IIP (E.T.S. de Ingeniería Informática)
Curso 2019-2020

Práctica 5. Selección: método `cross` *de la clase* `Point`

Profesores de IIP
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Índice

1. Objetivos y trabajo previo a la sesión de prácticas

El objetivo principal de esta práctica es trabajar con la sintaxis y la semántica de las instrucciones condicionales Java estudiadas en el tema 5 de teoría (*"Estructuras de control: selección"*). En concreto, se propone completar un método de una clase *"Tipo de Datos"* que define un punto en el plano cartesiano.

2. Descripción del problema

Un problema muy común en aplicaciones gráficas es el de comprobar si un punto es interior a un polígono. Un algoritmo conocido es el del *rayo* que, básicamente, consiste en avanzar desde el punto en una dirección fija, por ejemplo, paralela al eje X en sentido positivo, y contar el número de veces que se cruza algún lado del polígono. Si dicho número es par, el punto es exterior y si es impar, el punto es interior. Intuitivamente, si estamos dentro de un cercado de cualquier forma y avanzamos en una dirección fija, cuando hemos saltado la cerca un número impar de veces estamos fuera del cercado, y si saltamos la cerca un número par de veces seguiremos dentro (viceversa si partiéramos de un punto exterior del cercado). Formalmente, se trata de una aplicación del *teorema de Jordan*[1,2].

En esta práctica se propone implementar primero un método que, dado un punto `p`, compruebe si un rayo iniciado en `p` cruza un segmento de recta delimitado por los puntos `u` y `v`.

[1] `http://erich.realtimerendering.com/ptinpoly/`
[2] `https://wrf.ecse.rpi.edu//Research/Short_Notes/pnpoly.html`

Si se tiene un polígono dado por una secuencia de vértices, el método anterior permite comprobar si `p` es interior al polígono por el método del rayo, como en el ejemplo de la Figura 1. Básicamente, se debe contar el número de lados del polígono atravesados por el rayo y comprobar si es par o impar. Hay que tener cuidado si el rayo cruza el polígono por un vértice, dado que el vértice pertenece a dos lados diferentes, lo que se debe tener en cuenta a la hora de contar el número de cruzamientos. El algoritmo para contar el número de cruzamientos se implementará en la práctica 7.

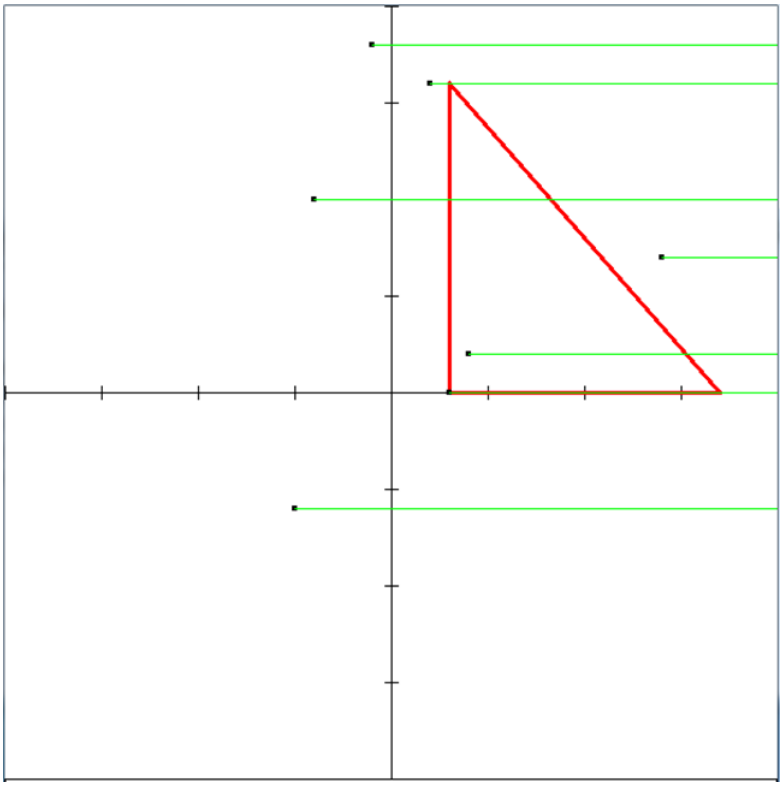


Figura 1: Triángulo y diferentes rayos que cruzan sus lados.

3. Diseño de las clases de la aplicación

Para la resolución del problema planteado, se completará la implementación de las siguientes clases:

- La clase *"Tipo de Datos"* `Point`, que representa un punto en el plano cartesiano mediante los siguientes atributos y métodos:
  - Atributos públicos: las constantes Java –variables finales (`final`) de clase (`static`)– de tipo `int` que definen el cruzamiento: `DONT_CROSS`, `LOW_CROSS`, `CROSS`, `HIGH_CROSS`, con valores `-1`, `0`, `1` y `2`, respectivamente.
  - Atributos privados: las variables de instancia `x` e `y` de tipo `double` que definen la abscisa y la ordenada del punto, respectivamente.
  - Métodos públicos: constructor, *getters*, *setters*, `toString`, `equals` y `cross`.

    El método `cross`, dados dos objetos de tipo `Point`, `u` y `v`, que representan los puntos extremos de un segmento de recta $\overrightarrow{uv}$, comprueba si el rayo, que se inicia en `this` y avanza paralelo al eje X en sentido positivo, cruza el segmento de recta $\overrightarrow{uv}$, es decir, pasa por un único punto del segmento. En la sección 4 se describe con detalle el análisis de casos necesario para su implementación.
- La clase *"Programa"* `RayTest`, en cuyo método `main` se prueba el método `cross` de la clase `Point` y se muestra el resultado que devuelve en la salida estándar y en la salida gráfica.

4. Diseño del método `cross`

El perfil del método `cross` es el siguiente:

```
/** Dado el rayo que se inicia en this y avanza paralelo al eje X
 *  en sentido +, comprueba si dicho rayo cruza el segmento de
 *  recta uv, es decir, pasa por un único punto del segmento.
 *  @param u Point, punto extremo del segmento de recta uv.
 *  @param v Point, punto extremo del segmento de recta uv.
 *  @return int, entero entre DONT_CROSS (-1), LOW_CROSS (0),
 *  CROSS (1), HIGH_CROSS (2), según los casos:
 *  - Si el rayo no cruza el segmento, devuelve DONT_CROSS.
 *  - Si el rayo lo cruza por el extremo más bajo, devuelve LOW_CROSS.
 *  - Si el rayo lo cruza por un punto entre u y v, devuelve CROSS.
 *  - Si el rayo lo cruza por el extremo más alto, devuelve HIGH_CROSS.
 */
public int cross(Point u, Point v)
```

Este método se puede abordar calculando por dónde atraviesa el rayo la recta definida por `u` y por `v`, teniendo en cuenta la pendiente de dicha recta. Supóngase, en primer lugar, que se ha seleccionado en una variable `pHigh` el punto de entre `u` y `v` más alto (el de mayor ordenada), y en una variable `pLow` el más bajo (el de menor ordenada):

- Si el segmento que pasa por ambos puntos es paralelo al eje X, es decir, `pHigh.y == pLow.y`, entonces ningún rayo cruza el segmento, pues no pasa por ningún punto, o pasa por todos (**Caso 1**).
- En caso contrario, lo más sencillo es comprobar primero si pasa por uno u otro punto extremo, diferenciando si pasa por `pHigh` (**Caso 2**) o por `pLow` (**Caso 3**), como se ve en los ejemplos de la Figura 2(a). Si no pasa por ninguno, se puede calcular el punto de corte del rayo con la recta que contiene el segmento. Sea dicha recta $ax + b$, y (`xCut`, `yCut`) el punto de corte que se desea calcular. Teniendo en cuenta que

$$\texttt{yCut} = \texttt{this.y},$$

$$a = \frac{\texttt{pHigh.y} - \texttt{pLow.y}}{\texttt{pHigh.x} - \texttt{pLow.x}},$$

$$b = \texttt{pLow.y} - a \cdot \texttt{pLow.x},$$

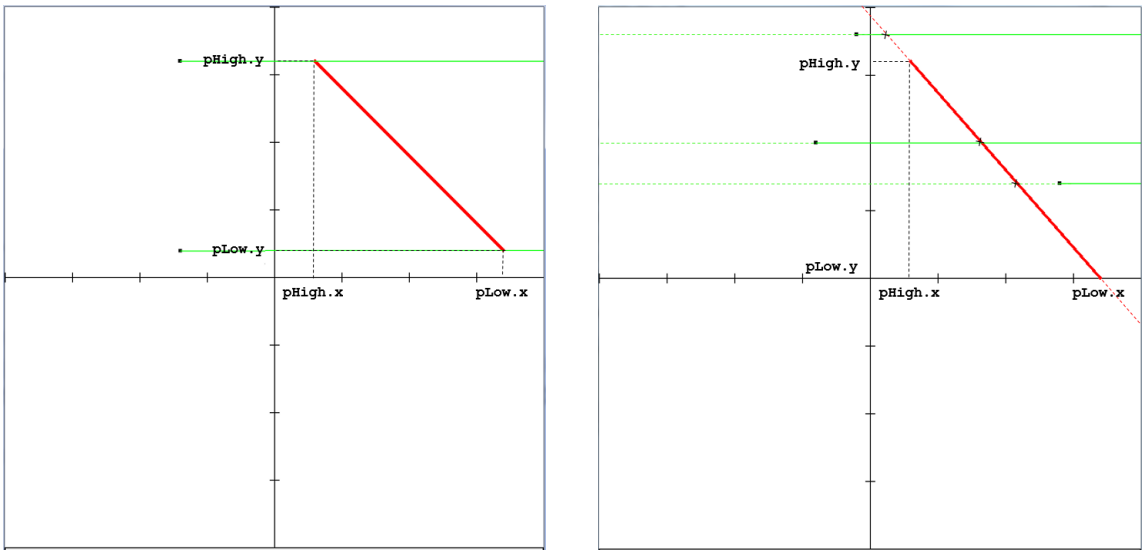
$$\texttt{xCut} = \frac{\texttt{yCut} - b}{a}$$

  entonces, la abscisa del punto de corte se calcula como:

```
double xCut = (this.y - pLow.y) * (pHigh.x - pLow.x)
          / (pHigh.y - pLow.y) + pLow.x;
```

  Hecho este cálculo, el rayo cruza el segmento si `yCut` se encuentra entre la ordenada más alta y la más baja del segmento, y `xCut` se encuentra a la derecha de `this.x` (**Caso 4**), si no el rayo no cruza el segmento (**Caso 5**), como se ve en los ejemplos de la Figura 2(b).

Todo ello se traduce en el análisis de casos de la Figura 3.



(a) Casos 2 y 3.

(b) Casos 4 y 5.

Figura 2: Cruces de un segmento por diferentes rayos.

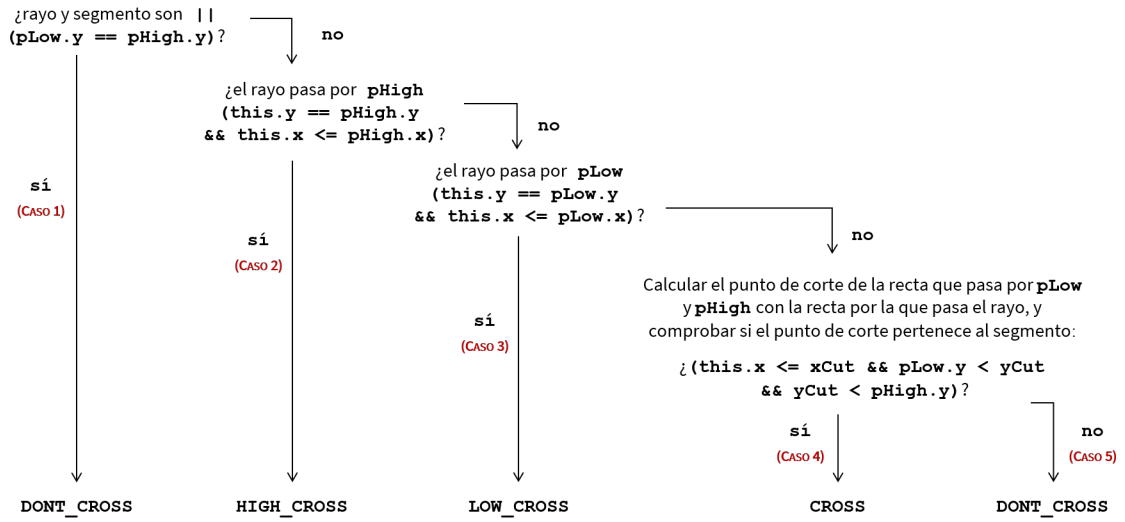


Figura 3: Análisis de casos del método `cross`.

5. Actividades de laboratorio

Actividad 1: crear el paquete BlueJ `pract5`

1. Descarga en el directorio `Downloads` los ficheros `Point.java` y `RayTest.java`, disponibles en la carpeta de material para la práctica 5 de *PoliformaT*.

2. Abre el proyecto *BlueJ* de trabajo de la asignatura (`iip`).

3. Crea un nuevo paquete (*Edición - Nuevo Paquete*) de nombre `pract5` y ábrelo (doble clic).

4. Agrega al paquete `pract5` las clases `Point` y `RayTest` (*Edición - Agregar Clase desde Archivo*). Comprueba que sus primeras líneas incluyen la directiva `package pract5;`, que indica que son clases del paquete.

Actividad 2: instalación de la librería gráfica `Graph2D`

Para poder mostrar gráficamente los segmentos y los cruzamientos de los rayos, se te proporciona una librería gráfica que permite representar gráficamente puntos y líneas, entre otros elementos, en un espacio bidimensional. Se trata de una librería desarrollada a propósito, en el ámbito de las asignaturas IIP y PRG, para facilitar a los alumnos de primer curso la obtención de resultados gráficos de forma sencilla.

La librería gráfica (clase `Graph2D` del paquete `graph2D`) se facilita como una librería en el fichero `graphLib.jar` (disponible en la carpeta *IIP:recursos/Laboratorio/Librería gráfica* de *PoliformaT*). Debes cargar esta librería como sigue:

1. Sitúa el fichero (`graphLib.jar`) en el proyecto de prácticas (`iip`).
2. En *Preferencias-Librerías* de *BlueJ*, añade el fichero `graphLib.jar`. Tendrás que arrancar nuevamente *BlueJ* para que la librería se cargue, como puedes ver en la Figura 4.

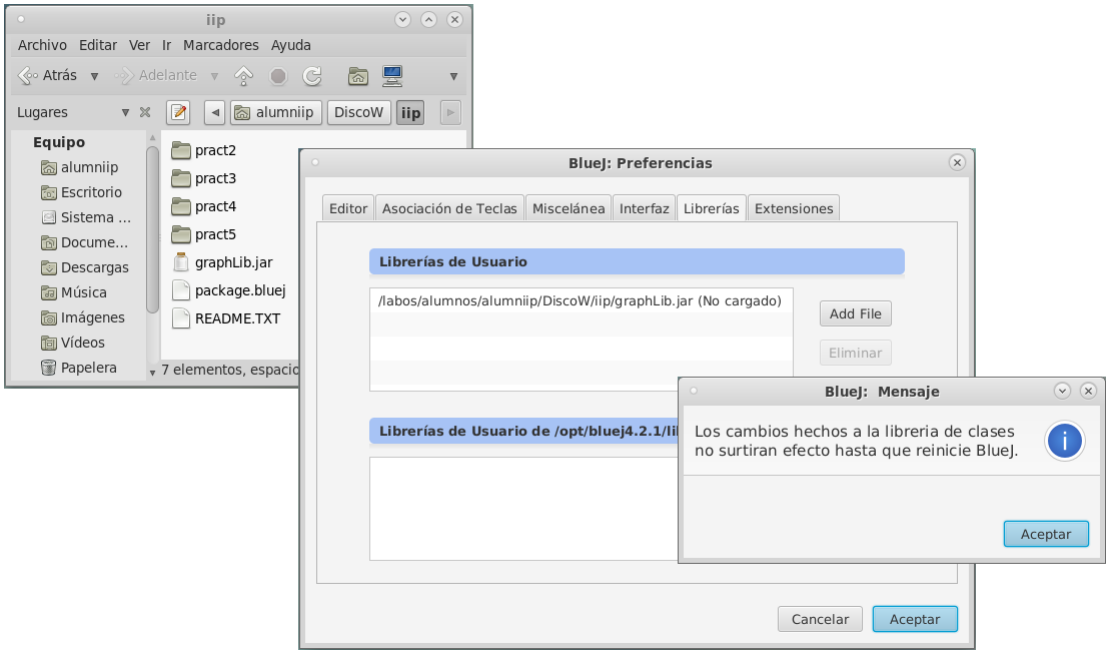


Figura 4: Instalación de la librería gráfica en *BlueJ*.

La documentación de la clase `Graph2D` se te proporciona en el fichero `docGraph2D.zip`. Descomprímelo en el proyecto `iip` para poder consultar dicha documentación (fichero `Graph2D.html`) cuando sea necesario.

Actividad 3: completar la clase `Point`

En el paquete `pract5`, completa la clase `Point` siguiendo al pie de la letra los comentarios que aparecen en su cuerpo. Al concluir, la clase debe contener:

1. Las variables de instancia y de clase que se indican en la sección 3.
2. Un método constructor por defecto que crea el punto (0.0, 0.0) y un método constructor que crea un punto de coordenadas dadas como parámetros.
3. Los métodos `get` y `set` asociados a cada una de las variables de instancia.
4. El método `distance` que devuelve la distancia entre el punto `this` y otro punto dado.

5. El método `move` que actualiza las coordenadas del punto `this` a los valores dados como parámetros.
6. El método `equals` que comprueba si dos objetos de tipo `Point` son iguales, es decir, si todos sus atributos coinciden.
7. El método `toString` que devuelve un `String` representando el punto `this` en el formato típico matemático, i.e., `(x,y)`.
8. El método `cross` que implementa el esquema de análisis de la Figura 3 para calcular el cruzamiento.

Actividad 4: completar la clase `RayTest` - salida estándar

La clase `RayTest` es la clase *"Programa"* en la que se comprueba si el método `cross` de la clase `Point` es correcto. Para ello, se escribirán los siguientes métodos:

- Método `main`, en el que se declaran tres vértices:

```
Point vert1 = new Point(3.0, 16.0),
    vert2 = new Point(3.0, 0.0),
    vert3 = new Point(17.0, 0.0);
```

que determinan los lados del triángulo de la Figura 1, de manera que el segmento delimitado por `vert2` y `vert3` es paralelo a los rayos y permite probar el **Caso 1** del análisis de casos, mientras que se puede usar cualquiera de los otros dos segmentos para probar el resto de casos. Se declaran además unos puntos a testear:

```
Point p1 = new Point(-1.0, 18.0), p2 = new Point(2.0, 16.0),
    p3 = new Point(-4.0, 10.0), p4 = new Point(14.0, 7.0),
    p5 = new Point(1.0, 0.0), p6 = new Point(-5.0, -6.0);
```

con los que se pueden cubrir todos los posibles casos del método, como en la Tabla 1.

Tabla 1: Casos de prueba para `p.cross(u, v)`.

Caso	Ejemplos	u	v	p
1	a	vert2	vert3	cualquiera
2	a	vert1	vert2	p2
	b	vert1	vert3	p2
3	a	vert1	vert2	p5
	b	vert1	vert3	p5
4	a	vert1	vert2	p3
	b	vert1	vert3	p3
5	a	vert1	vert2	p1
	b	vert1	vert2	p4
	c	vert1	vert2	p6
	d	vert1	vert3	p1
	e	vert1	vert3	p4
	f	vert1	vert3	p6

En el método se realizarán, al menos, cinco llamadas, escogidas de manera que se cubran los cinco casos posibles del algoritmo (en el código se te da resuelto, como guía, el **Caso 1**). Para cada llamada, se mostrará el resultado en la salida estándar en un formato como el del siguiente ejemplo, que ha tomado los casos de prueba **1a**, **2b**, **3b**, **4b** y **5e** de la Tabla 1:

```
Cruce del segmento (3.0,0.0) a (17.0,0.0) desde (-1.0,18.0) : DONT_CROSS
Cruce del segmento (3.0,16.0) a (17.0,0.0) desde (2.0,16.0) : HIGH_CROSS
Cruce del segmento (3.0,16.0) a (17.0,0.0) desde (1.0,0.0) : LOW_CROSS
Cruce del segmento (3.0,16.0) a (17.0,0.0) desde (-4.0,10.0) : CROSS
Cruce del segmento (3.0,16.0) a (17.0,0.0) desde (14.0,7.0) : DONT_CROSS
```

Nota que, en esta salida, el resultado del método `cross` se muestra con la palabra correspondiente `DONT_CROSS`, `HIGH_CROSS`, `LOW_CROSS`, `CROSS` en lugar del valor entero que devuelve el método. Con este propósito, se usará el método auxiliar `crossToString`, cuyo código se debe completar tal como se describe a continuación.

- Método `crossToString` en el que, **haciendo uso de una instrucción `switch` y de las constantes definidas en la clase `Point`**, debes actualizar adecuadamente el valor de la variable `res`.

Una vez completada y probada esta clase, puedes mejorar su código implementando un método estático `showCross` que, dados tres `Point` que representen un punto y los extremos de un segmento, incluya la llamada al mètodo `cross` y la salida por pantalla que se realizan para cada uno de los casos probados.

Actividad 5: completar la clase `RayTest` - salida gráfica

Haciendo uso de la librería `Graph2D` del paquete `graph2D`, se van a mostrar los casos de prueba en una salida gráfica, de modo que se puedan contrastar con los resultados obtenidos en la actividad anterior. Nota que, para utilizar esta librería, al comienzo de la clase `RayTest`, se ha incluido la directiva de importación de la clase gráfica:

```
import graph2D.Graph2D;
```

Una vez hecho esto, ya es posible definir objetos de dicha clase y operar sobre ellos en la clase `RayTest`. Como se ve en el código que se te proporciona, para crear el espacio de dibujo, se usa el constructor que crea un `Graph2D` a partir de las dimensiones reales de los ejes de coordenadas (`-20`, `20` para el eje X y `-20`, `20` para el eje Y), las dimensiones de la ventana (`600` $\times$ `600`), un color de fondo (`Color.WHITE`) y un título.

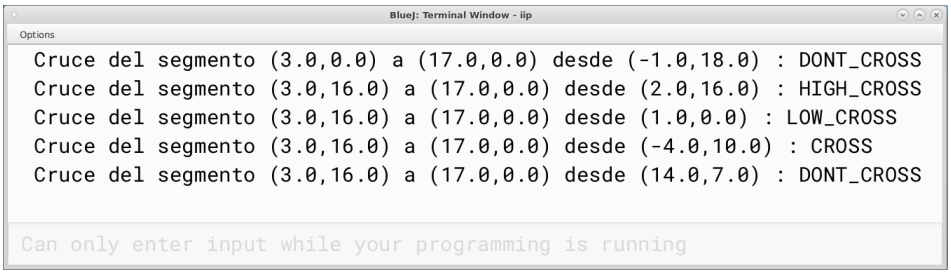
A continuación, el método de instancia `drawLine`, a partir de las coordenadas de los puntos `u` y `v`, un color (`Color.RED`) y un grosor (`3`), dibuja el segmento $\overrightarrow{uv}$ en el espacio de dibujo.

Por último, se invoca al método privado estático `drawRay` (de la clase `RayTest`) para dibujar cada uno de los puntos a testear (con el método `drawPoint`) y sus rayos (con el método `drawLine`). El código de este método auxiliar `drawRay` es el que sigue:

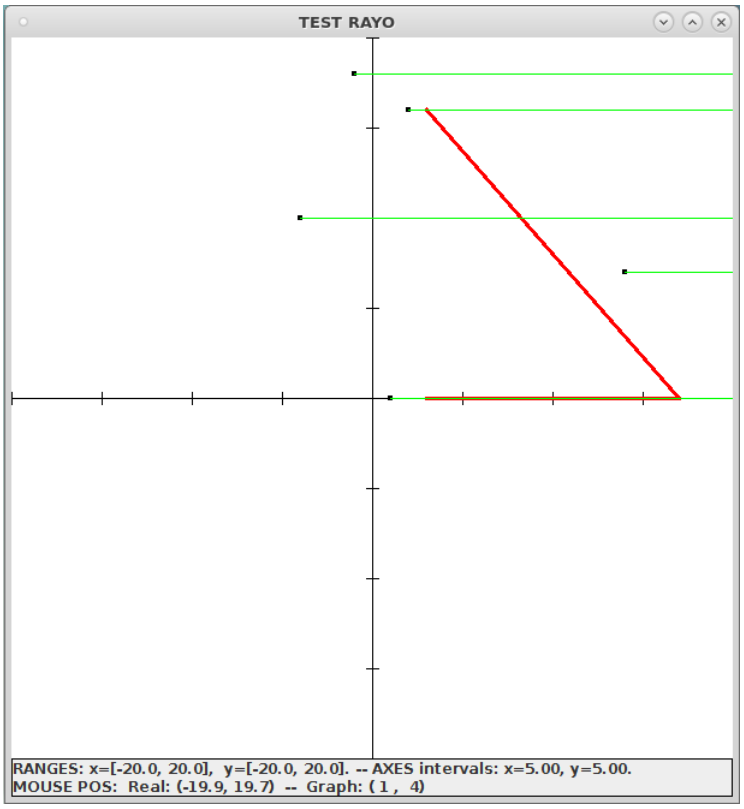
```
/** Dibuja en el espacio de dibujo gd el Point p y su rayo. */
private static void drawRay(Graph2D gd, Point p) {
    gd.drawPoint(p.getX(), p.getY(), Color.BLACK, 4);
    gd.drawLine(p.getX(), p.getY(), 20, p.getY(), Color.GREEN, 1);
}
```

En esta actividad debes completar el dibujo de los rayos de prueba, para visualizar su cruzamiento con el segmento $\overrightarrow{uv}$. Igual que para la salida estándar, en el código se te proporciona, como guía, la resolución del **Caso 1**.

El resultado en la salida gráfica del método `main`, por ejemplo, para los casos de prueba **1a**, **2b**, **3b**, **4b** y **5e** de la Tabla 1, debe ser el que se muestra en la Figura 5(b).



(a) Salida estándar.



(b) Salida gráfica.

Figura 5: Resultado del método `main` para los casos de prueba **1a**, **2b**, **3b**, **4b** y **5e** de la Tabla 1.

Actividad 6: comprobar el estilo de las clases `Point` y `RayTest`

Comprueba que el código de las clases escritas cumple las normas de estilo usando el `Checkstyle` de *BlueJ*, y corrígelo si no es el caso.

Actividad 7: validar la clase `Point`

Cuando tu profesor lo considere conveniente, dejará disponible en *PoliformaT* una clase de prueba (o *Unit Test*) para validar el código de tu clase `Point`. En general, para pasar los tests correctamente, hay que asegurarse de que se usan los mismos identificadores de atributos y métodos propuestos en este documento, siguiendo estrictamente las características sobre modificadores y parámetros propuestos en la cabecera de los mismos. Sigue estos pasos:

1. Descarga el archivo `PointUnitTest.class` sobre el directorio del paquete `pract5` y reabre tu proyecto `iip` desde *BlueJ*.

2. Elige la opción *Test All* del menú contextual que aparece al hacer clic con el botón derecho del ratón sobre el icono de la clase *Unit Test*. Se ejecutarán un conjunto de pruebas sobre los métodos implementados en la clase `Point`, comparando resultados esperados con los realmente obtenidos.

3. Si los métodos están bien, aparecerán marcados con el símbolo ✓ (green color) en la ventana *Test Results* de *BlueJ*. Por el contrario, si alguno de los métodos no funciona correctamente, entonces aparecerá marcado mediante el símbolo `X`. Si seleccionas cualquiera de las líneas marcadas con `X`, en la parte inferior de la ventana se muestra un mensaje más descriptivo sobre la posible causa de error.

4. Si, tras corregir errores y recompilar la clase, el icono de la *Unit Test* está rayada a cuadros, entonces cierra y vuelve a abrir el proyecto *BlueJ*.