University of Alberta Code Archive 2012

Geometry – 2D Primitives

Basics

```
typedef complex<double> point;
struct circle {
    point c; double r;
    circle(point c, double r):c(c),r(r){}
    circle(){}
};
double cross(const point &a, const point &b) {
    return imag(conj(a)*b);
}
double dot(const point &a, const point &b) {
    return real(conj(a)*b);
}
```

Area of intersection of two circles

```
double circ_inter_area(circle &a, circle &b) {
    double d = abs(b.c-a.c);
    if (d <= (b.r - a.r)) return a.r*a.r*M_PI;
    if (d <= (a.r - b.r)) return b.r*b.r*M_PI;
    if (d >= a.r + b.r) return 0;
    double alpha = acos((a.r*a.r+d*d-b.r*b.r)/(2*a.r*d));
    double beta  = acos((b.r*b.r+d*d-a.r*a.r)/(2*b.r*d));
    return a.r*a.r*(alpha-0.5*sin(2*alpha))+b.r*b.r*(beta-
0.5*sin(2*beta));
}
```

Points of intersection of two circles

```
// Intersects two circles and intersection points are in 'inter'
// -1-> outside, 0-> inside, 1-> tangent, 2-> 2 intersections
int circ_circ_inter(circle &a, circle &b, vector<point> &inter)
{
    double d2 = norm(b.c-a.c), rS = a.r+b.r, rD = a.r-b.r;
    if (d2 > rS*rS) return -1;
    if (d2 < rD*rD) return 0;
    double ca = 0.5*(1 + rS*rD/d2);
    point z = point(ca, sqrt((a.r*a.r/d2)-ca*ca));
    inter.push_back(a.c + (b.c-a.c)*z);
    if(abs(z.imag())>eps)
        inter.push_back(a.c + (b.c-a.c)*conj(z));
    return inter.size();
}
```

Line-circle intersection

```
// Intersects (infinite) line a-b with circle c
// Intersection points are in 'inter'
// 0 -> no intersection, 1 -> tangent, 2 -> two intersections
int line_circ_inter(point a, point b, circle c, vector<point>
&inter) {
    c.c -= a; b -= a;
    point m = b*real(c.c/b);
    double d2 = norm(m-c.c);
    if (d2 > c.r*c.r) return 0;
    double l = sqrt((c.r*c.r-d2)/norm(b));
    inter.push_back(a + m + l*b);
    if(abs(l)>eps)
        inter.push_back(a + m - l*b);
    return inter.size();
}
```

Line-line intersection

```
// Intersects point of lines a-b and c-d
// -1->coincide,0->parallel,1->intersected(inter. point in 'p')
int line_line_inter(point a, point b, point c, point d, point
&p) {
    if(abs(cross(b-a,d-c))>eps) {
        p = (cross((c-a),d-c)/cross(b-a,d-c))*(b-a)+a;
        return 1;
    }
    if(abs(cross(b-a,b-c))>eps)
        return 0;
    return -1;
}
```

Segment-segment intersection

```
// Intersect of segments a-b and c-d
//   -2  -> not parallel and no intersection
//   -1  -> coincide with no common point
//    0  -> parallel and not coincide
//    1  -> intersected ('p' is intersection of segments)
//    2  -> coincide with common points ('p' is one of the end
//          points lying on both segments)
int seg_seg_inter(point a, point b, point c, point d, point &p)
{
    int s = line_line_inter(a,b,c,d,p);
    if(s==0)
        return 0;
    if(s==-1) {
        // '<-eps' excludes endpoints in the coincide case
        if(dot(a-c,a-d)<eps) {
            p = a;
            return 2;
        }
        if(dot(b-c,b-d)<eps) {
            p=b;
            return 2;
        }
        if(dot(c-a,c-b)<eps) {
            p=c;
            return 2;
        }
        return -1;
    }
    // '<-eps' excludes endpoints in intersected case
    if(dot(p-a,p-b)<eps && dot(p-c,p-d)<eps)
        return 1;
    return -2;
}
```

Parabola-line intersection

```
// Find intersection of the line d-e and the parabola that
// is defined by point 'p' and line a-b
// Returns the number of intersections
// 'ans' has intersection points
int parabola_line_inter(point p, point a, point b, point d,
point e, vector<point> &ans) {
    b = b-a;
    p/=b; a/=b; d/=b; e/=b;
    a-=p; d-=p; e-=p;
    point n = (e-d)*point(0,1);
    double c = -dot(n,e);
    if(abs(n.imag())<eps) {
        if(abs(a.imag())>eps) {
            double x = -c/n.real();
            ans.push_back(point(x,a.imag()/2-x*x/(2*a.imag())));
        }
    } else {
        double aa = 1;
        double bb = -2*a.imag()*n.real()/n.imag();
        double cc = -2*a.imag()*c/n.imag()-a.imag()*a.imag();
        double delta = bb*bb-4*aa*cc;
        if(delta>-eps) {
            if(delta<0)
                delta = 0;
            delta = sqrt(delta);
            double x = (-bb+delta)/(2*aa);
            ans.push_back(point(x,(-c-n.real()*x)/n.imag()));
            if(delta>eps) {
                double x = (-bb-delta)/(2*aa);
                ans.push_back(point(x,(-c-
n.real()*x)/n.imag()));
            }
        }
    }
    for(int i=0;i<ans.size();i++)
        ans[i]=(ans[i]+p)*b;
    return ans.size();
}
```

Circle described by three points

```
// Returns whether they form a circle or not.
// 'center' and 'r' contain the circle if there is one
bool get_circle(point p1, point p2, point p3, point &center,
double &r) {
    double g = 2*imag(conj(p2-p1)*(p3-p2));
    if (abs(g) < eps) return false;
    center = p1*(norm(p3)-norm(p2));
    center += p2*(norm(p1)-norm(p3));
    center += p3*(norm(p2)-norm(p1));
    center /= point(0, g); r = abs(p1-center);
    return true;
}
```

Circle described by three lines

```
// Returns number of circles that are tangent to all three lines
// 'cirs' has all possible circles with radius > 0
// It has zero circles when two of them are coincide
// It has two circles when only two of them are parallel
// It has four circles when they form a triangle. In this case
// first circle is incircle. Next circles are ex-circles tangent
// to edge a,b,c of triangle respectively.
int get_circle(point a1, point a2, point b1, point b2, point c1,
point c2, vector<circle> &cirs) {
    point a,b,c;
    int sa=line_line_inter(a1,a2,b1,b2,c);
    int sb=line_line_inter(b1,b2,c1,c2,a);
    int sc=line_line_inter(c1,c2,a1,a2,b);
    if(sa==-1 || sb==-1 || sc==-1)
        return 0;
    if(sa+sb+sc==0)
        return 0;
    if(sb==0) {
        swap(a1,c1);
        swap(a2,c2);
    }
    if(sc==0) {
        swap(b1,c1);
        swap(b2,c2);
    }
    sa=line_line_inter(a1,a2,b1,b2,c);
    line_line_inter(b1,b2,c1,c2,a);
    line_line_inter(c1,c2,a1,a2,b);
    if(sa==0) {
        point v1 = polar(1.0,(arg(a2-a1)+arg(a-b))/2)+b;
        point v2 = polar(1.0,(arg(a1-a2)+arg(a-b))/2)+b;
        point v3 = polar(1.0,(arg(b2-b1)+arg(a-b))/2)+a;
        point v4 = polar(1.0,(arg(b1-b2)+arg(a-b))/2)+a;
        point p;
        if(line_line_inter(b,v1,a,v3,p)==0)
            swap(v3,v4);
        line_line_inter(b,v1,a,v3,p);
        circle c1,c2;
        c1.c = p;
        line_line_inter(b,v2,a,v4,p);
        c2.c = p;
        c1.r = c2.r = abs(((a1-b1)/(b2-b1)).imag()*abs(b2-
b1))/2;
        cirs.push_back(c1);
        cirs.push_back(c2);
    } else {
        if(abs(a-b)<eps)
            return 0;
        point bisec1[4][2];
        point bisec2[4][2];
        bisec1[0][0]=polar(1.0,(arg(c-a)+arg(b-a))/2);
        bisec1[0][1]=a;
        bisec2[0][0]=polar(1.0,(arg(c-b)+arg(a-b))/2);
        bisec2[0][1]=b;

        bisec1[1][0]=polar(1.0,(arg(c-a)+arg(b-a))/2);
        bisec1[1][1]=a;
        bisec2[1][0]=polar(1.0,(arg(c-b)+arg(b-a))/2);
        bisec2[1][1]=b;

        bisec1[2][0]=polar(1.0,(arg(a-b)+arg(c-b))/2);
        bisec1[2][1]=b;
        bisec2[2][0]=polar(1.0,(arg(a-c)+arg(c-b))/2);
        bisec2[2][1]=c;

        bisec1[3][0]=polar(1.0,(arg(b-c)+arg(a-c))/2);
        bisec1[3][1]=b;
        bisec2[3][0]=polar(1.0,(arg(b-a)+arg(a-c))/2);
        bisec2[3][1]=c;
        for(int i=0;i<4;i++) {
            point p;
            line_line_inter(bisec1[i][1],bisec1[i][1]+bisec1[i]
[0],bisec2[i][1],bisec2[i][1]+bisec2[i][0],p);
            circle c1;
            c1.c = p;
            c1.r = abs(((p-a)/(b-a)).imag())*abs(b-a);
            cirs.push_back(c1);
        }
    }
    return cirs.size();
}
```

Circle described by two points and one line

```
// Returns number of circles that pass through point a and b and
// are tangent to the line c-d
// 'ans' has all possible circles with radius > 0
int get_circle(point a, point b, point c, point d,
vector<circle> &ans) {
    point pa = (a+b)/2.0;
    point pb = (b-a)*point(0,1)+pa;
    vector<point> ta;
    parabola_line_inter(a,c,d,pa,pb,ta);
    for(int i=0;i<ta.size();i++)
        ans.push_back(circle(ta[i],abs(a-ta[i])));
    return ans.size();
}
```

Circle described by two lines and one point

```
// Returns number of circles that pass through point p and are
// tangent to the lines a-b and c-d
// 'ans' has all possible circles with radius greater than zero
int get_circle(point p, point a, point b, point c, point d,
vector<circle> &ans) {
    point inter;
    int st = line_line_inter(a,b,c,d,inter);
    if(st==-1) return 0;
    d-=c;
    b-=a;
    vector<point> ta;
    if(st==0) {
        point pa = point(0,imag((a-c)/d)/2)*d+c;
        point pb = b+pa;
        parabola_line_inter(p,a,a+b,pa,pb,ta);
    } else {
        if(abs(inter-p)>eps) {
            point bi;
            bi = polar(1.0,(arg(b)+arg(d))/2)+inter;
            vector<point> temp;
            parabola_line_inter(p,a,a+b,inter,bi,temp);
            ta.insert(ta.end(),temp.begin(),temp.end());
            temp.clear();
            bi = polar(1.0,(arg(b)+arg(d)+M_PI)/2)+inter;
            parabola_line_inter(p,a,a+b,inter,bi,temp);
            ta.insert(ta.end(),temp.begin(),temp.end());
        }
    }
    for(int i=0;i<ta.size();i++)
        ans.push_back(circle(ta[i],abs(p-ta[i])));
    return ans.size();
}
```

Geometry – 2D Misc

Heron's formula for triangle area

```
// Given side lengths a, b, c, returns area or -1 if triangle is
// impossible
double area_heron(double a, double b, double c) {
    if (a < b) swap(a, b);
    if (a < c) swap(a, c);
    if (b < c) swap(b, c);
    if (a > b+c) return -1;
    return sqrt((a+(b+c))*(c-(a-b))*(c+(a-b))*(a+(b-c))/16.0);
}
```

Rectangle in rectangle test

```
// Can rectangle with dims x*y fit inside box with dims w*h?
// Returns true for a "tight fit", if false is desired then swap
// strictness of inequalities.
bool rect_in_rect(double x, double y, double w, double h) {
    if (x > y) swap(x, y);
    if (w > h) swap(w, h);
    if (w < x) return false;
    if (y <= h) return true;
    double a = y*y - x*x;
    double b = x*h - y*w;
    double c = x*w - y*h;
    return a*a <= b*b + c*c;
}
```

Centroid and area of a simple polygon [O(N)]

```
// Points must be oriented (CW or CCW), and non-convex is OK
// Returns (nan,nan) is area of polygon is zero
point centroid(vector<point> p) {
    int n = p.size(); // should be at least 1
    double area = 0; point c(0,0);
    for(int i = n-1, j = 0; j < n; i = j++) {
        double a = (conj(p[i])*p[j]).imag()/2; //cross
        area += a;
        c += (p[i]+p[j])*(a/3);
    }
    c /= area;
    return c; // or return 'area' for the area of polygon
}
```

Point in polygon [O(N)]

```
// outside -> 0, inside -> 1, on the border -> 2
int pt_in_poly(const vector<point> &p, const point &a) {
    int n = p.size(); int inside = false;
    for (int i=0, j=n-1; i<n; j=i++) {
        if (abs(cross(a-p[i],a-p[j]))<eps && dot(a-p[i],a-
p[j])<eps)
            return 2;
        if (((imag(p[i])<=imag(a)) && (imag(a)<imag(p[j]))) ||
((imag(p[j])<=imag(a))) && (imag(a)<imag(p[i])))
            if (real(a)-real(p[i]) < (real(p[j])-
real(p[i]))*(imag(a)-imag(p[i])) / (imag(p[j])-imag(p[i])))
                inside = !inside;
    }
    return inside;
}
```

Convex-hull [O(N log N)]

```
// Assumes pts.size()>0 and returns ccw convex hull with no
// 3 collinear points and with duplicated left most side node
int comp(const point &a,const point &b) {
    if(abs(a.real()-b.real())>eps)
        return a.real()<b.real();
    if(abs(a.imag()-b.imag())>eps)
        return a.imag()<b.imag();
    return 0;
}
inline vector<point> convexhull (vector<point> &pts) {
    sort(pts.begin(),pts.end(),comp);
    vector<point> lower, upper;
    for(int i=0; i<(int)pts.size(); i++) {
        // <-eps include all points on border
        while (lower.size() >= 2 && cross(lower.back()-
lower[lower.size()-2], pts[i]-lower.back()) < eps)
            lower.pop_back();
        // >eps include all points on border
        while (upper.size() >= 2 && cross(upper.back()-
upper[upper.size()-2], pts[i]-upper.back()) > -eps)
            upper.pop_back();
        lower.push_back(pts[i]);
        upper.push_back (pts[i]);
    }
    lower.insert (lower.end(), upper.rbegin() + 1,
upper.rend());
    return lower;
}
```

Geometry – 3D

Primitives

```
struct point3 {
    double x, y, z;
    point3(double x=0, double y=0, double z=0):x(x),y(y),z(z){}
    point3 operator+(point3 p)const ?{ return point3(x + p.x, y
+ p.y, z + p.z); }
    point3 operator*(double k)const { return point3(k*x, k*y,
k*z); }
    point3 operator-(point3 p)const ?{ return *this + (p*-1.0);}
    point3 operator/(double k)const { return *this*(1.0/k); }
    double norm() { return x*x + y*y + z*z; }
    double abs() { return sqrt(norm()); }
    point3 normalize() { return *this/this->abs(); }
};
// dot product
double dot(point3 a, point3 b) {
    return a.x*b.x + a.y*b.y + a.z*b.z;
}
// cross product
point3 cross(point3 a, point3 b) {
    return point3(a.y*b.z - b.y*a.z, b.x*a.z - a.x*b.z, a.x*b.y
- b.x*a.y);
}
struct line {
    point3 a, b;
    line(point3 A=point3(), point3 B=point3()) : a(A), b(B) {}
    // Direction unit vector a -> b
    point3 dir() { return (b - a).normalize(); }
};
// Returns closest point on an infinite line u to the point p
point3 cpoint_iline(line u, point3 p) {
    point3 ud = u.dir();
    return u.a - ud*dot(u.a - p, ud);
}
// Returns Shortest distance between two infinite lines u and v
double dist_ilines(line u, line v) {
    return dot(v.a - u.a, cross(u.dir(), v.dir()).normalize());
}
// Finds the closest point on infinite line u to infinite line v
// Note: if (uv*uv - uu*vv) is zero then the lines are parallel
// and such a single closest point does not exist. Check for
// this if needed.
point3 cpoint_ilines(line u, line v) {
    point3 ud = u.dir(); point3 vd = v.dir();
    double uu = dot(ud, ud), vv = dot(vd, vd), uv = dot(ud, vd);
    double t = dot(u.a, ud) - dot(v.a, ud); t *= vv;
    t -= uv*(dot(u.a, vd) - dot(v.a, vd));
    t /= (uv*uv - uu*vv);
    return u.a + ud*t;
}
// Closest point on a line segment u to a given point p
point3 cpoint_lineseg(line u, point3 p) {
    point3 ud = u.b - u.a; double s = dot(u.a - p,
ud)/ud.norm();
    if (s < -1.0) return u.b;
    if (s > ?0.0) return u.a;
    return u.a - ud*s;
}
struct plane {
    point3 n, p;
    plane(point3 ni = point3(), point3 pi = point3()) : n(ni),
p(pi) {}
    plane(point3 a, point3 b, point3 c) : n(cross(b-a, c-
a).normalize()), p(a) {}
    //Value of d for the equation ax + by + cz + d = 0
    double d() { return -dot(n, p); }
};
// Closest point on a plane u to a given point p
point3 cpoint_plane(plane u, point3 p) {
    return p - u.n*(dot(u.n, p) + u.d());
}
// Point of intersection of an infinite line v and a plane u.
// Note: if dot(u.n, vd) == 0 then the line and plane do not
// intersect at a single point. Check for this if needed.
point3 iline_isect_plane(plane u, line v) {
    point3 vd = v.dir();
    return v.a - vd*((dot(u.n, v.a) + u.d())/dot(u.n, vd));
}
// Infinite line of intersection between two planes u and v.
// Note: if dot(v.n, uvu) == 0 then the planes do not intersect
// at a line. Check for this case if it is needed.
line isect_planes(plane u, plane v) {
    point3 o = u.n*-u.d(), uv = cross(u.n, v.n);
    point3 uvu = cross(uv, u.n);
    point3 a = o - uvu*((dot(v.n, o) + v.d())/(dot(v.n,
uvu)*uvu.norm()));
    return line(a, a + uv);
}
// Returns great circle distance (lat[-90,90], long[-180,180])
double greatcircle(double lt1, double lo1, double lt2, double
lo2, double r) {
    double a = M_PI*(lt1/180.0), b = M_PI*(lt2/180.0);
    double c = M_PI*((lo2-lo1)/180.0);
    return r*acos(sin(a)*sin(b) + cos(a)*cos(b)*cos(c));
}
// Rotates point p around directed line a->b with angle 'theta'
point3 rotate(point3 a, point3 b, point3 p, double theta) {
    point3 o = cpoint_iline(line(a,b),p);
    point3 perp = cross(b-a,p-o);
    return o+perp*sin(theta)+(p-o)*cos(theta);
}
```

Convex-hull 3D [O(N^2)]

```
// vector<hullFinder::hullFace> hull=hullFinder(pts).findHull();
// 'hull' will have triangular faces of convex-hull of the given
// points 'pts'. Some of them might be co-planar.
// There are O(pts.size()) of those disjoint triangles that
// cover all surface of convex hull
// Each element of hull is a hullFace which has indices of three
// vertices of a triangle
bool operator==(const point3 &p, const point3 &q) {
    return (abs(p.x - q.x) < eps) && (abs(p.y - q.y) < eps) && (abs(p.z - q.z) < eps);
}
point3 triNormal(const point3 &a, const point3 &b, const point3 &c) {
    return cross(a, b) + cross(b, c) + cross(c, a);
}
class hullFinder {
    const vector<point3> &pts;
    public:
    hullFinder(const vector<point3> &pts_) : pts(pts_), halfE(pts.size(), -1) {}
    struct hullFace {
        int u, v, w; point3 n;
        hullFace(int u_, int v_, int w_, const point3 &n_) : u(u_), v(v_), w(w_), n(n_) {}
    };
    vector<hullFinder::hullFace> findHull() {
        vector<hullFace> hull;
        int n = pts.size();
        if (n < 4) return hull;
        int p3 = 2; point3 tNorm;
        while ((p3 < n) && ((tNorm = triNormal(pts[0], pts[1], pts[p3])) == point3())) ++p3;
        int p4 = p3+1;
        while ((p4 < n) && (abs(dot(tNorm, pts[p4] - pts[0])) < eps)) ++p4;
        if (p4 >= n) return hull;
        edges.clear();
        edges.push_front(hullEdge(0, 1)); setF1(edges.front(), p3); setF2(edges.front(), p3);
        edges.push_front(hullEdge(1, p3)); setF1(edges.front(), 0); setF2(edges.front(), 0);
        edges.push_front(hullEdge(p3, 0)); setF1(edges.front(), 1); setF2(edges.front(), 1);
        addPt(p4);
        for (int i = 2; i < n; ++i)
            if ((i != p3) && (i != p4))
                addPt(i);
        for (list<hullEdge>::const_iterator e = edges.begin(); e != edges.end(); ++e) {
            if ((e->u < e->v) && (e->u < e->f1))
                hull.push_back(hullFace(e->u, e->v, e->f1, e->n1));
            else if ((e->v < e->u) && (e->v < e->f2))
                hull.push_back(hullFace(e->v, e->u, e->f2, e->n2));
        }
        return hull;
    }
    private:
    struct hullEdge {
        int u, v, f1, f2;
        point3 n1, n2;
        hullEdge(int u_, int v_):u(u_), v(v_), f1(-1), f2(-1) {}
    };
    list<hullEdge> edges;
    vector<int> halfE;
    void setF1(hullEdge &e, int f1) {
        e.f1 = f1;
        e.n1 = triNormal(pts[e.u], pts[e.v], pts[e.f1]);
    }
    void setF2(hullEdge &e, int f2) {
        e.f2 = f2;
        e.n2 = triNormal(pts[e.v], pts[e.u], pts[e.f2]);
    }
    void addPt(int i) {
        for (list<hullEdge>::iterator e = edges.begin(); e !=
edges.end(); ++e) {
            bool v1 = dot(pts[i] - pts[e->u], e->n1) > eps;
            bool v2 = dot(pts[i] - pts[e->u], e->n2) > eps;
            if (v1 && v2)
                e = --edges.erase(e);
            else if (v1) {
                setF1(*e, i);
                addCone(e->u, e->v, i);
            }
            else if (v2) {
                setF2(*e, i);
                addCone(e->v, e->u, i);
            }
        }
    }
    void addCone(int u, int v, int apex) {
        if (halfE[v] != -1) {
            edges.push_front(hullEdge(v, apex));
            setF1(edges.front(), u); setF2(edges.front(),
halfE[v]);
            halfE[v] = -1;
        }
        else halfE[v] = u;
        if (halfE[u] != -1) {
            edges.push_front(hullEdge(apex, u));
            setF1(edges.front(), v); setF2(edges.front(),
halfE[u]);
            halfE[u] = -1;
        }
        else halfE[u] = v;
    }
};
```

Combinatorics

(Un)Ranking of K-permutation out of N [O(K)]

```
void rec_unrank_perm(int n, int k, long long r, vector<int> &id,
vector<int> &pi) {
    if(k>0) {
        swap(id[n-1],id[r%n]);
        rec_unrank_perm(n-1,k-1,r/n,id,pi);
        pi.push_back(id[n-1]);
        swap(id[n-1],id[r%n]);
    }
}
// Returns a k-permutation corresponds to rank 'r' of n objects.
// 'id' should be a full identity permutation of size at least n
// and it remains the same at the end of the function
vector<int> unrank_perm(int n, int k, long long r, vector<int>
&id) {
    vector<int> ans;
    rec_unrank_perm(n,k,r,id,ans);
    return ans;
}
long long rec_rank_perm(int n, int k, vector<int> &pirev,
vector<int> &pi) {
    if(k==0)
        return 0;
    int s = pi[k-1];
    swap(pi[k-1],pi[pirev[n-1]-(n-k)]);
    swap(pirev[s],pirev[n-1]);
    long long ans = s+n*rec_rank_perm(n-1,k-1,pirev,pi);
    swap(pirev[s],pirev[n-1]);
    swap(pi[k-1],pi[pirev[n-1]-(n-k)]);
    return ans;
}
// Returns rank of the k-permutaion 'pi' of n objects.
// 'id' should be a full identity permutation of size at least n
// and it remains the same at the end of the function
long long rank_perm(int n, vector<int> &id, vector<int> pi) {
    for(int i=0;i<pi.size();i++)
        id[pi[i]] = i+n-pi.size();
    long long ans = rec_rank_perm(n, pi.size(), id, pi);
    for(int i=0;i<pi.size();i++)
        id[pi[i]] = pi[i];
    return ans;
}
```

(Un)Ranking of K-combination out of N [O(K log N)]

```
const int maxn = 100;
const int maxk = 10;
// combination[i][j] = j!/(i!*(j-i)!)
long long combination[maxk][maxn];
long long cumsum[maxk][maxn];
void initialize() { //~O(nk)
    memset(combination,0,sizeof combination);
    for(int i=0;i<maxn;i++)
        combination[0][i]=1;
    for(int i=1;i<maxk;i++)
        for(int j=1;j<maxn;j++)
            combination[i][j] = combination[i][j-
1]+combination[i-1][j-1];
    for(int i=0;i<maxk;i++)
        cumsum[i][0] = combination[i][0];
    for(int i=0;i<maxk;i++)
        for(int j=1;j<maxn;j++)
            cumsum[i][j] = cumsum[i][j-1]+combination[i][j];
}
// Returns rank of the given combination 'c' of n objects.
long long rank_comb(int n, vector<int> c) {
    long long ans = 0;
    int prev = -1;
    sort(c.begin(),c.end()); // comment this if it is sorted
    for(int i=0;i<c.size();i++) {
        ans += cumsum[c.size()-i-1][n-prev-2]-cumsum[c.size()-i-
1][n-c[i]-1];
        prev = c[i];
    }
    return ans;
}
struct comp{
    long long base;
    comp(long long base):base(base){}
    int operator ()(const long long &a,const long long &val) {
        return (base-a)>val;
    }
};
// Returns k-combination of rank 'r' of n objects
vector<int> unrank_comb(int n, int k, long long r) {
    vector<int> c;
    int prev = -1;
    for(int i=0;i<k;i++) {
        long long base = cumsum[k-i-1][n-prev-2];
        prev = n-1-(lower_bound(cumsum[k-i-1],cumsum[k-i-1]+n-
prev-1,r,comp(base))-cumsum[k-i-1]);
        r -= base-cumsum[k-i-1][n-prev-1];
        c.push_back(prev);
    }
    return c;
}
```

Graph Theory

Fast flow [O(V^{2}E)]

```
// find_flow returns max flow from s to t in an n-vertex graph.
// Use add_edge to add edges (directed/undirected) to the graph.
// Call clear_flow() before each testcase.
int c[maxn][maxn];
vector<int> adj[maxn];
int par[maxn];
int dcount[maxn+maxn];
int dist[maxn];
void add_edge(int a,int b,int cap,int rev_cap=0){
    c[a][b]+=cap;
    c[b][a]+=rev_cap;
    adj[a].push_back(b);
    adj[b].push_back(a);
}
void clear_flow(){
    memset(c,0,sizeof c);
    memset(dcount,0,sizeof dcount);
    for (int i=0;i<maxn;++i)
```

```
        adj[i].clear();
}
int advance(int v){
    for (int i=0;i<adj[v].size();++i){
        int w=adj[v][i];
        if (c[v][w]>0 && dist[v]==dist[w]+1){
            par[w]=v;
            return w;
        }
    }
    return -1;
}
int retreat(int v){
    int old=dist[v];
    --dcount[dist[v]];
    for (int i=0;i<adj[v].size();++i){
        int w=adj[v][i];
        if (c[v][w]>0)
            dist[v]=min(dist[v],dist[w]);
    }
    ++dist[v];
    ++dcount[dist[v]];
    if (dcount[old]==0)
        return -1;
    return par[v];
}
int augment(int s,int t){
    int delta=c[par[t]][t];
    for (int v=t;v!=s;v=par[v])
        delta=min(delta,c[par[v]][v]);
    for (int v=t;v!=s;v=par[v]){
        c[par[v]][v]-=delta;
        c[v][par[v]]+=delta;
    }
    return delta;
}
queue<int> q;
void bfs(int v){
    memset(dist,-1,sizeof dist);
    while (!q.empty()) q.pop();
    q.push(v);
    dist[v]=0;
    ++dcount[dist[v]];
    while (!q.empty()){
        v=q.front();
        q.pop();
        for (int i=0;i<adj[v].size();++i){
            int w=adj[v][i];
            if (c[w][v]>0 && dist[w]==-1){
                dist[w]=dist[v]+1;
                ++dcount[dist[w]];
                q.push(w);
            }
        }
    }
}
int find_flow(int n,int s,int t){
    bfs(t);
    int v=s;
    par[s]=s;
    int ans=0;
    while (v!=-1 && dist[s]<n){
        int newv=advance(v);
        if (newv!=-1)
            v=newv;
        else
            v=retreat(v);
        if (v==t){
            v=s;
            ans+=augment(s,t);
        }
    }
    return ans;
}
```

Flow and negative flow

```
const int inf=(int)1e9;
const int maxn = 300;
int x[maxn][maxn],m;
int c[maxn][maxn],n;
int f[maxn][maxn];
int flow_k,flow_t,mark[maxn];
int dfs(int v,int m){
    if (v==flow_t) return m;
    for (int i=0,x;i<n;++i)
        if ((c[v][i]-f[v][i]>=flow_k) && !mark[i]++)
            if (x=dfs(i,min(m,c[v][i]-f[v][i])))
                return (f[i][v]=-(f[v][i]+=x)),x;
    return 0;
}
// Input: n(# of vertices),s(source),t(sink),c[n][n](capacities)
// Finds flow from i to j (i.e. f[i][j]) in the maximum flow
// where f[i][j]=-f[j][i]
// Requirements: f[i][j] should be filled with initial flow
// before calling the function and c[i][j] >= f[i][j]
void flow(int s,int t){
    int flow_ans =0;
    flow_t = t;
    flow_k = 1;
    for (int i=0;i<n;++i)
        for (int j=0;j<n;++j)
            for (;flow_k<c[i][j];flow_k*=2);
    for (;flow_k;flow_k/=2){
        memset(mark,0,sizeof mark);
        for (;dfs(s,inf);)
            memset(mark,0,sizeof mark);
    }
}
// Input: m(# of vertices), x[m][m](capacities)
// Finds f[i][j] in a circular flow satisfying x[i][j]
// If you have a real sink and source set x[sink][source]=inf
// x[i][j]<0 means capacity of i->j is zero and a flow of at
least abs(x[i][j]) should go from j to i.
// If you have two capacities for i->j and j->i and some
// min flow for at least one of them you should resolve this
// before calling the function by filling some flow in f[i][j]
// and f[j][i]
// Returns false when can't satisfy x and returns false when
// x[i][j] and x[j][i] are both negative. Check this if needed
bool negative_flow(){
    for (int i=0;i<m;++i)
        for (int j=0;j<m;++j){
            if (x[i][j]<0){
                if (x[j][i]<0) return false;
                continue;
            }
            if (x[j][i]>=0){
                c[i][j]=x[i][j];
                continue;
            }
            c[i][j]=x[i][j]+x[j][i];
            c[j][i]=0;
            c[i][m+1]-=x[j][i];
            c[m][j]-=x[j][i];
            if (c[i][j]<0) return false;
        }
    n=m+2;
    flow(n-2,n-1);
    for (int i=0;i<m;++i)
        if (c[m][i]!=f[m][i])
            return false;
    for (int i=0;i<m;++i)
        for (int j=0;j<m;++j)
            if (x[i][j]<0){
                f[i][j]+=x[i][j];
                f[j][i]-=x[i][j];
            }
    return true;
}
```

Min cost max flow

```
//Input (zero based, non-negative edges):
//    n = |V|, e = |E|, s = source, t = sink
//    cost[v][u] = cost for each unit of flow from v to u
//    cap[v][u] = capacity
//Output of mcf():
//    Flow contains the flow value
//    Cost contains the minimum cost
//    f[n][n] contains the flow
const int maxn = 300;
const int inf = 1e9;
int cap[maxn][maxn],cost[maxn][maxn], f[maxn][maxn];
int p[maxn], d[maxn] , mark[maxn], pi[maxn];
int n , s , t, Flow , Cost;
int pot(int u,int v){
    return d[u] + pi[u] - pi[v];
}
int dijkstra(){
    memset( mark , 0 , sizeof mark );
    memset( p , -1 , sizeof p );
    for( int i = 0; i <= n; i++ )
        d[i] = inf;
    d[s] = 0;
    while(1){
        int u = n;
        for( int i=0; i<n; i++ )
            if( !mark[i] && d[i] < d[u] )
                u = i;
        if(u==n) break;
        mark[u] = 1;
        for (int v=0 ; v<n ; v++){
            if(!mark[v] && f[v][u] && d[v]>pot(u,v)-cost[v][u]){
                d[v] = pot(u,v) - cost[v][u];
                p[v] = u;
            }
            if( !mark[v] && f[u][v] < cap[u][v] && d[v] >
pot(u,v) + cost[u][v] ){
                d[v] = pot(u,v) + cost[u][v];
                p[v] = u;
            }
        }
    }
    for( int i = 0; i < n; i++ )
        if( pi[i] < inf )
            pi[i] += d[i];
    return mark[t];
}
void mcf(){
    memset( f , 0 , sizeof f );
    memset( pi , 0 , sizeof pi );
    Flow = Cost = 0;
    while(dijkstra()){
        int min = inf;
        for (int x = t ; x!=s ; x=p[x])
            if (f[x][p[x]])
                min = std::min(f[x][p[x]], min);
            else
                min = std::min(cap[p[x]][x] - f[p[x]][x], min);
        for (int x = t ; x!=s ; x=p[x])
            if (f[x][p[x]]){
                f[x][p[x]] -= min;
                Cost -= min*cost[x][p[x]];
            }else{
                f[p[x]][x] += min;
                Cost += min*cost[p[x]][x];
            }
        Flow += min;
    }
}
```

2-Sat & strongly connected component [O(V+E)]

```
// Vertices are numbered 0..n-1 for true states.
// False state of the variable i is i+n (i.e. other(i))
// For SCC 'n', 'adj' and 'adjrev' need to be filled.
// For 2-Sat set 'n' and use add_edge
// 0<=val[i]<=1 is the value for binary variable i in 2-Sat
// 0<=group[i]<2*n is the scc number of vertex i.
int n;
vector<int> adj[maxn*2];
vector<int> adjrev[maxn*2];
int val[maxn];
int marker,dfst,dfstime[maxn*2],dfsorder[maxn*2];
int group[maxn*2];
// For 2SAT Only
inline int other(int v){return v<n?v+n:v-n;}
inline int var(int v){return v<n?v:v-n;}
inline int type(int v){return v<n?1:0;}
//
void satclear() {
    for(int i=0;i<maxn+maxn;i++) {
        adj[i].resize(0);
        adjrev[i].resize(0);
    }
}
void dfs(int v){
    if(dfstime[v]!=-1)
        return;
    dfstime[v]=-2;
    int deg = adjrev[v].size();
    for(int i=0;i<deg;i++)
        dfs(adjrev[v][i]);
    dfstime[v] = dfst++;
}
void dfsn(int v) {
    if(group[v]!=-1)
        return;
    group[v]=marker;
    int deg=adj[v].size();
    for(int i=0;i<deg;i++)
        dfsn(adj[v][i]);
}
// For 2SAT Only
void add_edge(int a,int b) {
    adj[other(a)].push_back(b);
    adjrev[a].push_back(other(b));
    adj[other(b)].push_back(a);
    adjrev[b].push_back(other(a));
}
//
int solve() {
    dfst=0;
    memset(dfstime,-1,sizeof dfstime);
    for(int i=0;i<n+n;i++)
        dfs(i);
    memset(val,-1,sizeof val);
    for(int i=0;i<n+n;i++)
        dfsorder[n+n-dfstime[i]-1]=i;
    memset(group,-1,sizeof group);
    for(int i=0;i<n+n;i++) {
        marker=i;
        dfsn(dfsorder[i]);
    }
    // For 2SAT Only
    for(int i=0;i<n;i++) {
        if(group[i]==group[i+n])
            return 0;
        val[i]=(group[i]>group[i+n])?0:1;
    }
    //
    return 1;
}
```

Bipartite matching, vertex cover, edge cover, disjoint set [O(VE)]

```
// Input:
//    n: size of part1, m: size of part2
//    a[i]: neighbours of i-th vertex of part1
//    b[i]: neighbours of i-th vertex of part2
const int maxn=2020, maxm=2020;
int n, m;
vector <int> a[maxn], b[maxm];
int matched[maxn], mark[maxm], mate[maxm];
int dfs(int v){
    if (v<0) return 1;
    for (int i=0;i<a[v].size();++i)
        if (!mark[a[v][i]]++ && dfs(mate[a[v][i]]))
            return matched[mate[a[v][i]]=v]=1;
    return 0;
}
int set_mark(){
    memset(matched,0,sizeof matched);
    memset(mate,-1,sizeof mate);
    memset(mark,0,sizeof mark);
    for (int i=0;i<n;++i)
        for (int j=0;j<a[i].size();++j)
            if (mate[a[i][j]]<0){
                matched[mate[a[i][j]]=i]=1;
                break;
            }
    for (int i=0;i<n;++i)
        if (!matched[i] && dfs(i))
            memset(mark,0,sizeof mark);
    for (int i=0;i<n;++i)
        if (!matched[i])
            dfs(i);
}
// res.size(): size of matching
// res[i]: i-th edge of matching
// res[i].first is in part1, res[i].second is in part2
void matching (vector<pair<int,int> > &res){
    set_mark();
    res.clear();
    for (int i=0;i<m;++i)
        if (mate[i]>=0)
            res.push_back(pair <int,int> (mate[i], i));
}
// p1: vertices in part1, p2: vertices in part2
// union of p1 and p2 cover the edges of the graph
void vertex_cover (vector<int> &p1, vector<int> &p2){
    set_mark();
    p1.clear();
    p2.clear();
    for (int i=0;i<m;++i)
        if (mate[i]>=0)
            if (mark[i])
                p2.push_back(i);
            else
                p1.push_back(mate[i]);
}
// p1: vertices in part1, p2: vertices in part2
// union of p1 and p2 is the largest disjoint set of the graph
void disjoint_set (vector<int> &p1, vector<int> &p2){
    set_mark();
    p1.clear();
    p2.clear();
    for (int i=0;i<m;++i)
        if (mate[i]>=0 && mark[i])
            p1.push_back(mate[i]);
        else
            p2.push_back(i);
    for (int i=0;i<n;++i)
        if (!matched[i])
            p1.push_back(i);
}
// edges in res cover the vertices of the graph
// res[i].first is in part1, res[i].second is in part2
void edge_cover(vector<pair<int,int> > &res){
    set_mark();
    res.clear();
    for (int i=0;i<m;++i)
        if (mate[i]>=0)
            res.push_back(pair<int,int> (mate[i],i));
        else if (b[i].size())
            res.push_back(pair<int,int> (b[i][0],i));
    for (int i=0;i<n;++i)
        if (!matched[i] && a[i].size())
            res.push_back(pair<int,int> (i,a[i][0]));
}
```

Bipartite weighted matching [O(VE^2)]

```
// Input: n, m, w[n][m] (n <= m)
//        w[i][j] is the weight between the i-th vertex of part1
//        and the j-th vertex of part2. w[i][j] can be any
//        integer (including negative values)
// Output: res, size of res is n
const int inf = 1e7;
const int maxn=200,maxm=200;
int n, m, w[maxn][maxm],u[maxn], v[maxm];
int mark[maxn],mate[maxm], matched[maxn];
int dfs(int x){
    if (x<0) return 1;
    if (mark[x]++) return 0;
    for (int i=0 ; i<m ; i++)
        if (u[x]+v[i]-w[x][i]==0)
            if (dfs(mate[i]))
                return matched[mate[i]=x]=1;
    return 0;
}
void _2matching(){
    memset( mate , -1 , sizeof mate );
    memset( mark , 0 , sizeof mark );
    memset( matched , 0 , sizeof matched );
    for (int i=0 ; i<n ; i++)
        for (int j=0 ; j<m ; j++)
            if (mate[j]<0 && u[i]+v[j]-w[i][j]==0){
                matched[mate[j]=i]=1;
                break;
            }
    for (int i=0 ; i<n ; i++)
        if (!matched[i])
            if (dfs(i))
                memset( mark , 0 , sizeof mark );
}
void wmatching(vector <pair<int, int> > &res){
    for (int i=0 ; i<m ; i++)
        v[i] = 0;
    for (int i=0 ; i<n ; i++){
        u[i] = -inf;
        for (int j=0 ; j<m ; j++)
            u[i] = max(u[i],w[i][j]);
    }
    memset( mate , -1 , sizeof mate );
    memset( matched , 0 , sizeof matched );
    int counter = 0;
    while (counter!=n){
        for (int flag = 1; flag ; ){
            flag = 0;
            memset( mark , 0 , sizeof mark );
            for (int i=0 ; i<n ; i++)
                if (!matched[i] && dfs(i)){
                    counter++;
                    flag = 1;
                    memset(mark,0,sizeof mark);
                }
        }
        int epsilon = inf;
        for (int i=0 ; i<n ; i++)
            for (int j=0 ; j<m ; j++){
                if (!mark[i]) continue;
                if (mate[j]>=0)
                    if (mark[mate[j]]) continue;
                epsilon = min(epsilon, u[i] + v[j] - w[i][j]);
            }
        for (int i=0 ; i<n ; i++)
            if (mark[i])
                u[i] -= epsilon;
        for (int j=0 ; j<m ; j++)
            if (mate[j]>=0)
                if (mark[mate[j]])
                    v[j] += epsilon;
    }
    res.clear();
    for (int i=0 ; i<m ; i++)
        if (mate[i]!=-1)
            res.push_back(pair<int,int>(mate[i],i));
}
```

Cut edges and 2-edge-connected components [O(V+E)]

```
//input (zero based):
//       g[n] should be the adjacency list of the graph
//       g[i] is a vector of int
//output of cut_edge():
//        cut_edges is a vector of pair<int, int>
//        comp[comp_size] contains the 2 connected components
//        comp[i] is a vector of int
const int maxn = 1000;
typedef pair<int, int> edge;
vector<int> g[maxn];
int n, mark[maxn] , d[maxn] , jad[maxn];
vector<edge> cut_edges;
//for components only
vector<int> comp[maxn];
int comp_size;
vector<int> comp_stack;
//
void dfs(int x, int level){
    mark[x] = 1;
    //for components only
    comp_stack.push_back(x);
    //
    int t = 0;
    for (int i=0 ; i<(int)g[x].size() ; i++){
        int u = g[x][i];
        if (!mark[u]){
            jad[u] = d[u] = d[x] + 1;
            dfs(u, level+1);
            jad[x] = std::min(jad[u], jad[x]);
            if (jad[u]==d[u]){
                cut_edges.push_back(edge(u, x));
                //for components only
                while (comp_stack.back() != u){
                   comp[comp_size].push_back(comp_stack.back());
                   comp_stack.pop_back();
                }
                comp[comp_size++].push_back(u);
                comp_stack.pop_back();
                //
            }
        }else{
            if (d[u] == d[x] - 1) t++;
            if (d[u] != d[x] - 1 || t!=1)
                jad[x] = std::min(d[u], jad[x]);
        }
    }
    //for components only
    if (level == 0){
        while (comp_stack.size() > 0){
            comp[comp_size].push_back(comp_stack.back());
            comp_stack.pop_back();
        }
        comp_size++;
    }
    //
}
void cut_edge(){
    memset( mark , 0 , sizeof mark );
    memset( d , 0 , sizeof d );
    memset( jad , 0  , sizeof jad );
    cut_edges.clear();
    //for components only
    for (int i=0 ; i<maxn ; i++) comp[i].clear();
    comp_stack.clear();
    comp_size = 0;
    //
    for (int i=0 ; i<n ; i++)
        if (!mark[i]) dfs(i, 0);
}
```

Cut vertices and 2-connected components [O(V+E)]

```
//Input (zerobased):
//       g[n] should be the adjacency list of the graph
//       g[i] is a vector of int
//Output of cut_ver():
//       cut_vertex is a vector of int
//       comp[comp_size] contains the 2 connected components
//       comp[i] is a vector of int
const int maxn = 1000;
vector<int> g[maxn];
int d[maxn] , mark[maxn] , mark0[maxn] , jad[maxn];
int n;
vector<int> cut_vertex;

//for components only
vector<int> comp[maxn];
int comp_size;
vector<int> comp_stack;
//

void dfs(int x, int level){
    mark[x] = 1;
    //for components only
    comp_stack.push_back(x);
    //

    for (int i=0 ; i<(int)g[x].size() ; i++){
        int u = g[x][i];
        if (!mark[u]){
            jad[u] = d[u] = d[x] + 1;
            dfs(u, level+1);
            jad[x] = std::min(jad[u], jad[x]);
            if (jad[u] >= d[x] && d[x]){
                cut_vertex.push_back(x);
                //for components only
                while (comp_stack.back() != u){
                   comp[comp_size].push_back(comp_stack.back());
                   comp_stack.pop_back();
                }
                comp[comp_size].push_back(u);
                comp_stack.pop_back();
                comp[comp_size++].push_back(x);
                //

            }
        }else if ( d[u] != d[x] -1 )
            jad[x] = std::min(d[u], jad[x]);
    }
    //for components only
    if (level == 0){
        while (comp_stack.size() > 0){
            comp[comp_size].push_back(comp_stack.back());
            comp_stack.pop_back();
        }
        comp_size++;
    }
    //

}
int dfs0(int x){
    mark0[x] = 1;
    for (int i=0 ; i<(int)g[x].size() ; i++)
        if (!mark0[g[x][i]])
            return dfs0(g[x][i]);
    return x;
}
void cut_ver(){
    memset( mark , 0 , sizeof mark );
    memset( mark0 , 0 , sizeof mark0 );
    memset( d , 0 , sizeof d );
    memset( jad , 0  , sizeof jad );
    //for components only
    for (int i=0 ; i<maxn ; i++) comp[i].clear();
    comp_stack.clear();
    comp_size = 0;
    //
    cut_vertex.clear();
    for (int i=0 ; i<n ; i++)
        if (!mark[i])
            dfs(dfs0(i), 0);
}
```

Dijkstra [O(E log V)]

```
const int maxn = 1000;//Max # of vertices
int n;//# of vertices
vector <pair<int,int> > v[maxn];//weighted adjacency list
int d[maxn];//distance from source

struct comp {
    bool operator () (int a, int b)
    { return (d[a]!=d[b]) ? d[a]<d[b] : a<b; }
};
set <int,comp> mark;

void dijkstra (int source) {
    memset(d, -1, sizeof d);
    d[source] = 0;
    mark.clear();
    for (int i=0;i<n;++i)
        mark.insert(i);

    while (mark.size()){
        int x = *mark.rbegin();
        mark.erase(x);
        if (d[x]==-1)
            break;
        for (vector<pair<int,int> >::iterator it = v[x].begin()
; it != v[x].end() ; ++it){
            if (d[it->first]==-1 || d[x]+it->second < d[it-
>first]){
                mark.erase(it->first);
                d[it->first] = d[x]+it->second;
                mark.insert (it->first);
            }
        }
    }
}
```

Number Theory

Sieve of Eratosthenes [O(N loglog N)]

```
// Returns all prime numbers in [0,n]
int isnprime[maxn];
vector<int> sieve(int n) {
    memset(isnprime,0,sizeof isnprime);
    isnprime[0] = isnprime[1] = 1;
    vector<int> ps;
    for(int i=2;i<n;i++)
        if(!isnprime[i]) {
            ps.push_back(i);
            if(n/i>=i)
                for(int j=i*i;j<=n;j+=i)
                    isnprime[j]=1;
        }
    return ps;
}
```

Chinese remaindering and ext. Euclidean [O(N log Max(M_i))]

```
typedef long long int LLI;
LLI mod(LLI a, LLI m) { return ( (a%m) + m) % m; }

// Assumes non-negative input. Returns d such that d=a*ss+b*tt
LLI gcdex(LLI a, LLI b, LLI &ss, LLI &tt) {
    if (b==0){
        ss = 1;
        tt = 0;
        return a;
    }
    LLI g = gcdex(b,a%b,tt,ss);
    tt = tt - (a/b) * ss;
    return g;
}
// Returns x such that 0<=x<lcm(m_0, ..., m_(n-1)) and
// x==a_i (mod m_i), if such an x exists. If x does not exist -1
// is returned.
LLI chinese_rem(vector<LLI> &a, vector<LLI> &m) {
    LLI g, s, t, a_tmp, m_tmp;
    a_tmp = mod(a[0], m[0]);
    m_tmp = m[0];
    for (int i = 1; i < a.size(); ++i) {
        g = gcdex(m_tmp, m[i], s, t);
        if ((a_tmp - a[i]) % g) return -1;
        a_tmp = mod(a_tmp + (a[i] - a_tmp) / g * s * m_tmp,
m_tmp/g*m[i]);
        m_tmp = m[i] * m_tmp / gcdex(m[i], m_tmp, s, t);
    }
    return a_tmp;
}
```

Discrete logarithm solver [O(sqrt(P)]

```
// Given prime P, B>0, and N, finds least L
// such that B^L==N (mod P)
// Returns -1, if no such L exist.
map<int,int> mow;
int times(int a, int b, int m) {
    return (long long) a * b % m;
}
int power(int val, int power, int m) {
    int res = 1;
    for (int p = power; p; p >>= 1) {
        if (p & 1)
            res = times(res, val, m);
        val = times(val, val, m);
    }
    return res;
}
int discrete_log(int p, int b, int n) {
    int jump = sqrt(double(p));
    mow.clear();
    for (int i = 0; i < jump && i < p-1; ++i)
        mow[power(b,i,p)] = i+1;
    for (int i = 0, j; i < p-1; i += jump)
        if (j = mow[times(n,power(b,p-1-i,p),p)])
            return (i+j-1)%(p-1);
    return -1;
}
```

String

Manacher's algorithm [O(N)]

```
// Returns half of length of largest panlindrome centered at
// every position in the string
vector<int> manacher(string s) {
    vector<int> ans(s.size(),0);
    int maxi = 0;
    for(int i=1;i<s.size();i++) {
        int k = 0;
        if(maxi+ans[maxi]>=i)
            k = min(ans[maxi]+maxi-i,ans[2*maxi-i]);
        for(;s[i+k]==s[i-k] && i-k>=0 && i+k<s.size();k++);
        ans[i] = k-1;
        if(i+ans[i]>maxi+ans[maxi])
            maxi = i;
    }
    return ans;
}
```

KMP string matching [O(N+M)]

```
// Given strings t and p, return the indices of t where p occurs
// as a substring
vector<int> compute_prefix(string s) {
    vector<int> pi(s.size(),-1);
    int k = -1;
    for (int i=1; i<s.size(); i++) {
        while (k>=0 && s[k+1] != s[i])
            k = pi[k];
        if (s[k+1]==s[i]) k++;
        pi[i] = k;
    }
    return pi;
}
vector<int> kmp_match(string t, string p) {
    vector<int> pi = compute_prefix(p);
    vector<int> shifts;
    int m=-1;
    for (int i=0; i<t.size(); i++) {
        while (m>-1 && p[m+1]!=t[i]) m = pi[m];
        if (p[m+1] == t[i]) m++;
        if (m == p.size()-1) {
            shifts.push_back(i+1-p.size());
            m = pi[m];
        }
    }
    return shifts;
}
```

Suffix array [O(N log N)]

```
// Calculate the order of suffix starting from j-th character
// with length 2^i compared to other starting points
// order[i][j]>=0: order of suffix starting from j-th character
// with length 2^i
// suffix(j1,i)=suffix(j2,i) -> order[i][j1]=order[i][j2]
// suffix(j1,i)<suffix(j2,i) -> order[i][j1]<order[i][j2]
typedef pair<int,int> pii;
typedef pair<pii,int> p3i;
int order[maxlog][maxn];
// if N*log^2(N) is good enough don't write the next function
vector<p3i> buck[maxn];
void radix(vector<p3i> &a, int n, int t){
    for (int i=0 ; i<=n ; i++)
        buck[i].clear();
    for (int i=0 ; i<a.size() ; i++){
        int x;
        switch(t){
            case 1: x = a[i].first.first; break;
            case 2: x = a[i].first.second; break;
            case 3: x = a[i].second; break;
        }
        buck[x+1].push_back(a[i]);
    }
    a.clear();
    for (int i=0 ; i<=n ; i++)
        for (int j=0 ; j<buck[i].size() ; j++)
            a.push_back(buck[i][j]);
}
void suffix_array(vector<int> in) {
    int n = in.size();
    vector<p3i> sorted;
    for(int i=0;i<n;i++)
        sorted.push_back(p3i(pii(in[i],in[i]),i));
    sort(sorted.begin(), sorted.end());
    for(int k=0;k<maxlog;k++) {
        int cur = 0;z
        for (int i=0;i<n;i++) {
            if(i>0 && sorted[i-1].first!=sorted[i].first)
                cur++;
            order[k][sorted[i].second] = cur;
        }
        for(int i=0;i<n;i++) {
            int o1 = order[k][i];
            int o2 = -1;
            // Uncomment next line for non-circular sorting
            // if (i+(1<<k)<n)
                o2 = order[k][(i+(1<<k))%n];
            sorted[i] = p3i(pii(o1,o2),i);
        }
        // if n*log^2(n) is good enough use the following line
instead of the three radixes
        // sort(sorted.begin(), sorted.end());
        radix(sorted, n, 3);
        radix(sorted, n, 2);
        radix(sorted, n, 1);
    }
}
int common_prefix(int n, int i, int j) {
    int ans = 0;
    // Uncomment next line for non-circular sorting
    // if(i==j) return n-i-1;
    for(int k=maxlog-1;k>=0;k--) {
        if(order[k][i]==order[k][j]) {
            i=(i+(1<<k))%n;
            j=(j+(1<<k))%n;
            ans+=1<<k;
        }
    }
    return min(ans,n);
}
```

Misc

Longest ascending subsequence [O(N log N)]

```
typedef pair<int,int> pii;
int comp(const pii &a, const pii &b) {
    if(a.first!=b.first)
        return a.first<b.first;
    return a.second<b.second; // return 0 to find strictly
ascending subsequence
}
vector<int> lis(const vector<int> &in) {
    vector<pii> l;
    vector<int> par(in.size(),-1);
    for(int i=0;i<in.size();i++) {
        int ind =
lower_bound(l.begin(),l.end(),pii(in[i],i),comp)-l.begin();
        if(ind==l.size())
            l.push_back(pii(0,0));
        l[ind] = pii(in[i],i);
        if(ind!=0)
            par[i] = l[ind-1].second;
    }
    vector<int> ans;
    int ind = l.back().second;
    while(ind!=-1) {
        ans.push_back(in[ind]);
        ind = par[ind];
    }
    reverse(ans.begin(),ans.end());
    return ans;
}
```

Simplex

```
// m - number of (less than) inequalities
// n - number of variables
// c - (m+1) by (n+1) array of coefficients:
//   row 0        - objective function coefficients
//   row 1:m      - less-than inequalities
//   column 0:n-1 - inequality coefficients
//   column n     - inequality constants (0 for obj. function)
// x[n] - result variables
// Returns value - maximum value of objective function
//   (-inf for infeasible, inf for unbounded)
const int maxm = 400;  // leave one extra
const int maxn = 400;  // leave one extra
const double eps = 1e-9;
const double inf = 1.0/0.0;
double ine[maxm][maxn];
int basis[maxm], out[maxn];
void pivot(int m, int n, int a, int b) {
    int i,j;
    for (i=0;i<=m;i++)
        if (i!=a)
            for (j=0;j<=n;j++)
                if (j!=b)
                    ine[i][j] -= ine[a][j]*ine[i][b]/ine[a][b];
    for (j=0;j<=n;j++)
        if (j!=b) ine[a][j] /= ine[a][b];
    for (i=0;i<=m;i++)
        if (i!=a) ine[i][b] = -ine[i][b]/ine[a][b];
    ine[a][b] = 1/ine[a][b];
    i = basis[a];
    basis[a] = out[b];
    out[b] = i;
}
double simplex(int m, int n, double c[][maxn], double x[]) {
    int i,j,ii,jj;
    for (i=1;i<=m;i++)
        for (j=0;j<=n;j++)
            ine[i][j] = c[i][j];
    for (j=0;j<=n;j++)
        ine[0][j] = -c[0][j];
    for (i=0;i<=m;i++)
        basis[i] = -i;
    for (j=0;j<=n;j++)
        out[j] = j;
    for(;;) {
        for (i=ii=1;i<=m;i++)
            if (ine[i][n]<ine[ii][n] || (ine[i][n]==ine[ii][n]
&& basis[i]<basis[ii]))
                ii=i;
        if (ine[ii][n] >= -eps) break;
        for (j=jj=0;j<n;j++)
            if (ine[ii][j]<ine[ii][jj]-eps || (ine[ii]
[j]<ine[ii][jj]-eps && out[i]<out[j]))
                jj=j;
        if (ine[ii][jj] >= -eps) return -inf;
        pivot(m,n,ii,jj);
    }
    for(;;) {
        for (j=jj=0;j<n;j++)
            if (ine[0][j]<ine[0][jj] || (ine[0][j]==ine[0][jj]
&& out[j]<out[jj]))
                jj=j;
        if (ine[0][jj] > -eps) break;
        for (i=1,ii=0;i<=m;i++)
            if ((ine[i][jj]>eps) &&
                (!ii || (ine[i][n]/ine[i][jj] < ine[ii]
[n]/ine[ii][jj]-eps) ||
                 ((ine[i][n]/ine[i][jj] < ine[ii][n]/ine[ii]
[jj]+eps) &&
                  (basis[i] < basis[ii]))))
                ii=i;
        if (ine[ii][jj] <= eps) return inf;
        pivot(m,n,ii,jj);
    }
    for (j=0;j<n;j++)
        x[j] = 0;
    for (i=1;i<=m;i++)
        if (basis[i] >= 0)
            x[basis[i]] = ine[i][n];
    return ine[0][n];
}
```

Segment tree [O(log N)]

```
const int maxn = 1<<20; //must be a power of 2
long long seg[2*maxn];
// Add the value 'val' to the index 'num'
void add(int num, long long val) {
    num+=maxn;
    while(num>0) {
        seg[num]+=val;
        num>>=1;
    }
}
// returns sum of the elements in range [0,num]
long long get(int num) {
    num+=maxn;
    long long ans = 0;
    ans=seg[num]; // Comment this to change the range to [0,num)
    while(num>0) {
        if(num&1) {
            ans+=seg[num&(~1)];
        }
        num>>=1;
    }
    return ans;
}
```

Equation solving [O(NM(N+M)]

```
const double eps = 1e-7;
bool zero(double a){return (a<eps) && (a>-eps);}
// m = number of equations, n = number of variables,
// a[m][n+1] = coefficients matrix
// Returns double ans[n] containing the solution, if there is no
// solution returns NULL
double* solve(double **a, int m, int n){
    int cur=0;
    for (int i=0;i<n;++i){
        for (int j=cur;j<m;++j)
            if (!zero(a[j][i])){
                if (j!=cur) swap(a[j],a[cur]);
                for (int sat=0;sat<m;++sat){
                    if (sat==cur) continue;
                    double num=a[sat][i]/a[cur][i];
                    for (int sot=0;sot<=n;++sot)
                        a[sat][sot]-=a[cur][sot]*num;
                }
                cur++;
                break;
            }
    }
    for (int j=cur;j<m;++j)
        if (!zero(a[j][n]))
            return NULL;
    double* ans = new double[n];
    for (int i=0,sat=0;i<n;++i){
        ans[i] = 0;
        if (sat<m && !zero(a[sat][i])){
            ans[i] = a[sat][n] / a[sat][i];
            sat++;
        }
    }
    return ans;
}
```

Cubic equation solver

```
//Solves ax^3 + bx^2 + cx + d = 0
vector<double> solve_cubic(double a, double b, double c, double
d) {
    long double a1 = b/a, a2 = c/a, a3 = d/a;
    long double q = (a1*a1 - 3*a2)/9.0, sq = -2*sqrt(q);
    long double r = (2*a1*a1*a1 - 9*a1*a2 + 27*a3)/54.0;
    double z = r*r-q*q*q, theta;
    vector <double> res; res.clear();
    if (z<=0) {
        theta = acos(r/sqrt(q*q*q));
        res.push_back(sq*cos(theta/3.0) - a1/3.0);
        res.push_back(sq*cos((theta+2.0*M_PI)/3.0) - a1/3.0);
        res.push_back(sq*cos((theta+4.0*M_PI)/3.0) - a1/3.0);
        return res;
    }
    double v = pow(sqrt(z)+fabs(r),1/3.0);
    v += q/v;
    v *= (r < 0) ? 1 : -1;
    v -= a1 / 3.0;
    res.push_back(v);
    return res;
}
```

Calendar

```
const int MONTH_DAYS[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

// epoch is the first year of the world
const int epoch = 1700;
class Date{
    public:
        //month is zero based
    int year, month, day;

    Date(){}
    Date(int year, int month, int day):year(year), month(month-1), day(day){}
    bool operator < (const Date &date) const {
        if (year != date.year)
            return year < date.year;
        if (month != date.month)
            return month < date.month;
        return day < date.day;
    }
    friend ostream& operator << (ostream &out, const Date &date) {
        out << date.month+1 << "/" << date.day << "/" << date.year;
        return out;
    }
};
bool isLeap(int year){
    if (year % 400 == 0)
        return true;
    if (year % 100 == 0)
        return false;
    return (year % 4 == 0);
}
int getMonthDays(int year, int month){
    if (month != 1)
        return MONTH_DAYS[month];
    else
        return isLeap(year) ? 29 : 28;
}
//number of leap years between two years
int leapYears(int from, int to){ // [from, to)
    if (from >= to)
        return 0;
    to--;
    int fours = to / 4 - from / 4;
    int hundreds = to / 100 - from / 100;
    int fhundreds = to / 400 - from / 400;
    if (isLeap(from))
        return fours - hundreds + fhundreds + 1;
    return fours - hundreds + fhundreds;
}
int dateToDay (Date date){
    int year = date.year;
    int month = date.month;
    int day = date.day;
    int days = (year - epoch) * 365;
    days += leapYears(epoch, year);
    for (int i=0 ; i<month ; i++)
        days += getMonthDays(year, i);
    days += day;
    return days;
}
Date dayToDate (int days){
    int year = days / 365;
    year += epoch;
    days %= 365;
    while (days <= leapYears(epoch, year)){
        year--;
        days += 365;
    }
    days -= leapYears(epoch, year);
    int month = 0;
    for (; month<12 && days > getMonthDays(year, month);month++)
        days -= getMonthDays(year, month);
    return Date(year, month+1, days);
}
```

C++ IO format

```
#include <iostream> #include <iomanip> #include <cmath>
freopen("test.in","r",stdin);
freopen("test.out","w",stdout);
cout << fixed << setprecision(7) << M_PI << endl; // 3.1415927
cout << scientific << M_PI << endl; // 3.1415927e+000
int x=15, y=12094;
cout << setbase(10) << x << " " << y << endl; // 15 12094
cout << setbase(8)  << x << " " << y << endl; // 17 27476
cout << setbase(16) << x << " " << y << endl; // f 2f3e
x=5; y=9;
cout << setfill('0') << setw(2) << x << ":" << setw(2) << y <<
endl; // 05:09
printf ("%10d\n", 111); //       111
printf ("%010d\n", 111); //0000000111
printf ("%d %x %X %o\n", 200, 200, 200, 200); //200 c8 C8 310
printf ("%010.2f %e %E\n", 1213.1416, 3.1416, 3.1416);
//0001213.14 3.141600e+00 3.141600E+00
printf ("%*.*d\n",10, 5, 111); //     00111
printf ("%-*.*d\n",10, 5, 111); //00111
printf ("%+*.*d\n",10, 5, 111); //    +00111
char in[20]; int d;
scanf ("%s %*s %d",in,&d); //<- it's number 5
printf ("%s %d \n", in,d); //it's 5
```

Formulas

Pick's Theorem: $A=i+\frac{b}{2}-1$ (A:area,i:interior,b:boundary points)

Catalan Numbers: $C_n=\frac{1}{n+1}\binom{2n}{n}=\frac{4i-2}{i+1}C_{n-1}=\sum_{i=0}^{n-1}C_iC_{n-1-i}\ ,C_0=1$

Triangle: $c^2=a^2+b^2-2ab\cos(angle_c)$, $s=\frac{1}{2}(a+b+c)$,

$$inradius=\sqrt{\frac{(s-a)(s-b)(s-c)}{s}}\ ,\ exradii_a=\sqrt{\frac{s(s-b)(s-c)}{(s-a)}}$$

Spherical Cap: $V=\frac{\pi h}{6}(3a^2+h^2)$, $A=2\pi rh$ (a: radius of base of cap, r: radius of sphere, h: height of cap)

Common bugs

* READ THE STATEMENT AGAIN. TELL YOUR TEAMMATE IF NECESSARY
* Double check spell of literals
* Graph: Multiple components, Multiple edges, Loops
* Geometry: Be careful about +pi,-pi
* Initialization: Use memset/clear(). Don't expect global variables to be zero. Care about multiple tests.
* Precision and Range: Use long long if necessary. Use BigInteger/BigDecimal
* Derive recursive formulas that use sum instead of multiplication to avoid overflow.
* Small cases (n=0,1,negative)
* 0-based <=> 1-based
* Division by zero. Integer division a/(double)b
* Stack overflow (DFS on 1e5)
* Infinite loop?
* array bound check. maxn or x*maxn
* Don't use .size()-1 !
* “(int)-3 < (unsigned int) 2” is false!
* Check copy-pasted codes!
* Be careful about -0.0
* Remove debug info!
* Output format: Spaces at the end of line. Blank lines. View the output in VIM if necessary
* Add eps to double before getting floor or round