



**DEPARTAMENTO  
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

# Monografía: *Geometría básica en 2D*

Programación Competitiva II  
2do cuatrimestre 2023

Integrante	LU	Correo electrónico
Lucas Lucero	771/19	lucas.e.lucero@hotmail.com



**Facultad de Ciencias Exactas y Naturales**  
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

# Índice

<b>1. Introducción</b>	<b>5</b>
1.1. ¿Qué es la geometría? . . . . .	5
1.2. Un poco de historia . . . . .	5
<b>2. Motivación</b>	<b>6</b>
<b>3. Puntos/Vectores</b>	<b>7</b>
3.1. Puntos . . . . .	7
3.2. Vectores . . . . .	8
3.3. Suma y Resta entre un punto y un número (Traslación de un punto) . . . . .	10
3.4. Suma y Resta de Vectores . . . . .	11
3.5. Norma/Módulo/Longitud de un vector . . . . .	13
3.6. Distancia entre 2 puntos/vectores . . . . .	14
3.7. Producto escalar entre un vector y un número (Escala) . . . . .	15
3.8. Producto escalar entre dos vectores . . . . .	16
3.9. Proyección Ortogonal . . . . .	18
3.10. Contraproyección . . . . .	19
3.11. Rotación horaria y antihoraria de puntos/vectores . . . . .	20
3.12. Producto vectorial (o <i>producto cruz</i> ) . . . . .	21
3.13. Área de un triángulo . . . . .	22
<b>4. Rectas/Líneas</b>	<b>24</b>
4.1. Definición y representación . . . . .	24
4.2. Determinar si un punto está en una recta . . . . .	27
4.3. Ordenar puntos de una recta . . . . .	28
4.4. Distancia entre un punto y una recta . . . . .	30
4.5. Representación $ax + by + c = 0$ . . . . .	32
4.6. Rectas y semiplanos . . . . .	33
4.7. Problema Codeforces 498A: Crazy Town . . . . .	34
4.8. Determinar en qué semiplano se encuentra un punto respecto de una recta (paramétrica) . . . . .	35
4.9. Problema CSES: Point Location Test . . . . .	36
4.10. Determinar si dos rectas son paralelas . . . . .	37
4.11. Determinar en qué punto se intersecan dos rectas . . . . .	38
<b>5. Segmentos</b>	<b>39</b>

5.1. Definición y representación . . . . .	39
5.2. Determinar si un punto se encuentra en un segmento . . . . .	40
5.3. Determinar si dos segmentos se intersecan . . . . .	41
5.4. Problema CSES: Line Segment Intersection . . . . .	43
5.5. Determinar en qué punto se intersecan dos segmentos . . . . .	44
5.6. Punto medio . . . . .	44
<b>6. Polígonos</b>	<b>45</b>
6.1. Definición y representación . . . . .	45
6.2. Otras definiciones . . . . .	45
6.3. Perímetro de un polígono . . . . .	47
6.4. Área de un polígono . . . . .	48
6.5. Problema CSES: Polygon Area . . . . .	49
6.6. Determinar si un punto se encuentra en un polígono . . . . .	50
6.7. Problema CSES: Point in Polygon . . . . .	53
<b>7. Círculos</b>	<b>54</b>
7.1. Definición y representación . . . . .	54
7.2. Problema Codeforces 33D: Knights . . . . .	55
7.3. Encontrar un círculo a partir de 3 puntos . . . . .	58
<b>8. Comentarios finales</b>	<b>60</b>
8.1. Más aplicaciones prácticas de la geometría computacional . . . . .	60
8.2. Más ejercicios para practicar . . . . .	60
8.3. Algunos tópicos para profundizar más en la geometría computacional . . . . .	61
8.4. AGM Euclídeo con Triangulación de Delaunay . . . . .	62
8.4.1. Diagrama de Voronoi . . . . .	63
8.4.2. Triangulación de Delaunay . . . . .	64
8.4.3. Dualidad entre los Diagramas de Voronoi y las Triangulaciones de Delaunay	64
8.4.4. Utilidad . . . . .	65
8.4.5. Motivación y Bibliografía relacionada . . . . .	65
<b>9. Bibliografía</b>	<b>66</b>
<b>10. Apéndice (código de los ejercicios)</b>	<b>67</b>
10.1. Problema Codeforces 498A: Crazy Town . . . . .	67
10.2. Problema CSES: Point Location Test . . . . .	68
10.3. Problema CSES: Line Segment Intersection . . . . .	70

10.4. Problema CSES: Polygon Area . . . . .	72
10.5. Problema CSES: Point in Polygon . . . . .	73
10.6. Problema Codeforces 33D: Knights . . . . .	76

# 1. Introducción

## 1.1. ¿Qué es la geometría?

La geometría es una rama de la matemática que se ocupa del estudio de las propiedades de las figuras en el plano o el espacio, incluyendo: puntos, rectas, planos, paralelas, perpendiculares, curvas, superficies, polígonos, etc.).

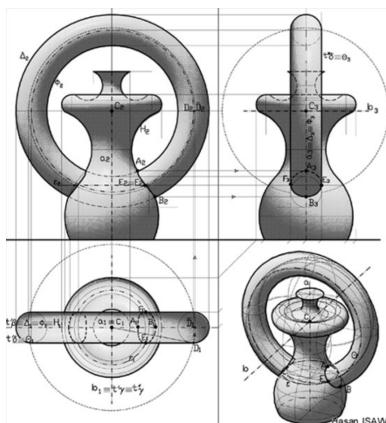
El campo de estudio de la geometría es bastante amplio, y ofrece distintas *subáreas* de especialización. Hay muchos aspectos de ella en las cuales se puede profundizar, y es ahí cuando notamos las sutilezas y características particulares de cada caso.

En este sentido, dado que puede ser muy abrumador al comienzo no saber por dónde empezar, en este apunte trabajaremos específicamente en 2 dimensiones. Esto simplifica la teoría subyacente, los problemas y sus soluciones, y además facilita la comprensión de los temas. Sin embargo, esto no quiere decir que no existan tópicos avanzados, ni que estemos severamente limitados desde el punto de vista competitivo, ya que en la mayoría de competencias, los problemas de geometría que aparecen, suelen ser en 2D, y no en 3D (o más dimensiones). Así, lo que ocurre es que con estudiar la geometría para 2 dimensiones, en realidad estamos cubriendo la mayoría de los casos (aunque no todos por supuesto, pero eso escapa del objetivo de este apunte, que es de servir como introducción).

## 1.2. Un poco de historia

La geometría es la base teórica de la geometría descriptiva o del dibujo técnico.

- La *geometría descriptiva* es un conjunto de técnicas geométricas que permite representar el espacio tridimensional, sobre una superficie bidimensional.



- Sus orígenes se remontan a la solución de problemas concretos relativos a medidas.

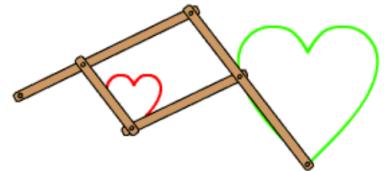
- También da como fundamento a instrumentos como el compás, el teodolito y el pantógrafo:



Compás



Teodolito



Pantógrafo

Teodolito: es un instrumento de medición mecánico-óptico que se utiliza para obtener ángulos verticales y horizontales, ya que tiene mucha precisión. Si se combina con otras herramientas auxiliares, puede medir distancias y desniveles.

Pantógrafo: es un mecanismo basado en las propiedades de los paralelogramos. Este instrumento tiene unas varillas conectadas de tal manera, que se pueden mover respecto de un punto fijo, que actúa como pivote. Se ideó originalmente para reproducir de forma manual dibujos originales a distinta escala.

## 2. Motivación

En Programación Competitiva:

- Siempre aparece un problema de geometría en ICPC (Competitive Programming 4, de los hermanos Halim):

No	Category	In This Book	Frequency
1.	Ad Hoc	Section 1.4-1.6	1-2
2.	(Heavy) Data Structure	Chapter 2	0-1
3.	Complete Search (Iterative/Recursive)	Section 3.2++	1-2
4.	Divide and Conquer	Section 3.3	0-1
5.	Greedy (the non-classic ones)	Section 3.4	1
6.	Dynamic Programming (the non-classic ones)	Section 3.5++	1-2
7.	Graph (except Network Flow/Graph Matching)	Chapter 4	1
8.	Mathematics	Chapter 5	1-2
9.	String Processing	Chapter 6	1
10.	Computational Geometry	Chapter 7	1
11.	Some Harder/Rare/Emerging Trend Problems	Chapter 8-9	2-3
Total in Set is usually $\leq 14$			10-17

Según regionales recientes de ICPC en Asia

- Si bien en la IOI no aparecen problemas de geometría en 3D o más dimensiones, si aparecen problemas en 2D (IOI Syllabus 2023).
- La mayoría de los problemas de geometría que aparecen en competencias en general, son en 2D.

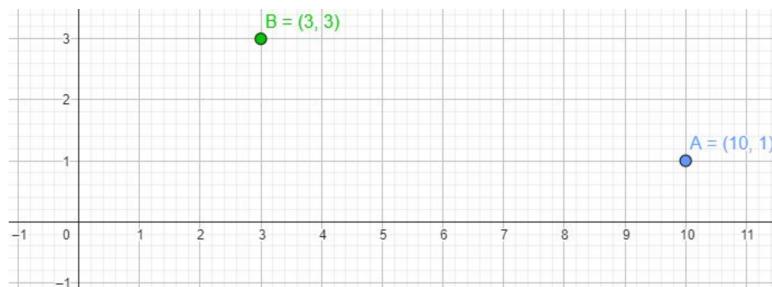
En la práctica:

- Se utiliza también en otras áreas además de la computación, como en física aplicada, mecánica, arquitectura, geografía, cartografía, astronomía, náutica, topografía, balística, etc., y es útil en la preparación de diseños e incluso en la fabricación de artesanía (veremos ejemplos de esto a lo largo del apunte, y también al final).

## 3. Puntos/Vectores

### 3.1. Puntos

Vamos a trabajar con puntos en el plano cartesiano, es decir en  $\mathbb{R}^2$ .



Notar que los puntos son pares **ordenados**, por lo que por ejemplo  $(1, 2) \neq (2, 1)$ .

En C++, los vamos a representar como:

```
struct Punto{
    int x; // coordenada x
    int y; // coordenada y
    Punto(int x = 0, int y = 0) : x(x), y(y){} // Constructor
};
```

Algo sobre lo que se va a enfatizar en varias partes de este apunte, es sobre tener cuidado con los errores numéricos, y el aprovechamiento de los tipos de datos exactos. Por eso, conviene evitar usar representaciones de números con punto flotante (como `float`, `double`, etc.) siempre que se pueda, y en cambio usar tipos de datos como `int`, `long long`, etc.

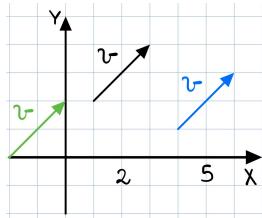
Lo mismo vale para las operaciones que no devuelven valores exactos. En caso de no poder evitarlas, conviene aplicarlas lo más al final posible, para reducir el error acumulado durante los cálculos.

## 3.2. Vectores

Un **vector** es una entidad que tiene un largo/longitud, una dirección, y un sentido. Intuitivamente, es una flecha en el plano.

### Vector libre

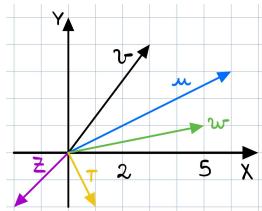
En su forma más pura y teórica, un vector **no está ligado a una posición específica fija**, sino que podemos moverlo por todo el plano, y sigue siendo el mismo vector, siempre y cuando mantenga su dirección, longitud y sentido. En el siguiente ejemplo, a pesar de estar en distintas posiciones, el vector  $v$  es siempre el mismo:



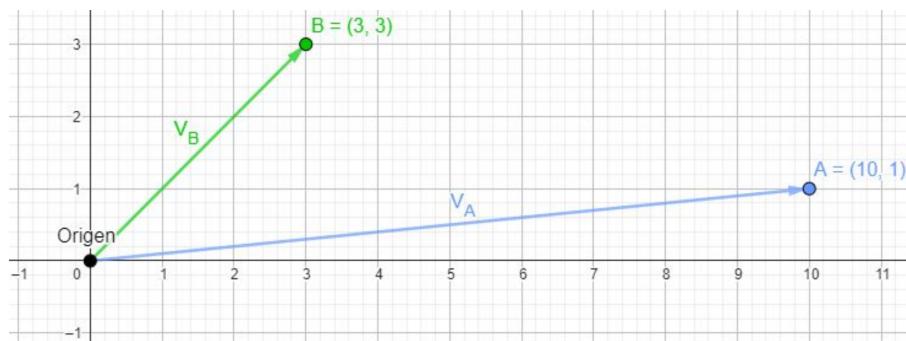
### Vector posicional

En muchas aplicaciones prácticas, especialmente en programación y física, se habla de vectores posicionales. Estos vectores **parten del origen** y apuntan a un punto específico. En este contexto, un vector representa un punto en el espacio (podemos hablar indistintamente de puntos y vectores), y **si cambiamos las coordenadas del punto, efectivamente estamos cambiando el vector**.

Ejemplos (los vectores tienen distinto nombre y color para enfatizar que son distintos):



Retomando lo dicho anteriormente, y aprovechando la visión de vector posicional, podemos decir que cada Punto, tiene un **vector** asociado desde el origen hasta dicho Punto:



Estos vectores se definen a través de sus coordenadas, al igual que los Puntos en sí, por lo que  $V_A = (10, 1)$  y  $V_B = (3, 3)$ .

Matemáticamente y en C++, estos vectores se definen igual que los Puntos, por lo que la representación queda:

```
struct Vec{  
    int x; // coordenada x  
    int y; // coordenada y  
    Vec(int x = 0, int y = 0) : x(x), y(y){} // Constructor  
};
```

Detalles:

- Aunque técnicamente podemos definir un **struct** con nombre ‘vector’, es recomendable evitarlo para no crear confusiones con la clase ‘vector’ de la biblioteca estándar. Por eso usamos el nombre ‘Vec’.
- Podemos usar indistintamente **Punto** o **Vec**. Podemos elegir por usar siempre **Punto** o usar siempre **Vec**, o usar uno u otro según el contexto, y la semántica que busquemos.

¿Por qué querríamos usar Vectores en vez de Puntos?

→ Porque los vectores nos permiten realizar operaciones útiles (como por ejemplo la resta de vectores), que tienen una representación geométrica. Esto ayuda mucho a la hora de resolver ejercicios en papel, ya que facilita visualizar lo que estamos haciendo, para después programarlo.

Dichas operaciones, incluyen:

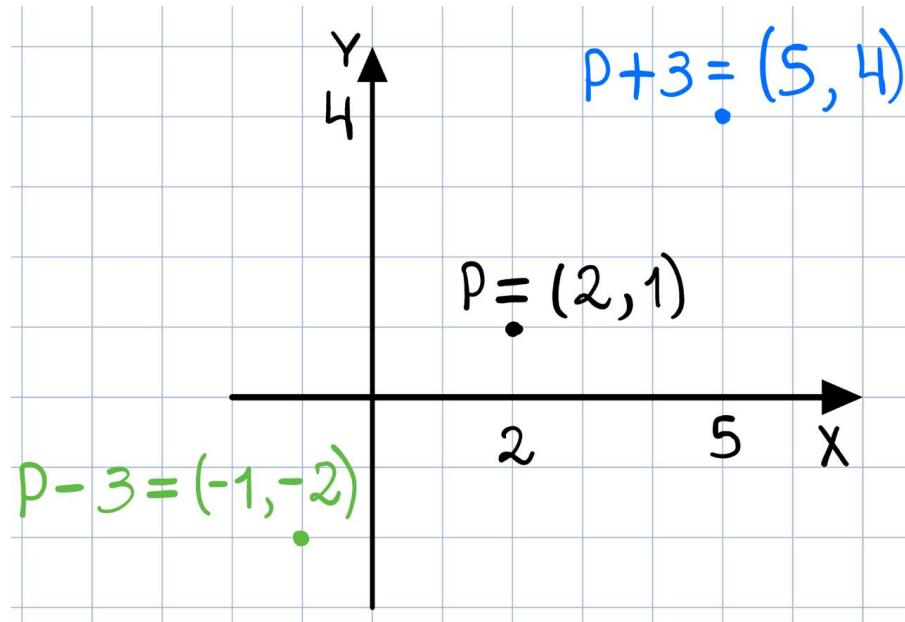
- Suma y resta de vectores.
- Producto escalar (producto punto, o  $\cdot$ ).
- Producto vectorial (producto cruz, o  $\times$ ).
- Longitud/norma de un vector.
- Distancia entre vectores.
- Escalas.
- etc.

### 3.3. Suma y Resta entre un punto y un número (Traslación de un punto)

La suma entre un punto  $p = (x_1, x_2)$  y un número (escalar)  $k$ , se define como:

$p + k = (x_1 + k, x_2 + k)$ . La resta es análoga.

De esta manera, esencialmente estamos moviendo el punto sobre el cual realizamos la operación.  
Ejemplos:



Implementación en C++:

```
struct Vec{
    // Coordenadas (x, y)
    int x, y;

    (...)

    // Traslaciones (suma o resta de un punto y un número)
    Vec operator+(int k) {return Vec(x + k, y + k);}
    Vec operator-(int k) {return Vec(x - k, y - k);}
};
```

### 3.4. Suma y Resta de Vectores

La suma y resta de vectores, **matemáticamente** se realiza componente a componente. Usando los vectores del ejemplo anterior, tenemos que:

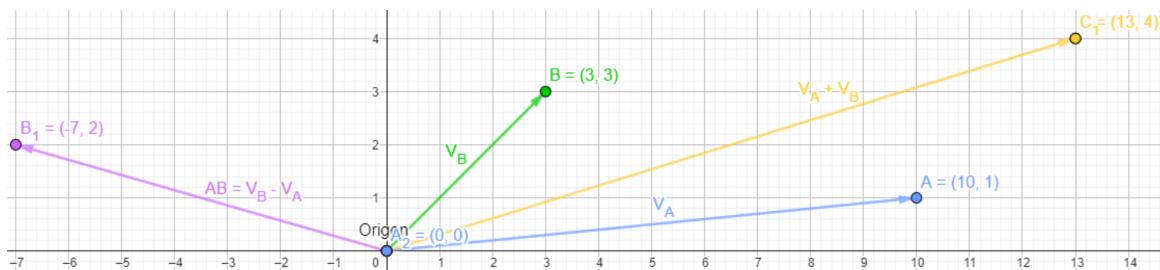
Dados  $V_A = (10, 1)$  y  $V_B = (3, 3)$ :

- $C = V_B + V_A = (3 + 10, 3 + 1) = (13, 4)$ .
- $AB = V_B - V_A = (3 - 10, 3 - 1) = (-7, 2)$ .

En el plano, esas operaciones se representan así:

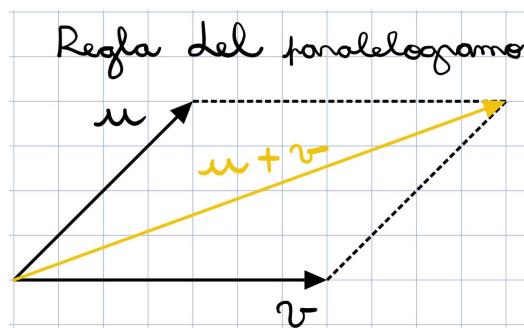


Los respectivos puntos/vectores de las operaciones, desde el origen:



**Geométricamente:**

- $V_B - V_A$  es un vector que va desde  $A$  hasta  $B$ .
- Para calcular  $V_B + V_A$  usamos la regla del paralelogramo:



En C++, podemos implementar esas operaciones de la siguiente manera:

```
struct Vec{
    int x, y; // Coordenadas (x, y)

    Vec(int x = 0, int y = 0) : x(x), y(y) {} // Constructor

    // Suma
    Vec operator+(Vec q) {return Vec(x + q.x, y + q.y);}

    // Resta
    Vec operator-(Vec q) {return Vec(x - q.x, y - q.y);}
};
```

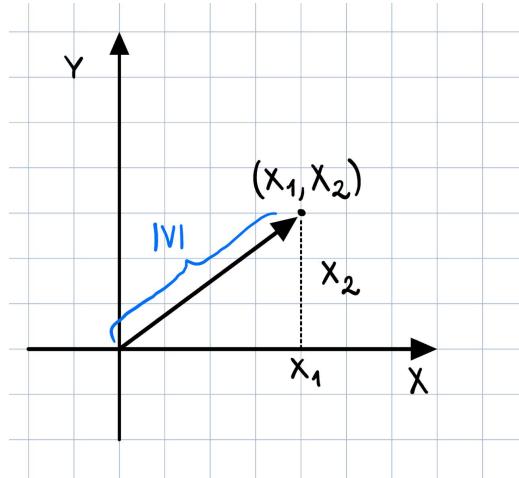
De esta forma, en nuestro código podemos usar los operadores de forma **infija**:

```
int main(){
    Vec A(5, 4), B(10, 10);
    Vec AB = B - A;
    cout << AB.x << AB.y << endl;
}
>> 5 6
```

### 3.5. Norma/Módulo/Longitud de un vector

La longitud de un vector (también se le dice *módulo* o *norma*) es la distancia al origen del punto asociado al vector, y se calcula haciendo Pitágoras sobre sus componentes:

Sea  $V = (x_1, x_2)$ . La longitud de  $V$  se define como  $|V| = \sqrt{x_1^2 + x_2^2}$ .



En C++, se implementa como:

```
struct Vec{
    int x, y; // Coordenadas (x, y)

    Vec(int x = 0, int y = 0) : x(x), y(y) {} // Constructor

    double norm() {return sqrt(x * x + y * y);} // Norma/Longitud/Módulo
};

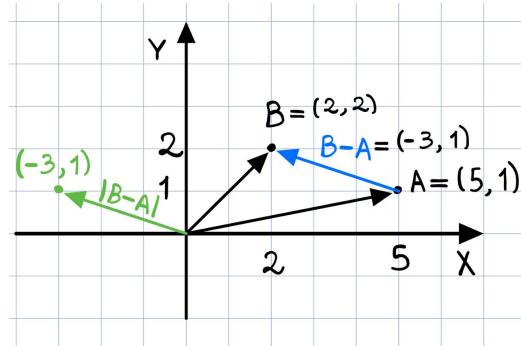
// Ejemplo
int main(){
    Vec A(4, 3);
    cout << A.norm() << endl;
}

>>  $\sqrt{4^2 + 3^2} = \sqrt{16 + 9} = \sqrt{25} = 5.000000$ 
```

### 3.6. Distancia entre 2 puntos/vectores

Dados dos puntos  $A$  y  $B$ , la distancia (Euclíadiana) entre  $A$  y  $B$  se define como  $d(A, B) = |B - A|$ .

Geométricamente, estamos haciendo lo siguiente:



Notar que  $d(A, B) = d(B, A)$

En C++, se implementa como:

```
double dist(Vec A, Vec B){  
    Vec AB = B - A;  
    return AB.norm();  
}
```

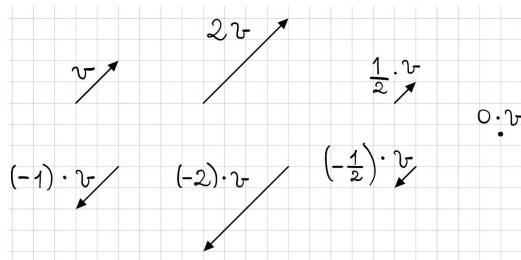
### 3.7. Producto escalar entre un vector y un número (Escala)

El producto escalar entre un vector  $v = (x_1, y_1)$  y un número  $k$  llamado escalar, se define como:  
 $k * v = (x_1 * k, y_1 * k)$ .

#### Interpretación geométrica

Cuando multiplicamos un vector por un escalar, **NO** estamos alterando su dirección, pero sí podemos alterar su longitud/norma/módulo, y también es posible alterar su sentido, según si  $k$  es negativo o no:

- Si  $k > 1$ , estamos aumentando su longitud. Intuitivamente, estamos “alargando” la flecha.
- Si  $0 < k < 1$ , estamos comprimiendo/acortando el vector.
- Si  $k = 0$ , el vector se colapsa y se convierte en el vector 0, es decir, un punto en el origen.
- Si  $k < 0$ , además de poder comprimir o estirar el vector, se invierte el sentido de la flecha.



Notar que si  $k = 1$ , el vector no cambia

Implementación en C++:

```
struct Vec{  
    // Coordenadas (x, y)  
    int x, y;  
  
    (...)  
  
    // Escala (producto entre un vector y un número)  
    Vec operator*(int k) {return Vec(x * a, y * a);} };
```

Tiene sus aplicaciones prácticas:

- Redimensionar o escalar gráficos en programas de diseño o juegos.
- En física, para cambiar unidades (por ejemplo, de metros a centímetros).
- En análisis de datos, para normalizar o estandarizar valores.

### 3.8. Producto escalar entre dos vectores

El producto escalar (o *producto punto*) entre dos vectores  $u = (x_1, y_1)$  y  $v = (x_2, y_2)$ , es un número que se define como:

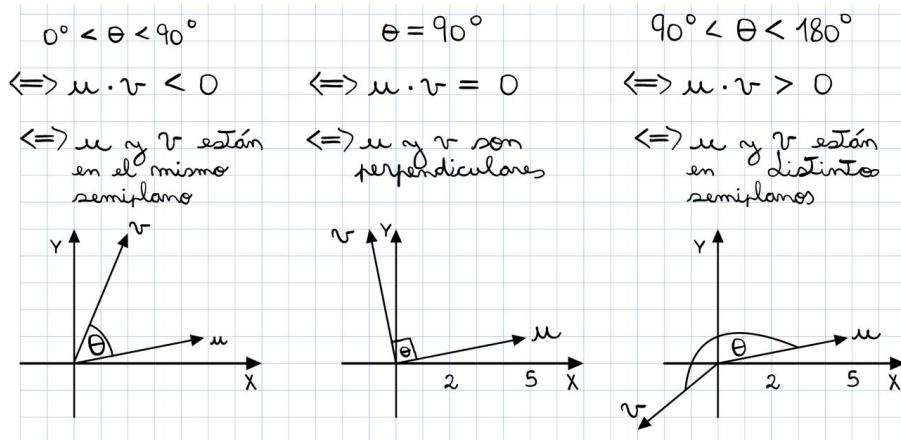
$u \cdot v = x_1 * x_2 + y_1 * y_2 = |u| * |v| * \cos \theta$ , donde  $\theta$  es el ángulo más chico entre  $u$  y  $v$  (este ángulo estará entre 0 y 180 grados, o lo que es lo mismo, 0 o  $\pi$  radianes).

A nivel código, no conviene usar la segunda definición para hacer el producto escalar entre dos vectores, ya que podemos llegar a introducir valores que **NO** son enteros, los cuales nos pueden llevar a acarrear un error de precisión, que resulte en un Wrong Answer.

Por el contrario, para hacer el producto escalar entre dos vectores en el código, vamos a utilizar **la primera definición, que nos permite operar directamente con las componentes de los vectores de forma segura**, sin arrastrar error (suponiendo que las componentes sean enteras).

La segunda definición la podemos utilizar a la hora de pensar y resolver los problemas en papel, ya que sirve para entender las relaciones entre vectores a nivel conceptual, y los ángulos comprendidos entre ellos, además de poder operar con los mismos.

Antes de pasar a la implementación, vale la pena mencionar algunas propiedades y conclusiones que podemos sacar, a partir del producto escalar entre dos vectores  $u$  y  $v$  no nulos:



Pedimos que  $u$  y  $v$  sean no nulos porque:

- El producto escalar entre cualquier vector con un vector nulo (vector con componentes  $(0, 0)$ ), siempre será 0, independientemente de la orientación o la norma del otro vector.
- El ángulo entre un vector, y un vector nulo no está bien definido.

Más propiedades:

- $\theta = 0^\circ \Leftrightarrow u \cdot v = |u| * |v| * 1 \Leftrightarrow u$  y  $v$  son **paralelos** y apuntan en el **mismo** sentido.
- $\theta = 180^\circ \Leftrightarrow u \cdot v = |u| * |v| * (-1) \Leftrightarrow u$  y  $v$  son **paralelos** y apuntan en **distinto** sentido.

En C++, lo implementamos como:

```
struct Vec{
    // Coordenadas (x, y)
    int x, y;

    (...)

    // Producto escalar entre dos vectores
    int operator*(Vec v) {return x * v.x + y * v.y;}

};

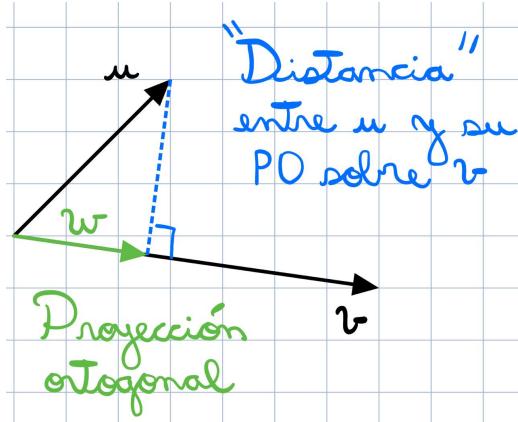
int main(){
    Vec A(1, 2), B(3, 4);
    cout << A * B << endl;

}
>> 11
```

### 3.9. Proyección Ortogonal

Intuitivamente, dados dos vectores  $u$  y  $v$ , la *proyección ortogonal* de  $u$  sobre  $v$ , es un nuevo vector  $w$ , que es la “sombra” de  $u$  que está en la dirección de  $v$  (ver imagen abajo).

Esta proyección, se obtiene dibujando una línea **perpendicular** desde la punta de  $u$  hasta  $v$



La longitud de esa proyección ortogonal, es un escalar, y se representa como  $|u| * \cos \theta$ , donde  $\theta$  es el ángulo más chico entre  $u$  y  $v$ .

#### Relación con el Producto Escalar entre dos vectores

El producto escalar entre dos vectores  $u$  y  $v$ , se puede interpretar como la longitud de la proyección ortogonal de  $u$  sobre  $v$ , multiplicada por la longitud de  $v$  (es la definición que ya habíamos visto, pero interpretada de otra manera):

$$u \cdot v = |u| \cos \theta |v|.$$

El vector  $w$  se obtiene de la siguiente manera:

$$w = \frac{u \cdot v}{|v|^2} v.$$

Notar que  $u \cdot v$  es el producto escalar de los vectores  $u$  y  $v$ , y esto da como resultado un número escalar.

Por otro lado,  $|v|^2$  es también un escalar.

$\frac{u \cdot v}{|v|^2}$  es la división de dos escalares, por lo que el resultado es un escalar.

$w = \frac{u \cdot v}{|v|^2} v$ . Finalmente, multiplicamos el escalar obtenido en el paso anterior, por  $v$ , por lo que obtenemos un nuevo vector, que resulta de escalar el vector  $v$  por ese factor escalar.

Implementación en C++:

```
struct Vec{
    // Coordenadas (x, y)
    int x, y;

    // Producto escalar entre dos vectores
    int operator*(Vec v) {return x * v.x + y * v.y;}

    // Escala (producto entre un vector y un número)
    Vec operator*(int k) {return Vec(x * k, y * k);}

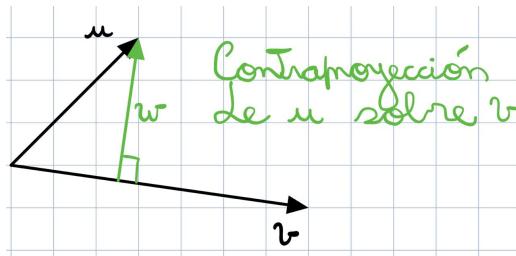
    // En vez de usar la función 'norm' que definimos antes, y elevar
    // al cuadrado el resultado, directamente calculamos el cuadrado de la
    // norma, es decir, no aplicamos la raíz cuadrada. Esto lo hacemos para
    // evitar introducir errores de precisión
    int sq_norm() {return x * x + y * y;}
};

// Calcula la proyección ortogonal de u sobre v
Vec proyeccion_ortogonal(Vec u, Vec v) {
    double factor_escalar = (u * v) / v.sq_norm(); // u . v / |v|^2

    // Multiplicamos el vector 'v' por el factor escalar
    return v * factor_escalar;
}
```

### 3.10. Contraproyección

La contraproyección  $u$  sobre  $v$  es el vector  $w$  que resulta de la diferencia entre  $u$  y  $\text{proy}(u, v)$ . Es decir,  $w = \text{contra\_proy}(u, v) = u - \text{proy}(u, v)$ .



Implementación en C++:

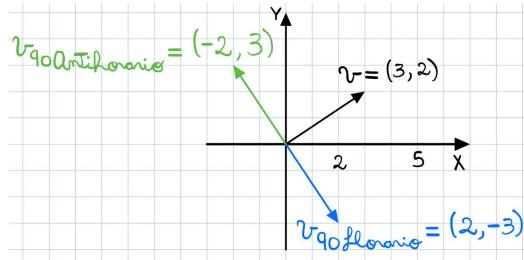
```
// Calcula la contraproyección de u sobre v
Vec contraproyeccion(Vec u, Vec v) {
    return u - proyeccion_ortogonal(u, v);
}
```

### 3.11. Rotación horaria y antihoraria de puntos/vectores

Dado el vector  $v = (x, y)$ , si queremos rotarlo  $90^\circ$  en sentido **horario**, entonces el vector resultante será  $v_{90\text{Horario}} = (y, -x)$ .

Por otro lado, dado el mismo vector  $v$ , el vector resultante de rotarlo  $90^\circ$  en sentido **antihorario**, es  $v_{90\text{Antihorario}} = (-y, x)$ .

Visualmente:



Implementación en C++:

```
struct Vec{
    // Coordenadas (x, y)
    int x, y;

    // Rotación 90° en sentido horario
    Vec rotar90Horario() {
        return Vec(y, -x);
    }

    // Rotación de 90° en sentido antihorario
    Vec rotar90Antihorario() {
        return Vec(-y, x);
    }
};
```

### 3.12. Producto vectorial (o *producto cruz*)

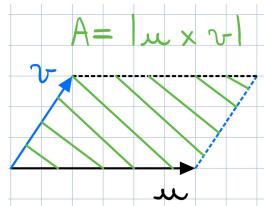
Matemáticamente, el producto vectorial está definido para  $\mathbb{R}^2$ , donde dados dos vectores  $u$  y  $v$  en  $\mathbb{R}^3$ , el producto vectorial es un tercer vector  $w$  que es perpendicular a  $u$  y a  $v$ , y cuya norma es el área del paralelogramo formado por  $u$  y  $v$ . Esta definición **no tiene un análogo directo** en  $\mathbb{R}^2$ , porque en  $\mathbb{R}^2$  no hay un "tercer eje" en el que pueda apuntar el resultado.

Sin embargo, si bien el producto vectorial como tal no tiene sentido en  $\mathbb{R}^2$ , le podemos dar una interpretación que es útil en Programación Competitiva:

Si tenemos dos vectores en  $\mathbb{R}^2$ ,  $u = (u_1, u_2)$  y  $v = (v_1, v_2)$ , podemos "inventar" un producto vectorial ficticio:  $u \times v = u_1 * v_2 - u_2 * v_1 = |u| * |v| \sin \theta$ , donde  $\theta$  es el ángulo más chico entre  $u$  y  $v$ .

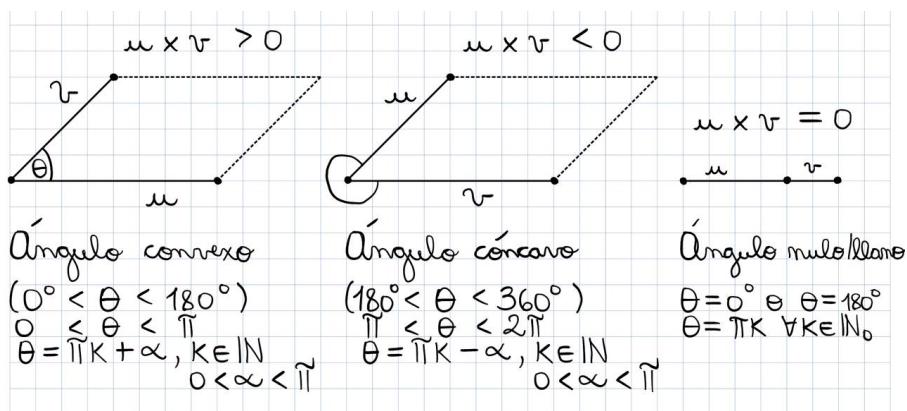
Algunas cosas a tener en cuenta:

- Este valor **no** es un vector, **es un escalar**.
- En  $\mathbb{R}^2$ ,  $|u \times v|$  es el área del paralelogramo formado por  $u$  y  $v$ :



Esta operación se utiliza con frecuencia, debido a la utilidad que tiene, puesto que de este producto, podemos sacar las siguientes conclusiones según el resultado:

- Si  $u \times v > 0 \Rightarrow v$  está a la "izquierda" de  $u$  en sentido **antihorario**.
- Si  $u \times v < 0 \Rightarrow v$  está a la derecha de  $u$  en sentido **horario**.
- Sean  $u$  y  $v$  no nulos,  $u \times v = 0 \Leftrightarrow$  los puntos  $u$  y  $v$  son **colineales** (están en la misma línea).
- Nota: decir que en  $\mathbb{R}^2$  los puntos  $u$  y  $v$  son colineales, **es lo mismo** que decir que los vectores asociados a  $u$  y a  $v$  son **paralelos** (tienen igual dirección), y lo notamos  $u/v$ .



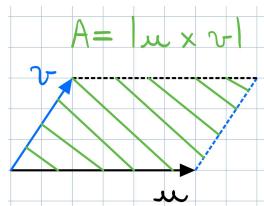
Implementación en C++:

```
struct Vec{  
    // Coordenadas (x, y)  
    int x, y;  
  
    (...)  
  
    // Producto cruz entre dos vectores  
    int operator^(Vec q) {return x * q.y - y * q.x;}  
};
```

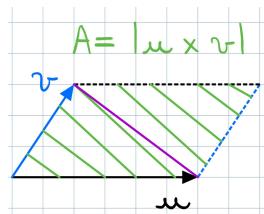
### 3.13. Área de un triángulo

Con lo que vimos hasta ahora, podemos deducir cómo obtener el área de un triángulo. Veamos cómo hacerlo:

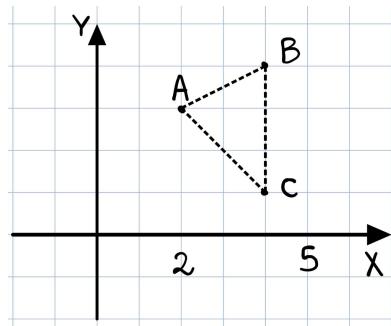
Vimos que dados dos vectores  $u$  y  $v$ ,  $|u \times v|$  es el área del paralelogramo formado por ellos:



Ahora, miremos lo que sucede si partimos a la mitad ese paralelogramo de forma conveniente (línea violeta):



Notamos que nos quedaron dos triángulos. Particularmente, nos interesa el que forman  $u$  y  $v$ . El área de ese triángulo, es justamente la mitad del área del paralelogramo, es decir  $A_{triángulo} = \frac{|u \times v|}{2}$ .



Implementación en C++:

```

struct Vec{
    // Coordenadas (x, y)
    int x, y;

    (...)

    // Producto cruz entre dos vectores
    int operator^(Vec q) {return x * q.y - y * q.x;}
};

int main(){
    Vec A, B, C;

    // Leemos los puntos que forman el triángulo
    cin >> A.x >> A.y;
    cin >> B.x >> B.y;
    cin >> C.x >> C.y;

    // Obtenemos cualquier par de lados del triángulo. En este caso tomamos
    // al punto A como referencia.

    // Hacemos las restas porque si trabajáramos directamente con A, B y C
    // para hacer el producto cruz, estaríamos usando los vectores que parten
    // desde el origen hasta dichos puntos para hacer el producto cruz, y
    // dichos vectores NO son los que forman el triángulo (pensar que el
    // triángulo está "flotando" en alguna parte del plano cartesiano)
    Vec AB = B - A;
    Vec AC = C - A;

    // Tomamos el valor absoluto, porque el producto cruz puede ser negativo
    // según la posición relativa de los vectores, y un área no puede ser
    // negativa
    cout << "Área del triángulo formado por A, B, y C: " << abs(AB ^ AC) / 2;

    return 0;
}

```

## 4. Rectas/Líneas

### 4.1. Definición y representación

Una recta es un conjunto infinito de puntos que se extiende indefinidamente en dos sentidos opuestos sobre un mismo plano. Cabe destacar que si bien tiene longitud infinita, no tiene ancho.

Hay muchas maneras de representar una recta, y la que usemos puede depender de cómo sea el input del problema, ya que a veces nos dan las rectas representadas de cierta forma, y por comodidad trabajamos con la representación dada. Algunas de las representaciones más comunes son:

- $L(x) = mx + b$ , donde  $m$  es la pendiente de la recta, y  $b$  es la ordenada al origen.
- $L(t) = p + t * v$ , donde  $v \in \mathbb{R}^2$  es el Vector dirección, y  $p$  es un Punto de paso en la recta.  $t \in \mathbb{R}$  es un escalar que representa cuánto se estira o contrae el vector  $v$ .
- $ax + by + c = 0$ , donde  $a$ ,  $b$ , y  $c$  son coeficientes constantes que determinan la recta.

Esta representación la veremos más adelante.

La representación  $L(x) = mx + b$  aunque conocida, resulta problemática. Veamos por qué.

La pendiente  $m$  de una recta se define como el cociente entre el cambio en el eje  $y$  (vertical), y el cambio en el eje  $x$  (horizontal) entre dos puntos en la recta.

Matemáticamente, se representa así:  $m = \frac{\Delta y}{\Delta x}$ .

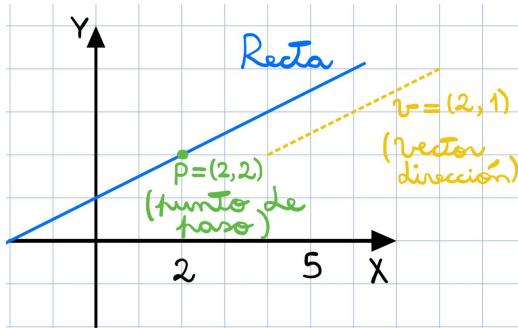
Para que ese cociente/división esté bien definido  $\Delta x$  tiene que ser distinto de cero. ¿Cuándo podría pasar que  $\Delta x$  fuera 0? En el caso de una recta vertical, ya que una recta vertical no tiene un cambio en el eje  $x$  (porque todas sus coordenadas  $x$  son iguales), por lo que su pendiente sería indefinida.

¿Entonces cómo hacemos para representar rectas verticales?

Tenemos que tratarlas como un caso aparte, representándolas con la ecuación  $x = c$  en vez de usar la pendiente. A nivel código, esto se traduce en un **if** para manejar este caso especial.

Por este motivo, vamos a utilizar las otras dos representaciones que son más cómodas.

Veamos la representación  $L(t) = p + t * v$ :



Esta representación es una forma paramétrica de representar una recta, porque describe todos los puntos en una recta, usando un único parámetro, en este caso  $t$ . Es decir, a medida que  $t$  varía, se describen todos los puntos de la recta.

Algo a notar, es que dada una recta en el plano, hay **infinitos** vectores que pueden actuar como un vector dirección para esa recta. Generalmente, de todas esas posibilidades, elegimos trabajar con las que resultan más “naturales” o cómodas.

Por ejemplo, en la imagen trabajamos con el vector dirección  $(2, 1)$ , que indica que para un cambio de 2 en el eje  $x$ , hay un cambio de 1 en el eje  $y$ , pero también podríamos haber elegido el vector dirección  $(1, 0.5)$ , que se obtiene de dividir por 2 al vector mencionado previamente.

Esto es porque en realidad cualquier múltiplo no nulo de  $(2, 1)$  también es un vector dirección para esa recta. La idea detrás de esto es que el vector indica una dirección particular, y cualquier escalamiento (múltiplo) de ese vector, también apunta en la misma dirección, y por lo tanto también es válido como vector dirección.

¿Cómo podemos obtener una recta representada de esta forma, a partir de dos puntos  $a$  y  $b$  que pasan por la recta  $L$ ?

- Para la parte del vector: como  $a$  y  $b$  se encuentran sobre  $L$ , el vector  $v = b - a$  representa la dirección de la recta (el vector que va desde el punto  $a$  hasta el punto  $b$ ).
- Para la parte del punto: como sabemos que tanto  $a$  como  $b$  se encuentran sobre  $L$ , podemos elegir a cualquiera de los dos como punto de paso. En particular, elegimos  $p = a$  como punto de paso.

De esta manera, la recta se escribe como  $L(t) = a + t * (b - a)$

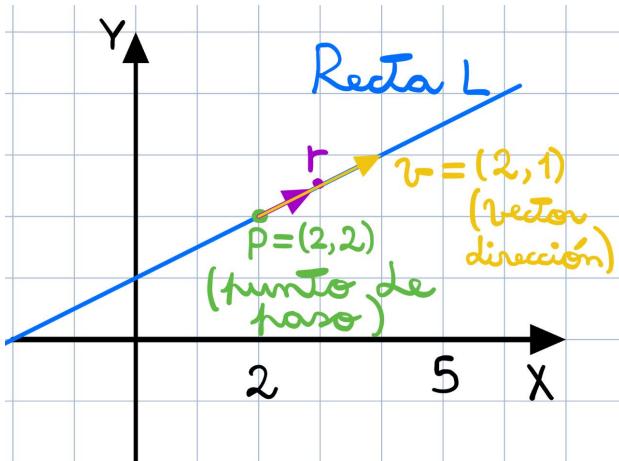
En C++, la podemos implementar como:

```
struct Recta{  
    // p es el punto de paso, y v es el vector dirección  
    Vec p, v;  
  
    // Constructor  
    Recta(Vec p, Vec q) : p(p), v(q - p) {}  
  
    // Obtener un punto en la recta dado un valor de t  
    Vec punto_en_recta(double t){  
        return p + t * v;  
    }  
};
```

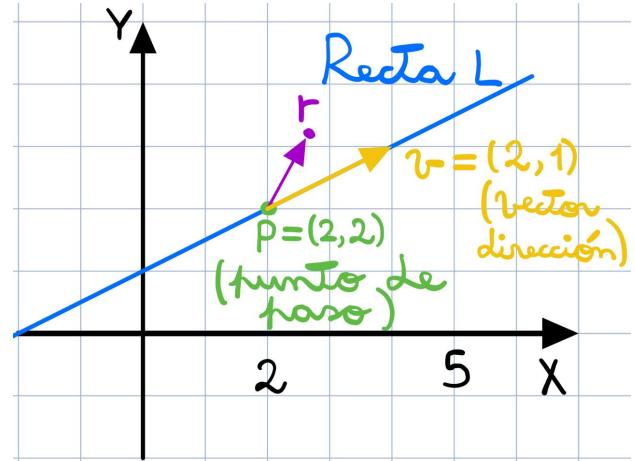
## 4.2. Determinar si un punto está en una recta

Dado un punto  $r$  y una recta  $L$  representada como  $L(t) = p + t * v$ , queremos determinar si  $r$  está en  $L$ .

Contando con el vector  $v$  (es decir, el vector dirección), y calculando el vector  $pr = r - p$  (vector que va desde el punto de paso  $p$  hasta el punto de interés  $r$ , tenemos lo siguiente visualmente:



$r$  está en  $L$



$r$  NO está en  $L$

Intuitivamente, si los vectores  $v$  y  $pr$  son paralelos (en la primera imagen lo son, pero en la segunda no), como ambos vectores “parten” del mismo punto  $p$ , y  $p$  es un punto que ya está en la recta (porque  $p$  es un punto de paso), entonces  $r$  también va a estar en la recta.

De esta manera, para saber si  $r$  está o no en  $L$ , basta con chequear si  $v$  y  $pr$  son paralelos:  $v \times pr = 0$ .

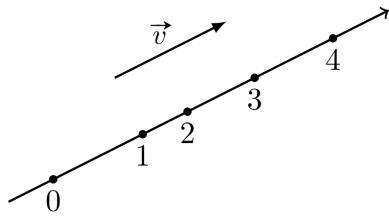
Implementación en C++:

```
struct Recta{
    // p es el punto de paso, y v es el vector dirección
    Vec p, v;
    (...)

    // Determina si la recta tiene o no al punto 'r'
    bool tiene_punto(Vec r){
        Vec pr = r - p;
        // Chequeamos si los vectores v y pr son paralelos
        return (v ^ pr) == 0;
    }
};
```

### 4.3. Ordenar puntos de una recta

Dados puntos de una recta  $L$ , queremos ordenarlos según el orden en el que aparecen, de acuerdo a la dirección del vector dirección  $v$ :



Para esto, podemos usar el producto punto, ya que se cumple que el punto  $A$  aparece antes que el punto  $B$  si  $A * v < B * v$ .

Para ver por qué podemos usar eso para ordenar los puntos, recordemos la definición del producto escalar entre dos vectores:

$$u \cdot v = |u||v| \cos \theta \text{ donde } \theta \text{ es el ángulo más chico entre } u \text{ y } v.$$

En nuestro caso, como los puntos están sobre la misma recta, están alineados, y por lo tanto, el ángulo  $\theta$  formado por cada punto de la recta, y el vector dirección  $v$ , es 0. De esa forma, tenemos que:  $u \cdot v = |u||v| \cos \theta = |u||v| \cos 0 = |u||v|$ .

Recordando la relación que hay entre la longitud de la proyección ortogonal de  $u$  sobre  $v$ , tenemos que la misma es  $|u| \cos \theta = |u| \cos 0 = |u|$ .

Resumiendo lo que tenemos hasta ahora:

- $u \cdot v = |u||v|$ .
- $|u|$  es la longitud de la proyección ortogonal de  $u$  sobre  $v$ .

En el contexto de este problema, la proyección del vector  $u$  sobre  $v$  es un vector cuya longitud es  $|u|$ , ya que ambos vectores están alineados, y el ángulo  $\theta$  entre ellos es 0. Sin embargo, es crucial recordar que la longitud  $|u|$  por sí misma es un escalar que representa la longitud del vector  $u$  pero no su dirección. La dirección de la proyección, viene dada por el vector sobre el que se proyecta, en este caso  $v$ .

Entonces el producto punto  $u \cdot v$  resulta en un escalar que es la longitud de la proyección de  $u$  sobre  $v$ , multiplicada por la longitud de  $v$ . Este escalar cuantifica la extensión de  $u$  en la dirección de  $v$ , considerando la longitud de  $v$ . Por ende, aunque  $|u|$  es la longitud de la proyección,  $u \cdot v$  integra también la dirección y sentido de  $v$ , permitiéndonos así ordenar los puntos sobre la recta, de acuerdo con la dirección definida por  $v$ .

Así, el producto punto  $u \cdot v$  es la longitud de  $u$  proyectada en la dirección de  $v$ . Por lo tanto, el chequeo de  $A * v < B * v$  calcula la proyección del vector  $A$  en la dirección de  $v$ , y hace lo mismo con el vector  $B$ . Luego comparar estos dos valores nos dice qué punto está "más adelante" en la recta en la dirección  $v$ .

Implementación en C++:

```
#include <algorithm> // Para std::sort
#include <vector>

struct Vec{
    // Coordenadas (x, y)
    int x, y;

    // Constructor
    Vec(int x = 0, int y = 0) : x(x), y(y) {}

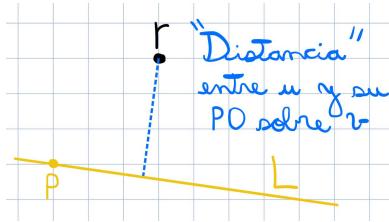
    // Producto escalar entre dos vectores
    int operator*(Vec v) {return x * v.x + y * v.y;}
};

bool comparar_por_proyeccion(Vec a, Vec b, Vec v){
    // Compara a y b por su proyección en la dirección de v
    return a * v < b * v;
}

void ordenar_puntos_segun_direccion(vector<Vec>& puntos, Vec direccion) {
    // Ordenamos los puntos en la dirección de 'direccion' usando el
    // producto escalar
    sort(puntos.begin(), puntos.end(),
        [&direccion](const Vec& a, const Vec& b) {
            return comparar_por_proyeccion(a, b, direccion);
    });
}
```

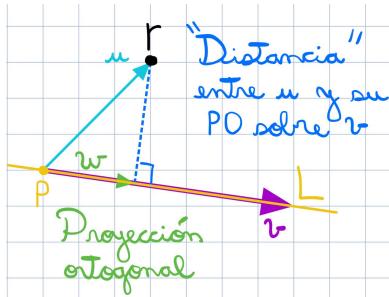
#### 4.4. Distancia entre un punto y una recta

Dado un punto  $r$ , y una recta  $L(t) = p + t * v$ , queremos encontrar cuál es la distancia de  $r$  a  $L$ . Intuitivamente:



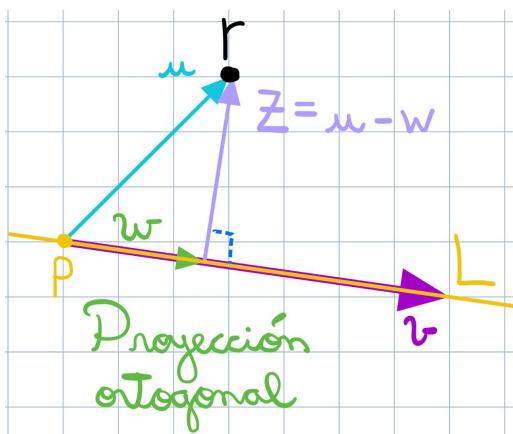
Para obtener dicha distancia, podemos usar lo que vimos de la proyección ortogonal. Para ello, utilizaremos el punto de paso  $p$  por comodidad, aunque en realidad eligiendo cualquier punto de la recta también funciona (la razón viene más adelante).

Lo primero que hacemos, es obtener el vector  $u = r - p$ . Luego, calculamos  $w = \text{proj}(u, v)$ :



La distancia que estamos buscando, es justamente la **longitud** de la línea perpendicular punteada entre  $u$  y la recta  $L$ .

Pero antes de poder conocer esa longitud, necesitamos primero el vector. Ese vector lo obtenemos haciendo  $z = u - w$ :



Finalmente, la respuesta es la longitud de  $z$ , es decir  $|z|$ .

Nota: este vector  $z$  que nos conseguimos, es la contraproyección de  $u$  sobre  $v$ , es decir  $z = \text{contra\_proj}(u, v) = u - \text{proj}(u, v) = u - w$ .

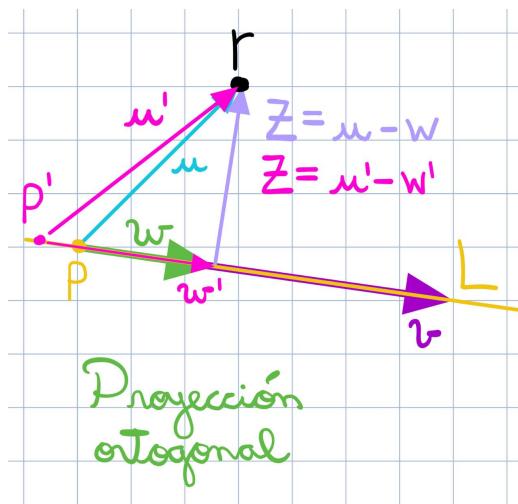
Además, como vector podríamos haber hecho  $w - u$ , y también habría estado bien para el propósito de obtener la longitud del vector (cambia el sentido pero no la longitud).

Implementación en C++:

```
double dist_punto_recta(Vec r, Recta L){  
    // Obtenemos el vector 'u' que va desde el punto de paso 'p' hasta 'r'  
    Vec u = r - L.p;  
  
    Vec z = contraproyeccion(u, L.v);  
    return z.norm();  
}
```

### Volviendo a lo que quedó pendiente de antes

Dijimos que en vez de usar el punto de paso  $p$ , podíamos usar cualquier punto  $p'$  en la recta  $L$ , y también iba a funcionar. ¿Por qué?



Porque íbamos a obtener el vector  $u'$  que va desde  $p'$  hasta  $r$ , para después, repetir el mismo proceso que antes, es decir, encontrar la proyección ortogonal de  $u'$  sobre  $v$ , que tiene la misma dirección que la recta, es decir tiene dirección  $v$ , y esto es independiente del punto de partida que elijamos en la recta, porque la dirección  $v$  es constante para todos los puntos de la recta, y por lo tanto, las proyecciones de  $u$  sobre  $v$ , y  $u'$  sobre  $v$ , son las mismas.

Además, el vector perpendicular ( $z$ ) a la recta, es siempre el mismo sin importar qué punto de la recta elijamos como referencia. Entonces como el vector  $z$  es siempre el mismo, la longitud del vector  $z$  es siempre la misma, y por ende la distancia del punto  $r$  a la recta también será siempre la misma, sin importar qué punto de la recta usemos como referencia.

## 4.5. Representación $ax + by + c = 0$

Retomemos la última representación propuesta para rectas:  $ax + by + c = 0$ , donde  $a$ ,  $b$  y  $c$  son coeficientes constantes.

### Relación con representación $L(x) = mx + b$

Esta representación, está relacionada con la de  $L(x) = mx + b$ , ya que podemos reescribir esta última como:  $y = mx + b$ . Ahora, si llevamos todos los términos al mismo lado, queda:  $mx - y + b = 0$ .

Esto nos da una ecuación en la última representación, con  $a = m$ ,  $b = -1$  y  $c = b$  (este  $b$  es la ordenada al origen, NO el coeficiente de la última representación).

### Interpretación de la representación

Los coeficientes  $a$  y  $b$  determinan la dirección de la recta. Por ejemplo el vector  $(-b, a)$  es perpendicular a la recta. Esto es lo que nos permite saber en qué dirección se encuentra la recta, a partir de cuál es la dirección que es perpendicular a ella.

El coeficiente  $c$  afecta a la posición de la recta en el plano (en general, desplazará la recta para arriba o para abajo, o en el caso de rectas verticales, para la izquierda o para la derecha), pero no a su dirección, y cabe recalcar que  $c$  no necesariamente es la ordenada al origen. Si queremos obtener la ordenada al origen, debemos hacer  $x = 0$ , y resolver para  $y$ .

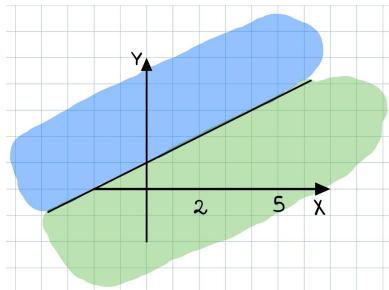
### Observaciones

- Si  $b = 0$ , la recta es vertical, ya que la ecuación de la recta se reduce a  $ax + c = 0$  (o sea, no depende de  $y$ ), que a su vez implica que  $x = -c/a$ . A menos que  $x$  sea 0, la recta no corta al eje  $y$ , aunque sí cortará al eje  $x$  en algún momento, dada su naturaleza de longitud infinita.

## 4.6. Rectas y semiplanos

La representación  $ax + by + c = 0$  para rectas, resulta conveniente a la hora de realizar ciertas operaciones. Antes de ver su utilidad, veamos algunos conceptos:

Decimos que una recta, divide al plano en dos semiplanos. Visualmente:



La región del plano sombreada de azul es un semiplano, y la región del plano sombreada de verde es el otro semiplano.

Formalmente, estas regiones/semitplanos se definen de la siguiente manera:

Dada una recta en el plano definida por la ecuación  $ax + by + c = 0$ , los semiplanos correspondientes son:

- $ax + by + c > 0$  (semiplano azul).
- $ax + by + c < 0$  (semiplano verde).

De esta manera, podemos utilizar el concepto de semiplano, para determinar de qué lado (semiplano) de la recta se encuentra un punto, que es lo que veremos a continuación.

## 4.7. Problema Codeforces 498A: Crazy Town

time limit per test: 1 second  
memory limit per test: 256 megabytes  
input: standard input  
output: standard output

Crazy Town is a plane on which there are  $n$  infinite line roads. Each road is defined by the equation  $a_i x + b_i y + c_i = 0$ , where  $a_i$  and  $b_i$  are not both equal to the zero. The roads divide the plane into connected regions, possibly of infinite space. Let's call each such region a block. We define an intersection as the point where at least two different roads intersect.

Your home is located in one of the blocks. Today you need to get to the University, also located in some block. In one step you can move from one block to another, if the length of their common border is nonzero (in particular, this means that if the blocks are adjacent to one intersection, but have no shared nonzero boundary segment, then it is not allowed to move from one to another one in one step).

Determine what is the minimum number of steps you have to perform to get to the block containing the university. It is guaranteed that neither your home nor the university is located on the road.

### Input

The first line contains two space-separated integers  $x_1, y_1$  ( $-10^6 \leq x_1, y_1 \leq 10^6$ ) — the coordinates of your home.

The second line contains two integers separated by a space  $x_2, y_2$  ( $-10^6 \leq x_2, y_2 \leq 10^6$ ) — the coordinates of the university you are studying at.

The third line contains an integer  $n$  ( $1 \leq n \leq 300$ ) — the number of roads in the city. The following  $n$  lines contain 3 space-separated integers ( $-10^6 \leq a_i, b_i, c_i \leq 10^6; |a_i| + |b_i| > 0$ ) — the coefficients of the line  $a_i x + b_i y + c_i = 0$ , defining the  $i$ -th road. It is guaranteed that no two roads are the same. In addition, neither your home nor the university lie on the road (i.e. they do not belong to any one of the lines).

### Output

Output the answer to the problem.

Dejando de lado la historia del enunciado, y enfocándonos en el apartado geométrico, tenemos dos puntos y muchas rectas.

Las rectas están representadas de la forma  $ax + by + c = 0$ , y para los puntos tenemos sus coordenadas.

El objetivo del ejercicio, es contar la cantidad de rectas con las cuales dichos puntos quedan en distintos semiplanos.

Para esto, podemos usar lo que vimos en la sección anterior, aprovechando fuertemente la representación de las rectas, que resulta muy cómoda para resolver el problema de forma sencilla.

La idea es tomar los dos puntos, y evaluarlos en la ecuación de cada una de las rectas (es decir, reemplazar la variable  $x$  por las coordenadas  $x$  de cada punto, y reemplazar la variable  $y$  por las coordenadas  $y$  de cada punto), guardando el resultado en ambos casos.

Luego, si los resultados de las evaluaciones tienen distinto signo, quiere decir que se encuentran en distintos semiplanos (para la recta con la que evaluamos). Entonces incrementamos en 1 un contador.

Este proceso lo repetimos para cada recta, e imprimimos la cantidad final después de haber considerado todas las rectas.

Finalmente, el algoritmo tiene complejidad  $O(n)$  donde  $n$  es la cantidad de rectas.

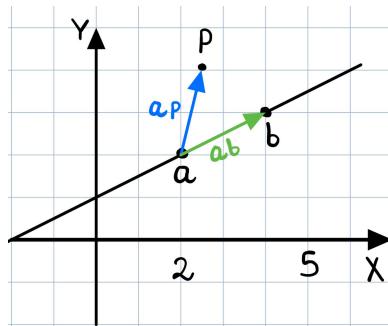
#### 4.8. Determinar en qué semiplano se encuentra un punto respecto de una recta (paramétrica)

También podemos resolver el problema con la representación paramétrica  $L(t) = p + t * v$  (definición).

Dada una recta que pasa por dos puntos  $a$  y  $b$  (que ya vimos en la sección *Rectas: Definición y representación* cómo obtenerla), queremos determinar en qué semiplano se encuentra un punto  $p$  (o si  $p$  está sobre la recta  $L$ ).

En el apartado de *Puntos/Vectores: Producto vectorial (o producto cruz)*, vimos que el producto cruz nos permite obtener la posición relativa de dos vectores. Entonces vamos a aprovechar eso, y vamos a obtener los vectores  $ab = b - a$  (que va desde  $a$  hasta  $b$ ) y  $ap = p - a$  (que va desde  $a$  hasta  $p$ ).

Ejemplo ilustrativo de lo que hicimos hasta ahora:



Ahora simplemente hay que hacer el producto cruz entre  $ab$  y  $ap$ , y responder acorde al resultado obtenido en dicha operación.

En el caso de la imagen,  $ab \times ap > 0$ , porque  $ap$  está a la izquierda de  $ab$  en sentido horario.

Implementación en C++:

```
// Determina de qué lado/semiplano de la recta formada por a y b, se encuentra
// p
int side(Vec a, Vec b, Vec p) {
    // Calculamos los vectores ab (desde a hasta b) y ap (desde a hasta p)
    Vec ab = b - a, ap = p - a;
    int res = ab ^ ap;
    if (res == 0) return 0; // p está en la recta que pasa por a y b
    if (res > 0) return 1; // p está a la izquierda de la recta
    return -1;           // p está a la derecha de la recta
}
```

## 4.9. Problema CSES: Point Location Test

**Time limit:** 1.00 s   **Memory limit:** 512 MB

There is a line that goes through the points  $p_1 = (x_1, y_1)$  and  $p_2 = (x_2, y_2)$ . There is also a point  $p_3 = (x_3, y_3)$ .

Your task is to determine whether  $p_3$  is located on the left or right side of the line or if it touches the line when we are looking from  $p_1$  to  $p_2$ .

### Input

The first input line has an integer  $t$ : the number of tests.

After this, there are  $t$  lines that describe the tests. Each line has six integers:  $x_1$ ,  $y_1$ ,  $x_2$ ,  $y_2$ ,  $x_3$  and  $y_3$ .

### Output

For each test, print "LEFT", "RIGHT" or "TOUCH".

### Constraints

- $1 \leq t \leq 10^5$
- $-10^9 \leq x_1, y_1, x_2, y_2, x_3, y_3 \leq 10^9$
- $x_1 \neq x_2$  or  $y_1 \neq y_2$

En este problema tenemos una recta que pasa por los puntos  $p_1 = (x_1, y_1)$  y  $p_2 = (x_2, y_2)$ . Queremos determinar en qué semiplano se encuentra el punto  $p_3$ , o si se encuentra sobre la recta.

Para resolverlo, podemos usar el producto cruz, que vimos antes que nos permite saber la "orientación"/posición relativa entre dos puntos/vectores.

La idea es usar el vector que va desde  $p_1$  a  $p_2$ , que se obtiene haciendo  $v_1 = p_2 - p_1$ , y el vector que va desde  $p_1$  a  $p_3$ , que se obtiene haciendo  $v_2 = p_3 - p_1$ . De esta manera, ambos vectores parten desde el mismo lugar (el punto de paso  $p_1$ ).

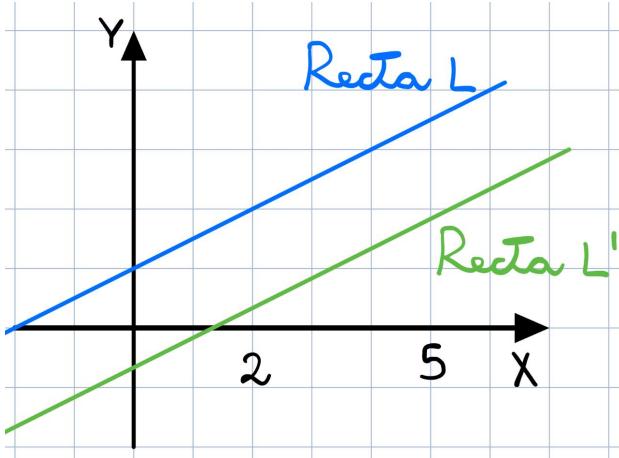
Esto último es importante, ya que para el chequeo de si  $p_3$  está sobre la recta, no alcanza con ver si los vectores  $v_1$  y  $v_2$  son paralelos, ya que podrían ser paralelos, pero estar en distintos lugares del plano, y que entonces  $p_3$  en realidad no esté en la recta. En cambio, como sabemos que ambos vectores parten del mismo punto, si son paralelos, entonces no sólo tienen igual dirección, sino que además están en la misma recta (esencialmente, es el mismo razonamiento que usamos en la sección *Determinar si un punto está en una recta*).

Ahora, simplemente hay que hacer el producto cruz entre ambos vectores, es decir  $v_1 \times v_2$ , y ver si es  $< 0$ ,  $= 0$ , o  $> 0$ , y en base a eso imprimir por salida estándar según corresponda.

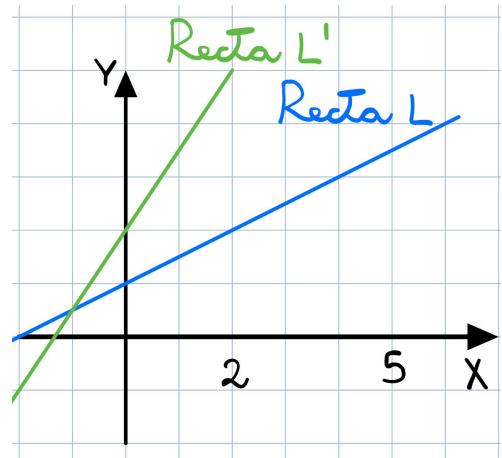
Este algoritmo tiene complejidad  $O(T)$  donde  $T$  es la cantidad de casos de prueba.

#### 4.10. Determinar si dos rectas son paralelas

Dadas dos rectas  $L$  y  $L'$  representadas de la manera que venimos viendo (paramétrica), queremos determinar si son paralelas. Notemos que alcanza con verificar que sus respectivos vectores dirección sean paralelos.



$L$  y  $L'$  son paralelas



$L$  y  $L'$  NO son paralelas

Observar que el único caso en el que dos rectas cualesquiera **NO** se intersecan, es cuando son paralelas y no coincidentes. En todos los demás casos, dada la naturaleza de longitud infinita de las rectas, se terminan intersecando en algún punto.

En caso de ser coincidentes, hay **infinitos** puntos de intersección.

Implementación en C++:

```
struct Recta{
    // p es el punto de paso, y v es el vector dirección
    Vec p, v;

    (...)

    // Determina si dos rectas son paralelas
    bool operator/(Recta L){return (v ^ L.v) == 0;}
};
```

## 4.11. Determinar en qué punto se intersecan dos rectas

Dadas dos rectas  $L = p1 + \alpha * v1$  y  $L' = p2 + \beta * v2$ , queremos encontrar el punto en el que se intersecan (si es que se intersecan).

Para ver si se intersecan, podemos usar lo que vimos en la sección anterior.

Ahora, para encontrar el punto de intersección (suponiendo que sí se intersecan), tenemos que resolver la siguiente igualdad:  $p1 + \alpha * v1 = p2 + \beta * v2$ .

Es decir, tenemos que encontrar los valores de  $\alpha$  y  $\beta$  para los cuales  $L$  y  $L'$  se intersecan.

A continuación veremos un método inventado por Agustín Gutiérrez para resolver esto, que resulta en una implementación sencilla.

Antes de empezar a aplicar el método, notemos una característica útil del producto cruz:

El producto cruz de un vector con sí mismo, es 0. Es decir, si tenemos el vector  $a = (a1, a2)$ , entonces  $a \times a = a1 * a2 - a2 * a1 = 0$ .

Visualmente/geométricamente, podemos pensar que estamos tratando de formar un paralelogramo con dos vectores idénticos, y que esencialmente lo que estamos obteniendo es una línea recta, por lo que no hay área (el área del “paralelogramo” es 0).

Aprovechando esa propiedad, aplicamos la operación “ $\times v2$ ” a ambos de la igualdad, consiguiendo eliminar la incógnita  $\beta$ :

$$(p1 + \alpha * v1) \times v2 = (p2 + \beta * v2) \times v2.$$

$$p1 \times v2 + \alpha * v1 \times v2 = p2 \times v2 + \beta * v2 \times v2 \text{ (Distributiva).}$$

$$p1 \times v2 + \alpha * v1 \times v2 = p2 \times v2 \text{ (Propiedad útil del producto cruz).}$$

$$\alpha * v1 \times v2 = p2 \times v2 - p1 \times v2.$$

$$\alpha * v1 \times v2 = (p2 - p1) \times v2 \text{ (Factor común).}$$

$$\alpha = \frac{(p2 - p1) \times v2}{v1 \times v2}.$$

Una vez que tenemos  $\alpha$ , concluimos entonces que el punto de intersección  $p$  es  $p = p1 + \alpha * v1$ .

### Algunas observaciones

- En el denominador, nos quedaron los vectores dirección de ambas rectas, por lo que es 0 cuando las rectas son paralelas. Esto **no** representa un problema, suponiendo que antes de realizar este método, verificamos si las rectas se intersecaban o no.
- Si estamos trabajando con coordenadas enteras, la única operación que puede no ser entera, es la división con la que obtenemos  $\alpha$ , pero como esta se realiza al final del método, no introducimos errores de precisión acumulativos, reduciendo así el margen de error.

Implementación en C++:

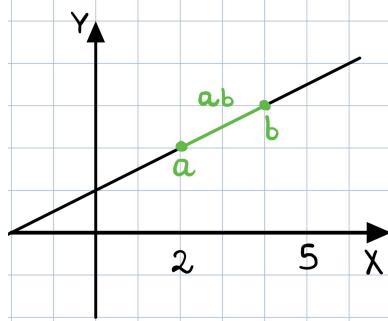
```
Vec interseccion(Recta r1, Recta r2){  
    double alpha = ((r2.p - r1.p) ^ r2.v) / (r1.v ^ r2.v);  
    return r1.p + alpha * r1.v;  
}
```

## 5. Segmentos

### 5.1. Definición y representación

Intuitivamente, un segmento es una “porción” de una línea recta, delimitada por dos puntos que actúan como extremos.

De aquí yace la principal diferencia entre recta y segmento, ya que dijimos que las rectas tienen longitud infinita, y que las mismas se extienden en ambas direcciones, mientras que los segmentos tienen una longitud finita y definida.



Más formalmente, dados dos puntos  $a$  y  $b$ , podemos definir el segmento  $ab$ , como los puntos de la recta que pasa por  $a$  y por  $b$ , que se encuentran entre  $a$  (inclusive) y  $b$  (inclusive).

**Nota:** los segmentos representan la distancia más corta entre dos puntos en el plano.

Implementación en C++:

```
struct Seg{  
    // Límites del segmento ab  
    Vec a, b;  
  
    // Constructor  
    Seg(Vec a, Vec b): a(a), b(b) {}  
};
```

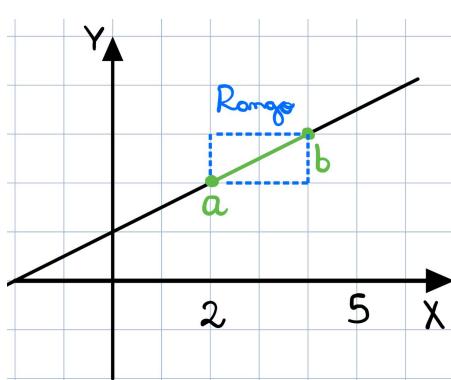
Esta es sólo una forma de representar segmentos. Después en la práctica, podemos representarlos como `pair<Vec, Vec>` por ejemplo, o simplemente como un par de 'Vec' "sueltos" (esto último fue lo que se hizo en los códigos de los problemas y en los ejemplos).

## 5.2. Determinar si un punto se encuentra en un segmento

Dado un segmento  $ab$  y un punto  $c$ , queremos verificar si  $c$  se encuentra en  $ab$ .

Observaciones:

- Si  $c = a$  o  $c = b$  (o sea,  $c$  es uno de los extremos del segmento), entonces sabemos que  $c$  se encuentra en el segmento.
- Si  $c$  no es ninguno de los bordes, hay 2 casos:
  - $c$  es colineal con los otros puntos, en cuyo caso, tenemos que verificar que la coordenada  $x$  de  $c$  esté en el rango definido por el segmento, y lo mismo con la coordenada  $y$ :



- $c$  **NO** es colineal, por lo que ni siquiera se encuentra en la misma recta, así que tampoco está en el segmento  $ab$ .

Implementación en C++:

```
// Determina si el punto 'c' se encuentra en el segmento ab
bool enSegmento(Vec a, Vec b, Vec c) {
    // Si c es uno de los extremos del segmento
    if (a == c || b == c) return true;

    // Comprobamos si los 3 puntos son colineales. Si no lo son, seguro que c
    // no está en el segmento, por lo que retornamos false
    if (side(a, b, c) != 0) return false;

    // Son colineales

    // Para que c esté en el segmento (a, b), sus coordenadas x e y deben
    // estar en el rango definido por las coordenadas 'x' e 'y' de a y b.

    bool xInRange = min(a.x, b.x) <= c.x && c.x <= max(a.x, b.x);
    bool yInRange = min(a.y, b.y) <= c.y && c.y <= max(a.y, b.y);

    return xInRange && yInRange;
}
```

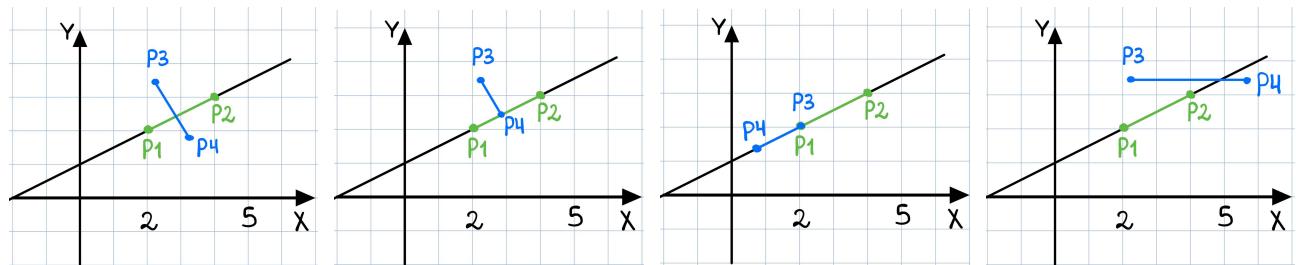
### 5.3. Determinar si dos segmentos se intersecan

Dados los segmentos  $p1p2$  y  $p3p4$ , queremos determinar si se intersecan.

Notemos lo siguiente:

- Puede ocurrir que un punto final de un segmento, coincida exactamente con un punto final del otro segmento. En este caso, podemos afirmar directamente que hay intersección.
- Si los segmentos en cuestión son colineales, para poder afirmar que se intersecan, hay que verificar que comparten al menos un punto.
- Cuando dos segmentos se intersecan, al menos uno de los puntos finales de un segmento (delimitador/extremo del segmento), está de un lado de la recta formada por el otro segmento, y el otro punto final está del lado opuesto. Esta condición debe cumplirse para ambos segmentos.

Algunos casos representados visualmente:



Implementación en C++:

```
// Verifica si los segmentos (p1, p2) y (p3, p4) se intersecan
bool se_intersecan(Vec p1, Vec p2, Vec p3, Vec p4) {
    // Comprobamos si algún punto final de un segmento coincide con algún
    // punto final del otro segmento.
    // Si dos puntos finales coinciden, entonces los segmentos se intersecan
    // en ese punto, por lo que retornamos true
    if (p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4) return true;

    // Determinamos la posición relativa de p3 con respecto a (p1, p2) y así
    // sucesivamente
    int s1 = side(p1, p2, p3);
    int s2 = side(p1, p2, p4);
    int s3 = side(p3, p4, p1);
    int s4 = side(p3, p4, p2);

    // Verificamos si los puntos son colineales.
    // Si lo son, tenemos que chequear si alguno de los puntos del primer
    // segmento está dentro del segundo segmento o viceversa.
    if (s1 == 0 && s2 == 0 && s3 == 0 && s4 == 0) {
        return enSegmento(p1, p2, p3) || enSegmento(p1, p2, p4) ||
               enSegmento(p3, p4, p1) || enSegmento(p3, p4, p2);
    }

    // Para que dos segmentos se intersequen, un extremo de un segmento debe
    // estar de un lado de la recta formada por el otro segmento y el otro
    // extremo del otro lado. Esto tiene que valer para ambos segmentos.

    // El uso de <= en lugar de < garantiza que también consideramos el caso
    // en que // un punto está exactamente en la recta formada por el otro
    // segmento (s1 o s2 o s3 o s4 es 0).
    // Si los segmentos comparten al menos un punto, se considera que
    // se intersecan
    return s1 * s2 <= 0 && s3 * s4 <= 0;
}
```

Este problema tiene muchas **aplicaciones prácticas** en el mundo real, especialmente en gráficos por computadora, diseño asistido por computadora (CAD), y en juegos donde la detección de colisiones es fundamental.

## 5.4. Problema CSES: Line Segment Intersection

**Time limit:** 1.00 s   **Memory limit:** 512 MB

There are two line segments: the first goes through the points  $(x_1, y_1)$  and  $(x_2, y_2)$ , and the second goes through the points  $(x_3, y_3)$  and  $(x_4, y_4)$ .

Your task is to determine if the line segments intersect, i.e., they have at least one common point.

### Input

The first input line has an integer  $t$ : the number of tests.

After this, there are  $t$  lines that describe the tests. Each line has eight integers  $x_1, y_1, x_2, y_2, x_3, y_3, x_4$  and  $y_4$ .

### Output

For each test, print "YES" if the line segments intersect and "NO" otherwise.

### Constraints

- $1 \leq t \leq 10^5$
- $-10^9 \leq x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4 \leq 10^9$
- $(x_1, y_1) \neq (x_2, y_2)$
- $(x_3, y_3) \neq (x_4, y_4)$

Es en esencia lo que acabamos de ver, hecho problema de competencia con juez online. El código del problema se encuentra en el apéndice.

El algoritmo propuesto para resolver el problema, tiene complejidad  $O(T)$ , donde  $T$  es la cantidad de casos de prueba.

## 5.5. Determinar en qué punto se intersecan dos segmentos

Dados los segmentos  $p1p2$  y  $p3p4$ , queremos determinar en qué punto se intersecan (si es que se intersecan).

Para ver si se intersecan, podemos usar lo que vimos en la sección anterior.

En caso de que se intersequen, procedemos a calcular su punto de intersección. ¿Cómo lo hacemos?

**De la misma forma que para calcular la intersección para rectas.** Como previamente verificamos si se intersecaban, y la respuesta fue que sí, entonces ese punto de intersección que encontramos (como si fueran rectas), es también un punto de intersección para los segmentos.

Como la función que hicimos para calcular la intersección entre dos rectas, tenía como precondición que ya sabíamos que se intersecaban, el algoritmo es simplemente verificar que los segmentos se intersecan con lo visto en la sección anterior, y en caso de que sí se intersequen, usar la función de intersección para calcular el punto de intersección.

## 5.6. Punto medio

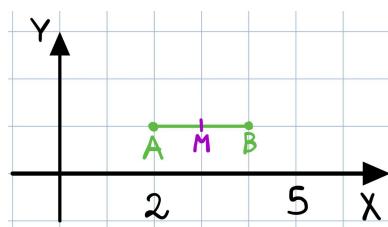
Dados dos puntos  $A = (x_1, y_1)$  y  $B = (x_2, y_2)$  en el plano, el punto medio  $M$ , es el punto que se encuentra exactamente a mitad de camino entre  $A$  y  $B$  a lo largo del segmento  $AB$ .

Este punto  $M$  cumple que:

- $d(A, M) = d(M, B)$ .
- $d(A, M) = \frac{d(A, B)}{2}$ .

Matemáticamente, las coordenadas de  $M$  se determinan como:  $M = \left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)$ .

Visualmente:



En C++ se implementa como:

```
// Dados dos puntos a y b, calcula su punto medio
Vec punto_medio(Vec a, Vec b){
    return Vec((a.x + b.x) / 2, (a.y + b.y) / 2);
}
```

## 6. Polígonos

### 6.1. Definición y representación

Un polígono, es una figura geométrica plana, que está limitada por una secuencia finita de segmentos rectos consecutivos, los cuales se cierran para formar un circuito cerrado. ¿Qué significa todo esto? Significa que el final de cada segmento que forma el polígono, es el comienzo del siguiente, y el final del último segmento es el comienzo del primero.

Los segmentos que limitan el polígono se llaman **lados**, y los puntos donde dos lados se encuentran se llaman **vértices**.

Visualmente (ejemplo ilustrativo)



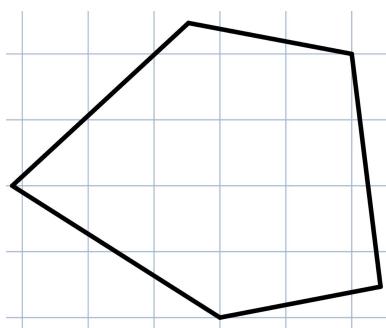
En C++ podemos representar un polígono como un vector **ordenado** de puntos, es decir `vector<Vec> poligono`.

### 6.2. Otras definiciones

- **Polígono convexo:** un polígono se dice que es convexo, si para cualquier par de puntos dentro del polígono, el segmento que conecta esos puntos está completamente contenido dentro del polígono.

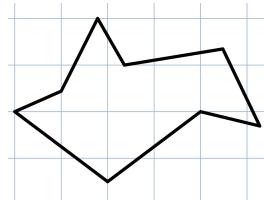
Una forma de verlo es que si trazamos una línea recta entre cualquier par de puntos dentro del polígono, esa línea no sale del polígono en ningún momento.

De forma equivalente, si tomamos dos vértices consecutivos de un polígono convexo, y trazamos una línea recta entre ellos, todos los vértices del polígono, estarán en el mismo lado (semiplano) de esa línea.

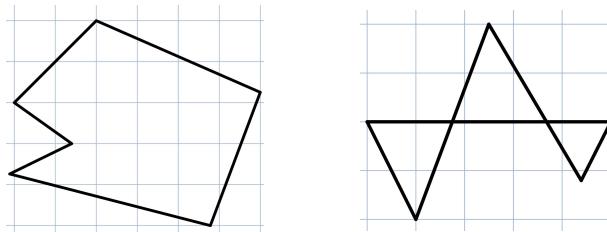


- **Polígono cóncavo:** un polígono es cóncavo si **NO** es convexo. ¿Qué quiere decir esto? Esto significa que existe al menos un par de puntos dentro del polígono, tal que el segmento que los conecta no está completamente contenido dentro del polígono.

Otra forma de verlo, es que si al igual que antes, tomamos dos vértices consecutivos de un polígono cóncavo, y trazamos una línea recta entre ellos, al menos uno de los vértices del polígono está en el lado opuesto de esa línea (distinto semiplano).



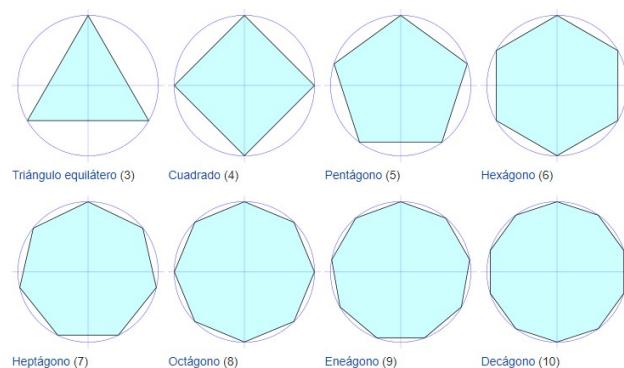
- **Polígono simple:** un polígono es simple si no se cruza a sí mismo. Esto significa que más allá de los puntos en los que los segmentos consecutivos se encuentran (es decir, los vértices), no hay otros puntos en común entre los segmentos. En otras palabras, ningún lado del polígono corta o interseca a otro lado, excepto en los vértices.



Por otro lado, vale la pena mencionar que simple **NO** implica convexo.

- **Polígono regular:** es un polígono en donde todos los lados tienen la misma longitud, y todos los ángulos tienen la misma medida. Algunos ejemplos comunes de polígonos regulares:

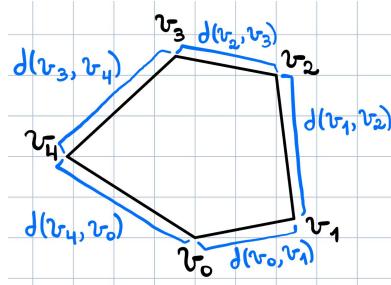
- Triángulo equilátero: triángulo con todos los lados de igual longitud, y tres ángulos de  $60^\circ$
- Cuadrado: cuadrilátero con cuatro lados de igual longitud, y cuatro ángulos rectos de  $90^\circ$
- etc.



### 6.3. Perímetro de un polígono

El perímetro de un polígono, es la suma de las longitudes de sus lados.

Para calcular el perímetro, podemos aprovechar que ya definimos previamente una función que calculaba la distancia entre 2 puntos. Luego, para calcular el perímetro, simplemente debemos sumar las distancias de cada vértice al siguiente dentro del polígono.



Implementación en C++:

```
double perimetro_poligono(const vector<Vec>& poligono){  
    double per = 0;  
    int n = poligono.size();  
    for (int i = 0; i < n - 1; i++){  
        per += dist(poligono[i], poligono[i + 1]);  
    }  
    // Conectar el último vértice con el primero  
    per += dist(poligono[n - 1], poligono[0]);  
  
    return per;  
}
```

Recordar la función dist:

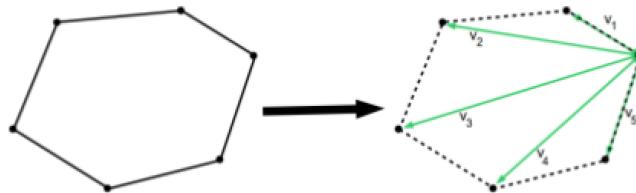
```
double dist(Vec A, Vec B){  
    Vec AB = B - A;  
    return AB.norm();  
}
```

## 6.4. Área de un polígono

A priori, no pareciera haber una manera sencilla de obtener el área de un polígono, sin embargo esto se puede hacer **triangulando**. Veamos de qué se trata:

Lo que hacemos es elegir un punto (generalmente el origen, o el primer vértice del polígono), y se conecta ese punto con cada par de vértices consecutivos del polígono. De esta manera, cada par de vértices, junto con el punto elegido, forman un triángulo. Así, la suma de las áreas de todos los triángulos, da como resultado el área del polígono.

Visualmente:



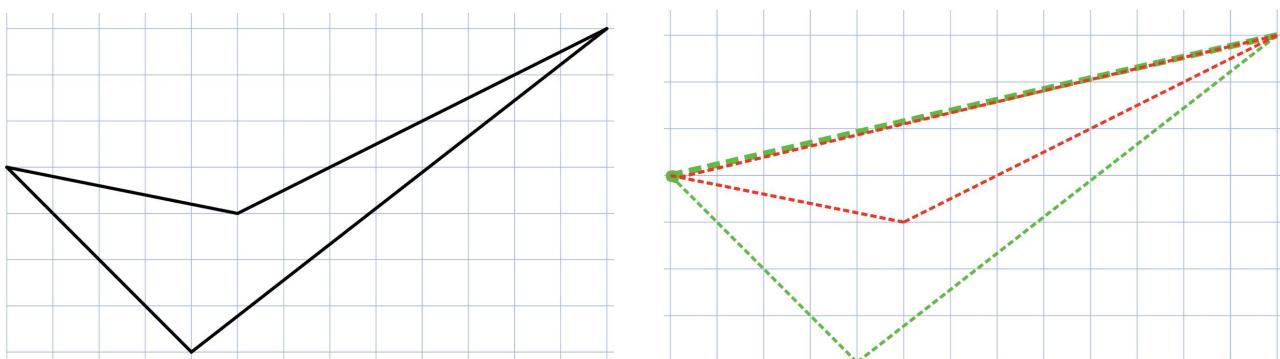
Ya vimos anteriormente cómo calcular el área de un triángulo, utilizando el producto cruz. Más concretamente, pensando en el área del paralelogramo formado por los vectores / 2.

Este método de las triangulaciones, tiene complejidad  $O(n)$ , donde  $n$  es la cantidad de vértices del polígono.

Antes de pasar a la implementación, hay que tener en cuenta una cosa.

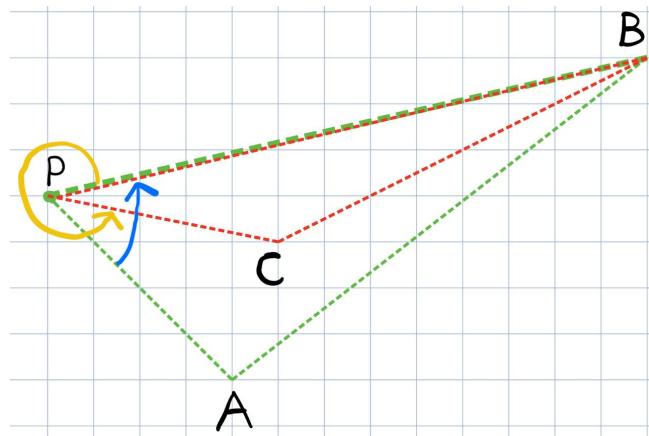
El ejemplo que vimos antes, era para un polígono convexo. Ahora, **¿Qué pasa con un polígono cóncavo?** ¿El algoritmo sigue funcionando?

Veamos qué sucede al triangular con un polígono cóncavo:



Como podemos observar, nos quedaron dos triángulos. Uno verde, que cubre todo el polígono y más (excedente), y uno rojo, que está afuera del polígono, y que de hecho **es exactamente el excedente**.

Aquí es donde está la clave:



Si hacemos el producto cruz para obtener el área del triángulo verde, el **ángulo nos queda así (de A hasta B, antihorario)**, por lo que el área del **triángulo verde** queda positiva.

En cambio, para el **triángulo rojo**, cuando hacemos el producto cruz, el **ángulo nos queda así (cóncavo, antihorario)**, que quiere decir que el área del **triángulo rojo** es **negativa**.

Entonces cuando sumemos las áreas de los triángulos:

Área = **PAB** + **PBC**, por lo que **PBC** nos saca el excedente que tiene **PAB**.

## 6.5. Problema CSES: Polygon Area

**Time limit:** 1.00 s   **Memory limit:** 512 MB

Your task is to calculate the area of a given polygon.

The polygon consists of  $n$  vertices  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ . The vertices  $(x_i, y_i)$  and  $(x_{i+1}, y_{i+1})$  are adjacent for  $i = 1, 2, \dots, n - 1$ , and the vertices  $(x_1, y_1)$  and  $(x_n, y_n)$  are also adjacent.

### Input

The first input line has an integer  $n$ : the number of vertices.

After this, there are  $n$  lines that describe the vertices. The  $i$ th such line has two integers  $x_i$  and  $y_i$ .

You may assume that the polygon is simple, i.e., it does not intersect itself.

### Output

Print one integer:  $2a$  where the area of the polygon is  $a$  (this ensures that the result is an integer).

### Constraints

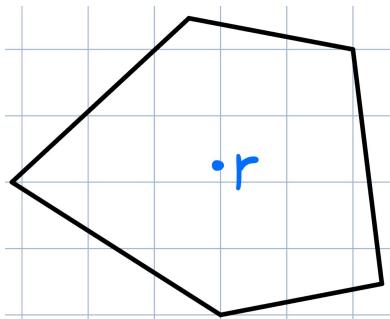
- $3 \leq n \leq 1000$
- $-10^9 \leq x_i, y_i \leq 10^9$

Es en esencia lo que acabamos de ver, hecho problema de competencia con juez online. El código del problema se encuentra en el apéndice.

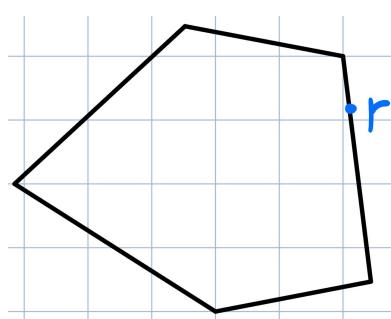
## 6.6. Determinar si un punto se encuentra en un polígono

Dado un punto  $r$  y un polígono  $p$ , queremos saber si  $r$  está adentro de  $p$  (o si está en el borde).

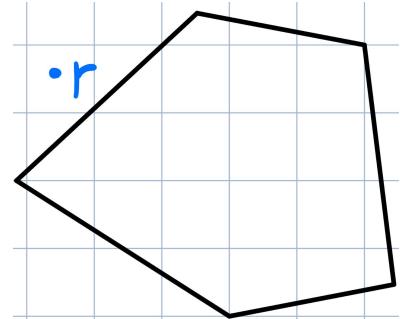
Visualmente:



$r$  adentro del polígono  $p$



$r$  en el borde del polígono  $p$



$r$  afuera del polígono  $p$

### Definición auxiliar: Rayo/Semirrecta

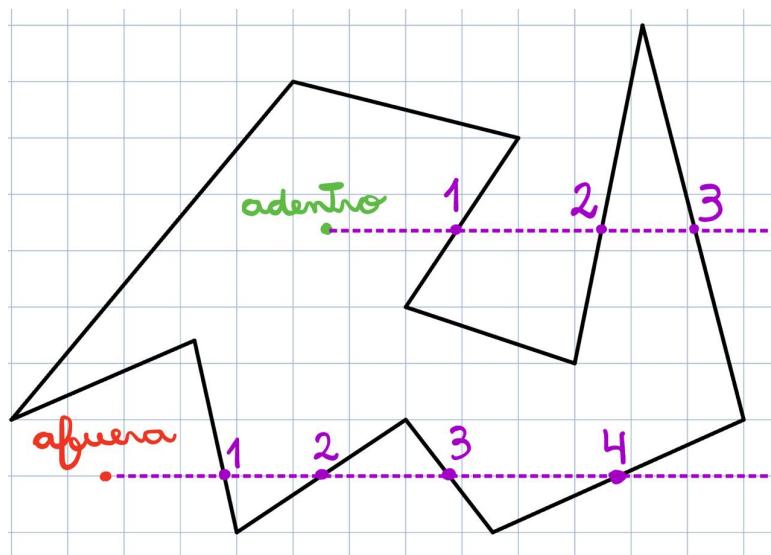
Previamente vimos que una *recta* es una sucesión infinita de puntos **que se extiende en dos sentidos opuestos infinitamente**.

Un *rayo* o *semirrecta* comienza en un punto, y se extiende infinitamente **en un solo sentido**.

Para resolver esto, podemos usar una variante del algoritmo de *Ray Casting*.

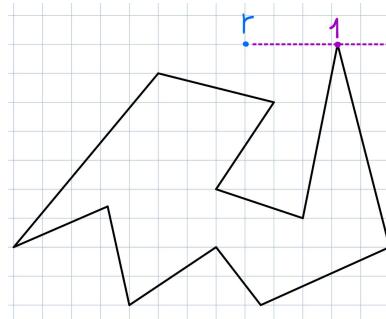
Este algoritmo consiste en lanzar un rayo desde el punto de interés (a priori, en cualquier dirección), y contar la cantidad de veces que el rayo interseca con los lados/bordes del polígono.

- Si toca una cantidad **par**, quiere decir que el punto está **afuera** del polígono.
- Si toca una cantidad **ímpar**, quiere decir que el punto está **adentro** del polígono.



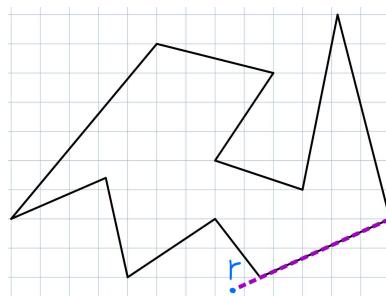
Ahora, no es tan simple la situación, ya que hay algunos casos borde que pueden ocurrir. Por ejemplo:

- El rayo pasa por un vértice del polígono:

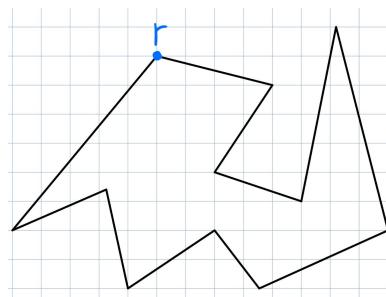


Notemos que el rayo toca 1 vez (cantidad impar) a los bordes del polígono, pero sin embargo el punto  $r$  está afuera del polígono.

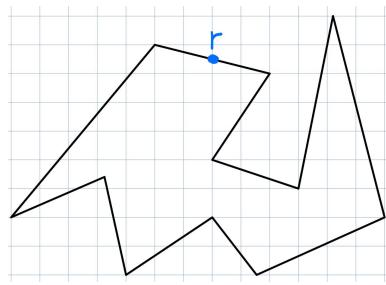
- La dirección del rayo, coincide con uno de los lados del polígono:



- El punto de interés está sobre un vértice del polígono:



- El punto de interés está sobre un lado del polígono:



Aquí es cuando utilizamos la variante del algoritmo de *Ray Casting*, que es aporte de Lucas Hernán Tarche, que simplifica la resolución de este problema.

La variante consiste en tirar el rayo en una dirección muy particular, en vez de simplemente "a la derecha", o en cualquier otra dirección. Dicha dirección, es la  $(1e9 + 7, 1e9 + 9)$ .

¿Por qué en esta dirección? Quizás a primera vista parecer que es una heurística, pero en realidad, esta elección no falla si los puntos enteros sobre la semirecta, caen afuera del rango en el que se encuentran los puntos.

La idea supone que el rango de los puntos, es  $-1e9 \leq x, y \leq 1e9$  para las coordenadas  $(x, y)$  de los mismos (aunque se puede adaptar según el caso, ya que los problemas suelen especificar las cotas del input).

En términos de la representación  $y = mx + b$ , la semirecta que constituye el rayo que estamos lanzando, tiene pendiente  $m = \frac{1e9+9}{1e9+7}$ . Además, sin pérdida de generalidad, asumimos que el punto de interés es  $(0, 0)$ , ya que en caso contrario, podemos trasladar el origen al punto de interés. En otras palabras, estamos lanzando el rayo desde  $(0, 0)$ , por eso sabemos que  $b = 0$ .

Observar que el rayo lanzado termina siendo la semirecta  $y = \frac{1e9+9}{1e9+7}x$ , donde  $(1e9 + 7)$  y  $(1e9 + 9)$  son coprimos, y  $x$  e  $y$  son enteros. A partir de su coprimalidad, podemos deducir que la fracción que conforma la pendiente es irreducible.

Así, si buscamos los puntos  $(x, y)$  con coordenadas enteras, tenemos que para que  $y$  sea entero,  $x$  tiene que ser múltiplo del denominador, es decir,  $x = (1e9 + 7) * k$  con  $k$  entero. De esta manera, el único punto con coordenadas enteras que tiene esa pinta, y que está en el rango especificado del problema, es el  $(0, 0)$ . Si tenemos en cuenta que las coordenadas de los vértices del polígono son enteras, entonces sabemos que el rayo no pasa por ningún vértice del polígono (a menos que el polígono tenga un vértice en  $(0, 0)$ ). Con esto estamos evitando en gran medida el primer caso borde.

Respecto al segundo caso borde mencionado, notemos que si la dirección del rayo, coincide con uno de los lados del polígono, necesariamente sucede que el rayo pasa por un vértice del polígono (que es el primer caso borde que ya tratamos). Es decir, tenemos que *pasa por lado  $\Rightarrow$  pasa por vértice*. A partir de esta observación, y utilizando el contrarrecíproco, tenemos que si *no pasa por vértice  $\Rightarrow$  no pasa por lado*, y como ya con lo explicado para el primer caso borde, vimos que el rayo no pasa por un vértice del polígono (a menos que el polígono tenga un vértice en  $(0, 0)$ ), tenemos que tampoco pasa por uno de sus lados.

De esta manera, los primeros dos casos borde mencionados, no terminan siendo un obstáculo.

Veamos cómo podemos manejar los casos restantes.

Para los casos en que el punto de interés está sobre un vértice o lado del polígono, en el algoritmo podemos incluir una verificación previa **antes** de lanzar el rayo para contar intersecciones, comprobando si el punto coincide con algún vértice del polígono, o si se encuentra en uno de sus lados. En caso afirmativo, podemos responder correctamente que el punto se encuentra en el borde del polígono.

## 6.7. Problema CSES: Point in Polygon

**Time limit:** 1.00 s   **Memory limit:** 512 MB

You are given a polygon of  $n$  vertices and a list of  $m$  points. Your task is to determine for each point if it is inside, outside or on the boundary of the polygon.

The polygon consists of  $n$  vertices  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ . The vertices  $(x_i, y_i)$  and  $(x_{i+1}, y_{i+1})$  are adjacent for  $i = 1, 2, \dots, n - 1$ , and the vertices  $(x_1, y_1)$  and  $(x_n, y_n)$  are also adjacent.

### Input

The first input line has two integers  $n$  and  $m$ : the number of vertices in the polygon and the number of points.

After this, there are  $n$  lines that describe the polygon. The  $i$ th such line has two integers  $x_i$  and  $y_i$ .

You may assume that the polygon is simple, i.e., it does not intersect itself.

Finally, there are  $m$  lines that describe the points. Each line has two integers  $x$  and  $y$ .

### Output

For each point, print "INSIDE", "OUTSIDE" or "BOUNDARY".

### Constraints

- $3 \leq n, m \leq 1000$
- $1 \leq m \leq 1000$
- $-10^9 \leq x_i, y_i \leq 10^9$
- $-10^9 \leq x, y \leq 10^9$

Es en esencia lo que acabamos de ver, hecho problema de competencia con juez online. El código del problema se encuentra en el apéndice.

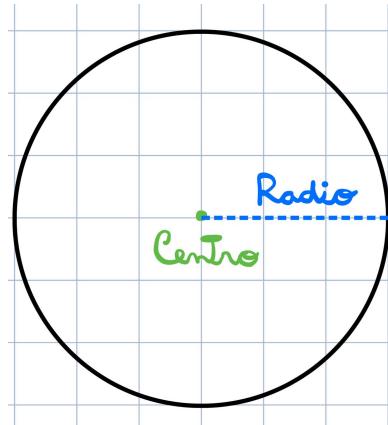
El algoritmo propuesto para resolver el problema, tiene complejidad  $O(n * m)$  donde  $n$  es la cantidad de vértices del polígono, y  $m$  es la cantidad de puntos.

## 7. Círculos

### 7.1. Definición y representación

Un círculo es una entidad geométrica que está únicamente determinada por su **centro** (un punto en el plano) y su **radio** (distancia desde el centro hasta cualquier punto en la circunferencia).

Visualmente:



#### Circunferencia

Es el conjunto de puntos en un plano, que se encuentra a una distancia fija (el radio) de un punto que actúa como centro de un círculo.

Intuitivamente podemos decir que es la línea que define el borde del círculo.

En C++ lo podemos representar como:

```
struct Circulo{
    Vec centro;
    int radio;

    Circulo(Vec c, int r) : centro(c), radio(r) {}

};
```

## 7.2. Problema Codeforces 33D: Knights

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

Berland is facing dark times again. The army of evil lord Van de Mart is going to conquer the whole kingdom. To the council of war called by the Berland's king Valery the Severe came  $n$  knights. After long discussions it became clear that the kingdom has exactly  $n$  control points (if the enemy conquers at least one of these points, the war is lost) and each knight will occupy one of these points.

Berland is divided into  $m + 1$  regions with  $m$  fences, and the only way to get from one region to another is to climb over the fence. Each fence is a circle on a plane, no two fences have common points, and no control point is on the fence. You are given  $k$  pairs of numbers  $a_i, b_i$ . For each pair you have to find out: how many fences a knight from control point with index  $a_i$  has to climb over to reach control point  $b_i$  (in case when Van de Mart attacks control point  $b_i$  first). As each knight rides a horse (it is very difficult to throw a horse over a fence), you are to find out for each pair the minimum amount of fences to climb over.

### Input

The first input line contains three integers  $n, m, k$  ( $1 \leq n, m \leq 1000, 0 \leq k \leq 100000$ ). Then follow  $n$  lines, each containing two integers  $Kx_i, Ky_i$  ( $-10^9 \leq Kx_i, Ky_i \leq 10^9$ ) — coordinates of control point with index  $i$ . Control points can coincide.

Each of the following  $m$  lines describes fence with index  $i$  with three integers  $r_i, Cx_i, Cy_i$  ( $1 \leq r_i \leq 10^9, -10^9 \leq Cx_i, Cy_i \leq 10^9$ ) — radius and center of the circle where the corresponding fence is situated.

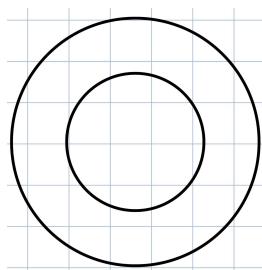
Then follow  $k$  pairs of integers  $a_i, b_i$  ( $1 \leq a_i, b_i \leq n$ ), each in a separate line — requests that you have to answer.  $a_i$  and  $b_i$  can coincide.

### Output

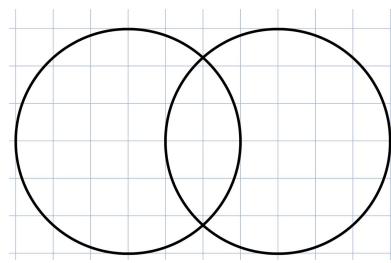
Output exactly  $k$  lines, each containing one integer — the answer to the corresponding request.

Dejando de lado la historia del problema, y pensándolo más en términos "geométricos", tenemos puntos y círculos. Van a haber muchas consultas, en donde en cada una de ellas, tendremos dos puntos de interés, así que para simplificar la explicación, pensemos que sólo tenemos dos puntos  $p1$  y  $p2$ .

Los círculos en el problema, actúan como cercas. Pueden haber círculos que adentro contengan otros círculos, pero no hay intersecciones de círculos, es decir, no hay una contención "parcial" entre un círculo y otro.



Esto puede pasar



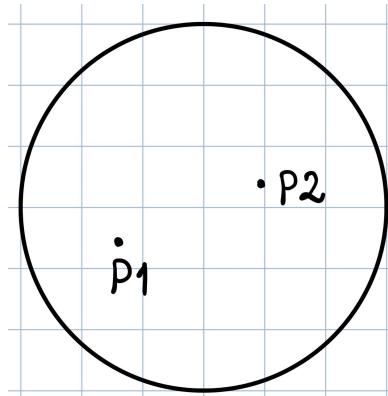
Esto no puede pasar

Además, nos garantizan que los puntos de interés no están en los bordes de los círculos (es decir, en las circunferencias), y que los puntos pueden coincidir (ser iguales).

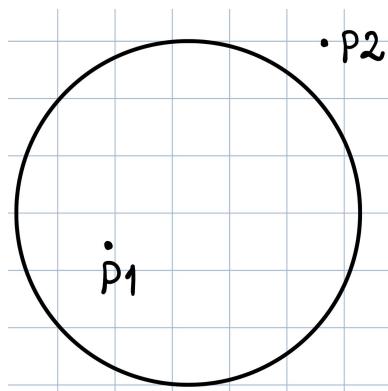
El objetivo es contar cuál es la mínima cantidad de cercas que tenemos que cruzar, para llegar de un punto al otro.

Una forma de resolver este problema usando geometría, es aprovechar la siguiente idea:

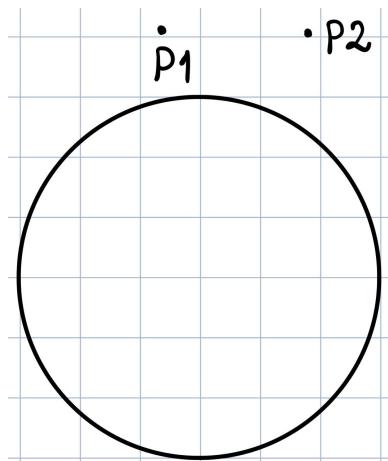
- Si un círculo contiene tanto a  $p_1$  como a  $p_2$ , entonces no necesitamos cruzar por ese círculo.



- Si un círculo contiene a  $p_1$  pero no a  $p_2$  (o viceversa), entonces sí o sí tenemos que cruzar por ese círculo.



- Si un círculo no contiene ni a  $p_1$  ni a  $p_2$ , entonces al igual que en el primer caso, no necesitamos cruzar por ese círculo.



Para determinar si un círculo  $C$  contiene o no a un punto determinado  $p$ , podemos calcular la distancia desde el centro de  $C$  hasta  $p$ , y si esa distancia es mayor al radio de  $C$ , entonces sabemos que  $C$  no contiene a  $p$ .

Análogamente, si la distancia resultante es menor al radio de  $C$ , entonces  $C$  contiene a  $p$ .

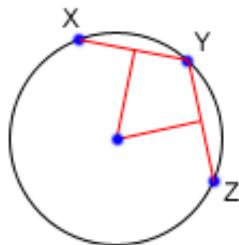
Observar que no miramos el caso en que la distancia es igual al radio de  $C$ , ya que el enunciado dice que un punto no puede estar exactamente en el borde de un círculo.

De esta manera, para resolver el problema podemos ver cuáles son los círculos que contienen a un punto de interés pero no al otro, e incrementar un contador cuando eso sucede. Como hay muchas consultas, hay que repetir el proceso para cada consulta, reiniciando el contador cada vez.

El algoritmo propuesto para resolver este problema, tiene complejidad  $O(m(n + k))$ , donde  $n$  es la cantidad de puntos,  $m$  es la cantidad de círculos y  $k$  es la cantidad de consultas.

### 7.3. Encontrar un círculo a partir de 3 puntos

Dados 3 puntos que no son colineales, esos puntos definen únicamente un círculo. Entonces, ¿Cómo podemos encontrar el centro y el radio de ese círculo?

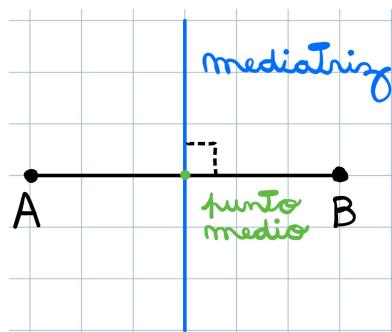


#### Definición auxiliar: Mediatrix

La mediatrix de un segmento  $AB$  es la recta que cumple con dos condiciones:

- Es perpendicular al segmento  $AB$ .
- Pasa por el punto medio del segmento  $AB$ .

Visualmente:



Comenzamos encontrando el vector dirección de la recta  $XY$ , que es el vector  $v = Y - X$  (vector que va desde  $X$  hasta  $Y$ ).

Luego, calculamos su punto medio  $p$ , donde  $p = \frac{X+Y}{2}$ .

Una vez hayado el punto medio, debemos hallar la dirección perpendicular a la recta  $XY$ . Para esto, podemos recordar lo visto en la sección *Rotación horaria y antihoraria de puntos/vectores*, donde vimos que para un vector dirección  $(a, b)$  en el plano, un vector perpendicular a este es  $(-b, a)$  o  $(b, -a)$ .

Ahora necesitamos dar la ecuación de la mediatrix. La misma es  $L(t) = p + t * v_{perp}$ , donde  $v_{perp}$  es alguna de las dos opciones del paso anterior (las dos sirven porque las rectas se extienden infinitamente en ambos sentidos).

Repetimos los pasos anteriores para los pares de puntos  $Y$  y  $Z$ .

Una vez que ya tenemos ambas mediatrixes, calculamos el punto de intersección entre ellas usando lo que vimos en la sección *Determinar en qué punto se intersecan dos rectas* (**Nota:** como los puntos originales no son colineales, es seguro que las mediatrixes se intersecan). El punto de intersección es el **centro** del círculo.

Para obtener el **radio** del círculo, simplemente calculamos la distancia desde el **centro**, a cualquiera de los 3 puntos originales.

Implementación en C++:

```
// Se asume que X, Y y Z no son colineales
Circulo encontrar_circulo(Vec X, Vec Y, Vec Z) {
    // Primero, encontramos los puntos medios de los segmentos XY e YZ
    Vec puntoMedioXY = punto_medio(X, Y);
    Vec puntoMedioYZ = punto_medio(Y, Z);

    // Después, hallamos las direcciones perpendiculares a
    // los segmentos XY e YZ
    Vec direccionPerpXY = (Y - X).rotar90Antihorario();
    Vec direccionPerpYZ = (Z - Y).rotar90Antihorario();

    // Luego, creamos las mediatrices como rectas
    // con la representación punto de paso + dirección
    Recta mediatrixXY(puntoMedioXY, direccionPerpXY);
    Recta mediatrixYZ(puntoMedioYZ, direccionPerpYZ);

    // Encontramos el centro del círculo, que es la intersección de
    // las mediatrices
    Vec centro = interseccion(mediatrixXY, mediatrixYZ);

    // Calculamos el radio del círculo como la distancia del centro a
    // cualquiera de los puntos originales
    double radio = (centro - X).norm();

    // Creamos y devolvemos el círculo
    return Circulo(centro, radio);
}
```

## 8. Comentarios finales

### 8.1. Más aplicaciones prácticas de la geometría computacional

- Robótica
  - Los brazos robóticos en la industria utilizan geometría para determinar cómo moverse y manipular objetos sin colisionar con otros objetos del entorno.
- Procesamiento de imágenes y visión por computadora
  - Se utiliza para detectar caras en imágenes, leer patentes de autos, identificar objetos, etc. mediante el reconocimiento de patrones con algoritmos geométricos.
- Bioinformática
  - Para entender mejor cómo funcionan las proteínas y otras biomoléculas, se modelan sus estructuras 3D. Estos modelos se construyen y comparan usando técnicas de geometría computacional.
- Manufactura y producción
  - En industrias donde se cortan materiales (como tela o metal), la geometría computacional ayuda a optimizar la disposición de las piezas para minimizar el desperdicio.
- Aeronáutica
  - Se realizan simulaciones basadas en geometría computacional, para optimizar el diseño de los aviones, y mejorar la eficiencia del combustible y la aerodinámica.

### 8.2. Más ejercicios para practicar

Algunos problemas para intentar después de haber leído el apunte:

- Problema F del TAP 2023.
- Problem - 1791B - Codeforces.
- UVA: 10263 Railway.
- UVA: 11227 The silver bullet.

### **8.3. Algunos tópicos para profundizar más en la geometría computacional**

- Teorema de Pick.
- Técnicas de barrido como Sweep Line.
- Rotating calipers.
- Par de puntos más cercanos y más lejanos.
- Convex Hull.
- Geometría computacional 3D.

## 8.4. AGM Euclídeo con Triangulación de Delaunay

Para no quemar ninguno de los otros tópicos sugeridos, decidí abordar otro tema. Más concretamente, nos enfocamos en el problema de obtener el AGM Euclídeo, donde los costos de las aristas corresponden a distancias Euclídeas entre los puntos incidentes.

Este problema es muy relevante, y tiene una aplicación real directa, que se puede ver reflejada en por ejemplo, minimizar los costos para conectar ciertas ciudades, aunque también aparece en otras áreas como las redes de transporte y las comunicaciones.

En el contexto de este problema, consideramos que potencialmente todos los puntos están conectados con todos (es decir, es un grafo completo). En este sentido, si quisieramos obtener el AGM, podríamos hacerlo con una complejidad  $O(n^2)$  utilizando Prim o Kruskal, ya que el grafo es denso.

Si bien la complejidad es polinomial en función del tamaño de la entrada, su característica cuadrática representa un problema cuando se quiere trabajar con grafos con muchos nodos, haciendo que el algoritmo sea prácticamente inutilizable para conjuntos de datos muy grandes, o para aplicaciones en tiempo real.

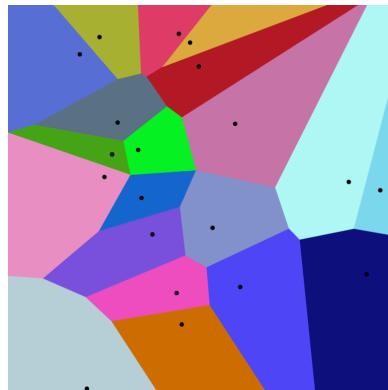
Por esta razón, nos centraremos en un enfoque más refinado, que aprovecha la geometría del problema para mejorar significativamente este aspecto, reduciendo la complejidad a  $O(n \log n)$ , mediante el uso de la *Triangulación de Delaunay*.

#### 8.4.1. Diagrama de Voronoi

Dado un conjunto  $S$  de  $n$  puntos, un diagrama de Voronoi es una partición del plano en distintas regiones. Cada región, está asociada a un punto  $p \in S$ , y además, contiene a todos los puntos  $q$  tal que  $q$  está más cerca de  $p$  que cualquier otro punto  $p' \in S$ .

En otras palabras, un punto  $q$  pertenece a la región de su "vecino" más cercano.

Visualmente:



Un diagrama de Voronoi puede ser representado por el grafo planar de los bordes entre regiones.

En ese sentido, hay un **vértice** en este grafo cuando 3 o más segmentos se intersecan, y una **arista** es un segmento que conecta a dos de estos vértices. En algunos casos, los segmentos pueden extenderse hasta el infinito, creando vértices en el infinito cuando las aristas no están contenidas dentro de un límite finito en el plano.

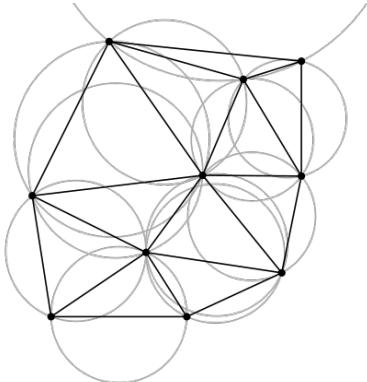
Debido a cómo se definieron las regiones, un segmento contiene a todos los puntos que son equidistantes a los dos puntos más cercanos en  $S$ , que denotan las regiones adyacentes. Esto significa que estos segmentos son las mediatrixes de la línea que conectaría a estos dos puntos.

Combinando los dos párrafos anteriores, tenemos que como un vértice en un diagrama de Voronoi, es un punto donde se encuentran 3 o más bordes de las regiones de Voronoi, y además, cada borde es equidistante a dos puntos en  $S$ , entonces un vértice de Voronoi es equidistante a 3 o más puntos de  $S$ .

En la geometría de triángulos, el punto equidistante a los 3 vértices de un triángulo es el circuncentro, y se encuentra en la intersección de las mediatrixes de los lados del triángulo (lo que vimos en la sección *Encontrar un círculo a partir de 3 puntos*).

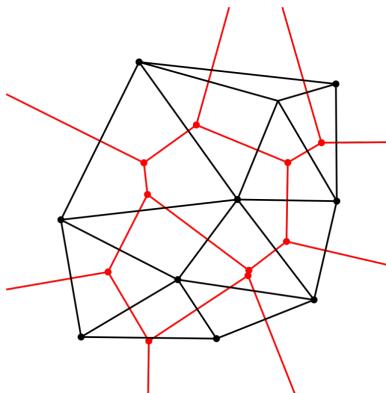
### 8.4.2. Triangulación de Delaunay

La triangulación de Delaunay es un conjunto de triángulos conectados, que cubren una región sin superposiciones. Estos triángulos tienen una propiedad especial: para cada triángulo, la circunferencia que pasa por sus vértices no contiene ningún otro punto de  $S$  en su interior.



Los puntos negros son los que conforman el conjunto  $S$

### 8.4.3. Dualidad entre los Diagramas de Voronoi y las Triangulaciones de Delaunay



\* El conjunto de puntos  $S$  son los puntos negros.

\* Los vértices y aristas rojas son del diagrama de Voronoi.

\* Los puntos y aristas negros son de la triangulación de Delaunay.

La triangulación de Delaunay es el grafo *dual* del diagrama de Voronoi:

- Cada vértice en el diagrama de Voronoi corresponde a un triángulo en la triangulación de Delaunay (y viceversa).
- Una arista en la triangulación de Delaunay conecta dos puntos  $\Leftrightarrow$  las regiones correspondientes en el diagrama de Voronoi, comparten un borde.

Generalmente, la triangulación de Delaunay precede a la del diagrama de Voronoi debido a su relativa simplicidad computacional y porque evita el manejo de vértices en el infinito. Esta priorización resulta práctica no sólo en cuanto a la eficiencia, sino también porque la triangulación es suficiente para muchas aplicaciones que requieren la estructura subyacente de los datos, antes de considerar la división de regiones más compleja que conlleva el diagrama de Voronoi. Así, la triangulación de Delaunay, se convierte en un paso intermedio sobre el cual se puede construir el diagrama de Voronoi si se necesita, permitiendo un enfoque modular.

#### 8.4.4. Utilidad

¿Por qué nos interesa esto? Bueno, resulta que el AGM Euclídeo es un subconjunto de las aristas de la triangulación de Delaunay.

Esto implica que si ya tenemos una Triangulación de Delaunay, encontrar el AGM Euclídeo es más eficiente, porque sólo tenemos que buscar el conjunto de aristas que forman el AGM, dentro una cantidad limitada de aristas (las de la triangulación), en lugar de considerar todas las posibles conexiones entre puntos.

Ahora bien, si la construcción de la triangulación de Delaunay es muy costosa, entonces no termina siendo del todo provechoso lo dicho en el párrafo anterior. Sin embargo, este no es el caso, ya que se puede hacer en  $O(n \log n)$ , ordenando los puntos como primer paso (el hecho de que los puntos estén ordenados, permite que se pueda construir la triangulación más eficientemente), y usando técnicas de barrido como Sweep Line.

Una vez construida la triangulación, podemos buscar el AGM usando Prim o Kruskal, aprovechando que la misma tiene muchas menos aristas (es decir, es un grafo ralo, ya que a lo sumo tiene  $3n - 6$  aristas) en tiempo  $O(n \log n)$ .

#### 8.4.5. Motivación y Bibliografía relacionada

Si bien aquí se ofreció un pantallazo general sobre estos temas, realmente son muy amplios, y tiene muchas más aplicaciones de las que se mencionaron en este apunte. Dichas aplicaciones abarcan a la Programación Competitiva (problemas que se pueden resolver eficientemente usando triangulaciones de Delaunay o diagramas de Voronoi), como aspectos de la vida real o la industria, como por ejemplo:

- En la meteorología, los diagramas de Voronoi pueden ayudar en la interpretación de los datos de los sensores, para predecir patrones climáticos.
- En biología, se usan estos conceptos para comprender la estructura de los tejidos y la organización celular.

Por eso, para quienes quieran profundizar más en estos temas, les recomiendo la siguiente bibliografía:

- Voronoi Diagram - Wikipedia.
- Delaunay triangulation - Wikipedia.
- [Tutorial] Voronoi Diagram and Delaunay Triangulation in  $O(n \log n)$  with Fortune's Algorithm. Excelente blog de Codeforces hecho por Monogon.
- De Berg, M., Cheong, O., Van Kreveld, M., & Overmars, M. (2011). Computational Geometry: Algorithms and Applications (3rd ed.). Springer.

## 9. Bibliografía

Algunos de los recursos que utilicé para preparar este apunte son:

### Sitios web:

- CP Algorithms: Basic Geometry - Algorithms for Competitive Programming.
- Topcoder: Geometry Concepts part 1: Basic Concepts (es un blog que tiene 3 partes).
- CodeForces: Catalog - Codeforces (tiene una sección de geometría con varios posts interesantes y con distinta complejidad).
- Wikipedia.
- Training Camp Argentina (Edición 2021).
- Canal de YouTube de Errichto: Errichto Algorithms - YouTube. Tiene muchos videos de temas de Programación Competitiva, incluyendo geometría.

### Libros:

- Handbook of geometry for competitive programmers. Muy buen recurso de geometría orientado a Programación Competitiva, por Victor Lecomte.
- Halim, S., Halim, F., & Effendy, S. (2020). Competitive Programming 4 (4th ed.)
- Competitive Programmer's Handbook de Antti Laaksonen.
- Laaksonen, A. (2017). Guide to Competitive Programming: Learning and Improving Algorithms Through Contests. Springer.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009).Introduction to Algorithms (3rd ed.). MIT Press.

**Nota:** es importante que sea la 3ra edición, (es decir la de 2009), porque en la 4ta edición que es del 2022, sacaron el apartado de geometría computacional.

## 10. Apéndice (código de los ejercicios)

### 10.1. Problema Codeforces 498A: Crazy Town

```
1 #include <iostream>
2 #include <vector>
3
4 typedef long long ll;
5
6 using namespace std;
7
8 struct Recta {
9     ll a, b, c;
10 };
11
12 // Eval a un punto en una Recta
13 ll evaluar(Recta r, ll x, ll y) {
14     return r.a * x + r.b * y + r.c;
15 }
16
17 int main() {
18     ios::sync_with_stdio(false);
19     cin.tie(0);
20
21     ll x1, y1, x2, y2;
22     cin >> x1 >> y1 >> x2 >> y2;
23
24     ll n;
25     cin >> n;
26
27     vector<Recta> rectas(n);
28     for (int i = 0; i < n; ++i) {
29         cin >> rectas[i].a >> rectas[i].b >> rectas[i].c;
30     }
31
32     int count = 0;
33     for (Recta r : rectas) {
34         ll eval1 = evaluar(r, x1, y1);
35         ll eval2 = evaluar(r, x2, y2);
36
37         // Si los puntos se encuentran en distintos semiplanos, incremento
38         // count.
39         if ((eval1 > 0 && eval2 < 0) || (eval1 < 0 && eval2 > 0)) {
40             count++;
41         }
42     }
43
44     cout << count << endl;
45
46     return 0;
47 }
```

## 10.2. Problema CSES: Point Location Test

```
1 #include <iostream>
2
3 using namespace std;
4
5 typedef long long ll;
6
7 struct Vec {
8     // Coordenadas (x, y)
9     ll x, y;
10
11    // Constructor
12    Vec(ll x = 0, ll y = 0) : x(x), y(y) {}
13
14    // Resta
15    Vec operator-(Vec q) {return Vec(x - q.x, y - q.y);}
16
17    // Producto cruz entre dos vectores
18    ll operator^(Vec q) {return x * q.y - y * q.x;}
19};
20
21 // Esta funcion determina la orientacion entre tres puntos p1, p2 y p3
22 string orientacion(Vec p1, Vec p2, Vec p3) {
23     Vec v1 = p2 - p1; // Vector p1p2
24     Vec v2 = p3 - p1; // Vector p1p3
25
26     ll producto_cruz = v1 ^ v2; // Usamos el operador ^ para el producto
cruz
27
28     if(producto_cruz < 0) {
29         return "RIGHT";
30     }
31     else if(producto_cruz > 0) {
32         return "LEFT";
33     }
34     else {
35         return "TOUCH";
36     }
37}
38
39 void test_case() {
40     ll x1, y1, x2, y2, x3, y3;
41
42     // Leemos las coordenadas de los puntos
43     cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
44
45     // Creamos los puntos
46     Vec p1(x1, y1);
47     Vec p2(x2, y2);
48     Vec p3(x3, y3);
49
50     // Determina la orientacion/posicion de p3 respecto de la recta que
pasa por p1 y p2
51     cout << orientacion(p1, p2, p3) << endl;
52}
53
54 int main() {
55     int T; // Cantidad de prueba
56}
```

```
57     cin >> T;
58     while(T--) {
59         test_case();
60     }
61     return 0;
62 }
```

### 10.3. Problema CSES: Line Segment Intersection

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 typedef long long ll;
7
8 struct Vec{
9     // Coordenadas (x, y)
10    ll x, y;
11
12    // Constructor
13    Vec(ll x = 0, ll y = 0) : x(x), y(y) {}
14
15    // Resta de dos vectores
16    Vec operator-(Vec q) {return Vec(x - q.x, y - q.y);}
17
18    // Producto escalar entre dos vectores
19    ll operator*(Vec v) {return x * v.x + y * v.y;}
20
21    // Comparacion de igualdad con puntos/vectores
22    bool operator==(Vec q) {return x == q.x && y == q.y;}
23
24    // Producto cruz entre dos vectores
25    ll operator^(Vec q) {return x * q.y - y * q.x;}
26};
27
28 // Determina de que lado de la recta formada por a y b, se encuentra c
29 int side(Vec a, Vec b, Vec c) {
30     // Obtenemos los vectores representativos de los segmentos (a, b) y (a, c)
31     Vec v = b - a, w = c - a;
32     ll res = v ^ w;
33     if (res == 0) return 0; // c est en la linea que pasa por a y b
34     if (res > 0) return 1; // c est a la izquierda de la linea
35     return -1;           // c est a la derecha de la linea
36 }
37
38 bool enSegmento(Vec a, Vec b, Vec c) {
39     // Si el punto es uno de los extremos del segmento
40     if (a == c || b == c) return true;
41
42     // Para que c est en el segmento (a, b), sus coordenadas x e y deben
43     // estar en el rango
44     // definido por las coordenadas 'x' e 'y' de a y b.
45     bool xInRange = min(a.x, b.x) <= c.x && c.x <= max(a.x, b.x);
46     bool yInRange = min(a.y, b.y) <= c.y && c.y <= max(a.y, b.y);
47     return xInRange && yInRange;
48 }
49
50 // Verifica si los segmentos (p1, p2) y (p3, p4) se intersecan
51 bool se_intersecan(Vec p1, Vec p2, Vec p3, Vec p4) {
52     // Comprobamos si algun punto de un segmento coincide con algun punto
53     // del otro segmento.
54     // Si dos puntos coinciden, entonces los segmentos se intersecan en ese
55     // punto
56     if (p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4) return true;
```

```

55 // Determinamos la posici n relativa de p3 con respecto a (p1, p2) y
56 // sucesivamente
57 int s1 = side(p1, p2, p3);
58 int s2 = side(p1, p2, p4);
59 int s3 = side(p3, p4, p1);
60 int s4 = side(p3, p4, p2);

61 // Verificamos si los puntos son colineales.
62 // Si lo son, tenemos que chequear si alguno de los puntos del
63 // primer segmento est dentro del segundo segmento o viceversa.
64 if (s1 == 0 && s2 == 0 && s3 == 0 && s4 == 0) {
65     return enSegmento(p1, p2, p3) || enSegmento(p1, p2, p4) ||
66     enSegmento(p3, p4, p1) ||
67         enSegmento(p3, p4, p2);
68 }

69 // Para que dos segmentos se intersequen, un extremo de un segmento debe
70 // estar de un lado de
71 // la lnea formada por el otro segmento y el otro extremo del otro
72 // lado. Esto tiene que
73 // valer para ambos segmentos.

74 // El uso de <= en lugar de < garantiza que tambi n consideramos el
75 // caso en que un punto
76 // est exactamente en la lnea formada por el otro segmento (s1 o s2
77 // o s3 o s4 es 0).
78 // Esto es consistente con la definici n del problema que dice que si
79 // los segmentos
80 // comparten al menos un punto, se considera que se intersecan
81 // return s1 * s2 <= 0 && s3 * s4 <= 0;
82 }

83

84 int main() {
85     ios::sync_with_stdio(false);
86     cin.tie(0);

87     int n;
88     cin >> n;
89     for (int i = 0; i < n; i++) {
90         Vec p1, p2, p3, p4;
91         cin >> p1.x >> p1.y >> p2.x >> p2.y >> p3.x >> p3.y >> p4.x >> p4.y;
92         if(se_intersecan(p1, p2, p3, p4)){
93             cout << "YES" << endl;
94         }
95         else {
96             cout << "NO" << endl;
97         }
98     }
99     return 0;

```

## 10.4. Problema CSES: Polygon Area

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 typedef long long ll;
7
8 struct Vec {
9     // Coordenadas (x, y)
10    ll x, y;
11
12    Vec operator-(Vec q) { return {x - q.x, y - q.y}; } // Resta de
13    vectores
14    ll operator^(Vec q) { return x * q.y - y * q.x; } // Producto cruz
15    entre dos vectores
16};
17
18 int main() {
19     ios::sync_with_stdio(false);
20     cin.tie(0);
21
22     int n;
23     cin >> n;
24     vector<Vec> vertices(n);
25
26     // Leemos los v rtices/puntos del pol gono
27     for (int i = 0; i < n; i++) {
28         cin >> vertices[i].x >> vertices[i].y;
29     }
30
31     // Calculamos el rea utilizando triangulaciones desde el v rtice 0
32     ll area = 0;
33     for (int i = 1; i < n - 1; i++) {
34         // El tri ngulo formado por los v rtices 0, i e i + 1 del
35         // pol gono
36         Vec v1 = vertices[i] - vertices[0];
37         Vec v2 = vertices[i + 1] - vertices[0];
38
39         // Calculamos el rea del tri ngulo haciendo el producto cruz
40         entre v1 y v2
41         // y acumulamos el resultado en 'area'. De esta manera, en 'area' se
42         van
43         // sumando las reas de todos los tri ngulos, y al finalizar,
44         // tendremos el
45         // el rea del pol gono
46         area += v1 ^ v2;
47     }
48
49     // Tomamos el valor absoluto para obtener el rea
50     // Por qu ? Porque el producto cruz puede dar negativo dependiendo de
51     // la orientaci n
52
53     // Observar que el problema pide '2a' donde 'a' es el rea
54     // del pol gono, por lo que no tenemos que dividir por 2 en este caso
55     cout << abs(area) << endl;
56
57     return 0;
58 }
```

## 10.5. Problema CSES: Point in Polygon

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 typedef long long ll;
7
8 struct Vec{
9     // Coordenadas (x, y)
10    ll x, y;
11
12    // Constructor
13    Vec(ll x = 0, ll y = 0) : x(x), y(y) {}
14
15    // Resta de dos vectores
16    Vec operator-(Vec q) {return Vec(x - q.x, y - q.y);}
17
18    // Producto escalar entre dos vectores
19    ll operator*(Vec v) {return x * v.x + y * v.y;}
20
21    // Comparacion de igualdad con puntos/vectores
22    bool operator==(Vec q) {return x == q.x && y == q.y;}
23
24    // Producto cruz entre dos vectores
25    ll operator^(Vec q) {return x * q.y - y * q.x;}
26};
27
28 // Determina de que lado de la recta formada por a y b, se encuentra c
29 int side(Vec a, Vec b, Vec c) {
30     // Obtenemos los vectores representativos de los segmentos (a, b) y
31     // (a, c)
32     Vec v = b - a, w = c - a;
33     ll res = v ^ w;
34     if (res == 0) return 0; // c est en la linea que pasa por a y b
35     if (res > 0) return 1; // c est a la izquierda de la linea
36     return -1; // c est a la derecha de la linea
37 }
38
39 // Verifica si los segmentos (p1, p2) y (p3, p4) se intersecan
40 bool se_intersecan(Vec p1, Vec p2, Vec p3, Vec p4) {
41     // Determinamos la posicion relativa de p3 con respecto a (p1, p2) y
42     // sucesivamente
43     int s1 = side(p1, p2, p3);
44     int s2 = side(p1, p2, p4);
45     int s3 = side(p3, p4, p1);
46     int s4 = side(p3, p4, p2);
47
48     // Para que dos segmentos se intersequen, un extremo de un segmento debe
49     // estar de un lado de
50     // la linea formada por el otro segmento y el otro extremo del otro
51     // lado. Esta lógica debe
52     // ser validada para ambos segmentos.
53
54     // Es muy improbable que la dirección del rayo coincida con uno de los
55     // lados del polígono debido
56     // a la dirección que elegimos para el mismo, pero incluso en un caso
57     // muy raro en
58     // donde eso sucediera, esta función ignora esas intersecciones, ya que
```

```

    en caso de
    // colinealidad, "s1 * s2 < 0 && s3 * s4 < 0" no va a ser < 0, y por lo
    tanto no lo
    // va a contar como intersección, lo cual es el comportamiento deseado
55
56    // No queremos contar esto como intersección, ya que podríamos
    terminar contando
57    // intersecciones adicionales que no corresponden a un cruce real. M s
    puntualmente,
58    // cada vértice de ese lado podría contarse potencialmente como una
    intersección, dando
59    // un resultado incorrecto
60    return s1 * s2 < 0 && s3 * s4 < 0;
61}
62
63 bool enSegmento(Vec a, Vec b, Vec c) {
64    // Si el punto es uno de los extremos del segmento
65    if (a == c || b == c) return true;
66
67    // Para que c esté en el segmento (a, b), sus coordenadas x e y deben
    estar en el rango
68    // definido por las coordenadas 'x' e 'y' de a y b.
69    bool xInRange = min(a.x, b.x) <= c.x && c.x <= max(a.x, b.x);
70    bool yInRange = min(a.y, b.y) <= c.y && c.y <= max(a.y, b.y);
71    return xInRange && yInRange;
72}
73
74 int main() {
75    ios::sync_with_stdio(false);
76    cin.tie(0);
77
78    int n, m;
79    cin >> n >> m;
80
81    vector<Vec> poligono(n);
82    for(int i = 0; i < n; i++) cin >> poligono[i].x >> poligono[i].y;
83
84    for(int i = 0; i < m; i++) {
85        Vec p;
86        cin >> p.x >> p.y;
87
88        bool boundary = false;
89        // Verificamos si el punto p está en el borde del polígono
90        for(int j = 0; j < n; j++){
91            if(enSegmento(poligono[j], poligono[(j + 1) % n], p) &&
92                side(poligono[j], poligono[(j + 1) % n], p) == 0){
93                boundary = true;
94                break;
95            }
96        }
97
98        if(boundary){
99            cout << "BOUNDARY" << endl;
100            continue;
101        }
102
103        // Creamos un rayo desde el punto p hacia un punto arbitrariamente
    lejano
104        Vec ray(1e9 + 7, 1e9 + 9);
105        int count = 0;

```

```
106     // Contamos cu ntas veces el rayo interseca con los lados del
107     pol gono
108     for(int j = 0; j < n; j++){
109         if(se_intersecan(poligono[j], poligono[(j + 1) % n], p, ray))
110             count++;
111     }
112
113     if(count % 2 == 1){
114         cout << "INSIDE" << endl;;
115     }
116     else {
117         cout << "OUTSIDE" << endl;
118     }
119 }
```

## 10.6. Problema Codeforces 33D: Knights

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 typedef long long ll;
6
7
8 struct Punto{
9     ll x, y;
10    Punto(ll x = 0, ll y = 0) : x(x), y(y) {}
11};
12
13 ll sq_dist(Punto a, Punto b){
14    ll dx = a.x - b.x;
15    ll dy = a.y - b.y;
16    return dx * dx + dy * dy;
17}
18
19 struct Circulo{
20    ll r;
21    Punto centro;
22
23    /*
24     Calculamos la distancia cuadrada en vez de la distancia, porque la
25     distancia
26     puede ser un n mero real que no necesariamente es entero, y nos molesta
27     comparar puntos flotantes
28     */
29    bool contiene_punto(Punto p){
30        // En algunos problemas, este chequeo ser a con    <=    , pero en
31        // este problema nos dicen que no hay puntos sobre la circunferencia
32        return sq_dist(centro, p) < r * r;
33    }
34};
35
36 int main(){
37    // Estas dos l neas hacen que cin/cout sea tan r pido como scanf/
38    printf
39    ios::sync_with_stdio(false);
40    cin.tie(0);
41
42    int n, m, k;
43    /*
44     n: cantidad de puntos
45     m: cantidad de cercas
46     k: cantidad de queries
47     */
48    while(cin >> n >> m >> k){
49        vector<Punto> puntos(n);
50        vector<Circulo> circulos(m);
51
52        for(int i = 0; i < n; i++) cin >> puntos[i].x >> puntos[i].y;
53        for(int i = 0; i < m; i++) cin >> circulos[i].r >> circulos[i].
54        centro.x >> circulos[i].centro.y;
55
56        // Precalculamos qu Puntos contiene cada Circulo
57        vector<vector<bool>> puntosDentroDe(m, vector<bool>(n, false));
58        for(int i = 0; i < n; i++) {
```

```

54         for(int j = 0; j < m; j++) {
55             if(circulos[j].contiene_punto(puntos[i])) {
56                 puntosDentroDe[j][i] = true;
57             }
58         }
59     }
60
61     for(int i = 0; i < k; i++){
62         int a, b;
63         cin >> a >> b;
64         a--; b--; // Para que quede indexado desde 0
65         int count = 0;
66         for(int j = 0; j < m; j++) {
67             if(puntosDentroDe[j][a] != puntosDentroDe[j][b]) count++;
68         }
69         cout << count << endl;
70     }
71 }
72 return 0;
73 }
```