## 一.计算几何模板

### 1.1 点+向量

#### 1.1.1 Point(Vector)类型

// Need: sgn()

typedef struct Point {

    double x, y;

    Point(double x = 0, double y = 0) : x(x), y(y) {}  // 构造函数, 初始值为 0

    // 重载运算符

    // 点 - 点 = 向量(向量AB = B - A)

    Point operator- (const Point &B) const { return Point(x - B.x, y - B.y); }

    // 点 + 点 = 点, 点 + 向量 = 向量, 向量 + 向量 = 向量

    Point operator+ (const Point &B) const { return Point(x + B.x, y + B.y); }

    // 向量 × 向量 (叉积)

    double operator^ (const Point &B) const { return x \* B.y - y \* B.x; }

    // 向量 · 向量 (点积)

    double operator\* (const Point &B) const { return x \* B.x + y \* B.y; }

    // 点 \* 数 = 点, 向量 \* 数 = 向量

    Point operator\* (const double &B) const { return Point(x \* B, y \* B); }

    // 点 / 数 = 点, 向量 / 数 = 向量

    Point operator/ (const double &B) const { return Point(x / B, y / B); }

    // 判断大小, 一般用于排序

    bool operator< (const Point &B) const { return x < B.x || (x == B.x && y < B.y); }

    // 判断相等, 点 == 点, 向量 == 向量, 一般用于判断和去重

    bool operator== (const Point &B) const { return sgn(x - B.x) == 0 && sgn(y - B.y) == 0; }

    // 判断不相等, 点 != 点, 向量 != 向量

    bool operator!= (const Point &B) const { return sgn(x - B.x) || sgn(y - B.y); }

} Vector;

#### 1.1.2 点积(Dot)

// 向量 · 向量 (点积)

double operator\* (Vector &A, Vector &B) { return A.x \* B.x + A.y \* B.y; }

#### 1.1.3 叉积(Cross)

// 向量 × 向量 (叉积)

double operator^ (Vector &A, Vector &B) { return A.x \* B.y - A.y \* B.x; }

#### 1.1.4 两点间距离(Dist)

// Need: (-, \*)

// 使用时请注意精度误差

double dist(Point a, Point b) { return sqrtl((a - b) \* (a - b)); }

#### 1.1.5 向量的模(Len)

// Need: (\*)

// 使用时请注意精度误差

double len(Vector A) { return sqrtl(A \* A); }

#### 1.1.6 单位向量(Norm)

// Need: (/), len()

// len函数存在sqrtl的使用，请注意精度误差

Vector norm(Vector A) { return A / len(A); }

#### 1.1.7 两向量的夹角(Angle)

// Need: (\*), len(), PI

// 所调用的len函数使用sqrtl，请注意精度误差

double Angle(Vector A, Vector B) {

    double t = acos((A \* B) / len(A) / len(B));

    return t;               // 返回 [0, π]

    return t \* (180 / PI);  // 返回 [0, 180] (角度)

}

#### 1.1.8 判断点在向量的哪边

// Need: (-, ^), sgn()

// 点在直线上, 返回 0 (三点共线)

// 点在直线的逆时针方向, 返回 1

// 点在直线的顺时针方向, 返回 -1

// 点 a, b (向量ab) 所在的直线和点 c

// 使用的时候要注意 a 和 b 的顺序, 有时顺序不同, 结果不同

int Cross(Point a, Point b, Point c) { return sgn((b - a) ^ (c - a)); }

// 两种分情况使用

double Cross(Point a, Point b, Point c) { return (b - a) ^ (c - a); }

#### 1.1.9 逆转角(Rotate)

// Need: (\*, ^)

// 向量 A 和要逆时针转的角度 [0, PI]

// PI / 2, 90度

Vector Rotate(Vector A, double b) {

    Vector B(sin(b), cos(b));

    return Vector(A ^ B, A \* B);

}

### **1.2 线**

#### 1.2.1 Line

struct Line {

    Point s, e;

    Line() {}

    Line(Point x, Point y):s(x), e(y) {}

};

#### 1.2.2 判断三点共线(In\_one\_line)

// Need: sgn(), 操作符重载(-, ^)

bool In\_one\_line(Point A, Point B, Point C) { return !sgn((B - A) ^ (C - B)); }

#### 1.2.3 点到直线的距离(Dist\_point\_to\_line)

// Need: (-, ^), len()

// 点 P 到直线 AB 的距离

double Dist\_point\_to\_line(Point P, Point A, Point B) {

    Vector v1 = B - A, v2 = P - A;

    return fabs((v1 ^ v2) / len(v1));

}

#### 1.2.4 点到线段的距离(Dist\_point\_to\_seg)

// Need: 操作符重载(==, -, \*, ^), len(), sgn()

double Dist\_point\_to\_seg(Point P, Point A, Point B) {

    if (A == B) return len(P - A);      // 如果重合, 那么就是两点的距离

    Vector v1 = B - A, v2 = P - A, v3 = P - B;

    if (sgn(v1 \* v2) < 0) return len(v2);   // AP 最短

    if (sgn(v1 \* v3) > 0) return len(v3);   // BP 最短

    return fabs((v1 ^ v2) / len(v1));       // 垂线

}

#### 1.2.5 判断点是否在线段上(OnSegment)

// Need: (-, \*, ^), sgn()

bool OnSegment(Point P, Point A, Point B) {

    Vector PA = A - P, PB = B - P;

    return sgn(PA ^ PB) == 0 && sgn(PA \* PB) <= 0;  // <=, 包括端点; <, 不包括端点

}

#### 1.2.6 判断直线与线段是否相交(Intersect\_line\_seg)

// Need: Cross()

// 相交, 返回 true

// 不相交, 返回 false

// 直线 ab 与线段 cd

bool Intersect\_line\_seg(Point a, Point b, Point c, Point d) {

    return Cross(a, b, c) \* Cross(a, b, d) <= 0;

}

#### 1.2.7 判断两直线平行(Line\_parallel)

// Need: (-, ^), sgn()

// 返回true: 平行/重合, false: 相交

bool Line\_parallel(Line A, Line B) { return sgn((A.s - A.e) ^ (B.s - B.e)) == 0; }

#### 1.2.8 求两直线交点

Point Intersection\_line(Point a, Point b, Point c, Point d) {

    Vector u = b - a, v = d - c;

    double t = ((a - c) ^ v) / (v ^ u);

    return a + u \* t;

}

### 1.3 多边形

#### 1.3.1 三角形面积

**海伦公式**

// Need: 操作符重载(-), len()

double Triangle\_area(Point A, Point B, Point C) {

    double a = len(A - B), b = len(A - C), c = len(B - C);

    double p = (a + b + c) / 2;

    return sqrtl(p \* (p - a) \* (p - b) \* (p - c));

}

**利用向量点叉运算**

// Need: (-, ^)

double Triangle\_area2(Point A, Point B, Point C) {

    return fabs((B - A) ^ (C - A)) / 2;

}

#### 1.3.2 三角形四心

外心：三边中垂线的交点，到三角形三个顶点的距离相同（外接圆圆心）。

内心：角平分线的交点，到三角形三边的距离相同（内切圆圆心）。

垂心：三条垂线的交点。

重心：三条中线的交点，到三角形三顶点距离的平方和最小的点，三角形内到三边距离之积最大的点。

三角形重心是三点各坐标的平均值

#### 1.3.3 求正多边形面积(Polygon\_area)

// Need: (-, ^)

// 因为叉积求得的三角形面积是有向的, 在外面的面积可以正负抵消掉

// 所以能够求任意多边形面积(凸, !凸)

// p[]下标从 0 开始, 长度为 n

double Polygon\_area(Point \*p, int n) {

    double area = 0;

    for (int i = 1; i <= n - 2; i++)

        area += (p[i] - p[0]) ^ (p[i + 1] - p[0]);

    return fabs(area / 2);  // 无向面积

    return area / 2;        // 有向面积

}

// Need: (^)

// 原理和上面相同, 不过是把原点(0, 0) 作为被指向点

// p[] 下标从 0 开始, 长度为 n

double Polygon\_area(Point \*p, int n) {

    double area = 0;

    for (int i = 0, j = n - 1; i < n; j = i++)

        area += (p[j] ^ p[i]);

    return fabs(area / 2);  // 无向面积

    return area / 2;        // 有向面积

}

#### 1.3.4 判断点是否在多边形内

// Need: sgn(), OnSegment()

// 适用于任意多边形, 不用考虑精度误差和多边形的给出顺序

// 点在多边形边上, 返回 -1

// 点在多边形内, 返回 1

// 点在多边形外, 返回 0

// p[] 的下标从 0 开始, 长度为 n

int InPolygon(Point P, Point \*p, int n) {

    bool flag = false;      // 相当于计数

    for (int i = 0, j = n - 1; i < n; j = i++) {

        Point p1 = p[i], p2 = p[j];

        if (OnSegment(P, p1, p2)) return -1;

        if (sgn(P.y - p1.y) > 0 == sgn(P.y - p2.y) > 0) continue;

        if (sgn((P.y - p1.y) \* (p1.x - p2.x) / (p1.y - p2.y) + p1.x - P.x) > 0)

            flag = !flag;

    }

    return flag;

}

#### 1.3.5 判断凸多边形

// Need: (-, ^), sgn()

// 顶点必须按顺时针(或逆时针)给出, 允许共线边

// p[] 下标从 0 开始, 长度为 n

bool Is\_contex(Point \*p, int n) {

    bool s[3] = {0, 0, 0};

    for (int i = 0, j = n - 1, k = n - 2; i < n; k = j, j = i++) {

        int cnt = sgn((p[i] - p[j]) ^ (p[k] - p[j])) + 1;

        s[cnt] = true;

        if (s[0] && s[2]) return false;

    }

    return true;

}

### **1.4 圆**

#### 1.4.1 Circle

struct Circle {

    Point o;

    double r;

    Circle(Point \_o = Point(), double \_r = 0) : o(\_o), r(\_r) {}

    // 圆的面积

    double Circle\_S() { return PI \* r \* r; }

    // 圆的周长

    double circle\_C() { return 2 \* PI \* r; }

};

#### 1.4.2 扇形面积(SectorArea)

// Need: (^), Angle(), sgn()

// Angle使用sqrtl，请注意精度误差

// 扇形的两向量A, B 和圆的半径 R ，请注意是优弧还是劣弧的扇形

double SectorArea(Point A, Point B, double R) {

    double angle = Angle(A, B);

    return fabs(R \* R \* angle / 2);

}

#### 1.4.3 点与圆的位置关系(Point\_with\_circle)

// Need: sgn(), dist()

// 点在圆上, 返回 0

// 点在圆外, 返回 -1

// 点在圆内, 返回 1

int Point\_with\_circle(Point p, Circle c) {

    double d = dist(p, c.o);

    if (sgn(d - c.r) == 0) return 0;

    if (sgn(d - c.r) > 0) return -1;

    return 1;

}

#### 1.4.4 直线与圆的关系(Line\_with\_circle)

// Need: sgn(), Dist\_point\_to\_line()

// 相切, 返回 0

// 相交, 返回 1

// 相离, 返回 -1

int Line\_with\_circle(Point A, Point B, Circle c) {

    double d = Dist\_point\_to\_line(c.o, A, B);

    if (sgn(d - c.r) == 0) return 0;

    if (sgn(d - c.r) > 0) return -1;

    return 1;

}

#### 1.4.5 直线与圆的交点(Intersection\_line\_circle)

// Need: (-, +, \*P, \*D, /), sgn()

// 直线与圆相交, 返回两点

// 直线与圆相切, 返回两个一样的相切点

pair<Point, Point> Intersection\_line\_circle(Point A, Point B, Circle c) {

    Vector AB = B - A;

    Vector pr = A + AB \* ((c.o - A) \* AB / (AB \* AB));

    double base = sqrt(c.r \* c.r - ((pr - c.o) \* (pr - c.o)));

    if (sgn(base) == 0) return make\_pair(pr, pr);

    Vector e = AB / sqrt(AB \* AB);

    return make\_pair(pr + e \* base, pr - e \* base);

}

#### 1.4.6 圆与圆的位置关系(Circle\_with\_circle)

// Need: dist()

// 相离, 返回 -1

// 外切, 返回 0

// 内切(A 包含 B), 返回 1

// 内切(B 包含 A), 返回 2

// 内含(A 包含 B), 返回 3

// 内含(B 包含 A), 返回 4

// 相交, 返回 5

int Circle\_with\_circle(Circle A, Circle B) {

    double len1 = dist(A.o, B.o);

    double len2 = A.r + B.r;

    if (sgn(len1 - len2) > 0) return -1;

    if (sgn(len1 - len2) == 0) return 0;

    if (sgn(len1 + len2 - 2 \* A.r) == 0) return 1;

    if (sgn(len1 + len2 - 2 \* B.r) == 0) return 2;

    if (sgn(len1 + len2 - 2 \* A.r) < 0) return 3;

    if (sgn(len1 + len2 - 2 \* B.r) < 0) return 4;

    return 5;

}

1.4.7 圆与圆的交点(Intersection\_circle\_circle)

// Need: (-, +), len()

// 相交, 返回两点坐标

// 相切, 返回两个一样的相切点

// 要先判断是否相交或相切再调用

pair<Point, Point> Intersection\_circle\_circle(Circle A, Circle B) {

    Vector AB = B.o - A.o;

    double d = len(AB);

    double a = acos((A.r \* A.r + d \* d - B.r \* B.r) / (2.0 \* A.r \* d));

    double t = atan2(AB.y, AB.x);

    Vector x(A.r \* cos(t + a), A.r \* sin(t + a));

    Vector y(A.r \* cos(t - a), A.r \* sin(t - a));

    return make\_pair(A.o + x, A.o + y);

}

#### 1.4.7 求圆外一点对圆的两个切点(TangentPoint\_point\_circle)

// Need: (-, \*, ^, +), dist()

// 返回两个切点坐标

// 一定要先判断点在圆外, 然后再调用

pair<Point, Point> TangentPoint\_point\_circle(Point p, Circle c) {

    double d = dist(p, c.o);

    double l = sqrt(d \* d - c.r \* c.r);

    Vector e = (c.o - p) / d;

    double angle = asin(c.r / d);

    Vector t1(sin(angle), cos(angle));

    Vector t2(sin(-angle), cos(-angle));

    Vector e1(e ^ t1, e \* t1);

    Vector e2(e ^ t2, e \* t2);

    e1 = e1 \* l + p;

    e2 = e2 \* l + p;

    return make\_pair(e1, e2);

}

#### 1.4.8 求三角形的外接圆(get\_circumcircle)

// Need: dist()

Circle get\_circumcircle(Point A, Point B, Point C) {

    double Bx = B.x - A.x, By = B.y - A.y;

    double Cx = C.x - A.x, Cy = C.y - A.y;

    double D = 2 \* (Bx \* Cy - By \* Cx);

    double x = (Cy \* (Bx \* Bx + By \* By) - By \* (Cx \* Cx + Cy \* Cy)) / D + A.x;

    double y = (Bx \* (Cx \* Cx + Cy \* Cy) - Cx \* (Bx \* Bx + By \* By)) / D + A.y;

    Point P(x, y);

    return Circle(P, dist(A, P));

}

#### 1.4.9 求三角形的内接圆(get\_incircle)

Circle get\_incircle(Point A, Point B, Point C) {

    double a = dist(B, C);

    double b = dist(A, C);

    double c = dist(A, B);

    Point p = (A \* a + B \* b + C \* c) / (a + b + c);

    return Circle(p, Dist\_point\_to\_line(p, A, B));

}

### 1.5 综合通用

#### 1.5.1 极角排序

// Need: (-, ^), len(), sgn()

// 排序常数大, 但精度高

Point p[N]; // 要排序的点

Point o(0, 0);  // 极点自定义

// 获取象限 (0, 1, 2, 3)

int Quadrant(Vector p) { return sgn(p.y < 0) << 1 | sgn(p.x < 0) ^ sgn(p.y < 0); }

// 比较函数

bool cmp(Point a, Point b) {

    Vector p = a - o, q = b - o;

    int x = Quadrant(p), y = Quadrant(q);

    if (x == y) {

        if (sgn(p ^ q) == 0) return len(p) < len(q);

        return sgn(p ^ q) > 0;

    }

    return x < y;

}

#### 1.5.2 凸包(Andrew算法)

// Need: (<), Cross()

Point s[N]; // 用来存凸包多边形的顶点

int top = 0;

// 点集 p[] 的下标从 1 开始, 长度为 n

void Andrew(Point \*p, int n) {

    sort(p + 1, p + n + 1);

    for (int i = 1; i <= n; i++) {  // 下凸包

        while (top > 1 && Cross(s[top - 1], s[top], p[i]) <= 0) top--;

        s[++top] = p[i];

    }

    int t = top;

    for (int i = n - 1; i >= 1; i--) {  // 上凸包

        while (top > t && Cross(s[top - 1], s[top], p[i]) <= 0) top--;

        s[++top] = p[i];

    }

    top--;  // 因为首尾都会加一次第一个点, 所以去掉最后一个

}

#### 1.5.3 最小覆盖圆问题(get\_min\_circle)

// Need: (+, /), sgn(), dist(), get\_circumcircle()

// p[] 下标从 0 开始, 长度为 n

Circle get\_min\_circle(Point \*p, int n) {

    // 随机化, 防止被卡

    for (int i = 0; i < n; i++) swap(p[rand() % n], p[rand() % n]);

    Point o = p[0];

    double r = 0;

    for (int i = 0; i < n; i++) {

        if (sgn(dist(o, p[i]) - r) <= 0) continue;

        o = (p[i] + p[0]) / 2;

        r = dist(p[i], p[0]) / 2;

        for (int j = 1; j < i; j++) {

            if (sgn(dist(o, p[j]) - r) <= 0) continue;

            o = (p[i] + p[j]) / 2;

            r = dist(p[i], p[j]) / 2;

            for (int k = 0; k < j; k++) {

                if (sgn(dist(o, p[k]) - r) <= 0) continue;

                o = get\_circumcircle(p[i], p[j], p[k]).o;

                r = dist(o, p[i]);

            }

        }

    }

    return Circle(o, r);

}

#### 1.5.4 平面上n个圆的面积并

#include <bits/stdc++.h>

using namespace std;

const int N = 1010;

const double eps = 1e-8;

const double PI = acos(-1.0);

inline int sgn(double x) { return x < -eps ? -1 : x > eps; }

typedef struct Point {

    double x, y;

    Point(double xx = 0, double yy = 0): x(xx), y(yy){}

    Point operator-(const Point &B) const { return Point(x - B.x, y - B.y); }

    double operator\*(const Point &B) const { return x \* B.x + y \* B.y; }

    bool operator< (const Point &B) const { return x < B.x || (x == B.x && y < B.y); }

    bool operator== (const Point &B) const { return !sgn(x - B.x) && !sgn(y - B.y); }

} Vector;

double dist(Point a, Point b) { return sqrt((a - b) \* (a - b)); }

struct Circle {

    Point o;

    double r;

    Circle(Point oo = Point(), double rr = 0):o(oo), r(rr){}

    bool operator<(const Circle &B) const { return o < B.o || o == B.o && r < B.r; }

    bool operator==(const Circle &B) const { return o == B.o && !sgn(r - B.r); }

};

Circle c[N];

pair<double, int> st[N << 1];

int n;

double cal(int x) {

    int top = 0, cnt = 0;

    for (int i = 1; i <= n; i++) {

        if (i == x) continue;

        double dis = dist(c[i].o, c[x].o);

        double r1 = c[x].r, r2 = c[i].r;

        if (sgn(r1 + dis - r2) <= 0) return 0.0;

        if (sgn(r2 + dis - r1) <= 0 || sgn(r2 + r1 - dis) <= 0) continue;

        double del = acos((r1 \* r1 + dis \* dis - r2 \* r2) / (2 \* r1 \* dis));

        double angle = atan2(c[i].o.y - c[x].o.y, c[i].o.x - c[x].o.x);

        double l = angle - del, r = angle + del;

        if (sgn(l + PI) < 0) l += 2 \* PI;

        if (sgn(r - PI) >= 0) r -= 2 \* PI;

        if (sgn(l - r) > 0) cnt++;

        st[++top] = {l, 1}, st[++top] = {r, -1};

    }

    st[0] = {-PI, 0}, st[++top] = {PI, 0};

    sort(st + 1, st + top + 1);

    double res = 0;

    for (int i = 1; i <= top; cnt += st[i++].second) {

        if (cnt) continue;

        Point o = c[x].o;

        double r = c[x].r, t1 = st[i - 1].first, t2 = st[i].first;

        res += r \* (r \* (t2 - t1) + o.x \* (sin(t2) - sin(t1)) - o.y \* (cos(t2) - cos(t1)));

    }

    return res;

}

double get\_area() {

    sort(c + 1, c + n + 1);

    n = unique(c + 1, c + n + 1) - c - 1;

    double ans = 0;

    for (int i = 1; i <= n; i++) ans += cal(i);

    return ans / 2;

}

int main() {

    cin >> n;

    for (int i = 1; i <= n; i++) scanf("%lf%lf%lf", &c[i].o.x, &c[i].o.y, &c[i].r);

    printf("%.3lf\n", get\_area());

    return 0;

}

#### 1.5.5 圆与多边形的面积交

// Need: (-, +, \*D, \*, ^, /), sgn(), Intersection\_line(点向量版), OnSegment(), Rotate()

// SectorArea(), Angle(), norm(), len(), dist(),

// 返回圆点到 ab 线段的距离, 并带回圆与线段的交点 pa, pb

double getDP2(Point a, Point b, Circle c, Point &pa, Point &pb) {

    Point o = c.o;

    double R = c.r;

    Point e = Intersection\_line(a, b - a, o, Rotate(b - a, PI / 2));    // 垂足点

    double d = dist(o, e);

    if (!OnSegment(e, a, b)) d = min(dist(o, a), dist(o, b));

    if (R <= d) return d;

    double Len = sqrt(R \* R - dist(o, e) \* dist(o, e));

    pa = e + norm(a - b) \* Len;

    pb = e + norm(b - a) \* Len;

    return d;

}

double getArea(Point a, Point b, Circle C) {    // 面积的交

    Point o = C.o;

    double R = C.r;

    if (sgn(a ^ b) == 0) return 0;  // 共线

    double da = dist(o, a), db = dist(o, b);

    if (sgn(R - da) >= 0 && sgn(R - db) >= 0) return (a ^ b) / 2;   // ab 在圆内

    Point pa, pb;

    double d = getDP2(a, b, C, pa, pb);

    if (sgn(R - d) <= 0) return SectorArea(a, b, R);    // ab 在圆外

    if (sgn(R - da) >= 0) return (a ^ pb) / 2 + SectorArea(pb, b, R);   // a 在圆内

    if (sgn(R - db) >= 0) return SectorArea(a, pa, R) + (pa ^ b) / 2;   // b 在圆内

    return SectorArea(a, pa, R) + (pa ^ pb) / 2 + SectorArea(pb, b, R); // ab 是割线

}

// 返回所求的面积交

double Intersection\_Area(Point \*p, int n, Circle C) {

    // 平移

    for (int i = 0; i < n; i++) p[i] = p[i] - C.o;

    C.o = Point(0.0, 0.0);

    double area = 0;

    for (int i = 0, j = n - 1; i < n; j = i++) area += getArea(p[j], p[i], C);

    return fabs(area);

}

// 调用

Point p[N];

int n;

Circle C;

double area = Intersection\_Area(p, n, C);

#### 1.5.6 自适应辛普森积分

// Need: sgn()

// 积分函数, 是啥填啥

// 类似高等数学求定积分

double f(double x) {

    // ...

}

// 辛普森公式

double simpson(double l, double r) {

    return (r - l) \* (f(l) + f(r) + 4 \* f((l + r) / 2)) / 6;

}

// 自适应

double asr(double l, double r, double ans) {

    double mid = (l + r) / 2, a = simpson(l, mid), b = simpson(mid, r);

    if (sgn(a + b - ans) == 0) return ans;

    return asr(l, mid, a) + asr(mid, r, b);

}

// 调用

double ans = asr(l, r, 0);

#### 1.5.7 平面最近点

// Need: dist()

Point p[N], t[N];   // p[] 存点, t[] 是辅助数组

double divide(int l, int r) {

    double d = 2e9;

    if (l == r) return d;

    int mid = l + r >> 1;

    Point tmp = p[mid];

    d = min(divide(l, mid), divide(mid + 1, r));    // 分治

    // 归并排序部分

    int k = 0, i = l, j = mid + 1, tt = 0;

    while (i <= mid && j <= r)

        if (p[i].y < p[j].y) t[k++] = p[i++];

        else t[k++] = p[j++];

    while (i <= mid) t[k++] = p[i++];

    while (j <= r) t[k++] = p[j++];

    for (i = l, j = 0; i <= r; i++, j++) p[i] = t[j];

    for (int i = 0; i < k; i++)

        if (fabs(tmp.x - t[i].x) < d) t[tt++] = t[i];

    for (int i = 0; i < tt; i++)

        for (int j = i + 1; j < tt && t[j].y - t[i].y < d; j++)

            d = min(d, dist(t[i], t[j]));

    return d;

}

// 调用

int n;  // 点的个数

double dis = divide(1, n);

## 二.数据结构模板

### 1.1 线段树

#### 1.1.1 可持久化线段树（求解区间第k大问题）

int root[N],idx;

struct segmentTree{

    segmentTree()

    {

        build(0,0);

    }

    segmentTree(int l,int r)

    {

        build(l,r);

    }

    struct node{

        int l,r;

        long long num,mark;

    }tr[NK];

    int build(int l,int r)

    {

        int now = ++idx;

        tr[now].mark = tr[now].num = 0;

        if(l==r)

        {

            return now;

        }

        int mid = (l + r) / 2;

        tr[now].l=build(l, mid);

        tr[now].r=build(mid + 1, r);

        return now;

    }

    int modify(int p,int l,int r,int l1,int r1,long long val)//表示对区间[l1,r1]中所有点进行+val操作，此时num实际存储的是最大值

    {

        int now = ++idx;

        int mid = (l + r) / 2;

        tr[now] = tr[p];

        tr[now].num += val;

        if(l >= l1 && r <= r1)

        {

            tr[now].mark += val;

            return now;

        }

        if(mid <= r1 && mid >= l1)

        {

            tr[now].l = modify(tr[p].l, l, mid, l1, r1, val);

        }

        if(r >= l1 && r <= r1)

        {

            tr[now].r = modify(tr[p].r, mid + 1, r, l1, r1, val);

        }

        return now;

    }

    int query(int p1, int p2, int l, int r, int val,int mark1,int mark2)//求解[l,r]次的总修改过程中，最大值>=val的最小下标

    {

        if(l==r)

        {

            return l;

        }

        int l1 = tr[p1].l,r1 = tr[p1].r, l2 = tr[p2].l, r2 = tr[p2].r;

        int mid = (l + r) / 2;

        mark1 = mark1 + tr[p1].mark;

        mark2 = mark2 + tr[p2].mark;

        if(tr[l2].num + mark2 - tr[l1].num - mark1 < val)

        {

            return query(r1, r2, mid + 1 , r, val, mark1 , mark2);

        }

        else

        {

            return query(l1, l2, l, mid, val, mark1 , mark2);

        }

    }

};

## 三.数学类

### 3.1 次方和公式

### 3.1.1 一次方

long long get\_1(long long x)

{

    return x\*(x+1)%mod\*inv2%mod;

}

#### 3.1.2 二次方

long long get\_2(long long x)

{

    return x\*(x+1)%mod\*(2\*x+1)%mod\*inv6%mod;

}

#### 3.1.3 三次方

long long get\_3(long long x)

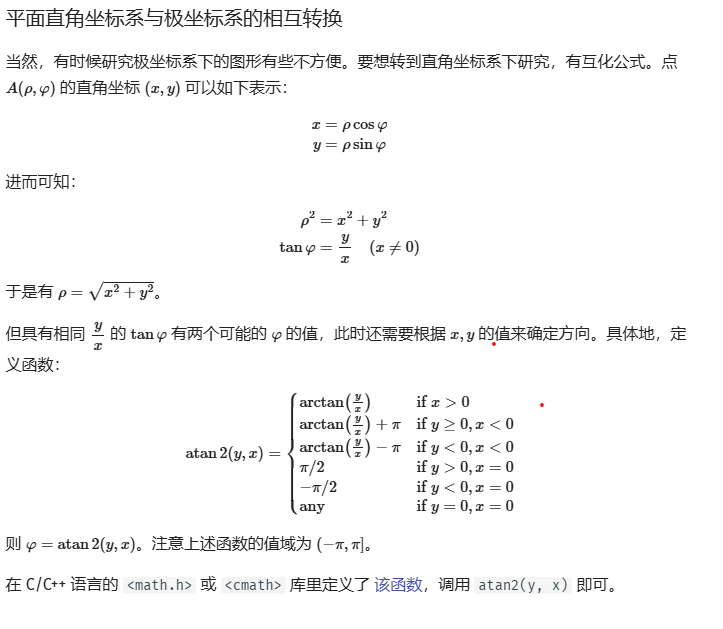
{

    return qpow(x\*(x+1)/2%mod,2);

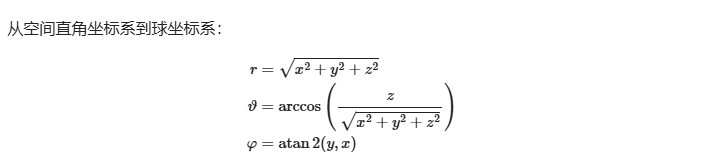
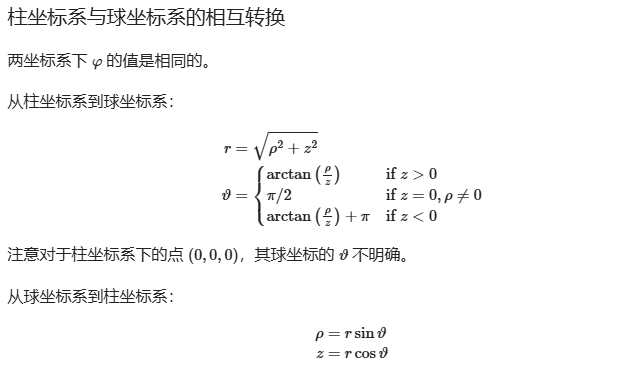
}

### 3.2 坐标转换（未完成）

#### 3.2.1 极坐标



#### 3.2.2 三维空间转换



### 3.3 组合数模板

#### 3.3.1 二维dp求组合数

long long c[N][N];

void init() {

    for(int i=0;i<=MAXN;i++)

    {

        for(int j=0;j<=i;j++)

        {

            if(!j)

            {

                c[i][j]=1;

            }

            else

            {

                c[i][j]=(c[i-1][j]+c[i-1][j-1])%mod;

            }

        }

    }

}

#### 3.3.2 线性求解组合数

void C(int x,int y)

{

    long long now=1;

    for(int i=1;i<=x;i++)

    {

        now=now\*inv[i]%mod\*(y-i+1)%mod;

    }

    return now;

}

## 四.素数相关

### 4.1 约数个数上限



### 4.2 查找因子

#### 4.2.1 欧拉筛

vector<int> pri;

bool not\_prime[N];

void pre(int n)

{

    for(int i=2; i<=n;i++)

    {

        if (!not\_prime[i])

        {

            pri.push\_back(i);

        }

        for (int pri\_j:pri)

        {

            if (i\*pri\_j>n)break;

            not\_prime[i\*pri\_j]=true;

            if (i%pri\_j==0)

            {

                break;

            }

        }

    }

}

#### 4.2.2 （o(n)空间,o(n logn)时间）查找因子

long long minpri[N];

//minpri[x]存储x的最小质因子，利用欧拉筛查找

bool not\_prime[N];

vector<int> pri;

void pre(int n)//n表示你要处理的数字最大范围

{

    for(int i=2; i<=n;i++)

    {

        if (!not\_prime[i])

        {

            pri.push\_back(i);

            minpri[i]=i;

        }

        for (int pri\_j:pri)

        {

            if (i\*pri\_j>n)break;

            not\_prime[i\*pri\_j]=true;

            minpri[i\*pri\_j]=pri\_j;

            if (i%pri\_j==0)

            {

                break;

            }

        }

    }

}

void dfs\_divisor(vector<int>&ct,vector<PII>&f,int now,int p)

{

    if(p==f.size())

    {

        ct.push\_back(now);

        return ;

    }

    int x=1;

    for(int i=0;i<=f[p].second;i++)

    {

        dfs\_divisor(ct,f,now\*x,p+1);

        x\*=f[p].first;

    }

}

void get\_divisor(vector<int> &ct,int now)//ct传入想要得到答案的vector，容器；now表示你想要查找因子的数字

{

    vector<PII> f;

    while(now!=1)

    {

        int cnt=0,x=minpri[now];

        while(now!=1&&now%x==0)

        {

            now=now/minpri[now];

            cnt++;

        }

        f.push\_back({x,cnt});

    }

    dfs\_divisor(ct,f,1,0);

}

### 4.3 扩展欧几里得

long long exgcd(long long a,long long b, long long& x,long long& y) {

  if (b == 0) {

    x = 1;

    y = 0;

    return a;

  }

  long long d = exgcd(b, a % b, x, y);

  long long temp = x;

  x = y;

  y = temp - a / b \* y;

  return d;

}

//通解

//long long gcd=exgcd(a,b,x,y);

//dx=b/gcd    dy=-a/gcd

//x+dx\*k y+dy\*k

## 五.图论

### 5.1 欧拉回路

void dfs(int x)

{

    for(auto k:g[x])

    {

        if(st[x][k]==1)continue;

        st[x][k]=st[k][x]=1;

        dfs(k);

    }

    stk.push(x);

}

### 5.2 tarjan强连通

int dfn[N], low[N], dfncnt, s[N], in\_stack[N], tp;

int scc[N], sc;  // 结点 i 所在 SCC 的编号

int sz[N];       // 强连通 i 的大小

void tarjan(int u) {

  low[u] = dfn[u] = ++dfncnt, s[++tp] = u, in\_stack[u] = 1;

  for (int i = h[u]; i; i = e[i].nex) {

    const int &v = e[i].t;

    if (!dfn[v]) {

      tarjan(v);

      low[u] = min(low[u], low[v]);

    } else if (in\_stack[v]) {

      low[u] = min(low[u], dfn[v]);

    }

  }

  if (dfn[u] == low[u]) {

    ++sc;

    while (s[tp] != u) {

      scc[s[tp]] = sc;

      sz[sc]++;

      in\_stack[s[tp]] = 0;

      --tp;

    }

    scc[s[tp]] = sc;

    sz[sc]++;

    in\_stack[s[tp]] = 0;

    --tp;

  }

}

## 六.字符串

### 6.1 字符串双哈希

long long p1[N],p2[N];

pair<long long,long long>h[N],h1[N];

const int P=1337;

long long n;

string s;

void init() {

    p1[0]=1,h[0]={0,0};

    p2[0]=1;

    for(int i=1;i<=n;i++) {

        p1[i]=p1[i-1]\*P%mod1;

        p2[i]=p2[i-1]\*P%mod2;

        h[i].first=h[i-1].first\*P%mod1+s[i]+7;

        h[i].first%=mod1;

        h[i].second=h[i-1].second\*P%mod2+s[i]+7;

        h[i].second%=mod2;

    }

}

void init1() {

    h1[n+1]={0,0};

    for(int i=n;i>=1;i--) {

        h1[i].first=h1[i+1].first\*P%mod1+s[i]+7;

        h1[i].first%=mod1;

        h1[i].second=h1[i+1].second\*P%mod2+s[i]+7;

        h1[i].second%=mod2;

    }

}

pair<long long,long long> get(int l,int r) {

    return {((h[r].first-h[l-1].first\*p1[r-l+1]%mod1)%mod1+mod1)%mod1,((h[r].second-h[l-1].second\*p2[r-l+1]%mod2)%mod2+mod2)%mod2};

}

pair<long long,long long> get1(int l,int r) {

    return {((h1[l].first-h1[r+1].first\*p1[r-l+1]%mod1)%mod1+mod1)%mod1,((h1[l].second-h1[r+1].second\*p2[r-l+1]%mod2)%mod2+mod2)%mod2};

}