Contents

1 Non-code things

1.1 Hash

Hash: 9538616d87aa2d06c37c129736430a98

```
tr -d ’[:space:]’ | md5sum | cut -d ’ ’ -f 1
```

1.2 Makefile

Hash: 1be30703415446aaf3a1260294222d71

```
CXX = g++
CXXFLAGS = -Wall -Wextra -pedantic -std=c++11 -O2
    ↪ -Wshadow -Wformat=2 -Wfloat-equal
    ↪ -Wconversion -Wlogical-op -Wshift-overflow=2
    ↪ -Wduplicated-cond -Wcast-qual -Wcast-align
DEBUGFLAGS = -D_GLIBCXX_DEBUG
    ↪ -D_GLIBCXX_DEBUG_PEDANTIC -fsanitize=address
    ↪ -fsanitize=undefined
    ↪ -fno-sanitize-recover=all -fstack-protector
    ↪ -D_FORTIFY_SOURCE=2

CXXFLAGS += $(DEBUGFLAGS)

TARGET := $(notdir $(CURDIR))
EXECUTE := ./$(TARGET)

CASES := $(sort $(basename $(wildcard *.in)))
TESTS := $(sort $(basename $(wildcard *.out)))

all: $(TARGET)

clean:
  -rm -rf $(TARGET) *.res

%: %.cpp
  $(LINK.cpp) $< $(LOADLIBES) $(LDLIBS) -o $@

run: $(TARGET)
  time $(EXECUTE)

%.res: $(TARGET) %.in
  time $(EXECUTE) < $*.in > $*.res

%.out: %

test_%: %.res %.out
  diff $*.res $*.out

runs: $(patsubst %,%.res,$(CASES))
test: $(patsubst %,test_%,$(TESTS))

.PHONY: all clean run test test_% runs

.PRECIOUS: %.res
```

1.3 vimrc

Hash: 8f870abf0ba8837fb91734ae9a941ba8

```
set nocp ai bs=2 cul hls ic is lbr ls=2 mouse=a nu
    ↪ ru sc scs smd so=3 sw=4 ts=4
filetype plugin indent on
syntax on

map gA m’ggVG"+y’’
```

1.4 nanorc

Hash: 4364dc56fff2b10d5aacd6dc61625802

```
set tabsize 4
set const
set autoindent
```

2 Geometry

2.1 Point

2.2 Geometric primitives

Hash: a1ef04616fa78cdafb4e4425490521b7

```cpp
/**
 * Author: Ulf Lundstrom
 * Date: 2009-02-26
 * License: CC0
 * Source: My head with inspiration from tinyKACTL
 * Description: Class to handle points in the plane.
 *  T can be e.g. double or long long. (Avoid int.)
 * Status: Works fine, used a lot
 */
#pragma once

template<class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x), y(y) {}
  bool operator<(P p) const { return tie(x,y) <
      ↪ tie(p.x,p.y); }
  bool operator==(P p) const { return
      ↪ tie(x,y)==tie(p.x,p.y); }
  P operator+(P p) const { return P(x+p.x, y+p.y); }
  P operator-(P p) const { return P(x-p.x, y-p.y); }
  P operator*(T d) const { return P(x*d, y*d); }
  P operator/(T d) const { return P(x/d, y/d); }
  T dot(P p) const { return x*p.x + y*p.y; }
  T cross(P p) const { return x*p.y - y*p.x; }
  T cross(P a, P b) const { return
      ↪ (a-*this).cross(b-*this); }
  T dist2() const { return x*x + y*y; }
  double dist() const { return
      ↪ sqrt((double)dist2()); }
  // angle to x-axis in interval [-pi, pi]
  double angle() const { return atan2(y, x); }
  P unit() const { return *this/dist(); } // makes
      ↪ dist()=1
  P perp() const { return P(-y, x); } // rotates +90
      ↪ degrees
  P normal() const { return perp().unit(); }
  // returns point rotated ’a’ radians ccw around
      ↪ the origin
  P rotate(double a) const {
    return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
};
```

Hash: 9f1809ec3ebb5947391f67f96f50df0b

```cpp
/**
 * Author: Ulf Lundstrom
 * Date: 2009-03-21
 * License: CC0
 * Source: Basic math
 * Description:\\
\begin{minipage}{75mm}
Returns the signed distance between point p and the
    ↪ line containing points a and b. Positive
    ↪ value on left side and negative on right as
    ↪ seen from a towards b. a==b gives nan. P is
```

```
     ↪ supposed to be Point<T> or Point3D<T> where T
     ↪ is e.g. double or long long. It uses products
     ↪ in intermediate steps so watch out for
     ↪ overflow if using int or long long. Using
     ↪ Point3D will always give a non-negative
     ↪ distance.
\end{minipage}
\begin{minipage}{15mm}
\includegraphics[width=\textwidth]{../content/geometry/lineDistance}
\end{minipage}
 * Status: tested
 */
#pragma once

#include "Point.h"

template<class P>
double lineDist(const P& a, const P& b, const P& p) {
  return (double)(b-a).cross(p-a)/(b-a).dist();
}
```

Hash: 787942a8a4b9ae5f94d99a027d75eb4f

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-03-21
 * License: CC0
 * Source:
 * Description:\\
\begin{minipage}{75mm}
Returns the shortest distance between point p and
     ↪ the line segment from point s to e.
\end{minipage}
\begin{minipage}{15mm}
\vspace{-10mm}
\includegraphics[width=\textwidth]{../content/geometry/SegmentDistance}
\end{minipage}
 * Status: tested
 * Usage:
 *  Point<double> a, b(2,2), p(1,1);
 *  bool onSegment = segDist(a,b,p) < 1e-10;
 */
#pragma once

#include "Point.h"

typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
  if (s==e) return (p-s).dist();
  auto d = (e-s).dist2(), t =
     ↪ min(d,max(.0,(p-s).dot(e-s)));
  return ((p-s)*d-(e-s)*t).dist()/d;
}
```

Hash: ef32639e7fd5e214102f21cd0975fb4b

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-03-21
 * License: CC0
 * Source:
 * Description:\\
\begin{minipage}{75mm}
If a unique intersetion point between the line
     ↪ segments going from s1 to e1 and from s2 to
     ↪ e2 exists r1 is set to this point and 1 is
     ↪ returned.
If no intersection point exists 0 is returned and if
     ↪ infinitely many exists 2 is returned and r1
     ↪ and r2 are set to the two ends of the common
     ↪ line.
The wrong position will be returned if P is
     ↪ Point<int> and the intersection point does
     ↪ not have integer coordinates.
Products of three coordinates are used in
     ↪ intermediate steps so watch out for overflow
     ↪ if using int or long long.
Use segmentIntersectionQ to get just a true/false
     ↪ answer.
\end{minipage}
\begin{minipage}{15mm}
\includegraphics[width=\textwidth]{../content/geometry/SegmentIntersection}
\end{minipage}
 * Status: Well tested with unitTest and with Kattis
     ↪ problem intersection.
 * Usage:
 * Point<double> intersection, dummy;
 * if
     ↪ (segmentIntersection(s1,e1,s2,e2,intersection,dummy)==1)
 *   cout << "segments intersect at " <<
     ↪ intersection << endl;
 */
#pragma once

#include "Point.h"

template<class P>
int segmentIntersection(const P& s1, const P& e1,
    const P& s2, const P& e2, P& r1, P& r2) {
  if (e1==s1) {
    if (e2==s2) {
      if (e1==e2) { r1 = e1; return 1; } //all equal
      else return 0; //different point segments
    } else return
        ↪ segmentIntersection(s2,e2,s1,e1,r1,r2);//swap
  }
  //segment directions and separation
  P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
  auto a = v1.cross(v2), a1 = v1.cross(d), a2 =
      ↪ v2.cross(d);
  if (a == 0) { //if parallel
    auto b1=s1.dot(v1), c1=e1.dot(v1),
         b2=s2.dot(v1), c2=e2.dot(v1);
    if (a1 || a2 ||
        ↪ max(b1,min(b2,c2))>min(c1,max(b2,c2)))
      return 0;
    r1 = min(b2,c2)<b1 ? s1 : (b2<c2 ? s2 : e2);
    r2 = max(b2,c2)>c1 ? e1 : (b2>c2 ? s2 : e2);
    return 2-(r1==r2);
  }
  if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
  if (0<a1 || a<-a1 || 0<a2 || a<-a2)
    return 0;
  r1 = s1-v1*a2/a;
  return 1;
}
```

Hash: 8ae8941e4b6fea60032757996cc23f67

```
/**
 * Author: Ulf Lundstrom, Simon Lindholm
 * Date: 2016-09-24
 * License: CC0
 * Source: SegmentIntersection.h
 * Description: Like segmentIntersection, but only
     ↪ returns true/false.
 * Products of three coordinates are used in
     ↪ intermediate steps so watch out for overflow
     ↪ if using int or long long.
 * Status: Relatively well tested.
 */
#pragma once

#include "Point.h"

template<class P>
bool segmentIntersectionQ(P s1, P e1, P s2, P e2) {
  if (e1 == s1) {
    if (e2 == s2) return e1 == e2;
    swap(s1,s2); swap(e1,e2);
  }
  P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
  auto a = v1.cross(v2), a1 = d.cross(v1), a2 =
      ↪ d.cross(v2);
  if (a == 0) { // parallel
    auto b1 = s1.dot(v1), c1 = e1.dot(v1),
         b2 = s2.dot(v1), c2 = e2.dot(v1);
    return !a1 && max(b1,min(b2,c2)) <=
        ↪ min(c1,max(b2,c2));
  }
  if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
  return (0 <= a1 && a1 <= a && 0 <= a2 && a2 <= a);
}
```

Hash: 7ec51c26be244a69e5d17667be0ca88b

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-03-21
 * License: CC0
 * Source:
 * Description:\\
\begin{minipage}{75mm}
If a unique intersetion point of the lines going
     ↪ through s1,e1 and s2,e2 exists r is set to
     ↪ this point and 1 is returned. If no
     ↪ intersection point exists 0 is returned and
     ↪ if infinitely many exists -1 is returned. If
     ↪ s1==e1 or s2==e2 -1 is returned. The wrong
     ↪ position will be returned if P is Point<int>
     ↪ and the intersection point does not have
     ↪ integer coordinates. Products of three
     ↪ coordinates are used in intermediate steps so
     ↪ watch out for overflow if using int or long
     ↪ long.
\end{minipage}
\begin{minipage}{15mm}
\includegraphics[width=\textwidth]{../content/geometry/lineI
\end{minipage}
 * Status: tested
 * Usage:
 *  point<double> intersection;
 *  if (1 ==
     ↪ LineIntersection(s1,e1,s2,e2,intersection))
 *    cout << "intersection point at " <<
     ↪ intersection << endl;
 */
#pragma once

#include "Point.h"
```

```
template<class P>
int lineIntersection(const P& s1, const P& e1, const
    ↪ P& s2,
    const P& e2, P& r) {
  if ((e1-s1).cross(e2-s2)) { //if not parallell
    r =
       ↪ s2-(e2-s2)*(e1-s1).cross(s2-s1)/(e1-s1).cross(e2-s2);
    return 1;
  } else
    return -((e1-s1).cross(s2-s1)==0 || s2==e2);
}
```

Hash: 588f94364662775562aac78326701b81

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-03-21
 * License: CC0
 * Source:
 * Description: Returns where $p$ is as seen from
     ↪ $s$ towards $e$. 1/0/-1 $\Leftrightarrow$
     ↪ left/on line/right. If the optional argument
     ↪ $eps$ is given 0 is returned if $p$ is
     ↪ within distance $eps$ from the line. P is
     ↪ supposed to be Point<T> where T is e.g.
     ↪ double or long long. It uses products in
     ↪ intermediate steps so watch out for overflow
     ↪ if using int or long long.
 * Status: tested
 * Usage:
 *  bool left = sideOf(p1,p2,q)==1;
 */
#pragma once

#include "Point.h"

template<class P>
int sideOf(const P& s, const P& e, const P& p) {
  auto a = (e-s).cross(p-s);
  return (a > 0) - (a < 0);
}
template<class P>
int sideOf(const P& s, const P& e, const P& p,
    ↪ double eps) {
  auto a = (e-s).cross(p-s);
  double l = (e-s).dist()*eps;
  return (a > l) - (a < -l);
}
```

Hash: c02e37d094f7d8211e3d7d60da38c2cd

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-04-09
 * License: CC0
 * Source: Basic geometry
 * Description: Returns true iff p lies on the line
     ↪ segment from s to e. Intended for use with
     ↪ e.g. Point<long long> where overflow is an
     ↪ issue. Use (segDist(s,e,p)<=epsilon) instead
     ↪ when using Point<double>.
 * Status:
 */
#pragma once

#include "Point.h"

template<class P>
bool onSegment(const P& s, const P& e, const P& p) {
  P ds = p-s, de = p-e;
  return ds.cross(de) == 0 && ds.dot(de) <= 0;
}
```

Hash: aff536935d1da349c9bda293fa673b90

```
/**
 * Author: Per Austrin, Ulf Lundstrom
 * Date: 2009-04-09
 * License: CC0
 * Source:
 * Description:\\
\begin{minipage}{75mm}
 Apply the linear transformation (translation,
     ↪ rotation and scaling) which takes line p0-p1
     ↪ to line q0-q1 to point r.
\end{minipage}
\begin{minipage}{15mm}
\vspace{-8mm}
\includegraphics[width=\textwidth]{../content/geometry/linearTransformation}
\vspace{-2mm}
\end{minipage}
 * Status: not tested
 */
#pragma once

#include "Point.h"

typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) {
  P dp = p1-p0, dq = q1-q0, num(dp.cross(dq),
      ↪ dp.dot(dq));
  return q0 + P((r-p0).cross(num),
      ↪ (r-p0).dot(num))/dp.dist2();
}
```

Hash: 4755f1d0abe4ea081c92fb7e84c62ead

```
/**
 * Author: Simon Lindholm
 * Date: 2015-01-31
 * License: CC0
 * Source:
 * Description: A class for ordering angles (as
     ↪ represented by int points and
 *  a number of rotations around the origin). Useful
     ↪ for rotational sweeping.
 *  Sometimes also represents points or vectors.
 * Usage:
 *  vector<Angle> v = {w[0], w[0].t360() ...}; //
     ↪ sorted
 *  int j = 0; rep(i,0,n) { while (v[j] <
     ↪ v[i].t180()) ++j; }
 *  // sweeps j such that (j-i) represents the
     ↪ number of positively oriented triangles with
     ↪ vertices at 0 and i
 * Status: Used, works well
 */
#pragma once

struct Angle {
  int x, y;
  int t;
  Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
  Angle operator-(Angle b) const { return {x-b.x,
      ↪ y-b.y, t}; }
  int quad() const {
    assert(x || y);
    if (y < 0) return (x >= 0) + 2;
    if (y > 0) return (x <= 0);
    return (x <= 0) * 2;
  }
  Angle t90() const { return {-y, x, t + (quad() ==
      ↪ 3)}; }
  Angle t180() const { return {-x, -y, t + (quad()
      ↪ >= 2)}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare
      ↪ distances
  return make_tuple(a.t, a.quad(), a.y * (ll)b.x) <
         make_tuple(b.t, b.quad(), a.x * (ll)b.y);
}

// Given two points, this calculates the smallest
    ↪ angle between
// them, i.e., the angle that covers the defined
    ↪ line segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
          make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a +
    ↪ vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b -
    ↪ angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu -
      ↪ (b < a)};
}
```

2.3 Circles

Hash: 1a1c0a9e74b421bcb9faff0111d3b650

```
/**
 * Author: Simon Lindholm
 * Date: 2015-09-01
 * License: CC0
 * Description: Computes a pair of points at which
     ↪ two circles intersect. Returns false in case
     ↪ of no intersection.
 * Status: somewhat tested
 */
#pragma once

#include "Point.h"

typedef Point<double> P;
bool circleIntersection(P a, P b, double r1, double
    ↪ r2,
    pair<P, P>* out) {
```

```
  P delta = b - a;
  assert(delta.x || delta.y || r1 != r2);
  if (!delta.x && !delta.y) return false;
  double r = r1 + r2, d2 = delta.dist2();
  double p = (d2 + r1*r1 - r2*r2) / (2.0 * d2);
  double h2 = r1*r1 - p*p*d2;
  if (d2 > r*r || h2 < 0) return false;
  P mid = a + delta*p, per = delta.perp() * sqrt(h2
      ↪ / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

Hash: 9297c614cdc28ebf23ab50505db5148b

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-04-06
 * License: CC0
 * Source:
 * Description:\\
\begin{minipage}{75mm}
Returns a pair of the two points on the circle with
    ↪ radius r centered around c whos tangent lines
    ↪ intersect p. If p lies within the circle
    ↪ NaN-points are returned. P is intended to be
    ↪ Point<double>. The first point is the one to
    ↪ the right as seen from the p towards c.
\end{minipage}
\begin{minipage}{15mm}
\includegraphics[width=\textwidth]{../content/geometry/circleTangents}
\end{minipage}
 * Status: tested
 * Usage:
 *  typedef Point<double> P;
 *  pair<P,P> p = circleTangents(P(100,2),P(0,0),2);
 */
#pragma once

#include "Point.h"

template<class P>
pair<P,P> circleTangents(const P &p, const P &c,
    ↪ double r) {
  P a = p-c;
  double x = r*r/a.dist2(), y = sqrt(x-x*x);
  return make_pair(c+a*x+a.perp()*y,
      ↪ c+a*x-a.perp()*y);
}
```

Hash: df5951dc406606b52b55563a5f96c7d5

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-04-11
 * License: CC0
 * Source: http://en.wikipedia.org/wiki/Circumcircle
 * Description:\\
\begin{minipage}{75mm}
The circumcirle of a triangle is the circle
    ↪ intersecting all three vertices. ccRadius
    ↪ returns the radius of the circle going
    ↪ through points A, B and C and ccCenter
    ↪ returns the center of the same circle.
\end{minipage}
\begin{minipage}{15mm}
\vspace{-2mm}
\includegraphics[width=\textwidth]{../content/geometry/circumcircle}
\end{minipage}
 * Status: tested
 */
#pragma once

#include "Point.h"

typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A-C).dist()/
      abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C-A, c = B-A;
  return A +
      ↪ (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
}
```

Hash: f687c28c5f8f29bb75443dfa50c490c7

```
/**
 * Author: Simon Lindholm
 * Date: 2017-04-20
 * License: CC0
 * Source: NAPC 2017 solution presentation
 * Description: Computes the minimum circle that
     ↪ encloses a set of points.
 * Time: expected O(n)
 * Status: fuzz-tested
 */
#pragma once

#include "circumcircle.h"

pair<double, P> mec2(vector<P>& S, P a, P b, int n) {
  double hi = INFINITY, lo = -hi;
  rep(i,0,n) {
    auto si = (b-a).cross(S[i]-a);
    if (si == 0) continue;
    P m = ccCenter(a, b, S[i]);
    auto cr = (b-a).cross(m-a);
    if (si < 0) hi = min(hi, cr);
    else lo = max(lo, cr);
  }
  double v = (0 < lo ? lo : hi < 0 ? hi : 0);
  P c = (a + b) / 2 + (b - a).perp() * v / (b -
      ↪ a).dist2();
  return {(a - c).dist2(), c};
}
pair<double, P> mec(vector<P>& S, P a, int n) {
  random_shuffle(S.begin(), S.begin() + n);
  P b = S[0], c = (a + b) / 2;
  double r = (a - c).dist2();
  rep(i,1,n) if ((S[i] - c).dist2() > r * (1 +
      ↪ 1e-8)) {
    tie(r,c) = (n == sz(S) ?
      mec(S, S[i], i) : mec2(S, a, S[i], i));
  }
  return {r, c};
}
pair<double, P> enclosingCircle(vector<P> S) {
  assert(!S.empty()); auto r = mec(S, S[0], sz(S));
  return {sqrt(r.first), r.second};
}
```

2.4 Polygons

Hash: 9166df562235fd1a8e2b57516151bc4b

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-03-22
 * License: CC0
 * Source: Basic geometry
 * Description: Returns true if p lies within the
     ↪ polygon described by the points between
     ↪ iterators begin and end. If strict false is
     ↪ returned when p is on the edge of the
     ↪ polygon. Answer is calculated by counting
     ↪ the number of intersections between the
     ↪ polygon and a line going from p to infinity
     ↪ in the positive x-direction. The algorithm
     ↪ uses products in intermediate steps so watch
     ↪ out for overflow. If points within epsilon
     ↪ from an edge should be considered as on the
     ↪ edge replace the line "if (onSegment..."
     ↪ with the comment bellow it (this will cause
     ↪ overflow for int and long long).
 * Time: O(n)
 * Status: tested with unitTest and Kattis problems
     ↪ copsrobbers, pointinpolygon and intersection
 * Usage:
 *  typedef Point<int> pi;
 *  vector<pi> v; v.push_back(pi(4,4));
 *  v.push_back(pi(1,2)); v.push_back(pi(2,1));
 *  bool in = insidePolygon(v.begin(),v.end(),
     ↪ pi(3,4), false);
 */
#pragma once

#include "Point.h"
#include "onSegment.h"
#include "SegmentDistance.h"

template<class It, class P>
bool insidePolygon(It begin, It end, const P& p,
    bool strict = true) {
  int n = 0; //number of isects with line from p to
      ↪ (inf,p.y)
  for (It i = begin, j = end-1; i != end; j = i++) {
    //if p is on edge of polygon
    if (onSegment(*i, *j, p)) return !strict;
    //or: if (segDist(*i, *j, p) <= epsilon) return
        ↪ !strict;
    //increment n if segment intersects line from p
    n += (max(i->y,j->y) > p.y && min(i->y,j->y) <=
        ↪ p.y &&
        ((*j-*i).cross(p-*i) > 0) == (i->y <= p.y));
  }
  return n&1; //inside if odd number of intersections
}
```

Hash: 0ffc13e743306abe11a6f0ca5127a3a8

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-03-21
 * License: CC0
 * Source: tinyKACTL
 * Description: Returns twice the signed area of a
     ↪ polygon.
 *  Clockwise enumeration gives negative area. Watch
     ↪ out for overflow if using int as T!
 * Status: Tested with unitTest, Kattis problems
     ↪ polygonarea and wrapping and UVa Online
     ↪ Judge Problem: 109 - SCUD Busters
 */
#pragma once

#include "Point.h"

template<class T>
T polygonArea2(vector<Point<T>>& v) {
  T a = v.back().cross(v[0]);
  rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
  return a;
}
```

Hash: 4d8d9f5e2326931e6979b549d5778b57

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-04-08
 * License: CC0
 * Source:
 * Description: Returns the center of mass for a
     ↪ polygon.
 * Status: Tested
 */
#pragma once

#include "Point.h"

typedef Point<double> P;
Point<double> polygonCenter(vector<P>& v) {
  auto i = v.begin(), end = v.end(), j = end-1;
  Point<double> res{0,0}; double A = 0;
  for (; i != end; j=i++) {
    res = res + (*i + *j) * j->cross(*i);
    A += j->cross(*i);
  }
  return res / A / 3;
}
```

Hash: 46b2f2d7768681a44530a1599018a8f3

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-03-21
 * License: CC0
 * Source:
 * Description:\\
\begin{minipage}{75mm}
 Returns a vector with the vertices of a polygon
     ↪ with everything to the left of the line
     ↪ going from s to e cut away.
\end{minipage}
\begin{minipage}{15mm}
\vspace{-6mm}
\includegraphics[width=\textwidth]{../content/geometry/PolygonCut}
\vspace{-6mm}
\end{minipage}
 * Status: tested but not extensively
 * Usage:
 *  vector<P> p = ...;
 *  p = polygonCut(p, P(0,0), P(1,0));
 */
#pragma once
```

```
#include "Point.h"
#include "lineIntersection.h"

typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P
    ↪ e) {
  vector<P> res;
  rep(i,0,sz(poly)) {
    P cur = poly[i], prev = i ? poly[i-1] :
        ↪ poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0)) {
      res.emplace_back();
      lineIntersection(s, e, cur, prev, res.back());
    }
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

Hash: e02b6fcbabc24bc080c3aa6b99326058

```
/**
 * Author: Johan Sannemo
 * Date: 2017-04-16
 * License: CC0
 * Source: Basic algorithm knowledge
 * Description:
\\\begin{minipage}{75mm}
Returns a vector of indices of the convex hull in
    ↪ counter-clockwise order.
Points on the edge of the hull between two other
    ↪ points are not considered part of the hull.
\end{minipage}
\begin{minipage}{15mm}
\vspace{-6mm}
\includegraphics[width=\textwidth]{../content/geometry/ConvexHull}
\vspace{-6mm}
\end{minipage}
 * Status: tested with Kattis problems convexhull
 * Usage:
 *  vector<P> ps, hull;
 *  trav(i, convexHull(ps)) hull.push_back(ps[i]);
 * Time: O(n \log n)
*/
#pragma once

#include "Point.h"

typedef Point<ll> P;
pair<vi, vi> ulHull(const vector<P>& S) {
  vi Q(sz(S)), U, L;
  iota(all(Q), 0);
  sort(all(Q), [&S](int a, int b){ return S[a] <
      ↪ S[b]; });
  trav(it, Q) {
#define ADDP(C, cmp) while (sz(C) > 1 &&
    ↪ S[C[sz(C)-2]].cross(\
  S[it], S[C.back()]) cmp 0) C.pop_back();
      ↪ C.push_back(it);
    ADDP(U, <=); ADDP(L, >=);
  }
  return {U, L};
}
```

```
vi convexHull(const vector<P>& S) {
  vi u, l; tie(u, l) = ulHull(S);
  if (sz(S) <= 1) return u;
  if (S[u[0]] == S[u[1]]) return {0};
  l.insert(l.end(), u.rbegin()+1, u.rend()-1);
  return l;
}
```

Hash: 21dec6a1a9af89bdadde0e55957ab2ed

```
/**
 * Author: Johan Sannemo
 * Date: 2017-03-12
 * License: CC0
 * Source: Wikipedia
 * Description: Calculates the max squared distance
     ↪ of a set of points.
 * Status: Tested.
 */
#pragma once

#include "ConvexHull.h"

vector<pii> antipodal(const vector<P>& S, vi& U, vi&
    ↪ L) {
  vector<pii> ret;
  int i = 0, j = sz(L) - 1;
  while (i < sz(U) - 1 || j > 0) {
    ret.emplace_back(U[i], L[j]);
    if (j == 0 || (i != sz(U)-1 && (S[L[j]] -
        ↪ S[L[j-1]])
          .cross(S[U[i+1]] - S[U[i]]) > 0)) ++i;
    else --j;
  }
  return ret;
}

pii polygonDiameter(const vector<P>& S) {
  vi U, L; tie(U, L) = ulHull(S);
  pair<ll, pii> ans;
  trav(x, antipodal(S, U, L))
    ans = max(ans, {(S[x.first] -
        ↪ S[x.second]).dist2(), x});
  return ans.second;
}
```

Hash: d229b1e99f0fc864a73fb9a5e1827285

```
/**
 * Author: Johan Sannemo
 * Date: 2017-04-13
 * License: CC0
 * Source: Inspired by old, broken tinyKACTL
 * Description: Determine whether a point t lies
     ↪ inside a given polygon (counter-clockwise
     ↪ order).
 * The polygon must be such that every point on the
     ↪ circumference is visible from the first
     ↪ point in the vector.
 * It returns 0 for points outside, 1 for points on
     ↪ the circumference, and 2 for points inside.
 * Usage:
 * Status: Tested at Moscow ICPC pre-finals workshop
 * Time: O(\log N)
 */
#pragma once
```

```
#include "Point.h"
#include "sideOf.h"
#include "onSegment.h"

typedef Point<ll> P;
int insideHull2(const vector<P>& H, int L, int R,
     ↪ const P& p) {
  int len = R - L;
  if (len == 2) {
    int sa = sideOf(H[0], H[L], p);
    int sb = sideOf(H[L], H[L+1], p);
    int sc = sideOf(H[L+1], H[0], p);
    if (sa < 0 || sb < 0 || sc < 0) return 0;
    if (sb==0 || (sa==0 && L == 1) || (sc == 0 && R
         ↪ == sz(H)))
      return 1;
    return 2;
  }
  int mid = L + len / 2;
  if (sideOf(H[0], H[mid], p) >= 0)
    return insideHull2(H, mid, R, p);
  return insideHull2(H, L, mid+1, p);
}

int insideHull(const vector<P>& hull, const P& p) {
  if (sz(hull) < 3) return onSegment(hull[0],
       ↪ hull.back(), p);
  else return insideHull2(hull, 1, sz(hull), p);
}
```

Hash: ae4ed87510f957e220c91b63a065ee9a

```
/**
 * Author: Johan Sannemo
 * Date: 2017-05-15
 * License: CC0
 * Source: thin air
 * Description: Line-convex polygon intersection.
     ↪ The polygon must be ccw and have no colinear
     ↪ points.
 *  isct(a, b) returns a pair describing the
     ↪ intersection of a line with the polygon:
 *  \begin{itemize*}
 *    \item $(-1, -1)$ if no collision,
 *    \item $(i, -1)$ if touching the corner $i$,
 *    \item $(i, i)$ if along side $(i, i+1)$,
 *    \item $(i, j)$ if crossing sides $(i, i+1)$
     ↪ and $(j, j+1)$.
 *  \end{itemize*}
 *  In the last case, if a corner $i$ is crossed,
     ↪ this is treated as happening on side $(i,
     ↪ i+1)$.
 *  The points are returned in the same order as the
     ↪ line hits the polygon.
 * Status: fuzz-tested
 * Time: O(N + Q \log n)
 */
#pragma once

#include "Point.h"

ll sgn(ll a) { return (a > 0) - (a < 0); }
typedef Point<ll> P;
struct HullIntersection {
  int N;
  vector<P> p;
  vector<pair<P, int>> a;

  HullIntersection(const vector<P>& ps) : N(sz(ps)),
       ↪ p(ps) {
    p.insert(p.end(), all(ps));
    int b = 0;
    rep(i,1,N) if (P{p[i].y,p[i].x} < P{p[b].y,
         ↪ p[b].x}) b = i;
    rep(i,0,N) {
      int f = (i + b) % N;
      a.emplace_back(p[f+1] - p[f], f);
    }
  }

  int qd(P p) {
    return (p.y < 0) ? (p.x >= 0) + 2
         : (p.x <= 0) * (1 + (p.y <= 0));
  }

  int bs(P dir) {
    int lo = -1, hi = N;
    while (hi - lo > 1) {
      int mid = (lo + hi) / 2;
      if (make_pair(qd(dir), dir.y * a[mid].first.x)
           ↪ <
        make_pair(qd(a[mid].first), dir.x *
             ↪ a[mid].first.y))
        hi = mid;
      else lo = mid;
    }
    return a[hi%N].second;
  }

  bool isign(P a, P b, int x, int y, int s) {
    return sgn(a.cross(p[x], b)) * sgn(a.cross(p[y],
         ↪ b)) == s;
  }

  int bs2(int lo, int hi, P a, P b) {
    int L = lo;
    if (hi < lo) hi += N;
    while (hi - lo > 1) {
      int mid = (lo + hi) / 2;
      if (isign(a, b, mid, L, -1)) hi = mid;
      else lo = mid;
    }
    return lo;
  }

  pii isct(P a, P b) {
    int f = bs(a - b), j = bs(b - a);
    if (isign(a, b, f, j, 1)) return {-1, -1};
    int x = bs2(f, j, a, b)%N,
        y = bs2(j, f, a, b)%N;
    if (a.cross(p[x], b) == 0 &&
        a.cross(p[x+1], b) == 0) return {x, x};
    if (a.cross(p[y], b) == 0 &&
        a.cross(p[y+1], b) == 0) return {y, y};
    if (a.cross(p[f], b) == 0) return {f, -1};
    if (a.cross(p[j], b) == 0) return {j, -1};
    return {x, y};
  }
};
```

2.5 Misc. Point Set Problems

Hash: e5aa7d9a4c0334e0f3550648d48e9c48

```
/**
 * Author: Per Austrin, Max Bennedich, Gunnar Kreitz
 * Date: 2004-03-15
 * Description: $i1$, $i2$ are the indices to the
     ↪ closest pair of points in the point vector
     ↪ $p$ after the call. The distance is returned.
 * Time: O(n \log n)
 */
#pragma once

#include "Point.h"

template<class It>
bool it_less(const It& i, const It& j) { return *i <
     ↪ *j; }
template<class It>
bool y_it_less(const It& i,const It& j) {return i->y
     ↪ < j->y;}

template<class It, class IIt> /* IIt =
     ↪ vector<It>::iterator */
double cp_sub(IIt ya, IIt yaend, IIt xa, It &i1, It
     ↪ &i2) {
  typedef typename iterator_traits<It>::value_type P;
  int n = yaend-ya, split = n/2;
  if(n <= 3) { // base case
    double a = (*xa[1]-*xa[0]).dist(), b = 1e50, c =
         ↪ 1e50;
    if(n==3) b=(*xa[2]-*xa[0]).dist(),
         ↪ c=(*xa[2]-*xa[1]).dist();
    if(a <= b) { i1 = xa[1];
      if(a <= c) return i2 = xa[0], a;
      else return i2 = xa[2], c;
    } else { i1 = xa[2];
      if(b <= c) return i2 = xa[0], b;
      else return i2 = xa[1], c;
  } }
  vector<It> ly, ry, stripy;
  P splitp = *xa[split];
  double splitx = splitp.x;
  for(IIt i = ya; i != yaend; ++i) { // Divide
    if(*i != xa[split] && (**i-splitp).dist2() <
         ↪ 1e-12)
      return i1 = *i, i2 = xa[split], 0;// nasty
           ↪ special case!
    if (**i < splitp) ly.push_back(*i);
    else ry.push_back(*i);
  } // assert((signed)lefty.size() == split)
  It j1, j2; // Conquer
  double a = cp_sub(ly.begin(), ly.end(), xa, i1,
       ↪ i2);
  double b = cp_sub(ry.begin(), ry.end(), xa+split,
       ↪ j1, j2);
  if(b < a) a = b, i1 = j1, i2 = j2;
  double a2 = a*a;
  for(IIt i = ya; i != yaend; ++i) { // Create strip
       ↪ (y-sorted)
    double x = (*i)->x;
    if(x >= splitx-a && x <= splitx+a)
         ↪ stripy.push_back(*i);
  }
  for(IIt i = stripy.begin(); i != stripy.end();
      ↪ ++i) {
    const P &p1 = **i;
    for(IIt j = i+1; j != stripy.end(); ++j) {
      const P &p2 = **j;
      if(p2.y-p1.y > a) break;
      double d2 = (p2-p1).dist2();
      if(d2 < a2) i1 = *i, i2 = *j, a2 = d2;
  } }
  return sqrt(a2);
}

template<class It> // It is random access iterators
    ↪ of point<T>
double closestpair(It begin, It end, It &i1, It &i2
    ↪ ) {
  vector<It> xa, ya;
  assert(end-begin >= 2);
  for (It i = begin; i != end; ++i)
    xa.push_back(i), ya.push_back(i);
  sort(xa.begin(), xa.end(), it_less<It>);
  sort(ya.begin(), ya.end(), y_it_less<It>);
  return cp_sub(ya.begin(), ya.end(), xa.begin(),
      ↪ i1, i2);
}
```

Hash: 463d8c628ce496a146cea3bd9137b644

```
/**
 * Author: Stanford
 * Date: Unknown
 * Source: Stanford Notebook
 * Description: KD-tree (2d, can be extended to 3d)
 * Status: Untested, but works for Stanford
 */
#pragma once

#include "Point.h"

typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();

bool on_x(const P& a, const P& b) { return a.x <
    ↪ b.x; }
bool on_y(const P& a, const P& b) { return a.y <
    ↪ b.y; }

struct Node {
  P pt; // if this is a leaf, the single point in it
  T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF; //
      ↪ bounds
  Node *first = 0, *second = 0;

  T distance(const P& p) { // min squared distance
      ↪ to a point
    T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
    T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
    return (P(x,y) - p).dist2();
  }

  Node(vector<P>&& vp) : pt(vp[0]) {
    for (P p : vp) {
      x0 = min(x0, p.x); x1 = max(x1, p.x);
      y0 = min(y0, p.y); y1 = max(y1, p.y);
    }
    if (vp.size() > 1) {
      // split on x if the box is wider than high
          ↪ (not best heuristic...)
      sort(all(vp), x1 - x0 >= y1 - y0 ? on_x :
          ↪ on_y);
      // divide by taking half the array for each
          ↪ child (not
      // best performance with many duplicates in
          ↪ the middle)
      int half = sz(vp)/2;
      first = new Node({vp.begin(), vp.begin() +
          ↪ half});
      second = new Node({vp.begin() + half,
          ↪ vp.end()});
    }
  }
};

struct KDTree {
  Node* root;
  KDTree(const vector<P>& vp) : root(new
      ↪ Node({all(vp)})) {}

  pair<T, P> search(Node *node, const P& p) {
    if (!node->first) {
      // uncomment if we should not find the point
          ↪ itself:
      // if (p == node->pt) return {INF, P()};
      return make_pair((p - node->pt).dist2(),
          ↪ node->pt);
    }

    Node *f = node->first, *s = node->second;
    T bfirst = f->distance(p), bsec = s->distance(p);
    if (bfirst > bsec) swap(bsec, bfirst), swap(f,
        ↪ s);

    // search closest side first, other side if
        ↪ needed
    auto best = search(f, p);
    if (bsec < best.first)
      best = min(best, search(s, p));
    return best;
  }

  // find nearest point to a point, and its squared
      ↪ distance
  // (requires an arbitrary operator< for Point)
  pair<T, P> nearest(const P& p) {
    return search(root, p);
  }
};
```

Hash: f9892ba5e448c002a0848d2a43695531

```
/**
 * Author: Mattias de Zalenski
 * Date: Unknown
 * Source: Geometry in C
 * Description: Computes the Delaunay triangulation
     ↪ of a set of points.
 *  Each circumcircle contains none of the input
     ↪ points.
 *  If any three points are colinear or any four are
     ↪ on the same circle, behavior is undefined.
 * Status: fuzz-tested
 * Time: O(n^2)
 */
#pragma once

#include "Point.h"
#include "3dHull.h"

template<class P, class F>
void delaunay(vector<P>& ps, F trifun) {
  if (sz(ps) == 3) { int d = (ps[0].cross(ps[1],
      ↪ ps[2]) < 0);
    trifun(0,1+d,2-d); }
  vector<P3> p3;
  trav(p, ps) p3.emplace_back(p.x, p.y, p.dist2());
  if (sz(ps) > 3) trav(t, hull3d(p3)) if
      ↪ ((p3[t.b]-p3[t.a]).
      cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
    trifun(t.a, t.c, t.b);
}
```

Hash: c2be574e451fd67675dee24b6d367b9a

```
/**
 * Author: Philippe Legault
 * Date: 2016
 * License: MIT
 * Source:
     ↪ https://github.com/Bathlamos/delaunay-triangulation/
 * Description: Fast Delaunay triangulation. There
     ↪ must be no duplicate points.
 * If all points are on a line, no triangles will be
     ↪ returned.
 * Should work for doubles as well, though there may
     ↪ be precision issues in 'circ'.
 * Returns triangles in order \{t[0][0], t[0][1],
     ↪ t[0][2], t[1][0], \dots\}, all
     ↪ counter-clockwise.
 * Time: O(n \log n)
 * Status: fuzz-tested
 */
#pragma once

#include "Point.h"

typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are
    ↪ < 2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any
    ↪ other point

struct Quad {
  bool mark; Q o, rot; P p;
  P F() { return r()->p; }
  Q r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return rot->r()->o->rot; }
};

bool circ(P p, P a, P b, P c) { // is p in the
    ↪ circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
      B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A +
      ↪ p.cross(c,a)*B > 0;
}
Q makeEdge(P orig, P dest) {
  Q q0 = new Quad{0,0,0,orig}, q1 = new
      ↪ Quad{0,0,0,arb},
    q2 = new Quad{0,0,0,dest}, q3 = new
       ↪ Quad{0,0,0,arb};
  q0->o = q0; q2->o = q2; // 0-0, 2-2
  q1->o = q3; q3->o = q1; // 1-3, 3-1
  q0->rot = q1; q1->rot = q2;
  q2->rot = q3; q3->rot = q0;
  return q0;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

pair<Q,Q> rec(const vector<P>& s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1],
       ↪ s.back());
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c :
       ↪ b->r() };
  }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = (sz(s) + 1) / 2;
  tie(ra, A) = rec({s.begin(), s.begin() + half});
  tie(B, rb) = rec({s.begin() + half, s.end()});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if
   ↪ (valid(e)) \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC),
       ↪ H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}

vector<P> triangulate(vector<P> pts) {
  sort(all(pts));  assert(unique(all(pts)) ==
     ↪ pts.end());
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1;
     ↪ pts.push_back(c->p); \
  q.push_back(c->r()); c = c->next(); } while (c !=
     ↪ e); }
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

2.6 3D

Hash: 1df80d4f5abfe37b44a15dd292e1f52f

```
/**
 * Author: Mattias de Zalenski
 * Date: 2002-11-04
 * Description: Magic formula for the volume of a
     ↪ polyhedron. Faces should point outwards.
 * Status: tested
 */
#pragma once

template<class V, class L>
double signed_poly_volume(const V& p, const L&
     ↪ trilist) {
  double v = 0;
  trav(i, trilist) v +=
     ↪ p[i.a].cross(p[i.b]).dot(p[i.c]);
  return v / 6;
}
```

Hash: 9ab11b67b89d035c3a72b683aed49177

```
/**
 * Author: Ulf Lundstrom with inspiration from
     ↪ tinyKACTL
 * Date: 2009-04-14
 * License: CC0
 * Source:
 * Description: Class to handle points in 3D space.
 *  T can be e.g. double or long long.
 * Usage:
 * Status: tested, except for phi and theta
 */
#pragma once

template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x),
     ↪ y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z); }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z); }
  P operator+(R p) const { return P(x+p.x, y+p.y,
     ↪ z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y,
     ↪ z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P operator/(T d) const { return P(x/d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y -
       ↪ y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return
     ↪ sqrt((double)dist2()); }
  //Azimuthal angle (longitude) to x-axis in
     ↪ interval [-pi, pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in
     ↪ interval [0, pi]
  double theta() const { return
     ↪ atan2(sqrt(x*x+y*y),z); }
  P unit() const { return *this/(T)dist(); } //makes
     ↪ dist()=1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit(); }
  //returns point rotated 'angle' radians ccw around
     ↪ axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u =
       ↪ axis.unit();
    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
  }
};
```

Hash: 2775062277ac79f2f1bf4941e753aa13

```
/**
 * Author: Johan Sannemo
 * Date: 2017-04-18
 * Source: derived from
     ↪ https://gist.github.com/msg555/4963794 by
     ↪ Mark Gordon
 * Description: Computes all faces of the
     ↪ 3-dimension hull of a point set.
 *  *No four points must be coplanar*, or else
     ↪ random results will be returned.
 *  All faces will point outwards.
 * Time: O(n^2)
 * Status: tested on SPOJ CH3D
 */
#pragma once

#include "Point3D.h"

typedef Point3D<double> P3;

struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};

struct F { P3 q; int a, b, c; };

vector<F> hull3d(const vector<P3>& A) {
  assert(sz(A) >= 4);
  vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1,
     ↪ -1}));
#define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i]))
      q = q * -1;
    F f{q, i, j, k};
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
    FS.push_back(f);
  };
  rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);

  rep(i,4,sz(A)) {
    rep(j,0,sz(FS)) {
      F f = FS[j];
      if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a,b).rem(f.c);
        E(a,c).rem(f.b);
        E(b,c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = sz(FS);
    rep(j,0,nw) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a,
    ↪ f.b, i, f.c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  trav(it, FS) if ((A[it.b] - A[it.a]).cross(
    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c,
        ↪ it.b);
  return FS;
};
```

Hash: 853b3abbe225b4a8e7217c00d13079ef

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-04-07
 * License: CC0
 * Source: My geometric reasoning
 * Description: Returns the shortest distance on the
     ↪ sphere with radius radius between the points
     ↪ with azimuthal angles (longitude) f1
     ↪ ($\phi_1$) and f2 ($\phi_2$) from x axis and
     ↪ zenith angles (latitude) t1 ($\theta_1$) and
     ↪ t2 ($\theta_2$) from z axis. All angles
     ↪ measured in radians. The algorithm starts by
     ↪ converting the spherical coordinates to
     ↪ cartesian coordinates so if that is what you
     ↪ have you can use only the two last rows.
     ↪ dx*radius is then the difference between the
     ↪ two points in the x direction and d*radius
     ↪ is the total distance between the points.
 * Status: somewhat tested locally
 tested with Kattis problem airlinehub
 to be tested with UVa 535
 */
#pragma once

double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}
```

3 Data Structure

Hash: d01d504b98495ea3340d6d2f6a6c6ffd

```
///////////////////////////
//
// LCT
//
///////////////////////////
struct T {
  bool rr;
  T *son[2], *pf, *fa;
} f1[N], *ff = f1, *f[N], *null;

void downdate(T *x) {
  if (x -> rr) {
    x -> son[0] -> rr = !x -> son[0] -> rr;
    x -> son[1] -> rr = !x -> son[1] -> rr;
    swap(x -> son[0], x -> son[1]);
    x -> rr = false;
  }
  // add stuff
}

void update(T *x) {
  // add stuff
}

void rotate(T *x, bool t) {
  T *y = x -> fa, *z = y -> fa;
  if (z != null)  z -> son[z -> son[1] == y] = x;
  x -> fa = z;
  y -> son[t] = x -> son[!t];
  x -> son[!t] -> fa = y;
  x -> son[!t] = y;
  y -> fa = x;
  update(y);
}

void xiao(T *x) {
  if (x -> fa != null)  xiao(x -> fa), x -> pf = x
    ↪ -> fa -> pf;
  downdate(x);
}

void splay(T *x) {
  xiao(x);
  T *y, *z;
  while (x -> fa != null) {
    y = x -> fa; z = y -> fa;
    bool t1 = (y -> son[1] == x), t2 = (z -> son[1]
      ↪ == y);
    if (z != null) {
      if (t1 == t2) rotate(y, t2), rotate(x, t1);
      else  rotate(x, t1), rotate(x, t2);
    }else rotate(x, t1);
  }
  update(x);
}

void access(T *x) {
  splay(x);
  x -> son[1] -> pf = x;
  x -> son[1] -> fa = null;
  x -> son[1] = null;
  update(x);
  while (x -> pf != null) {
    splay(x -> pf);
    x -> pf -> son[1] -> pf = x -> pf;
    x -> pf -> son[1] -> fa = null;
    x -> pf -> son[1] = x;
    x -> fa = x -> pf;
    splay(x);
  }
  x -> rr = true;
}

bool Cut(T *x, T *y) {
  access(x);
  access(y);
  downdate(y);
  downdate(x);
  if (y -> son[1] != x || x -> son[0] != null)
    return false;
  y -> son[1] = null;
  x -> fa = x -> pf = null;
  update(x);
  update(y);
  return true;
}

bool Connected(T *x, T *y) {
  access(x);
  access(y);
  return x == y || x -> fa != null;
}

bool Link(T *x, T *y) {
  if (Connected(x, y))
    return false;
  access(x);
  access(y);
  x -> pf = y;
  return true;
}


int main() {
  read(n); read(m); read(q);
  null = new T; null -> son[0] = null -> son[1] =
      ↪ null -> fa = null -> pf = null;
  for (int i = 1; i <= n; i++) {
    f[i] = ++ff;
    f[i] -> son[0] = f[i] -> son[1] = f[i] -> fa =
      ↪ f[i] -> pf = null;
    f[i] -> rr = false;
  }
  // init null and f[i]
}
```

4 String

Hash: f566cda2e4994762e460a8553dc79ff4

```
///////////////////////////
//
//SAM
//
///////////////////////////
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
int
    ↪ n,i,init,L,len,ll,q,h,ch,p,last[1700000],n1[1700000],du[1700000],s[1700000],fa[800001],l[1700000],son[1700000][3],par[1700000];
char S[8000001],k;
long long ans,sum[1600001];
void ins(int p,int ss,int k)
{
  int np=++len,q,nq;
  l[np]=l[p]+1;
  s[np]=1;
  while (p&&!son[p][k]) son[p][k]=np,p=par[p];
  if (!p) par[np]=1;
  else {
    q=son[p][k];
    if (l[p]+1==l[q]) par[np]=q;
    else {
      nq=++len;
      l[nq]=l[p]+1;
      s[nq]=0;
      memset(son[nq], son[q], sizeof son[q]);
      par[nq]=par[q];
      par[q]=nq;
      par[np]=nq;
      while (p&&son[p][k]==q) son[p][k]=nq,p=par[p];
    }
  }
  last[ss]=np;
}
int main()
{
  read(n);
  last[1]=init=len=1;
  for (i=2;i<=n;i++)
  {
    read(fa[i]);
    for (k=getchar();k<=32;k=getchar());
    ins(last[fa[i]],i,k-'a');
  }
}
```

5 Math

Hash: 82f0550573a34efcbcbaa2487bc74766

```
///////////////////////////
//SIMPLEX
//WARNING: segfaults on empty (size 0)
//max cx st Ax<=b, x>=0
//do 2 phases; 1st check feasibility;
//2nd check boundedness & ans
///////////////////////////
vector<double> simplex(vector<vector<double> > A,
    ↪ vector<double> b, vector<double> c) {
  int n = (int) A.size(), m = (int) A[0].size()+1, r
      ↪ = n, s = m-1;
  vector<vector<double> > D = vector<vector<double>
      ↪ > (n+2, vector<double>(m+1));
  vector<int> ix = vector<int> (n+m);
  for (int i=0; i<n+m; i++) ix[i] = i;
  for (int i=0; i<n; i++) {
    for (int j=0; j<m-1; j++)D[i][j]=-A[i][j];
    D[i][m-1] = 1;
    D[i][m] = b[i];
    if (D[r][m] > D[i][m]) r = i;
  }
  for (int j=0; j<m-1; j++) D[n][j]=c[j];
  D[n+1][m-1] = -1; int z = 0;
  for (double d;;) {
    if (r < n) {
      swap(ix[s], ix[r+m]);
      D[r][s] = 1.0/D[r][s];
      for (int j=0; j<=m; j++) if (j!=s) D[r][j] *=
          ↪ -D[r][s];
      for(int i=0; i<=n+1; i++)if(i!=r) {
        for (int j=0; j<=m; j++) if(j!=s) D[i][j] +=
            ↪ D[r][j] * D[i][s];
        D[i][s] *= D[r][s];
      }
    }
    r = -1; s = -1;
    for (int j=0; j <m; j++) if (s<0 || ix[s]>ix[j])
        ↪ {
      if (D[n+1][j]>eps || D[n+1][j]>-eps &&
          ↪ D[n][j]>eps) s = j;
    }
    if (s < 0) break;
    for (int i=0; i<n; i++) if(D[i][s]<-eps) {
      if (r < 0 || (d =
          ↪ D[r][m]/D[r][s]-D[i][m]/D[i][s]) < -eps
        || d < eps && ix[r+m] > ix[i+m]) r=i;
    }
    if (r < 0) return vector<double>(); // unbounded
  }
  if (D[n+1][m] < -eps) return vector<double>(); //
      ↪ infeasible
  vector<double> x(m-1);
  for (int i = m; i < n+m; i ++) if (ix[i] < m-1)
      ↪ x[ix[i]] = D[i-m][m];
  printf("%.2lf\n", D[n][m]);
  return x; // ans: D[n][m]
}
```

6 Graph

Hash: 56383bfdb29dfc826d0462e99b723479

```
//// Max clique N<64. Bit trick for speed
/**
* WishingBone's ACM/ICPC Routine Library
* maximum clique solver
*/
// clique solver calculates both size and
    ↪ consitution of maximum clique
// uses bit operation to accelerate searching
// graph size limit is 63, the graph should be
    ↪ undirected
// can optimize to calculate on each component, and
    ↪ sort on vertex degrees
// can be used to solve maximum independent set
class clique {
  public:
  static const long long ONE = 1;
  static const long long MASK = (1 << 21) - 1;
  char* bits;
  int n, size, cmax[63];
  long long mask[63], cons;
  // initiate lookup table
  clique() {
    bits = new char[1 << 21];
    bits[0] = 0;
    for (int i = 1; i < (1<<21); ++i)
      bits[i] = bits[i >> 1] + (i & 1);
  }
  ~clique() {
    delete bits;
  }
  // search routine
  bool search(int step,int siz,LL mor,LL con);
  // solve maximum clique and return size
  int sizeClique(vector<vector<int> >& mat);
  // solve maximum clique and return set
  vector<int>getClq(vector<vector<int> >&mat);
};
// step is node id, size is current sol., more is
    ↪ available mask, cons is constitution mask
bool clique::search(int step, int size,
          LL more, LL cons) {
  if (step >= n) {
    // a new solution reached
    this->size = size;
    this->cons = cons;
    return true;
  }
  long long now = ONE << step;
  if ((now & more) > 0) {
    long long next = more & mask[step];
    if (size + bits[next & MASK] +
        bits[(next >> 21) & MASK] +
        bits[next >> 42] >= this->size
     && size + cmax[step] > this->size) {
      // the current node is in the clique
      if (search(step+1,size+1,next,cons|now))
        return true;
    }
  }
  long long next = more & ~now;
  if (size + bits[next & MASK] +
      bits[(next >> 21) & MASK] +
      bits[next >> 42] > this->size) {
    // the current node is not in the clique
    if (search(step + 1, size, next, cons))
      return true;
  }
  return false;
}
// solve maximum clique and return size
int clique::sizeClique(vector<vector<int> >& mat) {
  n = mat.size();
  // generate mask vectors
  for (int i = 0; i < n; ++i) {
    mask[i] = 0;
    for (int j = 0; j < n; ++j)
      if (mat[i][j] > 0) mask[i] |= ONE << j;
  }
  size = 0;
  for (int i = n - 1; i >= 0; --i) {
    search(i + 1, 1, mask[i], ONE << i);
    cmax[i] = size;
  }
  return size;
}
// calls sizeClique and restore cons
vector<int> clique::getClq(
    vector<vector<int> >& mat) {
  sizeClique(mat);
  vector<int> ret;
  for (int i = 0; i < n; ++i)
    if ((cons&(ONE<<i)) > 0) ret.push_back(i);
  return ret;
}
```

Hash: cf425cb6a61a1641c2034d45b6b8f54a

```
#include <bits/stdc++.h>
using namespace std;
#define rep(i, n) for (int i = 0; i < n; i++)

#define N 110000
#define M 110000
#define inf 2000000000

struct edg {
    int u, v;
    int cost;
} E[M], E_copy[M];

int In[N], ID[N], vis[N], pre[N];

// edges pointed from root.
int Directed_MST(int root, int NV, int NE) {
  for (int i = 0; i < NE; i++)
    E_copy[i] = E[i];
    int ret = 0;
    int u, v;
    while (true) {
        rep(i, NV)   In[i] = inf;
        rep(i, NE) {
            u = E_copy[i].u;
            v = E_copy[i].v;
            if(E_copy[i].cost < In[v] && u != v) {
                In[v] = E_copy[i].cost;
                pre[v] = u;
            }
        }
        rep(i, NV) {
            if(i == root)   continue;
            if(In[i] == inf)    return -1; // no
                ↪ solution
        }

        int cnt = 0;
        rep(i, NV) {
          ID[i] = -1;
          vis[i] = -1;
        }
        In[root] = 0;
        rep(i, NV) {
            ret += In[i];
            int v = i;
            while(vis[v] != i && ID[v] == -1 && v !=
                ↪ root) {
                vis[v] = i;
                v = pre[v];
            }
            if(v != root && ID[v] == -1) {
                for(u = pre[v]; u != v; u = pre[u]) {
                    ID[u] = cnt;
                }
                ID[v] = cnt++;
            }
        }
        if(cnt == 0)    break;
        rep(i, NV) {
            if(ID[i] == -1) ID[i] = cnt++;
        }
        rep(i, NE) {
            v = E_copy[i].v;
            E_copy[i].u = ID[E_copy[i].u];
            E_copy[i].v = ID[E_copy[i].v];
            if(E_copy[i].u != E_copy[i].v) {
                E_copy[i].cost -= In[v];
            }
        }
        NV = cnt;
        root = ID[root];
    }
    return ret;
}
```

Hash: eb72f852273569a543ecd429ed57dc9d

```
///////////////////////////
//
// dominator tree
//
///////////////////////////

#define N 110000 //max number of vertices

vector<int> succ[N], prod[N], bucket[N], dom_t[N];
int semi[N], anc[N], idom[N], best[N], fa[N],
    ↪ tmp_idom[N];
int dfn[N], redfn[N];
int child[N], size[N];
int timestamp;

void dfs(int now) {
  dfn[now] = ++timestamp;
  redfn[timestamp] = now;
  anc[timestamp] = idom[timestamp] =
      ↪ child[timestamp] = size[timestamp] = 0;
  semi[timestamp] = best[timestamp] = timestamp;
  int sz = succ[now].size();
  for(int i = 0; i < sz; ++i) {
    if(dfn[succ[now][i]] == -1) {
      dfs(succ[now][i]);
      fa[dfn[succ[now][i]]] = dfn[now];
    }
    prod[dfn[succ[now][i]]].push_back(dfn[now]);
  }
}

void compress(int now) {
  if(anc[anc[now]] != 0) {
    compress(anc[now]);
    if(semi[best[now]] > semi[best[anc[now]]])
      best[now] = best[anc[now]];
    anc[now] = anc[anc[now]];
  }
}

inline int eval(int now) {
  if(anc[now] == 0)
    return now;
  else {
    compress(now);
    return semi[best[anc[now]]] >= semi[best[now]] ?
        ↪ best[now]
      : best[anc[now]];
  }
}

inline void link(int v, int w) {
  int s = w;
  while(semi[best[w]] < semi[best[child[w]]]) {
    if(size[s] + size[child[child[s]]] >=
        ↪ 2*size[child[s]]) {
      anc[child[s]] = s;
      child[s] = child[child[s]];
    } else {
      size[child[s]] = size[s];
      s = anc[s] = child[s];
    }
  }
  best[s] = best[w];
  size[v] += size[w];
  if(size[v] < 2*size[w])
    swap(s, child[v]);
  while(s != 0) {
    anc[s] = v;
    s = child[s];
  }
}

// idom[n] and other vertices that cannot be reached
    ↪ from n will be 0
void lengauer_tarjan(int n) { // n is the root's
    ↪ number
  memset(dfn, -1, sizeof dfn);
  memset(fa, -1, sizeof fa);
  timestamp = 0;
  dfs(n);
  fa[1] = 0;
  for(int w = timestamp; w > 1; --w) {
    int sz = prod[w].size();
    for(int i = 0; i < sz; ++i) {
      int u = eval(prod[w][i]);
      if(semi[w] > semi[u])
        semi[w] = semi[u];
    }
    bucket[semi[w]].push_back(w);
    //anc[w] = fa[w]; link operation for o(mlogm)
        ↪ version
                link(fa[w], w);
    if(fa[w] == 0)
      continue;
    sz = bucket[fa[w]].size();
    for(int i = 0; i < sz; ++i) {
      int u = eval(bucket[fa[w]][i]);
      if(semi[u] < fa[w])
        idom[bucket[fa[w]][i]] = u;
      else
        idom[bucket[fa[w]][i]] = fa[w];
    }
    bucket[fa[w]].clear();
  }
  for(int w = 2; w <= timestamp; ++w) {
    if(idom[w] != semi[w])
      idom[w] = idom[idom[w]];
  }
  idom[1] = 0;
  for(int i = timestamp; i > 1; --i) {
    if(fa[i] == -1)
      continue;
    dom_t[idom[i]].push_back(i);
  }
  memset(tmp_idom, 0, sizeof tmp_idom);
  for (int i = 1; i <= timestamp; i++)
    tmp_idom[redfn[i]] = redfn[idom[i]];
  memcpy(idom, tmp_idom, sizeof idom);
}
```

7 Java/Python

7.1 Java IO

7.2 Java BigInteger

7.3 Python IO