LABORATORIO 2: RECURSIVIDAD

Objetivos

Entender el comportamiento de los algoritmos recursivos

Implementación y modificación de algoritmos recursivos

Probar el consumo de recursos de los algoritmos recursivos

Entender los problemas de consumo de memoria en los algoritmos recursivos

Herramientas proporcionadas

En el moodle de la asignatura dispone de herramientas para la corrección automática de entregas de código. Estas herramientas son sensibles al uso de mayúsculas/minúsculas, por lo que deberá respetar escrupulosamente las instrucciones dadas en este guión respecto al nombre de clases y métodos. Si tiene dudas sobre cómo usar esta herramienta puede consultar alguno de los vídeo-tutoriales (*screencast*) disponibles:
http://www.lab.dit.upm.es/~prog/screencast/SubirProyectosEclipseAMoodle.flv

Introducción

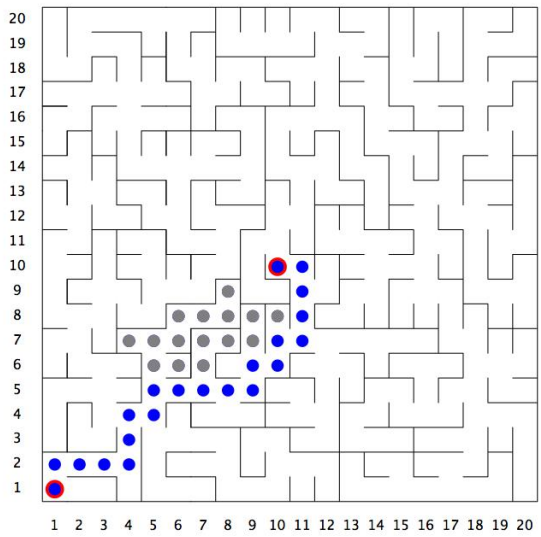
El software de juegos, así como el de optimización o el de búsqueda en grafos suele emplear algoritmos recursivos.

En esta práctica vamos a trabajar con algoritmos que permiten crear y resolver juegos de laberintos. Un ejemplo de este tipo de juegos es el que se emplean en http://algs4.cs.princeton.edu/41undirected/maze.swf

Un juego de laberinto tiene dos actividades fundamentales: construir el laberinto a seguir y resolver el laberinto. Las clases con las que trabajaremos incluyen métodos para las dos actividades. Además incluyen métodos para dibujar los laberintos. En la siguiente figura tenemos un ejemplo de laberinto con el que vamos a trabajar. Algunos detalles específicos de los laberintos con los que trabajaremos son:

1. La posición de salida que vamos a emplea es la 1,1. Representada gráficamente abajo a la izquierda.
2. El laberinto siempre es cuadrado y tiene NxN posiciones. En el ejemplo N=20.
3. La posición que buscamos es la posición central del laberinto. Si N es impar esa posición será (N+1)/2,(N+1)/2, y si es par N/2,N/2.
4. Los algoritmos de construcción de laberintos con los que trabajaremos permiten llegar a cualquier posición del laberinto desde cualquier posición.

5. En las implementaciones con las que trabajaremos, el camino de llegada al punto destino está representado en azul, y en gris están representados puntos por los que hemos pasado, y nos han llevado a puntos muertos o a puntos por los que ya hemos pasado (debemos evitar meternos en ciclos de donde no salimos). En rojo están marcados el punto origen y destino.



La práctica emplea la clase *StdDraw*. Es una clase que permite dibujar de forma sencilla elementos gráficos como líneas, círculos, puntos, …. Pero a diferencia de las bibliotecas estándar de Java, esta clase localiza las posiciones y tamaños de los objetos mediante parámetros *double* y permite escalar las figuras de forma sencilla (frente a los valores enteros absolutos de las bibliotecas estándar). Este escalado permite ver en el mismo espacio gráfico laberintos de diferente tamaño, sin tener que cambiar apenas nuestro programa. Otra diferencia importante es que la coordenada de origen del espacio gráfico está abajo a la izquierda (frente a la posición de arriba a la derecha, que es como suelen trabajar las bibliotecas Java). Esto no es mas que un simple resumen, porque la representación gráfica no es un objetivo fundamental de la práctica.

Actividades a desarrollar

Descarga del proyecto desde GitHub (Trabajo previo)

La estructura del proyecto a desarrollar, así como algunas de las clases necesarias se encuentran disponibles en el repositorio de la asignatura en GitHub: https://github.com/ALED-UPM/Tema2

Antes del inicio de la sesión de laboratorio, deberá:

1. **Configurar EGit** para acceder al repositorio Tema2 de la asignatura.
2. **Clonar el repositorio Tema2**.
3. **Importar el proyecto ALED-lab2** a su Eclipse.

4. **Explorar la clases *Laberinto* y *StdDraw*.**

5. **Contestar a las preguntas 1-3.**

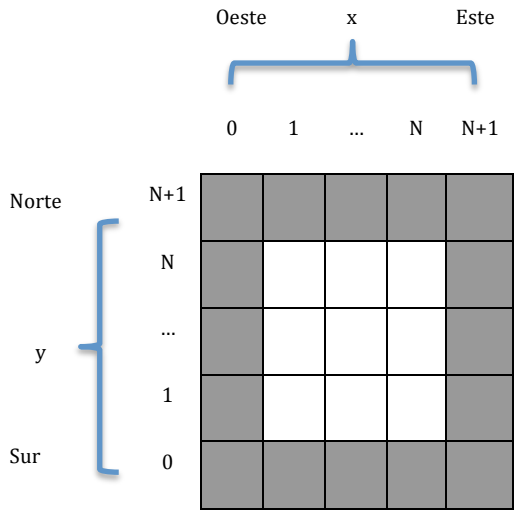
Implementar el método *resolver* (Trabajo en laboratorio)

Vamos a implementar el método que permite encontrar el camino desde la posición origen a la posición destino. La búsqueda del camino se implementa en dos métodos sobrecargados *resolver()* y *resolver(int x,int y).* El primero es público, es un método fachada; hace inicializaciones antes de empezar la búsqueda, y llama al segundo que es privado y es donde realmente se busca el camino. Nos centraremos en este segundo.

La clase *Laberinto* incluye los siguientes atributos:

```
protected boolean[][] norte;
protected boolean[][] este;
protected boolean[][] sur;
protected boolean[][] oeste;
protected boolean[][] visitado;
```

El constructor de *Laberinto* nos ha dejado inicializados *norte*, *sur, este* y *oeste*. El método *resolver()* (el punto de entrada fundamental de la clase) deja inicializado *visitado*. Todos estos atributos son arrays de tamaño N+2,N+2. *norte, sur, este*, y *oeste* nos indican cuando una posición tiene una pared en sentido norte, sur este u oeste. Los fija el generador de caminos (la implementación de la construcción del laberinto) y no debemos modificarlos. Son variables algo redundantes. Están así implementadas para hacer mas sencillos los algoritmos de *resolver*, pero debemos saber que *norte[x][y] == sur[x][y+1]* (para *y >= 0* e *y <= N*) y *este[x][y] == oeste[x+1][y]* (*para x >= 0 y x <= N*).



El atributo *visitado* nos servirá para controlar los caminos por los que anteriormente hemos pasado y no nos han llevado a una solución.

El tamaño de todos estos arrays es N+2,N+2 mientras que el laberinto como tal está representado en los rangos 1..N,1..N. Todos los arrays tienen un marco exterior para identificar los límites del laberinto. Por ejemplo, *visitados[0][y], visitados[N+1][y], visitados[x][0]* y *visitados[x][N+1]* están inicializados a **true**, el resto a **false**. Deben estar siempre a **true** *norte[x][0], sur[x][N+1], este[0][y]* y *oeste[N+1][y]*.

Empezaremos implementando el método *resolver(int x, int y)* de forma determinista (intentamos buscar caminos siempre en el mismo orden). Dos ejecuciones con el mismo laberinto, encuentran siempre el mismo resultado de la misma forma. El algoritmo que vamos a seguir parte de una posición *x,y* (x > 0, x < N+1, y > 0, y < N+1). Si esa posición ya ha sido visitada, la descartamos; si es la posición central (*Math.round(N/2.0), Math.round(N/2.0)*) hemos encontrado el punto central. Si no es ninguno de esos casos marcamos la posición como visitada, la pintamos de azul e intentamos **resolver,** de forma recursiva, moviéndonos en dirección norte, si no hay pared en esa dirección, después en dirección este, después sur y después oeste, siempre y cuando no haya paredes. Hemos elegido como orden las agujas del reloj. Si por ninguno de esos caminos hemos llegado a la posición central, pintamos la posición de gris, y retornamos diciendo que hemos llegado a punto muerto o visitado.

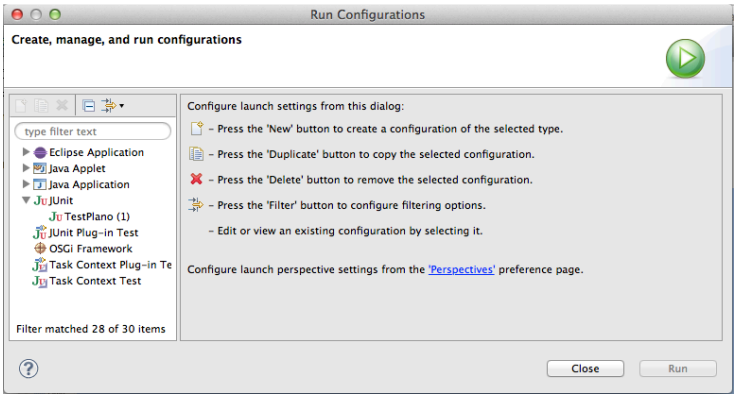
El pseudo código del algoritmo nos queda:

```
resolver(int x, int y)

    Si encontrado o visitado[x][y] return;
    visitado[x][y] = true;

    // marcamos la posicion de azul
    StdDraw.setPenColor(StdDraw.BLUE);
    StdDraw.filledCircle(x + 0.5, y + 0.5, 0.25);
    StdDraw.show(0);

    Si hemos llegado el punto central
       encontrado = true;

    Si no hay pared en norte[x][y]  resolver(x, y + 1);
    Si no hay pared en este[x][y]  resolver(x + 1, y);
    Si no hay pared en sur[x][y]  resolver(x, y - 1);
    Si no hay pared en oeste[x][y]  resolver(x - 1, y);

    Si encontrado return;

    // marcamos la posicion de gris
    StdDraw.setPenColor(StdDraw.GRAY);
    StdDraw.filledCircle(x + 0.5, y + 0.5, 0.25);
```

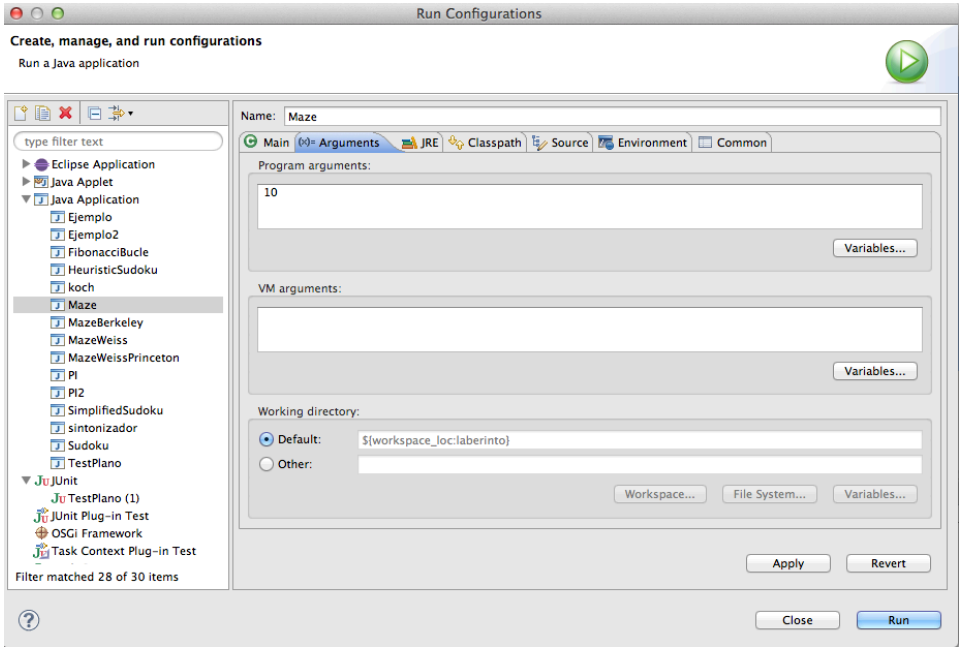
```
StdDraw.show(0);
```

Implementar el algoritmo y compilarlo.

Para probar el algoritmo tenemos que configurar una ejecución que pase como argumento el valor N. Creamos una configuración de ejecución con *Run->Run Configurations…*; seleccionamos *Java Applications* (entre los tipo de ejecuciones de la ventana a la izquierda) y creamos una nueva configuración de ejecución, presionando el botón de nueva configuración (arriba a la izquierda).



En la carpeta principal de la configuración de ejecución seleccionamos el proyecto, y la clase *Laberinto*, y en la carpeta de argumento ponemos *10* como argumento de programa.



1. Prueba la ejecución del algoritmo con números entre 5 y 100 (números mayores pueden crearnos problemas como veremos mas adelante).
2. **Responda a las preguntas 4-6** al final de este documento.
3. **Documente** la clase y genere la documentación **javadoc** correspondiente.

Algoritmo con exploración aleatoria y tiempos de ejecución (Trabajo en laboratorio)

Realizar las siguientes pruebas:

1. En el ejemplo anterior la exploración de caminos la hacemos dando prioridad al norte frente al resto, al este frente al sur y oeste, y al sur frente al oeste. Podemos cambiar el algoritmo para explorar de forma aleatoria. Podemos utilizar *Math.random()* para elegir de forma aleatoria el camino que seguiremos (dentro de los posibles). Debemos modificar el algoritmo para que tengan la misma probabilidad la exploración de cualquiera de los caminos.

2. Modificaremos el método público *resolver()* para sacar por salida estándar cuanto tiempo se ha tardado en resolver el camino. Podemos incluir como primera sentencia de este método
   ```
   long t=System.currentTimeMillis();
   ```
   y como última:
   ```
   System.out.println("Tiempo total "+(System.currentTimeMillis()-t));
   ```

3. Podemos ver que buena parte del tiempo se va en la representación gráfica. Si comentamos las sentencias
   StdDraw.show(0);
   del método *resolver(int x, int y)* y la introducimos al final del método *resolver(),* solamente se hacen las actualizaciones gráficas cuando la exploración está terminada. Compara los tiempos de ejecución de esta forma y la anterior (cuando se actualiza la representación gráfica por cada paso).

4. **Responda a las preguntas 7-10** al final de este documento.

5. Podemos generar errores en el programa si ejecutamos con una N=1000.

Implementación alternativa de método *generar* (Trabajo después de laboratorio)

Si ejecutamos el programa con un tamaño del orden de 200..1000, la ejecución genera una excepción (*StackOverflowError*) en el método *generar (*el método que fija que caminos se pueden seguir dejando paredes levantadas o tiradas). Ese error es consecuencia de la gran cantidad de anidamientos recursivos que tiene el método. Hay tantas llamadas pendientes de que terminen sus llamadas recursivas, que no hay espacio de pila suficiente para acumular la información de las llamadas pendientes de finalizar.

Si exploras el algoritmo de ese método verás que no es muy diferente que el algoritmo con el que hemos resuelto la solución del laberinto. En este caso el algoritmo parte de un laberinto con todas las paredes levantadas y va tirando de forma aleatoria paredes de la posición en la que nos encontramos, para con ello pasar a una posición vecina no visitada. De esa forma se derriban de forma aleatoria paredes; esto se hace mientras que una cierta posición tenga vecinos no visitados. Como esto se hace de forma aleatoria, no podemos asegurar que camino seguiremos. Pero en el peor caso podríamos llegar a tener un nivel de recursividad del orden de NxN (partimos de una posición no visitada, y siempre que pasamos a una nueva, esa tiene vecinos no visitados; hasta llegar al último no visitado). Si la N es muy grande el programa se queda sin pila donde almacenar parámetros, y retornos.

Probar la siguiente solución: Podemos incrementar el tamaño del laberinto si incrementamos el tamaño de la pila. Para eso debemos modificar la configuración de ejecución. La segunda figura de la página 5, muestra la carpeta en la que pasamos argumentos al programa y a la máquina virtual. Para incrementar el tamaño podemos pasar como argumento a la máquina virtual (*VM arguments*): *-Xss1000m.* En este caso estamos pidiendo una pila de 1000 Mega bytes. El tamaño que podemos poner depende de la memoria de nuestra máquina, y de la versión de máquina virtual con la que trabajamos. ¿Es posible ejecutar un laberinto de tamaño 300? Tamaños muy grandes pueden dar otros tipos de problemas en la máquina virtual.

Escribir una nueva clase LaberintoTodoAccesible

Las implementaciones que hemos hecho tienen sentido si el laberinto tiene solución. Hay varios algoritmos para generar el laberinto; la solución que empleamos en la clase *Laberinto* genera laberintos con solución, y con todas las posiciones del laberinto accesibles.

Vamos a implementar la clase ***LaberintoTodoAccesible*** que extenderá la clase *Laberinto*, y añadirá métodos que permitan comprobar que todas las posiciones del laberinto son accesibles antes de resolverlo. ***LaberintoTodoAccesible*** incluirá el método:

```
public boolean compruebaAccesibilidad()
```

Que devuelve *true* cuando todas las posiciones del laberinto son accesibles, y *false* cuando alguna posición no es accesible partiendo desde la posición 1,1.

Realizar los siguientes pasos:

1. ***Implementar el método compruebaAccesibilidad***.
2. **Documente** el método y los posibles métodos añadidos.
3. La clase incluye la implementación de:

   ```
   public boolean ejecutaConControlAccesibilidad(int numFilasColumnas)
   ```

   con una implementación similar al método *main* de *Laberinto*, que construye un laberinto, comprueba que todas las posiciones son accesibles, y si es así, resuelve el laberinto.
4. Ejecutar la clase ***LaberintoTodoAccesible*** y su método *main* que hace llamadas a `ejecutaConControlAccesibilidad` pasando como parámetro el número de filas del laberinto.
5. Hacer pruebas con diferentes tamaños (5-300).

Como la clase *Laberinto* siempre genera laberintos con todas las posiciones accesibles, vamos a crear un método en la clase ***LaberintoTodoAccesible*** que nos permita aislar posiciones del laberinto, al que podemos llamar después de haber construido el laberinto, pero antes de comprobar la accesibilidad y de resolver el laberinto. La cabecera puede ser:

```
protected void aisla(int x, int y)
```

Este método debe levantar las 4 paredes de la posición que fijan los parámetros x,y (teniendo en cuenta que esas paredes deben ser consistentes con las de sus posiciones vecinas). Tenemos implementado el método:

```
public boolean ejecutaConControlAccesibilidadYAislado(int numFilasColumnas)
```

Con la misma implementación que ***ejecutaConControlAccesibilidad*** pero ejecutando ***aisla*** para una cierta posición del laberinto, después de haber construido el laberinto y antes de comprobar la accesibilidad. El método *main* puede hacer llamas a `ejecutaConControlAccesibilidad` o a `ejecutaConControlAccesibilidadYAislado` para hacer ejecuciones con posiciones aisladas o no aisladas.

6. Implementar los métodos ***aisla***.
7. Documente estos los métodos.
8. Hacer pruebas con diferentes tamaños (5-300).

Entregas

Instrucciones para la entrega **durante la sesión de laboratorio**:

• Las **preguntas** planteadas deben **entregarse en papel al final de la sesión**. Las preguntas 1-3 estarán resueltas antes del laboratorio, y el resto podrán resolverse en el laboratorio.

• El **código fuente y la documentación** de las clase ***Laberinto***. Deberá entregarse en el moodle de la asignatura antes del final de la sesión de laboratorio.

• El nombre del fichero que incluya los ficheros requeridos no debe contener tildes ni espacios.

• En el contenido del fichero comprimido de la entrega, el código fuente debe estar en el directorio relativo: "ALED-lab2/src/es/upm/dit/aled/lab2/".

Instrucciones para la entrega a realizar **después del laboratorio**:

- El **código fuente y la documentación** de las clases ***LaberintoTodoAccesible*** **y Laberinto** deberán entregarse en el Moodle de la asignatura **antes de las 23:59 del 19 de octubre**, siguiendo las mismas instrucciones que la entrega de laboratorio respecto al nombre y ruta de los ficheros.

- Al realizar la entrega se le informará, que estará **sujeta a revisión** por parte del profesor, así como información sobre algunos **fallos detectados**. La evaluación de práctica se hará con el siguiente criterio:

    - 7 puntos máximo por la corrección del código de la clase *Laberinto* y *LaberintoTodoAccesible* ;

    - 3 puntos máximo por la corrección de la documentación;

- Para la evaluación de la entrega **sólo se considerará el último envío realizado**.

- **No se aceptarán entregas fuera de plazo** bajo ningún concepto.

AVISO MUY IMPORTANTE

Se recuerda a los alumnos que el trabajo es individual, y que la copia de entregas supondrá el **suspenso en la asignatura de forma automática**, **tanto para quien copia como para quien se deja copiar**.

No está permitido:

• Realizar este trabajo en grupo.

• Copiar el trabajo de otro alumno, ni permitir la copia del propio trabajo, ni siquiera parcialmente.

• Usar código publicado sin citar el origen.

Responda a las siguientes preguntas según se le indica en el enunciado del laboratorio. Entregue al profesor esta hoja con sus respuestas antes de la sesión.

Nombre:_____ **DNI/NIE**:_____

1. Enumera los métodos que tiene la clase *Laberinto*. ¿Cual es el método que está sin implementar?

2. ¿Qué argumentos presupone que tendrá el método *main*?

3. ¿Desde dónde se llama al método *generar*? ¿Que hace ese método?

4. ¿Cuales son las sentencias de *Ejecución Base* de la implementación del algoritmo?

5. ¿Cuales son las sentencias de *Ejecución Recursiva* de la implementación del algoritmo?

6. ¿Cuales son las sentencias de *Cálculos Generales* de la implementación del algoritmo?

7. ¿Cómo modificas el algoritmo para seleccionar de forma aleatoria un camino, teniendo en cuenta que no todos los caminos se pueden explorar?

8. ¿Cuánto es el tiempo de ejecución para un laberinto de tamaño 10? (cuando la actualización gráfica solo se hace al final de la resolución)

9. ¿Cuánto es el tiempo de ejecución para un laberinto de tamaño 50? (cuando la actualización gráfica solo se hace al final de la resolución). Si los métodos de generación de laberintos fallan comenta que esto ha pasado y reduce el tamaño del laberinto para que esto no suceda.

10. ¿Cuánto es el tiempo de ejecución para un laberinto de tamaño 100? (cuando la actualización gráfica solo se hace al final de la resolución). Si los métodos de generación de laberintos fallan comenta que esto ha pasado y reduce el tamaño del laberinto para que esto no suceda.