2048

Scala, programación funcional

Contenido

Trabajo realizado

Se han realizado todas las partes obligatorias de la práctica, así como todas las opcionales.
Existen los cuatro modos de juego indicados. En todo momento se evalúa cual es el mejor movimiento, lo cual se hace con una búsqueda de profundidad uno. Se recomendará por tanto realizar dicho movimiento al usuario. También se lleva un registro de la puntuación y de la puntuación acumulada la cual se actualiza cada vez que se pierde una se las 3 vidas.
Para realizar los movimientos se ha creado una única función que implementa el movimiento de derecha a izquierda por lo que para movernos en cualquier otra dirección deberemos rotar el tablero antes de pasarlo por esta función para que este en la posición correcta y luego deshacer la rotación realizada.
La rotación es la parte más costosa de toda la implementación pues para realizarla es necesario crear nuevos elementos adicionales pues la entrada debe ser una matriz de tamaño potencia de dos debido a las características del algoritmo divide y vencerás.
Se ha evitado en todo momento utilizar variables mutables, todas las que se usan son de tipo val. Adicionalmente todo el código se ha creado desde una prospectiva funcional recursiva donde el problema se fragmenta en trozos más simples que se enlazan de modo tal que sean una o varias las funciones aplicadas sobre cada elemento de la matriz hasta alcanzar un punto en el que la recursión finaliza y se devuelve el resultado montándolo de dentro hacia fuera de la pila de llamadas a las funciones.
Para realizar la interfaz se ha creado un trait instanciado mediante dos funciones, un para crear la implementación de la interfaz gráfica y otra para la implementación de la interfaz por consola. Cabe destacar que para hacer compatibles ambas interfaces, en la interfaz gráfica se utiliza una cola síncrona de tamaño uno a la que escriben los listeners asociados a los botones de modo que la ejecución quede bloqueada sin espera activa en la interfaz gráfica hasta que se pulse un botón del mismo modo que lo hace en la consola esperando a que el usuario introduzca algo por ella.

Uso del programa

Al principio se solicitará que se configure el juego, proporcionando el nivel e indicando el tipo de interfaz que se quiere, así como si se desea que los movimientos sean automáticos o no.
Para hacer un movimiento por consola: 1 = LEFT; 2 = RIGHT; 3 = DOWN; 4 = UP

Estilo empleado

Se ha intentado encapsular creando clausuras las funciones que realizan acciones concretas dentro de las funciones que utilizan acciones más generalesZ siempre y cuando una función concreta sea utilizada solo para una acción. Este es el caso de preMover que rota el tablero si es necesario de modo que esté en la posición correcta para ser movido. Dicha función se encapsula dentro de mover pues solo se utiliza ahí. De este modo creemos que queda más claro qué acciones realiza cada función del mismo modo que es más sencillo portar las funciones pues contiene dentro de ellas todas sus auxiliares.
Aunque las funciones encapsuladas pueden utilizar el ámbito de la función que las encapsula esto no se utiliza si no que todo lo que una función utiliza le es pasado por parámetro para de este modo evitar efectos colaterales.

Programación funcional

Durante la realización de la práctica hemos tenido que cambiar nuestra mentalidad para adaptarnos a este nuevo paradigma de programación.
Hemos podido apreciar ciertas cosas que nos han parecido de gran utilidad las cuales pueden aplicarse no solo para programar en este paradigma si no para aplicar en cualquier otro código realizado en un futuro. Adicionalmente hemos observado ciertas limitaciones que dan lugar a casos en los que el paradigma no podría aplicarse al completo.

Ventajas de la programación funcional

La programación funcional intenta reducir los problemas a casos básicos a partir de los cuales poder aportar soluciones a casos mucho más complejos lo cual abre las puertas a una nueva forma de abordar los problemas. No será necesario solucionar un problema de una sola vez siempre que este pueda partirse en otro más sencillos a partir de los cuales se pueda construir una solución.
Adicionalmente dividir un problema en casos más pequeños permite probar cada caso por separado de modo que a cada función implementada y probada podamos garantizar que hemos avanzado un paso más hacia la solución del problema. Intentando resolver todo el problema de una sola vez es mucho más difícil verificar que lo que hemos hecho funciona correctamente de modo que nos es más difícil encontrar fallos en aquello que hemos programado.
Otra de las grandes ventajas apreciada en la programación funcional es la seguridad que aporta debido ala reducción de los efectos colaterales. Si una función solo toma unos parámetros de entrada y produce una salida es más fácil acortar su rango de acción y por tanto encontrar y solucionar fallos que si afecta a variables dispersas por el código que pueden a su vez estarse usando en otras funciones.
Finalmente, y sin duda una de las cosas más atractivas de la programación funcional son las funciones es de primero orden las cuales abren todo un nuevo campo de posibilidades y de nuevas herramientas para solucionar los problemas. Configurar un algoritmo pasándole como parámetro una función anónima, o poder construir nuevas funciones por agregación de otras en tiempo de ejecución son dos nuevas herramientas con gran potencial que sin duda aportan nuevos métodos para resolver problemas de una forma más intuitiva y con menor cantidad de código.

Limitaciones de la programación funcional

La mayor limitación de la programación funcional reside en la sobrecarga que implica el constante uso de la pila el cual es mayor que el coste iterativo.
La recursión adicionalmente excepto en el caso de las funciones en tail recursión estará siempre limitada lo que puede hacer este paradigma inaplicable para resolver ciertos problemas.
Ambas cuestiones combinadas pueden hacer que el paradigma no sea aplicable en ciertos casos, extremos cuando se trabaja a bajo nivele en embebidos con recursos limitados o cuando haya que manejar datos de gran tamaño tales que pudieran saturar la pila de recursión fácilmente.
No obstante, la mentalidad adquirida a la hora de abordar un problema o la seguridad que aporta el paradigma debido a la reducción de los efectos colaterales es digna de ser siempre tenida en cuenta para poderse aplicar siempre que se pueda.

Funciones del juego

Todo el programa ha sido desarrollado con variables inmutables val y se han creado funciones privadas dentro de las funciones como método para organizar el código y encapsular las acciones. Además, se ha evitado el uso de estructuras iterativas optando por la recursividad y estructuras del tipo match-case.

1. def setupJuego()

Menú inicial del juego, donde se indica el modo de juego, automático o manual y el nivel y el tipo de interfaz, gráfica o por consola.

1.1. def gameStrater (nivel : Int, vidas : Int, puntos_totales : Int, window : Interfaz, ia : Boolean) Crea un juego nuevo llamando a gameLoop.

Su función es que gameLoop corte su recursión cada vez que se pierda una vida.

1.2. def createInterface (tablero : List[Int], nivel : Int) : Interfaz

Crea una interfaz para lo que pregunta al usuario si desea que se visual o por consola. Las interfaces son una implementación de un Trait

2. def gameLoop (nivel : Int, vidas : Int, puntos : Int, puntos_totales : Int, tablero : List[Int], window : Interfaz, ia : Boolean, skyp_update : Boolean) : (Int, Int, Boolean)

Bucle principal del juego.

Dentro de este bucle, se encarga de actualizar el estado de juego para lo que llama a mover actualizando el tablero y calculando los puntos, posteriormente actualiza la interfaz y notifica de todo lo necesario al usuario. Finalmente se comprobará si la partida ha terminado y de ser así terminará finalizará su recursión.

2.1. def finJuego (nivel:Int, vidas : Int, puntos : Int, window : Interfaz, ia : Boolean) : (Int, Int, Boolean)

Es llamada cuando el juego ha finalizado y se encarga de manejar este evento actualizando la interfaz, los puntos acumulados y las vidas.

3. def mover (tablero : List [Int], cols : Int, movement : Int) : (List[Int], Int)

Esta función realiza los movimientos sobre el tablero retornando este, así como los puntos a los que equivale el movimiento realizado. Se retornan ambas cosas juntas pues forman parte del mismo proceso y nos ha parecido una mejor forma de encapsular la movilidad que utilizando tres funciones distintas, una para preparar el tablero y hacer la mitad del movimiento, otra que contara los puntos y otra que finalizara el movimiento. Contar los puntos es sumar la matriz auxiliar que se utiliza para mover.

3.1. def preMover (tablero : List[Int], cols : Int, movimiento : Int) : List [Int]

Coloca la matriz según el movimiento que se vaya a realizar, la rota o invierte según sea necesario.

3.2. def postMover (tablero : List[Int], cols : Int, movimiento : Int) : List [Int] = movimiento match

Coloca la matriz según el movimiento que se haya a realizado, la rota o invierte según sea necesario. Es la operación contraria a preMover lo que nos permite hacer los cuatro movientos con el mismo método

3.3. def sumaTablero (tablero : List[Int]) : Int = tablero match {

Suma el contenido de un tablero. Se utiliza para calcular los puntos

4. def rotate90 (tablero : List[Int], cols : Int) : List [Int]

Con esta función se consigue hacer los movimientos de abajo-arriba para lo que es necesario rotar el tablero una o tres veces.

4.1. def log2 (number : Int) : Int

Se utiliza para calcular el tamaño de la matriz de rotación pues para poder rotar una matriz por divide y vencerás es requisito que esta sea de tamaño potencia de dos.

4.2. def _fillZeros (tablero : List[Int], cols : Int, fillFactor : Int) : List[Int] = tablero match {

Añade los ceros necesarios para lograr la mencionada matriz de tamaño una potencia de dos.

4.3. def rotate (tablero : List[Int], cols : Int) : List [Int]

Utiliza el algoritmo de divide y vencerás, para rotar la matriz 90 grados. Si las columnas llegan a 1, entonces devuelve la lista de la cabeza del tablero. En otro caso, divide la lista original en 4 y coge secciones de estas para rotarlas de forma recursiva. Finalmente, añade los pares de forma contigua.

4.4. def rotate90 (tablero : List[Int], cols : Int, rep : Int) : List[Int]

Rota varias veces seguidas utilizando la función anterior.

4.5. def flip (tablero : List[Int], cols : Int) : List [Int]

Equivalente a mover, pero para los movientes izquierda y derecha, invierte el tablero por el eje vertical.

4.6. def seccion (tablero : List[Int], cols : Int, iniciox : Int, finx : Int, inicioy : Int, finy : Int) : List[Int]

Obtiene una sección de una matriz. Es utilizada por rotar para ir dividiendo la matriz en cuartos.

4.7. def tomarFila (tablero : List[Int], cols : Int) : List [Int]

Toma una fila de una matriz

4.8. def join (l1 : List[Int], l2 : List[Int], cols : Int) : List [Int]

Une dos matrices fila a fila, es decir la fila uno con la uno de la otra matriz, la dos con la dos…

5. def crearPiezasNuevas (tablero : List[Int], cols : Int) : List[Int]

Con esta función se crean las piezas que aparecen tras realizar un movimiento.

5.1. def _nuevaPieza (pieza_actual : Int, last_seen : Int, amount_seen : Int) : Int

Comprueba si la pieza actual, es igual a la anterior y divisible entre dos, es decir, que son la misma pieza, entonces multiplica a la última vista por dos.

5.2. def _nextSeen (pieza_actual : Int, last_seen : Int) : Int

Dice que pieza hay que comprobar en la siguiente recursión. Para ello, comprueba primero si es 0, si es así coge la anterior guardada y si no, actualiza y devuelve la nueva.

5.3. def piezasSeguidas (amount_seen : Int, pieza_actual : Int, last_seen : Int) : Int

Comprueba cuantas piezas hay seguidas en la misma fila para no sumar en tríos o de forma desorganizada. Para ello se ayuda de dos funciones auxiliares. La primera comprueba, si la pieza actual es igual a la anterior, si es así devuelve un 1 a la variable de enteros devuelta y si no, devuelve 0. Y la segunda, una vez que se ha identificado que es la misma pieza la actual que la anterior, la borra para no repetirla en la siguiente recursión.

El cuerpo de la función principal es el siguiente: comprueba si el tablero no es nulo, a continuación, llama a la función 6.1. y si se ha llegado al final de la línea del tablero se añade la nueva pieza al tablero y se llama a la función, pero con los parámetros de cantidad y el último visto a 0. Si no es la ultima fila, se hace lo mismo, pero con los parámetros actualizados al siguiente a mirar.

6. def quitarCeros (tablero : List[Int], cols : Int) : List[Int]

Devuelve una lista de valores, donde no hay ceros. Si el tablero está vacío, genera una lista .si nos encontramos en el fin de la fila, añadimos la cabeza del tablero a la lista devuelta por generarlista y la concatenamos con la llamada a la función principal pasando como parámetros la cola del tablero. En cambio, si la cabeza es un 0, no la añadimos y llamamos a la función principal sin la cabeza del tablero. Por último, si no es el fin, la añadimos al tablero y llamamos recursivamente.

7. def mejorMoviento (tablero : List[Int], cols : Int) : Int

Es una función de predicción, donde se calcula cual puede ser el mejor movimiento, para conseguir más puntos. Para ello, analiza todos los posibles movimientos y saca el máximo de puntuación.

8. def isFull (tablero : List[Int], cols : Int) : Boolean

Función que comprueba si el tablero está lleno. Para lo que realiza los 4 movimientos posibles evaluando si el tablero cambia o no.

9. def generarLista(col:Int): List[Int]

Genera una lista con el número de columnas, inicializadas a 0, que se le pase como parámetro. Cuando este llegue a 0 devuelve la nueva lista.

10. def crearTablero (nivel:Int) : List[Int]

La función principal es crear el tablero inicial según el nivel seleccionado, para eso llama a la función de crearlista que devuelve la lista de 0's y luego a la de colocarSemillas que dado un vector de posiciones libres, las colocará en posiciones aleatorias del tablero.

11. def getEmptyPositions (tablero:List[Int]) : List[Int] =

Con esta función, recorremo el tablero de forma recursiva, comprobando que espacios están libres, es decir, a 0, para añadirlos a una nueva lista. Esta función ayuda a colocar las semillas en posiciones que sabemos que no están escritas todavía. Cuando se queda vacío el tablero, se devuelve la nueva lista.

12. def colocarSemillas(tablero:List[Int], semillas:List[Int], nivel: Int): List[Int]

Coloca los nuevos valores aleatorios en el tablero. Hasta que o no queden más huecos libres en él o haya puesto todas las semillas que se le indicaron. Primero obtiene una lista de las posiciones vacías del tablero y luego en ellas pone un número aleatorio perteneciente a otra lista para que se puedan generar números aleatorios distintos dependiendo del nivel.

12.1. def buscarIndice(l:List[Int], posicion: Int) : Int =

Busca en qué posición está la casilla del tablero que está vacía pasada como argumento. Para ello, si el tablero está vacío, devuelve nil, y si no, comprueba que el tablero tiene longitud 1 y devuelve la cabeza y si es más grande lo recorre recursivamente hasta encontrarlo.

12.2. def eliminarIndice(l:List[Int], posicion: Int) : List[Int] =

Es complemento de la función anterior y sirve para eliminar repetidos y que no puedan aparecer posiciones ya escritas anteriormente y previo al cálculo de la nueva lista de posiciones libres.

12.3. def ponerSemilla (l:List[Int], posicion: Int, valor: Int) : List[Int] =

Una vez sacado la posición donde hay un hueco en el tablero, se coloca el valor aleatorio. Para ello, se comprueba que la lista no esté vacía y entonces si la longitud coincide con la posición se añade el valor de la semilla aleatorio al final de la cola de lista. Y si no se vuelve a llamar a la función añadiendo la cabeza, para no perderla y moviendo la posición.

12.4. def _colocarSemillas(tablero:List[Int], semillas:List[Int], nivel: Int, restantes : Int): List[Int] :

Si ya no quedas semillas o restantes se devuelve el tablero tal y como ha quedado, es el caso base. En otro caso, se coge los valores que se pueden insertar, pertenecientes al nivel

seleccionado, se calcula una posición aleatoria libre donde colocar la semilla, un valor aleatorio de los disponibles para el nivel, se elimina la posición escogida para no repetirla en futuras iteraciones. Esto se repite hasta terminar de poner las semillas.

13. def isBetween (input : Int, min : Int, max : Int) : Boolean

Dice si el número pertenece al rango

14. def _inicioFila (length : Int, cols : Int) : Boolean

Dice si el elemento está al inicio de una fila

15. def _finFila (length : Int, cols : Int) : Boolean

Dice si el elemento está al final de una fila

Funciones de formato

Las siguientes funciones, dan formato a lo que se muestra por consola, ya sea imprimiendo la cabecera, dando color a los números o imprimiendo el tablero.

15.1. def printTablero (tablero:List[Int], cols : Int)

Imprime un tablero por terminal con los índices marcando la posición de cada ficha y estando las fichas centradas.

15.2. def _listNumbers (cols : Int) : String

Crea una lista con los números incremetándose desde 1 a cols.

15.3. def_buildString (text : String, ammount : Int) : String

Crea una cadena con el texto repetido ammount veces.

Funciones auxiliares

Son funciones que añaden valores al programa, ya sea escribiendo la puntuación en un archivo o recuperándola.

15.4. Def saveNumber (value : Int, file : String = "./.score")

Guarda un número en un fichero (No llega a utilizarse pues no es necesario)

15.5. Def retrieveNumber (file : String = "./.score") : Int

Lee un número de un fichero(No llega a utilizarse pues no es necesario)

15.6. Def printNumero (text : String ,puntos : Int)

Imprime un número aplicándole formato

15.7. Def getCols (nivel : Int) : Int

Dice las columnas que corresponden al nivel

15.8. Def getCantidadSemillas (nivel : Int) : Int

Dice la cantidad de semillas que corresponden al nivel

15.9. Def getListaSemillas (nivel : Int) : List[Int]

Dice las semillas posibles que corresponden al nivel

15.10. Def movementTxt (movement : Int) = movement

Convierte el número de movimiento a texto UP, LEFT, RIGHT, DOWN

15.11. def getNumber (text : String, min : Int, max : Int) : Int

Recogen los datos de teclado, limitando la entrada de datos a unos específicos según el dato pedido.

Funciones de la interfaz

Define el trait que nos permite controlar la interfaz. Este trait indica los métodos que deberán implementar las clases que quieran ser interfaz del juego. En nuestro caso habrá dos, ambas creadas instanciando dos funciones directamente ya que esto es posible en Scala. Frame represetará a la interfaz gráfica y console a la interfaz por terminal. El juego utilizará llamadas a los métodos del trait para actualizar la interfaz sin importarle si es una u otra.

```
abstract trait Interfaz extends MainFrame{
        def updateContent (tablero : List[Int]);      //MUESTRA EL TABLERO
        def setPoints (points : Int);                 //MUESTRA LOS PUNTOS
        def setAccPoints (acc_points : Int);          //MUESTRA LOS PUNTOS ACUMULADOS
        def setRecomendation (movement : Int);   //DA RECOMENDACIONES DE MOVIENTO AL USUARIO
        def setRecomendation (text : String);         //CAMPO GENERAL
        def setLives (lives : Int);                   //MUESTRA LAS VIDAS
        def getOption (option : String) : Int;        //Retorna una opcion
        def getMovement () : Int;                     //Retorna un moviento realizado
}
```

Ejemplos

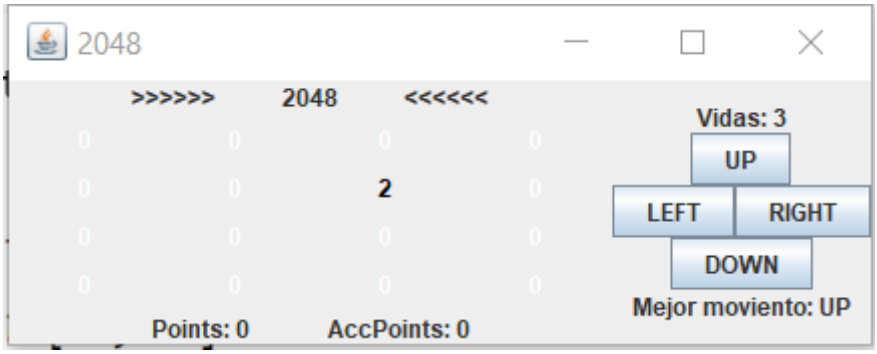
Modo consola.

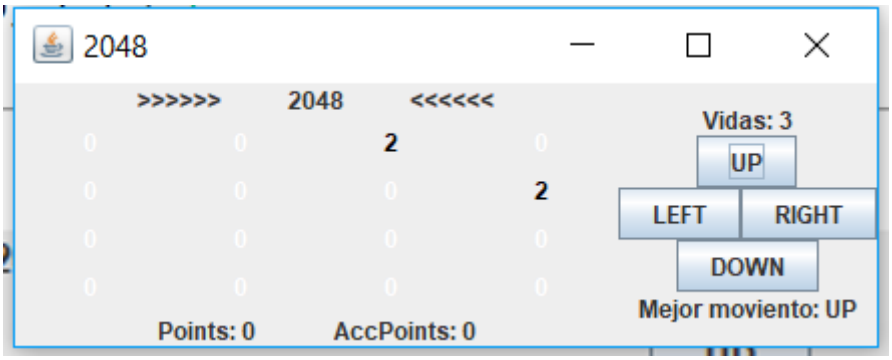
```
>>>>>>          2048          <<<<<<
-------------------------------------------
Seleccione nivel [ 1, 4 ] : 1
Movientos automáticos? [ 0, 1 ] : 0
Interfaz grafica? [ 0, 1 ] : 0
```

```
----------------------
Puntos: 0
----------------------
-------------------------------------------------------------------------------
|              |       1       |       2       |       3       |       4       |
-------------------------------------------------------------------------------
|       1      |       0       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
|       2      |       0       |       0       |       2       |       0       |
-------------------------------------------------------------------------------
|       3      |       0       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
|       4      |       0       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
Mejor moviento: LEFT
Moviento [ 1, 4 ] : 1


----------------------
Puntos: 0
----------------------
-------------------------------------------------------------------------------
|              |       1       |       2       |       3       |       4       |
-------------------------------------------------------------------------------
|       1      |       0       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
|       2      |       2       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
|       3      |       2       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
|       4      |       0       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
Mejor moviento: UP
Moviento [ 1, 4 ] : 2


----------------------
Puntos: 0
----------------------
-------------------------------------------------------------------------------
|              |       1       |       2       |       3       |       4       |
-------------------------------------------------------------------------------
|       1      |       0       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
|       2      |       0       |       0       |       0       |       2       |
-------------------------------------------------------------------------------
|       3      |       0       |       0       |       0       |       2       |
-------------------------------------------------------------------------------
|       4      |       0       |       0       |       2       |       0       |
-------------------------------------------------------------------------------
Mejor moviento: UP
Moviento [ 1, 4 ] : 3


----------------------
Puntos: 4
----------------------
-------------------------------------------------------------------------------
|              |       1       |       2       |       3       |       4       |
-------------------------------------------------------------------------------
|       1      |       0       |       0       |       2       |       4       |
-------------------------------------------------------------------------------
|       2      |       0       |       0       |       2       |       0       |
-------------------------------------------------------------------------------
|       3      |       0       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
|       4      |       0       |       0       |       0       |       0       |
-------------------------------------------------------------------------------
Mejor moviento: UP
Moviento [ 1, 4 ] :
```

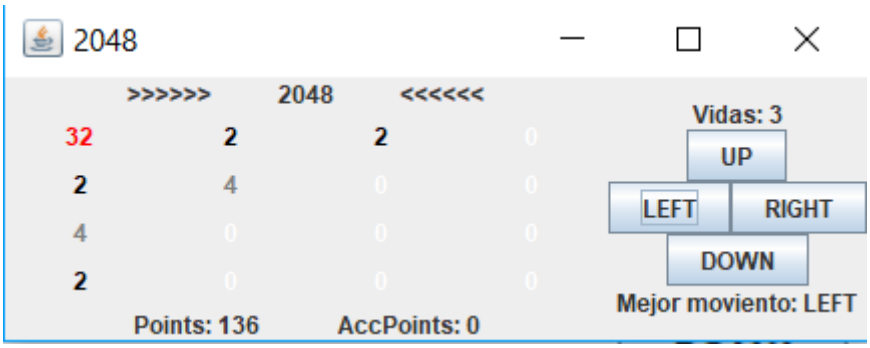
Modo interfaz

```
>>>>>>            2048            <<<<<<
-----------------------------------------
Seleccione nivel [ 1, 4 ] : 1
Movientos automáticos? [ 0, 1 ] : 0
Interfaz grafica? [ 0, 1 ] : 1
```





Varias jugadas después.



Los puntos se acumulan y las vidas se pierden al avanzar en el juego.