

# Geometría Computacional

Pablo Blanc  
(con diapos robadas a Melanie Sclar)

Buen Kilo de Pan Flauta

Training Camp 2017

- 1 Introducción
- 2 Los elementos más básicos: puntos, vectores y rectas
  - Algunas operaciones importantes entre vectores
  - Representación de recta y segmento
- 3 Algunos problemas
- 4 Sweep Line (Técnicas de barrido)
  - Par de puntos más cercano
  - Intersección de segmentos
  - Cápsula convexa (Convex Hull)

Generalmente, los problemas de geometría de ACM requieren calcular alguna cantidad (por ej., una distancia, un área, etc) relacionada con elementos geométricos como puntos, líneas, círculos, etc. Para resolverlos, debemos poder ser capaces de:

1. Representar los objetos geométricos involucrados en el problema, para poder operar con ellos.
2. Desarrollar un algoritmo para buscar la respuesta deseada.

Entonces, para poder llegar a la segunda parte (la más interesante), primero tenemos que ser capaces de llevar a cabo la primera.

**Aclaración:** Nos referiremos siempre a problemas en  $\mathbb{R}^2$ , ya que son los más comunes.

Un punto en el plano (o un vector desde el origen hasta dicho punto) se puede representar con un par ordenado de coordenadas en un sistema cartesiano:

$$\vec{P} = (x, y) \text{ con } x, y \in \mathbb{R}$$

Se puede representar como `pair<double, double>`, pero es poco declarativo. Mejor hacer un `struct`:

```
1 | struct Punto {  
2 |     double x;  
3 |     double y;  
4 | };
```

# Contenidos

- 1 Introducción
- 2 **Los elementos más básicos: puntos, vectores y rectas**
  - **Algunas operaciones importantes entre vectores**
  - Representación de recta y segmento
- 3 Algunos problemas
- 4 Sweep Line (Técnicas de barrido)
  - Par de puntos más cercano
  - Intersección de segmentos
  - Cápsula convexa (Convex Hull)

- La suma y resta de vectores se realiza componente a componente:

$$\vec{P} = \vec{P}_1 \pm \vec{P}_2 \quad \iff \quad (x, y) = (x_1 \pm x_2, y_1 \pm y_2)$$

- La longitud o *norma* de un vector es  $|\vec{P}| = \sqrt{x^2 + y^2}$ .
- La distancia euclídea entre dos puntos  $\vec{P}_1$  y  $\vec{P}_2$  es  $|\vec{P}_1 - \vec{P}_2|$

# Código de operaciones con puntos

```
1  class punto : public pair<int, int>
2  {
3  public:
4      punto operator-(const punto &r) {
5          punto res;
6          res.f=f-r.f;
7          res.s=s-r.s;
8          return res;
9      }
10     punto operator+(const punto &r) {
11         punto res=r;
12         res.f+=f;
13         res.s+=s;
14         return res;
15     }
16 };
```

# Producto escalar entre vectores

## Producto escalar

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

$$\vec{u} \cdot \vec{v} = x_1 x_2 + y_1 y_2 = |\vec{u}| |\vec{v}| \cos \theta$$

Observar que en particular si  $\theta = 90^\circ$  el producto escalar se anula. **Es decir, dos vectores son perpendiculares si y sólo si su producto escalar da 0.**



# Producto cruz entre vectores

El producto vectorial (o producto cruz) de dos vectores es un *vector* en la dirección normal al plano generado por ellos (¡esto puede ser muy útil para problemas en  $\mathbb{R}^3$ !).

Si  $u, v \in \mathbb{R}^2$  ( $u = (x_1, y_1)$  y  $v = (x_2, y_2)$ ), se define  $u \times v$  como:

$$\vec{u} \times \vec{v} = x_1 y_2 - x_2 y_1$$

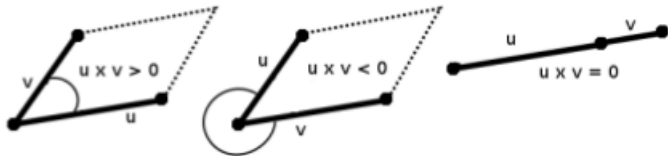
y se puede demostrar que:

$$|\vec{u} \times \vec{v}| = |\vec{u}||\vec{v}| \sin \theta,$$

y que además  $|u \times v|$  **es el área del paralelogramo determinado por los dos vectores.**

# Producto cruz entre vectores

El signo del producto indica la orientación de los vectores.



Además, notemos que aquí  $u \times v = 0$  significa que los dos vectores tienen igual dirección (¡no confundir con el producto escalar!).

# Contenidos

- 1 Introducción
- 2 **Los elementos más básicos: puntos, vectores y rectas**
  - Algunas operaciones importantes entre vectores
  - **Representación de recta y segmento**
- 3 Algunos problemas
- 4 Sweep Line (Técnicas de barrido)
  - Par de puntos más cercano
  - Intersección de segmentos
  - Cápsula convexa (Convex Hull)

# ¿Cómo represento una recta?

Una recta puede ser representada de más de una forma: las dos más usuales son:

- $L(x) = mx + b$ , siendo  $m$  la pendiente de la recta y  $b$  la ordenada al origen.
- $L(t) = \mathbf{p} + t\mathbf{v}$ , siendo  $\mathbf{v} \in \mathbb{R}^2$  el vector dirección y  $\mathbf{p}$  un punto en la recta.  $t \in \mathbb{R}$  es un escalar que representa cuánto se dilata o contrae el vector  $\mathbf{v}$ .

El primer enfoque tiene una posible desventaja, ¿cuál es?

# ¿Cómo represento una recta?

Una recta puede ser representada de más de una forma: las dos más usuales son:

- $L(x) = mx + b$ , siendo  $m$  la pendiente de la recta y  $b$  la ordenada al origen.
- $L(t) = \mathbf{p} + t\mathbf{v}$ , siendo  $\mathbf{v} \in \mathbb{R}^2$  el vector dirección y  $\mathbf{p}$  un punto en la recta.  $t \in \mathbb{R}$  es un escalar que representa cuánto se dilata o contrae el vector  $\mathbf{v}$ .

El primer enfoque tiene una posible desventaja, ¿cuál es?

La pendiente de una recta es un cociente entre dos números, y por ende debemos asegurarnos que no se indefina dicho cociente. Así, si se quiere representar la recta vertical  $x = c$ , se la deberá tratar como un caso especial en el código (¡Oh no, un *if!*).

Entonces, utilizaremos la representación  $L(t) = p + tv$ . ¿Cómo la obtenemos a partir de dos puntos que pasen por la recta  $L$ ?

Entonces, utilizaremos la representación  $L(t) = p + tv$ . ¿Cómo la obtenemos a partir de dos puntos que pasen por la recta  $L$ ?

Si  $a$  y  $b$  se encuentran sobre  $L$ , el vector  $v = b - a$  representa la dirección de la recta. Un punto  $p$  posible será el  $p = a$ . Así, la recta se escribirá como:

$$L(t) = a + t(b - a) \text{ con } t \in \mathbb{R}.$$

A partir de esto, podemos representar fácilmente un **segmento de recta** (por ejemplo, la sección de  $L$  entre  $a$  y  $b$ ). ¿Cómo?



A partir de esto, podemos representar fácilmente un **segmento de recta** (por ejemplo, la sección de  $L$  entre  $a$  y  $b$ ). ¿Cómo?

¡Aprovechándonos de  $t$ !

- Si  $t = 0$ ,  $L(0) = a$ .
- Si  $t = 1$ ,  $L(1) = b$ .
- Si  $0 < t < 1$ ,  $L(t)$  es parte del segmento determinado por  $a$  y  $b$ .
- Si  $t < 0$  o  $t > 1$ ,  $L(t)$  no será parte del segmento.

Entonces,  $L(t)$  pertenece al segmento ya dicho si y sólo si  $t \in [0, 1]$ .

# Área de un triángulo

Cómo sacaríamos el área de un triángulo.

# Área de un triángulo

Cómo sacaríamos el área de un triángulo.

Recordemos lo que vimos hace un rato: si  $u$  y  $v$  son dos vectores que forman un ángulo  $\theta$ ,  $|u \times v|$  es igual al área del paralelogramo que forman.



# Área de un triángulo

Cómo sacaríamos el área de un triángulo.

Recordemos lo que vimos hace un rato: si  $u$  y  $v$  son dos vectores que forman un ángulo  $\theta$ ,  $|u \times v|$  es igual al área del paralelogramo que forman.



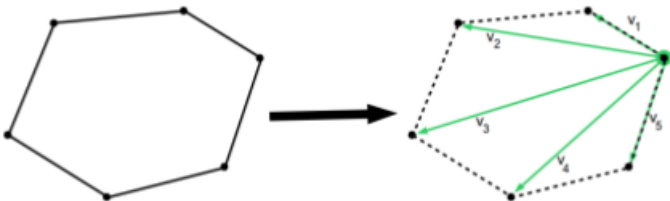
Así, el área del triángulo marcado es  $\frac{|u \times v|}{2}$ .

¿Cómo podemos aplicar lo ya visto para sacar el área de un polígono?

¿Cómo podemos aplicar lo ya visto para sacar el área de un polígono?

¡Triangulando! Veámoslo.

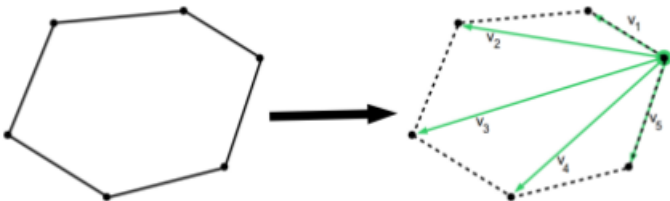
Se triangula desde un vértice cualquiera y se suman **en orden** los productos cruz. La superficie del polígono es la mitad del valor absoluto de este número.



Si el polígono es cóncavo, sigue valiendo el mismo algoritmo.

Complejidad:  $O(n)$  si  $n$  es la cantidad de vértices del polígono.

Se triangula desde un vértice cualquiera y se suman **en orden** los productos cruz. La superficie del polígono es la mitad del valor absoluto de este número.



Si el polígono es cóncavo, sigue valiendo el mismo algoritmo.

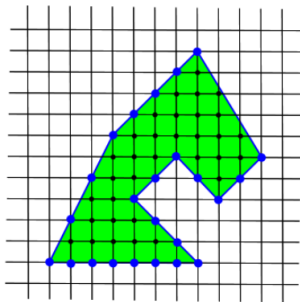
Complejidad:  $O(n)$  si  $n$  es la cantidad de vértices del polígono.



# Teorema de Pick

- $A$  es el área del polígono.
- $I$  es la cantidad de puntos interiores.
- $B$  es la cantidad de puntos del borde. Se pueden calcular usando el MCD de la longitud en  $x$  y en  $y$  de cada lado.

$$A = I + \frac{B}{2} - 1$$



# Un problema

Dado un polígono con  $n$  ( $1 \leq n \leq 10^5$ ) lados.  
¿Cuántos puntos de coordenadas enteras tiene adentro?

# Otro problema

El D del Tap del año pasado.

Dados  $n$  ( $1 \leq n \leq 1000$ ) puntos en el plano.

Determinar cuántos triángulos son semejantes al formado por los primeros 3 puntos.

# Contenidos

- 1 Introducción
- 2 Los elementos más básicos: puntos, vectores y rectas
  - Algunas operaciones importantes entre vectores
  - Representación de recta y segmento
- 3 Algunos problemas
- 4 Sweep Line (Técnicas de barrido)**
  - Par de puntos más cercano
  - Intersección de segmentos
  - Cápsula convexa (Convex Hull)

# ¿Qué es sweep line?

- La técnica de *sweep line* se caracteriza por tener una línea (vertical u horizontal generalmente, pero no necesariamente) que va barriendo todo el plano y recolectando información sobre puntos de él.
- Si se recuerdan ciertos *eventos* que van sucediendo a lo largo del barrido y se los analiza, se pueden obtener soluciones de una complejidad mejor que con otras técnicas.
- **Los eventos serán instantes del barrido en los que sucede algo relevante para el problema.**

En algún sentido, la técnica de sweep line es como la programación dinámica: uno aprende la idea general, pero luego en cada problema se aplica y se utiliza de manera diferente, adaptando la idea general ya vista.

Es por eso que creemos que la mejor manera de aprender esta técnica es viendo algunos ejemplos de problemas relevantes en los que se la utiliza.

# Contenidos

- 1 Introducción
- 2 Los elementos más básicos: puntos, vectores y rectas
  - Algunas operaciones importantes entre vectores
  - Representación de recta y segmento
- 3 Algunos problemas
- 4 Sweep Line (Técnicas de barrido)**
  - Par de puntos más cercano**
  - Intersección de segmentos
  - Cápsula convexa (Convex Hull)

## Problema

Dados  $n$  puntos en el plano, queremos encontrar dos tales que la distancia entre ellos sea la mínima posible (o sea, el par más cercano)

¿Ideas?



## Problema

Dados  $n$  puntos en el plano, queremos encontrar dos tales que la distancia entre ellos sea la mínima posible (o sea, el par más cercano)

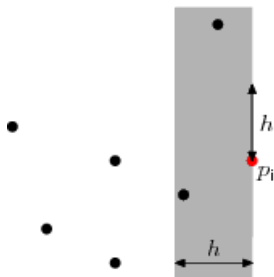
¿Ideas? ¿Cómo sé qué idea puede funcionar y cuál no, si no sé cuánto vale  $n$ ? Si  $n \leq 1000$ , el problema a resolver es muy distinto que si  $n \leq 1000000$ .

- Existe una solución trivial en  $O(n^2)$ , que es simplemente mirar cada par y elegir el de menor distancia entre todos los posibles.
- Es una solución muy fácil de implementar, así que si el problema es con  $n \leq 1000$  definitivamente deberíamos programar esa.
- ¿Pero y si  $n \leq 1000000$ ? Utilizaremos sweep line para encontrar una solución de mejor complejidad.

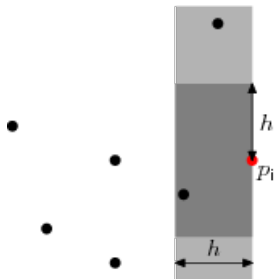
- El problema con el enfoque  $O(n^2)$  es que a cada punto lo comparamos con **todos** los demás. Tal vez podemos aprovecharnos de la posición en el plano de cada uno para no tener que hacer esto cada vez.
- Barreremos el plano con una recta vertical, de izquierda a derecha.
- Para ser coherentes con este barrido, ordenaremos los puntos crecientemente según su coordenada  $x$ . Esto tendrá una complejidad de  $O(n \lg n)$ .

- Llamemos  $h$  a la menor distancia entre dos puntos encontrada hasta el momento. La inicializaremos en  $\infty$  (en la práctica, un valor tan grande tal que sea imposible que haya un par de puntos tan lejanos).
- Supongamos que ya pasamos por primeros  $i - 1$  puntos con la línea vertical, y ahora chocamos al  $i$ -ésimo punto.

Mantendremos un set con los puntos (ya procesados) cuya coordenada  $x$  esté a una distancia menor a  $h$ . Estos puntos son los únicos candidatos a estar a distancia menor que  $h$  del punto  $i$ -ésimo, y por ende a ser una mejor solución que la actual.



También fijémonos que no cualquier punto cuya coordenada  $x$  esté a distancia menor a  $h$  es un candidato real a ser una mejor solución. Si la coordenada  $y$  de un punto está a una distancia mayor a  $h$ , tampoco será un candidato.



Así, sólo deberíamos quedarnos con los puntos cuya coordenada  $y$  esté en el intervalo  $(y_i - h, y_i + h)$ .

¿Cómo descartamos los puntos con coordenada  $x$  mayor a  $h$ , si tenemos a los candidatos en un set?

Así, sólo deberíamos quedarnos con los puntos cuya coordenada  $y$  esté en el intervalo  $(y_i - h, y_i + h)$ .

¿Cómo descartamos los puntos con coordenada  $x$  mayor a  $h$ , si tenemos a los candidatos en un set?

¡Manteniendo el set ordenado por su coordenada  $y$ !

Así, *quitar* los puntos cuya coordenada  $y$  está demasiado lejos de  $p_i$  se reduce simplemente a no mirar los puntos fuera de un intervalo.



Y ahora, para todos los puntos que quedaron en el set calculamos su distancia con  $p_i$  y actualizamos  $h$ .

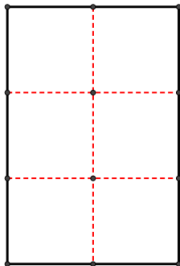
¡¿Pero cómo?! Potencialmente, hay muchos puntos en el set.

Y ahora, para todos los puntos que quedaron en el set calculamos su distancia con  $p_i$  y actualizamos  $h$ .

¡¿Pero cómo?! Potencialmente, hay muchos puntos en el set.

Esto no es tan así. Veamos que quedaron a lo sumo 6 candidatos (6 elementos en el set).

La observación importante es que todos los puntos anteriores están a una distancia mayor o igual a  $h$  entre sí, pues si no, la mínima distancia ya habría sido actualizada anteriormente.



No puede haber 2 puntos en el mismo cuadradito, pues luego esos 2 estarían a una distancia menor que  $h$ . Así, hay a lo sumo 6 puntos para mirar.

# Resumen

El algoritmo consta de 3 pasos importantes:

1. Ordenar los puntos por su coordenada  $x$  (para poder hacer el barrido con una línea vertical). Esto toma  $O(n \lg n)$ .
2. Insertar y remover cada punto una vez del set ordenado por coordenada  $y$ , que toma  $O(n \lg n)$  (insertar y remover un elemento toma  $O(\lg n)$  usando un set, y hay  $n$  puntos en total).
3. Comparar cada punto con una cantidad constante de candidatos ( $\leq 6$ ), lo que toma  $O(n)$  en total.

Así, en total el algoritmo es  $O(n \lg n)$ !

# Problema para pensar

<http://www.spoj.com/problems/CLOSEST/>

# Contenidos

- 1 Introducción
- 2 Los elementos más básicos: puntos, vectores y rectas
  - Algunas operaciones importantes entre vectores
  - Representación de recta y segmento
- 3 Algunos problemas
- 4 **Sweep Line (Técnicas de barrido)**
  - Par de puntos más cercano
  - **Intersección de segmentos**
  - Cápsula convexa (Convex Hull)

## Problema

Dados  $n$  segmentos en el plano (horizontales o verticales), queremos hallar todas las intersecciones entre dos segmentos.

Nuevamente, si entra en tiempo la solución trivial  $O(n^2)$ , hay que programar esa.

Si no, como es por ejemplo el caso  $n = 1000000$ , tenemos que pensar otra manera de resolverlo. Utilizaremos sweep line.

- De nuevo utilizaremos un barrido con una línea vertical, pero aquí la situación es distinta: no serán puntos los que crucen la línea, sino segmentos.
- Para representarlos, utilizaremos **eventos**. Los eventos serán coordenadas  $x$  en los que pasa algo relevante:
  - Por cada segmento horizontal, habrá un evento de apertura (cuando comienza el segmento) y uno de clausura (cuando pasamos por el final del mismo).
  - Por cada segmento vertical, habrá un evento (la aparición del segmento en una cierta coordenada  $x$ ).



# Pseudocódigo del algoritmo

```
1 lista = la lista de todos los eventos posibles
2 ordeno lista por las coordenadas x de sus elementos
3
4 S = set de segmentos ordenado por la coordenada y de sus elementos
5   (inicialmente vacío)
6
7 para cada event en lista
8   if ( event = horizontal de apertura )
9     inserto el segmento en el set S
10  if ( event = horizontal de clausura )
11    quito el segmento del set S
12  if ( event = vertical (segmento de y1 a y2, con y1 < y2) )
13    itero los segmentos de S en el rango (y1,y2)
14    imprimo las intersecciones halladas
```

**Detalle:** en un set se puede hallar el rango  $(y1,y2)$  en  $O(\lg n)$ , utilizando las funciones `lowerbound` y `upperbound`.

# Problema para pensar

<http://goo.gl/UT0O2U> (A Safe Bet - WF 2012)

# Contenidos

- 1 Introducción
- 2 Los elementos más básicos: puntos, vectores y rectas
  - Algunas operaciones importantes entre vectores
  - Representación de recta y segmento
- 3 Algunos problemas
- 4 Sweep Line (Técnicas de barrido)
  - Par de puntos más cercano
  - Intersección de segmentos
  - **Cápsula convexa (Convex Hull)**

## Problema

Se tiene un millón vacas en un campo. Se quieren encerrar todas las vacas con una cerca, de manera tal que se minimice la cantidad de alambre utilizado (las vacas deben quedar en el borde o dentro de la cerca).

¿Ideas?

## Problema

Se tiene un millón vacas en un campo. Se quieren encerrar todas las vacas con una cerca, de manera tal que se minimice la cantidad de alambre utilizado (las vacas deben quedar en el borde o dentro de la cerca).

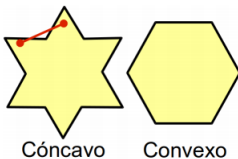
¿Ideas?

Para resolver el problema, tendremos que introducir dos conceptos antes.

# Polígonos convexos

## Polígono convexo

Un polígono es convexo si cumple que dados dos puntos cualesquiera en su interior, el segmento que los une está completamente contenido en él.

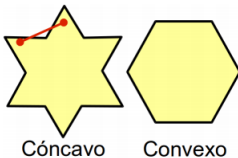


Equivalentemente, un polígono es convexo si todos sus ángulos interiores son menores que  $180^\circ$ .

# Polígonos convexos

## Polígono convexo

Un polígono es convexo si cumple que dados dos puntos cualesquiera en su interior, el segmento que los une está completamente contenido en él.



Equivalentemente, un polígono es convexo si todos sus ángulos interiores son menores que  $180^\circ$ .

Notar que el polígono que será solución al problema debe ser convexo.

## Cápsula convexa

Dados  $n$  puntos, la *cápsula convexa* es el menor polígono convexo que contiene a todos los puntos en su interior. Se puede probar que es única.





## Cápsula convexa

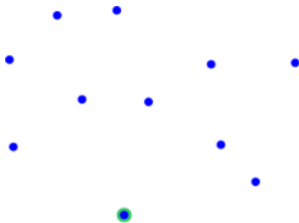
Dados  $n$  puntos, la *cápsula convexa* es el menor polígono convexo que contiene a todos los puntos en su interior. Se puede probar que es única.



Entonces, podemos ver que el problema se reduce a hallar la cápsula convexa. ¿Pero cómo lo hacemos?

# Algoritmo de Graham Scan

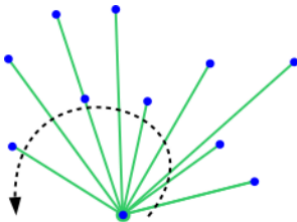
Se elige el punto de más abajo (con menor  $y$ ), y en caso de haber más de un punto con la misma  $y$  se elige el punto de más a la izquierda (o sea, con menor  $x$ ). A este punto lo llamaremos  $P$ . Elegirlo tiene una complejidad  $O(n)$ .



**Iremos barriendo el plano con una semirrecta que parta desde  $P$ .** Esta semirrecta no será vertical ni horizontal, como veníamos viendo, sino que **girará en sentido antihorario**.

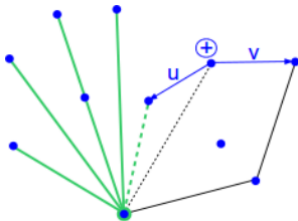
Iremos barriendo el plano con una semirrecta que parta desde  $P$ . Esta semirrecta no será vertical ni horizontal, como veníamos viendo, sino que **girará en sentido antihorario**.

Para poder hacerlo, tenemos que ordenar los puntos por su ángulo respecto a  $P$  y al eje  $x$ . Si dos puntos tienen el mismo ángulo, desempatamos por el más cercano primero.

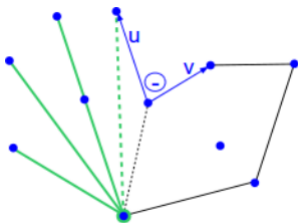


Ahora veamos cómo manejamos los eventos (en este caso, será simplemente la aparición de un punto en el barrido).

El próximo punto siempre está en la Convex Hull. En este caso,  $u \times v > 0$ .



Mientras que  $u \times v < 0$ , sacamos el punto anterior (¡forma un ángulo mayor que  $180^\circ$ !).



# Pseudocódigo del Graham Scan

```
1  vector<Punto> ConvexHull ( vector<Punto>& lista ) {
2      if( |lista| < 3 ) devolver lista
3      p = el punto de mas abajo y mas a la izquierda
4      Ordenar todos los puntos por angulo respecto a p
5          (desempatando por distancia a p, los mas cercanos primero)
6      pila = stack de puntos vacio
7      pila.push(lista[0]) // lista[0] == p
8      pila.push(lista[1])
9      i = 2
10     while ( i < N)
11         sea a el tope de pila , b el anteultimo de la pila (si existen)
12         if ( pila.size > 1 && pcruz(lista[i] - a , b - a) <= 0 )
13             pila.pop()
14         else
15             pila.push(lista[i])
16             i = i + 1
17     devolver pila
18 }
```

# Problemas para pensar

- [goo.gl/rT7Ji](https://goo.gl/rT7Ji)
- [goo.gl/IIEHC](https://goo.gl/IIEHC)



# Referencias

- *Introduction to Algorithms, 2nd Edition*. MIT Press.  
Thomas H. Cormen **Sección 33** (Computational Geometry)
- <https://www.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry1>
- <https://www.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry2>
- <https://www.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry3>
- <https://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lineSweep>