Geometría

Brian Morris Esquivel

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Training Camp Argentina 2019

Guía de la clase

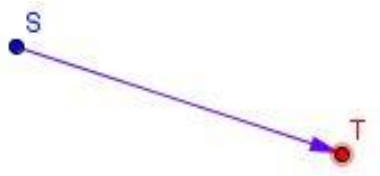
- **Estructura “punto”**
  - Algebra vectorial
  - Puntos como vectores
  - La estructura
- **Manejo de errores de precisión**
  - Errores de precisión
  - Cómo atacar errores de precisíón
- **Ángulos**
  - Amplitud angular como real
  - Amplitud angular como par de enteros
  - Aplicación

Estructura “punto”

Vectores

¿Qué es un vector?

Cuando hablamos de geometría euclideana podemos pensar a un vector como un *“segmento dirigido”*, con un punto inicial y un punto final.



Nosotros nos vamos a olvidar de esos “extremos”.
Nos interesa solo la **dirección**, el **sentido** y el **modulo** del mismo.

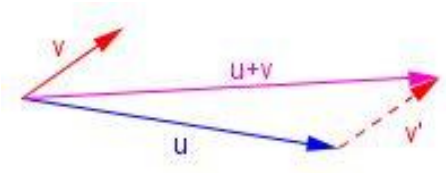
El vector lo podemos (y lo vamos a) representar en un sistema de ejes coordenados por sus ***“componentes”***.

$$\overline{\boldsymbol{V}} = (V_x, V_y)$$

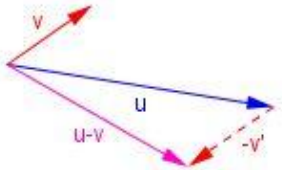
Operaciones con vectores

Podemos definir algunas operaciones entre vectores.

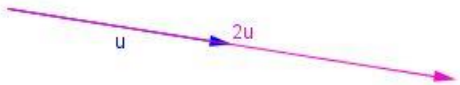
Suma de vectores:



Diferencia de vectores:



Producto de un vector por un escalar:



Módulo de un vector: $|\overline{\boldsymbol{u}}|$ = Distancia euclideana del origen al extremo

Producto escalar entre vectores: $\overline{\boldsymbol{u}} \times \overline{\boldsymbol{v}} = |\overline{\boldsymbol{u}}|.|\overline{\boldsymbol{v}}|.\cos(\hat{u,v})$

Producto cruz entre vectores: $\overline{\boldsymbol{u}}\hat{}\overline{\boldsymbol{v}} = |\overline{\boldsymbol{u}}|.|\overline{\boldsymbol{v}}|.\sin(\hat{u,v})$

Operaciones con vectores - Componentes

Y nos interesa mucho saber calcularlas a partir de sus componentes.

Suma de vectores: $(u_x, u_y) + (v_x, v_y) = (u_x + v_x, u_y + v_y)$

Diferencia de vectores: $(u_x, u_y) - (v_x, v_y) = (u_x - v_x, u_y - v_y)$

Producto de un vector por un escalar: $\alpha.(u_x, u_y) = (\alpha u_x, \alpha u_y)$

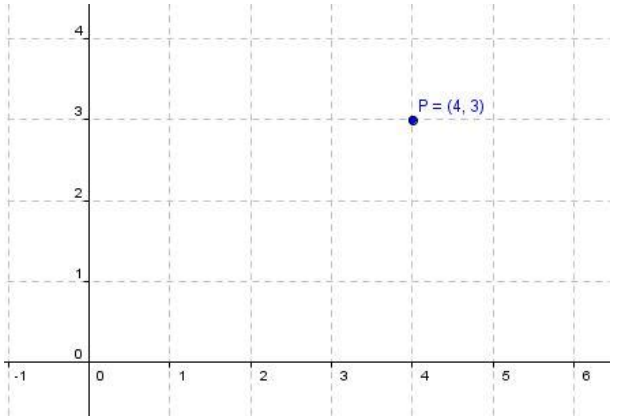
Módulo de un vector: $|(u_x, u_y)| = \sqrt{u_x^2 + u_y^2}$

Producto escalar entre vectores: $(u_x, u_y) \times (v_x, v_y) = u_x v_x + u_y v_y$

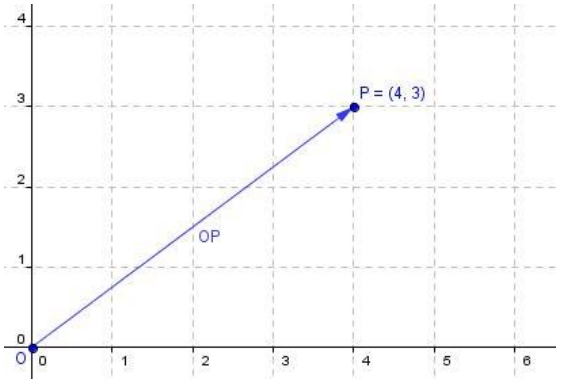
Producto cruz entre vectores: $(u_x, u_y)\hat{}(v_x, v_y) = u_x v_y - u_y v_x$

Vector asosciado a un punto

En problemas de ICPC nos vamos a encontrar con puntos definidos por sus coordenadas en un sistema de ejes cartesianos ortogonales.



Para cada uno de estos puntos podemos definir el **vector asosciado** a ese punto como un vector que “parte” del **origen** ($O = (0, 0)$) y “llega” a dicho punto.



Notemos que las componentes del vector asosciado son iguales a las coordenadas del punto.

$$OP_x = x_P, \qquad OP_y = y_P$$

Puntos como vectores

Dada la definición de vector asosciado, a partir de ahora nos vamos a permitir “operar entre puntos”.
Cada vez que nos refiramos a realizar una operación entre dos puntos estamos hablando de realizar esa operación entre sus vectores asosciados.

Nos vamos permitir por ejemplo, dados el **punto A** y el **punto B**, definir el **vector $\overline{AB}$** como:

$$B - A = \overline{AB}$$

De la forma en que trabajaremos, para la computadora no va a haber distinción entre un vector y un punto.
La diferenciación entre ellos va a estar únicamente en nuestra interpretación de que significa cada variable.

La estructura “punto”

Para trabajar cómodos en la PC vamos a definir la siguiente estructura:

```
typedef long long ll;
struct punto{
    ll x, y;
    punto(ll x = 0, ll y = 0): x(x), y(y){}                    // Constructor
    punto operator+(punto a){ return punto(x + a.x, y + a.y); }  // Suma de vectores
    punto operator-(punto a){ return punto(x - a.x, y - a.y); }  // Diferencia de vectores
    punto operator*(ll a){ return punto(x*a, y*a); }           // Producto por un escalar
    ll operator*(punto a){ return x*a.x + y*a.y; }             // Producto escalar
    ll operator^(punto a){ return x*a.y - y*a.x; }             // Producto cruz
    long double modulo(){ return hypotl(x, y); }               // Modulo del vector
};
```

En el caso de que los puntos tengan coordenadas no enteras:

```
typedef long double ld;
struct punto{
    ld x, y;
    punto(ld x = 0, ld y = 0): x(x), y(y){}
    punto operator+(punto a){ return punto(x + a.x, y + a.y); }
    punto operator-(punto a){ return punto(x - a.x, y - a.y); }
    punto operator*(ld a){ return punto(x*a, y*a); }
    ld operator*(punto a){ return x*a.x + y*a.y; }
    ld operator^(punto a){ return x*a.y - y*a.x; }
    ld modulo(){ return hypotl(x, y); }
};
```

Algunos nuevos tipos de variable

A partir de tener la estructura punto podemos definir un poco más fácil algunos tipos de variable que nos van a ser de utilidad al implementar problemas.

Puntos y vectores:

```
punto A, B;                    // Punto
punto AB;                      // Vector
```

Segmentos:

```
pair<punto, punto> S;    // Segmento
```

Polígonos y poligonales:

```
vector<punto> P, Q;        // Poligonos
vector<punto> PL;          // Poligonal
```

Algunas nuevas estructuras

A partir de tener la estructura punto también podemos definir nuevas estructuras que nos serán útiles en problemas específicos.

Estructura **linea**:

```
struct linea{
    punto P, PQ;     // Punto de paso y vector paralelo
    linea(punto P, punto Q): P(P), PQ(Q-P){}
    bool in(punto R){    // Dice si el punto esta en la recta
        punto PR = R - P;
        return (PR^PQ) == 0;
    }
};
```

Estructura **círculo**:

```
struct circulo{
    punto P;
    ll r;
    circulo(punto P, ll r): P(P), r(r){}
    bool in(punto Q){    // Dice si el punto esta estrictamente adentro
        return dist_sq(P, Q) < r*r;
    }
};
```

En esta clase no vamos a adentrarnos en el uso de estas estructuras (que pueden implementarse de forma mucho más completa).

Manejo de errores de precisión

Imprecisión

Cuando nos encontramos con problemas de geometría muchas veces nos vemos obligados a trabajar con números reales no enteros.

Por ejemplo, si necesitamos hallar el punto medio entre dos puntos a coordenadas enteras, éste no siempre va a tener coordenadas enteras.

Decimos que tenemos una **imprecisión**, cuando el valor calculado de una variable difiere mucho de su valor exacto.

En general, al trabajar con puntos flotantes, que el valor calculado difiera del valor exacto es **inevitable**. Nos ponemos como objetivo reducir esa diferencia todo lo que se pueda.

¿Por qué hay imprecisiones?

Las imprecisiones aparecen cuando tenemos que almacenar números reales (que los almacenamos como “puntos flotantes”).

Una diferencia importante entre los números enteros y los reales, es que, en general, los números reales no pueden ser representados en forma exacta por una computadora. Deben almacenarse truncados o redondeados.

¿Cuánta memoria requeriría almacenar en forma exacta el número π?

…
Exacto.

Al igual que cuando trabajamos en papel con una calculadora: cuantas más cuentas hagamos redondeando el resultado, más se alejará el resultado final de su valor exacto.

Tipos de errores de precisión

Se pueden diferenciar dos tipos de error como consecuencia de una imprecisión:

Respuesta imprecisa: Ocurre cuando damos como respuesta un resultado que se aleja mucho de la respuesta correcta. Por ejemplo:

Test: #9, time: 0 ms., memory: 1856 KB, exit code: 0, checker exit code: 1, verdict: WRONG_ANSWER

Input

```
99 23 530
```

Output

```
45004055.753419027
```

Answer

```
45004055.801775165000
```

Checker Log

```
wrong answer 1st numbers differ - expected: '45004055.8017751650', found: '45004055.7534190270', error = '0.0000000011'
```

Siempre que tengas que imprimir un valor real como respuesta se te va a dar un margen de error. Si respondés algo fuera de ese rango es porque tu algoritmo es muy impreciso.

Tipos de errores de precisión (cont.)

Falla en decisión: Ocurre cuando en algún momento del algoritmo debimos decidir si una condición es verdad (**true**) o mentira (**false**) en base a comparar dos variables reales guardadas como punto flotante. Si sus valores están muy cerca una imprecisión puede causar que la comparación falle.

Estos errores resultan bastante graves porque son más frecuentes y, relativamente, más difíciles de resolver.

Un ejemplo claro sobre este tipo de situaciones es cuando queremos saber si 3 puntos estan alineados. ¿Cómo lo harían? Piensenló.

Ejemplo 1

¡Bienvenido a la madriguera!

En la madriguera de los conejos hay diversión y juegos, hay todo lo que uno puede querer. Por eso Pascual, el conejo arquitecto, decidió mudarse aquí tras recibirse.

Intrigado por la belleza del lugar, la curiosidad de Pascual lo llevó a averiguar la forma que tenía la madriguera por dentro, le comentaron que era un polígono regular de N lados con distancia R del centro a los vertices.

La siguiente duda de Pascual es: ¿Cuál es el área de la superficie de la madriguera?

Desafortunadamente nadie conoce la respuesta, pero el sabe que con la información que recibió la puede calcular. ¿Cuál es el área?

Input: Vas a recibir los valores (enteros) de N y R.

Output: Debés responder la superficie de la mariguera.

Como corregir respuestas imprecisas – Papel

Hay que trabajar todo lo posible en papel.

Un error frecuente en problemas de geometría es escribir códigos como el siguiente para el ejemplo 1:

```
// Muchas funciones llevan una 'l' adelante para usarlas en long double
const ld PI = acosl(-1);
ld solve_feo(ll n, ll r){
    ld alpha = (2*PI)/n;     // Angulo central
    alpha /= 2;              // Medio angulo central
    ld l = 2*sinl(alpha)*r; // Lado del polígono
    ld h = cosl(alpha)*r;    // Apotema del polígono
    ld AT = h*l/2;           // Area de un triángulo
    ld A = AT*n;             // Area del polígono
    return A;
}
```

Cuando podríamos escribir códigos como el siguiente:

```
ld solve_lindo(ll n, ll r){
    ld A = n*r*r*sinl(2*PI/n)/2;
    return A;
}
```

Como corregir respuestas imprecisas – Papel (cont.)

Notemos que el primer código realiza al menos 10 operaciones con flotantes por ejecución y muchas de ellas son innecesarias.

Si eliminamos las variables auxiliares y escribimos todo como una expresión grande nos queda lo siguiente:

$$\frac{2 \sin\left(\frac{2\pi}{2n}\right) r \,.\cos\left(\frac{2\pi}{2n}\right) r.n}{2}$$

Donde es fácil ver que hay operaciones que están “de más”, por ejemplo, varias veces multiplicamos y dividimos por 2.

Debemos evitar este tipo de situaciones. Para hacerlo nos sentamos con papel y birome a reducir todas las expresiones que podamos. Apuntamos a **minimizar** la cantidad de cuentas que hace la PC.

Como corregir respuestas imprecisas – Papel (cont.)

Existe una ventaja del primer código con respecto al segundo, y es que éste es fácil de debuggear.

Las variables auxiliares nos ayudan a ver si las cuentas parciales están bien hechas y detectamos algún posible error más fácilmente.

Para conservar esta ventaja un código común de Brian es el siguiente:

```
ld solve_Brian(ll n, ll r){
    ld alpha = 2*PI/n;
//    printf("Angulo central: %.20Lf\n", alpha);
//    printf("Semiangulo central: %.20Lf\n", alpha/2);

//    ld l = 2*r*sinl(alpha/2), h = r*cosl(alpha/2);
//    printf("Lado: %.20Lf\n", l);
//    printf("Apotema: %.20Lf\n", h);
//    printf("Area triangulo-1: %.20Lf\n", h*l/2);
//    printf("Area triangulo-2: %.20Lf\n", r*r*sinl(alpha)/2);
    ld ans = n*r*r*sinl(alpha)/2;
//    printf("Area total: %.20Lf\n", ans);
    return ans;
}
```

Donde descomentamos siempre que queremos debuggear.

Ejemplo 2

Lluvia de Julio

En Julio hace terrible frío en el bosque, pero los animales no la pasan mal porque es cuando Gabriel el pingüino organiza su anual torneo de curling. Lo más curioso de este torneo es que los discos con los que se juega también están hechos de hielo.

Antes del torneo Gabriel debe fabricar los discos y para eso debe calcular cuanta agua necesitará. Los discos se fabrican como cilindros huecos de altura 1.

Mañana lloverá y será el momento de recolectar el agua, Gabriel necesita que lo ayudes y calcules el volumen total de agua necesario.

Si no lo ayudás el bosque entristecerá y caminará bajo la fría lluvia de Julio.

Input: Vas a recibir la cantidad N de discos y los radios externo R_i e interno r_i de cada uno de ellos.

Output: Debés responder el volumen total de los discos.

Como corregir respuestas imprecisas – Enteros

Hay que trabajar todo lo posible en enteros.

Aprendimos que el problema aparece cuando operamos con flotantes. Cuando operamos con enteros las cuentas son siempre exactas, en lo único que debemos ser cuiadosos es en no causar un **overflow**.

Realizar operaciones de suma, resta y mutiplicación entre números enteros no empeora la precisión de la respuesta, podemos decir que siempre son exactas.

Por eso, siempre que podamos reemplazar cuentas con reales por cuentas con enteros, bienvenidas sean.

Como corregir respuestas imprecisas – Enteros (cont.)

En el ejemplo 2, la respuesta al problema resulta:

$$S = \sum_{i=1}^{n} (R^2\pi - r^2\pi)$$

```
int n; scanf("%d", &n);
ld ans = 0;
for(int i = 0; i < n; i++){
    ll r, R;
    scanf("%lld %lld", &R, &r);
    ans += PI*R*R;
    ans -= PI*r*r;
}
printf("%.20Lf\n", ans);
```

Esto es un inconveniente, en este caso **no** vamos a poder evitar hacer muchas cuentas, pero **sí** podemos evitar hacerlas con flotantes. Podemos implementar lo siguiente:

$$S = \pi \sum_{i=1}^{n} (R^2 - r^2)$$

```
int n; scanf("%d", &n);
ll tot = 0;
for(int i = 0; i < n; i++){
    ll r, R;
    scanf("%lld %lld", &R, &r);
    tot += R*R - r*r;
}
printf("%.20Lf\n", tot*PI);
```

Y realizamos muchas cuentas pero solo una de ellas es en flotantes.

Ejemplo 3

Ten paciencia

Cansada de escalar árboles, Ludmila la ardilla pidió ayuda a su amiga Karen y juntas extendieron una red en el suelo para capturar las bellotas que caen. Ludmila tiene hambre pero sabe que si retira la red ahora no tendrá muchos frutos y el trabajo habría sido en vano, Karen la convence de ser paciente ya que "todo funcionará bien". Para matar el tiempo deciden contar cuantas bellotas cayeron dentro de la red hasta el momento.

La red tiene forma de polígono convexo. Sus bordes son gruesos y cuando una bellota cae sobre ellos rebota hacia alguno de sus costados. Por eso no hay bellotas sobre el borde. El grosor de la red mide aproximadamente 10^{-8}.

Input: Tamaño (entero) del contorno externo de la red y coordenadas de sus vertices (reales) dados en sentido horario. Cantidad de bellotas (entero) y coordenadas de cada una de ellas en el suelo (reales).

Output: Cantidad de bellotas dentro de la red

Ejemplo 3 – Solución

Cuando los puntos de un polígono convexo están dados en sentido horario un posible método para chequear si un punto está o no adentro de él es chequear que esté en el semiplano "izquierdo" de todos sus lados. Ya que la región del polígono es la intersección de todos estos semiplanos.

Para chequear en que semiplano está un punto respecto a un segmento usamos el producto cruz. Queda en ustedes ver por qué.

He aquí una implementación tras haber almacenado el polígono en P:

```
int ans = 0;
scanf("%d", &m);
for(int i = 0; i < m; i++){
    punto T; bool in = true;
    scanf("%Lf %Lf", &T.x, &T.y);
    for(int j = 0; j < n; j++){
        punto A = P[j], B = P[(j+1)%n];
        punto AB = B - A, AT = T - A;
        if((AB^AT) < 0)in = false;
    }
    if(in)ans++;
}
```

Como corregir fallas en comparación – En reales

Comparar con certeza absoulta un par de reales almacenados como punto flotante es imposible.

Cuando los reales están muy cerca, las pequeñas imprecisiones inevitables hacen que sea imposible distinguir cual de los dos es mayor, o si son iguales.

En estos casos generalmente el problema nos da una garantía extra para ayudarnos a evitar esos casos borde.

Veamos el caso del ejemplo.

Como corregir fallas en comparación – En reales (cont.)

En el ejemplo 3 se nos dice que no hay puntos dentro del polígono muy cerca del borde. Por eso consideramos que un punto esta adentro del polígono solo si esta “muy adentro”.

En estos casos definiremos una constante que nos ayudará bastante:

```
const double EPS = 1e-9;
```

O sea: $\epsilon = 10^{-9}$

El exponente varía dependiendo del problema (en el caso ejemplo un exponente de -8 parece razonable) pero -9 es el valor usual.

De esta manera al preguntarnos si “un punto esta adentro” eliminamos posibles **falsos positivos,** en teoría haríamos más probable un **falso negativo**, pero por las condiciones del problema no sucederá.

Como corregir fallas en comparación – En reales (cont.)

Escribimos:

```
if((AB^AT) < EPS)in = false;
```

en lugar de

```
if((AB^AT) < 0)in = false;
```

El método siempre es agregar un $+\epsilon$ en algún lado de la comparación, de que lado hacerlo dependerá de las condiciones del problema.
Si el problema dijera que no hay bellotas cerca del contorno **por afuera** escribiríamos:

```
if((AB^AT)+ EPS < 0)in = false;
```

Esta práctica es muy útil y no se limita solo a Geometría, si no a cualquier problema en donde debas comparar flotantes.

Aunque el problema no te diera este tipo de garantías, se suele considerar que dos reales $\boldsymbol{a}$ y $\boldsymbol{b}$ son iguales si y solo si:

$$|a - b| < \epsilon$$

Como corregir fallas en comparación – En reales (cont.)

Esta consideración se puede extender para reescribir las siguientes expresiones:

$a < b$	lo escribimos	$a < b - \epsilon$
$a \leq b$	lo escribimos	$a < b + \epsilon$

Ésto es de suma utilidad la mayoría de las veces.
Sin embargo hay casos en los que no son suficientes estas consideraciones.
Para esos casos, vamos a intentar trabajar en enteros.

Ejemplo 4

Dulces ositos míos

Los hijos de mamá oso, Abel y Beto, pelean mucho, en general son muy tranquilos pero siempre que salen a pescar o cazar se pelean por el territorio. Mamá oso se cansó de retarlos al respecto, así que además de corregirlos está buscando una forma de que salgan a cazar sin pelear. Para eso va a definir una "zona de caza" para Abel y otra distinta para Beto.

Ambas zonas son círculos cerrados en el plano del bosque y los ositos solo tienen permitido cazar en sus respectivas zonas. Si en algún momento aparece una presa en ambas zonas habrá una pelea entre ellos.

Mamá oso quiere saber si las zonas que eligió aseguran que no habrá peleas.

Input: Vas a recibir el centro (X_i, Y_i) y el radio R_i de ambas zonas.

Output: Debés responder "Safe place" si éstás seguro de que no habrán peleas y "Thunder and rain" si no es así.

Como corregir fallas en comparación – En enteros

Lo mejor que podemos hacer si necesitamos comparar es evitar a toda costa comparar dos números reales. Ya que en la PC nuestra única forma de almacenarlos es como puntos flotantes, y vimos que comparar dos puntos flotantes es muy inconveniente.

Siempre que podamos reemplazar una comparación en reales por una comparación en enteros vamos a hacerlo, ya que las comparaciones en enteros serán siempre exactas, sin error.

Como corregir fallas en comparación – En enteros (cont.)

Estamos de acuerdo en que la respuesta en el ejemplo 4 se halla decidiendo si la suma de los radios de las zonas es o no es menor que la distancia entre los centros.
A simple vista podemos implementar lo siguiente:

```
scanf("%lld %lld %d", &A.x, &A.y, &ra);
scanf("%lld %lld %d", &B.x, &B.y, &rb);
ld dist = (B-A).modulo();
printf("%s\n",(dist > ra+rb ? "Safe place" : "Thunder and rain"));
```

Eso pide comparar dos flotantes y bien sabemos que eso puede dar error. En este caso, como ambos lados son positivos, podemos elevar ambos lados de la inecuación al cuadrado:

$$|d|^2 > (r_a + r_b)^2$$
$$d_x^2 + d_y^2 > (r_a + r_b)^2$$

Como corregir fallas en comparación – En enteros (cont.)

Visto eso podemos implementar:

```
scanf("%lld %lld %lld", &A.x, &A.y, &ra);
scanf("%lld %lld %lld", &B.x, &B.y, &rb);
ll x = A.x - B.x, y = A.y - B.y, r = ra+rb;
printf("%s\n",(x*x + y*y > r*r ? "Safe place" : "Thunder and rain"));
```

Donde usamos solo comparaciones en enteros, que nunca van a fallar.

Voy a hacer énfasis en que todas estas “mejoras” en la precisión se consiguen a partir del trabajo en papel.
En los problemas de Geometría es muy importante resolver el problema y estar seguro de ***qué hacer*** antes de implementar.

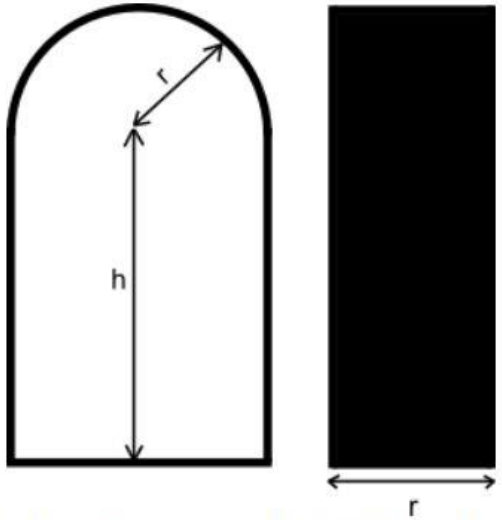
El *“tiempo fuera de la PC”* es **fundamental** en Geometría.

Problema desafío

C. Cupboard and Balloons

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

A girl named Xenia has a cupboard that looks like an arc from ahead. The arc is made of a semicircle with radius r (the cupboard's top) and two walls of height h (the cupboard's sides). The cupboard's depth is r, that is, it looks like a rectangle with base r and height $h + r$ from the sides. The figure below shows what the cupboard looks like (the front view is on the left, the side view is on the right).



Xenia got lots of balloons for her birthday. The girl hates the mess, so she wants to store the balloons in the cupboard. Luckily, each balloon is a sphere with radius $\frac{r}{2}$. Help Xenia calculate the maximum number of balloons she can put in her cupboard.

You can say that a balloon is in the cupboard if you can't see any part of the balloon on the left or right view. The balloons in the cupboard can touch each other. It is not allowed to squeeze the balloons or deform them in any way. You can assume that the cupboard's walls are negligibly thin.

https://codeforces.com/problemset/problem/342/C

Estructura fracción

Un último recurso para resolver un problema con reales en enteros es implementar la **"estructura fracción"**:

```
struct fraccion{
    ll num, den;
    fraccion(ll num = 0, ll den = 1): num(num), den(den){}
    fraccion operator+(fraccion a){
        fraccion ans(num*a.den+a.num*den, den*a.den);
        ans.normalize(); return ans;
    }
    fraccion operator*(fraccion a){
        fraccion ans(num*a.num, den*a.den);
        ans.normalize(); return ans;
    }
    void normalize(){
        if(den < 0){ num = -num; den = -den;}
        if(num == 0){ den = 1; return; }
        ll d = __gcd(abs(num), den);
        num /= d; den /= d;
    }
    bool operator<(fraccion a){ return num*a.den < a.num*den; }
};
```

Cuando los números reales con los que trabajamos son racionales podemos guardarlos como una fracción usando esta estructura.

Vemos que la comparación entre estructuras fracción es exacta.

Estructura fracción (cont.)

Podemos ver una función *normalize()* que convierte la fracción en su **fracción irreducible equivalente** con denominador positivo.

Usando este recurso podemos chequear equivalencia de fracciones miembro a miembro, y no tenemos necesidad de multiplicar.

Un inconveniente grande con usar esta estructura es que si realizás muchas operaciones entre *fracciones*, las representaciones pueden requerir numerador y denominador muy grandes y al comparar podemos obtener un overflow. Por eso no es muy recomendable usarlo a menos que el overflow este debidamente controlado.

Problema ejemplo

C. Mice problem

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Igor the analyst fell asleep on the work and had a strange dream. In the dream his desk was crowded with computer mice, so he bought a mousetrap to catch them.

The desk can be considered as an infinite plane, then the mousetrap is a rectangle which sides are parallel to the axes, and which opposite sides are located in points (x_1, y_1) and (x_2, y_2).

Igor wants to catch all mice. Igor has analysed their behavior and discovered that each mouse is moving along a straight line with constant speed, the speed of the i-th mouse is equal to (v_i^x, v_i^y), that means that the x coordinate of the mouse increases by v_i^x units per second, while the y coordinates increases by v_i^y units. The mousetrap is open initially so that the mice are able to move freely on the desk. Igor can close the mousetrap at any moment catching all the mice that are **strictly** inside the mousetrap.

Igor works a lot, so he is busy in the dream as well, and he asks you to write a program that by given mousetrap's coordinates, the initial coordinates of the mice and their speeds determines the earliest time moment in which he is able to catch all the mice. Please note that Igor can close the mousetrap only once.

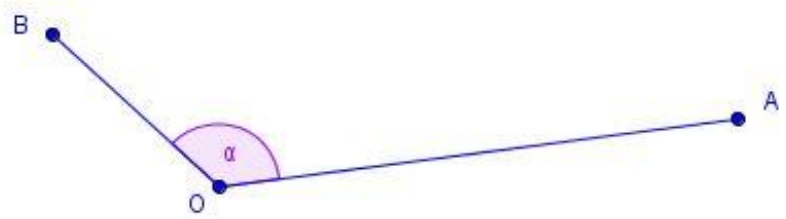
https://codeforces.com/problemset/problem/793/C

Implementación: https://codeforces.com/contest/793/submission/39324704

Ángulos

Amplitud de un ángulo

Sabemos que podemos definir un ángulo dados tres puntos.



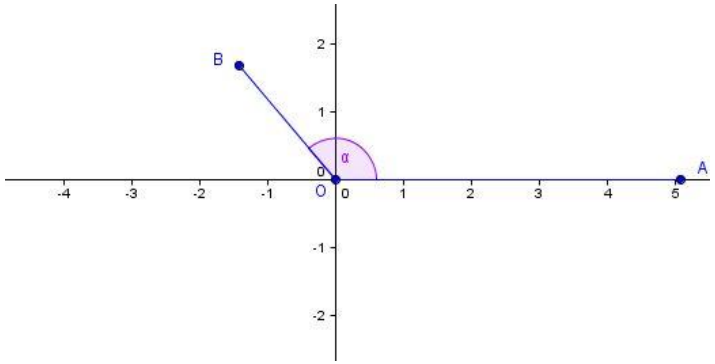
Popularmente, al referirse a amplitudes de ángulos se suele considerar el barrido más pequeño entre las semirrectas $\overrightarrow{OA}$ y $\overrightarrow{OB}$, sin importar el orden.

Para nosotros no.
La amplitud del ángulo es la amplitud barrida **antihorariamente desde $\overrightarrow{OA}$ hasta $\overrightarrow{OB}$**, en radianes. Resulta ser un real en el intervalo $[0, 2\pi)$

Hallar la amplitud de un ángulo

Nos planteamos el problema de hallar la amplitud de un ángulo dados los vértices O, A y B.

Para ayudarnos visualmente pensemos el ángulo centrado en el origen y con $\overrightarrow{OA}$ alineado al semieje positivo x:



Sabemos que para hallar la amplitud de un ángulo basta con conocer la **tangente** del ángulo y el **cuadrante** al que éste pertenece. Partiendo de esto existen muchas formas de trabajar algebraicamente para hallar la amplitud de un ángulo a partir de los vértices, busquemos una forma precisa de hacerlo.

Tangente de un ángulo

Dadas las coordenadas de los vértices.
¿Podemos hallar el **seno** y/o el **coseno** del ángulo?

¡Sí! Si recordamos las definiciones de producto escalar y de producto cruz ya que:

$$\overline{\boldsymbol{u}} \times \overline{\boldsymbol{v}} = |\overline{\boldsymbol{u}}|.|\overline{\boldsymbol{v}}|.\cos(\hat{u,v}) \Rightarrow \cos(\hat{u,v}) = \frac{\overline{\boldsymbol{u}} \times \overline{\boldsymbol{v}}}{|\overline{\boldsymbol{u}}|.|\overline{\boldsymbol{v}}|}$$

$$\overline{\boldsymbol{u}}\hat{}\overline{\boldsymbol{v}} = |\overline{\boldsymbol{u}}|.|\overline{\boldsymbol{v}}|.\sin(\hat{u,v}) \Rightarrow \sin(\hat{u,v}) = \frac{\overline{\boldsymbol{u}}\hat{}\overline{\boldsymbol{v}}}{|\overline{\boldsymbol{u}}|.|\overline{\boldsymbol{v}}|}$$

Pero si lo que queremos es la tangente no necesitamos hallar explicitamente el seno y el coseno, sino que podemos expresar:

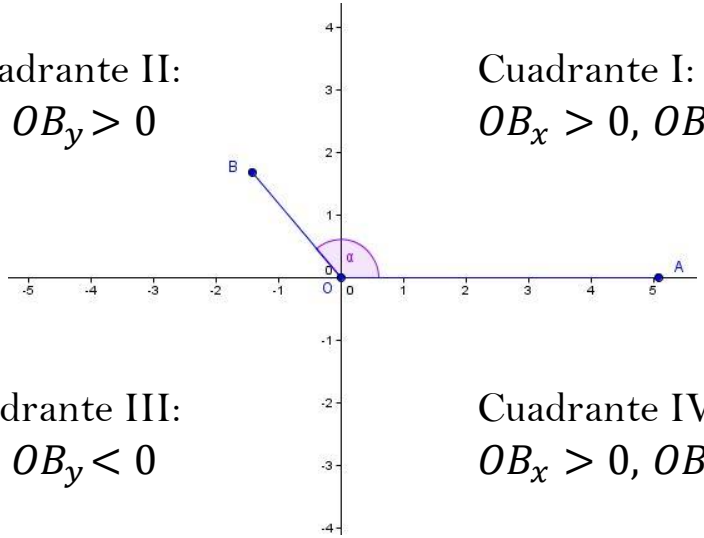
$$\tan(\hat{u,v}) = \frac{\sin(\hat{u,v})}{\cos(\hat{u,v})} = \frac{\overline{\boldsymbol{u}}\hat{}\overline{\boldsymbol{v}} \,/\, (|\overline{\boldsymbol{u}}|.|\overline{\boldsymbol{v}}|)}{\overline{\boldsymbol{u}} \times \overline{\boldsymbol{v}} \,/\, (|\overline{\boldsymbol{u}}|.|\overline{\boldsymbol{v}}|)} = \frac{\overline{\boldsymbol{u}}\hat{}\overline{\boldsymbol{v}}}{\overline{\boldsymbol{u}} \times \overline{\boldsymbol{v}}}$$

Cuadrante de un ángulo

Cuando tenemos el ángulo centrado en el origen con $\overrightarrow{OA}$ alineado con el semieje positivo x, es fácil ver que el "cuadrante" del ángulo esta determinado por las componentes de $\overrightarrow{OB}$, más precisamente, por los signos de estas.

Cuadrante II:
$OB_x < 0, OB_y > 0$

Cuadrante I:
$OB_x > 0, OB_y > 0$



Cuadrante III:
$OB_x < 0, OB_y < 0$

Cuadrante IV:
$OB_x > 0, OB_y < 0$

La observación que debemos hacer es que los signos de esas componentes $(\overline{OB}_x, \overline{OB}_y)$ son los mismos que los signos, respectivamente, del par $(\cos(\alpha), \sin(\alpha))$

Cuadrante de un ángulo (cont.)

Redujimos el problema de hallar el cuadrante a hallar los signos del seno y el coseno del ángulo.

Volvamos a las definiciones de producto escalar y producto cruz.

$\cos(\hat{u,v}) = \frac{\overline{u}\times\overline{v}}{|\overline{u}|.|\overline{v}|}$ implica que $\cos(\hat{u,v})$ y $(\overline{u} \times \overline{v})$ tienen el mismo signo.

$\sin(\hat{u,v}) = \frac{\overline{u}\hat{}\overline{v}}{|\overline{u}|.|\overline{v}|}$ implica que $\sin(\hat{u,v})$ y $(\overline{u}\hat{}\overline{v})$ tienen el mismo signo.

Llegamos a la conclusión de que nos basta ver los productos escalar y cruz para determinar el cuadrante de un ángulo.
Éste queda determinado por los signos de estos productos.

Hallar la amplitud de un ángulo (cont.)

Lo que acabamos de ver es suficiente para construir una implemetación linda que calcule la amplitud deseada.

Una posible implementación es:

```
const double PI = acos(-1);
double angle_imp(dot A, dot O, dot B){ // Respuesta en [0, 2*PI)
    dot OA = A - O, OB = B - O;
    ll Sin = OA^OB, Cos = OA*OB;
    double angle = atan((double)Sin / Cos);
    if(Cos < 0)angle += PI;
    if(angle < 0)angle += 2*PI;
    return angle;
}
```

Sin embargo, en C++, no es necesario realizar la implementación porque hay una función incorporada que calcula el angulo con buena precisión si le pasás el producto escalar y el producto cruz.

Hallar la amplitud de un ángulo (cont.)

La función $\boldsymbol{atan2l}()$ está pensada para recibir como parámetros dos números reales, el **seno** y el **coseno** de un ángulo.

Sin embargo, lo que importa de esos valores son **sus signos** y **la razón entre sus valores absolutos**, propiedades que se conservan si pasamos como parámetros el producto cruz y el producto escalar.

Esto es genial sobre todo cuando estamos trabajando con puntos a coordenadas enteras, ya que los productos se calculan sin error. Podemos esperar que internamente $\mathrm{atan2}l()$ funcione muy bien.

Podemos implementar:

```
double angle(punto A, punto O, punto B){
    punto OA = A - O, OB = B - O;
    return atan2l(OA^OB, OA*OB);
}
```

Nota importante: atan2() devuelve un ángulo en el intervalo $(-\boldsymbol{\pi}, \boldsymbol{\pi}]$

Comparar amplitudes de ángulos

Nos gustaría chequear si las amplitudes de dos ángulos son iguales, ¿Podemos hacerlo?

A simple vista parece que la respuesta es **"no"**, o que la respuesta es **"podemos compararlos usando ϵ"**, método con el cual tendríamos un poco de error y no una respuesta exacta.

Nos gustaría hacer una comparación exacta y para eso debemos hallar una forma de hacerlo usando números enteros.

Amplitud angular en enteros

Ya nos hemos convencido de que la amplitud de un ángulo dado por tres vértices a coordenadas enteras **puede determinarse por dos valores enteros**, el producto escalar y el producto cruz de sus *"lados"*.

¿Podemos guardar la amplitud de un ángulo como este par de enteros? Sí

Tras eso. ¿Cómo hacemos la comparación?

Recordemos *qué* propiedades del par definen al ángulo:

- El signo del producto escalar
- El signo del producto cruz
- La razón entre los productos

Si esas tres propiedades se comparten en ambos ángulos sus amplitudes son iguales.

Implementación

Una implementación completa de la comparación igualdad sería la siguiente:

```
int sgn(ll n){ return (n < 0 ? -1 : !!n); }
struct Ang{
    ll fst, snd;
    bool operator==(Ang T){
        if(sgn(fst) != sgn(T.fst))return false;
        if(sgn(snd) != sgn(T.snd))return false;
        if(fst*T.snd != snd*T.fst) return false;
        return true;
    }
};
```

La función $\boldsymbol{sgn}()$ nos dice el signo de un entero, devuelve: $\boldsymbol{-1}$ si es negativo, $\boldsymbol{1}$ si es positivo y $\boldsymbol{0}$ si es 0

Representación normalizada

En la implementación anterior necesitamos hacer una multiplicación para comparar. Esa operación nos podría llegar a molestar si el resultado excede lo almacenable por 64-bits.

Cuando trabajabamos con fracciones pudimos *"normalizar"* las fracciones para poder compararlas componente a componente. Aquí podemos hacer lo mismo.

Llevaremos un par a su "*par irreducible*", al igual que en la fracción, procurando conservar los signos intactos, de esta forma se conservan los signos y la razón entre las componentes.

Hecho esto podemos demostrar que todas las condiciones quedan verificadas al comparar los pares componente a componente.

Implementación

```
typedef pair<ll, ll> Ang;
void normalize(Ang &A){
    ll d = __gcd(abs(A.first), abs(A.second));
    A.first /= d; A.second /= d;
}
```

Si guardamos el ángulo explicitamente como un ***pair<ll, ll>*** la comparación de igualdad entre pares es inmediata, ya que está definida en C++ como igualdad componente a componente.

Comparación de desigualdad

Cuando queremos ordenar ángulos por amplitud, si están guardados como par, se nos complica evitar hacer la multiplicación de dos componentes.

Por eso una observación importante es que cuando sabés que dos ángulos no son iguales, podemos esperar que sus representaciones como número real no estén muy cerca, por lo que perdemos el miedo a compararlas usando ese valor. Podemos escribir:

```
typedef long double ld;
bool menor(Ang A, Ang B){
    if(A == B)return false;
    ld alpha = atan2l(A.first, A.second);
    ld beta = atan2l(B.first, B.second);
    if(alpha < 0)alpha += 2*PI;
    if(beta < 0)beta += 2*PI;
    return alpha < beta;
}
```

Problema ejemplo – Enunciado

Bosque paradisíaco

Érase una vez un lejano bosque paradisíaco donde el pasto es verde y las figuras geométricas son bonitas.
Un día la familia del circo llegó al bosque.
El director del circo les ordenó a los payasos levantar y preparar la carpa del circo donde harán su espectáculo. Para no contrastar con esta ciudad donde la belleza geometrica es muy importante les indicó que la carpa del circo debe verse, desde arriba, como un polígono regular.
Los payasos ya ubicaron los postes y antes de acomodar la carpa quieren saber si hicieron bien su trabajo.
¿Podrás ayudarlos?
Input: Recibirás las coordenadas (enteras) de los postes en el plano, en orden.
Output: Debés responder "SI" si estan bien ubicados o "NO" si no.

Problema ejemplo – Solución

Una vez leído vemos que en el problema recibimos las coordenadas de los vértices de un polígono y debemos decidir si es regular o no.

De teoría sabemos que un polígono es regular si y solo si se cumplen las siguientes dos condiciones:

- Todos sus lados son iguales
- Todos sus ángulos internos son iguales

Primero chequeamos la igualdad de las longitudes de los lados. Ésto es tarea sencilla, pero **recordemos** hacer la comparación **en enteros** y no en flotantes.

Y segundo chequeamos la igualdad de amplitudes de los ángulos. Podemos hacerlo **en enteros** con el método que acabamos de ver.

Problema ejemplo – Implementación

Estructura y variables globales:

```
struct punto{
    ll x, y;
    punto(ll x = 0, ll y = 0): x(x), y(y){}
    punto operator-(punto a){
        return punto(x-a.x, y-a.y);
    }
    ll operator*(punto a){ return x*a.x + y*a.y; }
    ll operator^(punto a){ return x*a.y - y*a.x; }
};
int n;
vector<punto> P;
```

Chequeo lados iguales:

```
ll sq_dist(punto a, punto b){
    ll x = a.x - b.x, y = a.y - b.y;
    return x*x + y*y;
}
bool checksides(){
    ll D = sq_dist(P[0], P[1]);
    forn(i, n){
        ll ND = sq_dist(P[i], P[(i+1)%n]);
        if(D != ND)return false;
    }
    return true;
}
```

Chequeo ángulos iguales:

```
typedef pair<ll, ll> pll;
pll angle(punto A, punto O, punto B){
    punto OA = A - O, OB = B - O;
    return {OA^OB, OA*OB};
}
void normalize(pll &A){
    ll d = __gcd(abs(A.fst), abs(A.snd));
    A.fst /= d; A.snd /= d;
}
vector<pll> ANG;
bool checkangles(){
    ANG.clear();
    forn(i, n){
        punto A = P[i], B = P[(i+1)%n], C = P[(i+2)%n];
        pll T = angle(A, B, C);
        normalize(T);
        ANG.pb(T);
    }
    forn(i, n)if(ANG[i] != ANG[0])return false;
    return true;
}
```

Main:

```
int main(){
    scanf("%d", &n);
    P.resize(n);
    forn(i, n)scanf("%lld %lld", &P[i].x, &P[i].y);
    bool anda = checksides() && checkangles();
    printf("%s\n", (anda ? "SI" : "NO"));
    return 0;
}
```

Usos generales del ángulo como par de enteros

Otro uso importante de la comparación exacta de ángulos es que, entre otras cosas, ahora puedo guardar ángulos dentro de un set<> o un map<> y tengo el poder de contar ángulos distintos.

Otro uso interesante, que es más importante como concepto que como problema, es que podemos decidir sencillamente cuando dos triángulos son semejantes.
Si no saben lo que significa “triangulos semejantes” pueden ir al siguiente link:
https://es.wikipedia.org/wiki/Semejanza_(geometr%C3%ADa)

Problema desafío

Como ejercicio de tarea intenten resolver el problema en el siguiente link:

Codeforces – Contest 1017 – Problema E

https://codeforces.com/problemset/problem/1017/E

Eso es todo

¡Gracias por escuchar!