The Grimoire of Programming

Chapter 1

The Trial of Number Theory

```cpp
//number.cpp
#include <cmath>
#include <complex>
namespace number {
    long long inverse (const long long &x, const long long &mod) {
        if (x == 1) return 1;
        return (mod - mod / x) * inverse (mod % x, mod) % mod;
    }
    int fpm (int x, int n, int mod) {
        register int ans = 1, mul = x;
        while (n) {
            if (n & 1) ans = int (1ll * ans * mul % mod);
            mul = int (1ll * mul * mul % mod);
            n >>= 1;
        }
        return ans;
    }
    namespace FFT {
        const int MAXN = 1E6;
        const double PI = acos (-1);
        typedef std::complex <double> Complex;
        Complex e[2][MAXN];
        int prepare (int n) {
            int len = 1;
            for (; len <= 2 * n; len <<= 1);
            for (int i = 0; i < len; i++) {
                e[0][i] = Complex (cos (2 * PI * i / len), sin (2 * PI * i / len));
                e[1][i] = Complex (cos (2 * PI * i / len), -sin (2 * PI * i / len));
            }
            return len;
        }
        void DFT (Complex *a, int n, int f) {
            for (int i = 0, j = 0; i < n; i++) {
                if (i > j) std::swap (a[i], a[j]);
                for (int t = n >> 1; (j ^= t) < t; t >>= 1);
            }
            for (int i = 2; i <= n; i <<= 1)
                for (int j = 0; j < n; j += i)
                    for (int k = 0; k < (i >> 1); k++) {
                        Complex A = a[j + k];
                        Complex B = e[f][n / i * k] * a[j + k + (i >> 1)];
                        a[j + k] = A + B;
                        a[j + k + (i >> 1)] = A - B;
                    }
            if (f == 1) {
                for (int i = 0; i < n; i++)
                    a[i] = Complex (a[i].real () / n, a[i].imag ());
            }
        }
    }
    namespace NTT {
        const int MAXN = 1E6;
        void DFT (int *a, int n, int f, int mod, int prt) {
            for (register int i = 0, j = 0; i < n; i++) {
                if (i > j) std::swap (a[i], a[j]);
                for (register int t = n >> 1; (j ^= t) < t; t >>= 1);
            }
            for (register int i = 2; i <= n; i <<= 1) {
                static int exp[MAXN];
                exp[0] = 1;
                exp[1] = fpm (prt, (mod - 1) / i, mod);
                if (f == 1) exp[1] = fpm (exp[1], mod - 2, mod);
                for (register int k = 2; k < (i >> 1); k++) {
                    exp[k] = int (1ll * exp[k - 1] * exp[1] % mod);
                }
                for (register int j = 0; j < n; j += i) {
                    for (register int k = 0; k < (i >> 1); k++) {
                        register int &pA = a[j + k], &pB = a[j + k + (i >> 1)];
                        register int A = pA, B = int (1ll * pB * exp[k] % mod);
                        pA = (A + B) % mod;
                        pB = (A - B + mod) % mod;
                    }
                }
            }
            if (f == 1) {
                register int rev = fpm (n, mod - 2, mod);
                for (register int i = 0; i < n; i++) {
```

```
                    a[i] = int (1ll * a[i] * rev % mod);
                }
            }
        }
        const int FFT[3] = {1045430273, 1051721729, 1053818881}, PRT[3] = {3, 6, 7};
        int CRT (int *a, int mod) {
            static int inv[3][3];
            for (int i = 0; i < 3; i++)
                for (int j = 0; j < 3; j++)
                    inv[i][j] = (int) inverse (FFT[i], FFT[j]);
            static int x[3];
            for (int i = 0; i < 3; i++) {
                x[i] = a[i];
                for (int j = 0; j < i; j++) {
                    int t = (x[i] - x[j] + FFT[i]) % FFT[i];
                    if (t < 0) t += FFT[i];
                    x[i] = int (1LL * t * inv[j][i] % FFT[i]);
                }
            }
            int sum = 1, ret = x[0] % mod;
            for (int i = 1; i < 3; i ++) {
                sum = int (1LL * sum * FFT[i - 1] % mod);
                ret += int (1LL * x[i] * sum % mod);
                if (ret >= mod) ret -= mod;
            }
            return ret;
        }
    }
}
#include <cstdio>
int main () {
    return 0;
}
```

Chapter 2

The Trial of Geometry

```
/*  Geometry template :
        Most templates of points, lines and circles on a 2D plane.
*/
#include <cmath>
#include <vector>
#include <algorithm>
namespace geometry {
    /*  Basic constant & function :
            EPS : fixes the possible error of data.
                i.e. x == y iff |x - y| < EPS.
            PI : the value of PI.
            int sgn (const double &x) : returns the sign of x.
            int cmp (const double &x, const double &y) : returns the sign of x - y.
            double sqr (const double &x) : returns x * x.
    */
    const double EPS = 1E-8;
    const double PI = acos (-1);
    int sgn (const double &x) { return x < -EPS ? -1 : x > EPS; }
    int cmp (const double &x, const double &y) { return sgn (x - y); }
    double sqr (const double &x) { return x * x; }
    /*  struct point : defines a point and its various utility.
            point (const double &x, const double &y) gives a point at (x, y).
            It also represents a vector on a 2D plane.
            point unit () const : returns the unit vector of (x, y).
            point rot90 () const :
                returns a point rotated 90 degrees counter-clockwise with respect to the origin.
            point _rot () const : same as above except clockwise.
            point rotate (const double &t) const : returns a point rotated t radian(s) counter-clockwise.
            Operators are mostly vector operations. i.e. vector +, -, *, / and dot/det product.
    */
    struct point {
        double x, y;
        point (const double &x = 0, const double &y = 0) : x (x), y (y) {}
        double norm () const { return sqrt (x * x + y * y); }
        double norm2 () const { return x * x + y * y; }
        point unit () const {
            double l = norm ();
            return point (x / l, y / l);
        }
        point rot90 () const {return point (-y, x); }
        point _rot90 () const {return point (y, -x); }
        point rotate (const double &t) const {
            double c = cos (t), s = sin (t);
            return point (x * c - y * s, x * s + y * c);
        }
    };
    bool operator == (const point &a, const point &b) {
        return cmp (a.x, b.x) == 0 && cmp (a.y, b.y) == 0;
    }
    bool operator != (const point &a, const point &b) {
        return ! (a == b);
    }
    bool operator < (const point &a, const point &b) {
        if (cmp (a.x, b.x) == 0) return cmp (a.y, b.y) < 0;
        return cmp (a.x, b.x) < 0;
    }
    point operator - (const point &a) { return point (-a.x, -a.y); }
    point operator + (const point &a, const point &b) {
        return point (a.x + b.x, a.y + b.y);
    }
    point operator - (const point &a, const point &b) {
        return point (a.x - b.x, a.y - b.y);
    }
    point operator * (const point &a, const double &b) {
        return point (a.x * b, a.y * b);
    }
    point operator / (const point &a, const double &b) {
        return point (a.x / b, a.y / b);
    }
    double dot (const point &a, const point &b) {
        return a.x * b.x + a.y * b.y;
    }
    double det (const point &a, const point &b) {
        return a.x * b.y - a.y * b.x;
    }
```

```
double dis (const point &a, const point &b) {
    return sqrt (sqr (a.x - b.x) + sqr (a.y - b.y));
}
/*  struct line : defines a line (segment) based on two points, s and t.
        line (const point &s, const point &t) gives a basic line from s to t.
        double length () const : returns the length of the segment.
*/
struct line {
    point s, t;
    line (const point &s = point (), const point &t = point ()) : s (s), t (t) {}
    double length () const { return dis (s, t); }
};
/*  Point & line interaction :
        bool point_on_line (const point &a, const line &b) : checks if a is on b.
        bool intersect_judgement (const line &a, const line &b) : checks if segment a and b intersect.
        point line_intersect (const line &a, const line &b) : returns the intersection of a and b.
            Fails on colinear or parallel situations.
        double point_to_line (const point &a, const line &b) : returns the distance from a to b.
        double point_to_segment (const point &a, const lint &b) : returns the distance from a to b.
            i.e. the minimized length from a to segment b.
        bool in_polygon (const point &p, const std::vector <point> &po) :
            checks if a is in a polygon with vetices po (clockwise or counter-clockwise order).
        double polygon_area (const std::vector <point> &a) :
            returns the signed area of polygon a (positive for counter-clockwise order, and vise-versa).
        point project_to_line (const point &a, const line &b) :
            returns the projection of a on b,
*/
bool point_on_line (const point &a, const line &b) {
    return sgn (det (a - b.s, b.t - b.s)) == 0 && sgn (dot (b.s - a, b.t - a)) <= 0;
}
bool two_side (const point &a, const point &b, const line &c) {
    return sgn (det (a - c.s, c.t - c.s)) * sgn (det (b - c.s, c.t - c.s)) < 0;
}
bool intersect_judgement (const line &a, const line &b) {
    if (point_on_line (b.s, a) || point_on_line (b.t, a)) return true;
    if (point_on_line (a.s, b) || point_on_line (a.t, b)) return true;
    return two_side (a.s, a.t, b) && two_side (b.s, b.t, a);
}
point line_intersect (const line &a, const line &b) {
    double s1 = det (a.t - a.s, b.s - a.s);
    double s2 = det (a.t - a.s, b.t - a.s);
    return (b.s * s2 - b.t * s1) / (s2 - s1);
}
double point_to_line (const point &a, const line &b) {
    return fabs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t);
}
double point_to_segment (const point &a, const line &b) {
    if (sgn (dot (b.s - a, b.t - b.s) * dot (b.t - a, b.t - b.s)) <= 0)
        return fabs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t);
    return std::min (dis (a, b.s), dis (a, b.t));
}
bool in_polygon (const point &p, const std::vector <point> & po) {
    int n = (int) po.size ();
    int counter = 0;
    for (int i = 0; i < n; ++i) {
        point a = po[i], b = po[ (i + 1) % n];
        /*          The following statement checks is p is on the border of the polygon.
                    The boolean returned may be changed if necessary.
                        i.e. the algorithm may check if p is strictly in the polygon.
        */
        if (point_on_line (p, line (a, b))) return true;
        int x = sgn (det (p - a, b - a)), y = sgn (a.y - p.y), z = sgn (b.y - p.y);
        if (x > 0 && y <= 0 && z > 0) counter++;
        if (x < 0 && z <= 0 && y > 0) counter--;
    }
    return counter != 0;
}
double polygon_area (const std::vector <point> &a) {
    double ans = 0.0;
    for (int i = 0; i < (int) a.size (); ++i)
        ans += det (a[i], a[ (i + 1) % a.size ()]) / 2.0;
    return ans;
}
point project_to_line (const point &a, const line &b) {
    return b.s + (b.t - b.s) * (dot (a - b.s, b.t - b.s) / (b.t - b.s).norm2 ());
}
/*  Centers of a triangle :
        returns various centers of a triangle with vertices (a, b, c).
*/
point incenter (const point &a, const point &b, const point &c) {
    double p = (a - b).norm () + (b - c).norm () + (c - a).norm ();
    return (a * (b - c).norm () + b * (c - a).norm () + c * (a - b).norm ()) / p;
}
point circumcenter (const point &a, const point &b, const point &c) {
    point p = b - a, q = c - a, s (dot (p, p) / 2, dot (q, q) / 2);
    double d = det (p, q);
    return a + point (det (s, point (p.y, q.y)), det (point (p.x, q.x), s)) / d;
}
point orthocenter (const point &a, const point &b, const point &c) {
    return a + b + c - circumcenter (a, b, c) * 2.0;
}
/*  struct circle defines a circle.
    circle (point c, double r) gives a circle with center c and radius r.
*/
struct circle {
    point c;
    double r;
    circle (point c = point (), double r = 0) : c (c), r (r) {}
};
```

```
bool operator == (const circle &a, const circle &b) {
    return a.c == b.c && cmp (a.r, b.r) == 0;
}
bool operator != (const circle &a, const circle &b) {
    return ! (a == b);
}
/*  Circle interaction :
        bool in_circle (const point &a, const circle &b) : checks if a is in or on b.
        circle make_circle (const point &a, const point &b) :
            generates a circle with diameter ab.
        circle make_circle (const point &a, const point &b, const point &c) :
            generates a circle passing a, b and c.
        std::pair <point, point> line_circle_intersect (const line &a, const circle &b) :
            returns the intersections of a and b.
            Fails if a and b do not intersect.
        std::pair <point, point> circle_intersect (const circle &a, const circle &b):
            returns the intersections of a and b.
            Fails if a and b do not intersect.
        std::pair <line, line> tangent (const point &a, const circle &b) :
            returns the tangent lines of b passing through a.
            Fails if a is in b.
*/
bool in_circle (const point &a, const circle &b) {
    return cmp (dis (a, b.c), b.r) <= 0;
}
circle make_circle (const point &a, const point &b) {
    return circle ((a + b) / 2, dis (a, b) / 2);
}
circle make_circle (const point &a, const point &b, const point &c) {
    point p = circumcenter (a, b, c);
    return circle (p, dis (p, a));
}
std::pair <point, point> line_circle_intersect (const line &a, const circle &b) {
    double x = sqrt (sqr (b.r) - sqr (point_to_line (b.c, a)));
    return std::make_pair (project_to_line (b.c, a) + (a.s - a.t).unit () * x,
                           project_to_line (b.c, a) - (a.s - a.t).unit () * x);
}
point __circle_intersect (const circle &a, const circle &b) {
    point r = (b.c - a.c).unit ();
    double d = dis (a.c, b.c);
    double x = .5 * ((sqr (a.r) - sqr (b.r)) / d + d);
    double h = sqrt (sqr (a.r) - sqr (x));
    return a.c + r * x + r.rot90 () * h;
}
std::pair <point, point> circle_intersect (const circle &a, const circle &b) {
    return std::make_pair (__circle_intersect (a, b), __circle_intersect (b, a));
}
std::pair <line, line> tangent (const point &a, const circle &b) {
    circle p = make_circle (a, b.c);
    return circle_intersect (p, b);
}
/*  Convex hull :
        std::vector <point> convex_hull (std::vector <point> a) :
            returns the convex hull of point set a (counter-clockwise).
*/
bool turn_left (const point &a, const point &b, const point &c) {
    return sgn (det (b - a, c - a)) >= 0;
}
bool turn_right (const point &a, const point &b, const point &c) {
    return sgn (det (b - a, c - a)) <= 0;
}
std::vector <point> convex_hull (std::vector <point> a) {
    int n = (int) a.size (), cnt = 0;
    std::sort (a.begin (), a.end ());
    std::vector <point> ret;
    for (int i = 0; i < n; ++i) {
        while (cnt > 1 && turn_left (ret[cnt - 2], a[i], ret[cnt - 1])) {
            --cnt;
            ret.pop_back ();
        }
        ret.push_back (a[i]);
        ++cnt;
    }
    int fixed = cnt;
    for (int i = n - 1; i >= 0; --i) {
        while (cnt > fixed && turn_left (ret[cnt - 2], a[i], ret[cnt - 1])) {
            --cnt;
            ret.pop_back ();
        }
        ret.push_back (a[i]);
        ++cnt;
    }
    ret.pop_back ();
    return ret;
}
/*  Minimum circle of a point set :
    circle minimum_circle (std::vector <point> p) : returns the minimum circle of point set p.
*/
circle minimum_circle (std::vector <point> p) {
    circle ret;
    std::random_shuffle (p.begin (), p.end ());
    for (int i = 0; i < (int) p.size (); ++i)
        if (!in_circle (p[i], ret)) {
            ret = circle (p[i], 0);
            for (int j = 0; j < i; ++j)
                if (!in_circle (p[j], ret)) {
                    ret = make_circle (p[j], p[i]);
                    for (int k = 0; k < j; ++k)
                        if (!in_circle (p[k], ret)) ret = make_circle (p[i], p[j], p[k]);
                }
        }
```

```
    return ret;
}
/*  Online half plane intersection (complexity = O(c.size ())) :
        std::vector <point> cut (const std::vector<point> &c, line p) :
            returns the convex polygon cutting convex polygon c with half plane p.
                (left hand with respect to vector p)
            If such polygon does not exist, returns an empty set.
            e.g.
                static const double BOUND = 1e5;
                convex.clear ();
                convex.push_back (point (-BOUND, -BOUND));
                convex.push_back (point (BOUND, -BOUND));
                convex.push_back (point (BOUND, BOUND));
                convex.push_back (point (-BOUND, BOUND));
                convex = cut (convex, line(point, point));
                if (convex.empty ()) { ... }
*/
std::vector <point> cut (const std::vector<point> &c, line p) {
    std::vector <point> ret;
    if (c.empty ()) return ret;
    for (int i = 0; i < (int) c.size (); ++i) {
        int j = (i + 1) % (int) c.size ();
        if (!turn_right (p.s, p.t, c[i])) ret.push_back (c[i]);
        if (two_side (c[i], c[j], p))
            ret.push_back (line_intersect (p, line (c[i], c[j])));
    }
    return ret;
}
/*  Offline half plane intersection (complexity = O(nlogn), n = h.size ()) :
        std::vector <point> half_plane_intersect (std::vector <line> h) :
            returns the intersection of half planes h.
                (left hand with respect to the vector)
            If such polygon does not exist, returns an empty set.
*/
bool turn_left (const line &l, const point &p) {
    return turn_left (l.s, l.t, p);
}
std::vector <point> half_plane_intersect (std::vector <line> h) {
    typedef std::pair <double, line> polar;
    std::vector <polar> g;
    g.resize (h.size ());
    for (int i = 0; i < (int) h.size (); ++i) {
        point v = h[i].t - h[i].s;
        g[i] = std::make_pair (atan2 (v.y, v.x), h[i]);
    }
    sort (g.begin (), g.end (), [] (const polar &a, const polar &b) {
        if (cmp (a.first, b.first) == 0)
            return sgn (det (a.second.t - a.second.s, b.second.t - a.second.s)) < 0;
        else
            return cmp (a.first, b.first) < 0;
    });
    h.resize (std::unique (g.begin (), g.end (), [] (const polar &a, const polar &b) {
        return cmp (a.first, b.first) == 0;
    }) - g.begin ());
    for (int i = 0; i < (int) h.size (); ++i)
        h[i] = g[i].second;
    int fore = 0, rear = -1;
    std::vector <line> ret;
    for (int i = 0; i < (int) h.size (); ++i) {
        while (fore < rear && !turn_left (h[i], line_intersect (ret[rear - 1], ret[rear]))) {
            --rear;
            ret.pop_back ();
        }
        while (fore < rear && !turn_left (h[i], line_intersect (ret[fore], ret[fore + 1])))
            ++fore;
        ++rear;
        ret.push_back (h[i]);
    }
    while (rear - fore > 1 && !turn_left (ret[fore], line_intersect (ret[rear - 1], ret[rear]))) {
        --rear;
        ret.pop_back ();
    }
    while (rear - fore > 1 && !turn_left (ret[rear], line_intersect (ret[fore], ret[fore + 1])))
        ++fore;
    if (rear - fore < 2) return std::vector <point> ();
    std::vector <point> ans;
    ans.resize (ret.size ());
    for (int i = 0; i < (int) ret.size (); ++i)
        ans[i] = line_intersect (ret[i], ret[ (i + 1) % ret.size ()]);
    return ans;
}
/*  Intersection of a polygon and a circle :
        double polygon_circle_intersect::main (const std::vector <point> &p, const circle &c) :
            returns the area of intersection of polygon p (vertices in either order) and c.
*/
namespace polygon_circle_intersect {
    //  The area of the sector with center (0, 0), radius r and segment ab.
    double sector_area (const point &a, const point &b, const double &r) {
        double c = (2.0 * r * r - (a - b).norm2 ()) / (2.0 * r * r);
        double al = acos (c);
        return r * r * al / 2.0;
    }
    //  The area of triangle (a, b, (0, 0)) intersecting circle (point (), r).
    double area (const point &a, const point &b, const double &r) {
        double dA = dot (a, a), dB = dot (b, b), dC = point_to_segment (point (), line (a, b)), ans = 0.0;
        if (sgn (dA - r * r) <= 0 && sgn (dB - r * r) <= 0) return det (a, b) / 2.0;
        point tA = a.unit () * r;
        point tB = b.unit () * r;
        if (sgn (dC - r) > 0) return sector_area (tA, tB, r);
        std::pair <point, point> ret = line_circle_intersect (line (a, b), circle (point (), r));
        if (sgn (dA - r * r) > 0 && sgn (dB - r * r) > 0) {
            ans += sector_area (tA, ret.first, r);
```

```
            ans += det (ret.first, ret.second) / 2.0;
            ans += sector_area (ret.second, tB, r);
            return ans;
        }
        if (sgn (dA - r * r) > 0)
            return det (ret.first, b) / 2.0 + sector_area (tA, ret.first, r);
        else
            return det (a, ret.second) / 2.0 + sector_area (ret.second, tB, r);
    }
    //  Main process.
    double main (const std::vector <point> &p, const circle &c) {
        double ret = 0.0;
        for (int i = 0; i < (int) p.size (); ++i) {
            int s = sgn (det (p[i] - c.c, p[ (i + 1) % p.size ()] - c.c));
            if (s > 0)
                ret += area (p[i] - c.c, p[ (i + 1) % p.size ()] - c.c, c.r);
            else
                ret -= area (p[ (i + 1) % p.size ()] - c.c, p[i] - c.c, c.r);
        }
        return fabs (ret);
    }
}
/*  Union of circles :
        std::vector <double> union_circle::main (const std::vector <circle> &c) :
            returns the union of circle set c.
            The i-th element is the area covered with at least i circles.
*/
namespace union_circle {
    struct cp {
        double x, y, angle;
        int d;
        double r;
        cp (const double &x = 0, const double &y = 0, const double &angle = 0,
            int d = 0, const double &r = 0) : x (x), y (y), angle (angle), d (d), r (r) {}
    };
    double dis (const cp &a, const cp &b) {
        return sqrt (sqr (a.x - b.x) + sqr (a.y - b.y));
    }
    double cross (const cp &p0, const cp &p1, const cp &p2) {
        return (p1.x - p0.x) * (p2.y - p0.y) - (p1.y - p0.y) * (p2.x - p0.x);
    }
    int cir_cross (cp p1, double r1, cp p2, double r2, cp &cp1, cp &cp2) {
        double mx = p2.x - p1.x, sx = p2.x + p1.x, mx2 = mx * mx;
        double my = p2.y - p1.y, sy = p2.y + p1.y, my2 = my * my;
        double sq = mx2 + my2, d = - (sq - sqr (r1 - r2)) * (sq - sqr (r1 + r2));
        if (sgn (d) < 0) return 0;
        if (sgn (d) <= 0) d = 0;
        else d = sqrt (d);
        double x = mx * ((r1 + r2) * (r1 - r2) + mx * sx) + sx * my2;
        double y = my * ((r1 + r2) * (r1 - r2) + my * sy) + sy * mx2;
        double dx = mx * d, dy = my * d;
        sq *= 2;
        cp1.x = (x - dy) / sq;
        cp1.y = (y + dx) / sq;
        cp2.x = (x + dy) / sq;
        cp2.y = (y - dx) / sq;
        if (sgn (d) > 0) return 2;
        else return 1;
    }
    bool circmp (const cp &u, const cp &v) {
        return sgn (u.r - v.r) < 0;
    }
    bool cmp (const cp &u, const cp &v) {
        if (sgn (u.angle - v.angle)) return u.angle < v.angle;
        return u.d > v.d;
    }
    double calc (cp cir, cp cp1, cp cp2) {
        double ans = (cp2.angle - cp1.angle) * sqr (cir.r)
                     - cross (cir, cp1, cp2) + cross (cp (0, 0), cp1, cp2);
        return ans / 2;
    }
    std::vector <double> main (const std::vector <circle> &c) {
        int n = c.size ();
        std::vector <cp> cir, tp;
        std::vector <double> area;
        cir.resize (n);
        tp.resize (2 * n);
        area.resize (n + 1);
        for (int i = 0; i < n; i++)
            cir[i] = cp (c[i].c.x, c[i].c.y, 0, 1, c[i].r);
        cp cp1, cp2;
        std::sort (cir.begin (), cir.end (), circmp);
        for (int i = 0; i < n; ++i)
            for (int j = i + 1; j < n; ++j)
                if (sgn (dis (cir[i], cir[j]) + cir[i].r - cir[j].r) <= 0)
                    cir[i].d++;
        for (int i = 0; i < n; ++i) {
            int tn = 0, cnt = 0;
            for (int j = 0; j < n; ++j) {
                if (i == j) continue;
                if (cir_cross (cir[i], cir[i].r, cir[j], cir[j].r, cp2, cp1) < 2) continue;
                cp1.angle = atan2 (cp1.y - cir[i].y, cp1.x - cir[i].x);
                cp2.angle = atan2 (cp2.y - cir[i].y, cp2.x - cir[i].x);
                cp1.d = 1;
                tp[tn++] = cp1;
                cp2.d = -1;
                tp[tn++] = cp2;
                if (sgn (cp1.angle - cp2.angle) > 0) cnt++;
            }
            tp[tn++] = cp (cir[i].x - cir[i].r, cir[i].y, PI, -cnt);
            tp[tn++] = cp (cir[i].x - cir[i].r, cir[i].y, -PI, cnt);
            std::sort (tp.begin (), tp.begin () + tn, cmp);
```

```
                int p, s = cir[i].d + tp[0].d;
                for (int j = 1; j < tn; ++j) {
                    p = s;
                    s += tp[j].d;
                    area[p] += calc (cir[i], tp[j - 1], tp[j]);
                }
            }
            return area;
        }
    }
}
#include <cstdio>
using namespace geometry;
int main() {
    return 0;
}
```

Chapter 3

The Trial of Graph

```
namespace graph {
    const int MAXN = 1E5, MAXM = 1E5;
    struct edge_list {
        int size;
        int begin[MAXN], dest[MAXM], next[MAXM];
        void clear (int n) {
            size = 0;
            for (int i = 0; i < n; i++)
                begin[i] = -1;
        }
        edge_list (int n = MAXN) {
            clear (n);
        }
        void add_edge (int u, int v) {
            dest[size] = v; next[size] = begin[u]; begin[u] = size++;
        }
    };
}
#include <cstdio>
int main () {
    return 0;
}
```

Chapter 4

The Trial of String

```cpp
//string.cpp
#include <string>
#include <vector>
#include <map>
namespace string {
    struct suffix_automaton {
        std::vector <std::map <char, int> > edges; // edges[i]  : the labeled edges from node i
        std::vector <int> link;             // link[i]   : the parent of i
        std::vector <int> length;           // length[i] : the length of the longest string in the ith class
        std::vector <int> terminals;        // terminals : the terminal state of the automaton
        int last;                       // the index of the equivalence class of the whole string
        suffix_automaton (std::string s) {
            edges.push_back (std::map <char,int> ());
            link.push_back (-1);
            length.push_back (0);
            last = 0;
            for (int i = 0; i < (int) s.size(); ++i) {
                edges.push_back (std::map <char,int> ());
                length.push_back (i + 1);
                link.push_back (0);
                int r = edges.size() - 1;
                int p = last;
                while (p >= 0 && edges[p].find (s[i]) == edges[p].end ()) {
                    edges[p][s[i]] = r;
                    p = link[p];
                }
                if (p != -1) {
                    int q = edges[p][s[i]];
                    if (length[p] + 1 == length[q]) {
                        link[r] = q;
                    } else {
                        edges.push_back (edges[q]);
                        length.push_back (length[p] + 1);
                        link.push_back (link[q]);
                        int qq = edges.size() - 1;
                        link[q] = qq;
                        link[r] = qq;
                        while (p >= 0 && edges[p][s[i]] == q) {
                            edges[p][s[i]] = qq;
                            p = link[p];
                        }
                    }
                }
                last = r;
            }
            int p = last;
            while (p > 0) {
                terminals.push_back (p);
                p = link[p];
            }
        }
    };
}
#include <cstdio>
int main () {
    return 0;
}
```

Chapter 5

Reference

5.1 vimrc

```
set ruler
set number
set tabstop=4
set softtabstop=4
set shiftwidth=4
set smartindent
set showmatch
set hlsearch
set incsearch
set autoread
set backspace=2
set mouse=a
syntax on
nmap <C-A> ggVG
vmap <C-C> "+y
nmap <C-P> "+p
autocmd FileType cpp set cindent
autocmd FileType cpp map <F3> :vsplit %<.in <CR>
autocmd FileType cpp map <F5> :!time ./%<.exe <CR>
autocmd FileType cpp map <F7> :!gdb ./%<.exe <CR>
autocmd FileType cpp map <F8> :!time ./%<.exe < %<.in <CR>
autocmd FileType cpp map <F9> :!g++ % -o %< -g -std=c++11 -Wall -Wextra -Wconversion && size %<.exe <CR>
autocmd FileType cpp map <F10> :!astyle -A2 -t -C -S -N -O -xd % <CR>
```