- Make a triangle

```
int dp[101][101];
//c <= a+b-1
int solve(int a, int b, int c){
	if (dp[a][b] != -1) return dp[a][b];

	int condition = (c <= a+b-1);
	if (condition)
		return 0;

	int A = 105, B = 105;
	if (a+1 <= 100)
		A = 1 + solve(a+1, b, c);
	if (b+1 <= 100)
		B = 1 + solve(a, b+1, c);

	return dp[a][b] = min(A,B);
}

int main(){
	ios::sync_with_stdio(false), cin.tie(0);
	memset(dp, -1, sizeof(dp));

	int a,b,c;
	cin >> a >> b >> c;

	if (c < a) swap(c,a);
	if (c < b) swap(c,b);
	cout << solve(a,b,c) << endl;

	return 0;
}
```

- Vasya and triangle

```
ll gcd(ll a, ll b){
	while(b) a %= b, swap(a,b);
	return a;
}
int main(){
	ios::sync_with_stdio(false), cin.tie(0);
	ll n, m, k; cin >> n >> m >> k;
	//If rectangle formed by the triangle is not an integer, then its impossible
	ll K = k;
```

```
	if (K%2==0) k=k/2;
	if ((m*n)%k != 0){
		cout << "NO\n";
		return 0;
	}
	ll g = gcd(k, n); k = k/g;
	ll a = n/g;

	g = gcd(k, m); k = k/g;
	ll b = m/g;

	if (K%2!=0) (a<n) ? a *= 2 : b *= 2;
	//printf("YES\n%d %d\n%d %d\n%d %d\n", 0, 0, a, 0, 0, b);
	if ((a*b)/2 != (m*n)/K){
		if(a%2==0) a/=2;
		else if(b%2==0) b/=2;
	}
	//printf("YES\n%d %d\n%d %d\n%d %d\n", 0, 0, a, 0, 0, b);
	cout << "YES\n" << "0 0\n" << a << " 0\n" << "0 " << b << endl;

	return 0;
}
```

- Minimum Bounding Rectangle

```
int main(){
	ios::sync_with_stdio(false), cin.tie(0);
	int T; cin >> T;
	while(T--){
		int q; cin >> q;
		int lx = INT_MAX, ly = INT_MAX, rx = INT_MIN, ry = INT_MIN;
		while(q--){
			char c; cin >> c;
			if (c=='p'){
				int x, y; cin >> x >> y;
				lx = min(lx, x); ly = min(ly, y);
				rx = max(rx, x); ry = max(ry, y);
			}else if(c=='l'){
				int x1, y1, x2, y2; cin >> x1 >> y1 >> x2 >> y2;
				lx = min(lx, x1); ly = min(ly, y1);
				lx = min(lx, x2); ly = min(ly, y2);
				rx = max(rx, x1); ry = max(ry, y1);
				rx = max(rx, x2); ry = max(ry, y2);
			}else{
				int x, y, r; cin >> x >> y >> r;
```

```
                        lx = min(lx, x-r); ly = min(ly, y-r);
                        lx = min(lx, x+r); ly = min(ly, y+r);
                        rx = max(rx, x-r); ry = max(ry, y-r);
                        rx = max(rx, x+r); ry = max(ry, y+r);
                    }
                }
                printf("%d %d %d %d\n", lx, ly, rx, ry);
        }
        return 0;
}
```

- Determine the shape

```
int main(){
        ios::sync_with_stdio(false), cin.tie(0);
        int q; cin >> q;
        for (int caso = 1; caso <= q; caso++){
                string tipe = "";
                PT A, B, C, D;
                double x,y;
                cin >> x >> y;
                A.x = x; A.y = y;
                cin >> x >> y;
                B.x = x; B.y = y;
                cin >> x >> y;
                C.x = x; C.y = y;
                cin >> x >> y;
                D.x = x; D.y = y;

                //Centroide
                PT CE = A;
                CE.x += B.x; CE.y += B.y;
                CE.x += C.x; CE.y += C.y;
                CE.x += D.x; CE.y += D.y;
                CE.x /= 4; CE.y /= 4;
                PT CEA = CE-A, CEB = CE-B, CEC = CE-C, CED = CE-D;
                vector<pair<double, PT> > pontos;
                pontos.push_back({angle(CEA), A});
                pontos.push_back({angle(CEB), B});
                pontos.push_back({angle(CEC), C});
                pontos.push_back({angle(CED), D});
                sort(pontos.begin(), pontos.end());

                A = pontos[0].second;
                B = pontos[1].second;
```

```
C = pontos[2].second;
D = pontos[3].second;
PT vec1 = A-B, vec2 = B-C, vec3 = C-D, vec4 = D-A;
//vec1 = getDir(A,B); vec2 = getDir(B,C); getDir(C,D); getDir(D,A);
double distAB = sqrt(vec1.x*vec1.x + vec1.y*vec1.y),
            distBC = sqrt(vec2.x*vec2.x + vec2.y*vec2.y),
            distCD = sqrt(vec3.x*vec3.x + vec3.y*vec3.y),
            distDA = sqrt(vec4.x*vec4.x + vec4.y*vec4.y);
//cout << distAB << " " << distBC << " " << distCD << " " << distDA << "\n";
if ((distAB==distCD) && (distBC==distDA)){
        double esc1 = abs((vec1.x*vec2.x) + (vec1.y*vec2.y));
        double esc2 = abs((vec3.x*vec4.x) + (vec3.y*vec4.y));
        double esc3 = abs((vec2.x*vec3.x) + (vec2.y*vec3.y));
        //cout << esc1 << " " << esc2 << " " << esc3 << endl;
        if (distAB==distDA){//distCD-distBC <= eps){
                //square or Rhombus
                //if all 90º -> square, else -> Rhombus
                //tipe = "Square or Rhombus";
                if (!esc1 && !esc2 && !esc3)
                        tipe = "Square";
                else
                        tipe = "Rhombus";
        }else{
                //rectangle or paralelogram
                //if all 90º -> rectangle, else -> paralelogram
                //tipe = "Rectangle or Parallelogram";
                if (!esc1 && !esc2 && !esc3)
                        tipe = "Rectangle";
                else
                        tipe = "Parallelogram";

        }
}
//Need to check if Trapezium
if (tipe==""){
        bool paral1, paral2;
        //AB e CD, AC e BD, AD e BC
        paral1 = parallel(A,B,C,D);
        paral2 = parallel(A,D,C,B);
        if (paral1 && !paral2) tipe = "Trapezium";
        paral1 = parallel(A,C,B,D);
        paral2 = parallel(A,D,B,C);
        if (paral1 && !paral2) tipe = "Trapezium";
```

```
                paral1 = parallel(A,D,B,C);
                paral2 = parallel(A,C,B,D);
                if (paral1 && !paral2) tipe = "Trapezium";
            }

            if (tipe == "") tipe = "Ordinary Quadrilateral";
            cout <<  "Case " << caso << ": "<< tipe << endl;

        }
        return 0;
}
```

- Triangle fun

```
int main(){
        int q; cin >> q;
        while(q--){
            PT A, B, C;
            double x,y;
            cin >> x >> y;
            A.x = x; A.y = y;
            cin >> x >> y;
            B.x = x; B.y = y;
            cin >> x >> y;
            C.x = x; C.y = y;

            PT D, E, F;
            D = (C-B)/3 + B;
            E = (A-C)/3 + C;
            F = (B-A)/3 + A;

            PT P, Q, R;
            P = computeLineIntersection(A, D, B, E);
            Q = computeLineIntersection(A, D, C, F);
            R = computeLineIntersection(B, E, C, F);
            vector<PT> triangulo;
            triangulo.pb(P);
            triangulo.pb(Q);
            triangulo.pb(R);

      cout << setprecision(0) << fixed << computeArea(triangulo) << endl ;
        }
        return 0;
}
```

- Pair of Line

```
int main(){
	ios::sync_with_stdio(false), cin.tie(0);
	int n; cin >> n;
	if (n <= 4){
		cout << "YES\n";
		return 0;
	}
	vector<PT> pts;
	while(n--){
		int x,y; cin >> x >> y;
		PT aux; aux.x = x; aux.y = y;
		pts.push_back(aux);
	}
	//collinear (PT a, PT b, PT c, PT d)
	vector<ii > pares = {{0,1}, {0,2}, {1,2}};
	for (int i = 0; i < 3; i++){
		//cout << ((collinear(a, b, pts[i], pts[i])) ? "YES" : "NO") << endl;
		PT a = pts[pares[i].a]; PT b = pts[pares[i].b];

		vector<PT> naoCol;
		for (int j = 0; j < int(pts.size()); j++){
			if (!collinear(a, b, pts[j], pts[j])){
				naoCol.pb(pts[j]);
			}
		}
		if (naoCol.size() <= 2) {cout << "YES\n"; return 0;}
		PT aa = naoCol[0], bb = naoCol[1];
		int count = 0;
		for (auto p : naoCol){
			if (!collinear(aa, bb, p, p)){
				count++;
			}
		}
		if (!count){
			cout << "YES\n"; return 0;
		}
	}
	cout << "NO\n";
	return 0;
}
```

Library de gap

```
const double inf = 1e100, eps = 0.0000000001;//1e-9;
const double PI = acos(-1.0L);

int cmp (double a, double b = 0) {
  if (abs(a-b) < eps) return 0;
  return (a < b) ? -1 : +1;
}

struct PT {
  double x, y;
  PT(double x = 0, double y = 0) : x(x), y(y) {}
  PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
  PT operator * (double c) const { return PT(x*c, y*c); }
  PT operator / (double c) const { return PT(x/c, y/c); }

  bool operator <(const PT &p) const {
    if(cmp(x, p.x) != 0) return x < p.x;
    return cmp(y, p.y) < 0;
  }
  bool operator ==(const PT &p) const {
    return !cmp(x, p.x) && !cmp(y, p.y);
  }
  bool operator != (const PT &p) const {
    return !(p == *this);
  }
};
double dot (PT p, PT q) { return p.x * q.x + p.y*q.y; }
double cross (PT p, PT q) { return p.x * q.y - p.y*q.x; }
double dist2 (PT p, PT q = PT(0, 0)) { return dot(p-q, p-q); }
double dist (PT p, PT q) { return hypot(p.x-q.x, p.y-q.y); }
double norm (PT p) { return hypot(p.x, p.y); }
PT normalize (PT p) { return p/hypot(p.x, p.y); }
double angle (PT p, PT q) { return atan2(cross(p, q), dot(p, q)); }
double angle (PT p) { double a = atan2(p.y, p.x); if (a<0) a+= 2*PI; return a; }
double polarAngle (PT p) {
  double a = atan2(p.y,p.x);
  return a < 0 ? a + 2*PI : a;
}

// - p.y*sen(+90), p.x*sen(+90)
PT rotateCCW90 (PT p) { return PT(-p.y, p.x); }
```

```
// - p.y*sen(-90), p.x*sen(-90)
PT rotateCW90 (PT p) { return PT(p.y, -p.x); }

PT rotateCCW (PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// !!! PT (int, int)
typedef pair<PT, int> Line;
PT getDir (PT a, PT b) {
  if (a.x == b.x) return PT(0, 1);
  if (a.y == b.y) return PT(1, 0);
  int dx = b.x-a.x;
  int dy = b.y-a.y;
  int g = __gcd(abs(dx), abs(dy));
  if (dx < 0) g = -g;
  return PT(dx/g, dy/g);
}

Line getLine (PT a, PT b) {
  PT dir = getDir(a, b);
  return {dir, cross(dir, a)};
}
// Projeta ponto c na linha a - b assumindo a != b
// a.b = |a| cost * |b|
PT projectPointLine (PT a, PT b, PT c) {
  return a + (b-a) * dot(b-a, c-a)/dot(b-a, b-a);
}

PT reflectPointLine (PT a, PT b, PT c) {
  PT p = projectPointLine(a, b, c);
  return p*2 - c;
}

// Projeta ponto c no segmento a - b
PT projectPointSegment (PT a, PT b, PT c) {
  double r = dot(b-a, b-a);
  if (cmp(r) == 0) return a;
  r = dot(b-a, c-a)/r;
  if (cmp(r, 0) < 0) return a;
  if (cmp(r, 1) > 0) return b;
  return a + (b - a) * r;
}
```

```
// Calcula distancia entre o ponto c e o segmento a - b
double distancePointSegment (PT a, PT b, PT c) {
  return dist(c, projectPointSegment(a, b, c));
}

// Parallel and opposite directions
// Determina se o ponto c esta em um segmento a - b
bool ptInSegment (PT a, PT b, PT c) {
  if (a == b) return a == c;
  a = a-c, b = b-c;
  return cmp(cross(a, b)) == 0 && cmp(dot(a, b)) <= 0;
}

// Determina se as linhas a - b e c - d sao paralelas ou colineares
bool parallel (PT a, PT b, PT c, PT d) {
  return cmp(cross(b - a, c - d)) == 0;
}

bool collinear (PT a, PT b, PT c, PT d) {
  return parallel(a, b, c, d) && cmp(cross(a - b, a - c)) == 0 && cmp(cross(c - d, c - a)) == 0;
}

// Calcula distancia entre o ponto (x, y, z) e o plano ax + by + cz = d
double distancePointPlane(double x, double y, double z, double a, double b, double c, double d)
{
    return abs(a * x + b * y + c * z - d) / sqrt(a * a + b * b + c * c);
}

// Determina se o segmento a - b intersecta com o segmento c - d
bool segmentsIntersect (PT a, PT b, PT c, PT d) {
  if (collinear(a, b, c, d)) {
    if (cmp(dist(a, c)) == 0 || cmp(dist(a, d)) == 0 || cmp(dist(b, c)) == 0 || cmp(dist(b, d)) == 0)
return true;
    if (cmp(dot(c - a, c - b)) > 0 && cmp(dot(d - a, d - b)) > 0 && cmp(dot(c - b, d - b)) > 0) return
false;
    return true;
  }
  if (cmp(cross(d - a, b - a) * cross(c - a, b - a)) > 0) return false;
  if (cmp(cross(a - c, d - c) * cross(b - c, d - c)) > 0) return false;
  return true;
}
```

```
// Calcula a intersecao entre as retas a - b e c - d assumindo que uma unica intersecao existe
// Para intersecao de segmentos, cheque primeiro se os segmentos se intersectam e que nao sao paralelos
// r = a1 + t*d1, (r - a2) x d2 = 0
PT computeLineIntersection (PT a, PT b, PT c, PT d) {
  b = b - a; d = c - d; c = c - a;
  assert(cmp(cross(b, d)) != 0);
  return a + b * cross(c, d) / cross(b, d);
}

// Calcula centro do circulo dado tres pontos
PT computeCircleCenter (PT a, PT b, PT c) {
  b = (a + b) / 2; // bissector
  c = (a + c) / 2; // bissector
  return computeLineIntersection(b, b + rotateCW90(a - b), c, c + rotateCW90(a - c));
}

vector<PT> circle2PtsRad (PT p1, PT p2, double r) {
  vector<PT> ret;
  double d2 = dist2(p1, p2);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return ret;
  double h = sqrt(det);
  for (int i = 0; i < 2; i++) {
    double x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
    double y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
    ret.push_back(PT(x, y));
    swap(p1, p2);
  }
  return ret;
}

// Calcula intersecao da linha a - b com o circulo centrado em c com raio r > 0
bool circleLineIntersection(PT a, PT b, PT c, double r) {
    return cmp(dist(c, projectPointLine(a, b, c)), r) <= 0;
}

vector<PT> circleLine (PT a, PT b, PT c, double r) {
  vector<PT> ret;
  PT p = projectPointLine(a, b, c), p1;
  double h = norm(c-p);
  if (cmp(h,r) == 0) {
    ret.push_back(p);
```

```
  } else if (cmp(h,r) < 0) {
    double k = sqrt(r*r - h*h);
    p1 = p + (b-a)/(norm(b-a))*k;
    ret.push_back(p1);
    p1 = p - (b-a)/(norm(b-a))*k;
    ret.push_back(p1);
  }
  return ret;
}

bool ptInsideTriangle(PT p, PT a, PT b, PT c) {
  if(cross(b-a, c-b) < 0) swap(a, b);
  if(ptInSegment(a,b,p)) return 1;
  if(ptInSegment(b,c,p)) return 1;
  if(ptInSegment(c,a,p)) return 1;
  bool x = cross(b-a, p-b) < 0;
  bool y = cross(c-b, p-c) < 0;
  bool z = cross(a-c, p-a) < 0;
  return x == y && y == z;
}

// Determina se o ponto esta num poligono convexo em O(lgn)
bool pointInConvexPolygon(const vector<PT> &p, PT q) {
  if (p.size() == 1) return p.front() == q;
  int l = 1, r = p.size()-1;
  while(abs(r-l) > 1) {
    int m = (r+l)/2;
    if(cross(p[m]-p[0] , q-p[0]) < 0) r = m;
    else l = m;
  }
  return ptInsideTriangle(q, p[0], p[l], p[r]);
}

// Determina se o ponto esta num poligono possivelmente nao-convexo
// Retorna 1 para pontos estritamente dentro, 0 para pontos estritamente fora do poligno
// e 0 ou 1 para os pontos restantes
// Eh possivel converter num teste exato usando inteiros e tomando cuidado com a divisao
// e entao usar testes exatos para checar se esta na borda do poligno
bool pointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for(int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    if((p[i].y <= q.y && q.y < p[j].y || p[j].y <= q.y && q.y < p[i].y) &&
```

```
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// Determina se o ponto esta na borda do poligno
bool pointOnPolygon(const vector<PT> &p, PT q) {
  for(int i = 0; i < p.size(); i++)
    if(cmp(dist2(projectPointSegment(p[i], p[(i + 1) % p.size()], q), q)) < 0)
      return true;
    return false;
}
// area / semiperimeter
double rIncircle (PT a, PT b, PT c) {
  double ab = norm(a-b), bc = norm(b-c), ca = norm(c-a);
  return abs(cross(b-a, c-a)/(ab+bc+ca));
}

// Calcula intersecao do circulo centrado em a com raio r e o centrado em b com raio R
vector<PT> circleCircle (PT a, double r, PT b, double R) {
  vector<PT> ret;
  double d = norm(a-b);
  if (d > r + R || d + min(r, R) < max(r, R)) return ret;
  double x = (d*d - R*R + r*r) / (2*d); // x = r*cos(R opposite angle)
  double y = sqrt(r*r - x*x);
  PT v = (b - a)/d;
  ret.push_back(a + v*x + rotateCCW90(v)*y);
  if (cmp(y) > 0)
    ret.push_back(a + v*x - rotateCCW90(v)*y);
  return ret;
}

double circularSegArea (double r, double R, double d) {
  double ang = 2 * acos((d*d - R*R + r*r) / (2*d*r)); // cos(R opposite angle) = x/r
  double tri = sin(ang) * r * r;
  double sector = ang * r * r;
  return (sector - tri) / 2;
}

// Calcula a area ou o centroide de um poligono (possivelmente nao-convexo)
// assumindo que as coordenadas estao listada em ordem horaria ou anti-horaria
// O centroide eh equivalente a o centro de massa ou centro de gravidade
```

```
double computeSignedArea (const vector<PT> &p) {
  double area = 0;
  for (int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area/2.0;
}

double computeArea(const vector<PT> &p) {
  return abs(computeSignedArea(p));
}

PT computeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * computeSignedArea(p);
  for(int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
  }
  return c / scale;
}

// Testa se o poligno listada em ordem CW ou CCW eh simples (nenhuma linha se intersecta)
bool isSimple(const vector<PT> &p) {
  for(int i = 0; i < p.size(); i++) {
    for(int k = i + 1; k < p.size(); k++) {
      int j = (i + 1) % p.size();
      int l = (k + 1) % p.size();
      if (i == l || j == k) continue;
      if (segmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

vector< pair<PT, PT> > getTangentSegs (PT c1, double r1, PT c2, double r2) {
  if (r1 < r2) swap(c1, c2), swap(r1, r2);
  vector<pair<PT, PT> > ans;
  double d = dist(c1, c2);
  if (cmp(d) <= 0) return ans;
  double dr = abs(r1 - r2), sr = r1 + r2;
```

```
  if (cmp(dr, d) >= 0) return ans;
  double u = acos(dr / d);
  PT dc1 = normalize(c2 - c1)*r1;
  PT dc2 = normalize(c2 - c1)*r2;
  ans.push_back(make_pair(c1 + rotateCCW(dc1, +u), c2 + rotateCCW(dc2, +u)));
  ans.push_back(make_pair(c1 + rotateCCW(dc1, -u), c2 + rotateCCW(dc2, -u)));
  if (cmp(sr, d) >= 0) return ans;
  double v = acos(sr / d);
  dc2 = normalize(c1 - c2)*r2;
  ans.push_back({c1 + rotateCCW(dc1, +v), c2 + rotateCCW(dc2, +v)});
  ans.push_back({c1 + rotateCCW(dc1, -v), c2 + rotateCCW(dc2, -v)});
  return ans;
}
```