**Geometry**

```
#include <bits/stdc++.h>
using namespace std;
struct Point
{
    int x, y;
};
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    if (val == 0) return 0;
    return (val > 0)? 1: 2;
}
void convexHull(Point points[], int n)
{
    if (n < 3) return;
    vector<Point> hull;
    int l = 0;
    for (int i = 1; i < n; i++)
        if (points[i].x < points[l].x)
            l = i;
    int p = l, q;
    do
    {
        hull.push_back(points[p]);
        q = (p+1)%n;
        for (int i = 0; i < n; i++)
        {
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;}
        p = q;}
    while (p != l);
    for (int i = 0; i < hull.size(); i++)
        cout << "(" << hull[i].x << ", "<< hull[i].y << ")\n";}
void angle_between_planes(float a1, float b1, float c1, float a2, float b2, float c2)
{
    float d = (a1 * a2 + b1 * b2 + c1 * c2);
    float e1 = sqrt(a1 * a1 + b1 * b1 + c1 * c1);
    float e2 = sqrt(a2 * a2 + b2 * b2 + c2 * c2);

    d = d / (e1 * e2);
    float pi = 3.14159;
    float A = (180 / pi) * (acos(d));
    printf("Angle is %.2f degree", A);}
# Area of a polygon with given n ordered vertices
Area =  1/2 * |[ (x1y2 + x2y3 + ... +xny1) - (x2y1 + x3y2 + ... + x1yn) ] |
# Calculate Volume of Dodecahedron
Volume = (15 + 7√5)*e 3 /4    Where e is length of an edge.
#Calculate volume and surface area of a cone
volume = 1/3(pi * r * r * h)
area = pi * r * s + pi * r^2
# Where s is the slant height of the cone, h is the height
# r^2 + h^2 = s^2
#Check if a line touches or intersects a circle
void checkCollision(int a, int b, int c, int x, int y, int radius){
    int dist = (abs(a * x + b * y + c)) / sqrt(a * a + b * b); // Finding the distance
    if (radius == dist) cout << "Touch" << endl;
    else if (radius > dist) cout << "Intersect" << endl;
    else cout << "Outside" << endl;}
# Check if four segments form a rectangle
struct Segment{
    int ax, ay;
    int bx, by;};
int getDis(pair<int, int> a, pair<int, int> b) {
    return (a.first - b.first)*(a.first - b.first) +
    (a.second - b.second)*(a.second - b.second);}
bool isPossibleRectangle(Segment segments[]){
    set< pair<int, int> > st;
    for (int i = 0; i < N; i++){
        st.insert(make_pair(segments[i].ax, segments[i].ay));
        st.insert(make_pair(segments[i].bx, segments[i].by));}
    if (st.size() != 4) return false;
    set<int> dist;
    for (auto it1=st.begin(); it1!=st.end(); it1++)
        for (auto it2=st.begin(); it2!=st.end(); it2++)
            if (*it1 != *it2)
                dist.insert(getDis(*it1, *it2));
    if (dist.size() > 3)
        return false;
    int distance[3];
    int i = 0;
    for (auto it = dist.begin(); it != dist.end(); it++)
        distance[i++] = *it;
    if (dist.size() == 2) // If line seqments form a square
        return (2*distance[0] == distance[1]);
    # distance of sides should satisfy pythagorean theorem
    return (distance[0] + distance[1] == distance[2]);}
# integers value make a rectangle
bool isRectangle(int a, int b, int c, int d){
    if (a ^ b ^ c ^ d) return false;
    else return true;}
# Check if three straight lines are concurrent or not
bool checkConcurrent(int a1, int b1, int c1, int a2,
                     int b2, int c2, int a3, int b3, int c3){
    return (a3 * (b1 * c2 - b2 * c1) +
            b3 * (c1 * a2 - c2 * a1) +  c3 * (a1 * b2 - a2 * b1) == 0);}
# Check whether a given point lies inside a triangle or not
float area(int x1, int y1, int x2, int y2, int x3, int y3){
    return abs((x1*(y2-y3) + x2*(y3-y1)+ x3*(y1-y2))/2.0);
}
bool isInside(int x1, int y1, int x2, int y2, int x3, int y3, int x, int y){
    float A = area (x1, y1, x2, y2, x3, y3);
    float A1 = area (x, y, x2, y2, x3, y3);
    float A2 = area (x1, y1, x, y, x3, y3);
    float A3 = area (x1, y1, x2, y2, x, y);
    return (A == A1 + A2 + A3);
}
# Check whether a point exists in circle sector or not.
void checkPoint(int radius, int x, int y, float percent, float startAngle){
    float endAngle = 360/percent + startAngle;
    float polarradius = sqrt(x*x+y*y);
```

```
    float Angle = atan(y/x);
    if (Angle>=startAngle && Angle<=endAngle && polarradius<radius)
        printf("Point (%d, %d) exist in the circle sector\n", x, y);
    else
        printf("Point (%d, %d) does not exist in the circle sector\n", x, y);
}
# Check whether four points make a parallelogram
struct point{
    double x, y;
    point() { }
    point(double x, double y)
        : x(x), y(y) { }
    bool operator<(const point& other) const
    {
        if (x < other.x){
            return true;}
        else if (x == other.x){
            if (y < other.y){
                return true;}}
        return false;}};
point getMidPoint(point points[], int i, int j){
    return point((points[i].x + points[j].x) / 2.0, (points[i].y + points[j].y) / 2.0);}
bool isParallelogram(point points[])
{
    map<point, vector<point> > midPointMap;
    int P = 4;
    for (int i = 0; i < P; i++){
        for (int j = i + 1; j < P; j++){
            point temp = getMidPoint(points, i, j);
            midPointMap[temp].push_back(point(i, j));}}
    int two = 0, one = 0;
    for (auto x : midPointMap)
    {
        if (x.second.size() == 2) two++;
        else if (x.second.size() == 1) one++;
        else
            return false;}
    if (two == 1 && one == 4)
        return true;
    return false;}
# Circle and Lattice Points(Lattice Points are points with
coordinates as integers in 2-D space.)
int countLattice(int r){
    if (r <= 0)
        return 0;
    int result = 4;
    for (int x=1; x<r; x++){
        int ySquare = r*r - x*x;
        int y = sqrt(ySquare);
        if(y*y == ySquare)
            result += 4;}
    return result;}
# Count Integral points inside a Triangle
# Picks Theeorem:
A = I + (B/2) -1
```

```
# A ==> Area of Polygon
# B ==> Number of integral points on edges of polygon
# I ==> Number of integral points inside the polygon
# Using the above formula, we can deduce,
# I = (2A - B + 2) / 2
# We can find A (area of triangle) using below Shoelace formula.
A = 1/2 * abs(x1(y2 - y3) + x2(y3 - y1) + x3(y1 - y2))
int getBoundaryCount(Point p,Point q){
    if (p.x==q.x)
        return abs(p.y - q.y) - 1;
    if (p.y == q.y)
        return abs(p.x-q.x) - 1;
    return gcd(abs(p.x-q.x),abs(p.y-q.y))-1;}
// Returns count of points inside the triangle
int getInternalCount(Point p, Point q, Point r){
    int BoundaryPoints = getBoundaryCount(p, q) +
    getBoundaryCount(p, r) + getBoundaryCount(q, r) + 3;
    // 3 extra integer points for the vertices
    int doubleArea = abs(p.x*(q.y - r.y) + q.x*(r.y - p.y) + r.x*(p.y - q.y));
    return (doubleArea - BoundaryPoints + 2)/2;}
# Count of parallelograms in a plane
int countOfParallelograms(int x[], int y[], int N){
    map<pair<int, int>, int> cnt; // Map to store frequency of mid points of diagonal
    for (int i=0; i<N; i++){
        for (int j=i+1; j<N; j++){
            int midX = x[i] + x[j];
            int midY = y[i] + y[j];
            cnt[make_pair(midX, midY)]++;}}
    int res = 0;
    for (auto it = cnt.begin(); it != cnt.end(); it++){
        int freq = it->second;
        res += freq*(freq - 1)/2}
    return res;}
# Distance between a point and a Plane in 3D
Distance = (| a*x1 + b*y1 + c*z1 + d |) / (sqrt( a*a + b*b + c*c))
void shortest_distance(float x1, float y1, float z1,
                       float a, float b, float c, float d){
    float f = fabs((a * x1 + b * y1 + c * z1 + d));
    float e = sqrt(a * a + b * b + c * c);
    printf("Perpendicular distance is %f", f/e);}
# Distance between two parallel Planes in 3-D
void distance(float a1, float b1, float c1, float d1,
              float a2, float b2, float c2, float d2){
    float x1,y1,z1,d;
    if (a1 / a2 == b1 / b2 && b1 / b2 == c1 / c2){
        x1 = y1 = 0;
        z1 =-d1 / c1;
        d = fabs(( c2 * z1 + d2)) / (sqrt(a2 * a2 + b2 * b2 + c2 * c2));
        printf("Perpendicular distance is %f\n", d);}
    else
        printf("Planes are not parallel");}
# Direction of a Point from a Line Segment
int directionOfPoint(point A, point B, point P)
{
    int RIGHT = 1, LEFT = -1, ZERO = 0;
```

```
    #subtracting co-ordinates of point A from B and P, to make A as origin
    B.x -= A.x; B.y -= A.y;
    P.x -= A.x; P.y -= A.y;
    int cross_product = B.x * P.y - B.y * P.x;
    if (cross_product > 0)
        return RIGHT;
    if (cross_product < 0)
        return LEFT;
    return ZERO; // return ZERO if cross product is zero.}

# closest pair
int compareX(const void* a, const void* b)  {
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);}

int compareY(const void* a, const void* b)  {
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);}

float dist(Point p1, Point p2)  {
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y));}
float bruteForce(Point P[], int n)  {
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;}
float min(float x, float y)  {
    return (x < y)? x : y;}
float stripClosest(Point strip[], int size, float d)  {
    float min = d; // Initialize the minimum distance as d
    qsort(strip, size, sizeof(Point), compareY);
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);
    return min;}
float closestUtil(Point P[], int n){
    if (n <= 3)
        return bruteForce(P, n);
      int mid = n/2;
    Point midPoint = P[mid];
    float dl = closestUtil(P, mid);
    float dr = closestUtil(P + mid, n - mid);
      float d = min(dl, dr);
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;
    return min(d, stripClosest(strip, j, d) );}
float closest(Point P[], int n)  {
    qsort(P, n, sizeof(Point), compareX);
    qsort(P, n, sizeof(Point), compareX);
    return closestUtil(P, n);}
# Determinant
void getCofactor(int mat[N][N], int temp[N][N], int p, int q, int n){
    int i = 0, j = 0;
    for (int row = 0; row < n; row++){
        for (int col = 0; col < n; col++){
            if (row != p && col != q){
                temp[i][j++] = mat[row][col];
                if (j == n - 1){
                    j = 0;
                    i++; }}}}}
int determinantOfMatrix(int mat[N][N], int n){
    int D = 0; // Initialize result
    if (n == 1)
        return mat[0][0];
    int temp[N][N]; // To store cofactors
    int sign = 1; // To store sign multiplier
    for (int f = 0; f < n; f++){
        getCofactor(mat, temp, 0, f, n);
        D += sign * mat[0][f] * determinantOfMatrix(temp, n - 1);
        sign = -sign;} return D;}
```

```
        assert(cross(b,c) != 0); // no circumcircle if A,B,C aligned
        return a + perp(b*sq(c) - c*sq(b))/cross(b,c)/2; }
    int circleLine(pt o, double r, line l, pair<pt,pt> &out) {
        double h2 = r*r - l.sqDist(o);
        if (h2 >= 0) { // the line touches the circle
            pt p = l.proj(o); // point P
            pt h = l.v*sqrt(h2)/abs(l.v); // vector parallel to l, of length h
            out = {p-h, p+h};}
        return 1 + sgn(h2);}
    int circleCircle(pt o1, double r1, pt o2, double r2, pair<pt,pt> &out) {
        pt d=o2-o1; double d2=sq(d);
        if (d2 == 0) {assert(r1 != r2); return 0;} // concentric circles
        double pd = (d2 + r1*r1 - r2*r2)/2; // = |O_1P| * d
        double h2 = r1*r1 - pd*pd/d2; // = h^2
        if (h2 >= 0) {
            pt p = o1 + d*pd/d2, h = perp(d)*sqrt(h2/d2);
            out = {p-h, p+h};}
        return 1 + sgn(h2);}
    int tangents(pt o1, double r1, pt o2, double r2, bool inner, vector<pair<pt,pt>> &out) {
        if (inner) r2 = -r2;
        pt d = o2-o1;
        double dr = r1-r2, d2 = sq(d), h2 = d2-dr*dr;
        if (d2 == 0 || h2 < 0) {assert(h2 != 0); return 0;}
        for (double sign : {-1,1}) {
            pt v = (d*dr + perp(d)*sqrt(h2)*sign)/d2;
            out.push_back({o1 + v*r1, o2 + v*r2});}
        return 1 + (h2 > 0);}};

#Extended Euclidean

tuple<int,int,int> gcd(int a, int b) {
    if (b == 0) {
        return {1,0,a};
    }
    else {
        int x,y,g;
        tie(x,y,g) = gcd(b,a%b);
        return {y,x-(a/b)*y,g};
    }
}

# x n mod m = x n mod (m-1) mod m, if m is prime and n is very large
```

Line Representation

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef double T;
typedef complex<T> pt;
#define x real()
#define y imag()
pt translate(pt v, pt p) {return p+v;}
pt scale(pt c, double factor, pt p) {return c + (p-c)*factor;}
pt rot(pt p, double a) {
    return {p.x*cos(a) - p.y*sin(a), p.x*sin(a) + p.y*cos(a)};}
pt rotation(pt p, double a) {return p * polar(1.0, a);}
pt perp(pt p) {return {-p.y, p.x};}
T dot(pt v, pt w) {return v.x*w.x + v.y*w.y;}
bool isPerp(pt v, pt w) {return dot(v,w) == 0;}
double angle(pt v, pt w) {
    double cosTheta = dot(v,w) / abs(v) / abs(w);
    return acos(max(-1.0, min(1.0, cosTheta)));}
T cross(pt v, pt w) {return v.x*w.y - v.y*w.x;}
T orient(pt a, pt b, pt c) {return cross(b-a,c-a);}
bool inAngle(pt a, pt b, pt c, pt p) {
    assert(orient(a,b,c) != 0);
    if (orient(a,b,c) < 0) swap(b,c);
    return orient(a,b,p) >= 0 && orient(a,c,p) <= 0;}
double orientedAngle(pt a, pt b, pt c) {
    if (orient(a,b,c) >= 0)
        return angle(b-a, c-a);
    else
        return 2*M_PI - angle(b-a, c-a);}
bool isConvex(vector<pt> p) {
    bool hasPos=false, hasNeg=false;
    for (int i=0, n=p.size(); i<n; i++) {
        int o = orient(p[i], p[(i+1)%n], p[(i+2)%n]);
        if (o > 0) hasPos = true;
        if (o < 0) hasNeg = true;}
    return !(hasPos && hasNeg);}
struct line{
    pt v;
    T c;
    line(pt v, T c) : v(v), c(c) {} // From direction vector v and offset c
    line(T a, T b, T c) : v({b,-a}), c(c) {} // From equation ax+by=c
    line(pt p, pt q) : v(q-p), c(cross(v,p)) {} // From points P and Q
T side(pt p) {return cross(v,p)-c;}
double dist(pt p) {return abs(side(p)) / abs(v);}
double sqDist(pt p) {return side(p)*side(p) / (double)sq(v);}
line perpThrough(pt p) {return {p, p + perp(v)};}
line translate(pt t) {return {v, c + cross(v,t)};}
line shiftLeft(double dist) {return {v, c + dist*abs(v)};}
bool inter(line l1, line l2, pt &out) {
    T d = cross(l1.v, l2.v);
    if (d == 0) return false;
    out = (l2.v*l1.c - l1.v*l2.c) / d; // requires floating-point coordinat
    return true;}
pt proj(pt p) {return p - perp(v)*side(p)/sq(v);}
pt refl(pt p) {return p - perp(v)*2*side(p)/sq(v);}
bool inDisk(pt a, pt b, pt p) {return dot(a-p, b-p) <= 0;}
bool inDisk(pt a, pt b, pt p) {return dot(a-p, b-p) <= 0;}
bool onSegment(pt a, pt b, pt p) {
    return orient(a,b,p) == 0 && inDisk(a,b,p);}
bool properInter(pt a, pt b, pt c, pt d, pt &out) {
    double oa = orient(c,d,a),
    ob = orient(c,d,b),
    oc = orient(a,b,c),
    od = orient(a,b,d);
    // Proper intersection exists iff opposite signs
    if (oa*ob < 0 && oc*od < 0) {
        out = (a*ob - b*oa) / (ob-oa);
        return true;}
return false;}
bool cmpProj(pt p, pt q) {return dot(v,p) < dot(v,q);}
double segPoint(pt a, pt b, pt p) {
    if (a != b) {
        line l(a,b);
        if (l.cmpProj(a,p) && l.cmpProj(p,b)) // if closest to projection
        return l.dist(p); // output distance to line
    }
    return min(abs(p-a), abs(p-b)); // otherwise distance to A or B
}
double segSeg(pt a, pt b, pt c, pt d) {
    pt dummy;
    if (properInter(a,b,c,d,dummy))
        return 0;
    return min({segPoint(a,b,c), segPoint(a,b,d),
        segPoint(c,d,a), segPoint(c,d,b)});}
double areaTriangle(pt a, pt b, pt c) {return abs(cross(b-a, c-a)) / 2.0;}
double areaPolygon(vector<pt> p) {
    double area = 0.0;
    for (int i = 0, n = p.size(); i < n; i++) {
        area += cross(p[i], p[(i+1)%n]); // wrap back to 0 if i == n-1
    }
    return abs(area) / 2.0;}
// true if P at least as high as A (blue part)
bool above(pt a, pt p) {return p.y >= a.y;}
// check if [PQ] crosses ray from A
bool crossesRay(pt a, pt p, pt q) {return (above(a,q) - above(a,p)) * orient(a,p,q) > 0;}
// if strict, returns false when A is on the boundary
bool inPolygon(vector<pt> p, pt a, bool strict = true) {
    int numCrossings = 0;
    for (int i = 0, n = p.size(); i < n; i++) {
        if (onSegment(p[i], p[(i+1)%n], a))
            return !strict;
        numCrossings += crossesRay(a, p[i], p[(i+1)%n]);
    }
    return numCrossings & 1; // inside if odd number of crossings
}
pt circumCenter(pt a, pt b, pt c) {
    b = b-a, c = c-a; // consider coordinates relative to A
    assert(cross(b,c) != 0); // no circumcircle if A,B,C aligned
    return a + perp(b*sq(c) - c*sq(b))/cross(b,c)/2; }
int circleLine(pt o, double r, line l, pair<pt,pt> &out) {
    double h2 = r*r - l.sqDist(o);
```

## Number Theory

```cpp
#include<bits/stdc++.h>
using namespace std;
long powmod(long base, long exp, long modulus) {
    base %= modulus;
    long result = 1;
        while (exp > 0) {
        if (exp & 1) result = (result * base) % modulus;
        base = (base * base) % modulus;
        exp >>= 1;}
    return result;}
// returns g = gcd(a, b); finds x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y){
    int xx = y = 0;
    int yy = x = 1;
    while (b){
        int q = a / b; t = b;
        b = a%b; a = t;
        t = xx; xx = x - q*xx;
        x = t; t = yy;
        yy = y - q*yy;
        y = t;}
    return a;}
// finds all solutions to ax = b (mod n)
vector modular_linear_equation_solver(int a, int b, int n)
{
    int x, y;
    vector ret;
    int g = extended_euclid(a, n, x, y);
    if (!(b%g)){
        x = mod(x*(b / g), n);
        for (int i = 0; i < g; i++)
            ret.push_back(mod(x + i*(n / g), n));}
    return ret;}
// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n){
    int x, y;
    int g = extended_euclid(a, n, x, y);
    if (g > 1) return -1;
    return mod(x, n);}
// Chinese remainder theorem (special case):
PII chinese_remainder_theorem(int m1, int r1, int m2, int r2)
{
    int s, t;
    int g = extended_euclid(m1, m2, s, t);
    if (r1%g != r2%g) return make_pair(0, -1);
    return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);}
PII chinese_remainder_theorem(const VI &m, const VI &r){
    PII ret = make_pair(r[0], m[0]);
    for (int i = 1; i < m.size(); i++){
        ret = chinese_remainder_theorem(ret.second, ret.first, m[i],r[i]);
        if (ret.second == -1) break;}
    return ret;}
// computes x and y such that ax + by = c  returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y){
    if (!a && !b){
        if (c) return false;
        x = 0;
        y = 0;
        return true;}
    if (!a){
        if (c % b) return false;
        x = 0;
        y = c / b;
        return true;}
    if (!b){
        if (c % a) return false;
        x = c / a;
        y = 0;
        return true;}
    int g = gcd(a, b);
    if (c % g) return false;
    x = c / g * mod_inverse(a / g, b / g);
    y = (c - a*x) / b;
    return true;}
// euler's totient
// the positive integers less than or equal to n that are relatively prime to n.
int phi (int n){
    int result = n;
    for (int i=2; i*i<=n; ++i)
        if(n %i==0){
            while(n %i==0)
                n /= i;
            result -= result / i;}
    if (n > 1)
    result -= result / n;
    return result;}
# number of factors p(n) = product of(ai + 1), ai is exponent of factorization
# sum of factors s(n) = product of (pi^(ai + 1) - 1)/(pi - 1)
# product of factors u(n) = n^p(n)
# euler's totient t(n) = product of((pi^(ai - 1))*(pi - 1))
# x^t(m) mod m = 1
# mod inverse x^-1 = x^(t(m) - 1)
# if m is prime mod inverse is x^-1 = x^(m-2)
# diophantine equation first solution is (x, y) then all pairs are
# (x + kb/gcd(a,b), y - ka/gcd(a,b))
# chineese reminder xk = m1*m2*...*mn/mk
# x = a1*x1*inversemod(x1, m1) + ... + an*xn*inversemod(xn, mn)
```