.bashrc bdbc

```
b() ( g++ -DLOC -O2 -std=c++20 -Wall -W
↪  -Wfatal-errors -Wconversion -Wshadow
↪  -Wlogical-op -Wfloat-equal -o $1.e $@ )

d() ( b $@ -O0 -g -D_GLIBCXX_DEBUG
↪  -fsanitize=address,undefined )

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

.vimrc 68f6

```
se ai cin cul ic is nu scs sw=4 ts=4 so=7 ttm=9
sy on
vn _ :w !cpp -dD -P -fpreprocessed \|
↪  sed -z sg\\sggg \| md5sum \| cut -c-4 <cr>
```

template.cpp 1989

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

using ll  = long long;
using vi  = vector<int>;
using pii = pair<int,int>;

#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()

#define PP(x,y) auto operator<<(auto&o, auto a)
↪  ->decltype(y,o) {o<<"("; x; return o<<")";}

PP(a.print(), a.print());
PP(o << a.x << ", " << a.y, a.y);
PP(for (auto i : a) o << i << ", ", all(a));

void DD(auto s, auto... k) {
  ([&] {
    while (cerr << *s++, 45 % ~*s);
    cerr << ": " << k;
  }(), ...);
} // 606c

#ifdef LOC
auto SS = signal(6, [](int) { *(int*)0=0; });
#define deb(x...)
↪  DD(":, "#x, __LINE__, x), cerr << endl
#else
#define deb(...)
#endif
#define DBP(x...) void print() { DD(#x, x); }

int main() {
  cin.tie(0)->sync_with_stdio(0);
  cout << fixed << setprecision(10);
} // 04a0
```

template.java 9841

```
import java.io.*;
import java.util.*;

public class Task extends PrintWriter {
  BufferedReader reader = new BufferedReader(
      new InputStreamReader(System.in), 32768);
  StringTokenizer tok;

  public static void main(String[] a) {
    try (Task t = new Task()) { t.solve(); }
  } // ffa0

  Task() { super(System.out); }

  String scan() {
    while (tok == null || !tok.hasMoreTokens())
      try {
        tok = new StringTokenizer(
          reader.readLine());
      } catch (Exception e) {
        throw new RuntimeException(e);
      } // fdda
    return tok.nextToken();
  } // 969e

  int scanInt() {
    return Integer.parseInt(scan());
  } // 4a91

  void solve() {
    int n = scanInt();
    printf("hello %d", n);
  } // 3fbe
} // 0966
```

various.bash

```
loo() ( # loo b/d prog.cpp gen.cpp
  set -e; $1 $2; $1 $3
  for ((;;)) {
    ./$3.e > gen.in
    time ./$2.e < gen.in > gen.out
  }
)

cmp() ( # cmp b/d prog.cpp brute.cpp gen.cpp
  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
  }
)

# Other compilation 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
# -fopt-info-all -fopt-info-missed
```

various.h

```
// If math constants like M_PI are undefined:
#define _USE_MATH_DEFINES

// Pragmas
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("arch=???,tune=???")
#define _GLIBCXX_GTHREAD_USE_WEAK 0

// Exit without calling destructors
cout << flush; _Exit(0);

// 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);

// Floats can represent integers up to 19*10^6
// Doubles can represent integers up to 9*10^15

// __lg(x) == floor(log2(x)), undefined for x=0
```

various.py

```
input().split(' ') # Read and split into words
print('abc', end='') # Print without newline

>>> from fractions import *
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> Fraction('1/2') * Fraction('4/3')
Fraction(2, 3)
>>> Fraction(16, 5).numerator
16
>>> Fraction(16, 5).denominator
5

>>> from decimal import *
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017 5...')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

geo2d/circle.h 3dc2

```
#include "vector.h"
#include "line.h" // For line intersections.

// 2D circle structure; UNIT-TESTED
struct circle {
  vec p;     // Center
  sc r2 = 0; // Squared radius
  DBP(p, r2);

  // Returns -1 if point q lies outside circle,
  // 0 if on the edge, 1 if strictly inside.
  // Depends on vec: -, len2
  int side(vec a) {
    return sgn(r2 - (p-a).len2());
  } // f399

#if FLOATING_POINT_GEOMETRY
  // Intersect with another circle.
  // Returns number of intersection points
  // (3 means circles are identical).
  // Arc is CCW w.r.t to `this`, CW for `a`.
  // Depends on vec: +, -, *, len2, perp
  int intersect(circle a, pair<vec,vec>& out) {
    vec d = a.p - p;
    sc d2 = d.len2();
    if (!sgn(d2)) return !sgn(r2-a.r2) * 3;
    sc pd = (d2+r2-a.r2)/2, h2 = r2-pd*pd/d2;
    vec h, t = p + d*(pd/d2);
    int s = sgn(h2)+1;
    if (s > 1) h = d.perp() * sqrt(h2/d2);
    out = {t-h, t+h};
    return s;
  } // 8289

  // Intersect with line.
  // Returns number of intersection points.
  // Points are in order given by a.v.perp().
  int intersect(line a, pair<vec, vec>& out) {
    sc d = a.dist(p), h2 = r2 - d*d;
    vec h, t = a.proj(p);
    int s = sgn(h2)+1;
    if (s > 1)
      h = a.v.perp() * sqrt(h2 / a.v.len2());
    out = {t-h, t+h};
    return s;
  } // 1ee2

  // Find normal vectors of tangents.
  // Returns number of tangent points
  // (3 means circle is degenerated to `a`).
  // Covered arc is CCW between vectors.
  int tangents(vec a, pair<vec, vec>& out) {
    vec d = a - p;
    sc d2 = d.len2(), h2 = d2 - r2;
    if (!sgn(d2)) return !sgn(h2) * 3;
    vec h, t = d * sqrt(r2);
    int s = sgn(h2)+1;
    if (s > 1) h = d.perp() * sqrt(h2);
    out = {(t-h)/d2, (t+h)/d2};
    return s;
  } // 9bf4

  // Find normal vectors of tangents.
  // Returns number of tangent points
  // (3 means circles are identical).
  // Arc for `this` is CCW between vectors.
  // For `a`, it is CW for outer, CCW for inner
  int tangents(circle a, bool inner,
               pair<vec, vec>& out) {
    vec d = a.p - p;
    sc d2 = d.len2();
    sc dr = sqrt(r2)+sqrt(a.r2)*(inner*2-1);
    sc h2 = d2 - dr*dr;
```

```
    if (!sgn(d2)) return !sgn(h2) * 3;
    vec h, t = d * dr;
    int s = sgn(h2)+1;
    if (s > 1) h = d.perp() * sqrt(h2);
    out = {(t-h)/d2, (t+h)/d2};
    return s;
  } // 55a9
#endif
}; // 3f47

#if FLOATING_POINT_GEOMETRY
// Circumcircle. Points must be non-aligned.
// Depends on vec: +,-,*,/, cross, len2, perp
circle circum(vec a, vec b, vec c) {
  b = b-a; c = c-a;
  sc s = b.cross(c);
  assert(sgn(s));
  vec p = a+(b*c.len2()-c*b.len2()).perp()/s/2;
  return { p, (p-a).len2() };
} // fbaa
#endif
```

geo2d/circle_min.h 9444

```
#include "vector.h" // FLOATING_POINT_GEOMETRY
#include "circle.h"

mt19937 rnd(123);

// Minimum circle enclosing a set of points.
circle minDisk(vector<vec> p) { // time: O(n)
  circle c;
  shuffle(all(p), rnd);
  rep(i, 0, sz(p)) if (c.side(p[i]) < 0) {
    c = {p[i], 0};
    rep(j, 0, i) if (c.side(p[j]) < 0) {
      c = {(p[i]+p[j])/2,(p[i]-p[j]).len2()/4};
      rep(k, 0, j) if (c.side(p[k]) < 0)
        c = circum(p[i], p[j], p[k]);
    } // 6e1c
  } // 458a
  return c;
} // 38d5
```

geo2d/convex_hull.h 8159

```
#include "vector.h"

// Find convex hull of points; time: O(n lg n)
// Points are returned counter-clockwise,
// first point is the bottom-leftmost.
// Depends on vec: -, cross, cmpXY
vector<vec> convexHull(vector<vec> points) {
  if (sz(points) <= 1) return points;
  sort(all(points), [](vec l, vec r) {
    return l.cmpYX(r) < 0;
  }); // 4b89
  vector<vec> h(sz(points)+1);
  int s = 0, t = 0;
  rep(i, 0, 2) {
    each(p, points) {
      for (; t >= s+2; t--)
        if ((p-h[t-2]).cross(p-h[t-1]) > eps)
          break;
      h[t++] = p;
    } // 9306
    reverse(all(points));
    s = --t;
  } // 3419
  h.resize(t - (t == 2 && h[0] == h[1]));
  return h;
} // 349e
```

```
// Find point p that maximizes 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.
// Depends on vec: -,cross,perp,upper,cmpAngle
int maxDot(const vector<vec>& h, vec q) {
  int b = 0, e = sz(h);
  while (b+1 < e) {
    int m = (b+e) / 2;
    vec s = h[m] - h[m-1];
    (q.perp().cmpAngle(s) > 0 ? b : e) = m;
  } // be8d
  return q.dot(h[b]-h[0]) > eps ? b : 0;
} // 26f4

#include "segment.h"

// Get distance from point to a hull; O(lg n)
// Returns -1 if point is strictly inside.
// Points are expected to be in the same order
// as output from convexHull function.
// Depends on: maxDot
// Depends on vec: -, dot, cross, len, perp,
//                 upper, cmpAngle
// Depends on seg: side, dist
double hullDist(const vector<vec>& h, vec q) {
  if (sz(h) == 1) return (q-h[0]).len();
  int b = (h[0]-q).upper() ? maxDot(h,{0,1}):0;
  int n = sz(h), e = b+n;
  vec p = h[b];
  while (b+1 < e) {
    int m = (b+e) / 2;
    vec s = h[m%n], t = h[++m%n];
    sc x = ((s-t).cross(q-s) < -eps ?
      (q-p).cross(s-p) : (s-t).dot(q-s));
    (sgn(x) < !(m%n) ? b : e) = m-1;
  } // eba8
  seg s{h[b%n], h[e%n]}, t{h[e%n], h[++e%n]};
  return s.side(q) + t.side(q) < 2 ?
    s.dist(q) : -1;
} // 6d03
```

geo2d/convex_hull_sum.h a935

```
#include "vector.h"

// Minkowski sum of given convex polygons.
// Points are expected to be in the same order
// as output from convexHull function; O(n+m)
// Depends on vec: +, -, cross, upper, cmpAngle
vector<vec> hullSum(const vector<vec>& A,
                    const vector<vec>& B) {
  int n = sz(A), m = sz(B), i = 0, j = 0;
  if (!n || !m) return {};
  vector<vec> C = {A[0]+B[0]};
  while (i+j < n+m) {
    vec a = A[(i+1)%n] - A[i%n];
    vec b = B[(j+1)%m] - B[j%m], v = C.back();
    int s = (i==n) - (j==m) ?: a.cmpAngle(b);
    if (s <= 0) v = v+a, i++;
    if (s >= 0) v = v+b, j++;
    C.pb(v);
  } // f93a
  C.pop_back();
  return C;
} // 00a3
```

geo2d/delaunay.h 2a73

```
#include "vector.h"
#include "../geo3d/convex_hull.h"

// Delaunay triangulation using 3D convex hull.
// Faces are in CCW order. Doesn't work if
// all points are colinear or on a same circle!
// time and memory: O(n log n)
vector<Triple> delaunay(vector<vec>& p) {
  assert(sz(p) >= 3);
  if (sz(p) == 3) {
    int d = ((p[1]-p[0]).cross(p[2]-p[0]) < 0);
    return {{0, 1+d, 2-d}};
  } // 5c2e

  vector<vec3> p3;
  each(e, p) p3.pb({e.x, e.y, e.len2()});

  auto hull = convexHull(p3);
  erase_if(hull, [&](auto& t) {
    vec a = p[t[0]], b = p[t[1]], c = p[t[2]];
    swap(t[1], t[2]);
    return (b-a).cross(c-a) > -eps;
  }); // 794c
  return hull;
} // 294b
```

geo2d/halfplanes.h 714f

```
#include "vector.h" // FLOATING_POINT_GEOMETRY
#include "line.h"

// Halfplane intersection; time: O(n lg n)
// Behaviour is undefined if intersection
// is unbounded, add bounding-box if necessary!
// Returns:
// - vertices of intersection area in CCW order
//   starting from bottom-leftmost vertex;
// - empty vector if intersection is empty.
// Degenerate cases are supported.
// Works only with floating-point geometry.
// Depends on vec: -, *, /, dot, cross, len,
//                 perp, ==, upper, cmpAngle
// Depends on line: side, intersect
vector<vec> intersectHalfs(vector<line> in) {
  sort(all(in), [](line a, line b) {
    return (a.v.perp().cmpAngle(b.v.perp()) ?:
            a.c*b.v.len() - b.c*a.v.len()) < 0;
  }); // 8e73

  int a = 1, b = 1, k = 0, n = sz(in);
  vector<line> dq(n+2);
  vector<vec> out(n+1);
  dq[1] = in[0];

  rep(i, 1, n+1) {
    line t = (i < n ? in[i] : dq[a]);
    while (a < b && t.side(out[b-1]) > 0) b--;
    while (a < b && t.side(out[a]) > 0) a++;
    if (t.intersect(dq[b], out[b])) dq[++b]=t;
  } // fecd

  out[0] = out[--b];
  rep(i, a, b)
    if (out[i] != out[0] && out[i] != out[k])
      out[++k] = out[i];
  out.resize(k+1);
  each(t, in) if (t.side(out[0]) > 0) return{};
  return out;
} // 33ee
```

geo2d/line.h 03dd

```
#include "vector.h"

// 2D line/halfplane structure; UNIT-TESTED
struct line {
  // For lines: v * point == c
  // For halfplanes: v * point <= c
  // (i.e. normal vector points outside)
  vec v;    // Normal vector
  sc c = 0; // Offset
  DBP(v, c);

  // Distance from point to line.
  // Depends on vec: dot, len
  double dist(vec a) {
    return fabs(v.dot(a) - c) / v.len();
  } // 79e6

  // Returns 0 if point a lies on the line,
  // 1 if on side where normal vector points,
  // -1 if on the other side.
  // Depends on vec: dot
  int side(vec a) { return sgn(v.dot(a)-c); }

#if FLOATING_POINT_GEOMETRY
  // Orthogonal projection of point on line.
  // Depends on vec: -, *, dot, len2
  vec proj(vec a) {
    return a - v * ((v.dot(a)-c) / v.len2());
  } // 406e

  // Intersect this line with line a, returns
  // true on success (false if parallel).
  // Intersection point is saved to `out`.
  // Depends on vec: -, *, /, cross, perp
  bool intersect(line a, vec& out) {
    sc d = v.cross(a.v);
    if (!sgn(d)) return 0;
    out = (v*a.c - a.v*c).perp() / d;
    return 1;
  } // c152
#endif
}; // c1c3

// Line through 2 points with normal vector
// pointing to the right of ab vector.
// Depends on vec: -, cross, perp
line through(vec a, vec b) {
  return { (a-b).perp(), a.cross(b) };
} // 9ac7

// Parallel line through point.
// Depends on vec: dot
line parallel(vec a, line b) {
  return { b.v, b.v.dot(a) };
} // 8e1c

// Perpendicular line through point.
// Depends on vec: cross, perp
line perp(vec a, line b) {
  return { b.v.perp(), b.v.cross(a) };
} // 7b75
```

geo2d/rmst.h 9cc3

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

// Rectilinear Minimum Spanning Tree
// (MST in Manhattan metric); time: O(n lg n)
// Returns MST weight. The spanning tree edges
// are saved in `out` as triples (dist, (u,v)).
// Depends on vec: -
ll rmst(vector<vec> points,
        vector<pair<ll, pii>>& edges) {
  vector<pair<ll, pii>> span;
  vi id(sz(points));
```

```
  iota(all(id), 0);

  rep(k, 0, 4) {
    map<ll, ll> S;
    sort(all(id), [&](int i, int j) {
      return (points[i]-points[j]).x <
             (points[j]-points[i]).y;
    }); // f699
    each(i, id) {
      auto it = S.lower_bound(-points[i].y);
      for (; it != S.end(); S.erase(it++)) {
        vec d = points[i] - points[it->y];
        if (d.y > d.x) break;
        span.push_back({d.x+d.y, {i, it->y}});
      } // 490e
      S[-points[i].y] = i;
    } // 0adf
    each(p, points) {
      if (k % 2) p.x = -p.x;
      else swap(p.x, p.y);
    } // 9ec4
  } // 87be

  FAU fau(sz(id));
  ll sum = 0;
  sort(all(span));
  edges.clear();
  each(e, span) if (fau.join(e.y.x, e.y.y))
    edges.pb(e), sum += e.x;
  return sum;
} // b2f5
```

geo2d/segment.h 08b3

```
#include "vector.h"

// 2D segment structure; UNIT-TESTED
struct seg {
  vec a, b; // Endpoints
  DBP(a, b);

  // Check if segment contains point p.
  // Depends on vec: -, dot, cross
  bool contains(vec p) {
    return (a-p).dot(b-p) <= eps &&
           !sgn((a-p).cross(b-p));
  } // 6a6e

  // Returns 0 if point p lies on the line ab,
  // 1 if to the left of the vector ab,
  // -1 if on the right of the vector ab.
  // Depends on vec: cross
  int side(vec p) {
    return sgn((b-a).cross(p-a));
  } // 20a4

  // Distance from segment to point.
  // Depends on vec: -, dot, cross, len
  double dist(vec p) const {
    if ((p-a).dot(b-a) <= eps)
      return (p-a).len();
    if ((p-b).dot(a-b) <= eps)
      return (p-b).len();
    return double(abs((p-a).cross(b-a))) /
      (b-a).len();
  } // a4c6

#if not FLOATING_POINT_GEOMETRY
  // Compare distance to p with sqrt(d2).
  // -1 if smaller, 0 if equal, 1 if greater
  // Depends on vec: -, dot, cross, len2
  int cmpDist(vec p, ll d2) const {
    if ((p-a).dot(b-a) <= 0)
      return sgn((p-a).len2()-d2);
    if ((p-b).dot(a-b) <= 0)
      return sgn((p-b).len2()-d2);
    ll c = (p-a).cross(b-a);
    return sgn(c*c - d2 * (b-a).len2());
  } // 7808
#endif
}; // b2ff
```

geo2d/vector.h ec69

```
// Scalar type: float or integer.
#if FLOATING_POINT_GEOMETRY
  using sc = double;
  constexpr sc eps = 1e-9;
#else
  using sc = ll;
  constexpr sc eps = 0;
#endif

// -1 if a < -eps, 1 if a > eps, 0 otherwise
int sgn(sc a) { return (a>eps) - (a < -eps); }

// 2D point/vector structure; UNIT-TESTED
struct vec {
  using P = vec;
  sc x = 0, y = 0;

  // The following methods are optional
  // and dependencies on them are noted
  // appropriately in library snippets.

  P operator+(P r) const {return{x+r.x,y+r.y};}
  P operator-(P r) const {return{x-r.x,y-r.y};}
  P operator*(sc r) const { return {x*r,y*r}; }
  P operator/(sc r) const { return {x/r,y/r}; }

  sc dot(P r)   const { return x*r.x + y*r.y; }
  sc cross(P r) const { return x*r.y - y*r.x; }
  sc len2()     const { return x*x + y*y; }
  double len()  const { return hypot(x, y); }
  P perp()      const { return {-y,x}; } // CCW

  double angle() const { //[0;2*PI] CCW from OX
    double a = atan2(y, x);
    return (a < 0 ? a+2*M_PI : a);
  } // 7095

  // Equality (with epsilon)
  bool operator==(vec r) const {
    return !sgn(x-r.x) && !sgn(y-r.y);
  } // 485a

  // Lexicographic compare by (y,x) (with eps)
  int cmpYX(P r) const {
    return sgn(y-r.y) ?: sgn(x-r.x);
  } // 1f37

  // Is above OX or on its non-negative part?
  bool upper() const {
    return (sgn(y) ?: sgn(x)) >= 0;
  } // 4fe4

  // Compare vectors by angles.
  // Depends on: cross, upper
  int cmpAngle(P r) const {
    return r.upper()-upper() ?: -sgn(cross(r));
  } // 2ab8

#if FLOATING_POINT_GEOMETRY
  // Rotate counter-clockwise by given angle.
  P rotate(double a) const {
    return {x*cos(a) - y*sin(a),
            x*sin(a) + y*cos(a)}; // 1890
  } // 97e3
#endif
}; // c380
```

geo3d/convex_hull.h 5f7c

```
#include "vector.h"

using Triple = array<int, 3>;
mt19937 rnd(123);

// 3D convex hull; time and memory: O(n log n)
// Returns list of hull faces with vertices
// in CCW order when "looking from outside".
// Doesn't work if all points are coplanar!
// Depends on vec3: -, dot, cross, len2
vector<Triple> convexHull(vector<vec3>& in) {
  int n = sz(in), g = 1;
  vector<Triple> ret, fv, fe;
  vector<vi> fb, bad(n);
  vector<vec3> fq, p(n);
  vi dead, ord(n), link(n, -1);

  iota(all(ord), 0);
  shuffle(all(ord), rnd);
  rep(i, 0, n) p[i] = in[ord[i]];

  // Only needed if there are 4 coplanar points
  vec3 a = p[0], b, c;
  rep(i, 1, n) if (g < 4) {
    swap(p[g], p[i]); swap(ord[g], ord[i]);
    if (g == 1)
      g += sgn((b = p[1]-a).len2());
    else if (g == 2)
      g += sgn((c = b.cross(p[2]-a)).len2());
    else
      g += !!sgn(c.dot(p[3]-a));
  } // 633d
  assert(g == 4); // Not everything coplanar

  auto add = [&](int i, int j, int k) {
    fv.pb({i, j, k});
    fe.pb({-1, -1, -1});
    fq.pb((p[j]-p[i]).cross(p[k]-p[i]));
    fb.pb({});
    dead.pb(1e9);
    return sz(fv)-1;
  }; // 4652

  rep(i, 0, 2) {
    fe[add(0, i+1, 2-i)] = {!i, !i, !i};
    rep(j, 3, n) {
      sc t = fq[i].dot(p[j]-p[0]);
      if (t >= -eps) {
        fb[i].pb(j);
        if (t > eps) bad[j].pb(i);
      } // d64f
    } // a567
  } // e5be

  rep(i, 3, n) {
    int v = -1;
    each(f, bad[i]) dead[f] = min(dead[f], i);
    each(f, bad[i]) if (dead[f] == i) {
      rep(j, 0, 3) if (dead[fe[f][j]] > i) {
        int u = fv[f][(j+1)%3], e = fe[f][j];
        v = fv[f][j];
        fe[g = link[v] = add(v, u, i)][0] = e;
        set_union(all(fb[f]), all(fb[e]),
          back_inserter(fb[g]));
        erase_if(fb[g], [&](int k) {
          return k <= i ||
            fq[g].dot(p[k]-p[fv[g][0]]) <= eps;
        }); // 3119
        each(k, fb[g]) bad[k].pb(g);
        rep(k, 0, 3) if (fv[e][k] == u) {
          fe[e][k] = g;
          break;
        } // c71e
      } // de51
      vi().swap(fb[f]);
    } // 9d4a
    while (v != -1 && fe[link[v]][1] == -1) {
      int u = fv[link[v]][1];
      fe[link[v]][1] = link[u];
      fe[link[u]][2] = link[v];
      v = u;
    } // 5cf7
    vi().swap(bad[i]);
  } // 343c

  rep(i, 0, sz(fv)) if (dead[i] >= n) {
    each(j, fv[i]) j = ord[j];
    ret.pb(fv[i]);
  } // 3c3b
  return ret;
} // 6324
```

geo3d/line.h dd10

```
#include "vector.h"

// 3D line structure; UNTESTED
struct line3 { // p + d*k == point
  vec3 p, d; // Point and direction

  // Distance from point to line.
  // Depends on vec3: dot, len
  double dist(vec3 a) {
    return d.cross(a-p).len() / d.len();
  } // eb4b

  // Distance between two lines.
  // Depends on vec3: -, dot, cross, len2
  double dist(line3 a) {
    vec3 n = d.cross(a.d);
    sc t = n.len2();
    if (!sgn(t)) return dist(a.p);
    return fabs(n.dot(a.p-p)) / sqrt(t);
  } // 22fa

#if FLOATING_POINT_GEOMETRY
  // Closest point to another line.
  // Assumes lines are not parallel!
  // Depends on vec3: -, dot, cross, len
  vec3 closest(line3 a) {
    vec3 n2 = a.d.cross(d.cross(a.d));
    return p + d * n2.dot(a.p-p) / d.dot(n2);
  } // fab4

  // Orthogonal projection of point on line.
  // Depends on vec3: -, *, dot, len2
  vec3 proj(vec3 a) {
    return p + d * (d.dot(a-p) / d.len2());
  } // 0187
#endif
}; // c870

// Line through 2 given points.
// Depends on vec: -
line3 through(vec3 a, vec3 b) {
  return {a, b-a};
} // 5b42
```

geo3d/plane.h 09fe

```
#include "vector.h"
#include "line.h" // For intersections
```

```
// 3D plane/halfspace structure; UNTESTED
struct plane {
  // For planes: v * point == c
  // For halfspaces: v * point <= c
  // (i.e. normal vector points outside)
  vec3 v;    // Normal vector
  sc c = 0; // Offset

  // Distance from point to plane.
  // Depends on vec3: dot, len
  double dist(vec3 a) {
    return fabs(v.dot(a) - c) / v.len();
  } // 79e6

  // Returns 0 if point a lies on the plane,
  // 1 if on side where normal vector points,
  // -1 if on the other side.
  // Depends on vec3: dot
  int side(vec3 a) { return sgn(v.dot(a)-c); }

#if FLOATING_POINT_GEOMETRY
  // Orthogonal projection of point on plane.
  // Depends on vec3: -, *, dot, len2
  vec3 proj(vec3 a) {
    return a - v * ((v.dot(a)-c) / v.len2());
  } // 406e

  // Intersect this plane with line a, returns
  // true on success (false if parallel).
  // Intersection point is saved to `out`.
  // Depends on vec3: -, *, dot
  bool intersect(line3 a, vec3& out) {
    sc t = v.dot(a.d);
    if (!sgn(t)) return 0;
    out = a.p - a.d * ((v.dot(a.p)-c) / t);
    return 1;
  } // a8f2

  // Intersect this plane with plane a, returns
  // true on success (false if parallel).
  // Depends on vec3: -, *, /, dot, cross, len2
  bool intersect(plane a, line3& out) {
    sc t = (out.d = v.cross(a.v)).len2();
    if (!sgn(t)) return 0;
    out.p = (a.v*c - v*a.c).cross(out.d) / t;
    return 1;
  } // 0fa6
#endif
}; // e26a

// Plane through 3 points with normal vector
// pointing upward when viewed CCW.
// Depends on vec3: -, dot, cross
plane span(vec3 a, vec3 b, vec3 c) {
  vec3 v = (b-a).cross(c-a);
  return {v, v.dot(a)};
} // 4fd9
```

geo3d/polyhedron_volume.h f3d3

```
#include "vector.h"

// Signed volume of a polyhedron; UNTESTED
// Faces orientation needs to be consistent.
// Depends on vec3: cross, dot
double volume(vector<vec3>& p, auto& faces) {
  double v = 0;
  for (auto [a, b, c] : faces)
    v += double(p[a].cross(p[b]).dot(p[c]));
  return v / 6;
} // 423d
```

geo3d/sphere.h fcf1

```
#include "vector.h"
#include "line.h" // For line intersections.

// 3D sphere structure; UNTESTED
struct sphere {
  vec3 p;    // Center
  sc r2 = 0; // Squared radius
  DBP(p, r2);

  // Returns -1 if point q lies outside sphere,
  // 0 if on the edge, 1 if strictly inside.
  // Depends on vec3: -, len2
  int side(vec3 a) {
    return sgn(r2 - (p-a).len2());
  } // f399

#if FLOATING_POINT_GEOMETRY
  // Intersect with line.
  // Returns number of intersection points.
  // Points are in order given by direction.
  int intersect(line3 a, pair<vec3,vec3>& out){
    sc d = a.dist(p), h2 = r2 - d*d;
    vec3 h, t = a.proj(p);
    int s = sgn(h2)+1;
    if (s > 1) h = a.d * sqrt(h2 / a.d.len2());
    out = {t-h, t+h};
    return s;
  } // 685b
#endif
}; // 2a59
```

geo3d/vector.h 83e4

```
// Scalar type: float or integer.
#if FLOATING_POINT_GEOMETRY
  using sc = double;
  constexpr sc eps = 1e-9;
#else
  using sc = ll;
  constexpr sc eps = 0;
#endif

// -1 if a < -eps, 1 if a > eps, 0 otherwise
int sgn(sc a) { return (a>eps) - (a < -eps); }

// 3D point/vector structure; UNTESTED
struct vec3 {
  using P = vec3;
  sc x = 0, y = 0, z = 0;

  // The following methods are optional
  // and dependencies on them are noted
  // appropriately in library snippets.

  P operator+(P r) const {
    return {x+r.x, y+r.y, z+r.z};
  } // 9aa2
  P operator-(P r) const {
    return {x-r.x, y-r.y, z-r.z};
  } // 39d7
  P operator*(sc r) const {
    return {x*r, y*r, z*r};
  } // f63f
  P operator/(sc r) const {
    return {x/r, y/r, z/r};
  } // c0d6
  sc dot(P r) const {
    return x*r.x + y*r.y + z*r.z;
  } // af4a
  P cross(P r) const {
    return {y*r.z - z*r.y, z*r.x - x*r.z,
            x*r.y - y*r.x}; // aa5e
  } // 28aa
  sc len2()    const { return x*x+y*y+z*z; }
  double len() const { return hypot(x,y,z); }

  // Equality (with epsilon)
  bool operator==(vec3 r) const {
    return !sgn(x-r.x) && !sgn(y-r.y) &&
           !sgn(z-r.z);
  } // 6e26

  // Angle between vectors in [0,2*PI]
  double angle(vec3 r) const {
    return atan2(cross(r).len(), dot(r));
  } // 8349

#if FLOATING_POINT_GEOMETRY
  // Rotate counter-clockwise around axis.
  P rotate(double angle, vec3 axis) const {
    auto s = sin(angle), c = cos(angle);
    P u = axis / axis.len();
    return u*dot(u)*(1-c)+(*this)*c-cross(u)*s;
  } // 0cd9
#endif
}; // 8cce
```

graphs/2sat.h 4b33

```
// 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;
  } // 98f3

  // Add (i => j) constraint
  void imply(int i, int j) {
    i = i*2 ^ i >> 31;
    j = j*2 ^ j >> 31;
    G[--i].pb(--j); G[j^1].pb(i^1);
  } // 8e25

  // 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 y, x = addVar();
    each(i, vars) {
      imply(x, y = addVar());
      imply(i, -x); imply(i, x = y);
    } // 24aa
  } // 3ed7

  // 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();
    } // 5594
    return 1;
  } // 1e58

  void dfs(int i) {
    if (flags[i]) return;
    flags[i] = 1;
    each(e, G[i]) dfs(e);
    order.pb(i);
  } // d076

  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;
  } // 4c1b
}; // 7be4
```

graphs/bellman_ineq.h cd51

```
struct Ineq {
  ll a, b, c; // a - b >= c
}; // 663a

// 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;
} // 241e
```

graphs/biconnected.h 41fc

```
// 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
// bi.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)) S[i].x ?: dfs(G, i, -1);
    rep(c, 0, sz(verts)) each(v, verts[c])
      at(v).pb(c);
  } // cfce

  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));
    } // 446d
```

```
    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]);
      } // 4fab
      S.resize(S[v].x);
    } // 6d66

    return low;
  } // 7fcc
}; // 3d4a
```

graphs/bridges_online.h ce5a

```
// 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);
  } // ed70

  // 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);
      for (int x : {u, v})
        while ((x = root(x)) != r)
          x = bp[bi(x)] = par[x];
    } 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);
    } // abc7
  } // a6fd

  // Get 2-edge connected component ID
  int bi(int v) { // amortized time: < O(lg n)
    return bp[v] + 1 ? bp[v] = bi(bp[v]) : v;
  } // 3206

  int root(int v) {
    return par[v] == -1 || bi(par[v]) != bi(v)
      ? v : par[v] = root(par[v]);
  } // 2d27

  void dfs(int v, int p) {
    cc[v] = cc[par[v] = p];
    each(e, G[v]) if (e != p) dfs(e, v);
  } // 85f5

  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];
    } // afed
  } // 7f56
}; // 3685
```

graphs/chordal_graph.h a894

```
vi perfectEliminationOrder(vector<vi>& g) {
  int top = 0, n = sz(g);
  vi ord, vis(n), indeg(n);
  vector<vi> bucket(n);
  rep(i, 0, n) bucket[0].push_back(i);
  for (int i = 0; i < n;) {
    while(bucket[top].empty()) --top;
    int u = bucket[top].back();
    bucket[top].pop_back();
    if(vis[u]) continue;
    ord.push_back(u);
    vis[u] = 1;
    ++i;
    each(v, g[u]) {
      if (vis[v]) continue;
      bucket[++indeg[v]].push_back(v);
      top = max(top, indeg[v]);
    } // 8043
  } // 858b
  reverse(all(ord));
  return ord;
} // 429f

bool isChordal(vector<vi>& g, vi ord) {
  int n = sz(g);
  set<pii> edg;
  rep(i, 0, n) each(v, g[i]) edg.insert({i,v});
  vi pos(n); rep(i, 0, n) pos[ord[i]] = i;
  rep(u, 0, n) {
    int mn = n;
    each(v, g[u]) if (pos[u] < pos[v])
      mn = min(mn, pos[v]);
    if (mn != n) {
      int p = ord[mn];
      each(v, g[u]) if (pos[v] > pos[u]
        && v != p && !edg.count({v, p}))
          return 0;
    } // 755f
  } // b0a2
  return 1;
} // 2dec
```

graphs/dense_dfs.h 5dcd

```
#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();
  } // 79e4

  // 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.
  void step(int i, auto 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++;
    } // 4c0a
  } // f045
}; // af18
```

graphs/directed_mst.h 94d4

```
#include "../structures/find_union_undo.h"
#include <ext/pb_ds/priority_queue.hpp>

struct Edge {
  int a, b;
  ll w;
  bool operator<(Edge r) const {
    return w > r.w;
  } // 5e2c
}; // 701d

// Find directed minimum spanning tree
// rooted at vertex `root`; O(m log n)
// Returns weight of found spanning tree.
// par[i] = parent of i-th vertex in the tree,
// par[root] = -1
ll dmst(vector<Edge>& edges,
        int n, int root, vi& par) {
  RollbackFAU dsu(n);
  vector<__gnu_pbds::priority_queue<Edge>>Q(n);
  vector<ll> delta(n);
  each(e, edges) Q[e.b].push(e);

  ll ans = 0;
  vi seen(n, -1), path(n);
  vector<Edge> ed(n), in(n, {-1, -1, 0});
  vector<tuple<int, int, vector<Edge>>> cycs;
  seen[root] = root;

  rep(s, 0, n)
    for (int u = s, pos = 0; seen[u] < 0;) {
      if (Q[u].empty()) return -1;
      auto e = Q[u].top();
      Q[u].pop();
      ans += e.w - delta[u];
      delta[u] = e.w;
      ed[pos] = in[u] = e;
      seen[path[pos++] = u] = s;
      if (seen[u = dsu.find(e.a)] == s) {
        int w, end = pos, t = dsu.time();
        while (dsu.join(u, w = path[--pos])) {
          if (sz(Q[w]) > sz(Q[u])) swap(u, w);
          for (auto f : Q[w]) {
            f.w += delta[u] - delta[w];
            Q[u].push(f);
          } // e37e
        } // f27c
        Q[w = dsu.find(u)].swap(Q[u]);
        delta[w] = delta[u];
        seen[u=w] = -1;
        cycs.pb({u, t, {&ed[pos], &ed[end]}});
      } // cc93
    } // f264

  reverse(all(cycs));
  for (auto &[u, t, e] : cycs) {
    auto s = in[u];
    dsu.rollback(t);
    each(f, e) in[dsu.find(f.b)] = f;
    in[dsu.find(s.b)] = s;
  } // f1a6
  par.resize(n);
  rep(i, 0, n) par[i] = in[i].a;
  return ans;
} // 428e
```

graphs/dominators.h 8db3

```
// Tarjan's algorithm for finding dominators
// in directed graph; time: O(m log n)
// Returns array of immediate dominators idom.
// idom[root] = root
// idom[v] = -1 if v is unreachable from root
vi dominators(const vector<vi>& G, int root) {
  int n = sz(G);
  vector<vi> in(n), bucket(n);
  vi pre(n, -1), anc(n, -1), par(n), best(n);
  vi ord, idom(n, -1), sdom(n, n), rdom(n);

  auto dfs = [&](auto f, int v, int p)->void {
    if (pre[v] == -1) {
      par[v] = p;
      pre[v] = sz(ord);
      ord.pb(v);
      each(e, G[v]) in[e].pb(v), f(f, e, v);
    } // 9c70
  }; // 495a

  auto find = [&](auto f, int v)->pii {
    if (anc[v] == -1) return {best[v], v};
    int b; tie(b, anc[v]) = f(f, anc[v]);
    if (sdom[b] < sdom[best[v]]) best[v] = b;
    return {best[v], anc[v]};
  }; // cf13

  rdom[root] = idom[root] = root;
  iota(all(best), 0);
  dfs(dfs, root, -1);

  rep(i, 0, sz(ord)) {
    int v = ord[sz(ord)-i-1], b = pre[v];
    each(e, in[v])
      b = min(b, pre[e] < pre[v] ? pre[e] :
                 sdom[find(find, e).x]);
    each(u, bucket[v]) rdom[u]=find(find,u).x;
    sdom[v] = b;
    anc[v] = par[v];
    bucket[ord[sdom[v]]].pb(v);
  } // 3663

  each(v, ord) idom[v] = (rdom[v] == v ?
    ord[sdom[v]] : idom[rdom[v]]);
  return idom;
} // b856
```

graphs/edge_color_bipart.h dbb2

```
// Bipartite edge coloring; time: O(nm)
// `edges` is list of (left vert, right vert),
// where vertices on both sides are indexed
// from 0 to n-1. Returns number of used colors
// (which is equal to max degree).
// col[i] = color of i-th edge [0..max_deg-1]
int colorEdges(vector<pii>& edges,
               int n, vi& col) {
  int m = sz(edges), c[2] = {}, ans = 0;
  vi deg[2];
  vector<vector<pii>> has[2];
  col.assign(m, 0);
  rep(i, 0, 2) {
    deg[i].resize(n+1);
    has[i].resize(n+1, vector<pii>(n+1));
```

```
} // 693b

auto dfs = [&](auto f, int x, int p)->void {
  pii i = has[p][x][c[!p]];
  if (has[!p][i.x][c[p]].y) f(f, i.x, !p);
  else has[!p][i.x][c[!p]] = {};
  has[p][x][c[p]] = i;
  has[!p][i.x][c[p]] = {x, i.y};
  if (i.y) col[i.y-1] = c[p]-1;
}; // 08b0

rep(i, 0, m) {
  int x[2] = {edges[i].x+1, edges[i].y+1};
  rep(d, 0, 2) {
    deg[d][x[d]]++;
    ans = max(ans, deg[d][x[d]]);
    for (c[d] = 1; has[d][x[d]][c[d]].y;)
      c[d]++;
  } // 9454
  if (c[0]-c[1]) dfs(dfs, x[1], 1);
  rep(d, 0, 2)
    has[d][x[d]][c[0]] = {x[!d], i+1};
  col[i] = c[0]-1;
} // 46c6
return ans;
} // 5ab4
```

graphs/edge_color_vizing.h a53f

```
// General graph edge coloring; time: O(nm)
// Finds (D+1)-edge-coloring of given graph,
// where D is max vertex degree.
// Returns vector of edge colors `col`.
// col[i] = color of i-th edge [0..D]
vi vizing(vector<pii>& edges, int n) {
  vi cc(n+1), ret(sz(edges)),
     fan(n), fre(n), loc;
  each(e, edges) cc[e.x]++, cc[e.y]++;
  int u, v, cnt = *max_element(all(cc)) + 1;
  vector<vi> adj(n, vi(cnt, -1));
  each(e, edges) {
    tie(u, v) = e;
    fan[0] = v;
    loc.assign(cnt, 0);
    int at = u, end = u, d, c = fre[u],
        ind = 0, i = 0;
    while (d = fre[v],
           !loc[d] && (v = adj[u][d]) != -1)
      loc[d] = ++ind, cc[ind] = d, fan[ind]=v;
    cc[loc[d]] = c;
    for (int cd = d; at+1; cd ^= c ^ d,
         at = adj[at][cd])
      swap(adj[at][cd], adj[end=at][cd^c^d]);
    while (adj[fan[i]][d] + 1) {
      int x = fan[i], y = fan[++i], f = cc[i];
      adj[u][f] = x; adj[x][f] = u;
      adj[y][f] = -1; fre[y] = f;
    } // 0024
    adj[u][d] = fan[i];
    adj[fan[i]][d] = u;
    for (int y : {fan[0], u, end})
      for (int& z = fre[y] = 0; adj[y][z]+1;)
        z++;
  } // 4240
  rep(i, 0, sz(edges))
    for (tie(u, v) = edges[i];
         adj[u][ret[i]] != v;) ++ret[i];
  return ret;
} // 028a
```

graphs/flow_edmonds_karp.h 4cbc

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

// Edmonds-Karp algorithm for finding
// maximum flow in graph; time: O(V*E^2)
struct MaxFlow {
  struct Edge {
    int dst, inv;
    flow_t flow, cap;
  }; // a53c

  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.pb({}); return sz(G)-1; }

  // Add edge from u to v with capacity cap
  // and reverse capacity rcap.
  // Returns edge index in adjacency list of u.
  int 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 });
    return sz(G[u])-1;
  } // c96a

  // Compute maximum flow from src to dst.
  flow_t maxFlow(int src, int dst) {
    flow_t i, m, f = 0;
    each(v, G) each(e, v) e.flow = 0;

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

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

      if (i == dst) {
        while (i != src) {
          auto& e = G[i][prev[i]];
          e.flow -= m;
          G[i = e.dst][e.inv].flow += m;
        } // 1f86
        f += m;
        goto nxt;
      } // 43a2

      each(e, G[i])
        if (add[e.dst] < 0 && e.flow < e.cap) {
          Q.push(e.dst);
          prev[e.dst] = e.inv;
          add[e.dst] = min(m, e.cap-e.flow);
        } // 4cdb
    } // 887e

    return f;
  } // cec0

  // Get flow through e-th edge of vertex v
  flow_t getFlow(int v, int e) {
    return G[v][e].flow;
  } // 0faf

  // Get if v belongs to cut component with src
  bool cutSide(int v) { return add[v] >= 0; }
}; // d858
```

graphs/flow_min_cost.h 7182

```
using flow_t = ll;
constexpr flow_t INF = 1e18;

// Min cost max flow using cheapest paths;
// time: O(nm + |f|*(m log n))
// or O(|f|*(m log n)) if costs are nonnegative
struct MCMF {
  struct Edge {
    int dst, inv;
    flow_t flow, cap, cost;
  }; // 20f7

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

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

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

  // Add edge from u to v.
  // Returns edge index in adjacency list of u.
  int addEdge(int u, int v,
              flow_t cap, flow_t cost) {
    G[u].pb({ v, sz(G[v]), 0, cap, cost });
    G[v].pb({ u, sz(G[u])-1, 0, 0, -cost });
    return sz(G[u])-1;
  } // 1095

  // Compute minimum cost maximum flow
  // from src to dst. `f` is set to flow value,
  // `c` is set to total cost value.
  // Returns false iff negative cycle
  // is reachable from from source.
  bool maxFlow(int src, int dst,
               flow_t& f, flow_t& c) {
    flow_t m;
    f = c = 0;
    each(v, G) each(e, v) e.flow = 0;

#if FLOW_NONNEGATIVE_COSTS
    vector<flow_t> pot(sz(G));
#else
    // Bellman-Ford O(n*m)
    vector<flow_t> pot(sz(G), INF);
    pot[src] = 0;
    int it = sz(G), ch = 1;
    while (ch-- && it--)
      rep(s, 0, sz(G)) if (pot[s] != INF)
        each(e, G[s]) if (e.cap)
          if ((m = pot[s]+e.cost) < pot[e.dst])
            pot[e.dst] = m, ch = 1;
    if (it < 0) return 0;
#endif

  nxt:
    vi prev(sz(G), -1);
    vector<flow_t> dist(sz(G), INF);
    priority_queue<pair<flow_t, int>> Q;
    add.assign(sz(G), -1);
    Q.push({0, src});
    add[src] = INF;
    dist[src] = 0;

    while (!Q.empty()) {
      auto [d, i] = Q.top();
      Q.pop();
      if (d != -dist[i]) continue;
      m = add[i];

      if (i == dst) {
        f += m;
        c += m * (dist[i]-pot[src]+pot[i]);
        while (i != src) {
          auto& e = G[i][prev[i]];
          e.flow -= m;
          G[i = e.dst][e.inv].flow += m;
        } // 1f86
        rep(j, 0, sz(G))
          pot[j] = min(pot[j]+dist[j], INF);
        goto nxt;
      } // 36d4

      each(e, G[i]) if (e.flow < e.cap) {
        d = dist[i]+e.cost+pot[i]-pot[e.dst];
        if (d < dist[e.dst]) {
          Q.push({-d, e.dst});
          prev[e.dst] = e.inv;
          add[e.dst] = min(m, e.cap-e.flow);
          dist[e.dst] = d;
        } // 5ee6
      } // b6b2
    } // 9eba
    return 1;
  } // a34d

  // Get flow through e-th edge of vertex v
  flow_t getFlow(int v, int e) {
    return G[v][e].flow;
  } // 0faf

  // Get if v belongs to cut component with src
  bool cutSide(int v) { return add[v] >= 0; }
}; // 691d
```

graphs/flow_push_relabel.h 5c4b

```
using flow_t = int;

// Push-relabel algorithm for maximum flow;
// O(V^2*sqrt(E)), but very fast in practice.
struct MaxFlow {
  struct Edge {
    int to, inv;
    flow_t rem, cap;
  }; // bc77

  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 k vertices
  MaxFlow(int k = 0) : G(k) {}

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

  // Add edge from u to v with capacity cap
  // and reverse capacity rcap.
  // Returns edge index in adjacency list of u.
  int 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 });
    return sz(G[u])-1;
  } // c96a

  void raise(int v, int h) {
    prv[nxt[prv[v]] = nxt[v]] = prv[v];
```

```
    hei[v] = h;
    if (extra[v] > 0) {
      bot[v] = act[h]; act[h] = v;
      high = max(high, h);
    } // d7ee
    if (h < n) cut = max(cut, h+1);
    nxt[v] = nxt[prv[v]] = h += n];
    prv[nxt[nxt[h] = v]] = v;
  } // 5274

  void global(int s, int t) {
    hei.assign(n, n*2);
    act.assign(n*2, -1);
    iota(all(prv), 0);
    iota(all(nxt), 0);
    hei[t] = high = cut = work = 0;
    hei[s] = n;
    for (int x : {t, s})
      for (Q.push(x); !Q.empty(); Q.pop()) {
        int v = Q.front();
        each(e, G[v])
          if (hei[e.to] == n*2 &&
              G[e.to][e.inv].rem)
            Q.push(e.to), raise(e.to,hei[v]+1);
      } // 1901
  } // 3181

  void push(int v, Edge& e, bool z) {
    auto f = min(extra[v], e.rem);
    if (f > 0) {
      if (z && !extra[e.to]) {
        bot[e.to] = act[hei[e.to]];
        act[hei[e.to]] = e.to;
      } // 9d90
      e.rem -= f; G[e.to][e.inv].rem += f;
      extra[v] -= f; extra[e.to] += f;
    } // 0ffb
  } // da44

  void discharge(int v) {
    int h = n*2, k = hei[v];

    rep(j, 0, sz(G[v])) {
      auto& e = G[v][arc[v]];
      if (e.rem) {
        if (k == hei[e.to]+1) {
          push(v, e, 1);
          if (extra[v] <= 0) return;
        } else h = min(h, hei[e.to]+1);
      } // 87c1
      if (++arc[v] >= sz(G[v])) arc[v] = 0;
    } // 9741

    if (k < n && nxt[k+n] == prv[k+n]) {
      rep(j, k, cut) while (nxt[j+n] < n)
        raise(nxt[j+n], n);
      cut = k;
    } else raise(v, h), work++;
  } // b64f

  // Compute maximum flow from src to dst
  flow_t maxFlow(int src, int dst) {
    extra.assign(n = sz(G), 0);
    arc.assign(n, 0);
    prv.resize(n*3);
    nxt.resize(n*3);
    bot.resize(n);
    each(v, G) each(e, v) e.rem = e.cap;

    each(e, G[src])
      extra[src] = e.cap, push(src, e, 0);
    global(src, dst);

    for (; high; high--)
      while (act[high] != -1) {
        int v = act[high];
        act[high] = bot[v];
        if (v != src && hei[v] == high) {
          discharge(v);
          if (work > 4*n) global(src, dst);
        } // 7dcc
      } // 26d4

    return extra[dst];
  } // aa5e

  // Get flow through e-th edge of vertex v
  flow_t getFlow(int v, int e) {
    return G[v][e].cap - G[v][e].rem;
  } // 812c

  // Get if v belongs to cut component with src
  bool cutSide(int v) { return hei[v] >= n; }
}; // b6f4
```

graphs/flow_with_demands.h e1c0

```
#include "flow_edmonds_karp.h"
//#include "flow_push_relabel.h" // if you need

// Flow with demands; time: O(maxflow)
struct FlowDemands {
  MaxFlow net;
  vector<vector<flow_t>> demands;
  flow_t total = 0;

  // Initialize for k vertices
  FlowDemands(int k = 0) : net(2) {
    while (k--) addVert();
  } // 7bdf

  // Add new vertex
  int addVert() {
    int v = net.addVert();
    demands.pb({});
    net.addEdge(0, v, 0);
    net.addEdge(v, 1, 0);
    return v-2;
  } // 48b6

  // Add edge from u to v with demand dem
  // and capacity cap (dem <= flow <= cap).
  // Returns edge index in adjacency list of u.
  int addEdge(int u, int v,
              flow_t dem, flow_t cap) {
    demands[u].pb(dem);
    demands[v].pb(0);
    total += dem;
    net.G[0][v].cap += dem;
    net.G[u+2][1].cap += dem;
    return net.addEdge(u+2, v+2, cap-dem) - 2;
  } // a403

  // Check if there exists a flow with value f
  // for source src and destination dst.
  // For circulation, you can set args to 0.
  bool canFlow(int src, int dst, flow_t f) {
    net.addEdge(dst += 2, src += 2, f);
    f = net.maxFlow(0, 1);
    net.G[src].pop_back();
    net.G[dst].pop_back();
    return f == total;
  } // 6285

  // Get flow through e-th edge of vertex v
  flow_t getFlow(int v, int e) {
    return net.getFlow(v+2,e+2)+demands[v][e];
  } // 6cf6
}; // f735
```

graphs/global_min_cut.h c9e3

```
// Find a minimum cut in an undirected graph
// with non-negative edge weights
// given its adjacency matrix M; time: O(n^3)
// `out` contains vertices on one side.
ll minCut(vector<vector<ll>> M, vi& out) {
  int n = sz(M);
  ll ans = INT64_MAX;
  vector<vi> co(n);
  rep(i, 0, n) co[i].pb(i);
  out.clear();
  rep(ph, 1, n) {
    auto w = M[0];
    size_t s = 0, t = 0;
    // O(V^2) -> O(E log V) with priority queue
    rep(it, 0, n-ph) {
      w[t] = INT64_MIN; s = t;
      t = max_element(all(w)) - w.begin();
      rep(i, 0, n) w[i] += M[t][i];
    } // 0831
    ll alt = w[t] - M[t][t];
    if (alt < ans) ans = alt, out = co[t];
    co[s].insert(co[s].end(), all(co[t]));
    rep(i, 0, n) M[s][i] += M[t][i];
    rep(i, 0, n) M[i][s] = M[s][i];
    M[0][t] = INT64_MIN;
  } // df69
  return ans;
} // 6664
```

graphs/gomory_hu.h a520

```
#include "flow_edmonds_karp.h"
//#include "flow_push_relabel.h" // if you need

struct Edge {
  int a, b; // vertices
  flow_t w; // weight
}; // c331

// 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;
  } // 5ae4

  return ret;
} // afdb
```

graphs/kth_shortest.h b346

```
constexpr ll INF = 1e18;

// Eppstein's k-th shortest path algorithm;
// time and space: O((m+k) log (m+k))
struct Eppstein {
  using T = ll; // Type for edge weights
  using Edge = pair<int, T>;

  struct Node {
    int E[2] = {}, s = 0;
    Edge x;
  }; // fc26

  T shortest; // Shortest path length
  priority_queue<pair<T, int>> Q;
  vector<Node> P{1};
  vi h;

  // Initialize shortest path structure for
  // weighted graph G, source s and target t;
  // time: O(m log m)
  Eppstein(vector<vector<Edge>>& G,
           int s, int t) {
    int n = sz(G);
    vector<vector<Edge>> H(n);
    rep(i,0,n) each(e,G[i]) H[e.x].pb({i,e.y});

    vi ord, par(n, -1);
    vector<T> d(n, -INF);
    Q.push({d[t] = 0, t});

    while (!Q.empty()) {
      auto v = Q.top();
      Q.pop();
      if (d[v.y] == v.x) {
        ord.pb(v.y);
        each(e, H[v.y]) if (v.x-e.y > d[e.x]) {
          Q.push({d[e.x] = v.x-e.y, e.x});
          par[e.x] = v.y;
        } // 5895
      } // 1b62
    } // 1a6d

    if ((shortest = -d[s]) >= INF) return;
    h.resize(n);

    each(v, ord) {
      int p = par[v];
      if (p+1) h[v] = h[p];
      each(e, G[v]) if (d[e.x] > -INF) {
        T k = e.y - d[e.x] + d[v];
        if (k || e.x != p)
          h[v] = push(h[v], {e.x, k});
        else
          p = -1;
      } // 5e05
    } // 31b9

    P[0].x.x = s;
    Q.push({0, 0});
  } // f546

  int push(int t, Edge x) {
    P.pb(P[t]);
    if (!P[t = sz(P)-1].s || P[t].x.y >= x.y)
      swap(x, P[t].x);
    if (P[t].s) {
      int i = P[t].E[0], j = P[t].E[1];
      int d = P[i].s > P[j].s;
      int k = push(d ? j : i, x);
      P[t].E[d] = k; // Don't inline k!
    } // 10e1
    P[t].s++;
    return t;
  } // a2dc
```

```
  // Get next shortest path length,
  // the first call returns shortest path.
  // Returns -1 if there's no more paths;
  // time: O(log k), where k is total count
  // of nextPath calls.
  ll nextPath() {
    if (Q.empty()) return -1;
    auto v = Q.top();
    Q.pop();
    for (int i : P[v.y].E) if (i)
      Q.push({ v.x-P[i].x.y+P[v.y].x.y, i });
    int t = h[P[v.y].x.x];
    if (t) Q.push({ v.x - P[t].x.y, t });
    return shortest - v.x;
  } // 08af
}; // 9a8d
```

graphs/matching_blossom.h 4650

```
// Edmond's Blossom algorithm for maximum
// matching in general graphs; time: O(nm)
// Returns matching size (edge count).
// match[v] = vert matched to v or -1
int blossom(vector<vi>& G, vi& match) {
  int n = sz(G), cnt = -1, ans = 0;
  match.assign(n, -1);
  vi lab(n), par(n), orig(n), aux(n, -1), q;

  auto blos = [&](int v, int w, int a) {
    while (orig[v] != a) {
      par[v] = w; w = match[v];
      if (lab[w] == 1) lab[w] = 0, q.pb(w);
      orig[v] = orig[w] = a; v = par[w];
    } // 319e
  }; // ab9e

  rep(i, 0, n) if (match[i] == -1)
    each(e, G[i]) if (match[e] == -1) {
      match[match[e] = i] = e; ans++; break;
    } // a22a

  rep(root, 0, n) if (match[root] == -1) {
    fill(all(lab), -1);
    iota(all(orig), 0);
    lab[root] = 0;
    q = {root};
    rep(i, 0, sz(q)) {
      int v = q[i];
      each(x, G[v]) if (lab[x] == -1) {
        lab[x] = 1; par[x] = v;
        if (match[x] == -1) {
          for (int y = x; y+1;) {
            int p = par[y], w = match[p];
            match[match[p] = y] = p; y = w;
          } // 30c1
          ans++;
          goto nxt;
        } // 6fd0
        lab[match[x]] = 0; q.pb(match[x]);
      } else if (lab[x] == 0 &&
                 orig[v] != orig[x]) {
        int a = orig[v], b = orig[x];
        for (cnt++;; swap(a, b)) if (a+1) {
          if (aux[a] == cnt) break;
          aux[a] = cnt;
          a = (match[a]+1 ?
            orig[par[match[a]]] : -1);
        } // 2776
        blos(x, v, a); blos(v, x, a);
      } // 45a1
    } // d488
    nxt:;
  } // 8d8a

  return ans;
} // f17b
```

graphs/matching_blossom_w.h 536a

```
// Edmond's Blossom algorithm for weighted
// maximum matching in general graphs; O(n^3)?
// Weights must be positive (I believe).
struct WeightedBlossom {
  struct edge { int u, v, w; };
  int n, s, nx;
  vector<vector<edge>> g;
  vi lab, match, slack, st, pa, S, vis;
  vector<vi> flo, floFrom;
  queue<int> q;

  // Initialize for k vertices
  WeightedBlossom(int k)
      : n(k), s(n*2+1),
        g(s, vector<edge>(s)),
        lab(s), match(s), slack(s), st(s),
        pa(s), S(s), vis(s), flo(s),
        floFrom(s, vi(n+1)) {
    rep(u, 1, n+1) rep(v, 1, n+1)
      g[u][v] = {u, v, 0};
  } // 5e51

  // Add edge between u and v with weight w
  void addEdge(int u, int v, int w) {
    u++; v++;
    g[u][v].w = g[v][u].w = max(g[u][v].w, w);
  } // d296

  // Compute max weight matching.
  // `count` is set to matching size,
  // `weight` is set to matching weight.
  // Returns vector `match` such that:
  // match[v] = vert matched to v or -1
  vi solve(int& count, ll& weight) {
    fill(all(match), 0);
    nx = n;
    weight = count = 0;
    rep(u, 0, n+1) flo[st[u] = u].clear();
    int tmp = 0;
    rep(u, 1, n+1) rep(v, 1, n+1) {
      floFrom[u][v] = (u-v ? 0 : v);
      tmp = max(tmp, g[u][v].w);
    } // a881
    rep(u, 1, n+1) lab[u] = tmp;
    while (matching()) count++;
    rep(u, 1, n+1)
      if (match[u] && match[u] < u)
        weight += g[u][match[u]].w;
    vi ans(n);
    rep(i, 0, n) ans[i] = match[i+1]-1;
    return ans;
  } // 9ca0

  int delta(edge& e) {
    return lab[e.u]+lab[e.v]-g[e.u][e.v].w*2;
  } // 7b58
  void updateSlack(int u, int x) {
    if (!slack[x] || delta(g[u][x]) <
      delta(g[slack[x]][x])) slack[x] = u;
  } // 1f7f
  void setSlack(int x) {
    slack[x] = 0;
    rep(u, 1, n+1) if (g[u][x].w > 0 &&
      st[u] != x && !S[st[u]])
        updateSlack(u, x);
  } // ee9c
  void push(int x) {
    if (x <= n) q.push(x);
    else rep(i, 0, sz(flo[x])) push(flo[x][i]);
  } // 594d
  void setSt(int x, int b) {
    st[x] = b;
    if (x > n) rep(i, 0, sz(flo[x]))
      setSt(flo[x][i],b);
  } // c5c8
  int getPr(int b, int xr) {
    int pr = int(find(all(flo[b]), xr) -
      flo[b].begin());
    if (pr % 2) {
      reverse(flo[b].begin()+1, flo[b].end());
      return sz(flo[b]) - pr;
    } else return pr;
  } // 399f
  void setMatch(int u, int v) {
    match[u] = g[u][v].v;
    if (u <= n) return;
    edge e = g[u][v];
    int xr = floFrom[u][e.u], pr = getPr(u,xr);
    rep(i, 0, pr)
      setMatch(flo[u][i], flo[u][i^1]);
    setMatch(xr, v);
    rotate(flo[u].begin(), flo[u].begin()+pr,
      flo[u].end());
  } // f19d
  void augment(int u, int v) {
    while (1) {
      int xnv = st[match[u]];
      setMatch(u, v);
      if (!xnv) return;
      setMatch(xnv, st[pa[xnv]]);
      u = st[pa[xnv]], v = xnv;
    } // bd23
  } // e61a
  int getLca(int u, int v) {
    static int t = 0;
    for (++t; u||v; swap(u, v)) {
      if (!u) continue;
      if (vis[u] == t) return u;
      vis[u] = t;
      u = st[match[u]];
      if (u) u = st[pa[u]];
    } // aa78
    return 0;
  } // 9f28
  void blossom(int u, int lca, int v) {
    int b = n+1;
    while (b <= nx && st[b]) ++b;
    if (b > nx) ++nx;
    lab[b] = S[b] = 0;
    match[b] = match[lca];
    flo[b].clear();
    flo[b].pb(lca);
    for (int x=u, y; x != lca; x = st[pa[y]]) {
      flo[b].pb(x);
      flo[b].pb(y = st[match[x]]);
      push(y);
    } // 63e0
    reverse(flo[b].begin()+1, flo[b].end());
    for (int x=v, y; x != lca; x = st[pa[y]]) {
      flo[b].pb(x);
      flo[b].pb(y = st[match[x]]);
      push(y);
    } // 63e0
    setSt(b, b);
    rep(x, 1, nx+1) g[b][x].w = g[x][b].w = 0;
    rep(x, 1, n+1) floFrom[b][x] = 0;
    rep(i, 0, sz(flo[b])) {
      int xs = flo[b][i];
      rep(x, 1, nx+1) if (!g[b][x].w ||
        delta(g[xs][x]) < delta(g[b][x]))
          g[b][x]=g[xs][x], g[x][b]=g[x][xs];
      rep(x, 1, n+1) if (floFrom[xs][x])
        floFrom[b][x] = xs;
    } // 5833
    setSlack(b);
  } // 9000
  void blossom(int b) {
    each(e, flo[b]) setSt(e, e);
    int xr = floFrom[b][g[b][pa[b]].u];
    int pr = getPr(b, xr);
    for (int i = 0; i < pr; i += 2) {
      int xs = flo[b][i], xns = flo[b][i+1];
      pa[xs] = g[xns][xs].u;
      S[xs] = 1; S[xns] = slack[xs] = 0;
      setSlack(xns); push(xns);
    } // f26f
    S[xr] = 1; pa[xr] = pa[b];
    rep(i, pr+1, sz(flo[b])) {
      int xs = flo[b][i];
      S[xs] = -1; setSlack(xs);
    } // a12a
    st[b] = 0;
  } // f750
  bool found(const edge& e) {
    int u = st[e.u], v = st[e.v];
    if (S[v] == -1) {
      pa[v] = e.u; S[v] = 1;
      int nu = st[match[v]];
      slack[v] = slack[nu] = S[nu] = 0;
      push(nu);
    } else if (!S[v]) {
      int lca = getLca(u, v);
      if (!lca) return augment(u, v),
        augment(v, u), 1;
      else blossom(u, lca, v);
    } // ddbb
    return 0;
  } // 1c00
  bool matching() {
    fill(S.begin(), S.begin()+nx+1, -1);
    fill(slack.begin(), slack.begin()+nx+1, 0);
    q = {};
    rep(x, 1, nx+1)
      if (st[x] == x && !match[x])
        pa[x] = S[x] = 0, push(x);
    if (q.empty()) return 0;
    while (1) {
      while (q.size()) {
        int u = q.front(); q.pop();
        if (S[st[u]] == 1) continue;
        rep(v, 1, n+1)
          if (g[u][v].w > 0 && st[u] != st[v]){
            if (!delta(g[u][v])) {
              if (found(g[u][v])) return 1;
            } else updateSlack(u, st[v]);
          } // b782
      } // 4d33
      int d = INT_MAX;
      rep(b, n+1, nx+1)
        if (st[b] == b && S[b] == 1)
```

```
        d = min(d, lab[b]/2);
    rep(x, 1, nx+1)
      if (st[x] == x && slack[x]) {
        if (S[x] == -1)
          d = min(d, delta(g[slack[x]][x]));
        else if (!S[x])
          d = min(d,delta(g[slack[x]][x])/2);
      } // 2a0e
    rep(u, 1, n+1) {
      if (!S[st[u]]) {
        if (lab[u] <= d) return 0;
        lab[u] -= d;
      } else if (S[st[u]] == 1) lab[u] += d;
    } // 4601
    rep(b, n+1, nx+1) if (st[b] == b) {
      if (!S[st[b]]) lab[b] += d*2;
      else if (S[st[b]] == 1) lab[b] -= d*2;
    } // e09b
    q = {};
    rep(x, 1, nx+1)
      if (st[x] == x && slack[x] &&
        st[slack[x]] != x &&
        !delta(g[slack[x]][x]) &&
        found(g[slack[x]][x])) return 1;
    rep(b, n+1, nx+1)
      if (st[b] == b && S[b] == 1 && !lab[b])
        blossom(b);
   } // a122
   return 0;
  } // e966
}; // 35de
```

graphs/matching_boski.h c8ac

```
// Bosek's algorithm for partially online
// bipartite maximum matching - white vertices
// are fixed, black vertices are added
// one by one; time: O(E*sqrt(V))
// Usage: Matching match(num_white);
// match[v] = index of black vertex matched to
//            white vertex v or -1 if unmatched
// match.add(indices_of_white_neighbours);
// Black vertices are indexed in order they
// were added, the first black vertex is 0.
struct Matching : vi {
  vector<vi> adj;
  vi rank, low, pos, vis, seen;
  int k = 0;

  // Initialize structure for n white vertices
  Matching(int n = 0) : vi(n, -1), rank(n) {}

  // Add new black vertex with its neighbours
  // given by `vec`. Returns true if maximum
  // matching is increased by 1.
  bool add(vi vec) {
    adj.pb(move(vec));
    low.pb(0); pos.pb(0); vis.pb(0);
    if (!adj.back().empty()) {
      int i = k;
    nxt:
      seen.clear();
      if (dfs(sz(adj)-1, ++k-i)) return 1;
      each(v, seen) each(e, adj[v])
        if (rank[e] < 1e9 && vis[at(e)] < k)
          goto nxt;
      each(v, seen) each(w, adj[v])
        rank[w] = low[v] = 1e9;
    } // 6aec
    return 0;
  } // d2a7

  bool dfs(int v, int g) {
    if (vis[v] < k) vis[v] = k, seen.pb(v);
    while (low[v] < g) {
      int e = adj[v][pos[v]];
      if (at(e) != v && low[v] == rank[e]) {
        rank[e]++;
        if (at(e) == -1 || dfs(at(e), rank[e]))
          return at(e) = v, 1;
      } else if (++pos[v] == sz(adj[v])) {
        pos[v] = 0; low[v]++;
      } // e532
    } // 3d88
    return 0;
  } // 8561
}; // 4560
```

graphs/matching_turbo.h 6439

```
// 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);

  auto dfs = [&](auto f, int i)->int {
    if (seen[i]) return 0;
    seen[i] = 1;
    each(e, G[i]) {
      if (match[e] < 0 || f(f, match[e])) {
        match[i] = e; match[e] = i;
        return 1;
      } // 893d
    } // 1c44
    return 0;
  }; // c8bd

  while (k) {
    seen.assign(sz(G), 0);
    k = 0;
    rep(i, 0, sz(G)) if (match[i] < 0)
      k += dfs(dfs, i);
    n += k;
  } // 62a7
  return n;
} // 616f

// 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));

  auto dfs = [&](auto f, int i, int c)->void {
    if (col[i]) return;
    col[i] = c+1;
    each(e, G[i]) f(f, e, !c);
  }; // b718

  auto aug = [&](auto f, int i)->void {
    if (seen[i] || col[i] != 1) return;
    seen[i] = 1;
    each(e, G[i]) seen[e] = 1, f(f, match[e]);
  }; // 3452

  rep(i, 0, sz(G)) dfs(dfs, i, 0);
  rep(i, 0, sz(G)) if (match[i]<0) aug(aug, i);
  rep(i, 0, sz(G))
    if (seen[i] == col[i]-1) ret.pb(i);
  return ret;
} // a4c1
```

graphs/matching_weighted.h ed77

```
// Minimum cost bipartite matching; O(n^2*m)
// Input is n x m cost matrix, where n <= m.
// Returns matching weight.
// L[i] = right vertex matched to i-th left
// R[i] = left vertex matched to i-th right
ll hungarian(const vector<vector<ll>>& cost,
             vi& L, vi& R) {
  if (cost.empty())
    return L.clear(), R.clear(), 0;
  int b, c = 0, n = sz(cost), m = sz(cost[0]);
  assert(n <= m);

  vector<ll> x(n), y(m+1);
  L.assign(n, -1);
  R.assign(m+1, -1);

  rep(i, 0, n) {
    vector<ll> sla(m, INT64_MAX);
    vi vis(m+1), prv(m, -1);
    for (R[b = m] = i; R[b]+1; b = c) {
      int a = R[b];
      ll d = INT64_MAX;
      vis[b] = 1;
      rep(j, 0, m) if (!vis[j]) {
        ll cur = cost[a][j] - x[a] - y[j];
        if (cur < sla[j])
          sla[j] = cur, prv[j] = b;
        if (sla[j] < d) d = sla[j], c = j;
      } // 6717
      rep(j, 0, m+1) {
        if (vis[j]) x[R[j]] += d, y[j] -= d;
        else sla[j] -= d;
      } // 8bb3
    } // 01c6
    while (b-m) c = b, R[c] = R[b = prv[b]];
  } // 50bb

  rep(j, 0, m) if (R[j]+1) L[R[j]] = j;
  R.resize(m);
  return -y[m];
} // 0430
```

graphs/matroids.h ca31

```
// 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.
vector<bool> intersectMatroids(
      auto& A, auto& B, int n) {
  vector<bool> ans(n);
  bool ok = 1;

  // NOTE: for weighted matroid intersection
  // find shortest augmenting paths
  // first by weight change, then by length
  // using Bellman-Ford, and skip this speedup:
  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);
    } // 9581

    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);
      } // bd2a
      ans[i] = 1;
    } // bf3e

    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;
        } // 51c8
        ok = 1;
        break;
      } // 384b

      each(j, G[i]) if (prev[j] == -1)
        que.push(j), prev[j] = i;
    } // 6eb6
  } // 3c97

  return ans;
} // 774e

// 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];
  } // 4dfb

  // Check if S+{k} is independent; time: O(1)
  bool canAdd(int k) {
    return tmp[color[k]] > 0;
  } // e312
}; // c7d0

// 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]);
  } // b8b7

  // 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);
  } // 1827

  // Check if S+{k} is independent; time: ~O(1)
  bool canAdd(int k) {
    return
      find(elems[k].x) != find(elems[k].y);
  } // 8ca4
}; // 19d3

// 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];
  } // 9d30

  // 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);
    } // f4e8
    rep(v, 0, n) if (!pre[v]) dfs(v, -1);
  } // dfe1

  // 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]);
  } // f6c5
}; // 4149

// 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;
        } // 1df5
        e ^= base[j];
      } // 8495
    } // 655e
  } // b68c

  // 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];
    } // 49d1
    return 0;
  } // 66ff
}; // 4af3
```

graphs/max_clique.h c219

```
// Quickly finds a maximum clique of a graph
// (given as symmetric bitset matrix;
// self-edges not allowed).
// time: ~1s for n=155 and worst case random
// graphs (p=.90). Faster for sparse graphs.
typedef vector<bitset<200>> vb;
struct MaxClique {
  double limit=0.025, pk=0;
  struct Vertex { int i, d=0; };
  typedef vector<Vertex> vv;
  vb e;
  vv V;
  vector<vi> C;
  vi qmax, q, S, old;
  void init(vv& r) {
    for (auto& v : r) v.d = 0;
    for (auto& v : r) for (auto j : r)
      v.d += e[v.i][j.i];
    sort(all(r),
      [](auto a, auto b) {return a.d > b.d;});
    int mxD = r[0].d;
    rep(i, 0, sz(r)) r[i].d = min(i, mxD) + 1;
  } // dabd
  void expand(vv& R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (sz(R)) {
      if (sz(q) + R.back().d <= sz(qmax))
        return;
      q.push_back(R.back().i);
      vv T;
      for(auto v:R) if (e[R.back().i][v.i])
        T.push_back({v.i});
      if (sz(T)) {
        if (S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1;
        int mnk = max(sz(qmax) - sz(q) + 1, 1);
        C[1].clear(), C[2].clear();
        for (auto v : T) {
          int k = 1;
          auto f=[&](int i){return e[v.i][i];};
          while (any_of(all(C[k]), f)) k++;
          if (k > mxk) mxk=k, C[mxk+1].clear();
          if (k < mnk) T[j++].i = v.i;
          C[k].push_back(v.i);
        } // e825
        if (j > 0) T[j - 1].d = 0;
        rep(k,mnk,mxk + 1) for (int i : C[k])
          T[j].i = i, T[j++].d = k;
        expand(T, lev + 1);
      } else if (sz(q) > sz(qmax)) qmax = q;
      q.pop_back(), R.pop_back();
    } // dea6
  } // f0ce
  vi solve() {
    init(V), expand(V); return qmax; } // 2243
  MaxClique(vb conn)
    : e(conn), C(sz(e)+1), S(sz(C)), old(S) {
    rep(i, 0, sz(e)) V.push_back({i});
  } // cd99
}; // b944
```

graphs/max_clique_chinese.h beca

```
constexpr int N = 405;

// Max clique heuristic that seems to work well
// with geometric packing problems. Vertices
// should be ordered by (X,Y), not shuffled.
struct MaxClique {
  bool g[N][N];
  int n, dp[N], st[N][N], ans, res[N], stk[N];

  void init(int n_) {
    n = n_;
    memset(g, 0, sizeof(g));
  } // 5413

  void addEdge(int u, int v, int w) {
    g[u][v] = w;
  } // 6fb6

  bool dfs(int sz, int num) {
    if (sz == 0) {
      if (num > ans) {
        ans = num;
        copy(stk+1, stk+1+num, res+1);
        return 1;
      } // 9ad5
      return 0;
    } // c6b6
    for (int i = 0; i < sz; i++) {
      if (sz-i+num <= ans) return 0;
      int u = st[num][i];
      if (dp[u]+num <= ans) return 0;
      int cnt = 0;
      rep(j, i+1, sz)
        if (g[u][st[num][j]])
          st[num+1][cnt++] = st[num][j];
      stk[num+1] = u;
      if (dfs(cnt, num + 1)) return 1;
    } // fddd
    return 0;
  } // aae9

  int solve() {
    ans = 0;
    memset(dp, 0, sizeof(dp));
    for (int i = n; i >= 1; i--) {
      int cnt = 0;
      rep(j, i+1, n+1)
        if (g[i][j]) st[1][cnt++] = j;
      stk[1] = i;
      dfs(cnt, 1);
      dp[i] = ans;
    } // 2361
    return ans;
  } // dcc6
}; // 7599
```

graphs/spfa.h 2179

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

// SPFA with subtree erasure heuristic;
// time: pessimistic O(nm), on random O(m)
// Returns array of distances or empty array
// if negative cycle is reachable from source.
// par[v] = parent in shortest path tree
vector<ll> spfa(vector<vector<Edge>>& G,
                vi& par, int src) {
  int n = sz(G);
  vi que, prv(n+1);
  iota(all(prv), 0);
  vi nxt = prv;
  vector<ll> dist(n, INT64_MAX);
  par.assign(n, -1);

  auto add = [&](int v, int p, ll d) {
    par[v] = p;
    dist[v] = d;
    prv[n] = nxt[prv[v] = prv[nxt[v] = n]] = v;
  }; // aeb1

  auto del = [&](int v) {
    nxt[prv[nxt[v]] = prv[v]] = nxt[v];
    prv[v] = nxt[v] = v;
  }; // df30

  for (add(src, -2, 0); nxt[n] != n;) {
    int v = nxt[n];
    del(v);
    each(e, G[v]) {
      ll alt = dist[v] + e.y;
      if (alt < dist[e.x]) {
        que = {e.x};
        rep(i, 0, sz(que)) {
          int w = que[i];
          par[w] = -1;
          del(w);
          each(f, G[w])
            if (par[f.x] == w) que.pb(f.x);
        } // c58d
        if (par[v] == -1) return {};
        add(e.x, v, alt);
      } // fd17
    } // 0e38
  } // b029

  return dist;
} // 0f19
```

graphs/strongly_connected.h 72ba

```
// 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);
  } // f0fa

  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]);
      } // 8ed0
      S.resize(S[v]);
    } // ecc7

    return low;
  } // f3c6
}; // a7ca
```

math/berlekamp_massey.h 7d12

```
constexpr int MOD = 998244353;

ll modInv(ll a, ll m) { // a^(-1) mod m
  if (a == 1) return 1;
  return ((a - modInv(m%a, a))*m + 1) / a;
} // c437

// 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
vector<ll> massey(vector<ll>& A) {
  if (A.empty()) return {};
  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;
      } // c350
    } // 79c7
  } // f70c

  C.resize(len+1);
  C.erase(C.begin());
  each(x, C) x = (MOD - x) % MOD;
  return C;
} // 20ce
```

math/bit_gauss.h 4b1a

```
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;
    } // a6bb

    rep(k, i+1, sz(A)) if (A[k][c]) A[k]^=A[i];
    swap(A[i], A[sz(col)]);
    col.pb(c);
  } // a953

  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();
  } // 4ca1

  return sz(col) < m ? 2 : 1;
} // 986e
```

math/bit_matrix.h 2e3f

```
using ull = uint64_t;

// Matrix over Z_2 (bits and xor)
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);
  } // 7ef0

  // 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;
  } // 28bd

  // 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;
  } // 98a8
}; // 4df7
```

math/continued_fractions.h 883b

```
// for N ~ 1e7; long double for N ~ 1e9
using dbl = double;

// Given N and a real number x >= 0, finds the
// closest rational approximation p/q with
// p, q < N. It will obey |p/q - x| < 1/qN.
// For consecutive convergents,
// p_{k+1}q_k - q_{k+1}p_k = (-1)^k.
// (p_k/q_k alternates between >x and <x.)
// If x is rational, y eventually becomes inf;
// if x is the root of a degree 2 polynomial
// the a's eventually become cyclic; O(lg n)
pair<ll, ll> approximate(dbl x, ll N) {
  ll LP=0, LQ=1, P=1, Q=0, inf = LLONG_MAX;
  for (dbl y = x;;) {
    ll lim = min(P ? (N-LP) / P : inf,
                 Q ? (N-LQ) / Q : inf),
       a = (ll)floor(y), b = min(a, lim),
       NP = b*P + LP, NQ = b*Q + LQ;
    if (a > b) {
      // If b > a/2, we have a semi-convergent
      // that gives us a better approximation;
      // if b = a/2, we *may* have one.
      // Return {P, Q} here for a more
      // canonical approximation.
      return (abs(x - (dbl)NP / (dbl)NQ)
        < abs(x - (dbl)P / (dbl)Q)) ?
        make_pair(NP, NQ) : make_pair(P, Q);
    } // 1ca8
    if (abs(y = 1/(y - (dbl)a)) > 3*N)
      return {NP, NQ};
    LP = P; P = NP;
    LQ = Q; Q = NQ;
  } // 2fd0
} // f41e
```

math/crt.h 4e5f

```
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;
} // 23c8

// Chinese Remainder Theorem; 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};
} // 35a8
```

math/fast_mod.h d65b

```
using ull = uint64_t;

// Compute a % b faster, where b is constant,
// but not known at compile time.
// Returns value in range [0,2b).
struct FastMod {
  ull b, m;
  FastMod(ull a) : b(a), m(-1ULL / a) {}
  ull operator()(ull a) { // a % b + (0 or b)
    return a - ull((__uint128_t(m)*a)>>64) * b;
  } // f27d
}; // 09d4
```

math/fft_complex.h 0d46

```
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};
} // 3b78
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));
  } // 92a9

  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;
    } // b389
  } // 84bf
} // 9dc8

// 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 = sz(a) + sz(b) - 1;
  if (len <= 0) return a.clear();
  int n = 2 << __lg(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);
} // 1796

// 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 = 2 << __lg(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;
} // 41bb

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 = 2 << __lg(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);
  } // e877

  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);
  } // e75d

  return ret;
} // df22
```

math/fft_mod.h 7f8c

```
// 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

// int128: M = (1LL<<32)*2147483641
//         R = 2379743102665616301LL

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;
  } // 66ca
  return t;
} // 1973

// 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;
  } // 0d98

  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;
    } // e4a6
  } // 8d38
} // 01f5

// 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 = sz(a) + sz(b) - 1;
  if (len <= 0) return a.clear();
  int n = 2 << __lg(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);
} // 24fe

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;
} // 23c8

// 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;

  auto 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;
  } // 2279
} // c493

// Big-integer multiplication note:
// - use convLong with base 10^6 for n < 10^6
// - use convLong with base 10^5 for n < 10^8
```

math/fft_online.h 637b

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

// Online convolution helper. Ensures that:
// out[m] = sum { f(i)*g(m-i) : 1 <= i <= m-1 }
// See usage example below.
void onlineConv(vector<Zp>& out, int m,
                auto f, auto g) {
  int len = m & ~(m-1), b = m-len;
  int e = min(m+len, sz(out));
  auto apply = [&](auto r, auto s) {
    vector<ll> P(m-b+1), Q(min(e-b, m));
    rep(i, max(b, 1), m) P[i-b] = r(i).x;
    rep(i, 1, sz(Q)) Q[i] = s(i).x;
    convolve(P, Q);
    rep(i, m, e) out[i] += P[i-b];
  }; // d14b
  apply(f, g);
  if (b) apply(g, f);
} // b6d6

// h[m] = 1 + sum h(i)*i * h(m-i)/(m-i)
void example(int n) {
  vector<Zp> h(n);
  for (int m = 1; m < n; m++) {
    onlineConv(h, m,
      [&](int i) { return h[i] * i; },
      [&](int i) { return h[i] / i; });
    h[m] += 1;
  } // 11ee
} // 369c
```

math/fwht.h a4d3

```
// 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.
template<bool inv>
void fwht(auto& 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) = make_pair(x+y, x-y); // XOR
        x += inv ? -y : y;               // AND
        y += inv ? -x : x;               // OR
      } // ceb0
    } // f260
  } // b094
  if (inv) each(e, b) e /= sz(b); // ONLY XOR
} // 45b6

// 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!
void bitConv(auto& a, auto b) {
  fwht<0>(a);
  fwht<0>(b);
  rep(i, 0, sz(a)) a[i] *= b[i];
  fwht<1>(a);
} // 7b82
```

math/gauss.h 8469

```
constexpr double eps = 1e-9;

// 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 && fabs(A[i][c]) < eps) c++;
    // For Zp:
    //while (c <= m && !A[i][c].x) c++;

    if (c >= m) {
      if (c == m) return 0;
      continue;
    } // a6bb

    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;
    } // 8dd5

    swap(A[i], A[sz(col)]);
    col.pb(c);
  } // 470a

  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]];
  } // 31b9

  return sz(col) < m ? 2 : 1;
} // fcf5
```

math/gauss_ortho.h 754f

```
using Row = vector<double>;
using Matrix = vector<Row>;
constexpr double eps = 1e-9;

// Given a system of n linear equations A
// over m variables, find dimensionality D
// of solution subspace, matrix M and vector t
// such that:
// - matrix M is orthogonal (i.e. M*M^T = I)
// - x is a solution <=> (Mx+t)[D..] = 0
// - x[D..] = 0 <=> M^T(x-t) is a solution
//   (in particular -M^T*t is a solution)
// Returns number of dimensions D, or -1 if
// there is no solution; time: O(n^2*m + n*m^2)
// Warning: numerical stability is kinda sus
int orthoGauss(Matrix& A, Matrix& M,
               Row& t, int m) {
  int d = m;
  t.assign(m, 0);
  M.assign(m, Row(m));
  rep(i, 0, m) M[i][i] = 1;

  rep(i, 0, sz(A)) {
    auto& w = A[i];
    double s = 0;
    rep(j, 0, d) s += w[j]*w[j];
    if (fabs(s) < eps) {
      if (fabs(w[m]) > eps) return -1;
      continue;
    } // e92f

    double r = sqrt(s);
    if (w[d-1] < 0) r = -r;
    s = sqrt((s + w[d-1]*r)*2);
    w[d-1] += r;
    rep(j, 0, d) w[j] /= s;
    r = w[m] / (w[d-1] * r * 2);
    rep(j, i+1, sz(A)) {
```

```
      s = 0;
      rep(k, 0, d) s += A[j][k] * w[k];
      s *= 2;
      rep(k, 0, d) A[j][k] -= s * w[k];
      A[j][m] -= s*r;
    } // 69fe

    rep(j, 0, m) {
      s = 0;
      rep(k, 0, d) s += M[k][j] * w[k];
      s *= 2;
      rep(k, 0, d) M[k][j] -= s * w[k];
    } // 692b

    s = -r;
    rep(k, 0, d) s += t[k] * w[k];
    s *= 2;
    rep(k, 0, d) t[k] -= s * w[k];
    d--;
  } // 789e

  return d;
} // a6c9
```

math/linear_rec.h 60be

```
constexpr ll MOD = 998244353;

using Poly = vector<ll>;

// Compute k-th term of an n-order linear
// recurrence C[i] = sum C[i-j-1]*D[j],
// given C[0..n-1] and D[0..n-1]; O(n^2 log k)
ll linearRec(const Poly& C,
             const Poly& D, ll k) {
  int n = sz(D);

  auto mul = [&](Poly a, Poly b) {
    Poly ret(n*2+1);
    rep(i, 0, n+1) rep(j, 0, n+1)
      ret[i+j] = (ret[i+j] + a[i]*b[j]) % MOD;
    for (int i = n*2; i > n; i--) rep(j, 0, n)
      ret[i-j-1] =
        (ret[i-j-1] + ret[i]*D[j]) % MOD;
    ret.resize(n+1);
    return ret;
  }; // e722

  Poly pol(n+1), e(n+1);
  pol[0] = e[1] = 1;

  for (k++; k; k /= 2) {
    if (k % 2) pol = mul(pol, e);
    e = mul(e, e);
  } // 13af

  ll ret = 0;
  rep(i,0,n) ret = (ret + pol[i+1]*C[i]) % MOD;
  return ret;
} // 3fd1
```

math/linear_rec_fast.h 58a8

```
#include "polynomial.h"

// Compute k-th term of an n-order linear
// recurrence C[i] = sum C[i-j-1]*D[j],
// given C[0..n-1] and D[0..n-1];
// time: O(n log n log k)
Zp linearRec(const Poly& C,
             const Poly& D, ll k) {
  Poly f(sz(D)+1, 1);
  rep(i, 0, sz(D)) f[i] = -D[sz(D)-i-1];
  f = pow({0, 1}, k, f);
  Zp ret = 0;
  rep(i, 0, sz(f)) ret += f[i]*C[i];
  return ret;
} // 5b8d
```

math/matrix.h 9bf7

```
#include "modular.h"

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;
} // ad1d

// 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;
} // b6bf
Matrix operator+(Matrix l, const Matrix& r) {
  return l += r;
} // d9b3

// 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;
} // 90a1
Matrix operator-(Matrix l, const Matrix& r) {
  return l -= r;
} // dc4f

// 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;
} // 52ca
Matrix& operator*=(Matrix& l, const Matrix& r){
  return l = l*r;
} // da8a

// 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;
  } // 4400
  return t;
} // 65ea

// 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;
} // 5650

// 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;
    } // f98a
    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;
      } // f519
      rank++;
    } // 4cd8
  } // 36e9
  return rank;
} // 6882

// Compute matrix rank; time: O(n^3)
#define rank rank_
int rank(Matrix A) {
  int s; return echelon(A, s);
} // c599

// 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;
} // b252

// 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];
  } // 1e97
  each(r, A) r.erase(r.begin(), r.begin()+n);
  return 1;
} // 65b9
```

math/miller_rabin.h 2d52

```
#include "modular64.h"

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

// Deterministic for p <= 1'050'535'501
// constexpr ll BASES[] = {
//   336'781'006'125, 9'639'812'373'923'155
// }; // d41d

// Deterministic for p <= 2^64
constexpr ll BASES[] = {
  2, 325, 9'375, 28'178,
  450'775, 9'780'504, 1'795'265'022
}; // 0e1d

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;
  } // 9342

  return 1;
} // bec2
```

math/modinv_precompute.h 2427

```
constexpr ll MOD = 234567899;

// Precompute modular inverses; time: O(n)
auto modInv = [] {
  vector<ll> v(MOD, 1); // You can lower size
  rep(i, 2, sz(v))
    v[i] = (MOD - (MOD/i) * v[MOD%i]) % MOD;
  return v;
}(); // 7806
```

math/modular.h 72a7

```
// 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;
} // c437

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;
  } // 66ca
  return t;
} // 1973

// 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; } // e986

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

  // 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()
}; // f730

// 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;
} // 23c8
```

math/modular64.h 4b73

```
// Modular arithmetic for modulus < 2^62
ll modAdd(ll x, ll y, ll m) {
  x += y;
  return x < m ? x : x-m;
} // b653
ll modSub(ll x, ll y, ll m) {
  x -= y;
  return x >= 0 ? x : x+m;
} // b073
// 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;
} // 1815
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);
  } // bd61
  return t;
} // c8ba
```

math/modular_generator.h f203

```
#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);
    } // 6069
  } // a0cc
  if (n > 1) fac.pb(n);
  return fac;
} // 4a2a
// 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).
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:;
  } // db24
} // 55e6
```

math/modular_log.h ac62

```
#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;
  } // f1d0
  t = modInv(t, p);
  rep(i, 0, m) {
    int j = small[b];
    if (j) return i*ll(m) + j - 1;
    b = b*t % p;
  } // c7ed
  return -1;
} // 5c26
```

math/modular_sqrt.h db16

```
#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).
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;
  } // f83f
} // 7a97
```

math/nimber.h d22e

```
// Arithmetic over 64-bit nimber field.
// 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 2^2^k
// form a field of characteristic 2.
// Addition is equivalent to bitwise xor.
using ull = uint64_t;
uint16_t npw[1<<16], nlg[1<<16];
// Multiply 64-bit nimbers a and b.
template<int half = 32, bool prec = 0>
ull nimMul(ull a, ull b) {
  if (a < 2 || b < 2) return a * b;
  if (!prec && half <= 8)
    return npw[(nlg[a] + nlg[b]) % 0xFFFF];
  constexpr ull tot = 1ull << half;
  ull c = a % tot, d = a >> half;
  ull e = b % tot, f = b >> half;
  ull p = nimMul<half/2, prec>(c, e);
  ull r = nimMul<half/2, prec>(d, f);
  ull s = nimMul<half/2, prec>(c^d, e^f);
  ull t = nimMul<half/2, prec>(r, tot/2);
  return p ^ t ^ (p ^ s) << half;
} // df59
int dummy = ([]() {
  rep(i, npw[0] = 1, 0xFFFF) {
    ull v = nimMul<16, 1>(npw[i-1], -1);
    nlg[npw[i] = uint16_t(v)] = uint16_t(i);
  } // 43d9
}(), 0);
// Compute a^e under nim arithmetic;
// O(lg M) nimber multiplications
ull nimPow(ull a, ull e) {
  ull t = 1;
  while (e) {
    if (e % 2) t = nimMul(t, a);
    e /= 2; a = nimMul(a, a);
  } // da53
  return t;
} // c06c
// Compute inverse of a in 2^64 nim-field;
// O(lg M) nimber multiplications
ull nimInv(ull a) {
  return nimPow(a, -2);
} // 4d01
```

math/phi_large.h 8703

```
#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;
} // 798e
```

math/phi_precompute.h 544a

```
constexpr int MAX_PHI = 1e7;
// Precompute Euler's totients; time: O(n lg n)
vi phi = [] {
  vi p(MAX_PHI+1);
  iota(all(p), 0);
  rep(i, 2, sz(p)) if (p[i] == i)
    for (int j = i; j < sz(p); j += i)
      p[j] = p[j] / i * (i-1);
  return p;
}(); // d94b
```

math/phi_prefix_sum.h 89f6

```
#include "phi_precompute.h"
constexpr int MOD = 998244353;
// Precompute Euler's totient prefix sums
// for small values; time: O(n lg n)
// phiSum[k] = sum from 0 to k-1
auto phiSum = [] {
  vector<ll> s(sz(phi)+1);
  rep(i, 0, sz(phi))
    s[i+1] = (s[i] + phi[i]) % MOD;
  return s;
}(); // b078
// Get prefix sum of phi(0) + ... + phi(n-1).
// For MOD > 4*10^9, answer will overflow.
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%2 ? n%MOD * ((n+1)/2 % MOD)
                : n/2%MOD * (n%MOD+1)) % MOD;
  for (ll s, i = 2; i <= n; i = s+1) {
    s = n / (n/i);
    ret -= (s-i+1)%MOD*getPhiSum(n/i+1) % MOD;
  } // fa0b
  return big[n] = ret = (ret%MOD + MOD) % MOD;
} // 1d5f
```

math/pi_large.h c04b

```
// Precompute prime counting function
// for small values; time: O(n lg lg n)
vector<ll> prl, pis = [] {
  constexpr int MAX_P = 1e7;
  vector<ll> p(MAX_P+1, 1);
  p[0] = p[1] = 0;
  for (int i = 2; i*i <= MAX_P; i++) if (p[i])
    for (int j = i*i; j <= MAX_P; j += i)
      p[j] = 0;
  rep(i, 1, sz(p)) {
    if (p[i]) prl.pb(i);
    p[i] += p[i-1];
  } // d28e
  return p;
}(); // f6a5
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;
} // 774f
// Count number of primes <= x;
// time: O(n^(2/3) * log(n)^(1/3))
// Set MAX_P to be > sqrt(x) 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;
```

```
} // a584

ret += partial(x, a) + (b+a-1)*(b-a)/2;
return big[x] = ret;
} // ea1d
```

math/pi_large_precomp.h e93e

```
#include "sieve.h"

// Count primes in given interval
// using precomputed table.
// Set MAX_P to sqrt(MAX_N)!
// Precomputed table will contain N_BUCKETS
// elements - check source size limit.
// If you need to pack more values,
// you can use `utils/packing.h`.

// Precomputed table size:
//   MAX_N=1e11, N_BUCKETS=1e4 -> 43.96 KB
//   MAX_N=1e11, N_BUCKETS=2e4 -> 85.55 KB

constexpr ll MAX_N = 1e11;
constexpr ll N_BUCKETS = 2e4;
constexpr ll BUCKET_SIZE = (MAX_N/N_BUCKETS)+1;
constexpr ll precomputed[] = {/* ... */};

// Unpack precomputed data.
// Warning: comment out during precomputing.
vector<ll> buckets = [] {
  vector<ll> ret(N_BUCKETS+1);
  ll d = 0;
  rep(i, 0, N_BUCKETS)
    ret[i+1] = ret[i] + (d += precomputed[i]);
  return ret;
}(); // db13

// Count primes in range [b;e) naively.
ll sieveRange(ll b, ll e) {
  bitset<BUCKET_SIZE> elems;
  b = max(b, 2LL);
  e = max(b, e);
  each(p, primesList) {
    ll c = max((b+p-1) / p, 2LL);
    for (ll i = c*p; i < e; i += p)
      elems.set(i-b);
  } // 9f7f
  return e-b-elems.count();
} // f028

// Run once on local computer to precompute
// table. Takes about 10 minutes for n = 1e11.
// First and last values for default params:
// 348513, -32447, -9941, -6221, -4585,
// ..., -162, -162, 563, -286, -949
void localPrecompute() {
  ll last = 0;
  for (ll i = 0; i <= MAX_N; i += BUCKET_SIZE){
    ll to = min(i+BUCKET_SIZE, MAX_N+1);
    ll cur = sieveRange(i, to);
    cout << cur-last << ',' << flush;
    last = cur;
  } // 93f9
  cout << endl;
} // e009

// Count number of primes <= x;
// time: O(BUCKET_SIZE*lg lg n + sqrt(n)/lg(n))
ll pi(ll x) {
  ll b = x/BUCKET_SIZE, j = b*BUCKET_SIZE;
  return buckets[b] + sieveRange(j, x+1);
} // 8582
```

math/pollard_rho.h 1d22

```
#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);
    } // 20e5
    if (d != n) return rho(out,d),rho(out,n/d);
  } // 423f
} // 0c30

// 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++;
  } // 2ab1
  return ret;
} // 471c
```

math/polynomial.h 6e75

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

using Poly = vector<Zp>;

// Cut off trailing zeroes; time: O(n)
void norm(Poly& P) {
  while (!P.empty() && !P.back().x)
    P.pop_back();
} // 8a8a

// 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;
} // d865

// 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;
} // 656e
Poly operator+(Poly l, const Poly& r) {
  return l += r;
} // d9b3

// 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;
} // c68b
Poly operator-(Poly l, const Poly& r) {
  return l -= r;
} // dc4f

// 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
    // Choose appropriate convolution method,
    // see fft_mod.h and fft_complex.h
    using v = vector<ll>;
    convolve<MOD, 62>(*(v*)&l, *(const v*)&r);
  } // 30c9
  norm(l);
  return l;
} // e8b3
Poly operator*(Poly l, const Poly& r) {
  return l *= r;
} // 2de3

// Compute inverse series mod x^n; O(n lg n)
// Requires P(0) != 0.
Poly invert(const Poly& P, int n) {
  assert(!P.empty() && P[0].x);
  Poly tmp{P[0]}, ret = {P[0].inv()};
  for (int i = 1; i < n; i *= 2) {
    rep(j, i, min(i*2, sz(P))) tmp.pb(P[j]);
    (ret *= Poly{2} - tmp*ret).resize(i*2);
  } // 904e
  ret.resize(n);
  return ret;
} // 9293

// 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 l.clear(), l;
  reverse(all(l));
  reverse(all(r));
  l.resize(d);
  l *= invert(r, d);
  l.resize(d);
  reverse(all(l));
  return l;
} // cf5e
Poly operator/(Poly l, const Poly& r) {
  return l /= r;
} // 152d

// Remainder modulo a polynomial; O(n lg n)
Poly operator%(const Poly& l, const Poly& r) {
  return l - r*(l/r);
} // 4fc8
Poly& operator%=(Poly& l, const Poly& r) {
  return l -= r*(l/r);
} // 80bb

// Compute a^e mod x^n, where a is polynomial;
// time: O(n log n log e)
Poly pow(Poly a, ll e, int n) {
  Poly t = {1};
  while (e) {
    if (e % 2) (t *= a).resize(n);
    e /= 2; (a *= a).resize(n);
  } // d0c6
  norm(t);
  return t;
} // ada1

// Compute a^e mod m, where a and m are
// polynomials; time: O(|m| log |m| log e)
Poly pow(Poly a, ll e, const Poly& m) {
  Poly t = {1};
  while (e) {
    if (e % 2) t = t*a % m;
    e /= 2; a = a*a % m;
  } // 66ca
  return t;
} // 6f9c

// 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();
  } // bd78
  return P;
} // c6c5

// 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;
  } // eec1
  return P;
} // e2f3

// Compute ln(P) mod x^n; time: O(n log n)
Poly log(const Poly& P, int n) {
  Poly a = integrate(derivate(P)*invert(P,n));
  a.resize(n);
  return a;
} // 5d6b

// Compute exp(P) mod x^n; time: O(n lg n)
// Requires P(0) = 0.
Poly exp(Poly P, int n) {
  assert(P.empty() || !P[0].x);
  Poly tmp{P[0]+1}, ret = {1};
  for (int i = 1; i < n; i *= 2) {
    rep(j, i, min(i*2, sz(P))) tmp.pb(P[j]);
    (ret *= (tmp - log(ret, i*2))).resize(i*2);
  } // c28a
  ret.resize(n);
  return ret;
} // bd42

// Compute sqrt(P) mod x^n; time: O(n log n)
#include "modular_sqrt.h"
bool sqrt(Poly& P, int n) {
  norm(P);
  if (P.empty()) return P.resize(n), 1;

  int tail = 0;
  while (!P[tail].x) tail++;
  if (tail % 2) return 0;

  ll sq = modSqrt(P[tail].x, MOD);
  if (sq == -1) return 0;

  Poly tmp{P[tail]}, ret = {sq};
```

```
  for (int i = 1; i < n - tail/2; i *= 2) {
    rep(j, i, min(i*2, sz(P)-tail))
      tmp.pb(P[tail+j]);
    (ret += tmp * invert(ret,i*2)).resize(i*2);
    each(e, ret) e /= 2;
  } // 2d41

  P.resize(tail/2);
  P.insert(P.end(), all(ret));
  P.resize(n);
  return 1;
} // b9b3

// Compute polynomial P(x+c); time: O(n lg n)
Poly shift(Poly P, Zp c) {
  int n = sz(P);
  Poly Q(n, 1);
  Zp fac = 1;
  rep(i, 1, n) {
    P[i] *= (fac *= i);
    Q[n-i-1] = Q[n-i] * c / i;
  } // 1c20
  P *= Q;
  if (sz(P) < n) return {};
  P.erase(P.begin(), P.begin()+n-1);
  fac = 1;
  rep(i, 1, n) P[i] /= (fac *= i);
  return P;
} // b11f

// Compute values P(x^0), ..., P(x^{n-1});
// time: O(n lg n)
Poly chirpz(Poly P, Zp x, int n) {
  if (P.empty()) return Poly(n);
  if (!x.x) {
    Poly Q(n, P[0]);
    rep(i, 1, sz(P)) Q[0] += P[i];
    return Q;
  } // ab77
  int k = sz(P);
  Poly Q(n+k);
  rep(i, 0, n+k) Q[i] = x.pow(i*ll(i-1)/2);
  rep(i, 0, k) P[i] /= Q[i];
  reverse(all(P));
  P *= Q;
  P.resize(n+k);
  rep(i, 0, n) P[i] = P[k+i-1] / Q[i];
  P.resize(n);
  return P;
} // 5c3c

// 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];
  } // c1c2
  return points;
} // 69b0

// 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];
} // b706
```

math/polynomial_interp.h a4cc

```
// Interpolate set of points (i, vec[i])
// and return it evaluated at x; O(n lg MOD)
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;
  } // 681d
  return ret;
} // dd92

// Given n points (x, f(x)) compute n-1-degree
// polynomial f that passes through them;
// time: O(n^2 lg MOD)
// 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;
  } // af1c
  return ret;
} // 7c2c
```

math/sieve.h a3cc

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

// Erathostenes sieve; time: O(n lg lg n)
vi primesList = [] {
  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);

  vi ret;
  rep(i, 0, MAX_P+1) if (primes[i]) ret.pb(i);
  return ret;
}(); // c997
```

math/sieve_factors.h 3cff

```
constexpr int MAX_P = 1e6;

// Erathostenes sieve that saves smallest
// factor for each number; time: O(n lg lg n)
vi factor = [] {
  vi f(MAX_P+1);
  iota(all(f), 0);
  for (int i = 2; i*i <= MAX_P; i++)
    if (f[i] == i)
      for (int j = i*i; j <= MAX_P; j += i)
        f[j] = min(f[j], i);
  return f;
}(); // ac6f

// 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;
  } // 664c
  return ret;
} // 56cb
```

math/sieve_segmented.h 655c

```
constexpr int MAX_P = 1e9;

// Cache-friendly Erathostenes sieve
// ~1.5s on Intel Core i5 for MAX_P = 10^9
// Memory usage: MAX_P/16 bytes
// The bitset stores only odd numbers.
auto primes = [] {
  constexpr int SEG = 1<<18;
  int j, sq = int(sqrt(MAX_P))+1;
  vector<pii> dels;
  bitset<MAX_P/2+1> ret;
  ret.set();
  ret.reset(0);

  for (int i = 3; i <= sq; i += 2) {
    if (ret[i/2]) {
      for (j = i*i; j <= sq; j += i*2)
        ret.reset(j/2);
      dels.pb({ i, j/2 });
    } // d26d
  } // d26d

  for (int i = sq/2; i <= sz(ret); i += SEG) {
    j = min(i+SEG, sz(ret));
    each(d, dels) for (; d.y < j; d.y += d.x)
      ret.reset(d.y);
  } // 6676
  return ret;
}(); // 7490

bool isPrime(int n) {
  return n == 2 || (n%2 && primes[n/2]);
} // eb6c
```

math/simplex.h ab7a

```
using dbl = double;
using Row = vector<dbl>;
using Matrix = vector<Row>;
#define mp make_pair

#define ltj(X) if (s == -1 || \
  mp(X[j], N[j]) < mp(X[s], N[s])) s = j

// Simplex algorithm; time: O(nm * pivots)
// Given m x n matrix A, vector b of length m,
// vector c of length n solves the following:
//   maximize c^T x, Ax <= b, x >= 0
// Output vector `x` contains optimal solution
// or some feasible solution in unbounded case.
// Returns objective value if bounded,
// +inf if unbounded, and -inf if no solution.
// You can test if double is inf using `isinf`.
// PARTIALLY TESTED
dbl simplex(const Matrix& A,
            const Row& b, const Row& c,
            Row& x, dbl eps = 1e-8) {
  int m = sz(b), n = sz(c);
  x.assign(n, 0);
  if (!n) return
    *min_element(all(b)) < -eps ? -1/.0 : 0;

  vi N(n+1), B(m);
  Matrix D(m+2, Row(n+2));

  auto pivot = [&](int r, int s) {
    dbl inv = 1 / D[r][s];
    rep(i, 0, m+2)
      if (i != r && abs(D[i][s]) > eps) {
        dbl tmp = D[i][s] * inv;
        rep(j,0,n+2) D[i][j] -= D[r][j] * tmp;
        D[i][s] = D[r][s] * tmp;
      } // 5281
    each(k, D[r]) k *= inv;
    each(k, D) k[s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }; // f56b

  auto solve = [&](int phase) {
    for (int y = m+phase-1;;) {
      int s = -1, r = -1;
      rep(j, 0, n+1)
        if (N[j] != -phase) ltj(D[y]);
      if (D[y][s] >= -eps) return 1;
      rep(i, 0, m)
        if (D[i][s] > eps && (r == -1 ||
          mp(D[i][n+1] / D[i][s], B[i]) <
          mp(D[r][n+1] / D[r][s], B[r]))) r=i;
      if (r == -1) return 0;
      pivot(r, s);
    } // 3bef
  }; // 614a
```

```
  rep(i, 0, m) {
    copy(all(A[i]), D[i].begin());
    B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];
  } // b705
  rep(j, 0, n) D[m][N[j] = j] = -c[j];
  N[n] = -1; D[m+1][n] = 1;

  int r = 0;
  rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
  if (D[r][n+1] < -eps) {
    pivot(r, n);
    if (!solve(2) || D[m+1][n+1] < -eps)
      return -1/.0;
    rep(i, 0, m) if (B[i] == -1) {
      int s = 0;
      rep(j, 1, n+1) ltj(D[i]);
      pivot(i, s);
    } // 78fd
  } // b52b
  bool ok = solve(1);
  rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
  return ok ? D[m][n+1] : 1/.0;
} // a69c
```

math/subset_sum.h aa1b

```
#include "polynomial.h"

// Count number of possible subsets that sum
// to t for each t = 1, ..., n; O(n log n)
// Input elements are given by frequency array,
// i.e. counts[x] = how many times elements x
// is contained in the multiset.
// Requires counts[0] == 0.
Poly subsetSum(Poly counts, int n) {
  assert(counts[0].x == 0);
  Poly mul(n);
  rep(i, 0, n)
    mul[i] = Zp(i).inv() * (i%2 ? 1 : -1);
  counts.resize(n);
  for (int i = n-2; i > 0; i--)
    for (int j = 2; i*j < n; j++)
      counts[i*j] += mul[j] * counts[i];
  return exp(counts, n);
} // c6ac
```

math/subset_sum_mod.h e358

```
// Shift-tree with splitmix64 hashing.
struct ShiftTree {
  vector<uint64_t> H;
  int len, delta;

  // Init tree of size n = 2^d.
  ShiftTree(int n) : H(n*2), len(n), delta(0) {
    assert(n && !(n & (n-1)));
  } // 5236

  // Set a[i] := 1; time: O(log n)
  void set(int i) {
    H[i = (i+len-delta) % len + len] = 1;
    for (int d = delta; i > 1; d /= 2)
      update(i = parent(i, d%2), d%2);
  } // d5e1

  // Cyclically shift by k to the right;
  // time: O(n / 2^j), where j max s.t. 2^j | k
  void shift(int k) {
    if (k %= len) {
      delta = (delta+len+k) % len;
      int div = k & ~(k-1), d = delta / div;
      for (int t = len/div/2; t >= 1; t /= 2) {
        rep(i, t, t*2) update(i, d%2);
        d /= 2;
      } // 45ce
    } // b582
  } // 1a6d

  // Find mismatches between T[a:b) and Q[a:b);
  // time: O((|D|+1) log n)
  void diff(vi& out, const ShiftTree& T,
            int vb, int ve, int lvl = -1,
            int b = 0, int e = -1,
            int i = 1, int j = 1) {
    if (e < 0) lvl = __lg(e=len)-1;
    if (b >= ve || vb >= e || H[i] == T.H[j])
      return;
    if (e-b == 1) return out.push_back(b);

    int m = (b+e) / 2;
    int s1 = (delta >> lvl) & 1;
    int s2 = (T.delta >> lvl) & 1;

    diff(out, T, vb, ve, lvl-1, b, m,
      left(i, s1), left(j, s2));
    diff(out, T, vb, ve, lvl-1, m, e,
      right(i, s1), right(j, s2));
  } // b60c

  void update(int i, int s) {
    auto x = H[left(i, s)] +
      H[right(i, s)] * 0x9E37'79B9'7F4A'7C15;
    x = (x ^ (x>>30)) * 0xBF58'476D'1CE4'E5B9;
    x = (x ^ (x>>27)) * 0x94D0'49BB'1331'11EB;
    H[i] = x ^ (x >> 31);
  } // 3447

  int parent(int i, int s) {
    int k = i + s;
    return k&i ? k/2 : k/4;
  } // 314f

  int left(int i, int s) {
    int k = i*2, j = k - s;
    return k&j ? j : k|j;
  } // b4eb

  int right(int i, int s) {
    return i*2 + !s;
  } // e440
}; // 6f54

int bitrev(int n, int bits) {
  int ret = 0;
  rep(i, 0, bits)
    ret |= ((n >> i) & 1) << (bits-i-1);
  return ret;
} // 23d1

// Find all attainable subset sums modulo m;
// time: O(m log m)
// Input elements are given by frequency array,
// i.e. counts[x] = how many times element x
// is contained in the input multiset.
// Size of `counts` is the modulus m.
// The returned array encodes solutions,
// which can be recovered using `recover`.
// ans[x] != -1 <=> subset with sum x exists
vi subsetSumMod(const vi& counts) {
  int mod = sz(counts), len = 1, k = 0;
  while (len < mod*2) len *= 2, k++;

  vi tmp, ans(mod, -1);
  ShiftTree T(len), Q(len);

  ans[0] = 0;
  T.set(0);
  Q.set(0);
  Q.set(-mod);

  rep(i, 1, len) {
    int x = bitrev(i, k);
    if (x >= mod || !counts[x]) continue;

    Q.shift(x - Q.delta);

    rep(j, 0, counts[x]) {
      tmp.clear();
      T.diff(tmp, Q, 0, mod);
      if (tmp.empty()) break;

      each(d, tmp) if (ans[d] == -1) {
        ans[d] = x;
        T.set(d);
        Q.set(d+x);
        Q.set(d+x-mod);
      } // ce75
    } // c2d1
  } // 9204

  return ans;
} // 0aa2

vi recoverSubset(const vi& dp, int s) {
  assert(dp[s] != -1);
  vi ret;
  while (s) {
    ret.pb(dp[s]);
    s = (s - dp[s] + sz(dp)) % sz(dp);
  } // ea17
  return ret;
} // 7103
```

segtree/general_config.h 6ef4

```
// Segment tree configurations to be used
// in general_fixed and general_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)
  using T = int; // Data type for update
                 // operations (lazy tag)
  static constexpr 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;
    int cnt = 0; // And node count.

    // Initialize as leaf (single value)
    void leaf() { sum=vMax=0; nMax=cnt=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;
      cnt += r.cnt;
    } // 262d

    // Apply update provided in `x`:
    // - update aggregated data and `lazy` tag
    // - return 0 if update should branch
    //   (can be used in "segment tree beats")
    // - if you change value of `x` it will be
    //   passed to next node to the right
    //   during updates
    bool apply(T& lazy, T& x) {
      lazy += x;
      sum += x*cnt;
      vMax += x;
      return 1;
    } // 4a4e
  }; // f11d
#elif TREE_MAX // (max; max, max count)
  // time: O(lg n)
  using T = int;
  static constexpr T ID = INT_MIN;

  struct Agg {
    // Aggregated data: max value, max count
    T vMax = INT_MIN, nMax = 0, cnt = 0;
    void leaf() { vMax = 0; nMax = cnt = 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);
      cnt += r.cnt;
    } // 8561

    bool apply(T& lazy, T& x) {
      if (vMax <= x) nMax = cnt;
      lazy = max(lazy, x);
      vMax = max(vMax, x);
      return 1;
    } // 118c
  }; // 9643
#elif TREE_SET // (=; sum, max, max count)
  // time: O(lg n)
  // Set ID to some unused value.
  using T = int;
  static constexpr T ID = INT_MIN;

  struct Agg {
    // Aggregated data: sum, max, max count
    T sum = 0, vMax = INT_MIN, nMax = 0, cnt=0;
    void leaf() { sum=vMax=0; nMax=cnt=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;
      cnt += r.cnt;
    } // 262d

    bool apply(T& lazy, T& x) {
      if (x != ID) {
        lazy = x;
        sum = x*cnt;
        vMax = x;
        nMax = cnt;
      } // 0f7e
      return 1;
    } // f684
  }; // 895c
#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}.
```

```
  // If both parts are provided, addition
  // is applied first, then minimum.
  using T = pii;
  static constexpr T ID = {0, INT_MAX};

  struct Agg {
    // Aggregated data: max value, max count,
    //                  second max value, sum
    int vMax = INT_MIN, nMax = 0;
    int max2 = INT_MIN, sum = 0, cnt = 0;
    void leaf() { sum=vMax=0; nMax=cnt=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;
      } // b074
      max2 = max(max2, r.max2);
      sum += r.sum;
      cnt += r.cnt;
    } // 1a7f

    bool apply(T& lazy, T& x) {
      if (max2 != INT_MIN && max2+x.x >= x.y)
        return 0;
      lazy.x += x.x;
      sum += x.x*cnt;
      vMax += x.x;
      if (max2 != INT_MIN) max2 += x.x;
      if (x.y < vMax) {
        sum -= (vMax-x.y) * nMax;
        vMax = x.y;
      } // 7025
      lazy.y = vMax;
      return 1;
    } // 46b3
  }; // 507e
#endif
```

segtree/general_fixed.h c33c

```
// Highly configurable statically allocated
// interval-interval segment tree; space: O(n)
struct SegTree {
  // Choose/write configuration
  #include "general_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;) pull(i);
  } // 0769

  void pull(int i) {
    (agg[i] = agg[i*2]).merge(agg[i*2+1]);
  } // ebdf

  void push(int i) {
    rep(c, 0, 2)
      agg[i*2+c].apply(lazy[i*2+c], lazy[i]);
    lazy[i] = ID;
  } // e5c9

  template<bool U>
  void go(int vb, int ve, int i, int b, int e,
          auto fn) {
    if (vb < e && b < ve)
      if (b < vb || ve < e || !fn(i)) {
        int m = (b+e) / 2;
        push(i);
        go<U>(vb, ve, i*2, b, m, fn);
        go<U>(vb, ve, i*2+1, m, e, fn);
        if (U) pull(i);
      } // 5ff2
  } // 399a

  // Modify interval [b;e) with val; O(lg n)
  void update(int b, int e, T val) {
    go<1>(b, e, 1, 0, len, [&](int i) {
      return agg[i].apply(lazy[i], val);
    }); // 2828
  } // e3e4

  // Query interval [b;e); time: O(lg n)
  Agg query(int b, int e) {
    Agg t; go<0>(b, e, 1, 0, len, [&](int i) {
      return t.merge(agg[i]), 1;
    }); // c9dd
    return t;
  } // 1c6e

  // Find smallest `j` such that
  // g(aggregate of [0,j)) is true; O(lg n)
  // The predicate `g` must be monotonic.
  // Returns -1 if no such prefix exists.
  int lowerBound(auto g) {
    if (!g(agg[1])) return -1;
    Agg x, s;
    int i = 1;
    for (; i < len; g(s) || (x = s, i++))
      push(i), (s = x).merge(agg[i *= 2]);
    return i - len + !g(x);
  } // f732
}; // c6c9
```

segtree/general_persistent.h f85c

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

  vector<Agg> agg{{}}; // Aggregated data
  vector<T> lazy{ID};  // Lazy tags
  vector<bool> cow{0}; // Copy children on push
  vi L{0}, R{0};       // Children links
  int len{1};          // Number of leaves

  // Initialize tree for n elements; O(lg n)
  SegTree(int n = 0) {
    int k = 3;
    while (len < n) len *= 2, k += 3;
    rep(i, 1, k) fork(0);
    iota(all(R)-3, 3);
    L = R;
    if (n--) {
      agg[k -= 3].leaf();
      agg[k+1].leaf();
      for (int i = k-3; i >= 0; i -= 3, n /= 2)
        (n%2 ? L[i] : ++R[i])++;
      while (k--) pull(k);
    } // 13a7
  } // 4a93

  // New version from version `i`; time: O(1)
  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;
  } // a21b

  void pull(int i) {
    (agg[i] = agg[L[i]]).merge(agg[R[i]]);
  } // 359c

  void push(int i, bool w) {
    if (w || lazy[i] != ID) {
      if (cow[i]) {
        int x = fork(L[i]), y = fork(R[i]);
        L[i] = x; R[i] = y; cow[i] = 0;
      } // 82ec
      agg[L[i]].apply(lazy[L[i]], lazy[i]);
      agg[R[i]].apply(lazy[R[i]], lazy[i]);
      lazy[i] = ID;
    } // 9f41
  } // 678e

  template<bool U>
  void go(int vb, int ve, int i, int b, int e,
          auto fn) {
    if (vb < e && b < ve)
      if (b < vb || ve < e || !fn(i)) {
        int m = (b+e) / 2;
        push(i, U);
        go<U>(vb, ve, L[i], b, m, fn);
        go<U>(vb, ve, R[i], m, e, fn);
        if (U) pull(i);
      } // 3fd0
  } // 3a95

  // Modify interval [b;e) with val
  // in tree version `j`; time: O(lg n)
  void update(int j, int b, int e, T val) {
    go<1>(b, e, j, 0, len, [&](int i) {
      return agg[i].apply(lazy[i], val);
    }); // 2828
  } // 9f22

  // Query interval [b;e) in tree version `j`;
  Agg query(int j, int b, int e) { // O(lg n)
    Agg t; go<0>(b, e, j, 0, len, [&](int i) {
      return t.merge(agg[i]), 1;
    }); // c9dd
    return t;
  } // 2c98

  // Find smallest `j` such that
  // g(aggregate of [0,j)) is true
  // in tree version `i`; time: O(lg n)
  // The predicate `g` must be monotonic.
  // Returns -1 if no such prefix exists.
  int lowerBound(int i, auto g) {
    if (!g(agg[i])) return -1;
    Agg x, s;
    int p = 0, k = len;
    while (L[i]) {
      push(i, 0);
      (s = x).merge(agg[L[i]]);
      k /= 2;
      i = g(s) ? L[i] : (x = s, p += k, R[i]);
    } // 7d74
    return p + !g(x);
  } // f7d7
}; // 21f9
```

segtree/point_fixed.h 14b6

```
// Point-interval segment tree
// - T - stored data type
// - ID - neutral element for query operation
// - f(a, b) - associative aggregate function
struct SegTree {
  using T = int;
  static constexpr T ID = INT_MIN;
  T f(T a, T b) { return max(a, b); }

  vector<T> V;
  int len = 1;

  // Initialize tree for n elements; time: O(n)
  SegTree(int n = 0, T def = 0) {
    while (len < n) len *= 2;
    V.resize(len+n, def);
    V.resize(len*2, ID);
    for (int i = len; --i;)
      V[i] = f(V[i*2], V[i*2+1]);
  } // ac47

  // Set element `i` to `val`; time: O(lg n)
  void set(int i, T val) {
    V[i += len] = val;
    while (i /= 2) V[i] = f(V[i*2], V[i*2+1]);
  } // 4bcd

  // Query interval [b;e); time: O(lg n)
  T query(int b, int e) {
    T x = ID, y = ID;
    b += len;
    for (e += len; b < e; b /= 2, e /= 2) {
      if (b % 2) x = f(x, V[b++]);
      if (e % 2) y = f(V[--e], y);
    } // 4ed0
    return f(x, y);
  } // 7816

  // Find smallest `j` such that
  // g(aggregate of [0,j)) is true; O(lg n)
  // The predicate `g` must be monotonic.
  // Returns -1 if no such prefix exists.
  int lowerBound(auto g) {
    if (!g(V[1])) return -1;
    T s, x = ID;
    int j = 1;
    while (j < len)
      if (!g(s = f(x, V[j *= 2]))) x = s, j++;
    return j - len + !g(x);
  } // 6cc5
}; // a0c7
```

segtree/point_persistent.h 4113

```
// Point-interval persistent segment tree
// - T - stored data type
// - ID - neutral element for query operation
// - f(a, b) - associative aggregate function
// First tree version number is 0.
struct SegTree {
  using T = int;
  static constexpr T ID = INT_MIN;
  T f(T a, T b) { return max(a, b); }

  vector<T> agg{ID};   // Aggregated data
  vector<bool> cow{1}; // Copy children on push
  vi L{0}, R{0};       // Children links
  int len{1};          // Number of leaves
```

```
  // Initialize tree for n elements; O(lg n)
  SegTree(int n = 0, T def = 0) {
    int k = 3;
    while (len < n) len *= 2, k += 3;
    rep(i, 1, k) fork(0);
    iota(all(R)-3, 3);
    L = R;
    if (n--) {
      k -= 3;
      agg[k] = agg[k+1] = def;
      for (int i = k-3; i >= 0; i -= 3, n /= 2)
        (n%2 ? L[i] : ++R[i])++;
      while (k--)
        agg[k] = f(agg[L[k]], agg[R[k]]);
    } // 6fde
  } // 3cfb

  // New version from version `i`; time: O(1)
  int fork(int i) {
    L.pb(L[i]); R.pb(R[i]);
    agg.pb(agg[i]); cow.pb(cow[i] = 1);
    return sz(L)-1;
  } // bb75

  // Set element `pos` to `val` in version `i`;
  // time: O(lg n)
  void set(int i, int pos, T val,
           int b = 0, int e = 0) {
    if (L[i]) {
      if (!e) e = len;
      if (cow[i]) {
        int x = fork(L[i]), y = fork(R[i]);
        L[i] = x; R[i] = y; cow[i] = 0;
      } // 82ec
      int m = (b+e) / 2;
      if (pos < m) set(L[i], pos, val, b, m);
      else set(R[i], pos, val, m, e);
      agg[i] = f(agg[L[i]], agg[R[i]]);
    } else {
      agg[i] = val;
    } // 23c8
  } // 7a55

  // Query interval [b;e) in tree version `i`;
  // time: O(lg n)
  T query(int i, int vb, int ve,
          int b = 0, int e = 0) {
    if (!e) e = len;
    if (vb >= e || b >= ve) return ID;
    if (b >= vb && e <= ve) return agg[i];
    int m = (b+e) / 2;
    return f(query(L[i], vb, ve, b, m),
             query(R[i], vb, ve, m, e));
  } // 2664

  // Find smallest `j` such that
  // g(aggregate of [0,j)) is true
  // in tree version `i`; time: O(lg n)
  // The predicate `g` must be monotonic.
  // Returns -1 if no such prefix exists.
  int lowerBound(int i, auto g) {
    if (!g(agg[i])) return -1;
    T x = ID;
    int p = 0, k = len;
    while (L[i]) {
      T s = f(x, agg[L[i]]);
      k /= 2;
      i = g(s) ? L[i] : (x = s, p += k, R[i]);
    } // 0fba
    return p + !g(x);
  } // 1a9a
}; // e4ec
```

structures/bitset_plus.h 6737

```
// 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; } // f164
OP(&)OP(|)OP(^)
```

structures/fenwick_tree.h ec21

```
// Fenwick tree (BIT tree); space: O(n)
// Default version: prefix sums
struct Fenwick {
  using T = ll;
  static constexpr 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);
  } // a047

  // 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;
  } // 9810

  // 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;
      } // de7f
    } // 929c
    return i+2;
  } // 4be9
}; // eb2e
```

structures/fenwick_tree_2d.h 9f31

```
// 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);
  } // d46b

  // 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;
  } // 08cf
}; // e570
```

structures/find_union.h f9a4

```
// 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]);
  } // 5bc1

  // 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;
  } // c721
}; // 3839
```

structures/find_union_undo.h 399f

```
// Disjoint set data structure
// with rollback; space: O(n)
// Operations work in O(log(n)) time.
struct RollbackFAU {
  vi G;
  vector<pii> his;

  RollbackFAU(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 : find(G[i]);
  } // e478

  // Current time (for rollbacks)
  int time() { return sz(his); }

  // Rollback all operations after time `t`
  void rollback(int t) {
    for (int i = time(); t < i--;)
      G[his[i].x] = his[i].y;
    his.resize(t);
  } // 3ef3

  // 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);
    his.pb({i, G[i]});
    his.pb({j, G[j]});
    G[i] += G[j]; G[j] = i;
    return 1;
  } // 1491
}; // 18ef
```

structures/hull_offline.h ed05

```
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));
    } // f27f
  }; // 10dc

  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 (1) {
      T e = S.back().end=S.back().intersect(l);
      if (sz(S) < 2 || S[sz(S)-2].end < e)
        break;
      S.pop_back();
    } // 044f
    S.pb(l);
  } // 3022

  // 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;
      }); // de77
    return t.a*x + t.b;
  } // b8de
}; // fa73
```

structures/hull_online.h 6884

```
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));
    } // f27f

    bool operator<(const Line& r) const {
      return modeQ ? end < r.end : a < r.a;
    } // cfab
  }; // 6046

  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;
  } // 63b8

  // 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));
  } // 4f69

  // 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;
  } // 7533
}; // 037e

bool Hull::modeQ = 0;
```

structures/intset.h 865f

```
// Bitset with fast predecessor and successor
// queries. Can handle 50-200mln operations
// per second. Assumes X86 shift overflows.
template<int N>
struct IntSet {
  uint64_t V[N/64+1] = {};
  IntSet<(N < 65 ? 0 : N/64+1)> up;

  // Is `i` contained in the set?
  bool has(int i) const {
    return (V[i/64] >> i) & 1;
  } // abab

  // Add `i` to the set.
  void add(int i) {
    if (!V[i/64]) up.add(i/64);
    V[i/64] |= 1ull << i;
  } // 342e

  // Delete `i` from the set.
  void del(int i) {
    if (!(V[i/64] &= ~(1ull<<i))) up.del(i/64);
  } // 256a

  // Find first element > i, or return -1.
  // `i` must be in range [0;N).
  int next(int i) {
    auto x = V[i/64] >> i;
    if (x &= ~1) return i+__builtin_ctzll(x);
    return (i = up.next(i/64)) < 0 ? i :
      i*64+__builtin_ctzll(V[i]);
  } // 8160

  // Find last element < i, or return -1.
  // `i` must be in range [0;N).
  int prev(int i) {
    auto x = V[i/64] << (63-i);
    if (x &= INT64_MAX)
      return i-__builtin_clzll(x);
    return (i = up.prev(i/64)) < 0 ? i :
      i*64+63-__builtin_clzll(V[i]);
  } // 4b0d
}; // 6ba3

template<>
struct IntSet<0> {
  void add(int) {}
  void del(int) {}
  int next(int) { return -1; }
  int prev(int) { return -1; }
}; // ace7
```

structures/li_chao_tree.h 5559

```
// Extended Li Chao tree; space: O(n)
// Let F be a family of functions,
// closed under function addition, such that
// for every f != g from the family F
// there exists x such that:
// f(z) <= g(z) for z <= x, else f(z) >= g(z)
// or
// g(z) <= f(z) for z <= x, else g(z) >= f(z).
// Typically F is family of linear functions.
// DS maintains a sequence c[0], ..., c[n-1]
// under operations max, add, query
// (see comments below for explanations).
// Configure by modifying:
// - T - type of sequence elements
// - Func - represents function from family F
// - ID_ADD - neutral element for function add
// - ID_MAX - neutral element for function max
// TESTED ON RANDS
struct LiChao {
  struct Func {
    ll a, b; // a*x + b

    // Evaluate function in point x
    ll operator()(ll x) const { return a*x+b; }

    // Sum of two functions
    Func operator+(Func r) const {
      return {a+r.a, b+r.b};
    } // f911
  }; // 633c

  static constexpr Func ID_ADD{0, 0};
  static constexpr Func ID_MAX{0, ll(-1e9)};

  vector<Func> val, lazy;
  int len;

  // Initialize tree for n elements; time: O(n)
  LiChao(int n = 0) {
    for (len = 1; len < n; len *= 2);
    val.resize(len*2, ID_MAX);
    lazy.resize(len*2, ID_ADD);
  } // c0ba

  void push(int i) {
    if (i < len) rep(j, 0, 2) {
      lazy[i*2+j] = lazy[i*2+j] + lazy[i];
      val[i*2+j] = val[i*2+j] + lazy[i];
    } // 54fc
    lazy[i] = ID_ADD;
  } // 1777

  // For each x in [vb;ve)
  // set c[x] = max(c[x], f(x));
  // time: O(log^2 n) in general case,
  //       O(log n) if [vb;ve) = [0;len)
  void max(int vb, int ve, Func f,
           int i = 1, int b = 0, int e = -1) {
    if (e < 0) e = len;
    if (vb >= e || b >= ve || i >= len*2)
      return;

    int m = (b+e) / 2;
    push(i);

    if (b >= vb && e <= ve) {
      auto& g = val[i];
      if (g(m) < f(m)) swap(g, f);
      if (g(b) < f(b))
        max(vb, ve, f, i*2, b, m);
      else
        max(vb, ve, f, i*2+1, m, e);
    } else {
      max(vb, ve, f, i*2, b, m);
      max(vb, ve, f, i*2+1, m, e);
    } // f2c0
  } // 03ed

  // For each x in [vb;ve)
  // set c[x] = c[x] + f(x);
  // time: O(log^2 n) in general case,
  //       O(1) if [vb;ve) = [0;len)
  void add(int vb, int ve, Func f,
           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) {
      lazy[i] = lazy[i] + f;
      val[i] = val[i] + f;
    } else {
      int m = (b+e) / 2;
      push(i);
      max(b, m, val[i], i*2, b, m);
      max(m, e, val[i], i*2+1, m, e);
      val[i] = ID_MAX;
      add(vb, ve, f, i*2, b, m);
      add(vb, ve, f, i*2+1, m, e);
    } // bbe5
  } // 259f

  // Get value of c[x]; time: O(log n)
  auto query(int x) {
    int i = x+len;
    auto ret = val[i](x);
    while (i /= 2)
      ret = ::max(ret+lazy[i](x), val[i](x));
    return ret;
  } // dfe4
}; // 0104
```

structures/max_queue.h 3e9e

```
// 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);
  } // 57a2

  // Pop from the front; time: O(1)
  void pop() {
    if (M.front() == Q.front()) M.pop_front();
    Q.pop_front();
  } // 101c

  // Get max element value; time: O(1)
  T max() const { return M.front(); }
}; // b6c4
```

structures/rmq.h b828

```
// 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.pb({});
      auto& prev = s[sz(s)-2];
      rep(i, 0, sz(vec)-h*2+1)
        s.back().pb(f(prev[i], prev[i+h]));
    } // 7c37
  } // 14ed

  // Query f(s[b], ... ,s[e-1]); time: O(1)
  T query(int b, int e) {
    if (b >= e) return ID;
    int k = __lg(e-b);
    return f(s[k][b], s[k][e - (1<<k)]);
  } // bb12
}; // c8f0
```

structures/treap.h 6156

```
// "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?
  }; // 3036

  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.pb({}); return sz(G)-1; }

  // Get size of node x subtree. x can be -1.
  int size(int x) { // time: O(1)
```

```
  return (x >= 0 ? G[x].size : 0);
} // 81cf

// 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
} // ed19

// 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;
    } // f7a7
  } // + any other aggregates
} // 46a3

// 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;
  } // fe19
  update(x);
} // 8211

// 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;
  } // 18c7

  G[r].E[0] = join(l, G[r].E[0]);
  update(r);
  return r;
} // b559

// 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;
  } // 054c
  return -1;
} // 0b9b

// 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;
} // be8b

// 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;
  } // 3f81
  return i;
} // ddad

// 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);
} // e418
}; // 17cc
```

structures/wavelet_tree.h 80d3

```
// 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;

  WaveletTree() {}

  // 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);
  } // a5e9

  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);
    } // ac25
    build(i*2, b, m);
    build(i*2+1, m, e);
  } // 8153

  // 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);
  } // 7861

  // 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);
  } // 71cf
}; // 49a9
```

structures/ext/hash_table.h 2d30

```
#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));
  } // 3a78
}; // 7d0e
```

structures/ext/heap.h d41d

```
#include <ext/pb_ds/priority_queue.hpp>
// Pairing heap: push O(1), pop O(lg n)
//   __gnu_pbds::priority_queue<T, Cmp>

// Standard priority_queue methods and:
// 1. Iterable
// 2. t.erase(iterator)                O(lg n)
// 3. t.modify(iterator, value)        O(lg n)
// 4. t1.join(t2) - merge t2 into t1   O(1)
```

structures/ext/rope.h 051f

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

structures/ext/tree.h a3bc

```
#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 5cc2

```
#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 fc9b

```
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({});
      } // 5ead
      i = nxt[i][c-AMIN];
    } // ace9
    accept[i].pb(id);
    return i;
  } // 27c8

  // Build automata; time: O(V*ALPHA)
  void build() {
    queue<int> que;
    each(e, nxt[0]) if (e) {
      suf[e] = 0; que.push(e);
    } // c34d
    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++];
      } // 8521
      accLink[i] = (accept[s].empty() ?
        accLink[s] : s);
    } // 1e8a
  } // 2561

  // Append `c` to state `i`
  int next(int i, char c) {
    return nxt[i][c-AMIN];
  } // 6bb7

  // 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.
  void accepted(int i, auto f) {
    while (i != -1) {
      each(a, accept[i]) if (f(a)) return;
      i = accLink[i];
    } // c175
  } // 1f0d
}; // 5c1c
```

text/alcs.h a97c

```
// All-substrings common sequences algorithm.
// Given strings A and B, algorithm computes:
//   C(i,j,k) = |LCS(A[:i), B[j:k))|
// in compressed form; time and space: O(n^2)
// To describe the compression, note that:
// 1. C(i,j,k-1) <= C(i,j,k) <= C(i,j,k-1)+1
// 2. If j < k and C(i,j,k) = C(i,j,k-1)+1,
//    then C(i,j+1,k) = C(i,j+1,k-1)+1
// 3. If j >= k, then C(i,j,k) = 0
// This allows us to store just the following:
//   ih(i,k) = min j s.t. C(i,j,k-1) < C(i,j,k)
struct ALCS {
  string A, B;
  vector<vi> ih;

  ALCS() {}

  // Precompute compressed matrix; time: O(nm)
  ALCS(string s, string t) : A(s), B(t) {
    int n = sz(A), m = sz(B);
    ih.resize(n+1, vi(m+1));
    iota(all(ih[0]), 0);
    rep(l, 1, n+1) {
      int iv = 0;
      rep(j, 1, m+1) {
        if (A[l-1] != B[j-1]) {
          ih[l][j] = max(ih[l-1][j], iv);
          iv = min(ih[l-1][j], iv);
        } else {
          ih[l][j] = iv;
          iv = ih[l-1][j];
        } // 7af8
      } // d115
    } // baff
  } // b761

  // Compute |LCS(A[:i), B[j:k))|; time: O(k-j)
  // Note: You can precompute data structure
  // to answer these queries in O(log n)
  // or compute all answers for fixed `i`.
  int operator()(int i, int j, int k) {
    int ret = 0;
    rep(q, j, k) ret += (ih[i][q+1] <= j);
    return ret;
  } // dabf

  // Compute subsequence LCS(A[:i), B[j:k));
  // time: O(k-j)
  string recover(int i, int j, int k) {
    string ret;
    while (i > 0 && j < k) {
      if (ih[i][k--] <= j) {
        ret.pb(B[k]);
        while (A[--i] != B[k]);
      } // 9d77
    } // 1edf
    reverse(all(ret));
    return ret;
  } // 738c

  // Compute LCS'es of given prefix of A,
  // and all prefixes of given suffix of B.
  // Returns vector L of length |B|+1 s.t.
  // L[k] = |LCS(A[:i), B[j:k))|; time: O(|B|)
  vi row(int i, int j) {
    vi ret(sz(B)+1);
    rep(k, j+1, sz(ret))
      ret[k] = ret[k-1] + (ih[i][k] <= j);
    return ret;
  } // 9167

  // Compute LCS'es of given prefix of A,
  // and all substrings of B; time: O(n^2)
  // Return matrix M such that:
  // M[j][k] = |LCS(A[:i), B[j:k))|
  vector<vi> matrix(int i) {
    vector<vi> ret;
    rep(j, 0, sz(B)+1) ret.pb(row(i, j));
    return ret;
  } // 15f7
}; // fd6b
```

text/hashing.h e912

```
using ull = uint64_t;

// Arithmetic mod 2^64-1.
// Around 2x slower than mod 2^64.
struct Hash {
  ull x;
  constexpr Hash(ull y = 0) : x(y) {}
  Hash operator+(Hash r) {
    return x + r.x + (x + r.x < x);
  } // b42e
  Hash operator-(Hash r) {
    return *this + ~r.x;
  } // e855
  Hash operator*(Hash r) {
    auto m = __uint128_t(x) * r.x;
    return Hash(ull(m)) + ull(m>>64);
  } // 1241
  auto get() const { return x + !~x; }
  bool operator==(Hash r) const {
    return get() == r.get();
  } // d4d5
  bool operator<(Hash r) const {
    return get() < r.get();
  } // 34a9
  void print() { cerr << x; }
}; // 6064

// Base for hashing (prime, big order).
constexpr Hash C = ll(1e11-981);

Hash powC(int n) { // C^n
  static vector<Hash> vec = {1};
  while (sz(vec) <= n) vec.pb(vec.back() * C);
  return vec[n];
} // 3ff8

// Precompute prefix hashes for a string.
struct HashInterval : vector<Hash> {
  HashInterval(auto& s) {
    pb(0);
    rep(i, 0, sz(s)) pb(at(i)*C + s[i]);
  } // 2b42
  // Get hash of interval [b;e)
  Hash operator()(int b, int e) {
    return at(e) - at(b) * powC(e-b);
  } // 4adc
}; // 6c35
```

text/kmp.h a014

```
// Computes prefsuf array; time: O(n)
// ps[i] = max prefsuf of [0;i); ps[0] := -1
vi kmp(auto& 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);
  } // 05aa
  return ps;
} // fa90

// Finds occurences of pat in vec; time: O(n)
// Returns starting indices of matches.
vi match(auto& 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);
  } // a1e9
  return ret;
} // c6e8
```

text/kmr.h 7b40

```
// 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 });
      } // a210

      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;
      } // bd2e
    } // cf37
  } // d7a7

  // Get representative of [begin;end); O(1)
  pii get(int begin, int end) {
    if (begin >= end) return {0, 0};
    int k = __lg(end-begin);
    return {ids[k][begin], ids[k][end-(1<<k)]};
  } // 6e1e

  // 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);
  } // 5d4e

  // 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;
  } // 455e
}; // 2fb3
```

text/lcp.h e309

```
// 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
vi lcpArray(auto& str, 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;
    } // 2cba
    if (k > 0) k--;
  } // 8b22
  return lcp;
} // 4202
```

text/lyndon_factorization.h 688c

```
// 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;
  } // 3f17
  return ret;
} // 0e48
```

text/main_lorentz.h 401c

```
#include "z_function.h"

struct Sqr {
  int begin, end, len;
}; // f012

// 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) {
  vector<Sqr> ans;
  vi pos(sz(s)/2+2, -1);

  rep(mid, 1, sz(s)) {
    int part = mid & ~(mid-1), off = mid-part;
    int end = min(mid+part, sz(s));
    auto a = s.substr(off, part);
    auto b = s.substr(mid, end-mid);

    string ra(a.rbegin(), a.rend());
```

```
    string rb(b.rbegin(), b.rend());

    rep(j, 0, 2) {
      // Set # to some unused character!
      vi z1 = prefPref(ra);
      vi 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+1], j);
        if (x > y) continue;

        int sb = (j ? end-y-l*2 : off+x);
        int se = (j ? end-x-l*2+1 : off+y+1);
        int& p = pos[l];

        if (p != -1 && ans[p].end == sb)
          ans[p].end = se;
        else
          p = sz(ans), ans.pb({sb, se, l});
      } // af4b

      a.swap(rb);
      b.swap(ra);
    } // 193e
  } // 4fa7

  return ans;
} // 5b80
```

text/manacher.h 4dbe

```
// Manacher algorithm; time: O(n)
// Finds largest radiuses for palindromes:
// p[0][i] for center between i-1 and i
// p[1][i] for center at i (single letter = 0)
array<vi, 2> manacher(auto& s) {
  int n = sz(s), l = 0, r = 0;
  array<vi, 2> p = {vi(n+1), vi(n)};
  rep(i, 0, n) rep(z, 0, 2) {
    int t = r-i+!z, &x = p[z][i];
    if (i < r) x = min(t, p[z][l+t]);
    int b = i-x-1, e = i+x+z;
    while (b >= 0 && e < n && s[b] == s[e])
      x++, b--, e++;
    if (r < e) l = b+1, r = e-1;
  } // 6fd0
  return p;
} // e211
```

text/min_rotation.h e4d6

```
// 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;
    } // 865b
    if (s[a+i] > s[b+i]) {
      a = b; break;
    } // 7628
  } // 40be
  return a;
} // 9ed8
```

text/monge.h e6a5

```
// NxN matrix A is simple (sub-)unit-Monge
// iff there exists a (sub-)permutation
// (N-1)x(N-1) matrix P such that:
//   A[x,y] = sum i>=x, j<y: P[i,j]
// The first column and last row are always 0.
// We represent these matrices implicitly
// using permutations p s.t. P[i,p(i)] = 1.

// (min, +) product of simple unit-Monge
// matrices represented by permutations P, Q,
// is also a simple unit-Monge matrix.
// The permutation that describes the product
// can be obtained by the following procedure:
// 1. Decompose P, Q into minimal sequences of
//    elementary transpositions.
// 2. Concatenate the transposition sequences.
// 3. Scan from left to right and remove
//    transpositions that decrease
//    inversion count (i.e. second crossings).
// 4. The reduced sequence represents result.

// Invert sub-permutation with values [0;n).
// Missing values should have value `def`.
vi invert(const vi& P, int n, int def) {
  vi ret(n, def);
  rep(i, 0, sz(P)) if (P[i] != def)
    ret[P[i]] = i;
  return ret;
} // 035e

// Split permutation P into half `lo`
// with values less than `k`, and half `hi`
// with remaining values, shifted by `k`.
// Missing rows from `lo` and `hi` are removed,
// original indices are in `loInd` and `hiInd`.
void split(const vi& P, int k, vi& lo, vi& hi,
           vi& loInd, vi& hiInd) {
  int i = 0;
  each(e, P) {
    if (e < k) lo.pb(e), loInd.pb(i++);
    else hi.pb(e-k), hiInd.pb(i++);
  } // c3a6
} // 7bb7

// Map sub-permutation into sub-permutation
// of length `n` on given indices sets.
vi expand(const vi& P, vi& ind1, vi& ind2,
          int n, int def) {
  vi ret(n, def);
  rep(k, 0, sz(P)) if (P[k] != def)
    ret[ind1[k]] = ind2[P[k]];
  return ret;
} // 0da7

// Compute (min, +) product of square
// simple unit-Monge matrices given their
// permutation representations; time: O(n lg n)
// Permutation of second matrix is inverted!
vi comb(const vi& P, const vi& invQ) {
  int n = sz(P);

  if (n < 100) {
    // 5s -> 1s speedup for ALIS for n = 10^5
    vi ret = invert(P, n, -1);
    rep(i, 0, sz(invQ)) {
      int from = invQ[i];
      rep(j, 0, i) from += invQ[j] > invQ[i];
      for (int j = from; j > i; j--)
        if (ret[j-1] < ret[j])
          swap(ret[j-1], ret[j]);
    } // 7cd1
    return invert(ret, n, -1);
  } // 679e

  vi p1, p2, q1, q2, i1, i2, j1, j2;
  split(P, n/2, p1, p2, i1, i2);
  split(invQ, n/2, q1, q2, j1, j2);

  p1 = expand(comb(p1, q1), i1, j1, n, -1);
  p2 = expand(comb(p2, q2), i2, j2, n, n);
  q1 = invert(p1, n, -1);
  q2 = invert(p2, n, n);

  vi ans(n, -1);
  int delta = 0, j = n;

  rep(i, 0, n) {
    ans[i] = (p1[i] < 0 ? p2[i] : p1[i]);
    while (j > 0 && delta >= 0)
      delta -= (q2[--j] < i || q1[j] >= i);

    if (p2[i] < j || p1[i] >= j)
      if (delta++ < 0)
        if (q2[j] < i || q1[j] >= i)
          ans[i] = j;
  } // c396

  return ans;
} // c059

// Helper function for `mongeMul`.
void padPerm(const vi& P, vi& has, vi& pad,
             vi& ind, int n) {
  vector<bool> seen(n);
  rep(i, 0, sz(P)) if (P[i] != -1) {
    ind.pb(i);
    has.pb(P[i]);
    seen[P[i]] = 1;
  } // 157e
  rep(i, 0, n) if (!seen[i]) pad.pb(i);
} // 103b

// Compute (min, +) product of
// simple sub-unit-Monge matrices given their
// permutation representations; time: O(n lg n)
// Left matrix has size sz(P) x sz(Q).
// Right matrix has size sz(Q) x n.
// Output matrix has size sz(P) x n.
// NON-SQUARE MATRICES ARE NOT TESTED!
vi mongeMul(const vi& P, const vi& Q, int n) {
  vi h1, p1, i1, h2, p2, i2;
  padPerm(P, h1, p1, i1, sz(Q));
  padPerm(invert(Q, n, -1), h2, p2, i2, sz(Q));

  h1.insert(h1.begin(), all(p1));
  h2.insert(h2.end(), all(p2));
  vi ans(sz(P), -1), tmp = comb(h1, h2);

  rep(i, 0, sz(i1)) {
    int j = tmp[i+sz(p1)];
    if (j < sz(i2)) {
      ans[i1[i]] = i2[j];
    } // 4d16
  } // c8a0
  return ans;
} // 3326

// Range Longest Increasing Subsequence Query;
// preprocessing: O(n lg^2 n), query: O(lg n)
#include "../structures/wavelet_tree.h"
struct ALIS {
  WaveletTree tree;
  ALIS() {}

  // Precompute data structure; O(n lg^2 n)
  ALIS(const vi& seq) {
    vi P = build(seq);
    each(k, P) if (k == -1) k = sz(seq);
    tree = {P, sz(seq)+1};
  } // f00f

  // Query LIS of s[b;e); time: O(lg n)
  int operator()(int b, int e) {
    return e - b -
      tree.count(b, sz(tree.seq[1]), 0, e);
  } // fb4a

  vi build(const vi& seq) {
    int n = sz(seq);
    if (!n) return {};
    int lo = *min_element(all(seq));
    int hi = *max_element(all(seq));

    if (lo == hi) {
      vi tmp(n);
      iota(all(tmp), 1);
      tmp.back() = -1;
      return tmp;
    } // 989d

    int mid = (lo+hi+1) / 2;
    vi p1, p2, i1, i2;
    split(seq, mid, p1, p2, i1, i2);

    p1 = expand(build(p1), i1, i1, n, -1);
    p2 = expand(build(p2), i2, i2, n, -1);
    each(j, i1) p2[j] = j;
    each(j, i2) p1[j] = j;
    return mongeMul(p1, p2, n);
  } // 6517
}; // 27ea
```

text/palindromic_tree.h f86e

```
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)
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)

#if MIN_PALINDROME_PARTITION
  // An extension that computes minimal
  // palindrome partition in O(n log n).
  vi diff{0, 0};   // len[i]-len[link[i]]
  vi slink{0, 0};  // Serial links
  vi series{0, 0}; // Series DP answer
  vi ans{0};       // DP answer for prefix
#endif

  int ext(int i) {
    while (len[i]+2 > sz(txt) ||
           txt[sz(txt)-len[i]-2] != txt.back())
      i = link[i];
    return i;
  } // d442

  // Append letter from [0;ALPHA); time: O(1)
  // (or O(lg n) for MIN_PALINDROME_PARTITION)
```

```
  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.pb({});

    #if MIN_PALINDROME_PARTITION
      diff.pb(len.back() - len[link.back()]);
      slink.pb(diff.back() == diff[link.back()]
        ? slink[link.back()] : link.back());
      series.pb(0);
    #endif
    } // e432
    last = to[last][x];

  #if MIN_PALINDROME_PARTITION
    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);
    } // ab3b
  #endif
  } // 14a4
}; // f8d9
```

text/suffix_array_linear.h 4e33

```
#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;
  } // 5fcf

  // 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;
  } // bfdc

  // 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);
  } // f9f1

  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++;
      } // 2f8a
      return code[l] < code[r];
    }); // 4cb3

  return suf;
} // 671f

// KS algorithm for suffix array; time: O(n)
vi sufArray(const string& str) {
  return sufArray(vi(all(str)), 255);
} // 2f32
```

text/suffix_automaton.h e45b

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

// Suffix automaton - minimal DFA that
// recognizes all suffixes of given string
// (and encodes all substrings);
// space: O(n*ALPHA); time: O(n)
// Paths from root are equivalent to substrings
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)

#if COUNT_SUBSTR_OCCURENCES
  vector<vi> inSufs; // Suffix-link tree
  vi cnt{0};         // Occurence count
#endif
#if COUNT_OUTGOING_PATHS
  vector<ll> paths;  // Out-path count
#endif

  SufDFA() {}

  // Build suffix automaton for given string
  // and compute extended stuff; time: O(n)
  SufDFA(const string& s) {
    each(c, s) add(c);
    finish();
  } // ec2e

  // 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); // COUNT_SUBSTR_OCCURENCES

    while (v != -1 && !to[v][x]) {
      to[v][x] = last;
      v = link[v];
    } // 4cfc

    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); // COUNT_SUBSTR_OCCURENCES
        link[last] = link[q] = sz(len)-1;
        while (v != -1 && to[v][x] == q) {
          to[v][x] = link[q];
          v = link[v];
        } // 784f
      } // 90aa
    } // af69
  } // 345a

  // 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];
  } // c363

  // Compute extended stuff (offline)
  void finish() {
  #if COUNT_SUBSTR_OCCURENCES
    inSufs.resize(sz(len));
    rep(i, 1, sz(link)) inSufs[link[i]].pb(i);
    dfsSufs(0);
  #endif
  #if COUNT_OUTGOING_PATHS
    paths.assign(sz(len), 0);
    dfs(0);
  #endif
  } // d3dc

#if COUNT_SUBSTR_OCCURENCES
  void dfsSufs(int v) {
    each(e, inSufs[v]) {
      dfsSufs(e);
      cnt[v] += cnt[e];
    } // 2469
  } // 0c60
#endif

#if COUNT_OUTGOING_PATHS
  void dfs(int v) {
    if (paths[v]) return;
    paths[v] = 1;
    each(e, to[v]) if (e) {
      dfs(e);
      paths[v] += paths[e];
    } // 22b3
  } // d004

  // Get lexicographically k-th substring
  // of represented string; time: O(|substr|)
  // Empty string has index 0.
  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;
        } // f307
        k -= paths[e];
      } // 29be
    } // 4600
    return s;
  } // e4af
#endif
}; // ef50
```

text/suffix_tree.h 8a6e

```
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 ? pii(s.x, s.y+1)
                           : pii(-1, -1);
    e = to[s.x][e];
    return e ? pii(e, L[e]+1) : pii(-1, -1);
  } // 0d7a

  // 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];
    } // 0393

    v = split({v, getR(v)-l+R[t]});
    link[t] = v;
    return t;
  } // 10bb

  // 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;
    } // 60c2
    cur = t;
  } // 1e43
}; // 8926
```

text/z_function.h 770b

```
// Computes Z function array; time: O(n)
// zf[i] = max common prefix of str and str[i:]
vi prefPref(auto& 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];
  } // e906
  zf[0] = n;
  return zf;
} // b7a7
```

trees/centroid_decomp.h 607a

```
// Centroid decomposition; space: O(n lg n)
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);
  } // 026c

  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];
  } // bbed

  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);
    } // dc82
  } // 37ee

  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;
      } // e0a5
    } // 3533

    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;
    } // 70b5
    return v;
  } // 217c
}; // 1253
```

trees/centroid_offline.h dd93

```
// 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);
  } // 8677

  // 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];
  } // 1c0d

  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;
      } // e0a5
    } // f31d
    process(v);
    on[v] = 0;
    each(e, G[v]) if (on[e]) decomp(e);
  } // f170

  // 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
}; // f598
```

trees/compress_tree.h 12da

```
#include "lca.h" // or lca_linear.h

using vpi = vector<pair<int, int>>;

// Given a rooted tree and a subset S of nodes,
// compute the minimal subtree that contains
// all the nodes by adding all pairwise LCA's
// and compressing edges; time: O(|S| log |S|)
// Returns a list of (par, orig\_index)
// representing a tree rooted at 0.
// The root points to itself.
vpi compressTree(LCA& lca, const vi& subset) {
  static vi rev; rev.resize(sz(lca.pre));
  vi li = subset, &T = lca.pre;
  auto cmp = [&](int a, int b) {
    return T[a] < T[b];
  }; // df37
  sort(all(li), cmp);
  int m = sz(li)-1;
  rep(i, 0, m) {
    int a = li[i], b = li[i+1];
    li.push_back(lca(a, b));
  } // 8757
  sort(all(li), cmp);
  li.erase(unique(all(li)), li.end());
  rep(i, 0, sz(li)) rev[li[i]] = i;
  vpi ret = {pii(0, li[0])};
  rep(i, 0, sz(li)-1) {
    int a = li[i], b = li[i+1];
    ret.emplace_back(rev[lca(a, b)], b);
  } // 5101
  return ret;
} // ef6b
```

trees/heavylight_decomp.h 5562

```
#include "../segtree/point_fixed.h"

// Heavy-Light Decomposition of tree
// with subtree query support; space: O(n)
struct HLD {
  // G[v] = children of v (no parents!)
  // G[v][0] = heavy child of v
  // par[v] = parent of vertex v
  // size[v] = size of subtree rooted at v
  // depth[v] = distance from root to v
  // pos[v] = index of v in "HLD preorder"
  // head[v] = first vertex of chain with v
  // len[v] = length of chain starting at v
  //          (0 if v is not head of chain)
  // order[i] = i-th vertex in "HLD preorder"
  vector<vi> G;
  vi par, size, depth, pos, head, len, order;
  SegTree tree; // Vertices are in HLD order

  HLD() {}

  // Initialize structure for tree G
  // and given root v; time: O(n lg n)
  HLD(vector<vi> H, int v)
      : G(move(H)), par(sz(G), -1),
        size(sz(G), 1), depth(sz(G)),
        pos(sz(G)), head(sz(G)), len(sz(G)) {
    dfs(v);
    go(v, v);
    tree = {sz(order)};
  } // 0adc

  void dfs(int v) {
    erase(G[v], par[v]);
    each(e, G[v]) {
      depth[e] = depth[par[e] = v] + 1;
      dfs(e);
      size[v] += size[e];
      if (size[e] > size[G[v][0]])
        swap(G[v][0], e);
    } // 2ffe
  } // 5f1c

  void go(int v, int h) {
    pos[v] = sz(order);
    len[head[v] = h]++;
    order.pb(v);
    each(e, G[v]) go(e, G[v][0] == e ? h : e);
  } // 89bf

  // Level Ancestor Query; time: O(lg n)
  int laq(int v, int level) {
    for (;; v = par[v]) {
      int k = level - depth[v = head[v]];
      if (k >= 0) return order[pos[v]+k];
    } // 45a8
  } // c3a8

  // Lowest Common Ancestor; time: O(lg n)
  int lca(int a, int b) {
    for (;;) {
      int ha = head[a], hb = head[b];
      if (ha == hb)
        return depth[a] < depth[b] ? a : b;
      if (depth[ha] > depth[hb]) a = par[ha];
      else b = par[hb];
    } // 1341
  } // 493e
```

```
  // Call func(begin, end, isAscending)
  // for each path segment in order
  // from a to b; time: O(lg n * time of func)
  // func can be called on empty intervals!
  void iterPath(int a, int b, auto func) {
    for (static vector<pii> tmp;;) {
      int ha = head[a], hb = head[b];
      if (ha == hb) {
        bool f = (pos[a] > pos[b]);
        if (f) swap(a, b);
        // Remove +1 from pos[a]+1 for vertex
        // queries (with +1 -> edges).
        func(pos[a]+1, pos[b]+1, !f);
        reverse(all(tmp));
        each(e, tmp) func(e.x, e.y, 0);
        return tmp.clear();
      } // 5b4d
      if (depth[ha] > depth[hb]) {
        func(pos[ha], pos[a]+1, 1);
        a = par[ha];
      } else {
        tmp.pb({pos[hb], pos[b]+1});
        b = par[hb];
      } // 1a37
    } // af03
  } // 771c

  // Query path between a and b; O(lg^2 n)
  SegTree::T queryPath(int a, int b) {
    auto ret = tree.ID;
    iterPath(a, b, [&](int i, int j, bool) {
      ret = tree.f(ret, tree.query(i, j));
    }); // 1113
    return ret;
  } // 4221

  // Query subtree of v; time: O(lg n)
  SegTree::T querySubtree(int v) {
    return tree.query(pos[v], pos[v]+size[v]);
  } // 23db
}; // 7f6f
```

trees/lca.h 048e

```
// 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]);
  } // d6ce

  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;
  } // e286

  // 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];
  } // 5514

  // 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];
  } // 27d8

  // 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;
  } // 75b3

  // 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;
  } // 07e0

  // 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));
  } // 46c9
}; // 2861
```

trees/lca_linear.h 22f7

```
// LAQ and LCA using jump pointers
// with linear memory; space: O(n)
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);
  } // 94cf

  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);
    } // b123
    post[v] = ++cnt;
  } // 3280

  // 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;
  } // f509

  // 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];
    } // fe08
    return a;
  } // 25ff

  // 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];
  } // 5514

  // 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;
  } // a340

  // 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));
  } // 34ed
}; // c19e
```

trees/link_cut_tree.h 23eb

```
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
  } // bae2

  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
  } // 8ec0

  void auxLink(int p, int i, int ch) {
    child[p][i] = ch;
    if (ch >= 0) par[ch] = p;
    update(p);
  } // 0a9a

  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);
  } // 4c76

  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);
        } // 2ebb
      } else { // zig
        rot(p, f);
      } // f8a2
    } // 446b
    push(x);
  } // 55a7

  // After this operation x becomes the end
  // of preferred path starting in root;
  void access(int x) { // amortized O(lg n)
    while (1) {
      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);
    } // 2b87
  } // d224

  // 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);
    } // 0064
  } // b246

  // 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;
  } // d78d

  // 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;
  } // fb4f

  // 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);
  } // 1908

  // 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;
  } // ae69
}; // 0197
```

util/arc_interval_cover.h ea39

```
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});
  } // b87d

  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);
  } // 5e6c

  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++;
  } // 7350
  return ans;
} // fc51
```

util/bit_hacks.h 2a84

```
// __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;

// Transpose 64x64 bit matrix
void transpose64(array<ull, 64>& M) {
  #define T(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;       \
      } // a290
  T(1,  0x5555'5555'5555'5555);
  T(2,  0x3333'3333'3333'3333);
  T(4,   0xF0F'0F0F'0F0F'0F0F);
  T(8,    0xFF'00FF'00FF'00FF);
  T(16,       0xFFFF'0000'FFFF);
  T(32,            0xFFFF'FFFFLL);
  #undef T
} // cba2

// 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));
} // 4c0c

// Permutation -> integer conversion.
int permToInt(vi& v) { // Not order preserving!
  int use = 0, i = 0, r = 0;
  each(x, v) {
    r = r * ++i +
      __builtin_popcount(use & -(1<<x));
    use |= 1 << x;
  } // d764
  return r;
} // 6574
```

util/bump_alloc.h 09f9

```
// 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];
} // fba6
void operator delete(void*) {}
```

util/compress_vec.h 33ee

```
// 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;
  } // 7eb0
  return old;
} // d53e
```

util/deque_undo.h 5c6d

```
// Deque-like undoing on data structures with
// amortized O(log n) overhead for operations.
// Maintains a deque of objects alongside
// a data structure that contains all of them.
// The data structure only needs to support
// insertions and undoing of last insertion
// using the following interface:
// - insert(...) - insert an object to DS
// - time() - returns current version number
// - rollback(t) - undo all operations after t
// Assumes time() == 0 for empty DS.
struct DequeUndo {
  DataStructure ds; // Configure DS type here.
  vector<tuple<int, int>> elems[2];
  vector<pii> his{{0,0}};

  // Push object to front or back of deque,
  // depending on side parameter.
  void push(auto val, bool side) {
    elems[side].pb(val);
    doPush(0, side);
  } // df9f

  // Pop object from front or back of deque,
  // depending on side parameter.
  void pop(int side) {
    auto &A = elems[side], &B = elems[!side];
    int cnt[2] = {};

    if (A.empty()) {
      assert(!B.empty());
      auto it = B.begin() + sz(B)/2 + 1;
      A.assign(B.begin(), it);
      B.erase(B.begin(), it);
      reverse(all(A));
      his.resize(1);
      cnt[0] = sz(A);
      cnt[1] = sz(B);
    } else {
      do {
        cnt[his.back().y ^ side]++;
        his.pop_back();
      } while (cnt[0]*2 < cnt[1] &&
               cnt[0] < sz(A));
    } // b4ef

    cnt[0]--;
    A.pop_back();
    ds.rollback(his.back().x);
    for (int i : {1, 0})
      while (cnt[i]) doPush(--cnt[i], i^side);
  } // 6eba

  void doPush(int i, bool s) {
    apply([&](auto... x) { ds.insert(x...); },
      elems[s].rbegin()[i]);
    his.pb({ds.time(), s});
  } // 4fed
}; // 189b
```

util/int128_io.h a481

```
istream& operator>>(istream& i, __int128& x) {
  char s[50], *p = s;
  for (i >> s, x = 0, p += *p < 48; *p;)
    x = x*10 + *p++ - 48;
  if (*s == 45) x = -x;
  return i;
} // 6015

// Note: Doesn't work for INT128_MIN!
ostream& operator<<(ostream& o, __int128 x) {
  if (x < 0) o << '-', x = -x;
  char s[50] = {}, *p = s+49;
  for (; x > 9; x /= 10) *--p = char(x%10+48);
  return o << ll(x) << p;
} // b9ed
```

util/inversion_vector.h dcdb

```
// Get inversion vector for sequence of
// numbers in [0;n); ret[i] = count of numbers
// greater 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;
    } // 4ed0
  } // a4f0
  return ret;
} // 86e4

// 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;
    } // 916f
  } // c9b5
  return ret;
} // 4bd8
```

util/longest_inc_subseq.h 98cb

```
// Longest Increasing Subsequence; O(n lg n)
vi lis(const vi& seq) {
  vi dp(sz(seq)+1, INT_MAX);
  vi ind(sz(dp), -1), prv(sz(dp));

  rep(i, 0, sz(seq)) {
    int j = int(lower_bound(1+all(dp), seq[i])
              - dp.begin());
    prv[i] = ind[j-1];
    dp[j] = seq[ind[j] = i];
  } // b229

  vi ret;
  int i = *--find(1+all(ind), -1);
  while (i != -1) ret.pb(i), i = prv[i];
  reverse(all(ret));
  return ret;
} // c0fc
```

util/max_rects.h 4b65

```
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
}; // e5d1

// 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];
      } // 6e7e
      rect.end = k-1;
    } // e03f
  } // 2796
  return ans;
} // f8f9
```

util/mo.h 2278

```
// 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;
}; // b76d

// 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);
} // 0fb9

// 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];
  }); // 9aea
  return ret;
} // 29f4
```

util/multinomial.h a0a3

```
// Computes n! / (k1! * .. * kn!)
ll multinomial(vi& v) {
  ll c = 1, m = v.empty() ? 1 : v[0];
  rep(i, 1, sz(v)) rep(j, 0, v[i])
    c = c * ++m / (j+1);
  return c;
} // d07d
```

util/packing.h 3971

```
// Utilities for packing precomputed tables.
// Encodes 13 bits using two characters.

// Example usage:
//   Writer out;
//   out.ints(-123, 8);
//   out.done();
//   cout << out.buf;
struct Writer {
  string buf;
  int cur = 0, has = 0;

  void done() {
    buf.pb(char(cur%91 + 35));
    buf.pb(char(cur/91 + 35));
    cur = has = 0;
  } // 0e09

  // Write unsigned b-bit integer.
  void intu(uint64_t v, int b) {
    assert(b == 64 || v < (1ull<<b));
    while (b--) {
      cur |= (v & 1) << has;
      if (++has == 13) done();
      v >>= 1;
    } // f132
  } // 0f64

  // Write signed b-bit integer (sign included)
  void ints(ll v, int b) {
    intu(v < 0 ? -v*2+1 : v*2, b);
  } // 08d0
}; // 7d0d

// Example usage:
//   Reader in("packed_data");
//   int firstValue = in.ints(8);
struct Reader {
  const char *buf;
  ll cur = 0;

  Reader(const char *s) : buf(s) {}

  // Read unsigned b-bit integer.
  uint64_t intu(int b) {
    uint64_t n = 0;
    rep(i, 0, b) {
      if (cur < 2) {
        cur = *buf++ + 4972;
        cur += *buf++ * 91;
      } // a930
      n |= (cur & 1) << i;
      cur >>= 1;
    } // f12f
    return n;
  } // 1f23

  // Read signed b-bit integer (sign included).
  ll ints(int b) {
    auto v = intu(b);
    return (v%2 ? -1 : 1) * ll(v/2);
  } // 1fc9
}; // 2217
```

util/parallel_binsearch.h 02bb

```
// Run `n` binary searches on [b;e) parallely.
// `cmp` should be lambda with arguments:
// 1) vector<pii>& - pairs (v, i)
//    which are queries if value for index i
//    is greater or equal to v;
//    pairs are sorted by v
// 2) vector<bool>& - output vector,
//    set true at index i if value
//    for i-th query is >= queried value
// Returns vector of found values;
// time: O((n+c) lg range), where c is cmp time
vi multiBS(int b, int e, int n, auto cmp) {
  if (b >= e) return vi(n, b);
  vector<pii> que(n), rng(n, {b, e});
  vector<bool> ans(n);

  rep(i, 0, n) que[i] = {(b+e)/2, i};

  for (int k = __lg(e-b); k >= 0; k--) {
    int last = 0, j = 0;
    cmp(que, ans);
    rep(i, 0, sz(que)) {
      pii &q = que[i], &r = rng[q.y];
      if (q.x != last) last = q.x, j = i;
      (ans[i] ? r.x : r.y) = q.x;
      q.x = (r.x+r.y) / 2;
      if (!ans[i]) swap(que[i], que[j++]);
    } // 4765
  } // 8bc8

  vi ret;
  each(p, rng) ret.pb(p.x);
  return ret;
} // 638f
```

util/radix_sort.h 0573

```
// Stable countingsort; time: O(k+sz(vec))
// See example usage in radixSort for pairs.
void countSort(vi& vec, auto 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];
} // ef86

// 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;
} // f272
```

Pick's theorem

For a simple polygon with integer vertices, area A, i grid points in the interior, and b grid points on the boundary: $A = i + b/2 - 1$.

Tutte matrix (perfect matching test)

$$M_{ij} = \begin{cases} x_{ij} & \text{if } ij \in E, i < j \\ -x_{ji} & \text{if } ij \in E, i > j \\ 0 & \text{otherwise} \end{cases}$$

$\det(M) = 0 \iff$ no perfect matching w.h.p.

Kirchhoff's theorem (# of spanning trees)

$$M_{ij} = \begin{cases} \deg_{\text{in}}(i) & \text{if } i = j \\ -\#(ij \text{ edges}) & \text{if } i \neq j \end{cases}$$

$M' = M$ with i-th row and column removed

$\det(M') = \#$ of oriented spanning trees rooted at i

Cayley's formula (# of labelled trees)

For degree sequence $d_1, \ldots, d_n$:

$$\frac{(n-2)!}{(d_1-1)!(d_2-1)!\cdots(d_n-1)!}$$

$n_1 n_2 \ldots n_k n^{k-2} =$ for k existing trees of size n_i

$kn^{n-k-1} =$ forests on n vertices with k components such that $1, \ldots, k$ belong to different components

$x_1 \ldots x_n (x_1 + \ldots + x_n)^{n-2} = \sum_T x_1^{d_1(T)} \ldots x_n^{d_n(T)}$

of partitions into positive integers

$$p(0) = 1$$

$$p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k-1)/2)$$

Bernoulli numbers

EGF of Bernoulli numbers is $B(t) = \frac{t}{e^t - 1}$ (FFT-able).

$B[0, \ldots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \ldots]$

Sums of powers:

$$\sum_{i=1}^{n} n^m = \frac{1}{m+1} \sum_{k=0}^{m} \binom{m+1}{k} B_k \cdot (n+1)^{m+1-k}$$

Stirling numbers of the first kind

Number of permutations on n items with k cycles.

$$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k)$$

$$\textstyle\sum_{k=0}^{n} c(n,k)x^k = x(x+1)\ldots(x+n-1)$$

Eulerian numbers

Number of permutations $\pi \in S_n$ in which exactly k elements are greater than the previous element. k j:s s.t. $\pi(j) > \pi(j+1)$, $k+1$ j:s s.t. $\pi(j) \geq j$, k j:s s.t. $\pi(j) > j$.

$$E(n,k) = (n-k)E(n-1,k-1) + (k+1)E(n-1,k)$$

$$E(n,0) = E(n,n-1) = 1$$

$$E(n,k) = \sum_{j=0}^{k} (-1)^j \binom{n+1}{j} (k+1-j)^n$$

Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

$$S(n,k) = S(n-1,k-1) + kS(n-1,k)$$

$$S(n,1) = S(n,n) = 1$$

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n$$

Bell numbers

Total number of partitions of n distinct elements. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \ldots$. For p prime,

$$B(p^m + n) \equiv mB(n) + B(n+1) \pmod{p}$$

$$B(n) = \sum_{k=0}^{n} \binom{n}{k} \cdot B(k)$$

Catalan numbers

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

$$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2} C_n,\ C_{n+1} = \sum C_i C_{n-i}$$

$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, \ldots$

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

Catalan convolution: find the count of balanced parentheses sequences consisting of $n+k$ pairs of parentheses where the first k symbols are open brackets.

$$C^k = \frac{k+1}{n+k+1}\binom{2n+k}{n}$$

Burnside's lemma

G = group that acts on a set X

X^g = set of elements fixed by $g \in G$

X/G = set of orbits, i.e. equivalence classes by G

$$|X/G| = \frac{1}{|G|}\sum_{g \in G} |X^g|$$

If $f(n)$ counts "configurations" (of some sort) of length n, we can ignore rotational symmetry using $G = \mathbb{Z}_n$ to get

$$g(n) = \frac{1}{n}\sum_{k=0}^{n-1} f(\gcd(n,k)) = \frac{1}{n}\sum_{k|n} f(k)\phi(n/k).$$

Sums

$$c^a + \ldots + c^b = \frac{c^{b+1} - c^a}{c-1} \qquad \text{if } c \neq 1$$

$$1 + \ldots + n = \frac{n(n+1)}{2}$$

$$1^2 + \ldots + n^2 = \frac{n(2n+1)(n+1)}{6}$$

$$1^3 + \ldots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$1^4 + \ldots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots,\ (|x| < \infty)$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots,\ (x \leq 1)$$

$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots,\ (|x| \leq 1)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots,\ (|x| < \infty)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots,\ (|x| < \infty)$$

Trigonometry

$$\sin(v \pm w) = \sin v \cos w \pm \cos v \sin w$$

$$\cos(v \pm w) = \cos v \cos w \mp \sin v \sin w$$

$$\sin v + \sin w = 2 \sin \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$\cos v + \cos w = 2 \cos \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$|\sin \frac{x}{2}| = \sqrt{\frac{1-\cos x}{2}} \qquad |\cos \frac{x}{2}| = \sqrt{\frac{1+\cos x}{2}}$$

$$\tan(v \pm w) = \frac{\tan v \pm \tan w}{1 \mp \tan v \tan w} \qquad |\tan \frac{x}{2}| = \sqrt{\frac{1-\cos x}{1+\cos x}}$$

$$(V+W)\tan(v-w)/2 = (V-W)\tan(v+w)/2$$

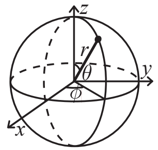
where V, W are lengths of sides opposite angles v, w.

$$a \cos x + b \sin x = r \cos(x - \phi)$$

$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2 + b^2}, \phi = \operatorname{atan2}(b,a)$.

Spherical coordinates



$$x = r \sin\theta \cos\phi \qquad r = \sqrt{x^2 + y^2 + z^2}$$

$$y = r \sin\theta \sin\phi \qquad \theta = \operatorname{acos}(z/\sqrt{x^2+y^2+z^2})$$

$$z = r \cos\theta \qquad \phi = \operatorname{atan2}(y,x)$$

Integrals

$$\int \sqrt{a^2 + x^2}dx = \frac{x}{2}\sqrt{a^2+x^2} + \frac{a^2}{2}\ln(x + \sqrt{a^2+x^2})$$

$$\int \sqrt{a^2 - x^2}dx = \frac{x}{2}\sqrt{a^2-x^2} + \frac{a^2}{2}\arcsin\frac{x}{|a|}$$

$$\int \frac{dx}{\sqrt{a^2-x^2}} = \arcsin\frac{x}{|a|} = -\arccos\frac{x}{|a|}$$

$$\int \frac{dx}{\sqrt{a^2+x^2}} = \ln(x + \sqrt{a^2+x^2})$$

Sub $s = \tan(x/2)$ to get: $dx = \frac{2\ ds}{1+s^2}$,

$$\sin x = \frac{2s}{1+s^2},\ \cos x = \frac{1-s^2}{1+s^2}$$

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

(Integration by parts)

$$\int \tan ax = -\frac{\ln|\cos ax|}{a}$$

$$\int x \sin ax = \frac{\sin ax - ax\cos ax}{a^2}$$

$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2}\operatorname{erf}(x), \quad \int xe^{ax}dx = \frac{e^{ax}}{a^2}(ax-1)$$

$$\frac{d}{dx}\tan x = 1 + \tan^2 x, \quad \frac{d}{dx}\arctan x = \frac{1}{1+x^2}$$

Curve length: $\int_a^b \sqrt{1 + (f'(x))^2}dx$

When $X(t), Y(t)$: $\int_a^b \sqrt{(X'(t))^2 + (Y'(t))^2}dt$

Solid of revolution vol: $\pi \int_a^b (f(x))^2 dx$

Surface area: $2\pi \int_a^b |f(x)|\sqrt{1+(f'(x))^2}dx$

Markov chains

A Markov chain is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ be a sequence of random variables generated by the Markov process. Then there is a transition matrix $\mathbf{P} = (p_{ij})$, with $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

π is a stationary distribution if $\pi = \pi\mathbf{P}$. If the Markov chain is irreducible (it is possible to get to any state from any state), then $\pi_i = \frac{1}{\mathbb{E}(T_i)}$ where $\mathbb{E}(T_i)$ is the expected time between two visits in state i. π_j/π_i is the expected number of visits in state j between two visits in state i.

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors, π_i is proportional to node i's degree.

A Markov chain is ergodic if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and aperiodic (i.e., the gcd of cycle lengths is 1). $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{1}\pi$.

A Markov chain is an A-chain if the states can be partitioned into two sets $\mathbf{A}$ and $\mathbf{G}$, such that all states in $\mathbf{A}$ are absorbing ($p_{ii} = 1$), and all states in $\mathbf{G}$ leads to an absorbing state in $\mathbf{A}$. The probability for absorption in state $i \in \mathbf{A}$, when the initial state is j, is $a_{ij} = p_{ij} + \sum_{k \in \mathbf{G}} a_{ik} p_{kj}$. The expected time until absorption, when the initial state is i, is $t_i = 1 + \sum_{k \in \mathbf{G}} p_{ki} t_k$.

Pythagorean Triples

Uniquely generated by

$$a = k \cdot (m^2 - n^2),\ \ b = k \cdot (2mn),\ \ c = k \cdot (m^2 + n^2),$$

$m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

Estimates

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

Mobius Function

$$\mu(n) = \begin{cases} 0 & n \text{ is not square free} \\ 1 & n \text{ has even number of prime factors} \\ -1 & n \text{ has odd number of prime factors} \end{cases}$$

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu(d) g(n/d)$$

$\sum_{d|n} \mu(d) = [n = 1]$ (very useful)

$g(n) = \sum_{n|d} f(d) \Leftrightarrow f(n) = \sum_{n|d} \mu(d/n) g(d)$

$g(n) = \sum_{1 \le m \le n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum \mu(m) g(\lfloor \frac{n}{m} \rfloor)$

Lucas' Theorem

Let n, m be non-negative integers and p a prime. Write $n = n_k p^k + ... + n_1 p + n_0$ and $m = m_k p^k + ... + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod{p}$.

CPUID

```
Vendors:
1: intel; 2: amd
```

```
Types:
1: bonnell, atom; 2: core2; 3: corei7
4: amdfam10h; 5: amdfam15h, shanghai;
6: silvermont, slm, istanbul
7: knl, bdver1; 8: bdver2; 9: btver2
10: amdfam17h; 11: knm; 12: goldmont
13: goldmont-plus; 14: tremont
15: amdfam19h; 18: grandridge
19: clearwaterforest
```

```
Subtypes:
1: nehalem; 2: westmere; 3: sandybridge;
4: barcelona; 7: btver1; 9: bdver3;
10: bdver4; 11: znver1; 12: ivybridge;
13: haswell; 14: broadwell; 15: skylake;
16: skylake-avx512;
17: cannonlake, sierraforest;
18: icelake-client; 19: icelake-server;
20: znver2; 21: cascadelake;
22: tigerlake; 23: cooperlake;
24: sapphirerapids, emeraldrapids;
25: alderlake, raptorlake, meteorlake, ...;
26: znver3; 27: rocketlake; 28: lujiazui;
29: znver4; 30: graniterapids;
31: graniterapids-d; 32: arrowlake;
33: arrowlake-s, lunarlake; 34: pantherlake;
35: yongfeng; 36: znver5;
```

```
Check CPU features using `man g++`.
Verify: __builtin_cpu_is __builtin_cpu_supports
```

CPUID submit

```cpp
#include "cpuid.h"

extern "C" struct {
  int vendor, type, subtype, features;
} __cpu_model;

int main() {
  char brand[50] = {};
  auto b = (unsigned*)brand;
  rep(i, 2, 5) {
    __get_cpuid(INT_MIN+i, b, b+1, b+2, b+3);
    b += 4;
  }

  auto m = __cpu_model;
  cout << brand << endl << m.vendor << ' ';
  cout << m.type << ' ' << m.subtype << endl;

  // Extract CPU subtype using 4 submissions.
  int submitID = 0; // Set to 0, 1, 2, 3.

  int t = m.subtype;
  while (submitID--) t /= 3;
  if (t%3 == 2) for (volatile int c=0;;) c=c;
  return t%3;
}
```

CPUID recovery

```cpp
int main() {
  // 0 = ANS, 1 = RTE, 2 = TLE
  int id = 0, status[4] = {0, 2, 1, 0};
  rep(i, 0, 4) id = id*3 + status[3-i];
  cout << id << endl;
}
```

Checklist

- `.vimrc`
- `.bashrc`
- `template.cpp`
- Hash verification
- Java
- Python
- Printing
- Clarifications
- Documentation
- Submit script
- Whitespace/case insensitivity
- Source code limit
- CPU on local machine
- CPU on checker
- Test Dijkstra speed
- `clock()`
- Judge errors

List binaries

```bash
echo $PATH | tr ':' ' ' | xargs ls |          \
  grep -v / | sort | uniq
```