目录

[TOC]

1 计算几何

1.1 二维几何

1.1.1 常用的函数和常量

```
const double eps = 1e-8;
const double inf = 1e20;
const double pi = acos(-1);
//实数域上的符号函数
inline int sgn(double x)
{
    if (fabs(x) < eps) return 0;
    if (x < 0) return -1;
    else return 1;
}
//实数域上的面积公式
inline double sqr(double x)
{
    return x * x;
}
```

1.1.2 笛卡尔域上的点运算

这里的点运算全部集合成结构体的描述方式。`struct Point`

```
struct Point
{
    double x, y;
    Point(){}    //创建一个平面点
    Point(double _x, double _y)  //对某个点赋值坐标
    {
        x = _x;
        y = _y;
    }
    bool operator == (Point b) const //重载 ==
    {
        return sgn(x - b.x) == 0 && sgn(y - b.y) == 0;
    }
    bool operator < (Point b) const  //重载 <，且第一关键字为 x
    {
        return sgn(x - b.x) == 0 ? sgn(y - b.y) : x < b.x;
    }
    Point operator - (const Point &b) const
    {
        return Point(x - b.x, y - b.y);
    }
    double operator ^ (const Point &b) const // 向量的叉积
    {
        return x * b.y - y * b.x;
    }
    double operator * (const Point &b) const //向量的点积
    {
        return x * b.x + y * b.y;
    }
    double len()  //返回长度（求三角形斜边长）
    {
        return hypot(x, y);
    }
    double len2()  //返回长度平方
    {
        return x * x + y * y;
    }
    double distance(Point b) //返回两点距离
    {
        return hypot(x - b.x, y - b.y);
    }
    Point operator + (const Point &b) const
```

```
{
    return Point(x + b.x, y + b.y);
}
//代数上的坐标放缩。
Point operator * (const double &k) const
{
    return Point(x * k, y * k);
}
Point operator / (const double &k) const
{
    return Point(x / k, y / k);
}
//计算 pa 和 pb 的夹角.
double rad(Point a, Point b)
{
    Point p = *this;
    return fabs(atan2(fabs((a - p) ^ (b - p)), (a - p) * (b - p)));
}
//化为长度为 r 的向量
Point trunc(double r)
{
    double l = len();
    if (!sgn(l)) return *this;
    r /= l;
    return Point(x * r, y * r);
}
//逆时针转 90 度
Point rotleft()
{
    return Point(-y, x);
}
//顺时针旋转 90 度
Point rotright()
{
    return Point(y, -x);
}
//绕点 p 旋转一个 angle 角度
Point rotate(Point p, double angle)
{
    Point v = (*this) - p;
    double c = cos(angle), s = sin(angle);
    return Point(p.x + v.x * c - v.y * s, p.y + v.x * s + v.y * c);
```

```
    }
};
```

1.1.3 对直线的操作

```
struct Line
{
    Point s, e;   //line 的讨论需要用到 Point
    Line(){}
    Line(Point _s, Point _e)
    {
        s = _s;
        e = _e;
    }
    bool operator == (Line v)
    {
        return (s == v.s) && (e == v.e);
    }
    //根据一个点和倾角确定一条直线
    Line(Point p, double angle)
    {
        s = p;
        if (sgn(ange- pi / 2) == 0) e = (s + Point(0, 1));
        else e = (s + Point(1, tan(angle)));
    }
    //ax + by + c = 0 型的直线
    Line(double a, double b, double c)
    {
        if (sgn(a) == 0) {
            s = Point(0, -c / b);
            e = Point(1, -c / b);
        }
        else if (sgn(b) == 0) {
            s = Point(-c / a, 0);
            e = Point(-c / a, 1);
        }
        else {
            s = Point(0, -c / b);
            e = Point(1, (-c - a) / b);
        }
    }
```

```
//求线段长度
double length()
{
    return s.distance(e);
}
//求直线倾角 (rad)
double angle()
{
    double k = atan2(e.y - s.y, e.x - s.x);
    if (sgn(k) < 0) k += pi;
    if (sgn(k - pi) == 0) k -= pi;
    return k;
}
/*
点和直线的关系:
1 : 在左侧
2 : 在右侧
3 : 在直线上
*/
int relation(Point p)
{
    int c = sgn((p - s) ^ (e - s));
    if (c < 0) return 1;
    else if (c > 0) return 2;
    else return 3;
}
//点在线段上的判断
bool pointonseg(Point p)
{
    return sgn((p - s) ^ (e - s)) == 0 && sgn((p - s) * (p - e)) <= 0;
}
//两向量平行 (对应的直线平行或重合)
bool parallel(Line v)
{
    return sgn((e - s) ^ (v.e - v.s)) == 0;
}
/*
两线段相交情况:(规范是指，两条线段之间只存在一个交点，而没有其他重合部分)
0: 不相交
1: 非规范相交
2: 规范相交
*/
```

```
int segcrossing(Line v)
{
    int d1 = sgn((e - s) ^ (v.s - s));
    int d2 = sgn((e - s) ^ (v.e - s));
    int d3 = sgn((v.e - v.s) ^ (s - v.s));
    int d4 = sgn((v.e - v.s) ^ (e - v.s));
    if ((d1 ^ d2) == -2 && (d2 ^ d4) == -2) return 2;
    return (d1 == 0 && sgn((v.s - s) * (v.s - e)) <= 0) ||
           (d2 == 0 && sgn((v.e - s) * (v.e - e)) <= 0) ||
           (d3 == 0 && sgn((s - v.s) * (s - v.e)) <= 0) ||
           (d4 == 0 && sgn((e - v.s) * (e - v.e)) <= 0);
}
//直线和线段的相交关系判断
//this line with segment v
//2: 规范相交
//1: 非规范相交
//0: 不相交
int linecrosseg(Line v)
{
    if ((*this).parallel(v)) return v.relation(s) == 3;
    return 2;
}
//两直线关系
//0: 平行
//1: 重合
//2: 相交
int linecrossline(Line v)
{
    if ((*this).parallel(v)) return v.relation(s) == 3;
    return 2;
}
//求两条直线的交点
//首先是保证不重合、不平行
Point crosspoint(Line v)
{
    double a1 = (v.e - v.s) ^ (s - v.s);
    double a2 = (v.e - v.s) ^ (e - v.s);
    return Point((s.x * a2 - e.x * a1) / (a2 - a1), (s.y * a2 - e.y * a1) / (a2 - a1));
}
//点到直线的距离
doule dispoint2line(Point p)
{
```

```
        return fabs((p - s) ^ (e - s)) / length();
    }
    //点到线段的距离
    double dispoint2seg(Point p)
    {
        if (sgn((p - s) * (e - s)) < 0 || sgn((p - e) * (s - e) < 0) return min(p.distance(s),
         ↪  p.distance(e));
        return dispoint2line(p);
    }
    //线段到线段的距离（前提是不相交）
    double disseg2seg(Line v)
    {
        return min(min(dispoint2seg(v.s), dispoint2seg(v.e)), min(v.dispoint2seg(s),
         ↪  v.dispoint2seg(e)));
    }
    //返回点 p 在直线上的投影点
    Point lineprog(Point p)
    {
        return s + (((e - s) * ((e - s) * (p - s))) / ((e - s).len2()));
    }
    //返回 p 关于直线的对称点
    Point pointbyline(Point p)
    {
        Point q = lineprog(p);
        return Point(2 * q.x - p.x, 2 * q.y - p.y);
    }
};
```

1.1.4 关于圆的操作

```
struct circle
{
    Point p; //圆心位置
    doule r; //半径
    circle(Point _p, double _r)
    {
        p = _p;
        r = _r;
    }
    circle(double x, double y, double _r)
    {
```

```
        p = Point(x, y);
        r = _r;
    }
    //求三角形的外接圆，利用两边的中垂线得圆心
    circle(Point a, Point b, Point c)
    {
        Line u = Line((a + b) / 2, ((a + b) / 2 + ((b - a).rotleft())));
        Line v = Line((b + c) / 2, ((b + c) / 2 + ((c - b).rotleft())));
        p = u.crosspoint(v);
        r = p.distance(a);
    }
    //三角形的内切圆，这里的 t 是为了区分外接圆和内切圆
    circle(Point a, Point b, Point c, bool t)
    {
        Line u, v;
        double m = atan2(b.y - a.y, b.x - a.x), n = atan2(c.y - a.y, c.x - a.x);
        u.s = a;
        u.e = u.s + Point(cos((n + m) / 2), sin((n + m) / 2));
        v.s = b;
        m = atan2(a.y - b.y, a.x - b.x), n = atan2(c.y - b.y, c.x - b.x);
        v.e = v.s + Point(cos((n + m) / 2), sin((n + m) / 2));
        p = u.crosspoint(v);
        r = Line(a, b).dispoint2seg(p);
    }
    bool operator == (circle v)
    {
        return (v.p == p) && sgn(r - v.r) == 0;
    }
    bool operator < (circle v) const
    {
        return ((p < v.p) || ((p == v.p) && sgn(r - v.r) < 0));
    }
    double are()
    {
        return pi * r * r;
    }
    double circlelen()
    {
        return 2 * pi * r;
    }
    //点和圆的关系
    //0: 园外
```

```
//1: 圆上
//2: 圆内
int relation(Point b)
{
    double dst = b.distance(p);
    if (sgn(dst - r) < 0) return 2;
    else if (sgn(dst - t) == 0) return 1;
    else return 0;
}
//线段和圆的关系
int relationseg(Line v)
{
    double dis = v.dispoint2seg(p);
    if (sgn(dis - r) < 0) return 2;
    else if (sgn(dis - r) == 0) return 1;
    else return 0;
}
//直线和圆的关系
int relationline(Line v)
{
    double dis = v.dispoint2line(p);
    if (sgn(dis - r) < 0) return 2;
    else if (sgn(dis - r) == 0) return 1;
    else return 0;
}
//两个圆的关系
//5; 相离
//4: 外切
//3: 相交
//2: 内切
//1: 内含
int relationcircle(circle v)
{
    double d = p.distance(v.p);
    if (sgn(d - r - v.r) > 0) return 5;
    else if (sgn(d - r - v.r) == 0) return 4;
    double l = fabs(r - v.r);
    if (sgn(d - r - v.r) < 0 && sgn(d - l) > 0) return 3;
    if (sgn(d - l) == 0) return 2;
    if (sgn(d - l) < 0) return 1;
}
//求直线和圆的交点，返回交点个数
```

```
    int pointcrossline(Line v, Point &p1, Point &p2){
        if(!(*this).relationline(v)) return 0;
        Point a = v.lineprog(p);
        double d = v.dispointtoline(p);
        d = sqrt(r * r - d * d);
        if(sgn(d) == 0) {
            p1 = a;
            p2 = a;
            return 1;
        }
        p1 = a + (v.e - v.s).trunc(d);
        p2 = a - (v.e - v.s).trunc(d);
        return 2;
    }
    //求两个圆的交点，返回 0 表示没有交点，返回 1 是一个交点，2 是两个交点
    int pointcrosscircle(circle v,Point &p1,Point &p2)
    {
        int rel = relationcircle(v);
        if(rel == 1 || rel == 5)return 0;
        double d = p.distance(v.p);
        double l = (d * d + r * r - v.r * v.r) / (2 * d);
        double h = sqrt(r * r - l * l);
        Point tmp = p + (v.p - p).trunc(l);
        p1 = tmp + ((v.p - p).rotleft().trunc(h));
        p2 = tmp + ((v.p - p).rotright().trunc(h));
        if(rel == 2 || rel == 4) return 1;
        return 2;
    }
}
```

1.2 凸包问题

什么是凸包？可以理解为用一根橡皮筋把平面上的所有点都围住，凸包具有面积和周长的最佳性。构建凸包的方法我们采用的是 Andrew 算法，时间复杂度 $O(nlogn)$。

```
Point p[maxn];   //存放点集
Point ans[maxn]; //存放凸包
void Andrew()
{
    sort (p, p + n);
    int p1 = 0, p2;
    for (int i = 0,i < n;i ++) {
```

```
        while (p1 > 1 && sgn(cross(ans[p1 - 1] - ans[p1 - 2], p[i] - ans[p1 - 2])) <= 0) p1 --;
         ↪  //叉积
        ans[p1 ++] = p[i];
    }
    p2 = p1;
    for (int i = n - 2;i >= 0;i --) {
        while (p2 > p1 && sgn(cross(ans[p2 - 1] - ans[p2 - 2], p[i] - ans[p2 - 2])) <= 0) p2 --;
        ans[p2 ++] = p[i];
    }
    if (n < 1) p2 --;
    //求凸包的周长
    double targe = 0.0;
    for (int i = 0;i < p2;i ++) targe += distance(ans[i], ans[i + 1]);
    return ;
}
```

1.3 旋转卡壳

旋转卡壳算法在凸包算法的基础上，通过枚举凸包上某一条边的同时维护其他需要的点，能够在线性时间内求解如凸包直径、最小矩形覆盖等和凸包性质相关的问题。

1.3.1 求凸包直径

给定平面上 n 个点，求所有点对之间的**最长距离**。首先把凸包的**节点编号**存在一个栈里，第一个和最后一个的编号相同。

```
inline ll get_length()
{
    ll mx = 0ll;
    if (top == 0) mx = 0;
    else if (top == 1) mx = dis(ans[0], ans[1]);
    else {
        ans[top] = ans[0];
        int j = 2;
        for (int i = 0;i < top;i ++) {
            while (cross(ans[i + 1] - ans[i], ans[j] - ans[i + 1]) < cross(ans[i + 1] - ans[i],
             ↪  ans[(j + 1) % top] - ans[i + 1])) j = (j + 1) % top;
            mx = max(mx, max(dis(ans[j], ans[i]), dis(ans[j], ans[i + 1])));
        }
    }
    return mx;
}
```

1.3.2 最小矩形覆盖

给定一些点的坐标，求能够**覆盖所有点的最小面积的矩形**

```
double pf(double x) return x * x;
double dis(int p, int q) return pf(a[p].x - a[q].x) + pf(a[p].y - a[q].y);
double sqr(int p, int q, int y) return fabs((a[q] - a[p]) ^ (a[y] - a[q])); // 叉积
double dot(int p, int q, int y) return fabs((a[q] - a[p]) * (a[y] - a[q])); //点积
void minRectangle()
{
    int j = 3, l = 2, r = 2;
    double t1, t2, t3, ans = 2e20;
    for (int i = 1;i <= top;i ++) {
        while (sqr(sta[i], sta[i + 1], sta[j]) <= sqr(sta[i], sta[i + 1], sta[j % top + 1])) j = j %
         ↪ top + 1;
        while (dot(sta[i + 1], sta[r % top + 1], sta[i]) >= dot(sta[i + 1], sta[r], sta[i])) r = r %
         ↪ top + 1;
        if (i == 1) l = r;
        while (dot(sta[i + 1], sta[l % top + 1], sta[i]) <= dot(sta[i + 1], sta[l], sta[i])) l = l %
         ↪ top + 1;
        t1 = sqr(sta[i], sta[i + 1], sta[j]);
        t2 = dot(sta[i + 1], sta[r], sta[i]) + dot(sta[i + 1], sta[l], sta[i]);
        t3 = dot(sta[i + 1], sta[i + 1], sta[i]);
        ans = min(ans, t1 * t2 / t3);
    }
}
```

1.4 平面最近点对

在这里介绍一种时间复杂度为 $O(nlognlogn)$ 的算法求解二维平面上的两点间最短距离。其实，这里用到了分治的思想。将所给平面上 n 个点的集合 S 分成两个子集 S1 和 S2，每个子集中约有 n/2 个点。然后在每个子集中递归地求最接近的点对。

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
using namespace std;
const double inf = 1e20;
const int maxn = 100005;

struct Point{
```

```
    double x, y;
}point[maxn];

int n, mpt[maxn];

//以 x 为基准排序
bool cmpxy(const Point& a, const Point& b)
{
    if (a.x != b.x)
        return a.x < b.x;
    return a.y < b.y;
}

bool cmpy(const int& a, const int& b)
{
    return point[a].y < point[b].y;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

double dis(int i, int j)
{
    return sqrt((point[i].x - point[j].x)*(point[i].x - point[j].x) + (point[i].y -
     ↪ point[j].y)*(point[i].y - point[j].y));
}

double Closest_Pair(int left, int right){
    double d = inf;
    if (left == right)
        return d;
    if (left + 1 == right)
        return dis(left, right);
    int mid = (left + right) >> 1;
    double d1 = Closest_Pair(left, mid);
    double d2 = Closest_Pair(mid + 1, right);
    d = min(d1, d2);
    int i, j, k = 0;
    //分离出宽度为 d 的区间
    for (i = left; i <= right; i++){
```

```
        if (fabs(point[mid].x - point[i].x) <= d)
            mpt[k++] = i;
    }
    sort(mpt, mpt + k, cmpy);
    //线性扫描
    for (i = 0; i < k; i++){
        for (j = i + 1; j < k && point[mpt[j]].y - point[mpt[i]].y<d; j++){
            double d3 = dis(mpt[i], mpt[j]);
            if (d > d3)    d = d3;
        }
    }
    return d;
}

int main(){
    while (~scanf("%d", &n) && n){
        for (int i = 0; i < n; i++)
            scanf("%lf %lf", &point[i].x, &point[i].y);
        sort(point, point + n, cmpxy);
        printf("%.2lf\n", Closest_Pair(0, n - 1));
    }
    return 0;
}
```