Curso 2016-2017
Estructura de Datos y Algoritmia
Práctica 2 (16 de octubre)

Departamento de Lenguajes y Sistemas Informáticos

Objetivo

En esta segunda práctica trabajaremos con tipos de datos lineales y los utilizaremos en una aplicación concreta. En particular:

1. Nos familiarizaremos con algunas colecciones de datos lineales del API de Java.
2. Se implementará el tipo de datos lista.
3. Aplicación de los tipos de datos anteriores junto con las clases implementadas en la primera práctica para la resolución de un problema de exploración de caminos.

Plazo de entrega: desde el lunes 21 de noviembre hasta el **viernes 25 de noviembre**.

Más información al final de este documento.

1. Clase Lista

Aunque somos conscientes de que podríais utilizar cualquiera de las implementaciones de la lista que ofrece Java en el API, tenéis que ser capaces de hacer vuestra propia implementación.

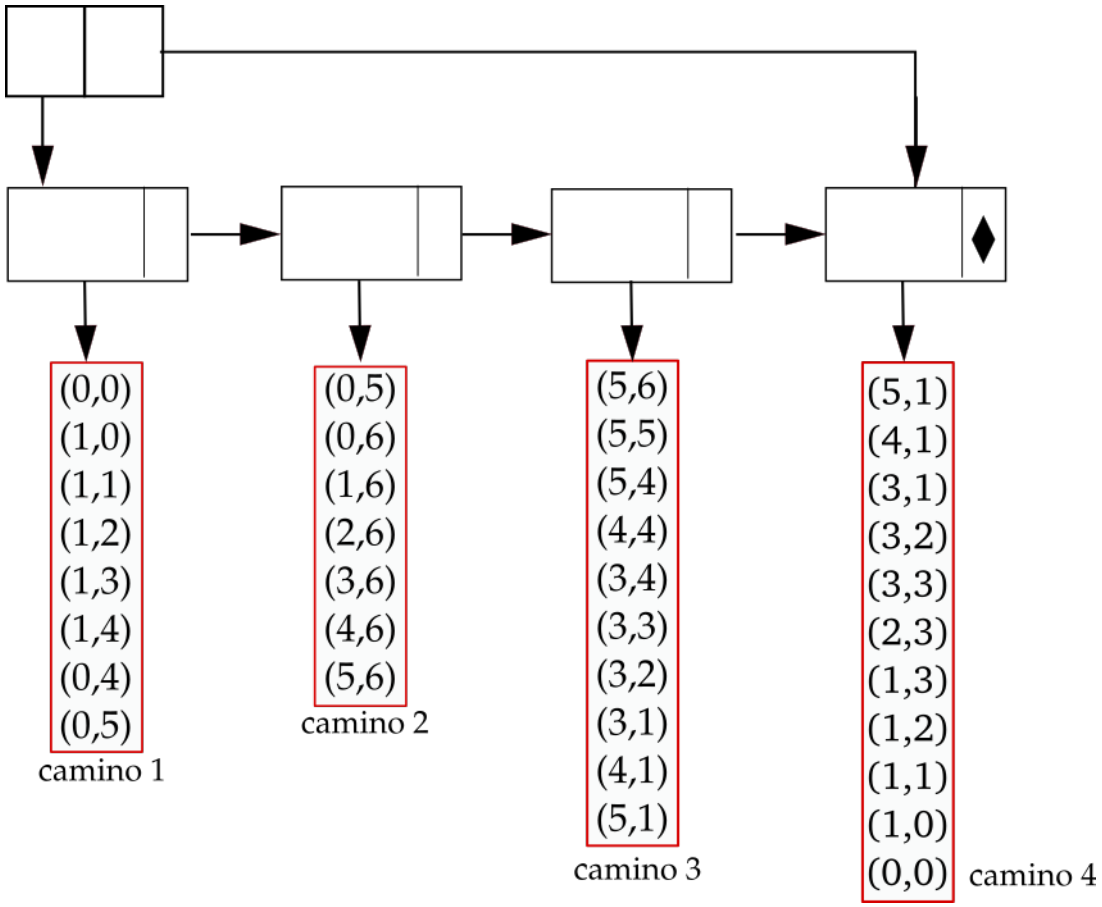
La clase **Lista** que hay que implementar contendrá:

- las variables de instancia:
  - `private NodoL pr;`
  - `private NodoL ul;`
- y los métodos de instancia:
  - `public Lista()`: inicializa las variables de instancia a sus valores por defecto (`null`).
  - `public boolean esVacia()`: nos indica si la lista está vacía.

- `public void insertaCabeza(ArrayList<Coordenadas> v)`: crea un nodo que contendrá a `v` y lo insertará al principio de la lista, si `v` no es nulo.
- `public void insertaCola(ArrayList<Coordenadas> v)`: crea un nodo que contendrá a `v` y lo añadirá al final de la lista, si `v` no es nulo.
- `public void inserta(ArrayList<Coordenadas> v, int i) throws IndexOutOfBoundsException`: crea un nodo que contendrá a `v` y lo añadirá en la posición `i` de la lista, si `v` no es nulo y la posición existe en la lista, empezando en el 0. Si `i` no es una posición válida de la lista el método debe lanzar y propagar la excepción `IndexOutOfBoundsException`.
- `public boolean borraCabeza()`: quita de la lista el primer objeto (a la cabeza de la lista), devolviendo `true` si la operación se ha podido realizar.
- `public boolean borraCola()`: quita el último objeto (a la cola) de la lista, devolviendo `true` si la operación se ha podido realizar.
- `public void borra(int i) throws IndexOutOfBoundsException` : borra el nodo que ocupa la posición `i` de la lista, si la posición existe en la lista, empezando en el 0. Si `i` no es una posición válida de la lista el método debe lanzar y propagar la excepción `IndexOutOfBoundsException`.
- `public boolean borra(ArrayList<Coordenadas> v)`: quita la primera ocurrencia de `v` en la lista, si no la encuentra devuelve `false`.
- `public void escribeLista()`: escribe la lista a la salida estándar, con el siguiente formato:

  camino 1: (0,0)(1,0)(1,1)(1,2)(1,3)(1,4)(0,4)(0,5)
  camino 2: (0,5)(0,6)(1,6)(2,6)(3,6)(4,6)(5,6)
  camino 3: (5,6)(5,5)(5,4)(4,4)(3,4)(3,3)(3,2)(3,1)(4,1)(5,1)
  camino 4: (5,1)(4,1)(3,1)(3,2)(3,3)(2,3)(1,3)(1,2)(1,1)(1,0)(0,0)

Gráficamente:



- `public int enLista(ArrayList<Coordenadas> v)`: nos dice si `v` está en la lista o no. Si está, devuelve su posición, en otro caso devuelve -1.
- `public ArrayList<Coordenadas> getCamino(int pos) throws IndexOutOfBoundsException`: devuelve el camino contenido en el nodo de la lista que ocupa la posición `pos` (empezando a contar desde 0). Si `pos` no es una posición válida de la lista o la lista está vacía el método debe lanzar y propagar la excepción `IndexOutOfBoundsException`.

2. Clase NodoL

Esta clase se definirá como privada de la clase Lista (de manera que sólo la lista puede acceder a un objeto de tipo NodoL). Contendrá

- las variables de instancia siguientes:
  - `private ArrayList<Coordenadas> camino;`
  - `private NodoL next;`
- y los siguientes métodos de instancia:

- `public NodoL( ArrayList<Coordenadas> v)`: inicializa `camino` al valor pasado como parámetro y el nodo a `null`.
- `public NodoL( ArrayList<Coordenadas> v, NodoL n)`: inicializa `camino` al valor pasado como parámetro y el nodo a `n`.
- `public void cambiaNext( NodoL n)`: cambia el valor del nodo apuntado por `next` a `n`.
- `public NodoL getNext( )`: devuelve el nodo apuntado por `next`.
- `public ArrayList<Coordenadas> getCamino()`: devuelve el camino contenido en el nodo.
- `public void escribeCamino()`: muestra por pantalla ordenadamente (de la primera a la última) las coordenadas que forman el camino que contiene el nodo sin separadores (espacios en blanco, retornos de carro, ...). Ejemplo de salida para el último nodo de la lista de la figura:
  (5,1)(4,1)(3,1)(3,2)(3,3)(2,3)(1,3)(1,2)(1,1)(1,0)(0,0)
- `public boolean comparaCamino(ArrayList<Coordenadas> v)`: devuelve cierto si `v` es igual al vector almacenado en el nodo.

3. Clase exPath

Esta clase, que se utilizará para explorar el plano y encontrar el camino a cada destino, contendrá:

- las variables de instancia:
  - `private Coordenadas coor;`
  - `private ArrayList<Coordenadas> camino;`
- y los métodos de instancia:
  - `public exPath(int x, int y, ArrayList<Coordenadas> v)`: crea el objeto Coordenadas `coor` inicializándolo con los valores `x` e `y`, crea un duplicado del array de coordenadas `v` en `camino` y le añade la coordenada creada;
  - `public Coordenadas getCoordenadas()`: devuelve las coordenadas almacenadas en la variable de instancia `coor`;
  - `public ArrayList<Coordenadas> getCamino()`: devuelve el camino almacenado en la variable de instancia `camino`;

Esta clase se usará en la aplicación de la práctica, y sus variables de instancia se usarán para almacenar las coordenadas de la casilla sobre la que nos encontramos (en `coor`) y para guardar el camino recorrido hasta el momento (en `camino`).

4. Tipos lineales en el API de Java

En la práctica se podrán utilizar los tipos `ArrayList` y `Stack` del API de Java.

- En el caso del tipo `ArrayList`, los objetos almacenados serán de tipo `Coordenadas`;
- En el caso del tipo Pila (`Stack`), los objetos almacenados serán de tipo `exPath`, por lo que la declaración de un objeto de tipo `Pila` será: `Stack<exPath> v;`

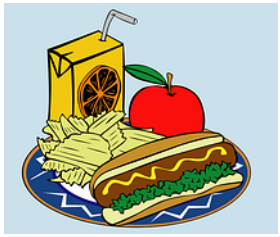
Además se utilizarán la clases `Plano`, `Coordenadas` y `Casilla` implementadas en la primera práctica para poder leer y explorar planos.

5. Aplicación: juego de supervivencia

Te despiertas en un bosque con un gran dolor de cabeza, fruto de algún golpe recibido para que perdieras el conocimiento. Todo lo que sabes es que la zona está totalmente amurallada y tienes que salir de allí lo antes posible de alguna manera. Sin embargo, para tener éxito con este objetivo hay que encontrar comida para sobrevivir. El bosque tiene el aspecto de un laberinto en el que te puedes encontrar caminos inaccesibles, trampas, pero también ayuda.

La exploración empieza en el punto definido como comienzo. A partir de ahí debes realizar movimientos de exploración siguiendo las órdenes que te han indicado en la tarjeta que te encontrarás en el bolsillo de tu cazadora.

En todo momento tienes que evaluar tu posición y los destinos alcanzables desde ella, destinos que sabes te proporcionarán el alimento necesario. Cuando determines que la posición actual es peligrosa o que no puedes seguir explorando con seguridad, debes realizar movimientos de retroceso: **vuelve progresivamente sobre tus propios pasos** hasta llegar a una posición anterior más satisfactoria, y continuar desde ella la exploración por otros caminos. Hay que tener en cuenta que, cuando retrocedes, no tienes por qué realizar el camino completo de regreso, sino que puedes retroceder hasta un punto intermedio y continuar desde él su exploración. Una vez que llegues a un destino, cogerás el paquete de comida que encuentres, aprovecharás para retomar fuerzas con esta comida, y descansarás un rato; después continuarás hasta el siguiente destino. Este proceso lo repetirás mientras existan destinos en el plano y no encuentres ninguna puerta que te permita salir. Si no se da esta situación, regresarás a la posición de comienzo donde te recogerán.



Implementación

El terreno a explorar se implementa con un `Plano`, donde las posiciones peligrosas o comprometidas (barrancos, montañas, simas ...) se marcan en la matriz con una letra concreta. En concreto:

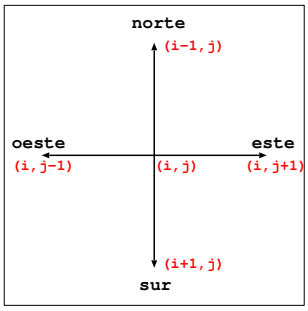
- cualquier obstáculo se representa por la letra `'o'`;
- la posición inicial de comienzo se representa por la letra `'c'`;
- los diferentes objetivos (destinos) se representan por la letra `'d'` y podrán encontrarse en cualquier zona del plano que se ha utilizado para realizar la exploración;
- las puertas pueden ser de ayuda o ser contratiempos en el camino, se especificará en cada caso estas situaciones, vendrán representadas por la letra `'p'`;
- las posiciones por las que pasas hasta un destino se van marcando con la letra `'v'` (evitando de esta forma entrar en bucles infinitos);
- las posiciones viables y no visitadas estarán marcadas por la letra `'l'`;

El terreno a explorar se leerá desde un fichero de texto con el formato visto en la primera práctica. Toda la información correcta se almacenará en un objeto de tipo `Plano`.

El programa utilizará una `Pila` de objetos de tipo `exPath` para implementar tus movimientos y poder obtener los caminos:

- La exploración comienza almacenando en la pila la salida [1], esto es:
  - un objeto de la clase `exPath` con la coordenada `(x,y)` correspondiente, y un camino vacío al que añade la coordenada de partida;
- Entonces, mientras la pila no esté vacía:
  - se desapila la cima de la pila y el elemento desapilado será la posición actual;
  - consultamos en el plano las características de esa posición actual:
    - si es un comienzo (letra `'c'`), entonces debemos marcar esa posición en el plano como visitado, y apilar `en el orden (norte, sur, este, oeste)` con tantos objetos `exPath`, con esa posición como coordenadas, como posibles caminos a seguir haya. Este sentido de la exploración podrá cambiar en algún momento a `(sur, norte, oeste, este)`;

[1] que es la casilla de comienzo del plano



- si es una posición libre (letra `'l'`), entonces debemos marcar esa posición en el plano como visitado, y apilar `en el orden (norte, sur, este, oeste)` con tantos objetos `exPath`, con esa posición como coordenadas, como posibles caminos a seguir haya. Este sentido de la exploración podrá cambiar en algún momento a `(sur, norte, oeste, este)`;
- si es un destino (letra `'d'`), entonces tenemos que insertar en la cola de la lista el camino asociado a esa posición. Para continuar el proceso de buscar más destinos hay que realizar una serie de inicializaciones:
  - cambiamos la naturaleza del destino alcanzado de `'d'` a `'l'`;
  - reinicializamos todas las posiciones visitadas (`'v'`) a su situación original (`'l'`);
  - vaciamos la pila.

  Por último, apilamos en la pila el destino alcanzado, que corresponde ahora a la nueva salida (como se ha explicado anteriormente);
- si es una puerta (letra `'p'`), tenemos que descubrir la propiedad de la misma, y se nos puedan dar varias circunstancias (las propiedades vienen representadas por letras a su vez):
  - si es una `'B'` la puerta está bloqueada (es como si fuera un obstáculo), por lo tanto hay que retroceder a la posición anterior;
  - si es una `'P'` la puerta se abrirá sin problema, podemos seguir;
  - si es una `'S'` podemos ir desde ese punto hasta el siguiente destino no visitado más cercano [2] (si existe) sin tener en cuenta obstáculos ni nada [3]; si no quedan destinos por visitar se considera el comienzo como un destino;
  - si es una `'T'` es una trampa, debemos desactivarla (pasa a ser de tipo `'P'`) para no volver a caer en ella, pero como ya hemos caído nos teletransportamos al comienzo, por lo tanto añadimos las coordenadas del comienzo al camino realizado hasta ahora. Este camino se encola en la lista y se reinicializan las casillas a su valor inicial, se vacía la pila y se apila las coordenadas del comienzo para empezar un nuevo camino cambiando el sentido de la exploración [4];

[2] usando la distancia definida en la práctia 1

[3] el camino mostrado será hasta esa puerta y de ahí a las coordenadas del destino encontrado

[4] ojo, los destinos ya visitados anteriormente ya no existen

◇ si es una `'X'` hemos encontrado una salida, encolamos el camino y terminamos;

Hay que tener en cuenta que cuando pasamos por una puerta, ésta sigue siéndolo como tal.

- Una vez que se han visitado todos los destinos (la pila está vacía y no se han encontrado más destinos), si no hemos encontrado la salida en una puerta, toca volver al comienzo. Para ello, tenemos que

  - reinicializar todas las posiciones visitadas (`'v'`) a su situación original (`'l'`);
  - hacemos un último recorrido **desde la última posición visitada** hasta la casilla de comienzo.

  Siguiendo el algoritmo anterior, debemos seguir las mismas acciones realizadas con los otros destinos para volver al comienzo. Este último camino hay que añadirlo también a la cola de la lista. En caso de que no se alcance el comienzo se muestra el mensaje ``` ``HAS PERDIDO EL JUEGO'' ```.

- Por último, hay que mostrar la lista de caminos explorados.

Formas parte de un equipo de supervivencia, y se te ha encargado una prueba que evaluará tu destreza, registrando en una pila tus movimientos de tal forma que sepas volver en todo momento a la posición del comienzo deshaciendo movimientos hechos previamente, salvo que encuentres una salida. Solo tú, y gracias a tu experiencia, serás capaz de acabar con éxito esta misión.

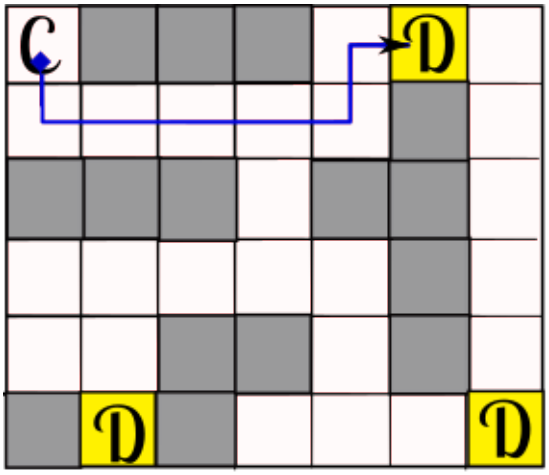
Para ello debes implementar una clase `Exploracion` donde ejecutarás la aplicación. Esta clase recibirá como parámetros de entrada tres argumentos, donde:

- `arg1` representa el fichero con la información del plano
- `arg2` representa las filas del plano
- `arg3` representa las columnas del plano

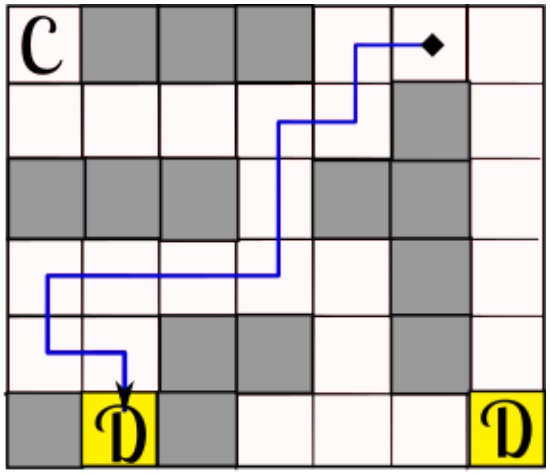
En la misma clase tienes que implementar un método main que:

- abra el fichero de texto que se te pasa como parámetro para lectura y lo almacene en un objeto de tipo `Plano`, cerrándolo posteriormente; el fichero será correcto, en cuanto a que no se desechará ninguna información, siempre habrá un comienzo y las coordenadas de las puertas, destinos y comienzo serán válidas en el plano;
- ejecute el algoritmo de exploración descrito, escribiendo al final la lista de caminos recorridos con el método de lista `void escribeLista()` y, en su caso, el mensaje asociado cuando se pierde el juego.

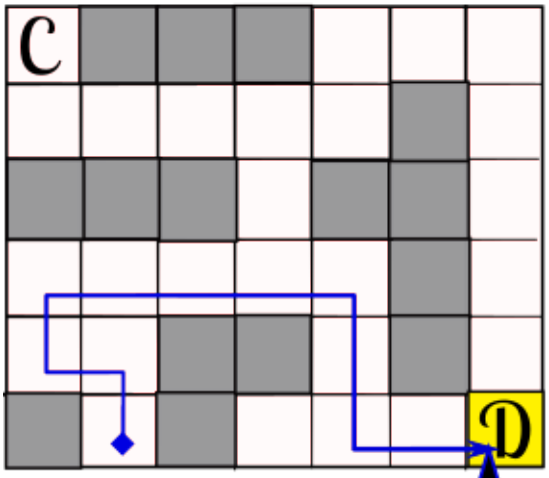
A continuación hay varios ejemplos de la ejecución del problema; en el primero no hay puertas:



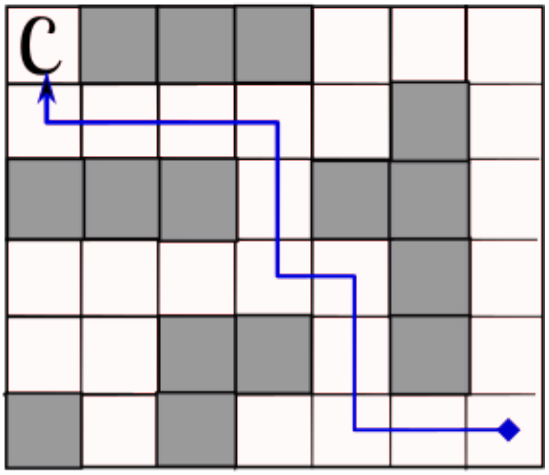
camino 1: (0,0)(1,0)(1,1)(1,2)(1,3)(1,4)(0,4)(0,5)



camino 2:(0,5)(0,4)(1,4)(1,3)(2,3)(3,3)(3,2)(3,1)(3,0)
(4,0)(4,1)(5,1)

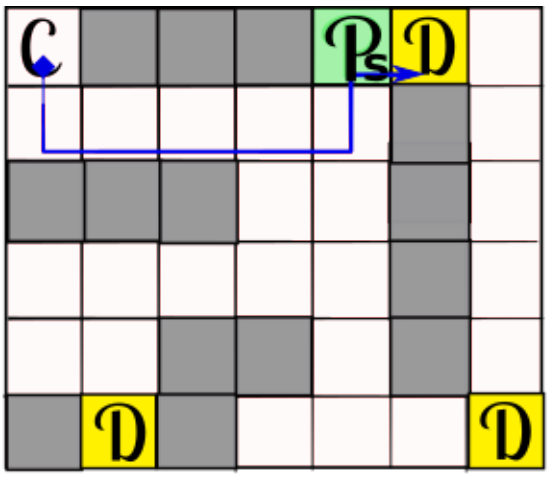


camino 3:(5,1)(4,1)(4,0)(3,0)(3,1)(3,2)(3,3)(3,4)(4,4)
(5,4)(5,5)(5,6)

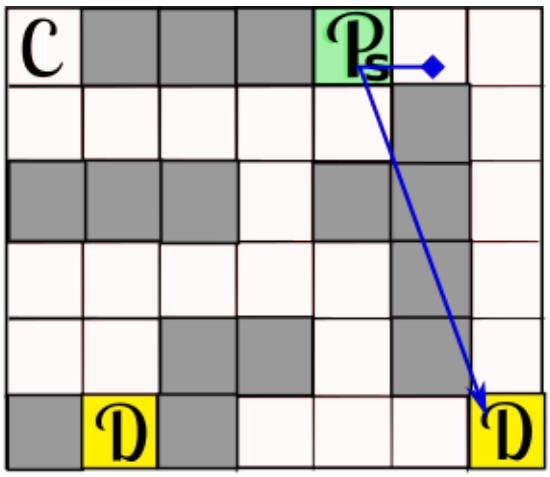


camino 4:(5,6)(5,5)(5,4)(4,4)(3,4)(3,3)((2,3)(1,3)(1,2)
(1,2)(1,0)(0,0)

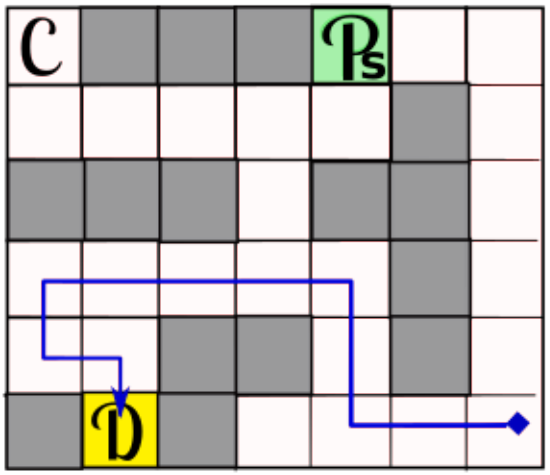
En el segundo hay una puerta 'S', entonces se busca el destino más cercano no visitado y se salta directamente allí (usando la distancia vista en la práctica 1):



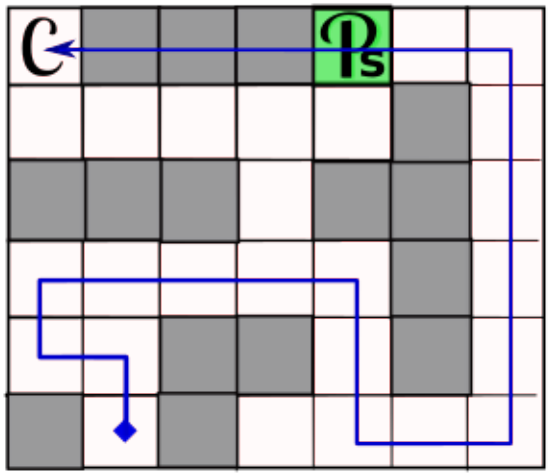
camino 1:(0,0)(1,0)(1,1)(1,2)(1,3)(1,4)(0,4)(0,5)



(la puerta hace saltar al destino más cercano)
camino 2:(0,5)(0,4)(5,6)

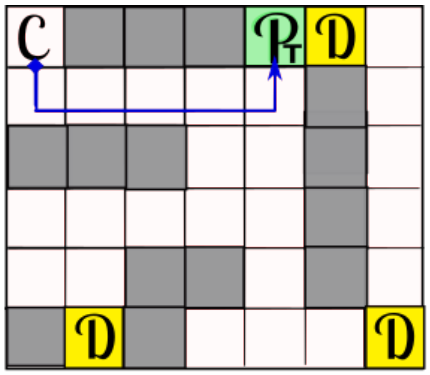


camino 3:(5,6)(5,5)(5,4)(4,4)(3,4)(3,3)(3,2)(3,1)(3,0)
(4,0)(4,1)(5,1)

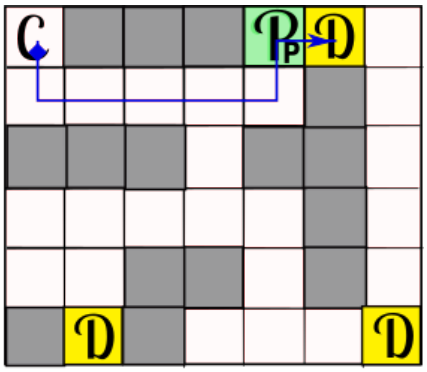


(el camino pasa por una puerta y desde ahi
salta al comienzo)
camino 4:(5,1)(4,1)(4,0)(3,0)(3,1)(3,2)(3,3)(3,4)(4,4)
(5,4)(5,5)(5,6)(4,6)(3,6)(2,6)(1,6)(0,6)(0,5)(0,4)(0,0)

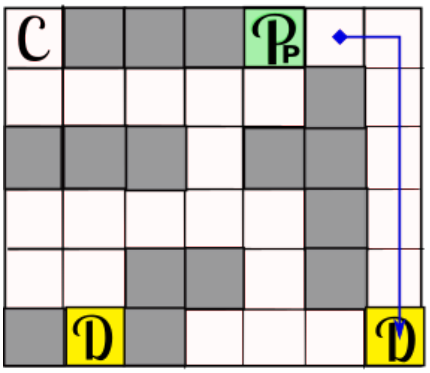
En el tercero hay una trampa que nos obliga a volver al comienzo según lo explicado en el enunciado; una vez se pasa por ella, la propiedad de la puerta pasa a ser 'P', es decir, se puede pasar por ella sin ningún problema:



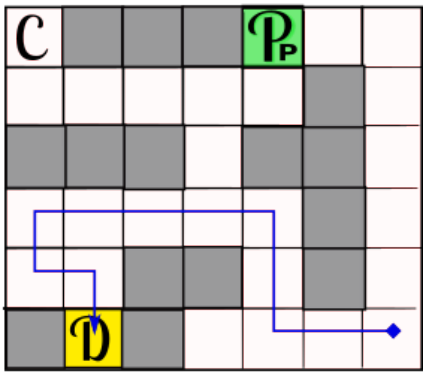
el primer camino es hasta la trampa
añadiendo el comienzo
camino 1:(0,0)(1,0)(1,1)(1,2)(1,3)(1,4)(0,4)(0,0)



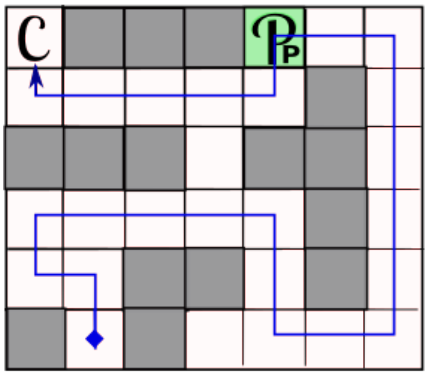
a partir de aqui hay cambio de sentido
camino 2:(0,0)(1,0)(1,1)(1,2)(1,3)(1,4)(0,4)(0,5)



camino 3:(0,5)(0,6)(1,6)(2,6)(3,6)(4,6)(5,6)



camino 4:(5,6)(5,5)(5,4)(4,4)(3,4)(3,3)(3,2)(3,1)(3,0)
(4,0)(4,1)(5,1)



camino 5:(5,1)(4,1)(4,0)(3,0)(3,1)(3,2)(3,3)(3,4)(4,4)
(5,4)(5,5)(5,6)(4,6)(3,6)(2,6)(1,6)(0,6)(0,5)(0,4)(1,4)
(1,3)(1,2)(1,1)(0,0)

Documentación

Documenta en el mismo código (con un comentario antes de cada variable o método) **como mínimo** los siguientes elementos, con una breve descripción (o más extensa, dependiendo de las aclaraciones añadidas en cada método que se describe a continuación):

- las variables y métodos de instancia (o clase) añadidos por tí, justificando su necesidad e incluyendo la palabra "NEW" delante;
- la aplicación con todo detalle;
- los siguientes métodos de instancia:
  - de la clase `Lista`:
    - `public int enLista(ArrayList<Coordenadas>v)`
    - `public void borra(int i)`

Normas generales

Entrega de la práctica:

- Lugar de entrega: servidor de prácticas del DLSI, dirección `http://pracdlsi.dlsi.ua.es`
- Plazo de entrega: desde el lunes 21 de noviembre hasta el **viernes 25 de noviembre**.
- Se debe entregar la práctica en un fichero comprimido con todos los ficheros `.java` creados y ningún directorio de la siguiente manera

```
tar cvfz practica2.tgz *.java
```

- No se admitirán entregas de prácticas por otros medios que no sean a través del servidor de prácticas.
- El usuario y contraseña para entregar prácticas es el mismo que se utiliza en UACloud.
- La práctica se puede entregar varias veces, pero sólo se corregirá la última entregada.
- Los programas deben poder ser compilados sin errores con el compilador de Java existente en la distribución de Linux de los laboratorios de prácticas; si la práctica no se puede compilar su calificación será 0. Se recomienda compilar y probar la práctica con el autocorrector inmediatamente antes de entregarla.

- Los ficheros fuente deber estar adecuadamente documentados usando comentarios en el propio código, sin utilizar en ningún caso acentos ni caracteres especiales.
- Es imprescindible que se respeten estrictamente los formatos de salida indicados ya que la corrección se realizará de forma automática.
- Para evitar errores de compilación debido a codificación de caracteres que **descuenta 0.5** de la nota de la práctica, se debe seguir una de estas dos opciones:
  - Utilizar EXCLUSIVAMENTE caracteres ASCII (el alfabeto inglés) en vuestros ficheros, incluídos los comentarios. Es decir, no poner acentos, ni ñ, ni ç, etcétera.
  - Entrar en el menú de Eclipse Edit ¿Set Encoding, aparecerá una ventana en la que hay que seleccionar UTF-8 como codificación para los caracteres.
- Al principio de cada fichero entregado (primera línea) debe aparecer el DNI y nombre del autor de la práctica (dentro de un comentario) tal como aparece en UACloud (pero sin acentos ni caracteres especiales).

  Ejemplo:
  `DNI 23433224 MUÑOZ PICÓ, ANDRÉS` ⇒ NO
  `DNI 23433224 MUNOZ PICO, ANDRES` ⇒ SI

Sobre la evaluación en general:

- La práctica debe ser un trabajo original de la persona que entrega; en caso de detectarse indicios de copia de una o más entregas se suspenderá la práctica con un 0 a todos los implicados.
- La influencia de la nota de esta práctica sobre la nota final de la asignatura se detallan en las transparencias de presentación de la misma.
- La nota del apartado de documentación supone el 10 % de la nota de la práctica.

Probar la práctica

- En UACloud se publicará un corrector de la práctica con algunas pruebas (se recomienda realizar pruebas más exhaustivas).
- El corrector viene en un archivo comprimido llamado `correctorP2.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar: `tar xfvz correctorP2.tgz`.

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica2-prueba`: dentro de este directorio están los ficheros
    - `p01.pl`: fichero de texto con el plano usado en esta prueba;
    - `p01.java`: programa en Java con un método main que realiza una serie de pruebas sobre la práctica.
    - `p01.txt`: fichero de texto con la salida correcta a las pruebas realizadas en `p01.java`.
- Una vez que tenemos esto, debemos copiar nuestros ficheros fuente (sólo aquellos con extensión .java) al mismo directorio donde está el fichero `corrige.sh`.
- Sólo nos queda ejecutar el *shell-script*. Primero debemos darle permisos de ejecución. Para ello ejecutar:

  ```
  chmod +x corrige.sh
  corrige.sh
  ```