Catalogue

Common

```
const double eps = 1e-8;
const double pi = acos(-1.0);

int sgn(double d) {
   if (d > eps)
      return 1;
   if (d < -eps)
      return -1;
   return 0;
}

void to_normal(double& d, double l = 1.0) {
   if (d > l)
      d = l;
   if (d < -l)
      d = -l;
}
```

Dimension of Two

Point

```
struct point {
   double x, y;
   point(double _x = 0, double _y = 0): x(_x), y(_y) {
   }
   void input() {
      scanf("%lf%lf", &x, &y);
   }
   double len() const {
      return sqrt(x * x + y * y);
   }
   point trunc(double l) const {
      double r = l / len();
      return point(x * r, y * r);
   }
   point rotate_left() const {
        return point(-y, x);
    }
    point rotate_left(double ang) const {
        double c = cos(ang), s = sin(ang);
        return point(x * c - y * s, y * c + x * s);
    }
    point rotate_right() const {
        return point(y, -x);
    }
    point rotate_right(double ang) const {
        double c = cos(ang), s = sin(ang);
        return point(x * c + y * s, y * c - x * s);
    }
};

bool operator==(const point& p1, const point& p2) {
    return sgn(p1.x - p2.x) == 0 && sgn(p1.y - p2.y) == 0;
}

bool operator<(const point& p1, const point& p2) {
    return sgn(p1.x - p2.x) == 0 ? sgn(p1.y - p2.y) < 0 : p1.x < p2.x;
}

bool operator>(const point& p1, const point& p2) {
    return sgn(p1.x - p2.x) == 0 ? sgn(p1.y - p2.y) > 0 : p1.x > p2.x;
}

point operator+(const point& p1, const point& p2) {
    return point(p1.x + p2.x, p1.y + p2.y);
}

point operator-(const point& p1, const point& p2) {
    return point(p1.x - p2.x, p1.y - p2.y);
}

double operator^(const point& p1, const point& p2) {
    return p1.x * p2.x + p1.y * p2.y;
}

double operator*(const point& p1, const point& p2) {
    return p1.x * p2.y - p1.y * p2.x;
}

point operator*(const point& p, double r) {
    return point(p.x * r, p.y * r);
}

point operator/(const point& p, double r) {
    return point(p.x / r, p.y / r);
}
```

Relationship of Point and Line Segment

```
double get_distance(const point&p, const point& p1, const point& p2) {
    if (sgn((p2 - p1) ^ (p - p1)) <= 0)
        return (p - p1).len();
    if (sgn((p1 - p2) ^ (p - p2)) <= 0)
        return (p - p2).len();
    return abs((p1 - p) * (p2 - p) / (p1 - p2).len());
}
```

Relationship of Line Segments

```
bool get_intersection(const point& p1, const point& p2, const point& p3, const point& p4, point& c) {
    double d1 = (p2 - p1) * (p3 - p1), d2 = (p2 - p1) * (p4 - p1);
    double d3 = (p4 - p3) * (p1 - p3), d4 = (p4 - p3) * (p2 - p3);
    int s1 = sgn(d1), s2 = sgn(d2), s3 = sgn(d3), s4 = sgn(d4);
    if (s1 == 0 && s2 == 0 && s3 == 0 && s4 == 0)
        return false;
    c = point((p3.x * d2 - p4.x * d1) / (d2 - d1), (p3.y * d2 - p4.y * d1) / (d2 - d1));
    return s1 * s2 <= 0 && s3 * s4 <= 0;
}
```

Relationship of Point and Line

```
double get_distance(const point&p, const point& p1, const point& p2) {
    return abs((p1 - p) * (p2 - p) / (p1 - p2).len());
}
```

```
point get_perpendicular(const point& p, const point& p1, const point& p2) {
   double d = (p1 - p) * (p2 - p) / (p1 - p2).len();
   return p - (p2 - p1).rotate_left().trunc(d);
}

point get_reflection(const point& p, const point& p1, const point& p2) {
   double d = (p1 - p) * (p2 - p) / (p1 - p2).len();
   return p - (p2 - p1).rotate_left().trunc(d * 2.0);
}
```

Relationship of Point and Polygon

```
int get_position(const point& p, const point* pol, int n) {
   double ang = 0;
   for (int i = 0; i < n; ++i) {
      point p1 = pol[i] - p, p2 = pol[(i + 1) % n] - p;
      double c = (p1 ^ p2) / (p1.len() * p2.len());
      to_normal(c);
      ang += sgn(p1 * p2) * acos(c);
   }
   ang = abs(ang);
   return ang < 0.5 * pi ? -1 : (ang < 1.5 * pi ? 0 : 1);
}
```

Relationship of Point and Convex Polygon

```
int get_position(const point& p, const point* pol, int n) {
   point c((pol[n - 1].x + pol[0].x) / 2.0, (pol[n - 1].y + pol[0].y) / 2.0);
   int s = sgn((pol[0] - pol[n - 1]) * (p - pol[n - 1]));
   if (s < 0)
      return -1;
   if (s == 0)
      return sgn((p - pol[n - 1]) ^ (p - pol[0])) <= 0 ? 0 : -1;
   int lb = 0, ub = n - 1;
   while (lb != ub) {
      int mid = (lb + ub + 1) / 2;
      if (sgn((pol[mid] - c) * (p - c)) >= 0)
         lb = mid;
      else
         ub = mid - 1;
   }
   return sgn((pol[lb + 1] - pol[lb]) * (p - pol[lb]));
}
```

Relationship of Line and Convex Polygon

```
struct edge {
   int id;
   point v;
   double ang;
   edge() {
   }
   edge(int _id, const point& _v): id(_id), v(_v) {
      ang = atan2(v.y, v.x);
      if (sgn(ang - pi) == 0)
         ang = -pi;
   }
};

bool operator<(const edge& e1, const edge& e2) {
   return sgn(e1.ang - e2.ang) < 0;
}

edge e[max_n];
point l1, l2;

void pre_compute(point* pol, int n) {
   for (int i = 0; i < n; ++i) {
      pol[n + i] = pol[i];
      e[i] = edge(i, pol[i + 1] - pol[i]);
   }
   sort(e, e + n);
}

bool is_less(const point& p1, const point& p2) {
   return sgn((l1 - p1) * (l2 - p1) - (l1 - p2) * (l2 - p2)) < 0;
}

bool get_intersection(const point* pol, int n, const point& p1, const point& p2,
point& c1, point& c2) {
   int p_l = e[(lower_bound(e, e + n, edge(-1, p1 - p2)) - e) % n].id;
   int p_r = e[(lower_bound(e, e + n, edge(-1, p2 - p1)) - e) % n].id;
   if (sgn((p2 - p1) * (pol[p_l] - p1)) * sgn((p2 - p1) * (pol[p_r] - p1)) >= 0)
      return false;
   l1 = p2, l2 = p1;
   int k1 = (lower_bound(pol + p_l, pol + (p_r < p_l ? p_r + n : p_r) + 1, p1,
is_less) - pol) % n;
   l1 = p1, l2 = p2;
   int k2 = (lower_bound(pol + p_r, pol + (p_l < p_r ? p_l + n : p_l) + 1, p2,
is_less) - pol) % n;
   c1 = get_intersection(p1, p2, pol[k1], pol[(k1 + n - 1) % n]);
   c2 = get_intersection(p1, p2, pol[k2], pol[(k2 + n - 1) % n]);
   return true;
}
```

Circle

```
struct circle {
   point c;
   double r;
   circle() {
   }
   circle(const point& _c, double _r): c(_c), r(_r) {
   }
   void input() {
      c.input();
      scanf("%lf", &r);
   }
   double area() const {
      return pi * r * r;
   }
   int get_intersection(const point& p1, const point& p2, point& c1, point& c2)
const {
      double d = (p1 - c) * (p2 - c) / (p1 - p2).len();
      if (sgn(abs(d) - r) >= 0)
         return 0;
      point pp = c - (p2 - p1).rotate_left().trunc(d);
      double l = sqrt(r * r - d * d);
      c1 = pp - (p2 - p1).trunc(l);
      c2 = pp + (p2 - p1).trunc(l);
      int res = 0;
      res |= (sgn((p1 - c1) ^ (p2 - c1)) <= 0 ? 1 : 0) << 0;
      res |= (sgn((p1 - c2) ^ (p2 - c2)) <= 0 ? 1 : 0) << 1;
      return res;
   }
   bool get_intersection(const circle& cir, point& c1, point& c2) const {
      double d = (c - cir.c).len();
      if (sgn(d - (r + cir.r)) >= 0 || sgn(d - abs(r - cir.r)) <= 0)
         return false;
      double p = (d + r + cir.r) / 2.0;
      double h = sqrt(abs(p * (p - d) * (p - r) * (p - cir.r))) * 2.0 / d;
      point pp = c + (cir.c - c).trunc((r * r + d * d - cir.r * cir.r) / (2.0 *
d));
      c1 = pp - (cir.c - c).rotate_left().trunc(h);
      c2 = pp + (cir.c - c).rotate_left().trunc(h);
      return true;
   }
   bool get_tangency_points(const point& p, point& t1, point& t2) const {
      double d = (p - c).len();
      if (sgn(d - r) <= 0)
         return false;
      point pp = c + (p - c).trunc(r * r / d);
      double h = sqrt(abs(r * r - (r * r * r * r) / (d * d)));
      t1 = pp - (p - c).rotate_left().trunc(h);
      t2 = pp + (p - c).rotate_left().trunc(h);
      return true;
   }
   vector<pair<point, point> > get_tangency_points(const circle& cir) const {
      vector<pair<point, point> > t;
      double d = (c - cir.c).len();
      if (sgn(d - abs(cir.r - r)) <= 0)
         return t;
      double l = sqrt(abs(d * d - (cir.r - r) * (cir.r - r)));
      double h1 = r * l / d, h2 = cir.r * l / d;
      point p = (r > cir.r ? cir.c - c : c - cir.c);
      point pp1 = c + p.trunc(sqrt(abs(r * r - h1 * h1))), pp2 = cir.c +
p.trunc(sqrt(abs(cir.r * cir.r - h2 * h2)));
      t.push_back(make_pair(pp1 + p.rotate_left().trunc(h1), pp2 +
p.rotate_left().trunc(h2)));
      t.push_back(make_pair(pp1 - p.rotate_left().trunc(h1), pp2 -
p.rotate_left().trunc(h2)));
      if (sgn(d - (r + cir.r)) <= 0)
         return t;
      double d1 = d * r / (r + cir.r), d2 = d * cir.r / (r + cir.r);
      point pp3 = c + (cir.c - c).trunc(r * r / d1), pp4 = cir.c + (c - cir.c).trunc(cir.r * cir.r / d2);
      double h3 = sqrt(abs(r * r - (r * r * r * r) / (d1 * d1))), h4 = sqrt(abs(cir.r * cir.r - (cir.r * cir.r * cir.r * cir.r) / (d2 * d2)));
      t.push_back(make_pair(pp3 + (cir.c - c).rotate_left().trunc(h3), pp4 + (c - cir.c).rotate_left().trunc(h4)));
      t.push_back(make_pair(pp3 - (cir.c - c).rotate_left().trunc(h3), pp4 - (c - cir.c).rotate_left().trunc(h4)));
      return t;
   }
   double get_intersection_area(const point& p1, const point& p2) const {
      point v1 = (p1 - c), v2 = (p2 - c);
      double d1 = v1.len(), d2 = v2.len();
      point c1, c2;
      int s = get_intersection(p1, p2, c1, c2);
      if (s == 0) {
         if (sgn(d1 - r) > 0 && sgn(d2 - r) > 0) {
            double t = (v1 ^ v2) / (d1 * d2);
            to_normal(t);
            return r * r * acos(t) / 2.0;
         }
         return abs(v1 * v2 / 2.0);
      }
      if (s == 1) {
         point k = c1 - c;
         double t = (v1 ^ k) / (d1 * k.len());
         to_normal(t);
         return abs(v2 * k / 2.0) + r * r * acos(t) / 2.0;
      }
      if (s == 2) {
         point k = c2 - c;
         double t = (v2 ^ k) / (d2 * k.len());
         to_normal(t);
         return abs(v1 * k / 2.0) + r * r * acos(t) / 2.0;
      }
      point k1 = c1 - c, k2 = c2 - c;
      double t1 = (v1 ^ k1) / (d1 * k1.len());
      to_normal(t1);
      double t2 = (v2 ^ k2) / (d2 * k2.len());
      to_normal(t2);
      return abs(k1 * k2 / 2.0) + r * r * (acos(t1) + acos(t2)) / 2.0;
   }
   double get_intersection_area(const circle& cir) const {
      double d = (c - cir.c).len();
      if (sgn(d - (r + cir.r)) >= 0)
         return 0;
      if (sgn(d - abs(r - cir.r)) <= 0)
         return min(area(), cir.area());
      double c1 = (r * r + d * d - cir.r * cir.r) / (2.0 * r * d);
      double c2 = (cir.r * cir.r + d * d - r * r) / (2.0 * cir.r * d);
      to_normal(c1);
      to_normal(c2);
      double p = (r + cir.r + d) / 2.0;
      double s = sqrt(p * (p - r) * (p - cir.r) * (p - d));
      return acos(c1) * r * r + acos(c2) * cir.r * cir.r - s * 2.0;
   }
};
```

Convex Hull

```
int dn, hd[max_n], un, hu[max_n];

void get_convex_hull(point* p, int n, point* pol, int& m) {
   sort(p, p + n);
   dn = un = 2;
   hd[0] = hu[0] = 0;
   hd[1] = hu[1] = 1;
   for (int i = 2; i < n; ++i) {
      for (; dn > 1 && sgn((p[hd[dn - 1]] - p[hd[dn - 2]]) * (p[i] - p[hd[dn - 1]])) <= 0; --dn);
      for (; un > 1 && sgn((p[hu[un - 1]] - p[hu[un - 2]]) * (p[i] - p[hu[un - 1]])) >= 0; --un);
      hd[dn++] = hu[un++] = i;
   }
   m = 0;
   for (int i = 0; i < dn - 1; ++i)
      pol[m++] = p[hd[i]];
   for (int i = un - 1; i > 0; --i)
      pol[m++] = p[hu[i]];
}
```

Half-plane Intersection (O(n^2))

```
void get_intersection(point* pol1, int n1, const point& p1, const point& p2, point*
pol2, int& n2) {
   n2 = 0;
   if (n1 == 0)
      return;
   point v = p2 - p1;
   int last_s = sgn(v * (pol1[n1 - 1] - p1));
   for (int i = 0; i < n1; ++i) {
      int s = sgn(v * (pol1[i] - p1));
      if (s == 0) {
         pol2[n2++] = pol1[i];
      } else if (s < 0) {
         if (last_s > 0)
            pol2[n2++] = get_intersection(p1, p2, i == 0 ? pol1[n1 - 1] : pol1[i
- 1], pol1[i]);
      } else if (s > 0) {
         if (last_s < 0)
            pol2[n2++] = get_intersection(p1, p2, i == 0 ? pol1[n1 - 1] : pol1[i
- 1], pol1[i]);
         pol2[n2++] = pol1[i];
      }
      last_s = s;
   }
}
```

Half-plane Intersection (O(n * lg(n)))

```
struct half_plane {
   point p1, p2;
   double ang;
   half_plane() {
   }
   half_plane(const point& _p1, const point& _p2): p1(_p1), p2(_p2) {
      ang = atan2(p2.y - p1.y, p2.x - p1.x);
      if (sgn(ang - pi) == 0)
         ang = -pi;
   }
   int get_position(const point& p) const {
      return sgn((p2 - p1) * (p - p1));
   }
};

bool operator<(const half_plane& pl1, const half_plane& pl2) {
   return sgn(pl1.ang - pl2.ang) == 0 ? pl1.get_position(pl2.p1) < 0 : pl1.ang
< pl2.ang;
}

double operator^(const half_plane& pl1, const half_plane& pl2) {
   return (pl1.p2 - pl1.p1) ^ (pl2.p2 - pl2.p1);
}

double operator*(const half_plane& pl1, const half_plane& pl2) {
   return (pl1.p2 - pl1.p1) * (pl2.p2 - pl2.p1);
}

point get_intersection(const half_plane& pl1, const half_plane& pl2) {
   double d1 = (pl1.p2 - pl1.p1) * (pl2.p1 - pl1.p1), d2 = (pl1.p2 - pl1.p1) *
(pl2.p2 - pl1.p1);
   return point((pl2.p1.x * d2 - pl2.p2.x * d1) / (d2 - d1), (pl2.p1.y * d2 - pl2.p2.y
* d1) / (d2 - d1));
}

void get_intersection(const half_plane* pl, int n, point* pol, int& m) {
   m = 0;
   deque<int> deq1;
   deque<point> deq2;
   deq1.push_back(0);
   deq1.push_back(1);
   deq2.push_back(get_intersection(pl[0], pl[1]));
   for (int i = 2; i < n; ++i) {
      while (!deq2.empty() && pl[i].get_position(deq2.back()) <= 0) {
         if (sgn(pl[deq1[deq1.size() - 2]] * pl[i]) <= 0 && sgn(pl[deq1.back()]
* pl[i]) >= 0)
            return;
         deq1.pop_back();
         deq2.pop_back();
      }
      while (!deq2.empty() && pl[i].get_position(deq2.front()) <= 0) {
         deq1.pop_front();
         deq2.pop_front();
```

```
        }
        deq2.push_back(get_intersection(pl[deq1.back()], pl[i]));
        deq1.push_back(i);
        while (deq2.size() > 1 && pl[deq1.front()].get_position(deq2.back()) <= 0)
{
            deq1.pop_back();
            deq2.pop_back();
        }
        while (deq2.size() > 1 && pl[deq1.back()].get_position(deq2.front()) <= 0)
{
            deq1.pop_front();
            deq2.pop_front();
        }
    }
    m = deq2.size();
    copy(deq2.begin(), deq2.end(), pol);
    pol[m++] = get_intersection(pl[deq1.front()], pl[deq1.back()]);
}
```

Diameter of a Set of Points

```
double get_max_distance(point* p, int n, point* pol, int& m) {
    get_convex_hull(p, n, pol, m);
    double dis = 0;
    for (int i = 0, j = dn - 1; i < m; ++i) {
        dis = max(dis, (pol[j] - pol[i]).len());
        while (sgn((pol[(i + 1) % m] - pol[i]) * (pol[(j + 1) % m] - pol[j])) > 0)
{
            j = (j + 1) % m;
            dis = max(dis, (pol[j] - pol[i]).len());
        }
    }
    return dis;
}
```

Dimension of Three

Point

```
struct point {
    double x, y, z;
    point(double _x = 0, double _y = 0, double _z = 0): x(_x), y(_y), z(_z) {
    }
    void input() {
        scanf("%lf%lf%lf", &x, &y, &z);
    }
    double len() const {
        return sqrt(x * x + y * y + z * z);
    }
    point trunc(double l) const {
        double r = l / len();
        return point(x * r, y * r, z * r);
    }
};

bool operator==(const point& p1, const point& p2) {
    return sgn(p1.x - p2.x) == 0 && sgn(p1.y - p2.y) == 0 && sgn(p1.z - p2.z) ==
0;
}

bool operator<(const point& p1, const point& p2) {
    return sgn(p1.x - p2.x) == 0 ? (sgn(p1.y - p2.y) == 0 ? sgn(p1.z - p2.z) < 0 :
p1.y < p2.y) : p1.x < p2.x;
}

bool operator>(const point& p1, const point& p2) {
    return sgn(p1.x - p2.x) == 0 ? (sgn(p1.y - p2.y) == 0 ? sgn(p1.z - p2.z) > 0 :
p1.y > p2.y) : p1.x > p2.x;
}

point operator+(const point& p1, const point& p2) {
    return point(p1.x + p2.x, p1.y + p2.y, p1.z + p2.z);
}

point operator-(const point& p1, const point& p2) {
    return point(p1.x - p2.x, p1.y - p2.y, p1.z - p2.z);
}

double operator^(const point& p1, const point& p2) {
    return p1.x * p2.x + p1.y * p2.y + p1.z * p2.z;
}
```

```
point operator*(const point& p1, const point& p2) {
    return point(p1.y * p2.z - p1.z * p2.y, p1.z * p2.x - p1.x * p2.z, p1.x * p2.y - p1.y * p2.x);
}

point operator*(const point& p, double r) {
    return point(p.x * r, p.y * r, p.z * r);
}

point operator/(const point& p, double r) {
    return point(p.x / r, p.y / r, p.z / r);
}
```

Relationship of Point and Line Segment

```
double get_distance(const point& p, const point& p1, const point& p2) {
    if (sgn((p2 - p1) ^ (p - p1)) <= 0)
        return (p - p1).len();
    if (sgn((p1 - p2) ^ (p - p2)) <= 0)
        return (p - p2).len();
    return abs(((p1 - p) * (p2 - p)).len() / (p1 - p2).len());
}
```

Relationship of Point and Line

```
double get_distance(const point& p, const point& p1, const point& p2) {
    return abs(((p1 - p) * (p2 - p)).len() / (p1 - p2).len());
}

point get_perpendicular(const point& p, const point& p1, const point& p2) {
    point v = (p1 - p) * (p2 - p);
    double d = v.len() / (p1 - p2).len();
    return p - (v * (p2 - p1)).trunc(d);
}

point get_reflection(const point& p, const point& p1, const point& p2) {
    point v = (p1 - p) * (p2 - p);
    double d = v.len() / (p1 - p2).len();
    return p - (v * (p2 - p1)).trunc(d * 2.0);
}
```

Relationship of Point and Plane

```
double get_distance(const point& p, const point& p1, const point& p2, const point& p3) {
    point v = (p2 - p1) * (p3 - p1);
    return abs((v ^ (p - p1)) / v.len());
}

point get_perpendicular(const point& p, const point& p1, const point& p2, const point& p3) {
    point v = (p2 - p1) * (p3 - p1);
    double d = (v ^ (p - p1)) / v.len();
    return p - v.trunc(d);
}

point get_reflection(const point& p, const point& p1, const point& p2, const point& p3) {
    point v = (p2 - p1) * (p3 - p1);
    double d = (v ^ (p - p1)) / v.len();
    return p - v.trunc(d * 2.0);
}
```

Relationship of Line Segment and Plane

```
bool get_intersection(const point& p1, const point& p2, const point& pl1, const point& pl2, const point& pl3, point& c) {
    point v = (pl2 - pl1) * (pl3 - pl1);
    double d1 = v ^ (p1 - pl1), d2 = v ^ (p2 - pl1);
    int s1 = sgn(d1), s2 = sgn(d2);
    if (s1 == 0 && s2 == 0)
        return false;
    c = point((p1.x * d2 - p2.x * d1) / (d2 - d1), (p1.y * d2 - p2.y * d1) / (d2 - d1), (p1.z * d2 - p2.z * d1) / (d2 - d1));
    return s1 * s2 <= 0;
}
```

Convex Hull

```
struct face {
   int a, b, c;
   face(int _a = 0, int _b = 0, int _c = 0): a(_a), b(_b), c(_c) {
   }
};

const int max_n = 300 + 10, max_f = max_n * 2;

int n1, n2, pos[max_n][max_n];
face buf1[max_f], buf2[max_f], *p1, *p2;

int get_position(const point& p, const point& p1, const point& p2, const point&
p3) {
   return sgn((p2 - p1) * (p3 - p1) ^ (p - p1));
}

void check(int k, int a, int b, int s) {
   if (pos[b][a] == 0) {
      pos[a][b] = s;
      return;
   }
   if (pos[b][a] != s)
      p2[n2++] = (s < 0 ? face(k, b, a) : face(k, a, b));
   pos[b][a] = 0;
}

void get_convex_hull(point* p, int n, face* pol, int& m) {
   for (int i = 1; i < n; ++i) {
      if (p[i] != p[0]) {
         swap(p[i], p[1]);
         break;
      }
   }
   for (int i = 2; i < n; ++i) {
      if (sgn(((p[0] - p[i]) * (p[1] - p[i])).len()) != 0) {
         swap(p[i], p[2]);
         break;
      }
   }
   for (int i = 3; i < n; ++i) {
      if (get_position(p[i], p[0], p[1], p[2]) != 0) {
         swap(p[i], p[3]);
         break;
      }
   }
   p1 = buf1;
   p2 = buf2;
   n1 = n2 = 0;
   for (int i = 0; i < n; ++i)
      fill(pos[i], pos[i] + n, 0);
   p1[n1++] = face(0, 1, 2);
   p1[n1++] = face(2, 1, 0);
   for (int i = 3; i < n; ++i) {
      n2 = 0;
      for (int j = 0; j < n1; ++j) {
         int s = get_position(p[i], p[p1[j].a], p[p1[j].b], p[p1[j].c]);
         if (s == 0)
            s = -1;
         if (s <= 0)
            p2[n2++] = p1[j];
         check(i, p1[j].a, p1[j].b, s);
         check(i, p1[j].b, p1[j].c, s);
         check(i, p1[j].c, p1[j].a, s);
      }
      swap(p1, p2);
      swap(n1, n2);
   }
   m = n1;
   copy(p1, p1 + n1, pol);
}
```