```
#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>
using namespace std;

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) // important constant; alternative #define PI (2.0
* acos(0.0))

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

// struct point_i { int x, y; };    // basic raw form, minimalist mode
struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                         // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };         // user-defined

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                         // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}        // user-defined
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                   // useful for sorting
      return x < other.x;          // first criteria , by x-coordinate
    return y < other.y; }          // second criteria, by y-coordinate
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) {                  // Euclidean distance
                      // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    // multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }

struct line { double a, b, c; };          // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;           // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

// not needed since we will use the more robust form: ax + by + c = 0
(see above)
struct line2 { double m, c; };      // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
 if (abs(p1.x - p2.x) < EPS) {           // special case: vertical line
   l.m = INF;                    // l contains m = INF and c = x_value
```

```
    l.c = p1.x;                 // to denote vertical line x = x_value
    return 0;   // we need this return variable to differentiate result
  }
  else {
    l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
    l.c = p1.y - l.m * p1.x;
    return 1;     // l contains m and c of the line equation y = mx + c
} }

bool areParallel(line l1, line l2) {       // check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {           // also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;            // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

struct vec { double x, y;  // name: `vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;                                              // always -m
  l.b = 1;                                                // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }                // compute this

void closestPoint(line l, point p, point &ans) {
  line perpendicular;         // perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {              // special case 1: vertical line
    ans.x = -(l.c);   ans.y = p.y;      return; }

  if (fabs(l.a) < EPS) {            // special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);   return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);          // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
  point b;
  closestPoint(l, p, b);                     // similar to distToLine
```

```
  vec v = toVec(p, b);                              // create a vector
  ans = translate(translate(p, v), v); }         // translate p twice

double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));                   // translate a to c
  return dist(p, c); }           // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);                    // closer to a
    return dist(p, a); }         // Euclidean distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);                    // closer to b
    return dist(p, b); }         // Euclidean distance between p and b
  return distToLine(p, a, b, c); }          // run distToLine as above

double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

//// another variant
//int area2(point p, point q, point r) { // returns 'twice' the area of
this triangle A-B-c
//  return p.x * q.y - p.y * q.x +
//         q.x * r.y - q.y * r.x +
//         r.x * p.y - r.y * p.x;
//}

// note: to accept collinear points, we have to change the `> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

int main() {
  point P1, P2, P3(0, 1); // note that both P1 and P2 are (0.00, 0.00)
  printf("%d\n", P1 == P2);                                    // true
  printf("%d\n", P1 == P3);                                   // false

  vector<point> P;
  P.push_back(point(2, 2));
  P.push_back(point(4, 3));
```

```
  P.push_back(point(2, 4));
  P.push_back(point(6, 6));
  P.push_back(point(2, 6));
  P.push_back(point(6, 5));

  // sorting points demo
  sort(P.begin(), P.end());
  for (int i = 0; i < (int)P.size(); i++)
    printf("(%.2lf, %.2lf)\n", P[i].x, P[i].y);

  // rearrange the points as shown in the diagram below
  P.clear();
  P.push_back(point(2, 2));
  P.push_back(point(4, 3));
  P.push_back(point(2, 4));
  P.push_back(point(6, 6));
  P.push_back(point(2, 6));
  P.push_back(point(6, 5));
  P.push_back(point(8, 6));

  /*
  // the positions of these 7 points (0-based indexing)
  6   P4      P3  P6
  5           P5
  4   P2
  3       P1
  2   P0
  1
  0 1 2 3 4 5 6 7 8
  */

  double d = dist(P[0], P[5]);
  printf("Euclidean distance between P[0] and P[5] = %.2lf\n", d); //
should be 5.000

  // line equations
  line l1, l2, l3, l4;
  pointsToLine(P[0], P[1], l1);
  printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l1.a, l1.b, l1.c); //
should be -0.50 * x + 1.00 * y - 1.00 = 0.00

  pointsToLine(P[0], P[2], l2); // a vertical line, not a problem in "ax
+ by + c = 0" representation
  printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l2.a, l2.b, l2.c); //
should be 1.00 * x + 0.00 * y - 2.00 = 0.00

  // parallel, same, and line intersection tests
  pointsToLine(P[2], P[3], l3);
  printf("l1 & l2 are parallel? %d\n", areParallel(l1, l2)); // no
  printf("l1 & l3 are parallel? %d\n", areParallel(l1, l3)); // yes, l1
(P[0]-P[1]) and l3 (P[2]-P[3]) are parallel

  pointsToLine(P[2], P[4], l4);
  printf("l1 & l2 are the same? %d\n", areSame(l1, l2)); // no
  printf("l2 & l4 are the same? %d\n", areSame(l2, l4)); // yes, l2
(P[0]-P[2]) and l4 (P[2]-P[4]) are the same line (note, they are two
different line segments, but same line)

  point p12;
```

```
  bool res = areIntersect(l1, l2, p12); // yes, l1 (P[0]-P[1]) and l2
(P[0]-P[2]) are intersect at (2.0, 2.0)
  printf("l1 & l2 are intersect? %d, at (%.2lf, %.2lf)\n", res, p12.x,
p12.y);

  // other distances
  point ans;
  d = distToLine(P[0], P[2], P[3], ans);
  printf("Closest point from P[0] to line         (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, d);
  closestPoint(l3, P[0], ans);
  printf("Closest point from P[0] to line V2      (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, dist(P[0], ans));

  d = distToLineSegment(P[0], P[2], P[3], ans);
  printf("Closest point from P[0] to line SEGMENT (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to A (or P[2]) =
(2.00, 4.00)
  d = distToLineSegment(P[1], P[2], P[3], ans);
  printf("Closest point from P[1] to line SEGMENT (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to midway between AB
= (3.20, 4.60)
  d = distToLineSegment(P[6], P[2], P[3], ans);
  printf("Closest point from P[6] to line SEGMENT (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to B (or P[3]) =
(6.00, 6.00)

  reflectionPoint(l4, P[1], ans);
  printf("Reflection point from P[1] to line      (P[2]-P[4]): (%.2lf,
%.2lf)\n", ans.x, ans.y); // should be (0.00, 3.00)

  printf("Angle P[0]-P[4]-P[3] = %.2lf\n", RAD_to_DEG(angle(P[0], P[4],
P[3]))); // 90 degrees
  printf("Angle P[0]-P[2]-P[1] = %.2lf\n", RAD_to_DEG(angle(P[0], P[2],
P[1]))); // 63.43 degrees
  printf("Angle P[4]-P[3]-P[6] = %.2lf\n", RAD_to_DEG(angle(P[4], P[3],
P[6]))); // 180 degrees

  printf("P[0], P[2], P[3] form A left turn? %d\n", ccw(P[0], P[2],
P[3])); // no
  printf("P[0], P[3], P[2] form A left turn? %d\n", ccw(P[0], P[3],
P[2])); // yes

  printf("P[0], P[2], P[3] are collinear? %d\n", collinear(P[0], P[2],
P[3])); // no
  printf("P[0], P[2], P[4] are collinear? %d\n", collinear(P[0], P[2],
P[4])); // yes

  point p(3, 7), q(11, 13), r(35, 30); // collinear if r(35, 31)
  printf("r is on the %s of line p-r\n", ccw(p, q, r) ? "left" :
"right"); // right

  /*
  // the positions of these 6 points
     E<--  4
           3       B D<--
           2   A C
           1
  -4-3-2-1 0 1 2 3 4 5 6
          -1
```

```
          -2
    F<--    -3
  */

  // translation
  point A(2.0, 2.0);
  point B(4.0, 3.0);
  vec v = toVec(A, B); // imagine there is an arrow from A to B (see the
diagram above)
  point C(3.0, 2.0);
  point D = translate(C, v); // D will be located in coordinate (3.0 +
2.0, 2.0 + 1.0) = (5.0, 3.0)
  printf("D = (%.2lf, %.2lf)\n", D.x, D.y);
  point E = translate(C, scale(v, 0.5)); // E will be located in
coordinate (3.0 + 1/2 * 2.0, 2.0 + 1/2 * 1.0) = (4.0, 2.5)
  printf("E = (%.2lf, %.2lf)\n", E.x, E.y);

  // rotation
  printf("B = (%.2lf, %.2lf)\n", B.x, B.y); // B = (4.0, 3.0)
  point F = rotate(B, 90); // rotate B by 90 degrees COUNTER clockwise, F
= (-3.0, 4.0)
  printf("F = (%.2lf, %.2lf)\n", F.x, F.y);
  point G = rotate(B, 180); // rotate B by 180 degrees COUNTER clockwise,
G = (-4.0, -3.0)
  printf("G = (%.2lf, %.2lf)\n", G.x, G.y);

  return 0;
}
```

```
#include <cstdio>
#include <cmath>
using namespace std;

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                      // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };          // constructor

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                      // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };      // constructor

int insideCircle(point_i p, point_i c, int r) { // all integer version
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;             // all integer
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } //inside/border/outside

bool circle2PtsRad(point p1, point p2, double r, point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
              (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true; }         // to get the other center, reverse p1 and p2

int main() {
  // circle equation, inside, border, outside
  point_i pt(2, 2);
  int r = 7;
  point_i inside(8, 2);
  printf("%d\n", insideCircle(inside, pt, r));             // 0-inside
  point_i border(9, 2);
  printf("%d\n", insideCircle(border, pt, r));          // 1-at border
  point_i outside(10, 2);
  printf("%d\n", insideCircle(outside, pt, r));           // 2-outside

  double d = 2 * r;
  printf("Diameter = %.2lf\n", d);
  double c = PI * d;
  printf("Circumference (Perimeter) = %.2lf\n", c);
  double A = PI * r * r;
  printf("Area of circle = %.2lf\n", A);

  printf("Length of arc   (central angle = 60 degrees) = %.2lf\n", 60.0 /
360.0 * c);
  printf("Length of chord (central angle = 60 degrees) = %.2lf\n",
sqrt((2 * r * r) * (1 - cos(DEG_to_RAD(60.0)))));
  printf("Area of sector  (central angle = 60 degrees) = %.2lf\n", 60.0 /
360.0 * A);
```

```
  point p1;
  point p2(0.0, -1.0);
  point ans;
  circle2PtsRad(p1, p2, 2.0, ans);
  printf("One of the center is (%.2lf, %.2lf)\n", ans.x, ans.y);
  circle2PtsRad(p2, p1, 2.0, ans);     // we simply reverse p1 with p2
  printf("The other center  is (%.2lf, %.2lf)\n", ans.x, ans.y);

  return 0;
}
```

```
#include <cstdio>
#include <cmath>
using namespace std;

#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                        // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };          // constructor

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                        // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };      // constructor

double dist(point p1, point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y); }

double perimeter(double ab, double bc, double ca) {
  return ab + bc + ca; }

double perimeter(point a, point b, point c) {
  return dist(a, b) + dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {
  // Heron's formula, split sqrt(a * b) into sqrt(a) * sqrt(b); in
implementation
  double s = 0.5 * perimeter(ab, bc, ca);
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) * sqrt(s - ca); }

double area(point a, point b, point c) {
  return area(dist(a, b), dist(b, c), dist(c, a)); }

//====================================================================
// from ch7_01_points_lines
struct line { double a, b, c; }; // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;           // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

bool areParallel(line l1, line l2) {        // check coefficient a + b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;           // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
```

```
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

struct vec { double x, y;  // name: `vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }
//====================================================================

double rInCircle(double ab, double bc, double ca) {
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                   // no inCircle center

  line l1, l2;                    // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);           // get their intersection point
  return 1; }

double rCircumCircle(double ab, double bc, double ca) {
  return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
// returns 1 if there is a circumCenter center, returns 0 otherwise
// if this function returns 1, ctr will be the circumCircle center
// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3, point &ctr, double &r){
  double a = p2.x - p1.x, b = p2.y - p1.y;
  double c = p3.x - p1.x, d = p3.y - p1.y;
  double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
  double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
  double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x - p2.x));
  if (fabs(g) < EPS) return 0;
```

```
  ctr.x = (d*e - b*f) / g;
  ctr.y = (a*f - c*e) / g;
  r = dist(p1, ctr);  // r = distance from center to 1 of the 3 points
  return 1; }

// returns true if point d is inside the circumCircle defined by a,b,c
int inCircumCircle(point a, point b, point c, point d) {
  return (a.x - d.x) * (b.y - d.y) * ((c.x - d.x) * (c.x - d.x) + (c.y -
d.y) * (c.y - d.y)) +
         (a.y - d.y) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y -
d.y)) * (c.x - d.x) +
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.x -
d.x) * (c.y - d.y) -
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.y -
d.y) * (c.x - d.x) -
         (a.y - d.y) * (b.x - d.x) * ((c.x - d.x) * (c.x - d.x) + (c.y -
d.y) * (c.y - d.y)) -
         (a.x - d.x) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y -
d.y)) * (c.y - d.y) > 0 ? 1 : 0;
}

bool canFormTriangle(double a, double b, double c) {
  return (a + b > c) && (a + c > b) && (b + c > a); }

int main() {
  double base = 4.0, h = 3.0;
  double A = 0.5 * base * h;
  printf("Area = %.2lf\n", A);

  point a;                                          // a right triangle
  point b(4.0, 0.0);
  point c(4.0, 3.0);

  double p = perimeter(a, b, c);
  double s = 0.5 * p;
  A = area(a, b, c);
  printf("Area = %.2lf\n", A);           // must be the same as above

  double r = rInCircle(a, b, c);
  printf("R1 (radius of incircle) = %.2lf\n", r);              // 1.00
  point ctr;
  int res = inCircle(a, b, c, ctr, r);
  printf("R1 (radius of incircle) = %.2lf\n", r);        // same, 1.00
  printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y);   // (3.00, 1.00)

  printf("R2 (radius of circumcircle) = %.2lf\n", rCircumCircle(a, b,
c)); // 2.50
  res = circumCircle(a, b, c, ctr, r);
  printf("R2 (radius of circumcircle) = %.2lf\n", r);   // same, 2.50
  printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y);   // (2.00, 1.50)

  point d(2.0, 1.0);               // inside triangle and circumCircle
  printf("d inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b,
c, d));
  point e(2.0, 3.9);   // outside the triangle but inside circumCircle
  printf("e inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b,
c, e));
  point f(2.0, -1.1);                               // slightly outside
```

```
  printf("f inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b,
c, f));

  // Law of Cosines
  double ab = dist(a, b);
  double bc = dist(b, c);
  double ca = dist(c, a);
  double alpha = RAD_to_DEG(acos((ca * ca + ab * ab - bc * bc) / (2.0 *
ca * ab)));
  printf("alpha = %.2lf\n", alpha);
  double beta  = RAD_to_DEG(acos((ab * ab + bc * bc - ca * ca) / (2.0 *
ab * bc)));
  printf("beta  = %.2lf\n", beta);
  double gamma = RAD_to_DEG(acos((bc * bc + ca * ca - ab * ab) / (2.0 *
bc * ca)));
  printf("gamma = %.2lf\n", gamma);

  // Law of Sines
  printf("%.2lf == %.2lf == %.2lf\n", bc / sin(DEG_to_RAD(alpha)), ca /
sin(DEG_to_RAD(beta)), ab / sin(DEG_to_RAD(gamma)));

  // Phytagorean Theorem
  printf("%.2lf^2 == %.2lf^2 + %.2lf^2\n", ca, ab, bc);

  // Triangle Inequality
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 5,
canFormTriangle(3, 4, 5)); // yes
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 7,
canFormTriangle(3, 4, 7)); // no, actually straight line
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 8,
canFormTriangle(3, 4, 8)); // no

  return 0;
}
```

```
#include <algorithm>
#include <cstdio>
#include <cmath>
#include <stack>
#include <vector>
using namespace std;

#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                      // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}        // user-defined
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

struct vec { double x, y;  // name: `vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

double dist(point p1, point p2) {                // Euclidean distance
  return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-
1]
    result += dist(P[i], P[i+1]);
  return result; }

// returns the area, which is half the determinant
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }

double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

// note: to accept collinear points, we have to change the `> 0'
// returns true if point r is on the left side of line pq
```

```
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

// returns true if we always make the same turn while examining
// all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;   // a point/sz=2 or a line/sz=3 is not
convex
  bool isLeft = ccw(P[0], P[1], P[2]);               // remember one
result
  for (int i = 1; i < sz-1; i++)            // then compare with the
others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;            // different sign -> this polygon is
concave
  return true; }                                   // this polygon is
convex

// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;    // assume the first vertex is equal to the last
vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
         sum += angle(P[i], pt, P[i+1]);                  // left
turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }                // right
turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v));
}

// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a,
Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);       // Q[i] is on the left of
ab
    if (left1 * left2 < -EPS)        // edge (Q[i], Q[i+1]) crosses line
ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
```

```
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());        // make P's first point = P's last
point
  return P; }

point pivot;
bool angleCmp(point a, point b) {                 // angle-sorting
function
  if (collinear(pivot, a, b))                               // special
case
    return dist(pivot, a) < dist(pivot, b);    // check which one is
closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two
angles

vector<point> CH(vector<point> P) {   // the content of P may be
reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner
case
    return P;                          // special case, the CH is P
itself
  }

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with
P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                    // use this global variable as
reference
  sort(++P.begin(), P.end(), angleCmp);              // we do not sort
P[0]

  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);   // initial
S
  i = 2;                                         // then, we check the
rest
  while (i < n) {           // note: N must be >= 3 for this method to
work
    j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);  // left turn,
accept
    else S.pop_back(); }   // or pop the top of S until we have a left
turn
  return S; }                                         // return the
result

int main() {
  // 6 points, entered in counter clockwise order, 0-based indexing
```

```
  vector<point> P;
  P.push_back(point(1, 1));
  P.push_back(point(3, 3));
  P.push_back(point(9, 1));
  P.push_back(point(12, 4));
  P.push_back(point(9, 7));
  P.push_back(point(1, 7));
  P.push_back(P[0]); // loop back

  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // 31.64
  printf("Area of polygon = %.2lf\n", area(P)); // 49.00
  printf("Is convex = %d\n", isConvex(P)); // false (P1 is the culprit)

  //// the positions of P6 and P7 w.r.t the polygon
  //7 P5--------------P4
  //6 |                  \
  //5 |                    \
  //4 |   P7                P3
  //3 |   P1___            /
  //2 | / P6    \ ___    /
  //1 P0              P2
  //0 1 2 3 4 5 6 7 8 9 101112

  point P6(3, 2); // outside this (concave) polygon
  printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); //
false
  point P7(3, 4); // inside this (concave) polygon
  printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); //
true

  // cutting the original polygon based on line P[2] -> P[4] (get the
left side)
  //7 P5--------------P4
  //6 |               |  \
  //5 |               |    \
  //4 |               |     P3
  //3 |   P1___       |    /
  //2 | /       \ ___ |  /
  //1 P0              P2
  //0 1 2 3 4 5 6 7 8 9 101112
  // new polygon (notice the index are different now):
  //7 P4--------------P3
  //6 |               |
  //5 |               |
  //4 |               |
  //3 |   P1___       |
  //2 | /       \ ___ |
  //1 P0              P2
  //0 1 2 3 4 5 6 7 8 9

  P = cutPolygon(P[2], P[4], P);
  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // smaller now
29.15
  printf("Area of polygon = %.2lf\n", area(P)); // 40.00

  // running convex hull of the resulting polygon (index changes again)
  //7 P3--------------P2
  //6 |               |
  //5 |               |
  //4 |   P7          |
```

```
  //3 |               |
  //2 |               |
  //1 P0--------------P1
  //0 1 2 3 4 5 6 7 8 9

  P = CH(P); // now this is a rectangle
  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // precisely
28.00
  printf("Area of polygon = %.2lf\n", area(P)); // precisely 48.00
  printf("Is convex = %d\n", isConvex(P)); // true
  printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); //
true
  printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); //
true

  return 0;
}
```

```
// 15-Puzzle Problem with IDA*

#include <algorithm>
#include <cstdio>
#include <map>
using namespace std;

#define INF 1000000000
#define ROW_SIZE 4 // ROW_SIZE is a matrix of 4 x 4
#define PUZZLE (ROW_SIZE*ROW_SIZE)
#define X 15

int p[PUZZLE];
int lim, nlim;
int dr[] = { 0,-1, 0, 1}; // E,N,W,S
int dc[] = { 1, 0,-1, 0}; // R,U,L,D
map<int, int> pred;
map<unsigned long long, int> vis;
char ans[] = "RULD";

inline int h1() { // heuristic: sum of Manhattan distances (compute all)
  int ans = 0;
  for (int i = 0; i < PUZZLE; i++) {
    int tgt_i = p[i] / 4, tgt_j = p[i] % 4;
    if (p[i] != X)
      ans += abs(i / 4 - tgt_i) + abs(i % 4 - tgt_j); // Manhattan
distance
  }
  return ans;
}

inline int h2(int i1, int j1, int i2, int j2) { // heuristic: sum of
manhattan distances (compute delta)
  int tgt_i = p[i2 * 4 + j2] / 4, tgt_j = p[i2 * 4 + j2] % 4;
  return -(abs(i2 - tgt_i) + abs(j2 - tgt_j)) + (abs(i1 - tgt_i) + abs(j1
- tgt_j));
}

inline bool goal() {
  for (int i = 0; i < PUZZLE; i++)
    if (p[i] != X && p[i] != i)
      return false;
  return true;
}

inline bool valid(int r, int c) {
  return 0 <= r && r < 4 && 0 <= c && c < 4;
}

inline void swap(int i, int j, int new_i, int new_j) {
  int temp = p[i * 4 + j];
  p[i * 4 + j] = p[new_i * 4 + new_j];
  p[new_i * 4 + new_j] = temp;
}

bool DFS(int g, int h) {
  if (g + h > lim) {
    nlim = min(nlim, g + h);
    return false;
  }
```

```
  if (goal())
    return true;

  unsigned long long state = 0;
  for (int i = 0; i < PUZZLE; i++) { // transform 16 numbers into 64
bits, exactly into ULL
    state <<= 4; // move left 4 bits
    state += p[i]; // add this digit (max 15 or 1111)
  }

  if (vis.count(state) && vis[state] <= g) // not pure backtracking...
this is to prevent cycling
    return false; // not good
  vis[state] = g; // mark this as visited

  int i, j, d, new_i, new_j;
  for (i = 0; i < PUZZLE; i++)
    if (p[i] == X)
      break;
  j = i % 4;
  i /= 4;

  for (d = 0; d < 4; d++) {
    new_i = i + dr[d]; new_j = j + dc[d];
    if (valid(new_i, new_j)) {
      int dh = h2(i, j, new_i, new_j);
      swap(i, j, new_i, new_j); // swap first
      pred[g + 1] = d;
      if (DFS(g + 1, h + dh)) // if ok, no need to restore, just go ahead
        return true;
      swap(i, j, new_i, new_j); // restore
    }
  }

  return false;
}

int IDA_Star() {
  lim = h1();
  while (true) {
    nlim = INF; // next limit
    pred.clear();
    vis.clear();
    if (DFS(0, h1()))
      return lim;
    if (nlim == INF)
      return -1;
    lim = nlim; // nlim > lim
    if (lim > 45) // pruning condition in the problem
      return -1;
  }
}

void output(int d) {
  if (d == 0)
    return;
  output(d - 1);
  printf("%c", ans[pred[d]]);
}
```

```
int main() {
#ifndef ONLINE_JUDGE
  freopen("in.txt", "r", stdin);
#endif

  int N;
  scanf("%d", &N);
  while (N--) {
    int i, j, blank = 0, sum = 0, ans = 0;
    for (i = 0; i < 4; i++)
      for (j = 0; j < 4; j++) {
        scanf("%d", &p[i * 4 + j]);
        if (p[i * 4 + j] == 0) {
          p[i * 4 + j] = X; // change to X (15)
          blank = i * 4 + j; // remember the index
        }
        else
          p[i * 4 + j]--; // use 0-based indexing
      }

    for (i = 0; i < PUZZLE; i++)
      for (j = 0; j < i; j++)
        if (p[i] != X && p[j] != X && p[j] > p[i])
          sum++;
    sum += blank / ROW_SIZE;

    if (sum % 2 != 0 && ((ans = IDA_Star()) != -1))
      output(ans), printf("\n");
    else
      printf("This puzzle is not solvable.\n");
  }

  return 0;
}
```

```
// Forming Quiz Teams

#include <algorithm>            // if you have problems with this C++
code,
#include <cmath>            // consult your programming text books
first...
#include <cstdio>
#include <cstring>
using namespace std;
        /* Forming Quiz Teams, the solution for UVa 10911 above */
          // using global variables is a bad software engineering
practice,
int N, target;                  // but it is OK for competitive
programming
double dist[20][20], memo[1 << 16];  // 1 << 16 = 2^16, note that max N =
8

double matching(int bitmask) {                        // DP state =
bitmask
                       // we initialize `memo' with -1 in the main
function
  if (memo[bitmask] > -0.5)          // this state has been computed
before
    return memo[bitmask];                  // simply lookup the memo
table
  if (bitmask == target)                // all students are already
matched
    return memo[bitmask] = 0;                              // the cost is
0

  double ans = 2000000000.0;               // initialize with a large
value
  int p1, p2;
  for (p1 = 0; p1 < 2 * N; p1++)
    if (!(bitmask & (1 << p1)))
      break;                                // find the first bit that is
off
  for (p2 = p1 + 1; p2 < 2 * N; p2++)              // then, try to match
p1
    if (!(bitmask & (1 << p2)))     // with another bit p2 that is also
off
      ans = min(ans,                                 // pick the
minimum
                dist[p1][p2] + matching(bitmask | (1 << p1) | (1 <<
p2)));

  return memo[bitmask] = ans;    // store result in a memo table and
return
}

int main() {
  int i, j, caseNo = 1, x[20], y[20];
  // freopen("10911.txt", "r", stdin);      // redirect input file to
stdin

  while (scanf("%d", &N), N) {                    // yes, we can do this
:)
    for (i = 0; i < 2 * N; i++)
      scanf("%*s %d %d", &x[i], &y[i]);                // '%*s' skips
names
```

```
    for (i = 0; i < 2 * N - 1; i++)         // build pairwise distance
table
      for (j = i + 1; j < 2 * N; j++)       // have you used `hypot'
before?
        dist[i][j] = dist[j][i] = hypot(x[i] - x[j], y[i] - y[j]);

    // use DP to solve min weighted perfect matching on small general
graph
    for (i = 0; i < (1 << 16); i++) memo[i] = -1.0;  // set -1 to all
cells
    target = (1 << (2 * N)) - 1;
    printf("Case %d: %.2lf\n", caseNo++, matching(0));
} } // return 0;
```

```
// ACORN, UVa 1231, LA 4106

#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

int main() {
  int i, j, c, t, h, f, a, n, acorn[2010][2010], dp[2010];

  scanf("%d", &c);
  while (c--) {
    scanf("%d %d %d", &t, &h, &f);
    memset(acorn, 0, sizeof acorn);
    for (i = 0; i < t; i++) {
      scanf("%d", &a);
      for (j = 0; j < a; j++) {
        scanf("%d", &n);
        acorn[i][n]++; // there is an acorn here
      }
    }

    for (int tree = 0; tree < t; tree++) // initialization
      dp[h] = max(dp[h], acorn[tree][h]);
    for (int height = h - 1; height >= 0; height--)
      for (int tree = 0; tree < t; tree++) {
        acorn[tree][height] +=
          max(acorn[tree][height + 1], // from this tree, +1 above
          ((height + f <= h) ? dp[height + f] : 0)); // best from tree at
height + f
        dp[height] = max(dp[height], acorn[tree][height]); // update this
too
      }
    printf("%d\n", dp[0]); // solution will be here
  }
  // ignore the last number 0

  return 0;
}
```

```
// World Finals Stockholm 2009, A - A Careful Approach, UVa 1079, LA 4445

#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

int i, n, caseNo = 1, order[8];
double a[8], b[8], L, maxL;

double greedyLanding() {  // with certain landing order, and certain L,
try
          // landing those planes and see what is the gap to b[order[n -
1]]
  double lastLanding = a[order[0]];      // greedy, 1st aircraft lands
ASAP
  for (i = 1; i < n; i++) {                      // for the other
aircrafts
    double targetLandingTime = lastLanding + L;
    if (targetLandingTime <= b[order[i]])
       // can land: greedily choose max of a[order[i]] or
targetLandingTime
      lastLanding = max(a[order[i]], targetLandingTime);
    else
      return 1;
  }
  // return +ve value to force binary search to reduce L
  // return -ve value to force binary search to increase L
  return lastLanding - b[order[n - 1]];
}

int main() {
  while (scanf("%d", &n), n) {                            // 2 <= n <=
8
    for (i = 0; i < n; i++) {   // plane i land safely at interval [ai,
bi]
      scanf("%lf %lf", &a[i], &b[i]);
      a[i] *= 60; b[i] *= 60;  // originally in minutes, convert to
seconds
      order[i] = i;
    }

    maxL = -1.0;                               // variable to be searched
for
    do {                           // permute plane landing order, up to
8!
      double lo = 0, hi = 86400;              // min 0s, max 1 day =
86400s
      L = -1;                           // start with an infeasible
solution
      while (fabs(lo - hi) >= 1e-3) {        // binary search L, EPS =
1e-3
        L = (lo + hi) / 2.0;   // we want the answer rounded to nearest
int
        double retVal = greedyLanding();                // round down
first
        if (retVal <= 1e-2) lo = L;                      // must increase
L
        else                hi = L;          // infeasible, must decrease
L
```

```
      }
      maxL = max(maxL, L);              // get the max over all
permutations
    }
    while (next_permutation(order, order + n));     // try all
permutations

    // other way for rounding is to use printf format string:
%.0lf:%0.2lf
    maxL = (int)(maxL + 0.5);                    // round to nearest
second
    printf("Case %d: %d:%0.2d\n", caseNo++, (int)(maxL/60),
(int)maxL%60);
  }

  return 0;
}
```

```
#include <cmath>
#include <cstdio>
using namespace std;

#define MAX_N 3                              // adjust this value as
needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) {
  // input: N, Augmented Matrix Aug, output: Column vector X, the answer
  int i, j, k, l; double t;

  for (i = 0; i < N - 1; i++) {            // the forward elimination
phase
    l = i;
    for (j = i + 1; j < N; j++)       // which row has largest column
value
      if (fabs(Aug.mat[j][i]) > fabs(Aug.mat[l][i]))
        l = j;                                   // remember this row
l
    // swap this pivot row, reason: minimize floating point error
    for (k = i; k <= N; k++)            // t is a temporary double
variable
      t = Aug.mat[i][k], Aug.mat[i][k] = Aug.mat[l][k], Aug.mat[l][k] =
t;
    for (j = i + 1; j < N; j++)     // the actual forward elimination
phase
      for (k = N; k >= i; k--)
        Aug.mat[j][k] -= Aug.mat[i][k] * Aug.mat[j][i] / Aug.mat[i][i];
  }

  ColumnVector Ans;                          // the back substitution
phase
  for (j = N - 1; j >= 0; j--) {                          // start from
back
    for (t = 0.0, k = j + 1; k < N; k++) t += Aug.mat[j][k] * Ans.vec[k];
    Ans.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j]; // the answer is
here
  }
  return Ans;
}

int main() {
  AugmentedMatrix Aug;
  Aug.mat[0][0] = 1; Aug.mat[0][1] = 1; Aug.mat[0][2] = 2; Aug.mat[0][3]
= 9;
  Aug.mat[1][0] = 2; Aug.mat[1][1] = 4; Aug.mat[1][2] = -3; Aug.mat[1][3]
= 1;
  Aug.mat[2][0] = 3; Aug.mat[2][1] = 6; Aug.mat[2][2] = -5; Aug.mat[2][3]
= 0;

  ColumnVector X = GaussianElimination(3, Aug);
  printf("X = %.1lf, Y = %.1lf, Z = %.1lf\n", X.vec[0], X.vec[1],
X.vec[2]);

  return 0;
}
```

```
#include <cstdio>
#include <vector>
using namespace std;

#define MAX_N 1000

vector< vector<int> > children;

int L[2*MAX_N], E[2*MAX_N], H[MAX_N], idx;

void dfs(int cur, int depth) {
  H[cur] = idx;
  E[idx] = cur;
  L[idx++] = depth;
  for (int i = 0; i < children[cur].size(); i++) {
    dfs(children[cur][i], depth+1);
    E[idx] = cur;                              // backtrack to current
node
    L[idx++] = depth;
  }
}

void buildRMQ() {
  idx = 0;
  memset(H, -1, sizeof H);
  dfs(0, 0);                        // we assume that the root is at index
0
}

int main() {
  children.assign(10, vector<int>());
  children[0].push_back(1); children[0].push_back(7);
  children[1].push_back(2); children[1].push_back(3);
children[1].push_back(6);
  children[3].push_back(4); children[3].push_back(5);
  children[7].push_back(8); children[7].push_back(9);

  buildRMQ();
  for (int i = 0; i < 2*10-1; i++) printf("%d ", H[i]);
  printf("\n");
  for (int i = 0; i < 2*10-1; i++) printf("%d ", E[i]);
  printf("\n");
  for (int i = 0; i < 2*10-1; i++) printf("%d ", L[i]);
  printf("\n");

  return 0;
}
```

```
#include <cstdio>
using namespace std;

#define abs_val(a) (((a)>=0)?(a):-(a))
typedef long long ll;

ll mulmod(ll a, ll b, ll c) { // returns (a * b) % c, and minimize
overflow
  ll x = 0, y = a % c;
  while (b > 0) {
    if (b % 2 == 1) x = (x + y) % c;
    y = (y * 2) % c;
    b /= 2;
  }
  return x % c;
}

ll gcd(ll a, ll b) { return !b ? a : gcd(b, a % b); }       // standard
gcd

ll pollard_rho(ll n) {
  int i = 0, k = 2;
  ll x = 3, y = 3;                // random seed = 3, other values
possible
  while (1) {
    i++;
    x = (mulmod(x, x, n) + n - 1) % n;               // generating
function
    ll d = gcd(abs_val(y - x), n);                       // the key
insight
    if (d != 1 && d != n) return d;         // found one non-trivial
factor
    if (i == k) y = x, k *= 2;
} }

int main() {
  ll n = 2063512844981574047LL;    // we assume that n is not a large
prime
  ll ans = pollard_rho(n);          // break n into two non trivial
factors
  if (ans > n / ans) ans = n / ans;          // make ans the smaller
factor
  printf("%lld %lld\n", ans, n / ans);  // should be: 1112041493
1855607779
} // return 0;
```

```
#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

#define MAX_N 1000                            // adjust this value as
needed
#define LOG_TWO_N 10            // 2^10 > 1000, adjust this value as
needed

class RMQ {                                                // Range Minimum
Query
private:
  int _A[MAX_N], SpT[MAX_N][LOG_TWO_N];
public:
  RMQ(int n, int A[]) {    // constructor as well as pre-processing
routine
    for (int i = 0; i < n; i++) {
      _A[i] = A[i];
      SpT[i][0] = i; // RMQ of sub array starting at index i + length
2^0=1
    }
    // the two nested loops below have overall time complexity = O(n log
n)
    for (int j = 1; (1<<j) <= n; j++) // for each j s.t. 2^j <= n, O(log
n)
      for (int i = 0; i + (1<<j) - 1 < n; i++)    // for each valid i,
O(n)
        if (_A[SpT[i][j-1]] < _A[SpT[i+(1<<(j-1))][j-1]])            //
RMQ
          SpT[i][j] = SpT[i][j-1];    // start at index i of length 2^(j-
1)
        else                    // start at index i+2^(j-1) of length 2^(j-
1)
          SpT[i][j] = SpT[i+(1<<(j-1))][j-1];
  }

  int query(int i, int j) {
    int k = (int)floor(log((double)j-i+1) / log(2.0));    // 2^k <= (j-
i+1)
    if (_A[SpT[i][k]] <= _A[SpT[j-(1<<k)+1][k]]) return SpT[i][k];
    else                                          return SpT[j-
(1<<k)+1][k];
} };

int main() {
  // same example as in chapter 2: segment tree
  int n = 7, A[] = {18, 17, 13, 19, 15, 11, 20};
  RMQ rmq(n, A);
  for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
      printf("RMQ(%d, %d) = %d\n", i, j, rmq.query(i, j));

  return 0;
}
```

```
// Modular Fibonacci

#include <cmath>
#include <cstdio>
#include <cstring>
using namespace std;

typedef long long ll;
ll MOD;

#define MAX_N 2                                   // increase this if
needed
struct Matrix { ll mat[MAX_N][MAX_N]; };     // to let us return a 2D
array

Matrix matMul(Matrix a, Matrix b) {            // O(n^3), but O(1) as n =
2
  Matrix ans; int i, j, k;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      for (ans.mat[i][j] = k = 0; k < MAX_N; k++) {
        ans.mat[i][j] += (a.mat[i][k] % MOD) * (b.mat[k][j] % MOD);
        ans.mat[i][j] %= MOD;             // modulo arithmetic is used
here
      }
  return ans;
}

Matrix matPow(Matrix base, int p) {  // O(n^3 log p), but O(log p) as n =
2
  Matrix ans; int i, j;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      ans.mat[i][j] = (i == j);                   // prepare identity
matrix
  while (p) {        // iterative version of Divide & Conquer
exponentiation
    if (p & 1)                     // check if p is odd (the last bit is
on)
      ans = matMul(ans, base);                             // update
ans
    base = matMul(base, base);                           // square the
base
    p >>= 1;                                               // divide p by
2
  }
  return ans;
}

int main() {
  int i, n, m;

  while (scanf("%d %d", &n, &m) == 2) {
    Matrix ans;                             // special matrix for
Fibonaccci
    ans.mat[0][0] = 1;  ans.mat[0][1] = 1;
    ans.mat[1][0] = 1;  ans.mat[1][1] = 0;
    for (MOD = 1, i = 0; i < m; i++)                       // set MOD =
2^m
      MOD *= 2;
```

```
    ans = matPow(ans, n);                                      // O(log
n)
    printf("%lld\n", ans.mat[0][1]);                     // this if
fib(n)
  }

  return 0;
}
```

```
// Sending email
// standard SSSP problem
// demo using Dijkstra's and SPFA

#include <cstdio>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;

#define INF 2000000000

int i, j, t, n, m, S, T, a, b, w, caseNo = 1;
vector<vii> AdjList;

int main() {
#ifndef ONLINE_JUDGE
  freopen("in.txt", "r", stdin);
#endif

  scanf("%d", &t);
  while (t--) {
    scanf("%d %d %d %d", &n, &m, &S, &T);

    // build graph
    AdjList.assign(n, vii());
    while (m--) {
      scanf("%d %d %d", &a, &b, &w);
      AdjList[a].push_back(ii(b, w)); // bidirectional
      AdjList[b].push_back(ii(a, w));
    }

/*
    // Dijkstra from source S
    vi dist(n, INF); dist[S] = 0;
    priority_queue< ii, vii, greater<ii> > pq; pq.push(ii(0, S)); // sort
based on increasing distance

    while (!pq.empty()) { // main loop
      ii top = pq.top(); pq.pop(); // greedy: pick shortest unvisited
vertex
      int d = top.first, u = top.second;
      if (d != dist[u]) continue;
      for (j = 0; j < (int)AdjList[u].size(); j++) { // all outgoing
edges from u
        int v = AdjList[u][j].first, weight_u_v = AdjList[u][j].second;
        if (dist[u] + weight_u_v < dist[v]) { // if can relax
          dist[v] = dist[u] + weight_u_v; // relax
          pq.push(ii(dist[v], v)); // enqueue this neighbor
        }                          // regardless it is already in pq or
not
      }
    }
*/

    // SPFA from source S
```

```
    // initially, only S has dist = 0 and in the queue
    vi dist(n, INF); dist[S] = 0;
    queue<int> q; q.push(S);
    vi in_queue(n, 0); in_queue[S] = 1;

    while (!q.empty()) {
      int u = q.front(); q.pop(); in_queue[u] = 0;
      for (j = 0; j < (int)AdjList[u].size(); j++) { // all outgoing
edges from u
        int v = AdjList[u][j].first, weight_u_v = AdjList[u][j].second;
        if (dist[u] + weight_u_v < dist[v]) { // if can relax
          dist[v] = dist[u] + weight_u_v; // relax
          if (!in_queue[v]) { // add to the queue only if it's not in the
queue
            q.push(v);
            in_queue[v] = 1;
          }
        }
      }
    }

    printf("Case #%d: ", caseNo++);
    if (dist[T] != INF) printf("%d\n", dist[T]);
    else                printf("unreachable\n");
  }

  return 0;
}
```

```
// Roman Numerals

#include <cstdio>
#include <cstdlib>
#include <ctype.h>
#include <map>
#include <string>
using namespace std;

void AtoR(int A) {
  map<int, string> cvt;
  cvt[1000] = "M"; cvt[900] = "CM"; cvt[500] = "D"; cvt[400] = "CD";
  cvt[100]  = "C"; cvt[90]  = "XC"; cvt[50]  = "L"; cvt[40]  = "XL";
  cvt[10]   = "X"; cvt[9]   = "IX"; cvt[5]   = "V"; cvt[4]   = "IV";
  cvt[1]    = "I";
  // process from larger values to smaller values
  for (map<int, string>::reverse_iterator i = cvt.rbegin();
       i != cvt.rend(); i++)
    while (A >= i->first) {
      printf("%s", ((string)i->second).c_str());
      A -= i->first; }
  printf("\n");
}

void RtoA(char R[]) {
  map<char, int> RtoA;
  RtoA['I'] = 1;   RtoA['V'] = 5;   RtoA['X'] = 10;   RtoA['L'] = 50;
  RtoA['C'] = 100; RtoA['D'] = 500; RtoA['M'] = 1000;

  int value = 0;
  for (int i = 0; R[i]; i++)
    if (R[i+1] && RtoA[R[i]] < RtoA[R[i+1]]) {     // check next char
first
      value += RtoA[R[i+1]] - RtoA[R[i]];                // by definition
      i++; }                                              // skip this
char
    else value += RtoA[R[i]];
  printf("%d\n", value);
}

int main() {
#ifndef ONLINE_JUDGE
  freopen("in.txt", "r", stdin);
#endif

  char str[1000];

  while (gets(str) != NULL) {
    if (isdigit(str[0])) AtoR(atoi(str)); // Arabic to Roman Numerals
    else                 RtoA(str); // Roman to Arabic Numerals
  }

  return 0;
}
```

```
// Tunnelling the Earth
// Great Circle distance + Euclidean distance

#include <cstdio>
#include <cmath>
using namespace std;

#define PI acos(-1.0)
#define EARTH_RAD (6371009) // in meters

double gcDistance(double pLat, double pLong,
                  double qLat, double qLong, double radius) {
  pLat *= PI / 180; pLong *= PI / 180;
  qLat *= PI / 180; qLong *= PI / 180;
  return radius * acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong) +
                       cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
                       sin(pLat)*sin(qLat));
}

double EuclediانDistance(double pLat, double pLong, // 3D version
                         double qLat, double qLong, double radius) {
  double phi1 = (90 - pLat) * PI / 180;
  double theta1 = (360 - pLong) * PI / 180;
  double x1 = radius * sin(phi1) * cos(theta1);
  double y1 = radius * sin(phi1) * sin(theta1);
  double z1 = radius * cos(phi1);

  double phi2 = (90 - qLat) * PI / 180;
  double theta2 = (360 - qLong) * PI / 180;
  double x2 = radius * sin(phi2) * cos(theta2);
  double y2 = radius * sin(phi2) * sin(theta2);
  double z2 = radius * cos(phi2);

  double dx = x1 - x2, dy = y1 - y2, dz = z1 - z2;
  return sqrt(dx * dx + dy * dy + dz * dz);
}

int main() {
  int TC;
  double lat1, lon1, lat2, lon2;

  scanf("%d", &TC);
  while (TC--) {
    scanf("%lf %lf %lf %lf", &lat1, &lon1, &lat2, &lon2);
    printf("%.0lf\n", gcDistance(lat1, lon1, lat2, lon2, EARTH_RAD) -
                      EuclediانDistance(lat1, lon1, lat2, lon2,
EARTH_RAD));
  }

  return 0;
}
```

```
// Come and Go
// check if the graph is strongly connected, i.e. the SCC of the graph is
the graph itself (only 1 SCC)

#include <algorithm>
#include <cstdio>
#include <iostream>
#include <vector>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define DFS_WHITE -1

int i, j, N, M, V, W, P, dfsNumberCounter, numSCC;
vector<vii> AdjList, AdjListT;
vi dfs_num, dfs_low, S, S_copy, visited;                     // global
variables

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <=
dfs_num[u]
  S.push_back(u);           // stores u in a vector based on order of
visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      tarjanSCC(v.first);
    if (visited[v.first])                                // condition for
update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
  }

  if (dfs_low[u] == dfs_num[u]) {         // if this is a root (start) of
an SCC
    ++numSCC;
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      if (u == v) break;
    }
  }
}

void Kosaraju(int u, int pass) {      // pass = 1 (original), 2
(transpose)
  dfs_num[u] = 1;
  vii neighbor;
  if (pass == 1) neighbor = AdjList[u]; else neighbor = AdjListT[u];
  for (int j = 0; j < (int)neighbor.size(); j++) {
    ii v = neighbor[j];
    if (dfs_num[v.first] == DFS_WHITE)
      Kosaraju(v.first, pass);
  }
  S.push_back(u);       // as in finding topological order in Section
4.2.5
}

int main() {
```

```
#ifndef ONLINE_JUDGE
  freopen("in.txt", "r", stdin);
#endif

  while (scanf("%d %d", &N, &M), (N || M)) {
    AdjList.assign(N, vii());
    AdjListT.assign(N, vii()); // the transposed graph
    for (i = 0; i < M; i++) {
      scanf("%d %d %d", &V, &W, &P); V--; W--;
      AdjList[V].push_back(ii(W, 1)); // always
      AdjListT[W].push_back(ii(V, 1));
      if (P == 2) { // if this is two way, add the reverse direction
        AdjList[W].push_back(ii(V, 1));
        AdjListT[V].push_back(ii(W, 1));
      }
    }

    //// run Tarjan's SCC code here
    //dfs_num.assign(N, DFS_WHITE); dfs_low.assign(N, 0);
visited.assign(N, 0);
    //dfsNumberCounter = numSCC = 0;
    //for (i = 0; i < N; i++)
    //  if (dfs_num[i] == DFS_WHITE)
    //    tarjanSCC(i);

    // run Kosaraju's SCC code here
    S.clear();  // first pass is to record the `post-order' of original
graph
    dfs_num.assign(N, DFS_WHITE);
    for (i = 0; i < N; i++)
      if (dfs_num[i] == DFS_WHITE)
        Kosaraju(i, 1);

    numSCC = 0;   // second pass: explore the SCCs based on first pass
result
    dfs_num.assign(N, DFS_WHITE);
    for (i = N-1; i >= 0; i--)
      if (dfs_num[S[i]] == DFS_WHITE) {
        numSCC++;
        Kosaraju(S[i], 2);
      }

    // if SCC is only 1, print 1, otherwise, print 0
    printf("%d\n", numSCC == 1 ? 1 : 0);
  }

  return 0;
}
```