.
// Leaves have labels of form [L;infinity),
// use getR to get current right endpoint.
// Suffix links are valid only for internal
// nodes (non-leaves).
struct SufTree {
  Vi txt; // Text for which tree is built
  // to[v][c] = edge with label starting with c
  //            from node v
  vector<array<int, ALPHA>> to{ {} };
  Vi L{0}, R{0}; // Parent edge label endpoints
  Vi par{0};     // Parent link
  Vi link{0};    // Suffix link
  Pii cur{0, 0}; // Current state

  // Get current right end of node label
  int getR(int i) { return min(R[i],sz(txt)); }

  // Follow edge `e` of implicit node `s`.
  // Returns (-1, -1) if there is no edge.
  Pii next(Pii s, int e) {
    if (s.y < getR(s.x))
      return txt[s.y] == e ? mp(s.x, s.y+1)
                           : mp(-1, -1);
    e = to[s.x][e];
    return e ? mp(e, L[e]+1) : mp(-1, -1);
  } // d41d

  // Create dedicated node for implicit node
  // and all its suffixes
  int split(Pii s) {
    if (s.y == R[s.x]) return s.x;

    int t = sz(to); to.pb({});
    to[t][txt[s.y]] = s.x;
    L.pb(L[s.x]);
    R.pb(L[s.x] = s.y);
    par.pb(par[s.x]);
    par[s.x] = to[par[t]][txt[L[t]]] = t;
    link.pb(-1);

    int v = link[par[t]], l = L[t] + !par[t];
    while (l < R[t]) {
      v = to[v][txt[l]];
      l += getR(v) - L[v];
    } // d41d

    v = split({v, getR(v)-l+R[t]});
    link[t] = v;
    return t;
  } // d41d

  // Append letter from [0;ALPHA) to the back
  void add(int x) { // amortized time: O(1)
    Pii t; txt.pb(x);
    while ((t = next(cur, x)).x == -1) {
      int m = split(cur);
      to[m][x] = sz(to);
      to.pb({});
      par.pb(m);
      L.pb(sz(txt)-1);
      R.pb(INT_MAX);
      link.pb(-1);
      cur = {link[m], getR(link[m])};
      if (!m) return;
    } // d41d
    cur = t;
  } // d41d
}; // d41d
```

text/z_function.h d41d

```cpp
// Computes Z function array; time: O(n)
// zf[i] = max common prefix of str and str[i:]
template<class T> Vi prefPref(const T& str) {
  int n = sz(str), b = 0, e = 1;
  Vi zf(n);
  rep(i, 1, n) {
    if (i < e) zf[i] = min(zf[i-b], e-i);
    while (i+zf[i] < n &&
      str[zf[i]] == str[i+zf[i]]) zf[i]++;
    if (i+zf[i] > e) b = i, e = i+zf[i];
  } // d41d
  zf[0] = n;
  return zf;
} // d41d
```

trees/centroid_decomp.h d41d

```cpp
// Centroid decomposition; space: O(n lg n)
// UNTESTED
struct CentroidTree {
  // child[v] = children of v in centroid tree
  // par[v] = parent of v in centroid tree
  //          (-1 for root)
  // depth[v] = depth of v in centroid tree
  //            (0 for root)
  // ind[v][i] = index of vertex v in i-th
  //             centroid subtree from root
  // size[v] = size of centroid subtree of v
  // subtree[v] = list of vertices
  //              in centroid subtree of v
  // dists[v] = distances from v to vertices
  //            in its centroid subtree
  //            (in the order of subtree[v])
  // neigh[v] = neighbours of v
  //            in its centroid subtree
  // dir[v][i] = index of centroid neighbour
  //             that is first vertex on path
  //             from centroid v to i-th vertex
  //             of centroid subtree
  //             (-1 for centroid)
  vector<Vi> child, ind, dists, subtree,
             neigh, dir;
  Vi par, depth, size;
  int root; // Root centroid

  CentroidTree() {}

  CentroidTree(vector<Vi>& G)
      : child(sz(G)), ind(sz(G)), dists(sz(G)),
        subtree(sz(G)), neigh(sz(G)),
        dir(sz(G)), par(sz(G), -2),
        depth(sz(G)), size(sz(G)) {
    root = decomp(G, 0, 0);
  } // d41d

  void dfs(vector<Vi>& G, int v, int p) {
    size[v] = 1;
    each(e, G[v]) if (e != p && par[e] == -2)
      dfs(G, e, v), size[v] += size[e];
  } // d41d
```

```
  void layer(vector<Vi>& G, int v,
             int p, int c, int d) {
    ind[v].pb(sz(subtree[c]));
    subtree[c].pb(v);
    dists[c].pb(d);
    dir[c].pb(sz(neigh[c])-1);
    each(e, G[v]) if (e != p && par[e] == -2) {
      if (v == c) neigh[c].pb(e);
      layer(G, e, v, c, d+1);
    } // d41d
  } // d41d

  int decomp(vector<Vi>& G, int v, int d) {
    dfs(G, v, -1);
    int p = -1, s = size[v];
  loop:
    each(e, G[v]) {
      if (e != p && par[e] == -2 &&
          size[e] > s/2) {
        p = v; v = e; goto loop;
      } // d41d
    } // d41d

    par[v] = -1;
    size[v] = s;
    depth[v] = d;
    layer(G, v, -1, v, 0);

    each(e, G[v]) if (par[e] == -2) {
      int j = decomp(G, e, d+1);
      child[v].pb(j);
      par[j] = v;
    } // d41d
    return v;
  } // d41d
}; // d41d
```

trees/centroid_offline.h d41d

```
// Helper for offline centroid decomposition
// Usage: CentroidDecomp(G);
// Constructor calls method `process`
// for each centroid subtree.
struct CentroidDecomp {
  vector<Vi>& G; // Reference to target graph
  vector<bool> on; // Is vertex enabled?
  Vi size; // Used internally

  // Run centroid decomposition for graph g
  CentroidDecomp(vector<Vi>& g)
      : G(g), on(sz(g), 1), size(sz(g)) {
    decomp(0);
  } // d41d

  // Compute subtree sizes for subtree rooted
  // at v, ignoring p and disabled vertices
  void computeSize(int v, int p) {
    size[v] = 1;
    each(e, G[v]) if (e != p && on[e])
      computeSize(e, v), size[v] += size[e];
  } // d41d

  void decomp(int v) {
    computeSize(v, -1);
    int p = -1, s = size[v];
  loop:
    each(e, G[v]) {
      if (e != p && on[e] && size[e] > s/2) {
        p = v; v = e; goto loop;
      } // d41d
    } // d41d
    process(v);
    on[v] = 0;
    each(e, G[v]) if (on[e]) decomp(e);
  } // d41d

  // Process current centroid subtree:
  // - v is centroid
  // - boundary vertices have on[x] = 0
  // Formally: Let H be subgraph induced
  // on vertices such that on[v] = 1.
  // Then current centroid subtree is
  // connected component of H that contains v
  // and v is its centroid.
  void process(int v) {
    // Do your stuff here...
  } // d41d
}; // d41d
```

trees/heavylight_decomp.h d41d

```
#include "../structures/segtree_point.h"

// Heavy-Light Decomposition of tree
// with subtree query support; space: O(n)
struct HLD {
  // Subtree of v = [pos[v]; pos[v]+size[v])
  // Chain with v = [chBegin[v]; chEnd[v])
  Vi par;     // Vertex parent
  Vi size;    // Vertex subtree size
  Vi depth;   // Vertex distance to root
  Vi pos;     // Vertex position in "HLD" order
  Vi chBegin; // Begin of chain with vertex
  Vi chEnd;   // End of chain with vertex
  Vi order;   // "HLD" preorder of vertices
  SegTree tree; // Verts are in HLD order

  HLD() {}

  // Initialize structure for tree G
  // and given root; time: O(n lg n)
  // MODIFIES ORDER OF EDGES IN G!
  HLD(vector<Vi>& G, int root)
      : par(sz(G)), size(sz(G)),
        depth(sz(G)), pos(sz(G)),
        chBegin(sz(G)), chEnd(sz(G)) {
    dfs(G, root, -1);
    decomp(G, root, -1, 0);
    tree = {sz(order)};
  } // d41d

  void dfs(vector<Vi>& G, int v, int p) {
    par[v] = p;
    size[v] = 1;
    depth[v] = p < 0 ? 0 : depth[p]+1;

    int& fs = G[v][0];
    if (fs == p) swap(fs, G[v].back());

    each(e, G[v]) if (e != p) {
      dfs(G, e, v);
      size[v] += size[e];
      if (size[e] > size[fs]) swap(e, fs);
    } // d41d
  } // d41d

  void decomp(vector<Vi>& G,
              int v, int p, int chb) {
    pos[v] = sz(order);
    chBegin[v] = chb;
    chEnd[v] = pos[v]+1;
    order.pb(v);

    each(e, G[v]) if (e != p) {
      if (e == G[v][0]) {
        decomp(G, e, v, chb);
        chEnd[v] = chEnd[e];
      } else {
        decomp(G, e, v, sz(order));
      } // d41d
    } // d41d
  } // d41d

  // Get root of chain containing v
  int chRoot(int v) {return order[chBegin[v]];}

  // Level Ancestor Query; time: O(lg n)
  int laq(int v, int level) {
    while (true) {
      int k = pos[v] - depth[v] + level;
      if (k >= chBegin[v]) return order[k];
      v = par[chRoot(v)];
    } // d41d
  } // d41d

  // Lowest Common Ancestor; time: O(lg n)
  int lca(int a, int b) {
    while (chBegin[a] != chBegin[b]) {
      int ha = chRoot(a), hb = chRoot(b);
      if (depth[ha] > depth[hb]) a = par[ha];
      else b = par[hb];
    } // d41d
    return depth[a] < depth[b] ? a : b;
  } // d41d

  // Call func(chBegin, chEnd) on each path
  // segment; time: O(lg n * time of func)
  template<class T>
  void iterPath(int a, int b, T func) {
    while (chBegin[a] != chBegin[b]) {
      int ha = chRoot(a), hb = chRoot(b);
      if (depth[ha] > depth[hb]) {
        func(chBegin[a], pos[a]+1);
        a = par[ha];
      } else {
        func(chBegin[b], pos[b]+1);
        b = par[hb];
      } // d41d
    } // d41d

    if (pos[a] > pos[b]) swap(a, b);
    // Remove +1 from pos[a]+1 for vertices
    // queries (with +1 -> edges).
    func(pos[a]+1, pos[b]+1);
  } // d41d

  // Query path between a and b; O(lg^2 n)
  SegTree::T queryPath(int a, int b) {
    auto ret = SegTree::ID;
    iterPath(a, b, [&](int i, int j) {
      ret = SegTree::merge(ret,
          tree.query(i, j));
    }); // d41d
    return ret;
  } // d41d

  // Query subtree of v; time: O(lg n)
  SegTree::T querySubtree(int v) {
    return tree.query(pos[v], pos[v]+size[v]);
  } // d41d
}; // d41d
```

trees/lca.h d41d

```
// LAQ and LCA using jump pointers
// space: O(n lg n)

struct LCA {
  vector<Vi> jumps;
  Vi level, pre, post;
  int cnt{0}, depth;

  LCA() {}

  // Initialize structure for tree G
  // and root r; time: O(n lg n)
  LCA(vector<Vi>& G, int root)
      : jumps(sz(G)), level(sz(G)),
        pre(sz(G)), post(sz(G)) {
    dfs(G, root, root);
    depth = int(log2(sz(G))) + 2;
    rep(j, 0, depth) each(v, jumps)
      v.pb(jumps[v[j]][j]);
  } // d41d

  void dfs(vector<Vi>& G, int v, int p) {
    level[v] = p == v ? 0 : level[p]+1;
    jumps[v].pb(p);
    pre[v] = ++cnt;
    each(e, G[v]) if (e != p) dfs(G, e, v);
    post[v] = ++cnt;
  } // d41d

  // Check if a is ancestor of b; time: O(1)
  bool isAncestor(int a, int b) {
    return pre[a] <= pre[b] &&
           post[b] <= post[a];
  } // d41d

  // Lowest Common Ancestor; time: O(lg n)
  int operator()(int a, int b) {
    for (int j = depth; j--;)
      if (!isAncestor(jumps[a][j], b))
        a = jumps[a][j];
    return isAncestor(a, b) ? a : jumps[a][0];
  } // d41d

  // Level Ancestor Query; time: O(lg n)
  int laq(int a, int lvl) {
    for (int j = depth; j--;)
      if (lvl <= level[jumps[a][j]])
        a = jumps[a][j];
    return a;
  } // d41d

  // Get distance from a to b; time: O(lg n)
```

```
  int distance(int a, int b) {
    return level[a] + level[b] -
           level[operator()(a, b)]*2;
  } // d41d

  // Get k-th vertex on path from a to b,
  // a is 0, b is last; time: O(lg n)
  // Returns -1 if k > distance(a, b)
  int kthVertex(int a, int b, int k) {
    int c = operator()(a, b);
    if (level[a]-k >= level[c])
      return laq(a, level[a]-k);
    k += level[c]*2 - level[a];
    return (k > level[b] ? -1 : laq(b, k));
  } // d41d
}; // d41d
```

trees/lca_linear.h d41d

```
// LAQ and LCA using jump pointers
// with linear memory; space: O(n)
// UNTESTED

struct LCA {
  Vi par, jmp, depth, pre, post;
  int cnt{0};

  LCA() {}

  // Initialize structure for tree G
  // and root v; time: O(n lg n)
  LCA(vector<Vi>& G, int v)
      : par(sz(G), -1), jmp(sz(G), v),
        depth(sz(G)), pre(sz(G)), post(sz(G)) {
    dfs(G, v);
  } // d41d

  void dfs(vector<Vi>& G, int v) {
    int j = jmp[v], k = jmp[j], x =
      depth[v]+depth[k] == depth[j]*2 ? k : v;
    pre[v] = ++cnt;
    each(e, G[v]) if (!pre[e]) {
      par[e] = v; jmp[e] = x;
      depth[e] = depth[v]+1;
      dfs(G, e);
    } // d41d
    post[v] = ++cnt;
  } // d41d

  // Level Ancestor Query; time: O(lg n)
  int laq(int v, int d) {
    while (depth[v] > d)
      v = depth[jmp[v]] < d ? par[v] : jmp[v];
    return v;
  } // d41d

  // Lowest Common Ancestor; time: O(lg n)
  int operator()(int a, int b) {
    if (depth[a] > depth[b]) swap(a, b);
    b = laq(b, depth[a]);
    while (a != b) {
      if (jmp[a] == jmp[b])
        a = par[a], b = par[b];
      else
        a = jmp[a], b = jmp[b];
    } // d41d
    return a;
  } // d41d

  // Check if a is ancestor of b; time: O(1)
  bool isAncestor(int a, int b) {
    return pre[a] <= pre[b] &&
           post[b] <= post[a];
  } // d41d

  // Get distance from a to b; time: O(lg n)
  int distance(int a, int b) {
    return depth[a] + depth[b] -
           depth[operator()(a, b)]*2;
  } // d41d

  // Get k-th vertex on path from a to b,
  // a is 0, b is last; time: O(lg n)
  // Returns -1 if k > distance(a, b)
  int kthVertex(int a, int b, int k) {
    int c = operator()(a, b);
    if (depth[a]-k >= depth[c])
      return laq(a, depth[a]-k);
    k += depth[c]*2 - depth[a];
    return (k > depth[b] ? -1 : laq(b, k));
  } // d41d
}; // d41d
```

trees/link_cut_tree.h d41d

```
constexpr int INF = 1e9;

// Link/cut tree; space: O(n)
// Represents forest of (un)rooted trees.
struct LinkCutTree {
  vector<array<int, 2>> child;
  Vi par, prev, flip, size;

  // Initialize structure for n vertices; O(n)
  // At first there's no edges.
  LinkCutTree(int n = 0)
      : child(n, {-1, -1}), par(n, -1),
        prev(n, -1), flip(n, -1), size(n, 1) {}

  void push(int x) {
    if (x >= 0 && flip[x]) {
      flip[x] = 0;
      swap(child[x][0], child[x][1]);
      each(e, child[x]) if (e>=0) flip[e] ^= 1;
    } // + any other lazy path operations
  } // d41d

  void update(int x) {
    if (x >= 0) {
      size[x] = 1;
      each(e, child[x]) if (e >= 0)
        size[x] += size[e];
    } // + any other path aggregates
  } // d41d

  void auxLink(int p, int i, int ch) {
    child[p][i] = ch;
    if (ch >= 0) par[ch] = p;
    update(p);
  } // d41d

  void rot(int p, int i) {
    int x = child[p][i], g = par[x] = par[p];
    if (g >= 0) child[g][child[g][1] == p] = x;
    auxLink(p, i, child[x][!i]);
    auxLink(x, !i, p);
    swap(prev[x], prev[p]);
    update(g);
  } // d41d

  void splay(int x) {
    while (par[x] >= 0) {
      int p = par[x], g = par[p];
      push(g); push(p); push(x);
      bool f = (child[p][1] == x);
      if (g >= 0) {
        if (child[g][f] == p) { // zig-zig
          rot(g, f); rot(p, f);
        } else { // zig-zag
          rot(p, f); rot(g, !f);
        } // d41d
      } else { // zig
        rot(p, f);
      } // d41d
    } // d41d
    push(x);
  } // d41d

  // After this operation x becomes the end
  // of preferred path starting in root;
  void access(int x) { // amortized O(lg n)
    while (true) {
      splay(x);
      int p = prev[x];
      if (p < 0) break;

      prev[x] = -1;
      splay(p);

      int r = child[p][1];
      if (r >= 0) swap(par[r], prev[r]);
      auxLink(p, 1, x);
    } // d41d
  } // d41d

  // Make x root of its tree; amortized O(lg n)
  void makeRoot(int x) {
    access(x);
    int& l = child[x][0];
    if (l >= 0) {
      swap(par[l], prev[l]);
      flip[l] ^= 1;
      update(l);
      l = -1;
      update(x);
    } // d41d
  } // d41d

  // Find root of tree containing x
  int find(int x) { // time: amortized O(lg n)
    access(x);
    while (child[x][0] >= 0)
      push(x = child[x][0]);
    splay(x);
    return x;
  } // d41d

  // Add edge x-y; time: amortized O(lg n)
  // Root of tree containing y becomes
  // root of new tree.
  void link(int x, int y) {
    makeRoot(x); prev[x] = y;
  } // d41d

  // Remove edge x-y; time: amortized O(lg n)
  // x and y become roots of new trees!
  void cut(int x, int y) {
    makeRoot(x); access(y);
    par[x] = child[y][0] = -1;
    update(y);
  } // d41d

  // Get distance between x and y,
  // returns INF if x and y there's no path.
  // This operation makes x root of the tree!
  int dist(int x, int y) { // amortized O(lg n)
    makeRoot(x);
    if (find(y) != x) return INF;
    access(y);
    int t = child[y][0];
    return t >= 0 ? size[t] : 0;
  } // d41d
}; // d41d
```

util/arc_interval_cover.h d41d

```
using dbl = double;

// Find size of smallest set of points
// such that each arc contains at least one
// of them; time: O(n lg n)
int arcCover(vector<pair<dbl, dbl>>& inters,
             dbl wrap) {
  int n = sz(inters);

  rep(i, 0, n) {
    auto& e = inters[i];
    e.x = fmod(e.x, wrap);
    e.y = fmod(e.y, wrap);
    if (e.x < 0) e.x += wrap, e.y += wrap;
    if (e.x > e.y) e.x += wrap;
    inters.pb({e.x+wrap, e.y+wrap});
  } // d41d

  Vi nxt(n);
  deque<dbl> que;
  dbl r = wrap*4;
  sort(all(inters));

  for (int i = n*2-1; i--;) {
    r = min(r, inters[i].y);
    que.push_front(inters[i].x);
    while (!que.empty() && que.back() > r)
      que.pop_back();
    if (i < n) nxt[i] = i+sz(que);
  } // d41d

  int a = 0, b = 0;
  do {
    a = nxt[a] % n;
    b = nxt[nxt[b]%n] % n;
  } while (a != b);

  int ans = 0;
  while (b < a+n) {
    b += nxt[b%n] - b%n;
    ans++;
```

```
  } // d41d
  return ans;
} // d41d
```

util/bit_hacks.h d41d

```
// __builtin_popcount - count number of 1 bits
// __builtin_clz - count most significant 0s
// __builtin_ctz - count least significant 0s
// __builtin_ffs - like ctz, but indexed from 1
//                 returns 0 for 0
// For ll version add ll to name

using ull = uint64_t;

#define T64(s,up)                            \
  for (ull i=0; i<64; i+=s*2)                \
    for (ull j = i; j < i+s; j++) {          \
      ull a = (M[j] >> s) & up;              \
      ull b = (M[j+s] & up) << s;            \
      M[j] = (M[j] & up) | b;                \
      M[j+s] = (M[j+s] & (up<<s)) | a;       \
    } // d41d

// Transpose 64x64 bit matrix
void transpose64(array<ull, 64>& M) {
  T64(1,  0x5555555555555555);
  T64(2,  0x3333333333333333);
  T64(4,  0xF0F0F0F0F0F0F0F0F);
  T64(8,  0xFF00FF00FF00FF);
  T64(16, 0xFFFF0000FFFF);
  T64(32, 0xFFFFFFFFLL);
} // d41d

// Lexicographically next mask with same
// amount of ones.
int nextSubset(int v) {
  int t = v | (v - 1);
  return (t + 1) | (((~t & -~t) - 1) >>
      (__builtin_ctz(v) + 1));
} // d41d
```

util/bump_alloc.h d41d

```
// Allocator, which doesn't free memory.

char mem[400<<20]; // Set memory limit
size_t nMem;

void* operator new(size_t n) {
  nMem += n; return &mem[nMem-n];
} // d41d
void operator delete(void*) {}
```

util/compress_vec.h d41d

```
// Compress integers to range [0;n) while
// preserving their order; time: O(n lg n)
// Returns mapping: compressed -> original
Vi compressVec(vector<int*>& vec) {
  sort(all(vec),
    [](int* l, int* r) { return *l < *r; });
  Vi old;
  each(e, vec) {
    if (old.empty() || old.back() != *e)
      old.pb(*e);
    *e = sz(old)-1;
  } // d41d
  return old;
} // d41d
```

util/inversion_vector.h d41d

```
// Get inversion vector for sequence of
// numbers in [0;n); ret[i] = count of numbers
// smaller than perm[i] to the left; O(n lg n)
Vi encodeInversions(Vi perm) {
  Vi odd, ret(sz(perm));
  int cont = 1;

  while (cont) {
    odd.assign(sz(perm)+1, 0);
    cont = 0;

    rep(i, 0, sz(perm)) {
      if (perm[i] % 2) odd[perm[i]]++;
      else ret[i] += odd[perm[i]+1];
      cont += perm[i] /= 2;
    } // d41d
  } // d41d
  return ret;
} // d41d

// Count inversions in sequence of numbers
// in [0;n); time: O(n lg n)
ll countInversions(Vi perm) {
  ll ret = 0, cont = 1;
  Vi odd;

  while (cont) {
    odd.assign(sz(perm)+1, 0);
    cont = 0;

    rep(i, 0, sz(perm)) {
      if (perm[i] % 2) odd[perm[i]]++;
      else ret += odd[perm[i]+1];
      cont += perm[i] /= 2;
    } // d41d
  } // d41d
  return ret;
} // d41d
```

util/longest_inc_subseq.h d41d

```
// Longest Increasing Subsequence; O(n lg n)
int lis(const Vi& seq) {
  Vi dp(sz(seq), INT_MAX);
  each(c, seq) *lower_bound(all(dp), c) = c;
  return int(lower_bound(all(dp), INT_MAX)
             - dp.begin());
} // d41d
```

util/max_rects.h d41d

```
struct MaxRect {
  // begin = first column of rectangle
  // end = first column after rectangle
  // hei = height of rectangle
  // touch = columns of height hei inside
  int begin, end, hei;
  Vi touch; // sorted increasing
}; // d41d

// Given consecutive column heights find
// all inclusion-wise maximal rectangles
// contained in "drawing" of columns; time O(n)
vector<MaxRect> getMaxRects(Vi hei) {
  hei.insert(hei.begin(), -1);
  hei.pb(-1);
  Vi reach(sz(hei), sz(hei)-1);
  vector<MaxRect> ans;

  for (int i = sz(hei)-1; --i;) {
    int j = i+1, k = i;
    while (hei[j] > hei[i]) j = reach[j];
    reach[i] = j;

    while (hei[k] > hei[i-1]) {
      ans.pb({ i-1, 0, hei[k], {} });
      auto& rect = ans.back();

      while (hei[k] == rect.hei) {
        rect.touch.pb(k-1);
        k = reach[k];
      } // d41d
      rect.end = k-1;
    } // d41d
  } // d41d
  return ans;
} // d41d
```

util/mo.h d41d

```
// Modified MO's queries sorting algorithm,
// slightly better results than standard.
// Allows to process q queries in O(n*sqrt(q))

struct Query {
  int begin, end;
}; // d41d

// Get point index on Hilbert curve
ll hilbert(int x, int y, int s, ll c = 0) {
  if (s <= 1) return c;
  s /= 2; c *= 4;
  if (y < s)
    return hilbert(x&(s-1), y, s, c+(x>=s)+1);
  if (x < s)
    return hilbert(2*s-y-1, s-x-1, s, c);
  return hilbert(y-s, x-s, s, c+3);
} // d41d

// Get good order of queries; time: O(n lg n)
Vi moOrder(vector<Query>& queries, int maxN) {
  int s = 1;
  while (s < maxN) s *= 2;

  vector<ll> ord;
  each(q, queries)
    ord.pb(hilbert(q.begin, q.end, s));

  Vi ret(sz(ord));
  iota(all(ret), 0);
  sort(all(ret), [&](int l, int r) {
    return ord[l] < ord[r];
  }); // d41d
  return ret;
} // d41d
```

util/parallel_binsearch.h d41d

```
// Run `count` binary searches on [begin;end),
// `cmp` arguments:
// 1) vector<Pii>& - pairs (value, index)
//    which are queries if value of index is
//    greater or equal to value,
//    sorted by value
// 2) vector<bool>& - true at index i means
//    value of i-th query is >= queried value
// Returns vector of found values.
// Time: O((n+c) lg n), where c is cmp time.
template<class T>
Vi multiBS(int begin,int end,int count,T cmp) {
  vector<Pii> ranges(count, {begin, end});
  vector<Pii> queries(count);
  vector<bool> answers(count);

  rep(i,0,count) queries[i]={(begin+end)/2,i};

  for (int k = uplg(end-begin); k > 0; k--) {
    int last = 0, j = 0;
    cmp(queries, answers);

    rep(i, 0, sz(queries)) {
      Pii &q = queries[i], &r = ranges[q.y];
      if (q.x != last) last = q.x, j = i;

      (answers[i] ? r.x : r.y) = q.x;
      q.x = (r.x+r.y) / 2;

      if (!answers[i])
        swap(queries[i], queries[j++]);
    } // d41d
  } // d41d

  Vi ret;
  each(p, ranges) ret.pb(p.x);
  return ret;
} // d41d
```

util/radix_sort.h d41d

```
// Stable countingsort; time: O(k+sz(vec))
// See example usage in radixSort for pairs.
template<class F>
void countSort(Vi& vec, F key, int k) {
  static Vi buf, cnt;
  vec.swap(buf);
  vec.resize(sz(buf));
  cnt.assign(k+1, 0);
  each(e, buf) cnt[key(e)]++;
  rep(i, 1, k+1) cnt[i] += cnt[i-1];
  for (int i = sz(vec)-1; i >= 0; i--)
    vec[--cnt[key(buf[i])]] = buf[i];
} // d41d

// Compute order of elems, k is max key; O(n)
Vi radixSort(const vector<Pii>& elems, int k) {
  Vi order(sz(elems));
  iota(all(order), 0);
  countSort(order,
    [&](int i) { return elems[i].y; }, k);
  countSort(order,
    [&](int i) { return elems[i].x; }, k);
  return order;
} // d41d
```