目录

1 FFT+NTT

1.1 FFT

```
const int MAXN = 200020;
namespace FFT {
    const int mod = 998244353;
    struct cp {
        double x, y;
        cp(double _r = 0, double _i = 0) : x(_r), y(_i) {}
        cp operator+(const cp &tb) { return cp(x + tb.x, y + tb.y); }
        cp operator-(const cp &tb) { return cp(x - tb.x, y - tb.y); }
        cp operator*(const cp &tb) { return cp(x * tb.x - y * tb.y, y * tb.x + x * tb.y); }
    } A[MAXN], B[MAXN]; // 三倍空间
    const double pi = acos(-1.0);
    int L, R[MAXN]; int limit = 1;
    void FFT(cp a[], int type) {
        for (int i = 0; i < limit; i++)
            if (i < R[i]) swap(a[i], a[R[i]]);
        for (int j = 1; j < limit; j <<= 1) {
            cp Wn(cos(pi / j), type * sin(pi / j));
            for (int k = 0; k < limit; k += (j << 1)) {
                cp w(1, 0);
                for (int i = 0; i < j; i++, w = w * Wn) {
                    cp x = a[i + k], y = w * a[j + k + i];
                    a[i + k] = x + y; a[j + k + i] = x - y;
                }
            }
        }
    }
};
using FFT::L; using FFT::R; using FFT::limit;
char A[MAXN], B[MAXN]; int sum[MAXN];
int main() { // A * B
    while (~scanf("%s%s", A + 1, B + 1)) {
        int n = strlen(A + 1), m = strlen(B + 1);
        n--, m--;
        for (int i = 0; i <= n; i++) FFT::A[i].x = A[n - i + 1] - '0', FFT::A[i].y = 0;
        for (int i = 0; i <= m; i++) FFT::B[i].x = B[m - i + 1] - '0', FFT::B[i].y = 0;
        L = 0, limit = 1;
        while (limit <= n + m) limit <<= 1, L++;
        for (int i = n+1; i < limit; i++) FFT::A[i].x = 0, FFT::A[i].y = 0;
        for (int i = m+1; i < limit; i++) FFT::B[i].x = 0, FFT::B[i].y = 0;
        for (int i = 0; i < limit; i++)
            R[i] = (R[i >> 1] >> 1) | ((i & 1) << (L - 1));
        FFT::FFT(FFT::A, 1);
        FFT::FFT(FFT::B, 1);
        for (int i = 0; i < limit; i++) FFT::A[i] = FFT::A[i] * FFT::B[i];
        FFT::FFT(FFT::A, -1);
        for (int i = 0; i <= n + m; i++) {
            sum[i] = (int) (FFT::A[i].x / limit + 0.5);
        }
        sum[n+m+1] = 0;
        for (int i = 0; i <= n + m; i++) {
            sum[i + 1] += sum[i] / 10;
            sum[i] %= 10;
        }
        int len = n + m + 1;
        while (sum[len] == 0 && len > 0) len--;
        for (int i = len; i >= 0; i--)printf("%d", sum[i]);
        printf("\n");
    }
}
```

1.2 NTT

2 计算几何

2.1 长方体在三维空间中运动离目标点最近距离（2017ICPC 青岛 H）

```
struct Point3 be, en; // kuangbin
double ang(Point3 v1, Point3 v2) {
    return acos((v1 * v2) / (v1.len() * v2.len()));
}
struct Line3; // kuangbin

//点p绕向量ov旋转ang角度，旋转方向是向量ov叉乘向量op
Point3 rotate3(Point3 p, Point3 v, double angle) {
    double ret[3][3], a[3];
    v = v / v.len();
    ret[0][0] = (1.0 - cos(angle)) * v.x * v.x + cos(angle);
    ret[0][1] = (1.0 - cos(angle)) * v.x * v.y - sin(angle) * v.z;
    ret[0][2] = (1.0 - cos(angle)) * v.x * v.z + sin(angle) * v.y;
    ret[1][0] = (1.0 - cos(angle)) * v.y * v.x + sin(angle) * v.z;
    ret[1][1] = (1.0 - cos(angle)) * v.y * v.y + cos(angle);
    ret[1][2] = (1.0 - cos(angle)) * v.y * v.z - sin(angle) * v.x;
    ret[2][0] = (1.0 - cos(angle)) * v.z * v.x - sin(angle) * v.y;
    ret[2][1] = (1.0 - cos(angle)) * v.z * v.y + sin(angle) * v.x;
    ret[2][2] = (1.0 - cos(angle)) * v.z * v.z + cos(angle);
    for (int i = 0; i < 3; i++) a[i] = ret[i][0] * p.x + ret[i][1] * p.y + ret[i][2] * p.z;
    return Point3(a[0], a[1], a[2]);
}

int main() {
    int T; scanf("%d", &T);
    while (T--) {
        scanf("%lf%lf%lf", &be.x, &be.y, &be.z);    // 起始点
        scanf("%lf%lf%lf", &en.x, &en.y, &en.z);    // 目标点
        Point3 face = Point3(1, 0, 0), head = Point3(0, 0, 1);
        int m; char opt[3]; scanf("%d", &m);
        double res = 1.0 * inf;
        while (m--) {
            double d, t; scanf("%lf%s%lf", &d, opt, &t);
            double dx = d * cos(ang(Point3(1, 0, 0), face)), dy = d * cos(ang(Point3(0, 1, 0),
                face)), dz =
                    d * cos(ang(Point3(0, 0, 1), face));
            Point3 nbe = be + Point3(dx, dy, dz);
            Line3 lane = Line3(be, nbe);

            res = min(res, lane.dispointtoseg(en));
            if (opt[0] == 'U') { // 抬头
                Point3 v = face ^head;
                face = rotate3(face, v, t);
                head = rotate3(head, v, t);
            } else if (opt[0] == 'D') { // 低头
                Point3 v = head ^face;
                face = rotate3(face, v, t);
                head = rotate3(head, v, t);
            } else if (opt[0] == 'L') { // 左转
                face = rotate3(face, head, t);
            } else if (opt[0] == 'R') { // 后转
                Point3 v = head * (-1);
                face = rotate3(face, v, t);
            }
            be = nbe;
        }
        printf("%.2f\n", res);
    }
}
```

2.2 最小球覆盖（模拟退火）

```
const double eps = 1e-8;
const double start_T = 10000; // 初始温度记得设足够高
struct point3d {
    double x, y, z;
} data[150];
int n;
double dis(point3d a, point3d b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y) + (a.z - b.z) * (a.z - b.z));
}
double solve() {
    double step = start_T, ans = 1e30, mt;
    point3d z;
    z.x = z.y = z.z = 0;
    int s = 0;
    while (step > eps) {
        for (int i = 0; i < n; i++)
            if (dis(z, data[s]) < dis(z, data[i])) s = i;
        mt = dis(z, data[s]);
        ans = min(ans, mt);
        z.x += (data[s].x - z.x) / mt * step;
        z.y += (data[s].y - z.y) / mt * step;
        z.z += (data[s].z - z.z) / mt * step;
        step *= 0.98;
    }
    return ans;
}
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%lf%lf%lf", &data[i].x, &data[i].y, &data[i].z);
    }
    double ans = solve();
    printf("%.15f\n", ans);
}
```

2.3 求 n 个点的带权类费马点（模拟退火）

```
const int MAXN = 10005;
const double eps = 1e-8;
struct node {
    double x, y, weight;
} nd[MAXN];
int n;
node solve() {
    double step = 1000;
    node ans;
    while (step > eps) {
        double x = 0, y = 0;
        for (int i = 1; i <= n; i++) {
            double tmp = sqrt((ans.x - nd[i].x) * (ans.x - nd[i].x) + (ans.y - nd[i].y) *
                (ans.y - nd[i].y));
            if (fabs(tmp) < eps) continue;
            x += nd[i].weight / tmp * (nd[i].x - ans.x);
            y += nd[i].weight / tmp * (nd[i].y - ans.y);
        }
        double tmp = sqrt(x * x + y * y);
        if (fabs(tmp) >= eps) {
            ans.x += step / tmp * x;
            ans.y += step / tmp * y;
        }
        step *= 0.98;
    }
    return ans;
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%lf%lf%lf", &nd[i].x, &nd[i].y, &nd[i].weight);
    }
    node res = solve();
    printf("%.3lf %.3lf", res.x, res.y);

}
```

3 他人计算几何

3.1 st1vdy

```
namespace geometry {
#define db long double
```

```
#define pi acos(-1.0)
    constexpr db eps = 1e-7;
    int sign(db k) {
        if (k > eps) return 1;
        else if (k < -eps) return -1;
        return 0;
    }
    int cmp(db k1, db k2) { // k1 < k2 : -1, k1 == k2 : 0, k1 > k2 : 1
        return sign(k1 - k2);
    }
    int inmid(db k1, db k2, db k3) { // k3 在 [k1, k2] 内
        return sign(k1 - k3) * sign(k2 - k3) <= 0;
    }

    struct point { // 点类
        db x, y;
        point() {}
        point(db x_, db y_) :x(x_), y(y_) {}
        point operator + (const point& k) const { return point(k.x + x, k.y + y); }
        point operator - (const point& k) const { return point(x - k.x, y - k.y); }
        point operator * (db k) const { return point(x * k, y * k); }
        point operator / (db k1) const { return point(x / k1, y / k1); }
        point turn(db k1) { return point(x * cos(k1) - y * sin(k1), x * sin(k1) + y * cos(k1)); } // 逆时针旋转
        point turn90() { return point(-y, x); } // 逆时针方向旋转 90 度
        db len() { return sqrt(x * x + y * y); } // 向量长度
        db len2() { return x * x + y * y; } // 向量长度的平方
        db getPolarAngle() { return atan2(y, x); } // 向量极角
        db dis(point k) { return ((*this) - k).len(); } // 到点k的距离
        point unit() { db d = len(); return point(x / d, y / d); } // 单位向量
        point getdel() { // 将向量的方向调整为指向第一/四象限 包括y轴正方向
            if (sign(x) == -1 || (sign(x) == 0 && sign(y) == -1))
                return (*this) * (-1);
            else return (*this);
        }
        bool operator < (const point& k) const { // 水平序排序 x坐标为第一关键字,y坐标第二关键字
            return x == k.x ? y < k.y : x < k.x;
        }
        bool getP() const { // 判断点是否在上半平面 含x负半轴 不含x正半轴及零点
            return sign(y) == 1 || (sign(y) == 0 && sign(x) == -1);
        }
    };
    db cross(point k1, point k2) { return k1.x * k2.y - k1.y * k2.x; } // 向量 k1,k2 的叉积
    db dot(point k1, point k2) { return k1.x * k2.x + k1.y * k2.y; }   // 向量 k1,k2 的点积
    db rad(point k1, point k2) { // 向量 k1,k2 之间的有向夹角
        return atan2(cross(k1, k2), dot(k1, k2));
    }
    int inmid(point k1, point k2, point k3) { // k1 k2 k3共线时 判断点 k3 是否在线段 k1k2 上
        return inmid(k1.x, k2.x, k3.x) && inmid(k1.y, k2.y, k3.y);
    }
    int compareAngle(point k1, point k2) { // 比较向量 k1,k2 的角度大小 角度按照atan2()函数定义
        // k1 < k2 返回 1, k1 >= k2 返回 0
        return k1.getP() < k2.getP() || (k1.getP() == k2.getP() && sign(cross(k1, k2)) > 0);
    }
    point proj(point k1, point k2, point q) { // q 到直线 k1,k2 的投影
```

```
        point k = k2 - k1; return k1 + k * (dot(q - k1, k) / k.len2());
    }
    point reflect(point k1, point k2, point q) { return proj(k1, k2, q) * 2 - q; } // q 关于直线 k1,k2 的对称点
    int counterclockwise(point k1, point k2, point k3) { // k1 k2 k3 逆时针1 顺时针-1 否则0
        return sign(cross(k2 - k1, k3 - k1));
    }
    int checkLL(point k1, point k2, point k3, point k4) { // 判断直线 k1k2 和直线k3k4 是否相交
        // 即判断直线 k1k2 和 k3k4 是否平行 平行返回0 不平行返回1
        return sign(cross(k2 - k1, k4 - k3)) != 0;
    }
    point getLL(point k1, point k2, point k3, point k4) { // 求 k1k2 k3k4 两直线交点
        db w1 = cross(k1 - k3, k4 - k3), w2 = cross(k4 - k3, k2 - k3);
        return (k1 * w2 + k2 * w1) / (w1 + w2);
    }
    int intersect(db l1, db r1, db l2, db r2) { // 判断 [l1,r1] 和 [l2, r2] 是否相交
        if (l1 > r1) swap(l1, r1);
        if (l2 > r2) swap(l2, r2);
        return cmp(r1, l2) != -1 && cmp(r2, l1) != -1;
    }
    int checkSS(point k1, point k2, point k3, point k4) { // 判断线段 k1k2 和线段 k3k4 是否相交
        return intersect(k1.x, k2.x, k3.x, k4.x) && intersect(k1.y, k2.y, k3.y, k4.y) &&
            sign(cross(k3 - k1, k4 - k1)) * sign(cross(k3 - k2, k4 - k2)) <= 0 &&
            sign(cross(k1 - k3, k2 - k3)) * sign(cross(k1 - k4, k2 - k4)) <= 0;
    }
    db disSP(point k1, point k2, point q) { // 点 q 到线段 k1k2 的最短距离
        point k3 = proj(k1, k2, q);
        if (inmid(k1, k2, k3)) return q.dis(k3);
        else return min(q.dis(k1), q.dis(k2));
    }
    db disLP(point k1, point k2, point q) { // 点 q 到直线 k1k2 的最短距离
        point k3 = proj(k1, k2, q);
        return q.dis(k3);
    }
    db disSS(point k1, point k2, point k3, point k4) { // 线段 k1k2 和线段 k3k4 的最短距离
        if (checkSS(k1, k2, k3, k4)) return 0;
        else return min(min(disSP(k1, k2, k3), disSP(k1, k2, k4)),
            min(disSP(k3, k4, k1), disSP(k3, k4, k2)));
    }
    bool onLine(point k1, point k2, point q) { // 判断点 q 是否在直线 k1k2 上
        return sign(cross(k1 - q, k2 - q)) == 0;
    }
    bool onSegment(point k1, point k2, point q) { // 判断点 q 是否在线段 k1k2 上
        if (!onLine(k1, k2, q)) return false;
        return inmid(k1, k2, q);
    }
    void polarAngleSort(vector<point>& p, point t) { // p为待排序点集 t为极角排序中心
        sort(p.begin(), p.end(), [&](const point& k1, const point& k2) {
            return compareAngle(k1 - t, k2 - t);
            });
    }

    struct line { // 直线 / 线段类
        point p[2];
        line() {}
        line(point k1, point k2) { p[0] = k1, p[1] = k2; }
        point& operator [] (int k) { return p[k]; }
        point dir() { return p[1] - p[0]; } // 向量 p[0] -> p[1]
        bool include(point k) { // 判断点是否在直线上
            return sign(cross(p[1] - p[0], k - p[0])) > 0;
        }
        bool includeS(point k) { // 判断点是否在线段上
            return onSegment(p[0], p[1], k);
        }
        line push(db len) { // 向外（左手边）平移 len 个单位
            point delta = (p[1] - p[0]).turn90().unit() * len;
            return line(p[0] - delta, p[1] - delta);
        }
    };

    bool parallel(line k1, line k2) { // 判断是否平行
        return sign(cross(k1.dir(), k2.dir())) == 0;
    }
    bool sameLine(line k1, line k2) { // 判断是否共线
        return parallel(k1, k2) && parallel(k1, line(k2.p[0], k1.p[0]));
    }
    bool sameDir(line k1, line k2) { // 判断向量 k1 k2 是否同向
        return parallel(k1, k2) && sign(dot(k1.dir(), k2.dir())) == 1;
    }
    bool operator < (line k1, line k2) {
        if (sameDir(k1, k2)) return k2.include(k1[0]);
        return compareAngle(k1.dir(), k2.dir());
    }
    point getLL(line k1, line k2) {  // 求 k1 k2 两直线交点 不要忘了判平行！
        return getLL(k1[0], k1[1], k2[0], k2[1]);
    }
    bool checkpos(line k1, line k2, line k3) {  // 判断是否三线共点
        return k3.include(getLL(k1, k2));
    }

    struct circle { // 圆类
        point o;
        double r;
        circle() {}
        circle(point o_, double r_) : o(o_), r(r_) {}
        int inside(point k) {  // 判断点 k 和圆的位置关系
            return cmp(r, o.dis(k)); // 圆外:-1, 圆上:0, 圆内:1
        }
    };

    int checkposCC(circle k1, circle k2) { // 返回两个圆的公切线数量
        if (cmp(k1.r, k2.r) == -1) swap(k1, k2);
        db dis = k1.o.dis(k2.o);
        int w1 = cmp(dis, k1.r + k2.r), w2 = cmp(dis, k1.r - k2.r);
        if (w1 > 0) return 4; // 外离
        else if (w1 == 0) return 3; // 外切
        else if (w2 > 0) return 2;  // 相交
        else if (w2 == 0) return 1; // 内切
        else return 0; // 内离(包含)
    }
    vector<point> getCL(circle k1, point k2, point k3) { // 求直线 k2k3 和圆 k1 的交点
        // 沿着 k2->k3 方向给出 相切给出两个
        point k = proj(k2, k3, k1.o);
        db d = k1.r * k1.r - (k - k1.o).len2();
        if (sign(d) == -1) return {};
        point del = (k3 - k2).unit() * sqrt(max((db)0.0, d));
        return { k - del,k + del };
    }
    vector<point> getCC(circle k1, circle k2) { // 求圆 k1 和圆 k2 的交点
        // 沿圆 k1 逆时针给出, 相切给出两个
        int pd = checkposCC(k1, k2); if (pd == 0 || pd == 4) return {};
        db a = (k2.o - k1.o).len2(), cosA = (k1.r * k1.r + a -
            k2.r * k2.r) / (2 * k1.r * sqrt(max(a, (db)0.0)));
        db b = k1.r * cosA, c = sqrt(max((db)0.0, k1.r * k1.r - b * b));
        point k = (k2.o - k1.o).unit(), m = k1.o + k * b, del = k.turn90() * c;
        return { m - del,m + del };
    }
    vector<point> tangentCP(circle k1, point k2) { // 点 k2 到圆 k1 的切点 沿圆 k1 逆时针给出
        db a = (k2 - k1.o).len(), b = k1.r * k1.r / a, c = sqrt(max((db)0.0, k1.r * k1.r - b * b));
        point k = (k2 - k1.o).unit(), m = k1.o + k * b, del = k.turn90() * c;
        return { m - del,m + del };
    }
    vector<line> tangentOutCC(circle k1, circle k2) {
        int pd = checkposCC(k1, k2);
        if (pd == 0) return {};
        if (pd == 1) {
            point k = getCC(k1, k2)[0];
            return { line(k,k) };
        }
        if (cmp(k1.r, k2.r) == 0) {
            point del = (k2.o - k1.o).unit().turn90().getdel();
            return { line(k1.o - del * k1.r,k2.o - del * k2.r),
                line(k1.o + del * k1.r,k2.o + del * k2.r) };
        }
        else {
            point p = (k2.o * k1.r - k1.o * k2.r) / (k1.r - k2.r);
            vector<point> A = tangentCP(k1, p), B = tangentCP(k2, p);
            vector<line> ans; for (int i = 0; i < A.size(); i++)
                ans.push_back(line(A[i], B[i]));
            return ans;
        }
    }
    vector<line> tangentInCC(circle k1, circle k2) {
        int pd = checkposCC(k1, k2);
        if (pd <= 2) return {};
        if (pd == 3) {
            point k = getCC(k1, k2)[0];
            return { line(k, k) };
        }
        point p = (k2.o * k1.r + k1.o * k2.r) / (k1.r + k2.r);
        vector<point> A = tangentCP(k1, p), B = tangentCP(k2, p);
        vector<line> ans;
        for (int i = 0; i < (int)A.size(); i++) ans.push_back(line(A[i], B[i]));
        return ans;
    }
    vector<line> tangentCC(circle k1, circle k2) { // 求两圆公切线
        int flag = 0;
        if (k1.r < k2.r) swap(k1, k2), flag = 1;
        vector<line> A = tangentOutCC(k1, k2), B = tangentInCC(k1, k2);
        for (line k : B) A.push_back(k);
        if (flag) for (line& k : A) swap(k[0], k[1]);
        return A;
    }
    db getAreaUnionCT(circle k1, point k2, point k3) { // 圆 k1 与三角形 k2k3k1.o 的有向面积交
        point k = k1.o; k1.o = k1.o - k; k2 = k2 - k; k3 = k3 - k;
        int pd1 = k1.inside(k2), pd2 = k1.inside(k3);
        vector<point> A = getCL(k1, k2, k3);
        if (pd1 >= 0) {
            if (pd2 >= 0) return cross(k2, k3) / 2;
            return k1.r * k1.r * rad(A[1], k3) / 2 + cross(k2, A[1]) / 2;
        }
        else if (pd2 >= 0) {
            return k1.r * k1.r * rad(k2, A[0]) / 2 + cross(A[0], k3) / 2;
        }
        else {
            int pd = cmp(k1.r, disSP(k2, k3, k1.o));
            if (pd <= 0) return k1.r * k1.r * rad(k2, k3) / 2;
            return cross(A[0], A[1]) / 2 + k1.r * k1.r * (rad(k2, A[0]) + rad(A[1], k3)) / 2;
        }
    }
    circle getCircle(point k1, point k2, point k3) { // 三点确定一个圆
        db a1 = k2.x - k1.x, b1 = k2.y - k1.y, c1 = (a1 * a1 + b1 * b1) / 2;
        db a2 = k3.x - k1.x, b2 = k3.y - k1.y, c2 = (a2 * a2 + b2 * b2) / 2;
        db d = a1 * b2 - a2 * b1;
        point o = point(k1.x + (c1 * b2 - c2 * b1) / d, k1.y + (a1 * c2 - a2 * c1) / d);
        return circle(o, k1.dis(o));
    }
    circle minCircleCovering(vector<point> A) { // 最小圆覆盖 O(n)随机增量法
        random_shuffle(A.begin(), A.end());
        circle ans = circle(A[0], 0);
        for (int i = 1; i < A.size(); i++) {
            if (ans.inside(A[i]) == -1) {
                ans = circle(A[i], 0);
                for (int j = 0; j < i; j++) {
                    if (ans.inside(A[j]) == -1) {
                        ans.o = (A[i] + A[j]) / 2;
                        ans.r = ans.o.dis(A[i]);
                        for (int k = 0; k < j; k++) {
                            if (ans.inside(A[k]) == -1)
                                ans = getCircle(A[i], A[j], A[k]);
                        }
                    }
                }
            }
        }
        return ans;
    }

    struct polygon { // 多边形类
        int n; // 点数
        vector<point> p;
        polygon() {}
        polygon(vector<point> a) {
```

```
        n = (int)a.size();
        p = a;
    }
    db area() { // 多边形有向面积
        if (n < 3) return 0;
        db ans = 0;
        for (int i = 1; i < n - 1; i++)
            ans += cross(p[i] - p[0], p[i + 1] - p[0]);
        return 0.5 * ans;
    }
    int inConvexHull(point a) { // O(logn)判断点是否在凸包内 1内部 0边界 -1外部
        // 必须保证凸多边形是一个水平序凸包且不能退化
        // 退化情况 比如凸包退化成线段 可使用 onSegment() 函数特判
        auto check = [&](int x) {
            int ccw1 = counterclockwise(p[0], a, p[x]),
                ccw2 = counterclockwise(p[0], a, p[x + 1]);
            if (ccw1 == -1 && ccw2 == -1) return 1;
            else if (ccw1 == 1 && ccw2 == 1) return -1;
            else if (ccw1 == -1 && ccw2 == 1) return 0;
            else return 0;
        };
        if (counterclockwise(p[0], a, p[1]) <= 0 && counterclockwise(p[0], a, p.back()) >= 0) {
            int l = 1, r = n - 2, mid;
            while (l <= r) {
                mid = (l + r) >> 1;
                int chk = check(mid);
                if (chk == 1) l = mid + 1;
                else if (chk == -1) r = mid;
                else break;
            }
            int res = counterclockwise(p[mid], a, p[mid + 1]);
            if (res < 0) return 1;
            else if (res == 0) return 0;
            else return -1;
        }
        else return -1;
    }
};

int checkPolyP(polygon poly, point q) { // O(n)判断点是否在一般多边形内
    // 必须保证简单多边形的点按逆时针给出 返回 2 内部, 1 边界, 0 外部
    int pd = 0;
    for (int i = 0; i < poly.n; i++) {
        point u = poly.p[i], v = poly.p[(i + 1) % poly.n];
        if (onSegment(u, v, q)) return 1;
        if (cmp(u.y, v.y) > 0) swap(u, v);
        if (cmp(u.y, q.y) >= 0 || cmp(v.y, q.y) < 0) continue;
        if (sign(cross(u - v, q - v)) < 0) pd ^= 1;
    }
    return pd << 1;
}
bool checkConvexHull(polygon poly) { // 检测多边形是否是凸包
    int sgn = counterclockwise(poly.p[0], poly.p[1], poly.p[2]);
    for (int i = 1; i < poly.n; i++) {
```

```cpp
            int ccw = counterclockwise(poly.p[i], poly.p[(i + 1) % poly.n], poly.p[(i + 2) % poly.n]);
            if (sgn != ccw) return false;
        }
        return true;
    }
    db convexDiameter(polygon poly) { // 0(n)旋转卡壳求凸包直径 / 平面最远点对的平方
        int n = poly.n; // 请保证多边形是凸包
        db ans = 0;
        for (int i = 0, j = n < 2 ? 0 : 1; i < j; i++) {
            for (;; j = (j + 1) % n) {
                ans = max(ans, (poly.p[i] - poly.p[j]).len2());
                if (sign(cross(poly.p[i + 1] - poly.p[i], poly.p[(j + 1) % n] - poly.p[j])) <= 0) break;
            }
        }
        return ans;
    }

    vector<point> convexHull(vector<point> A, int flag = 1) { // 凸包 flag=0 不严格 flag=1 严格
        int n = A.size(); vector<point> ans(n + n);
        sort(A.begin(), A.end()); int now = -1;
        for (int i = 0; i < A.size(); i++) {
            while (now > 0 && sign(cross(ans[now] - ans[now - 1], A[i] - ans[now - 1])) < flag)
                now--;
            ans[++now] = A[i];
        }
        int pre = now;
        for (int i = n - 2; i >= 0; i--) {
            while (now > pre && sign(cross(ans[now] - ans[now - 1], A[i] - ans[now - 1])) < flag)
                now--;
            ans[++now] = A[i];
        }
        ans.resize(now);
        return ans;
    }
    polygon getConvexHull(vector<point> A, int flag = 1) { // 凸包 flag=0 不严格 flag=1
        return polygon(convexHull(A, flag));
    }
    vector<point> convexCut(vector<point> A, point k1, point k2) { // 半平面 k1k2 切凸包 A
        int n = A.size(); // 保留所有满足 k1 -> p -> k2 为逆时针方向的点
        A.push_back(A[0]);
        vector<point> ans;
        for (int i = 0; i < n; i++) {
            int ccw1 = counterclockwise(k1, k2, A[i]);
            int ccw2 = counterclockwise(k1, k2, A[i + 1]);
            if (ccw1 >= 0) ans.push_back(A[i]);
            if (ccw1 * ccw2 <= 0) ans.push_back(getLL(k1, k2, A[i], A[i + 1]));
        }
        return ans;
    }

    vector<line> getHL(vector<line>& L) { // 求半平面交 逆时针方向存储
        sort(L.begin(), L.end());
        deque<line> q;
        for (int i = 0; i < (int)L.size(); ++i) {
            if (i && sameDir(L[i], L[i - 1])) continue;
            while (q.size() > 1 && !checkpos(q[q.size() - 2], q[q.size() - 1], L[i]))
                q.pop_back();
            while (q.size() > 1 && !checkpos(q[1], q[0], L[i])) q.pop_front();
            q.push_back(L[i]);
        }
        while (q.size() > 2 && !checkpos(q[q.size() - 2], q[q.size() - 1], q[0])) q.pop_back();
        while (q.size() > 2 && !checkpos(q[1], q[0], q[q.size() - 1])) q.pop_front();
        vector<line> ans;
        for (int i = 0; i < q.size(); ++i) ans.push_back(q[i]);
        return ans;
    }

    db closestPoint(vector<point>& A, int l, int r) { // 最近点对, 先要按照 x 坐标排序
        if (r - l <= 5) {
            db ans = 1e20;
            for (int i = l; i <= r; ++i)
                for (int j = i + 1; j <= r; j++)
                    ans = min(ans, A[i].dis(A[j]));
            return ans;
        }
        int mid = l + r >> 1;
        db ans = min(closestPoint(A, l, mid), closestPoint(A, mid + 1, r));
        vector<point> B;
        for (int i = l; i <= r; i++)
            if (abs(A[i].x - A[mid].x) <= ans)
                B.push_back(A[i]);
        sort(B.begin(), B.end(), [&](const point& k1, const point& k2) {
            return k1.y < k2.y;
            });
        for (int i = 0; i < B.size(); i++)
            for (int j = i + 1; j < B.size() && B[j].y - B[i].y < ans; j++)
                ans = min(ans, B[i].dis(B[j]));
        return ans;
    }
}
using namespace geometry;
```

3.2 forever97

```
const double EPS = 1e-8;
//----------------------------------------------------------------------
// double cmp
int dcmp(double x) { return fabs(x) < EPS ? 0 : (x < 0 ? -1 : 1); }
struct Point {
    double x, y, z;
    Point() { x = y = z = 0; }
    Point(double x, double y, double z) : x(x), y(y), z(z) {}
    Point operator+(Point a) { return Point(x + a.x, y + a.y, z + a.z); }
    Point operator-(Point a) { return Point(x - a.x, y - a.y, z - a.z); }
    Point operator*(double k) { return Point(x * k, y * k, z * k); }
    Point operator/(double k) { return Point(x / k, y / k, z / k); }
    double operator*(Point a) { return x * a.x + y * a.y + z * a.z; }  // 点积
    Point operator^(Point a) {
        return Point(y * a.z - z * a.y, z * a.x - x * a.z, x * a.y - y * a.x);
    } // 叉积
    double length() { return sqrt(x * x + y * y + z * z); }
};
double Angle(Point a, Point b) { return acos(a * b / a.length() / b.length()); }
Point projection(Point v, Point u) {  // 向量v在u上投影
    double scalar = (v * u) / (u * u);
    return u * scalar;
}
Point projection(Point p, Point a, Point b,
                 Point c) {  // 点p在平面ABC上的投影
    Point u = (b - a) ^ (c - a), v = p - a;
    double scalar = (v * u) / (u * u);
    return p - (u * scalar);
}
double dist(Point p, Point a, Point b) {  // 点p到直线ab的距离
    p = p - a;
    Point proj = projection(p, b - a);
    return sqrt(p * p - proj * proj);
}
//点到线段
double DistanceToSegment(Point p, Point a, Point b) {
    if (a == b) return (p - a).length();
    Vector v1 = b - a, v2 = p - a, v3 = p - b;
    if (dcmp(v1 * v2) < 0)
        return v2.length();
    else if (dcmp(v1 * v3) > 0)
        return v3.length();
    else
        return (v1 ^ v2).length() / v1.length();
}
double area(Point a, Point b, Point c) {  // 三角形ABC的面积
    double h = dist(a, b, c);
    return (h * (b - c).length()) / 2;
}
double volume(Point x, Point y, Point z) {  // 三个向量构成的体积
    Point base = Point(y.y * z.z - y.z * z.y, y.z * z.x - y.x * z.z,
                       y.x * z.y - y.y * z.x);
    return fabs(x.x * base.x + x.y * base.y + x.z * base.z) / 3;
}
//--------------------------------------------------------------------
//空间直线
struct Line {
    Point a, b;
};
//空间直线间距离
double LineToLine(Line u, Line v, Point &tmp) {
    tmp = (u.a - u.b) ^ (v.a - v.b);
    return fabs((u.a - v.a) * tmp) / tmp.length();
}
//--------------------------------------------------------------------
//面： 点+法线
//面交线： 两面交线与两面法线均垂直，法线叉积为其方向矢量.
//角平分面： 法向量为两平面法向量相加(内角)或相减(外角).
struct Plane {
    Point p0, n;  // n:法线
    Plane() {}
    Plane(Point nn, Point pp0) {
        n = nn / nn.length();
        p0 = pp0;
    }
    Plane(Point a, Point b, Point c) {
        Point nn = (b - a) ^ (c - a);
        n = nn / nn.length();
        p0 = a;
    }
};
//角平分面
Plane jpfPlane(Point a1, Point a2, Point b, Point c) {
    Plane p1(a1, b, c), p2(a2, c, b);
    Point temp = p1.n + p2.n;  // 法向量为两平面法向量相加(内角)或相减(外角)
    return Plane(temp ^ (c - b), b);
}
//线面交点取线上任意两点
Point LinePlaneIntersection(Point p1, Point p2, Plane a) {
    Point p0 = a.p0;
    Point n = a.n, v = p2 - p1;
    double t = n * (p0 - p1) / (n * (p2 - p1));  //映射到法向量的比例
    return p1 + v * t;
}
//三面交点
Point PlaneInsertion(Plane a, Plane b, Plane c) {
    //两面交线与两面法线均垂直，法线叉积为其方向矢量
    Point nn = a.n ^ b.n, use = nn ^ a.n;
    Point st = LinePlaneIntersection(a.p0, a.p0 + use, b);  //得交线上一点
    return LinePlaneIntersection(st, st + nn, c);
}
double DistanceToPlane(Point p, Plane a) {
    Point p0 = a.p0, n = a.n;
    return fabs((p - p0) * n / n.length());
}
// 判定四点共面
bool isOnePlane(Point a, Point b, Point c, Point d) {
    double t = (d - a) * ((b - a) ^ (c - a));
    return dcmp(t) == 0;
}
//-------------------------------------------------------------------
//已知3点坐标，求平面ax+by+cz+d=0;
void getPlane(Point p1, Point p2, Point p3, double &a, double &b, double &c,
              double &d) {
    a = ((p2.y - p1.y) * (p3.z - p1.z) - (p2.z - p1.z) * (p3.y - p1.y));
    b = ((p2.z - p1.z) * (p3.x - p1.x) - (p2.x - p1.x) * (p3.z - p1.z));
    c = ((p2.x - p1.x) * (p3.y - p1.y) - (p2.y - p1.y) * (p3.x - p1.x));
    d = (0 - (a * p1.x + b * p1.y + c * p1.z));
}
/*
    题意：给定四点，求是否能够成四面体，若能则求出其内接圆心和半径
    分析：
        是否能构成四面体：
   三点成面的法线和另一点与三点中任一点相连的向量是否垂直？ 四面体内接球 球心：
   任意三个角平分面的交点 半径： 交点到任意面的距离
```

```
*/
int main() {
    Point p[4];
    while (~scanf("%lf%lf%lf", &p[0].x, &p[0].y, &p[0].z)) {
        for (int i = 1; i <= 3; i++)
            scanf("%lf%lf%lf", &p[i].x, &p[i].y, &p[i].z);
        if (isOnePlane(p[0], p[1], p[2], p[3])) {
            puts("0 0 0 0");
            continue;
        }
        Plane a = jpfPlane(p[3], p[2], p[1],
                           p[0]),  // 三个角平分面的交点即为圆心
            b = jpfPlane(p[3], p[0], p[1], p[2]),
              c = jpfPlane(p[3], p[1], p[0], p[2]);
        Plane d(p[0], p[1], p[2]);
        Point center = PlaneInsertion(a, b, c);
        double r = DistanceToPlane(center, d);
        printf("%.4f %.4f %.4f %.4f\n", center.x, center.y, center.z, r);
    }
}
```