

# Índice

1. Notas previas - IMPORTANTE LEER	1
2. Puntos	1
2.1. Distancia entre dos puntos . . . . .	1
2.2. Rotar un punto . . . . .	2
3. Vectores	2
3.1. Magnitud de un vector . . . . .	2
3.2. Dirección de un vector . . . . .	2
3.3. Vectores paralelos y perpendiculares . . . . .	2
3.4. Ángulo entre vectores . . . . .	2
4. Líneas	2
4.1. Distancias . . . . .	2
4.1.1. Distancia de un punto a una recta . . . . .	2
4.1.2. Distancia de un punto a un segmento . . . . .	2
4.1.3. Distancia entre dos segmentos . . . . .	3
4.2. Verificar si dos rectas son paralelas o perpendiculares . . . . .	3
4.3. Intersección de dos rectas . . . . .	3
4.4. Verificar si tres puntos son colineales . . . . .	4
5. Círculos	4
5.1. Área y perímetro . . . . .	4
5.2. Determinar si un punto está dentro o fuera de un círculo . . . . .	4
5.3. Arco, sector, cuerda y segmento . . . . .	4
5.4. Círculo formado por 3 puntos . . . . .	4
5.5. Recta tangente a un círculo . . . . .	5
6. Triángulos	5
6.1. Desigualdad triangular . . . . .	5
6.2. Área de un triángulo dados sus lados . . . . .	5
6.3. Área de un triángulo dados sus puntos . . . . .	5
6.4. Ley del seno y del coseno . . . . .	5
6.5. Círculo inscrito y circunscrito . . . . .	6
7. Polígonos	6
7.1. Tipos de polígonos . . . . .	6
7.2. Perímetro . . . . .	6
7.3. Área . . . . .	6
7.4. Centroide . . . . .	6
7.5. Verificar si un polígono es convexo . . . . .	7
7.6. Verificar si un punto está dentro de un polígono . . . . .	7
7.7. Convex Hull . . . . .	8

## 1. Notas previas - IMPORTANTE LEER

- Se usarán algunas clases estándar de Java para simplificar las cosas
- No se presenta el código implementado de todas las fórmulas, ya que muchas pueden ser programadas fácilmente
- $\pi$  está definido en Java en la constante `Math.PI`
- Se trabajará siempre con los ángulos en **radianes**. Para convertir un ángulo  $\alpha$  en grados, en uno en radianes  $\theta$ :  $\theta = (\alpha \cdot \pi)/180$
- El enunciado de los problemas generalmente establece un margen de error. Ese es el  $\epsilon$  que se usará para las comparaciones entre números de punto flotante. Usualmente lo defino como variable estática de clase (por ejemplo, `static double eps = 1e-6;`).

## 2. Puntos

Para representar un punto se usa la clase de Java `Point2D.Double`

```
1 Point2D.Double o = new Point2D.Double(); // El punto
   ↪ (0, 0)
2 Point2D.Double p = new Point2D.Double(0.5, 2.75);
```

### 2.1. Distancia entre dos puntos

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Se puede usar un método de Java:

```
1 double d = o.distance(p);
```

También hay un método que devuelve el cuadrado de la distancia, para evitar sacar la raíz cuadrada y perder precisión:

```
1 double dSquare = o.distanceSq(p);
```

## 2.2. Rotar un punto

Se rota un punto  $(x, y)$  por un ángulo  $\theta$  en sentido **antihorario**.

```
1 public static Point2D.Double rotar(Point2D.Double p,  
    ↪ double theta) {  
2     double x = p.getX() * Math.cos(theta) - p.getY() * Math  
    ↪ .sin(theta);  
3     double y = p.getX() * Math.sin(theta) + p.getY() * Math  
    ↪ .cos(theta);  
4     return new Point2D.Double(x, y);  
5 }
```

## 3. Vectores

Sean dos puntos  $p_1 = (x_1, y_1)$  y  $p_2 = (x_2, y_2)$ . El vector que va desde  $p_1$  hacia  $p_2$  se define como:

$$V = \langle v_1, v_2 \rangle = \langle x_2 - x_1, y_2 - y_1 \rangle$$

Sugiero representarlos con un array de tipo *double* (*double[] v*).

### 3.1. Magnitud de un vector

$$|V| = \sqrt{v_1^2 + v_2^2}$$

### 3.2. Dirección de un vector

El ángulo entre un vector  $V = \langle v_1, v_2 \rangle$  y los ejes  $x$  y  $y$  es:

$$\cos \theta_x = \frac{v_1}{|V|}, \quad \cos \theta_y = \frac{v_2}{|V|}$$

En Java la función para sacar el coseno inverso (y poder despejar  $\theta$  en estas fórmulas) es *Math.acos*

### 3.3. Vectores paralelos y perpendiculares

```
1 public static int vectoresParalelosPerpend(double[] a,  
    ↪ double[] b) {  
2     if (Math.abs(a[1] * b[0] - a[0] * b[1]) < eps) {  
3         return 1; // Son paralelos  
4     } else if (Math.abs(a[0] * b[0] + a[1] * b[1]) < eps) {  
5         return 2; // Son perpendiculares  
6     } else {
```

```
7         return 0; // Ni paralelos ni perpendiculares  
8     }  
9 }
```

## 3.4. Ángulo entre vectores

El menor ángulo entre dos vectores  $A = \langle a_1, a_2 \rangle$  y  $B = \langle b_1, b_2 \rangle$  es:

$$\cos \theta = \frac{a_1 b_1 + a_2 b_2}{|A||B|}$$

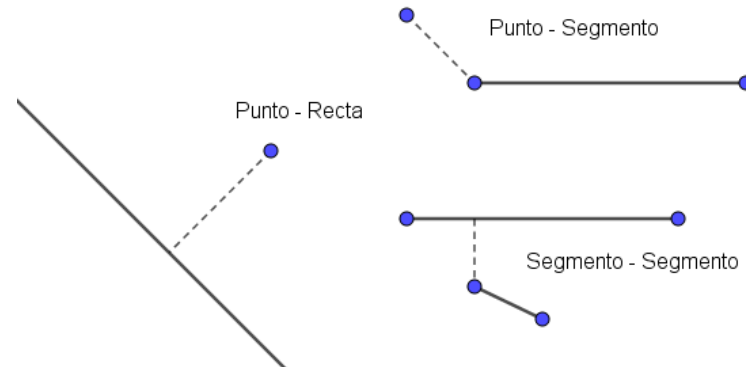
En Java la función para sacar el coseno inverso (y poder despejar  $\theta$  en esta fórmula) es *Math.acos*

## 4. Líneas

Para representar una recta definida por dos puntos se usa la clase de Java *Line2D.Double*

```
1 Point2D.Double p1 = new Point2D.Double(1, 0);  
2 Point2D.Double p2 = new Point2D.Double(3, 5);  
3 Line2D.Double linea = new Line2D.Double(p1, p2);
```

### 4.1. Distancias



#### 4.1.1. Distancia de un punto a una recta

```
1 Point2D.Double p3 = new Point2D.Double(-2, -1);  
2 double dLinea = linea.ptLineDist(p3);
```

#### 4.1.2. Distancia de un punto a un segmento

```

1 // Podemos representar un segmento con la misma clase
   ↪ Line2D.Double
2 Line2D.Double segmento = new Line2D.Double(p1, p2);
3 Point2D.Double p4 = new Point2D.Double(-2, -1);
4 double dSegmento = segmento.ptSegDist(p4);

```

#### 4.1.3. Distancia entre dos segmentos

Sean dos segmentos  $s_1, s_2$ . La distancia más corta entre ellos es:

```

1 public static double distanciaSegmentos(Line2D.Double s1,
   ↪ Line2D.Double s2) {
2     double[] d = new double[4];
3     d[0] = s2.ptSegDist(s1.getP1());
4     d[1] = s2.ptSegDist(s1.getP2());
5     d[2] = s1.ptSegDist(s2.getP1());
6     d[3] = s1.ptSegDist(s2.getP2());
7     double min = Double.MAX_VALUE;
8     for (int i = 0; i < 4; i++) {
9         if (d[i] < min) {
10             min = d[i];
11         }
12     }
13     return min;
14 }

```

#### 4.2. Verificar si dos rectas son paralelas o perpendiculares

Se puede hacer utilizando vectores, pero con este método no se puede especificar el punto de intersección, o determinar si las dos rectas son la misma (en caso de requerir esto, ver la siguiente sección, 4.3).

Usa la función de vectores paralelos y perpendiculares (ver sección 3.3).

```

1 public static int lineasParalelasPerpend(Line2D.Double l1
   ↪ , Line2D.Double l2) {
2     double[] v1 = { l1.getX2() - l1.getX1(), l1.getY2() -
   ↪ l1.getY1() };
3     double[] v2 = { l2.getX2() - l2.getX1(), l2.getY2() -
   ↪ l2.getY1() };
4     return vectoresParalelosPerpend(v1, v2);
5 }

```

#### 4.3. Intersección de dos rectas

Si las dos rectas no son paralelas, se intersectan en un punto. Se halla la ecuación general de ambas rectas y se soluciona un sistema lineal de ecuaciones para

hallar el punto de intersección.

La ecuación general de una recta tiene la forma  $Ax + By + C = 0$ .

```

1 public static double[] ecuacionGeneral(Line2D.Double l) {
2     double[] coef = new double[3];
3     Point2D p1 = l.getP1();
4     Point2D p2 = l.getP2();
5     if (Math.abs(p1.getX() - p2.getX()) < eps) {
6         coef[0] = 1.0;
7         coef[1] = 0.0;
8         coef[2] = -p1.getX();
9     } else {
10        coef[0] = -(p1.getY() - p2.getY()) / (p1.getX() - p2.
   ↪ getX());
11        coef[1] = 1.0;
12        coef[2] = -coef[0] * p1.getX() - p1.getY();
13    }
14    return coef;
15 }
16
17 public static double[] interseccion(Line2D.Double l1,
   ↪ Line2D.Double l2) {
18     double[] e1 = ecuacionGeneral(l1);
19     double[] e2 = ecuacionGeneral(l2);
20     if (Math.abs(e1[0] - e2[0]) < eps && Math.abs(e1[1] -
   ↪ e2[1]) < eps) {
21         if (Math.abs(e1[2] - e2[2]) < eps) {
22             return null; // Las lineas son la misma
23         }
24         double[] p = { Double.NaN, Double.NaN }; // Las
   ↪ lineas son paralelas
25         return p;
26     } else {
27         // Las lineas se intersectan
28         double x = (e2[1] * e1[2] - e1[1] * e2[2]) / (e2[0] *
   ↪ e1[1] - e1[0] * e2[1]);
29         double y = Math.abs(e1[1]) > eps ? -(e1[0] * x + e1
   ↪ [2]) : (e2[0] * x + e2[2]);
30         double[] p = { x, y };
31         return p;
32     }
33 }

```

#### 4.4. Verificar si tres puntos son colineales

Se puede usar, por ejemplo, para verificar si el punto  $p_3$  pertenece a la recta definida por los puntos  $p_1$  y  $p_2$ .

```
1 public static boolean colineales(Point2D.Double p1,
  ↪ Point2D.Double p2, Point2D.Double p3) {
2     double det = p1.x * (p2.y - p3.y) + p2.x * (p3.y - p1.y
  ↪ ) + p3.x * (p1.y - p2.y);
3
4     if (Math.abs(det) < eps) {
5         return true;
6     } else {
7         return false;
8     }
9 }
```

### 5. Círculos

Un círculo se define con un centro  $(x_0, y_0)$  y un radio  $r$ .

#### 5.1. Área y perímetro

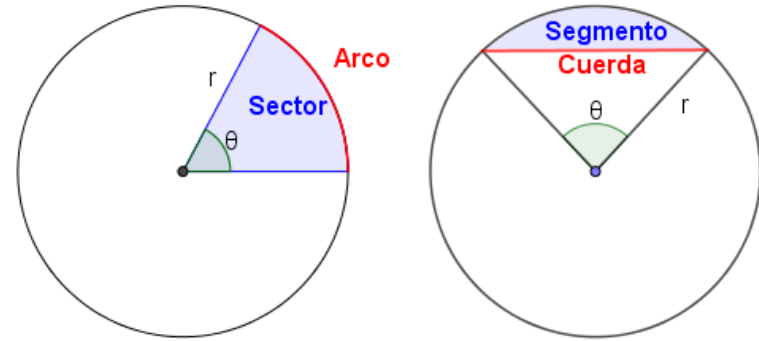
Área:  $\pi r^2$

Perímetro:  $2\pi r$

#### 5.2. Determinar si un punto está dentro o fuera de un círculo

```
1 public static int puntoEnCirculo(Point2D.Double c, double
  ↪ r, Point2D.Double p) {
2     double r2 = r * r;
3     double dist = c.distanceSq(p);
4     if (Math.abs(dist - r2) < eps) {
5         return 0; // En el borde
6     } else if (dist > r2) {
7         return 1; // Afuera
8     } else {
9         return 2; // Adentro
10    }
11 }
```

#### 5.3. Arco, sector, cuerda y segmento



Longitud del Arco:  $r\theta$

Área del Sector:  $\frac{r^2\theta}{2}$

Longitud de la Cuerda:  $2r \cdot \sin(\frac{\theta}{2})$

Área del Segmento:  $\frac{r^2}{2} \cdot (\theta - \sin \theta)$

#### 5.4. Círculo formado por 3 puntos

Sean tres puntos  $p_1$ ,  $p_2$  y  $p_3$ . Si no son colineales, existe un círculo que pasa por los tres. Se hallan los coeficientes de la ecuación general de la circunferencia  $x^2 + y^2 + Ax + By + C = 0$ , de los cuales se puede extraer el radio y el centro de la misma.

```
1 public static double[] circuloPuntos(Point2D.Double p1,
  ↪ Point2D.Double p2, Point2D.Double p3) {
2     double x1 = p1.getX(), y1 = p1.getY();
3     double x2 = p2.getX(), y2 = p2.getY();
4     double x3 = p3.getX(), y3 = p3.getY();
5
6     double det = x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1
  ↪ - y2);
7     if (Math.abs(det) < eps) {
8         // Los puntos son colineales, no existe circulo que
  ↪ pase por los 3
9         return null;
10    } else {
11        double k1 = -(x1 * x1) - (y1 * y1);
12        double k2 = -(x2 * x2) - (y2 * y2);
13        double k3 = -(x3 * x3) - (y3 * y3);
14
15        double d1 = k1 * (y2 - y3) + k2 * (y3 - y1) + k3 * (
  ↪ y1 - y2);
```

```

16  double d2 = x1 * (k2 - k3) + x2 * (k3 - k1) + x3 * (
    ↪ k1 - k2);
17  double d3 = x1 * (y2 * k3 - y3 * k2) + x2 * (y3 * k1
    ↪ - y1 * k3) + x3 * (y1 * k2 - y2 * k1);
18
19  double A = d1 / det;
20  double B = d2 / det;
21  double C = d3 / det;
22
23  // Los parametros del circulo
24  double x0 = -A / 2;
25  double y0 = -B / 2;
26  double r = 0.5 * Math.sqrt(A * A + B * B - 4 * C * C)
    ↪ ;
27
28  double[] params = { x0, y0, r };
29  return params;
30 }
31 }

```

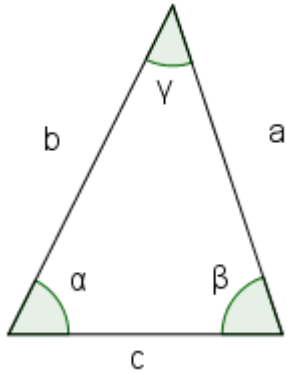
### 5.5. Recta tangente a un círculo

Es posible hallar la ecuación general de la recta tangente en un punto  $(x_1, y_1)$  a una circunferencia con centro  $(x_0, y_0)$  y radio  $r$ .

Sean  $w = x_1 - x_0$  y  $t = y_1 - y_0$ . La ecuación general de la recta tangente es:

$$wx + ty + (-wx_0 - ty_0 - r^2) = 0$$

## 6. Triángulos



### 6.1. Desigualdad triangular

Todo triángulo cumple las siguientes desigualdades:

$$a + b > c \quad a + c > b \quad b + c > a$$

### 6.2. Área de un triángulo dados sus lados

Sea el  $s$  el semiperímetro del triángulo:

$$s = \frac{a + b + c}{2}$$

El área del triángulo es (fórmula de Herón):

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

### 6.3. Área de un triángulo dados sus puntos

Si se tienen las coordenadas  $(x_i, y_i)$  de los tres puntos que definen un triángulo, su área es:

$$A = \frac{1}{2} \cdot |x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|$$

### 6.4. Ley del seno y del coseno

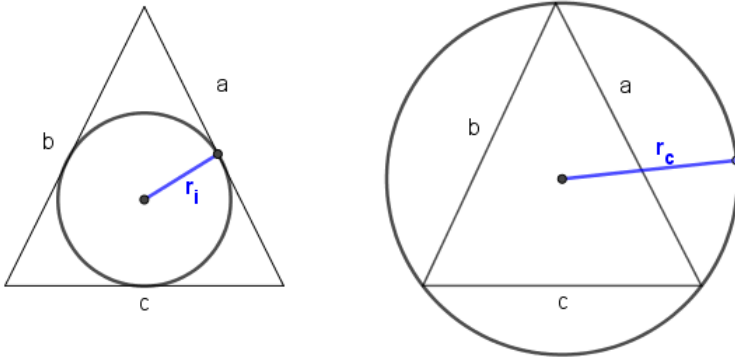
$$\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)}$$

$$a^2 = b^2 + c^2 - 2bc \cdot \cos(\alpha)$$

$$b^2 = a^2 + c^2 - 2ac \cdot \cos(\beta)$$

$$c^2 = a^2 + b^2 - 2ab \cdot \cos(\gamma)$$

## 6.5. Círculo inscrito y circunscrito



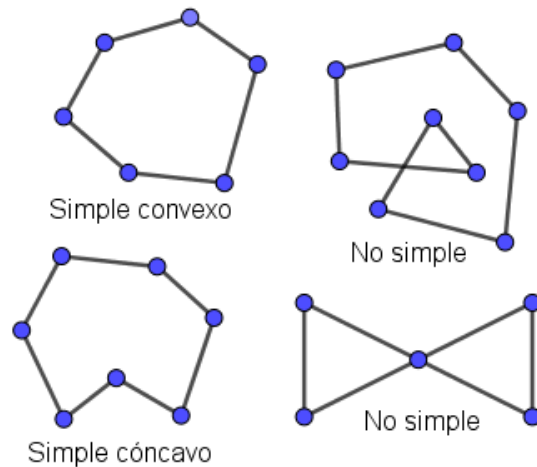
$$r_i = \frac{A}{s}, \quad r_c = \frac{abc}{4A}$$

## 7. Polígonos

Un polígono de  $n$  puntos se representa como un vector de  $n$  posiciones de tipo *Point2D.Double*.

Se asume por defecto que los puntos **están ordenados en sentido antihorario**. Si el polígono no está ordenado, estos algoritmos no sirven. Habría que ordenarlo primero.

### 7.1. Tipos de polígonos



### 7.2. Perímetro

```
1 public static double perimetro(Point2D.Double[] poligono)
2     ↪ {
3     int n = poligono.length;
4     double p = 0.0;
5     for (int i = 0; i < n; i++) {
6         int j = (i + 1) % n;
7         p += poligono[i].distance(poligono[j]);
8     }
9     return p;
}
```

### 7.3. Área

El algoritmo sólo sirve para polígonos simples (que no se intersectan a sí mismos). Si el polígono está ordenado en sentido horario, retorna el área negativa.

```
1 public static double area(Point2D.Double[] poligono) {
2     int n = poligono.length;
3     double a = 0.0;
4     for (int i = 0; i < n; i++) {
5         int j = (i + 1) % n;
6         a += poligono[i].x * poligono[j].y - poligono[j].x *
7             ↪ poligono[i].y;
8     }
9     return a / 2.0;
}
```

### 7.4. Centroide

Es el “centro de masa” de un polígono, la posición media de todos los puntos del mismo. Si se dibujara el polígono en papel y se recortara, se podría balancear en un palillo ubicado en el centroide.

El algoritmo sólo sirve para polígonos simples.

```
1 public static Point2D.Double centroide(Point2D.Double[]
2     ↪ poligono) {
3     int n = poligono.length;
4     double a = area(poligono);
5     double cx = 0.0, cy = 0.0;
6
7     for (int i = 0; i < n; i++) {
8         int j = (i + 1) % n;
9     }
}
```

```

8      cx += (poligono[i].x + poligono[j].x) * (poligono[i].
      ↪ x * poligono[j].y - poligono[j].x * poligono[i
      ↪ ].y);
9  }
10
11  for (int i = 0; i < n; i++) {
12      int j = (i + 1) % n;
13      cy += (poligono[i].y + poligono[j].y) * (poligono[i].
      ↪ x * poligono[j].y - poligono[j].x * poligono[i
      ↪ ].y);
14  }
15
16  cx = cx / (6 * a);
17  cy = cy / (6 * a);
18  return new Point2D.Double(cx, cy);
19  }

```

## 7.5. Verificar si un polígono es convexo

```

1  // Retorna positivo si p1,p2,p3 hacen un giro a la
      ↪ izquierda
2  // Retorna negativo si p1,p2,p3 hacen un giro a la
      ↪ derecha
3  // Retorna 0 si p1,p2,p3 son colineales
4  public static int giro(Point2D.Double p1, Point2D.Double
      ↪ p2, Point2D.Double p3) {
5      double det = p1.x * (p2.y - p3.y) + p2.x * (p3.y - p1.y
      ↪ ) + p3.x * (p1.y - p2.y);
6      if (Math.abs(det) < eps) {
7          return 0;
8      } else if (det > 0) {
9          return 1;
10     } else {
11         return -1;
12     }
13 }
14
15 public static boolean esConvexo(Point2D.Double[] poligono
      ↪ ) {
16     int n = poligono.length;
17     boolean inicial = giro(poligono[0], poligono[1],
      ↪ poligono[2]) >= 0;
18     for (int i = 1; i < n; i++) {

```

```

19     int j = (i + 1) % n;
20     int k = (i + 2) % n;
21     boolean actual = giro(poligono[i], poligono[j],
      ↪ poligono[k]) >= 0;
22     if (actual != inicial)
23         return false;
24     }
25     return true;
26 }

```

## 7.6. Verificar si un punto está dentro de un polígono

Funciona para polígonos cóncavos y convexos.

```

1  public static double angulo(Point2D.Double p1, Point2D.
      ↪ Double p2, Point2D.Double p3) {
2      double[] v1 = { p1.x - p2.x, p1.y - p2.y };
3      double[] v2 = { p3.x - p2.x, p3.y - p2.y };
4      double m1 = Math.sqrt(v1[0] * v1[0] + v1[1] * v1[1]);
5      double m2 = Math.sqrt(v2[0] * v2[0] + v2[1] * v2[1]);
6      double theta = (v1[0] * v2[0] + v1[1] * v2[1]) / (m1 *
      ↪ m2);
7      return Math.acos(theta);
8  }
9
10 // Usa tambien el metodo 'giro' definido anteriormente
11 public static boolean puntoEnPoligono(Point2D.Double[]
      ↪ poligono, Point2D.Double p) {
12     int n = poligono.length;
13     double anguloTotal = 0.0;
14     for (int i = 0; i < n; i++) {
15         int j = (i + 1) % n;
16         if (giro(poligono[i], p, poligono[j]) == 1) {
17             anguloTotal += angulo(poligono[i], p, poligono[j]);
18         } else {
19             anguloTotal -= angulo(poligono[i], p, poligono[j]);
20         }
21     }
22     if (Math.abs(Math.abs(anguloTotal) - 2 * Math.PI) < eps
      ↪ ) {
23         return true;
24     } else {
25         return false;
26     }
27 }

```

## 7.7. Convex Hull

Algoritmo *Andrew's Monotone Chain*. Toma un conjunto de  $n$  puntos y halla su envolvente convexa, que es el polígono convexo más pequeño que los encierra a todos. La envolvente convexa es un subconjunto de los  $n$  puntos.

El algoritmo tiene una complejidad de  $O(n \log n)$ .

```
1 // Usa el metodo 'giro' definido anteriormente
2 public static Point2D.Double[] convexHull(Point2D.Double
   ↪ [] puntos) {
3     int n = puntos.length;
4     int k = 0;
5     Point2D.Double[] hull = new Point2D.Double[2 * n];
6
7     Arrays.sort(puntos, new Comparator<Point2D.Double>() {
8         @Override
9         public int compare(Point2D.Double p1, Point2D.Double
   ↪ p2) {
10         if (Math.abs(p1.x - p2.x) < eps) {
11             if (Math.abs(p1.y - p2.y) < eps)
12                 return 0;
13             else if (p1.y < p2.y)
14                 return -1;
15             else
16                 return 1;
17         } else {
18             if (p1.x < p2.x)
19                 return -1;
20             else
21                 return 1;
22         }
23     }
```

```
24     });
25
26     // Hull inferior
27     for (int i = 0; i < n; i++) {
28         while (k >= 2 && giro(hull[k - 2], hull[k - 1],
   ↪ puntos[i]) <= 0) {
29             k--;
30         }
31         hull[k] = puntos[i];
32         k++;
33     }
34
35     // Hull superior
36     int t = k + 1;
37     for (int i = n - 2; i >= 0; i--) {
38         while (k >= t && giro(hull[k - 2], hull[k - 1],
   ↪ puntos[i]) <= 0) {
39             k--;
40         }
41         hull[k] = puntos[i];
42         k++;
43     }
44     if (k > 1) {
45         hull = Arrays.copyOfRange(hull, 0, k - 1); // Quitar
   ↪ vertices que no pertenecen al hull despues de k
46     }
47
48     return hull;
49 }
```