Geometria computacional

November 29, 2019

1 Algunos elementos básicos de geometria computacional.

En general, la resolución de problemas de GC en concursos o entrevistas **no es fácil**:

- Tipicamente, puede haber maneras por del concebidor del problema de meter **casos de prueba patologicos** que a uno se le puede olvidar facil de considerar (colinearidad; concavidad...).
- Los calculos con **punto flotante** pueden llevar a errores, aunque toda la lógica es correcta.
- La implementación es en general fastidiosa (codigo más largo).

Prever tener una **librería de funciones básicas escritas o documentadas**.
Cuando es posible, hacer los caculos en **precisión de punto fijo** (con enteros).
Cuando se trabaja en punto flotante, tener cuidado a la igualdad!

```
if (x==y) { // NO!
if (fabs(x-y)<EPS) { // SI

if (x>=0.0) { // NO!
if (x>-EPS) { // SI
```

En la gran mayoria de los casos $EPS = 10^{-9}$ es más que suficiente.

2 Funciones básicas

- Se necesita mucha disciplina y estructuración del código! Usar **POO tanto como se pueda**.
- Prever clases para puntos, lineas...

```
// Code from S. Halim
// Basic raw form, minimalist mode
struct point_i {
    int x, y;
    // Whenever possible, work with point_i
    point_i() { x = y = 0; }
    // Default constructor
    point_i(int _x, int _y) : x(_x), y(_y) {}
};
```

```
// Code from S. Halim
// User-defined
struct point {
        double x, y;
        // Only used if more precision is needed
        point() { x = y = 0.0; }
        // Default constructor
        point(double _x, double _y) : x(_x), y(_y) {}
        // Copy constructor
        point(const point &p) : x(p.x), y(p.y) {}
};
```

Para facilitar el uso del código, **sobrecargar operadores**:

```
bool operator == (const point &other) const {
    return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); }
};
```

```
#include <cmath>
// Code from S. Halim
// Euclidean distance
double dist(point p1, point p2) {
    return hypot(p1.x - p2.x, p1.y - p2.y);
}
```

2.1 Puntos y lineas...

```
// Code from S. Halim
struct vec {
    double x, y;
    vec(const double &_x, const double &_y) : x(_x), y(_y) {}
};

vec toVec(const point &a, const point &b) {
    // Convert 2 points to vector a->b
    return vec(b.x - a.x, b.y - a.y);
}

vec scale(const vec &v, double s) {
    return vec(v.x * s, v.y * s);
}

point translate(const point &p, const vec &v) {
    // translate p according to v
    return point(p.x + v.x , p.y + v.y);
}
double dot(const vec &a, const vec &b) { return (a.x * b.x + a.y * b.y); }
double norm_sq(const vec &v) { return v.x * v.x + v.y * v.y; }
```

```
// Code from S. Halim
// Returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(const point &p, const point &a, const point &b, point &c) {
    // formula: c = a + u * ab
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    c = translate(a, scale(ab, u)); // translate a to c
    return dist(p, c); // Euclidean distance between p and c
}

// Code from S. Halim
// Returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(const point &p, const point &a, const point &b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    // Computes the relative position of the projection of p on [ab]
    double u = dot(ap, ab) / norm_sq(ab);
    if (u < 0.0) {
        c = point(a.x, a.y); // Closer to a
        return dist(p, a);
    }
    // Euclidean distance between p and a
    if (u > 1.0) {
        c = point(b.x, b.y);
        // closer to b
        return dist(p, b);
    }
    // Euclidean distance between p and b
    return distToLine(p, a, b, c);
}
```

UVA 10263

Railway is a broken line of N segments. The problem is to find such a position for the railway station that the distance from it to the given point M is the minimal.

Input: The input will consist of several input blocks. Each input block begins with two lines with coordinates Xm and Ym of the point M. In the third line there is N — the number of broken line segments. The next 2N + 2 lines contain the X and the Y coordinates of consecutive broken line corners. The input is terminated by .

Output: For each input block there should be two output lines. The first one contains the first coordinate of the station position, the second one contains the second coordinate. Coordinates are the floating-point values with four digits after decimal point.

Trivial con la función de distancia punto/segmento: Recorrer todos los segmentos.

```
int main() {
  point m;
```

```
  int N;
  while (cin >> m.x) {
    cin >> m.y;
    cin >> N;
    point rails[N+1];
    double dmin = std::numeric_limits<double>::max();
    point c,cmin;
    for (int i=0;i<=N;i++) {
      cin >> rails[i].x >> rails[i].y;
      if (i>=1) {
        double d = distToLineSegment(m,rails[i],rails[i-1],c);
        if (d<dmin) {
          dmin=d;
          cmin=c;
        }
      }
    }
    cout << fixed;
    cout << setprecision(4);
    cout << cmin.x << endl;
    cout << cmin.y << endl;
  }
}
```

2.2 Poligonos

Implementación simple como contenedor (ciclico) de puntos:

```
// 6 points, entered in counter clockwise order, 0-based indexing
vector<point> P;
P.push_back(point(1, 1)); // P0
P.push_back(point(3, 3)); // P1
P.push_back(point(9, 1)); // P2
P.push_back(point(12, 4)); // P3
P.push_back(point(9, 7)); // P4
P.push_back(point(1, 7)); // P5
P.push_back(P[0]); // important: loop back
```

UVA 10577

The Archeologists of the Current Millenium (ACM) now and then discover ancient artifacts located at vertices of regular polygons. The moving sand dunes of the desert render the excavations difficult and thus once three vertices of a polygon are discovered there is a need to cover the entire polygon with protective fabric.

Input Input contains multiple cases. Each case describes one polygon. It starts with an integer $n \leq 50$, the number of vertices in the polygon, followed by three pairs of real numbers giving the x and y coordinates of three vertices of the polygon. The numbers are separated by whitespace. The input ends with a n equal 0, this case should not be processed.

Output For each line of input, output one line in the format shown below, giving the smallest area of a rectangle which can cover all the vertices of the polygon and whose sides are parallel to the x and y axes.

- A partir de los 3 puntos, construir el circulo circunscrito (centro y radio).
- Deducir la lista de todos los puntos de poligono regular (porque el ángulo entre dos vertices tiene que ser $\frac{2\pi}{N}$).
- Actualizar los valores extremos de x,y.

```
int main() {
  point a,b,c;
  int N,nTest=1;
  while (cin >> N) {
    if (N==0) break;
    cin >> a.x;
    cin >> a.y;
    cin >> b.x;
    cin >> b.y;
    cin >> c.x;
    cin >> c.y;

    // Middle of [ab]
    point mab   = middle(a,b);
    point mabTr = translate(mab,vec(b.y-a.y,a.x-b.x));
    line medab;
    pointsToLine(mab,mabTr,medab);
    // Middle of [ac]
    point mac   = middle(a,c);
    point macTr = translate(mac,vec(c.y-a.y,a.x-c.x));
    line medac;
    pointsToLine(mac,macTr,medac);
    // Center as intersection of bisectors
    point center;
    areIntersect(medab,medac,center);
    // Radius
    double radius = dist(center,a);
    double xmin = std::numeric_limits<double>::max();
    double ymin = std::numeric_limits<double>::max();
    double xmax =-std::numeric_limits<double>::max();
    double ymax =-std::numeric_limits<double>::max();
    double t    = atan2(a.y-center.y,a.x-center.x);
    for (int i=0;i<N;i++) {
      double x = center.x + radius*cos(t+i*M_PI*2.0/N);
      if (x<xmin) xmin = x;
      if (x>xmax) xmax = x;
      double y = center.y + radius*sin(t+i*M_PI*2.0/N);
      if (y<ymin) ymin = y;
      if (y>ymax) ymax = y;
```

```
    }

    cout << fixed;
    cout << setprecision(3);
    cout << "Polygon " << nTest++ << ": " << (xmax-xmin)*(ymax-ymin) << endl;
  }
}
```

2.3 Area de un poligono

Para calcular el area de un poligno (convexo o concavo), con los puntos dados en un orden u otro; utilizar el determinante de la matriz siguiente:

$$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-1} & y_{n-1} \end{bmatrix}$$

$$A = \frac{1}{2}(x_0y_1 + x_1y_2 + \dots + x_{n-1}y_0 - x_1y_0 - x_2y_1 - \dots - x_0y_{n-1})$$

```
// returns the area, which is half the determinant
double area(const vector<point> &P) {
    double result = 0.0, x1, y1, x2, y2;
    for (int i = 0; i < (int)P.size()-1; i++) {
        x1 = P[i].x; x2 = P[i+1].x;
        y1 = P[i].y; y2 = P[i+1].y;
        result += (x1 * y2 - x2 * y1);
    }
    return fabs(result) / 2.0;
}
```

UVA 10060

How can a manhole be a hole if it is covered? Perhaps, to prove a manhole a hole, most of the manholes of Dhaka are uncovered. So now manhole means a hole to catch a man. Anyway, the new Mayor of Dhaka does not like this definition and he has recently been highly acclaimed by general people for ordering corresponding department to cover all the manholes of the city within a month.

Manhole Cover Manufacturing Corporation (MCMC) somehow managed to get the order. (Yes, this is a big deal, since a lot of manhole covers are to be made). MCMC makes the cover using steel, and they import polygonal steel sheets of different shapes and thickness from abroad. Then they melt the sheets to make the circular manhole covers, which also differ in size and thickness. MCMC needs a program which, given dimensions of a number of steel sheets, will calculate how many manhole cover can be made from these sheets. You are to help them by writing the program.

Input: The input file consists of several data blocks. Each data block starts with an integer N, the number of polygonal steel sheets. i-th line of the next N lines starts with thickness of the i-th sheet followed by co-ordinates of the polygon's corner points in some order (clockwise or anti-clockwise). Each line consists of a series of real numbers in following format:

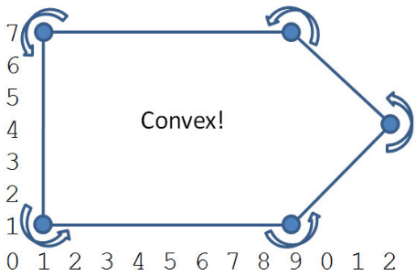
TiX0Y0X1Y1X2Y2 . . . XnYnX0Y0

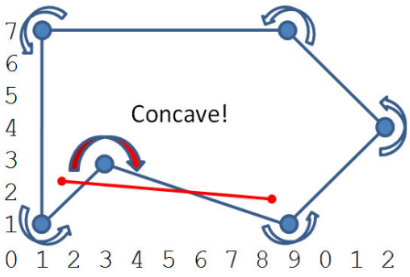
Where Ti is the thickness of the sheet, and XiYi are the coordinates of corner points. The line ends with co-ordinate of the first point. Last line of each data block will have two real numbers, R and T, radius and thickness of the manhole cover respectively. Input file ends with a data block with N = 0.

Output: For each data block, print the number of manhole cover in separate line.

- Con la función de **cálculo del area** del poligono, es facil.
- Elemento tecnico: los poligonos no necesariamente vienen en una sola linea (mejor detectar el final del input de un poligono por detectar el primer punto).

```
// returns the area, which is half the determinant
double area(const vector<point> &P) {
    double result = 0.0, x1, y1, x2, y2;
    for (int i = 0; i < (int)P.size()-1; i++) {
        x1 = P[i].x; x2 = P[i+1].x;
        y1 = P[i].y; y2 = P[i+1].y;
        result += (x1 * y2 - x2 * y1);
    }
    return fabs(result) / 2.0;
}
int main() {
  int N;
  while (cin >> N) {
    if (N==0) break;
    string s;
    vector<point> sheets[N];
    double thickness[N];
    double volume = 0.0;
    for (int i=0;i<N;i++) {
      cin >> thickness[i];
      point p;
      cin >> p.x;
      cin >> p.y;
      sheets[i].push_back(p);
      while (1) {
        cin >> p.x;
        cin >> p.y;
        sheets[i].push_back(p);
        if (p==sheets[i][0]) break;
      }
      volume += thickness[i]*area(sheets[i]);
    }
    // Holes
    double r,th;
    cin >> r;
    cin >> th;
    double volumeHole = th*M_PI*r*r;
```






```
    cout << fixed;
    cout << setprecision(3);
    cout << (int)(volume/volumeHole) << endl;
  }
}
```

2.4 Test de convexidad

Un poligono es **convexo** cuando, para cualquier par de puntos $\mathbf{p}, \mathbf{q}$ ubicados dentro del poligno, todos los puntos del segmento $[\mathbf{p}, \mathbf{q}]$ están también incluidos en el poligono.

Ahora, para comprobar convexidad, es imposible implementar tal cual esta definición y verificar **todos** los segmentos con vertices incluidos en el poligono.

- Podemos simplemente verificar si todos los grupos de **3 vertices consecutivos** del poligono (que corresponderán a un **recorrido en sentido horario o anti-horario**) giran siempre en el mismo sentido (el horario o el anti-horario).
- Si hay un par de tripletas que tienen sentido distinto, el poligono **no es convexo**.
- Test de orientación: $O(1)$.
- Complejidad total **lineal**.

- Un elemento fundamental es evaluar si tres puntos $\mathbf{p}, \mathbf{q}, \mathbf{r}$ consecutivos vienen **en sentido horario o anti-horario**.
- Para determinarlo: considerar el **producto cruz** de los vectores (planares) $\overrightarrow{\mathbf{pq}}$ y $\overrightarrow{\mathbf{pr}}$ (a los cuales les agregamos una tercera coordenada nula):

$$\overrightarrow{\mathbf{pq}} \times \overrightarrow{\mathbf{pr}} = \overrightarrow{\mathbf{pq}} \times \overrightarrow{\mathbf{qr}} = -\overrightarrow{\mathbf{qp}} \times \overrightarrow{\mathbf{qr}}$$

```
// Code from S. Halim's book
// Cross-product
double cross(const vec &a, const vec &b) {
    return a.x * b.y - a.y * b.x;
}
// Returns true if point r is on the left side of line pq
```

```cpp
bool ccw(const point &p, const point &q, const point &r) {
    return cross(toVec(p, q), toVec(p, r)) > 0;
}

// Code from S. Halim's book
bool isConvex(const vector<point> &P) {
    // returns true if all three
    int sz = (int)P.size(); // consecutive vertices of P form the same turns
    if (sz <= 3) return false; // Remember that we use a representation with duplicate first/las
    // a point/sz=2 or a line/sz=3 is not convex

    bool isLeft = ccw(P[0], P[1], P[2]); // Take the first result
    for (int i = 1; i < sz-1; i++)
        // then compare with the others
        if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
            return false;
    // different sign -> this polygon is concave
    return true;
} // this polygon is convex
```

Ojo:

- En varios problemas, se podrá dar poligonos con **grupos de vértices colineales**.
- Seguirá funcionando el algoritmo?
- Como manejarlo?

- Modificar la función de **test de orientación**.
- Usar una **tolerancia con respecto a un valor de referencia** (si la tripleta estaba orientada positivamente, tolerar hasta -EPS).

```cpp
// Returns true if point r is on the left side of line pq
bool ccw(const point &p, const point &q, const point &r, bool ref) {
    return ref?(cross(toVec(p, q), toVec(p, r))>-EPS):(cross(toVec(p, q), toVec(p, r))<EPS);
}
```

2.5 Test de inclusion en un poligono

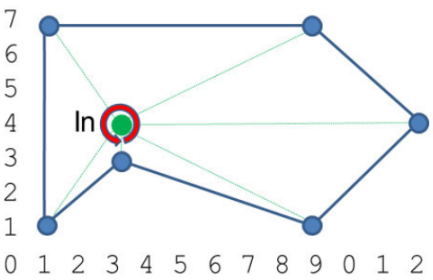
Un test muy común es verificar si un punto **p** está incluido (o no) dentro de un polígono **P**.

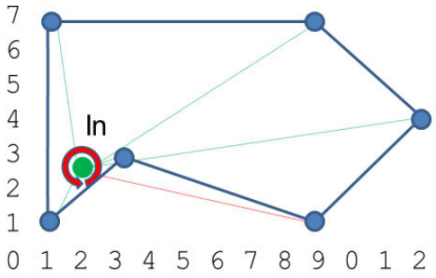
2.5.1 Algoritmo del 'winding number'

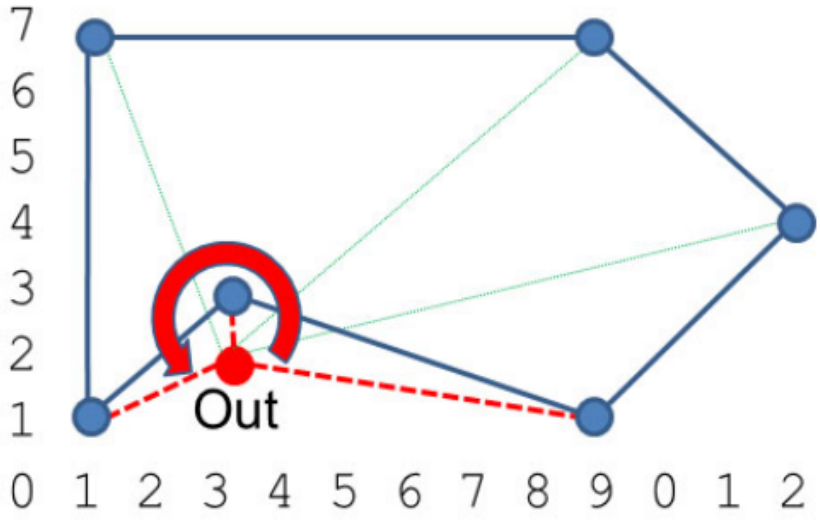
- Funciona para cualquier tipo de poligono simple (convexo o concavo).
- Consiste en calcular la suma de angulos:

$$\mathbf{P}[i]\widehat{\mathbf{pP}[i+1]}$$

  donde $\mathbf{P}[i], \mathbf{P}[i+1]$ son los puntos consecutivos del poligono **P**.
- Contar los ángulos **con signo** (positivos cuando el tercer punto es a la izquierda del vector formado por los dos primeros, negativos sino).


7
6
5
4
3
2
1
In
0 1 2 3 4 5 6 7 8 9 0 1 2


7
6
5
4
3
2
1
In
0 1 2 3 4 5 6 7 8 9 0 1 2


7
6
5
4
3
2
1
Out
0 1 2 3 4 5 6 7 8 9 0 1 2

El punto **p** es **adentro** de **P** si y sólo si la suma de los angulos es $\pm 2\pi$.

```
   c++  double angle(const point &a, const point &o, const point &b) {      //
returns angle aob in rad (without sign)     vec oa = toVector(o, a), ob =
toVector(o, b);     return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}

// returns true if point p is in either convex/concave polygon
bool inPolygon(const &point pt, const vector<point> &P) {
    if ((int)P.size() == 0) return false;
    double sum = 0;
    // assume the first vertex is equal to the last vertex
    for (int i = 0; i < (int)P.size()-1; i++) {
        if (ccw(pt, P[i], P[i+1]))
            sum += angle(P[i], pt, P[i+1]); // left turn/ccw
        else
            sum -= angle(P[i], pt, P[i+1]); // right turn/cw
     }
    return fabs(fabs(sum) - 2*PI) < EPS;
}
```

2.5.2 Algoritmo del crossing number

En este metodo, igualmente simple de implementar, contamos el número de veces que un **rayo** (semi-recta) saliendo de **p** cruza aristas del polígono. Igual que el anterior, funciona tanto para poligonos concavos o convexos.

Si el número es **par**, el punto está **afuera**. Sino, está adentro.

Pensar en las **secuencias de interfaces adentro/afuera que está cruzando el rayo**: para un punto de adentro estará forzosamente como in/out/in/out..../in/out.

Problemas de implementación posibles:

- los cruces contados tienen que ser in/out o out/in. En caso de pasar sobre un vertice, puede no ser el caso.
- puede ser que el rayo elegido se confunda con una arista del poligono...
- un punto que es sobre la frontera del poligono, pertenece al poligono?

```
bool inPolygon(const &point pt, const vector<point> &P) {
{
    int    cn = 0;    // the  crossing number counter
    // loop through all edges of the polygon
    for (int i = 0; i < (int)P.size()-1; i++) {
        line pty(0,1,-pt.y);
        point pint;
        if (intersect(pty,P[i], P[i+1],pint))
            if (pt.x < pint.x)
```

```
                ++cn;   // crossing at the right of pt.x
    }
    return (cn&1);    // 0 if even (out), and 1 if  odd (in)
}
```

2.5.3 Test para poligonos convexos

En el caso de que **P** sea convexo, un test aun más simple:

- Formar todos los triangulos $\mathbf{p}, \mathbf{P}[i], \mathbf{P}[i+1]$.
- Sumar los áreas de todos ellos.
- Si el punto está adentro, la suma es **igual** al area del poligono; sino, es superior.

UVA 478

Given a list of figures (rectangles, circles, and triangles) and a list of points in the x-y plane, determine for each point which figures (if any) contain the point.

Input: There will be n(10) figures descriptions, one per line. The first character will designate the type of figure (“r”, “c”, “t” for rectangle, circle, or triangle, respectively). This character will be followed by values which describe that figure.

- For a rectangle, there will be four real values designating the x-y coordinates of the upper left and lower right corners.
- For a circle, there will be three real values, designating the x-y coordinates of the center and the radius.
- For a triangle, there will be six real values designating the x-y coordinates of the vertices.

The end of the list will be signalled by a line containing an asterisk in column one. The remaining lines will contain the x-y coordinates, one per line, of the points to be tested. The end of this list will be indicated by a point with coordinates 9999.9 9999.9; these values should not be included in the output. Points coinciding with a figure border are not considered inside.

Output: For each point to be tested, write a message of the form: Point i is contained in figure j for each figure that contains that point. If the point is not contained in any figure, write a message of the form: Point i is not contained in any figure Points and figures should be numbered in the order in which they appear in the input.

Note: See the picture on the right for a diagram of these figures and data points.

```
bool inCircle(const point &pt,const point &pc,const double &r) {
  return dist(pt,pc)<r;
}

bool inRectangle(const point &pt,const point &ul,const point &lr) {
  return pt.x>ul.x && pt.x<lr.x && pt.y<ul.y && pt.y>lr.y;
}

bool inConvexPolygon(const point &pt, const vector<point> &p) {
  double sum=0.0;
  for (int i=0;i<p.size()-1;i++) {
    vector<point> t(4); t[0]=p[i]; t[1]=pt; t[2]=p[i+1]; t[3]=t[0];
    sum += area(t);
```

```
  }
  return (fabs(sum-area(p))<EPS);
}

int main() {
  string s="#";
  vector<point> circles;
  vector<double> radiuses;
  vector<vector<point> > triangles;
  vector<vector<point> > rectangles;
  map<int,pair<int,char>> corr;
  int j=1;
  while (s[0]!='*') {
    getline(cin,s);
    switch (s[0]) {
      case '*':
        break;
      case 't': {
          vector<point> t(4);
          stringstream ss(s);
          char c;
          ss >> c;
          ss >> t[0].x >> t[0].y >> t[1].x >> t[1].y >> t[2].x >> t[2].y;
          t[3]=t[0];
          triangles.push_back(t);
          corr[j++]=make_pair(triangles.size()-1,'t');
        }
        break;
      case 'r': {
          vector<point> r(2);
          stringstream ss(s);
          char c;
          ss >> c;
          ss >> r[0].x >> r[0].y >> r[1].x >> r[1].y;
          rectangles.push_back(r);
          corr[j++]=make_pair(rectangles.size()-1,'r');
        }
        break;
      case 'c': {
          point pc; double radius;
          stringstream ss(s);
          char c;
          ss >> c;
          ss >> pc.x >> pc.y >> radius;
          circles.push_back(pc);
          radiuses.push_back(radius);
          corr[j++]=make_pair(circles.size()-1,'c');
        }
```

```
        break;
    }
  }
  point p;
  int k=1;
  while (cin >> p.x >> p.y) {
    if (fabs(p.x-9999.9)<EPS && fabs(p.y-9999.9)<EPS)  break;
    bool inSome = false;
    for (int l=1;l<j;l++) {
      if (corr[l].second=='c' && inCircle(p,circles[corr[l].first],radiuses[corr[l].first])) {
        cout << "Point " << k << " is contained in figure " << l << endl;
        inSome = true;
      }
      if (corr[l].second=='r' && inRectangle(p,rectangles[corr[l].first][0],rectangles[corr[l].f
          cout << "Point " << k << " is contained in figure " << l << endl;
          inSome = true;
      }
      if (corr[l].second=='t' && inConvexPolygon(p,triangles[corr[l].first])) {
          cout << "Point " << k << " is contained in figure " << l << endl;
          inSome = true;
      }
    }
    if (!inSome) {
      cout << "Point " << k << " is not contained in any figure" << endl;
    }
    k++;
  }

}
```

Un test rápido que se puede incluir:

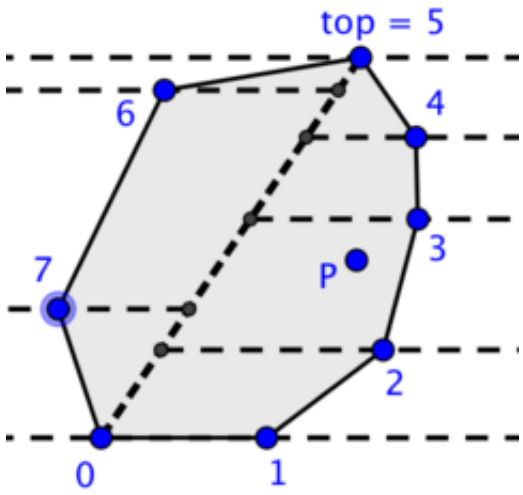
- guardar la **caja englobante** (rectangulo con valores mínimas y máximas de x,y);
- empezar los tests de inclusión con esa caja.

2.5.4 Test eficiente para poligonos convexos (1)

En el caso de poligonos convexos, hay manera de hacer los tests para multiples queries de manera mucho mas eficientes: sea $y(\mathbf{p})$ la coordenada del query.

1. Estimar los mínimos y máximos valores de y entre todos los puntos de **P** (costo?)
2. Observar que entre los indices del polígono correspondiendo a y_{min} y y_max, tenemos intervalos de segmentos con y variando de manera monotona.
3. En cada intervalo, hacer una **búsqueda binaria** para ubicar a los segmentos que cruzan a $y = y(\mathbf{p})$.
4. Verificar por fín con los x de esas intersecciones (x_1, x_2) están tales que

$$x_1 \leq x(\mathbf{p}) \leq x_2.$$



- Preprocesamiento $O(n)$ para determinar las extremidades.
- **Consultas en** $O(\log n)$.
- Muy interesante si muchas queries.

2.5.5 Test eficiente para poligonos convexos (2)

Otro enfoque (similar en idea):

- en lugar de ordenar dos sub-conjuntos de puntos con su coordenada y, elegir un **punto pivote** entre los puntos del poligono (p.ej. $\mathbf{P}[0]$).
- determinar (por ejemplo), el **ángulo polar** de cada $\mathbf{P}[i]$.
- por construcción la variación de este angulo **estará monotona**.

Luego determinar por **una sola busqueda binaria** en qué triangulo $\mathbf{P}[0], \mathbf{P}[i], \mathbf{P}[i+1]$ está el angulo polar del punto-query.

Finalmente, hacer el **check de que el punto está o no en el triangulo**.

Todo queda en $O(n)$.

2.6 Envolvente convexa

El **poligono convexo de area minima** que envuelva (contenga) un cierto conjunto de puntos.

Analogia con la **banda elástica**.

2.6.1 Algoritmo ingenuo

Un algoritmo ingenuo para evaluar el envolvente convenxo ($CH(\mathbf{P})$) se basa en su **estructura de aristas**:

- El convex hull está conformado por aristas de puntos.
- Para todos los puntos de $CH(\mathbf{P})$ hay dos aristas juntando ellos que están incluidas dentro de $CH(\mathbf{P})$.

Algorithm SLOWCONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list $\mathcal{L}$ containing the vertices of $\mathcal{CH}(P)$ in clockwise order.
1. $E \leftarrow \emptyset$.
2. **for** all ordered pairs $(p,q) \in P \times P$ with p not equal to q
3. **do** *valid* $\leftarrow$ **true**
4. **for** all points $r \in P$ not equal to p or q
5. **do if** r lies to the left of the directed line from p to q
6. **then** *valid* $\leftarrow$ **false**.
7. **if** *valid* **then** Add the directed edge $\overrightarrow{pq}$ to E.
8. From the set E of edges construct a list $\mathcal{L}$ of vertices of $\mathcal{CH}(P)$, sorted in clockwise order.

- Al considerar cada una de esas aritas $[\mathbf{p}, \mathbf{q}]$, si está orientada de tal manera que $\mathbf{p} \rightarrow \mathbf{q}$ en sentido horario alo largo de $CH(\mathbf{P})$, todos los otros puntos de $\mathbf{P} \backslash \{\mathbf{p}, \mathbf{q}\}$ están a **la derecha**.
- Al considerar todas las aristas orientadas, me deben se salir todas las buenas.

- Las aristas seleccionadas están dirigidas de tal forma que los otros puntos estén a la derecha de ellas: para todas las tripletas (p, m, q) deberíamos de ir de p a q en el sentido horario.
- Podríamos aplicar el mismo algoritmo para construir el $CH(\mathbf{P})$ en el sentido contra-horario.

- Una implementación simple de la línea 8 toma $O(n^2)$ pero se puede mejorar a $O(n \log n)$ (cómo?).
- Se repasa $n^2 - n$ pares de puntos y para cada una checamos $n - 2$ puntos para encontrar los que están del lado derecho: $O(n^3)$.

- **Complejidad total cúbica**.

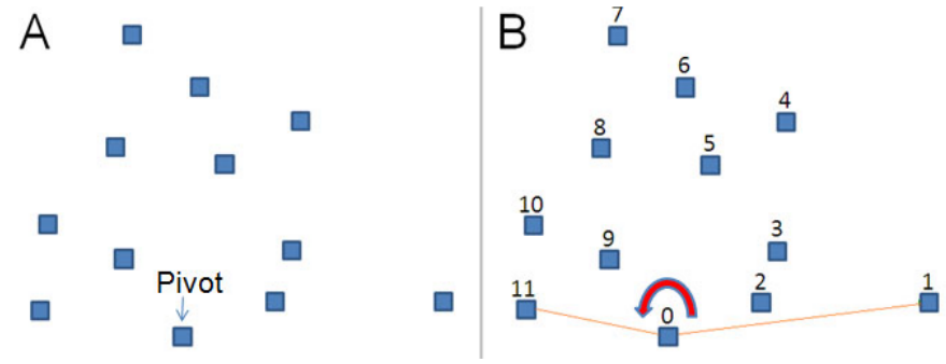
Algunas dificultades:

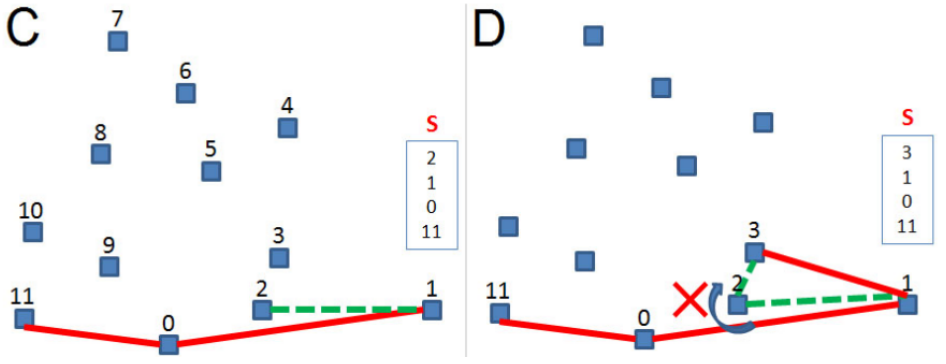
- Puntos pueden no estar a la derecha pero estar en realidad **co-lineares*: Se necesita verificar y procesar a parte.
- En ese caso, riesgo con errores de punto flotante, que puede llevar a no tener todas las aristas.

2.6.2 Algoritmo de Graham

- El algoritmo de Graham trabaja con los puntos de **P** sin necesitar duplicar el primer punto.
- Uno de esos puntos es **elegido como pivote**. Por ejemplo, el punto de coordenada y más chica.
- Se evalua el angulo polar de todos los otros puntos con respecto a este punto y se les ordena por valor creciente.
- Las aristas $0 - i$ vienen por construcción en **orden anti-horario**.

- Mantenemos una **pila S** de puntos candidatos a estar en el envolvente convexo.
- Como propriedad nos aseguraremos en cualquier momento que esta pila es tal que los 3 ultimos puntos puestos estén **en el sentido anti-horario** (dicho de otra manera, que el último punto agregado es siempre a la **izquierda** del vector formado por los dos anteriores.


A
Pivot
B
7
6
4
8
5
10
9
3
11
2
1
0


C
7
6
4
8
5
10
9
3
11
2
1
0
S
2
1
0
11
D
3
1
0
11

- Para cada punto p de **P**, intentamos insertar este punto una vez. Si la propriedad no se cumple, corregimos.

Aquí: inicialmente, por construcción, $N-1$, 0 y 1 están en el CH y **aparecen en orden anti horario**.

Al agregar 2, seguimos verificando la propriedad de orden anti-horario.

Cuando pasamos a 3, la tripleta (1, 2, 3) está en **sentido horario**: no se podría tener un poligono convexo al final, así que quitamos el punto 2.

Al hacer eso, **mantenemos la propriedad**: por qué?

Simplemente, si 2 se encuentra a la izquierda de (0,1), y 3 está a la derecha de (1,2), por lo que 3 queda a la izquierda de (0,1).

- Procedemos de la misma manera para todos los puntos consecutivos.
- Al terminar, **lo que queda en S son los puntos de la envolvente convexa**.
- El algoritmo elimina todos las vueltas a la derecha.

- Complejidad lineal del ciclo de construcción de **S**.
- Complejidad $O(n \log n)$ para el ordenamiento inicial.

```
// Code from S. Halim's book
point pivot(0, 0);
bool angleCmp(const point &a, const point &b) {
    // angle-sorting function
    if (collinear(pivot, a, b))
        // special case
        return dist(pivot, a) < dist(pivot, b); // check which one is closer
    double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
    double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
    return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;  // compare two angles
}

vector<point> CH(const vector<point> &P) {
    // the content of P may be reshuffled
    int i, j, n = (int)P.size();
    if (n <= 3) {     // special case, the CH is P itself
        if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
        return P;
    }

    // first, find P0 = point with lowest Y and if tie: rightmost X
    int P0 = 0;
    for (i = 1; i < n; i++)
        if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
            P0 = i;
    point temp = P[0]; P[0] = P[P0]; P[P0] = temp;
    // swap P[P0] with P[0]

    // second, sort points by angle w.r.t. pivot P0
    pivot = P[0]; // use this global variable as reference
```

```
    sort(++P.begin(), P.end(), angleCmp); // we do not sort P[0]

    // third, the ccw tests
    vector<point> S;
    S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);
    // initial S
    i = 2;
    // then, we check the rest
    while (i < n) {
        // note: N must be >= 3 for this method to work
        j = (int)S.size()-1;
        if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]); // left turn, accept
        else S.pop_back(); // or pop the top of S until we have a left turn
    }
    return S;
}
```

UVA 10065

Yes, as you have apprehended the Useless Tile Packers (UTP) pack tiles. The tiles are of uniform thickness and have simple polygonal shape. For each tile a container is custom-built. The floor of the container is a convex polygon and under this constraint it has the minimum possible space inside to hold the tile it is built for. But this strategy leads to wasted space inside the container. The UTP authorities are interested to know the percentage of wasted space for a given tile.

Input: The input file consists of several data blocks. Each data block describes one tile. The first line of a data block contains an integer N ($3 \leq N \leq 100$) indicating the number of corner points of the tile. Each of the next N lines contains two integers giving the (x, y) co-ordinates of a corner point (determined using a suitable origin and orientation of the axes) where $0 \leq x, y \leq 1000$. Starting from the first point given in the input the corner points occur in the same order on the boundary of the tile as they appear in the input. No three consecutive points are co-linear. The input file terminates with a value of '0' for N.

Output: For each tile in the input output the percentage of wasted space rounded to two digits after the decimal point. Each output must be on a separate line. Print a blank line after each output block.

- Simplemente calcular la CH.
- Luego calcular las dos areas: la de la envolvente y la del poligono.

```
int main() {
  int N,k=1;
  while (cin >> N) {
    if (N==0) break;
    vector<point> p(N+1);
    for (int i=0;i<N;i++) {
      cin >> p[i].x;
      cin >> p[i].y;
    }
    p[N].x=p[0].x;
```

```
    p[N].y=p[0].y;
    // Area of P
    double areaP = area(p);
    vector<point> ch = CH(p);
    double areaCH= area(ch);
    cout << "Tile #" << k++ << endl;
    cout << fixed;
    cout << setprecision(2);
    cout << "Wasted Space = " << 100.0*(areaCH-areaP)/areaCH << " %" << endl;
    cout << endl;
  }
}
```

USACO 2014 January Contest, Gold

Cow curling is a popular cold-weather sport played in the Moolympics.
Like regular curling, the sport involves two teams, each of which slides N heavy stones (3 <= N <= 50,000) across a sheet of ice. At the end of the game, there are 2N stones on the ice, each located at a distinct 2D point.

Scoring in the cow version of curling is a bit curious, however. A stone is said to be "captured" if it is contained inside a triangle whose corners are stones owned by the opponent (a stone on the boundary of such a triangle also counts as being captured). The score for a team is the number of opponent stones that are captured.

Please help compute the final score of a cow curling match, given the locations of all 2N stones.

INPUT FORMAT:

- Line 1: The integer N.
- Lines 2..1+N: Each line contains 2 integers specifying the x and y coordinates of a stone for team A (each coordinate lies in the range -40,000 .. +40,000).
- Lines 2+N..1+2N: Each line contains 2 integers specifying the x and y coordinates of a stone for team B (each coordinate lies in the range -40,000 .. +40,000).

OUTPUT FORMAT:

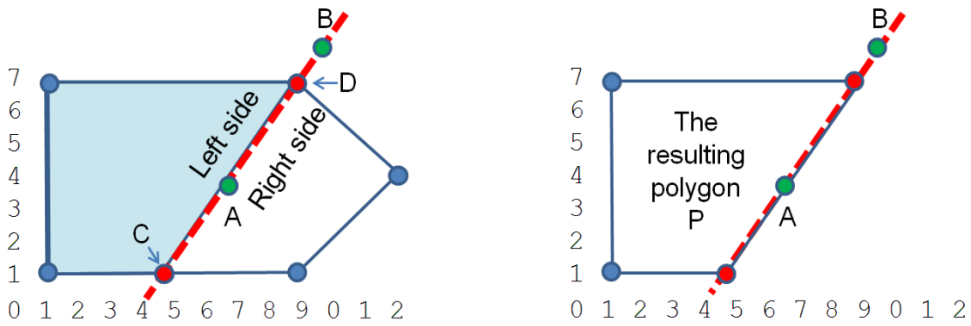
- Line 1: Two space-separated integers, giving the scores for teams A and B.

- Los puntos capturados son los puntos dentro de la envolvente convexa!
- La CH **se puede siempre triangulizar a partir de los puntos del conjunto**.
- En el otro sentido, **si un punto esta fuera del CH, no puede haber un triangulo de puntos del conjunto inicial que lo contenga** (prueba por el absurdo).

Basta calcular el CH de cada conjunto (cada uno en $N \log N$).
Luego verificar si los puntos del otro conjunto están incluidos en el CH.

UVA 11265

Once upon a time, there lived a great sultan, who was very much fond of his wife. He wanted to build a Tajmahal for his wife (ya, our sultan idolized Mughal emperor Shahjahan). But alas, due to budget cuts, loans, dues and many many things, he had no fund to build something so big. So, he decided to build a beautiful garden, inside his palace. The garden is a rectangular piece of



land. All the palaces water lines run through the garden, thus, dividing it into pieces of varying shapes. He wanted to cover each piece of the land with flowers of same kind.

The figure above shows a sample flower garden. All the lines are straight lines, with two end points on two different edge of the rectangle. Each piece of the garden is covered with the same kind of flowers. The garden has a small fountain, located at position (x, y). You can assume that, it is not on any of the lines. He wants to fill that piece with her favourite flower. So, he asks you to find the area of that piece.

Input: Input contains around 500 test cases. Each case starts with 5 integers, N, W, H, x, y, the number of lines, width and height of the garden and the location of the fountain. Next N lines each contain 4 integers, x1 y1 x2 y2, the two end points of the line. The end points are always on the boundary of the garden. You can assume that, the fountain is not located on any line. Constraints: 1. $1 \leq W, H \leq 200,000$ 2. $1 \leq N \leq 500$

Output: For each test case, output the case number, with the area of the piece covered with her favourite flower. Print 3 digits after the decimal point.

- Hay que determinar el **área del polígono en el cual está ubicado la fuente**.
- Primero implica **recuperar la representación del polígono** en el cual está la fuente.
- Para obtenerla: tomamos todos los segmentos de uno por uno; cada uno divide el poligono actual en dos; vemos de que lado esta la fuente (con test de signo con el corte); tomamos el siguiente segmento para dividir de nuevo el sub-poligono en la cual está la fuente e iteramos.

Eso implica tener un algoritmo para **dividir el poligono por una linea**.

2.7 Cortar un poligono por una linea

Lo veremos con poligonos **convexos**: Dado un poligono **P** y una recta definida por dos puntos **p** y **q**, queremos obtener (si es el caso) la representación de uno de los dos poligonos resultantes del corte (**Q**).

Idea:

- Iterar entre todos los vértices de **P**.
- Si un vertex $\mathbf{P}[i]$ está ala lizquierda de linea orientada $(\mathbf{p}, \mathbf{q})$ (usar el test ccw), agregar $\mathbf{P}[i]$ a un nuevo poligono **Q**, sino dejarle en **P**.

- Durante el recorrido, cuando encontramos una arista del poligono que interseca la linea $(\mathbf{p}, \mathbf{q})$, determinamos esa intersección y la agregamos.

- En algun momento, regresaremos del otro lado de $(\mathbf{p}, \mathbf{q})$: detectamos el siguiente segmento que cruza la recta, agregamos la intersección y terminamos agregando los siguientes puntos de **P** hasta el vértice inicial.

```
// line segment p-q intersect with line A-B.
point lineIntersectSeg(const point &p, const point &q, const point &A, const point &B) {
    double a = B.y - A.y;
    double b = A.x - B.x;
    double c = B.x * A.y - A.x * B.y;
    double u = fabs(a * p.x + b * p.y + c);
    double v = fabs(a * q.x + b * q.y + c);
    return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v));
}

// cuts polygon P along the line formed by point a -> point b
// and returns the part on the left
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(const point &a, const point &b, const vector<point> &P) {
    vector<point> Q;
    for (int i = 0; i < (int)P.size(); i++) {
        double left1 = cross(toVec(a, b), toVec(a, P[i])), left2 = 0;
        if (i != (int)P.size()-1)
            left2 = cross(toVec(a, b), toVec(a, P[i+1]));
        // When on the left, the point is copied to Q
        if (left1 > -EPS) // P[i] is on the left of ab
            Q.push_back(P[i]);
        // Detect a crossing
        if (left1 * left2 < -EPS) // edge (Q[i], Q[i+1]) crosses line ab
            Q.push_back(lineIntersectSeg(P[i], P[i+1], a, b));
    }
    if (!Q.empty() && !(Q.back() == Q.front()))
        Q.push_back(Q.front()); // make P's first point = P's last point
    return Q;
}
```

2.7.1 Par de puntos ḿas alejados

El algoritmo ingenuo para determinar el diametro: **recorrer todos los pares de puntos** de **P** para determinar la distancia máxima. Tiempo cuadrático.

Afortunadamente, podemos hacer mejor que eso.

Nos apoyamos sobre las nociones de:

- **rectas soportes** que son aquellas que (1) pasan por un punto **p** del conjunto **P** y (2) deja a todos los puntos de $\mathbf{P}\backslash\mathbf{p}$ en el mismo semiplano que delimita.
- **puntos antipodales** que son puntos **p**, **q** de **P** por los cuales pasan rectas soporte paralelas, tales que cada recta deja a los otros puntos de cada lado (o sea, todos los puntos están **entre** las rectas).

[de Wikipedia]
Propriedades para un poligono **convexo P**:

- Si consideramos una de sus aristas, $\mathbf{P}[k-1]\mathbf{P}[k]$ y recorremos los vértices de **P** en el **orden antihorario** desde $\mathbf{P}[k]$. Si $\mathbf{P}[i]$ es el **primer** vértice más lejano del segmento $\mathbf{P}[k-1]\mathbf{P}[k]$ durante el recorrido, ningún vértice entre $\mathbf{P}[k]$ y $\mathbf{P}[i]$ puede formar un par antipodal con $\mathbf{P}[k]$.
- Si consideramos una de sus aristas, $\mathbf{P}[k]\mathbf{P}[k+1]$ y recorremos los vértices de **P** en el **orden antihorario** desde $\mathbf{P}[k+1]$. Si $\mathbf{P}[u]$ es el último vértice más lejano del segmento $\mathbf{P}[k]\mathbf{P}[k+1]$ durante el recorrido, ningún vértice entre $\mathbf{P}[u]$ y $\mathbf{P}[k]$ puede formar un par antipodal con $\mathbf{P}[k]$.

Observación: es equivalente maximizar la **distancia de los puntos al segmento** que maximizar el **área** del triangulo correspondiente.

Otra propriedad: un **par de puntos antipodales de un conjunto de puntos** necesariamente **pertenecen a la envolvente convexa del conjunto**.

Entonces encontrar pares de puntos antipodales en un conjunto de puntos arbitrario se reduce a **buscar pares de puntos antipodales en la envolvente convexa** del conjunto de puntos.

Un algoritmo para hacerlo se llama algoritmo de **calibre rotatorio**.
[de http://www-cgrl.cs.mcgill.ca/~godfried/research/calipers.html]

- Idea: Recorrer los segmentos del poligono y recorrer **paralelamente** los puntos antipodales, que van a ser por las propriedades descritas arriba **encontrados entre los puntos más alejados del segmento precedente y el punto más alejado del actual**.
- Usar el criterio de area para detectar los puntos más alejados: como la distancia (y entonces el area) va creciendo hasta llegar a su máximo, podemos detectar el momento en que decrece:

```
while (area(k,k+1,i+1) > area(k,k+1,i)) {
```

- Un punto delicado que considerar es los segmentos **paralelos** (en ese caso cada extremidad de segmento es antipodal con cada extremidad del otro).

```
vector<point> P;
vector<pair<int,int> > antipodals;

void getAntipodals() {
 int k = 0;
 int i = k+1, i0=i+1;
 int n = P.size();
 // Determines the farthest point from [k,k+1].
 while (area(k,k+1,i+1) > area(k,k+1,i)) {
   i0 = ++i; // i0 will have the point where we started for i.
 }
 // Cycling wih the rotating calipers
 while (i<n-1) {
```

```
    k++;
    //
    antipodals.push_back(make_pair(k,i));
    // Search farthest point from [k,k+1]
    while (area(k,k+1,i+1) > area(k,k+1,i)) {
      i++;
      // When finishing the cycle, leaves
      if (k==i0 || i==n-1)
          return;
      antipodals.push_back(make_pair(k,i));
    }
    // This is to handle the case when [i,i+1] is parallel to [k,k+1]
    if (area(k,k+1,i+1) == area(k,k+1,i)) {
      if (k!=i0 && i!=n-1)
        antipodals.push_back(make_pair(k,i+1));
      else
        antipodals.push_back(make_pair(k+1,i));
    }
  }
}
```

Si queremos el diámetro de un conjunto de puntos en el plano, es decir la mayor distancia entre dos puntos. Si **p** y **q** son puntos extremos de ese diametros, es claro que **tienen que ser antipodales**: Si trazamos dos rectas paralelas, perpendiculares a $[\mathbf{p}, \mathbf{q}]$ y pasando respectivamente por **p** y **q**, todos los puntos del conjunto tienen que estar en la banda entre esas rectas; si no fuera el caso, y que encontremos un **r** allá de **p** (resp. **q**), tendríamos también una distancia mayor con **r**, **q** (resp. **r**, **p**).

Entonces **encontrar el diámetro de un conjunto de puntos**, recorremos todos los pares de puntos antipodales de su envolvente convexa buscando los más alejados en ese recorrido.

Costo:

- $O(n \log(n))$ para la envolvente convexa,
- $O(n)$ para los pares antipodales y la actualización del par más lejano.

2.7.2 Par de puntos más cercanos

Igual que en el caso anterior (distancia máxima), podemos **recorrer todos los pares**, y el desempeno estará cuadrático.

Vamos a ver cómo mejorar este desempeno.

Caso 1D

En una dimensión, el problema se puede plantear de manera fácil como problema de tipo Divide and Conquer:

- Cortar los datos en el dato mediano, entre dos sub-grupos $\mathbf{S}_1$ y $\mathbf{S}_2$.
- Recursivamente, llamar a la función sobre $\mathbf{S}_1$ y $\mathbf{S}_2$.

El resultado es:

- o $\min_{i,j \in \mathbf{S}_1} d_{ij}$ (los dos puntos $\mathbf{p}_1$ y $\mathbf{q}_1$ en $\mathbf{S}_1$)

- o $\min_{i,j \in \mathbf{S}_2} d_{ij}$ (los dos puntos $\mathbf{p}_2$ y $\mathbf{q}_2$ en $\mathbf{S}_2$)
- o los dos puntos están del lado puesto.

En 1D, el 3r caso (los dos puntos del lado opuesto) se resuelve muy facilmente:

- el primer punto tiene que ser el **mayor de** $\mathbf{S}_1$ ($\mathbf{m}_1$),
- el segundo punto tiene que ser el **menor de** $\mathbf{S}_2$ ($\mathbf{m}_2$).

Esto no se generaliza facilmente en 2D (o en dimensiones mayores): Hay **más candidatos** de ambos lados (que el único candidato del caso 1D)

Sea:

$$\delta = \min(\|\mathbf{p}_1 - \mathbf{q}_1\|, \|\mathbf{p}_2 - \mathbf{q}_2\|)$$

entonces, la solución final es $\min \delta, \|\mathbf{m}_1\mathbf{m}_2\|$.

Una observación importante (para la **generalización** a mayor dimensiones): para poder considerar la 3a opción, los puntos $\mathbf{m}_i$ **tienen que ubicarse a menos de** δ de la separación.

Caso 1D: complejidad

- Requerimos determinar

- La recurrencia del DaC está dada por

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Resultando en $O(n \log n)$ (Master Theorem).

Generalización a 2D:

- Nos podemos inspirar del caso 1D y aplicar el DaC.
- **Dividimos los datos** en dos sub-grupos $\mathbf{S}_1$ y $\mathbf{S}_2$, por ejemplo por la mediana de la coordenada x.
- Resolvemos los dos sub-problemas en $\mathbf{S}_1$ y $\mathbf{S}_2$.
- Chequemos entre possibles candidatos situados en ambos lados de la separación si algun par no da una distancia inferior.

Una primera opción sería verificar todos los puntos y luego todas los pares (pero nos llevaría a algo **cuadrático**).

Sea $p \in \mathbf{S}_1$, ubicado a menos de δ de la linea mediana.

Observamos que los puntos de $\mathbf{S}_2$ ubicados a una distancia de $\mathbf{p}$ inferior a δ no pueden estar fuera de un **rectangulo** de tamano $\delta \times 2\delta$.

Un **punto clave**: en realidad no puede haber muchos puntos en este rectangulo porque sabemos que la distancia entre puntos en $\mathbf{S}_2$ es por definición superior a δ!

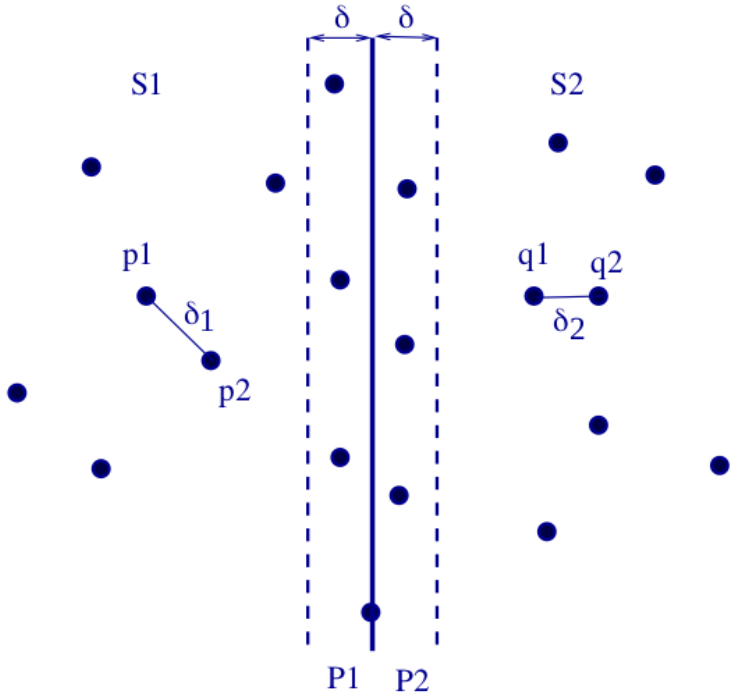
Puede haber al máximo 6 puntos!

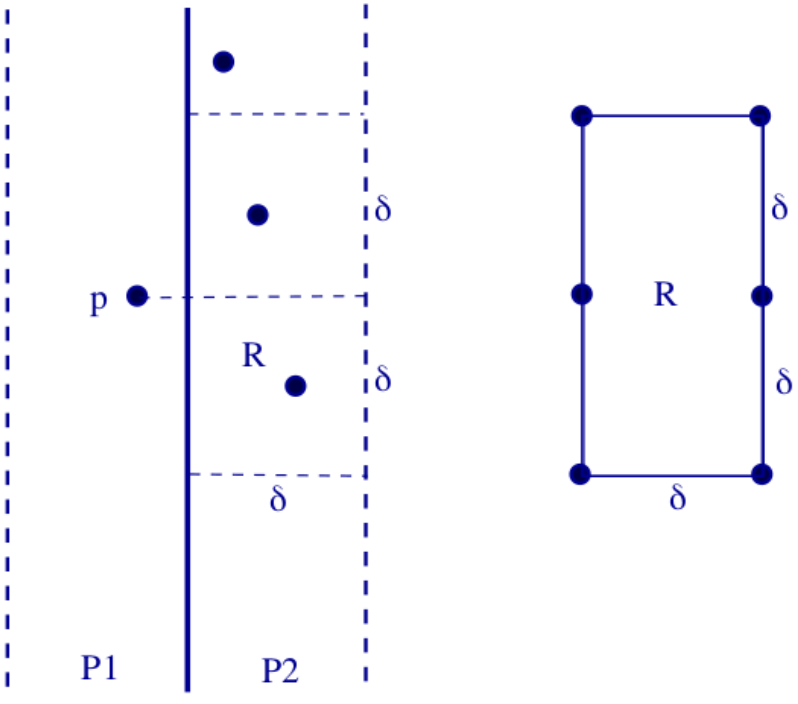
Significa que para la fase de conquer en DaC tenemos que comparar distancia de tipo $\|\mathbf{p} - \mathbf{q}\|$ para (en peor caso)

$$\frac{n}{2} \times 6 \text{ pares.}$$

Lo único que nos falta es poder encontrar esos máximo 6 "vecinos" rapidamente.

Idea:


δ
δ
S1
S2
p1
δ1
p2
q1
q2
δ2
P1
P2


δ
p
R
δ
δ
P1
P2
δ
R
δ
δ

- proyectar todos los puntos de $\mathbf{P}_1$ y $\mathbf{P}_2$ sobre la linea mediana;
- usar la misma recursión del algoritmo para implementar un mergesort con la coordenada y!
- recorrer la lista ordenada de puntos en la banda central para determinar, para un punto de $\mathbf{P}_1$, los puntos de $\mathbf{P}_2$ que pueden estar en la vecindad.

El costo de esta operación será lineal!
Costo de la recursión: otra vez

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

o sea un costo total de $O(n \log n)$.
UVA 10245

Given a set of points in a two dimensional space, you will have to find the distance between the closest two points.

Input: The input file contains several sets of input. Each set of input starts with an integer N($0 \leq$ N ≤ 10000), which denotes the number of points in this set. The next N line contains the coordinates of N twodimensional points. The first of the two numbers denotes the X-coordinate and the latter denotes the Y -coordinate. The input is terminated by a set whose N = 0. This set should not be processed. The value of the coordinates will be less than 40000 and non-negative.

Output: For each set of input produce a single line of output containing a floating point number (with four digits after the decimal point) which denotes the distance between the closest two points. If there is no such two points in the input whose distance is less than 10000, print the line 'INFINITY'

2.7.3 Intersección de segmentos.

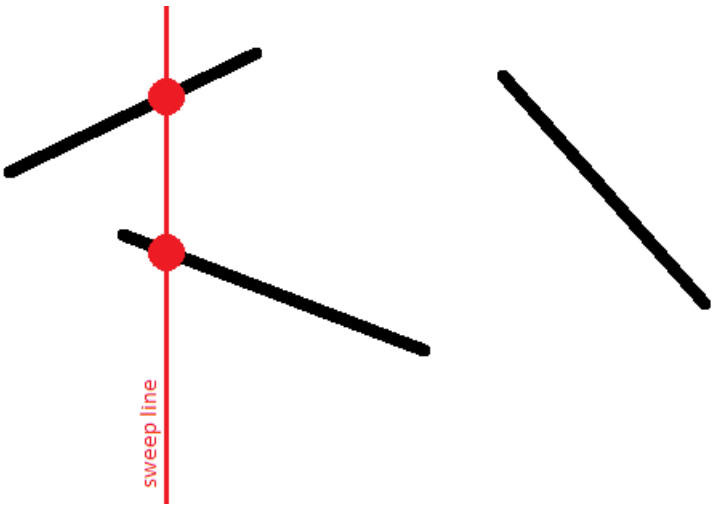
Dado un conjunto **S** de n segmentos de recta cerrados en el plano, reportar un punto de intersección de segmentos en **S** (o decir que no hay).

- ¿Algoritmo de fuerza bruta?
- ¿Complejidad?

Idea básica similar al problema del par de puntos más cercanos: Segmentos cercanos son candidatos a ser verificados, los más lejanos no.

- Imaginar una linea vertical $x = X$ empezando desde $-\infty$ y moviendose hacia la derecha.
- Mientras mueve, interseca diferentes segmentos.
- Para un segmento dado, ese segmento **aparece como intersección con la linea de barrido, a un cierto momento, y luego desaparece**.
- Mantenemos, mientras X varia, el orden relativo de los segmentos intersecando la linea de barrido,
- Dos segmentos se cruzan implica que **deben de aparecer adyacentes en la lista ordenada de intersecciones** con la linea de barrido.

- Podemos contentarnos de describir el barrido en **sólo los eventos de aparición y desaparición de segmentos**.
- Al tener una **aparición** de un segmento,
  - determinamos donde queda en la estructura de segmentos (tipicamente un set);



  - checamos intersección con sus dos vecinos;
  - lo insertamos;

- Al tener una **desaparición** de un segmento,
  - lo buscamos en la estructura;
  - checamos intersección entre sus dos vecinos;
  - lo quitamos.

Dos operadores de comparación:

- para ordenar los **eventos**, operador usando la coordenada x del evento (min/max de las extremidades).
- para ordenar los **segmentos al nivel de un evento**, operador usando la coordenada y de la intersección con el barrido.

Complejidad:

- $2n$ eventos.
- Ordenamiento en $O(n \log n)$.
- Busquedas en $\log n$ ($2n$ de ellas).
- Calculos de intersecciones: $3n$ máximo.
- Inserciones/deleciones en $O(\log n)$.

Total $O(n \log n)$.

```
// Code inspired from https://cp-algorithms.com/geometry/intersecting_segments.html
set<seg> s;
// Keep the position of the segments
vector<set<seg>::iterator> position;

set<seg>::iterator prev(set<seg>::iterator it) {
    return it == s.begin() ? s.end() : --it;
```

```
}

set<seg>::iterator next(set<seg>::iterator it) {
    return ++it;
}

pair<int, int> findIntersection(const vector<seg>& segs) {
    vector<event> evs;
    // Push the starting/ending segment events
    for (int i = 0; i < segs.size(); i++) {
        evs.push_back(event(min(segs[i].p.x, segs[i].q.x), +1, i));
        evs.push_back(event(max(segs[i].p.x, segs[i].q.x), -1, i));
    }
    // Sort the events by their x coordinate
    sort(evs.begin(), evs.end());
    //
    s.clear();
    position.resize(segs.size());
    for (int i = 0; i < evs.size(); i++) {
        // Which segment?
        int id = evs[i].id;
        // Which event
        if (evs[i].tp == +1) {
            // Locate its place in s
            set<seg>::iterator nxt = s.lower_bound(segs[id]),
            prv = prev(nxt);
            // Test intersection between next and the new one
            if (nxt != s.end() && intersect(*nxt, segs[id]))
                return make_pair(nxt->id, id);
            // Test intersection between prev and the new one
            if (prv != s.end() && intersect(*prv, segs[id]))
                return make_pair(prv->id, id);
            // Insert it
            position[id] = s.insert(nxt, segs[id]);
        } else {
            // Determine the one above and the one
            set<seg>::iterator nxt = next(position[id]), prv = prev(position[id]);
            // Intersection prev/next?
            if (nxt != s.end() && prv != s.end() && intersect(*nxt, *prv))
                return make_pair(prv->id, nxt->id);
            // Remove the segment
            s.erase(position[id]);
        }
    }
    // Not found
    return make_pair(-1, -1);
}
```