16-1-2023

RESOLUCIÓN DE SUDOKU DE MANERA INTERACTIVA

MARIO GARCÍA GONZÁLEZ
ÁLVARO MAYA CANO

UVUS:
margargon44
alvmaycan

ÍNDICE

1. Propuesta de trabajo 2
2. Resumen del trabajo 2
3. Fragmentos de código 3
   3.1. Sudoku.hs 3
   3.2. SudokuAux.hs 8
Ejemplos de sudoku 13
4. Ejecución 14
5. Uso de librerías 19
6. Problemas durante la realización del trabajo 20

1. Propuesta de trabajo

Este proyecto consiste en realizar un sudoku de forma interactiva. Para ello hay dos formas de hacerlo, de forma manual en el que se le va insertando valores a las casillas vacías (en nuestro caso serían el 0) y de forma automática en el cual devuelve el sudoku ya resuelto.

2. Resumen del trabajo

El proyecto lo hemos dividido en dos archivos .hs; sudoku.hs y SudokuAux.hs. En el primer archivo, se encuentran todas las funciones cuales su función principal sea la de interactuar con el usuario, mientras que en SudokuAux.hs encontraremos todas las funciones auxiliares que necesitemos para que sea funcional el sudoku.

También tenemos 3 niveles de dificultad a la hora de realizar ese sudoku: fácil, medio y difícil. Cada nivel está compuesto por el sudoku de inicio y el sudoku ya solucionado.

Para la resolución del sudoku, hemos elaborado unas funciones que escriban un documento .txt que siempre que haya modificaciones se sobreescriba y se muestre en pantalla. De esta forma, mantenemos los originales intactos y únicamente es necesario modificar este último.

A la hora de resolver el sudoku, quisimos que el usuario tuviese mucha información que le pudiera ayudar a la resolución del sudoku, por eso, tenemos algunas opciones para ver las posibilidades en una casilla o que valores faltan en la fila, columna o bloque en el que se encuentra. Todas estas funciones las veremos mejor en el siguiente apartado de fragmentos de código.

Como norma general, el usuario tendrá que teclear el número asignado a la opción que quiera ejecutar para hacer su llamada, y en caso de teclear otra tecla, se ejecutará otra opción que normalmente es una llamada recursiva al punto donde nos encontrábamos previamente.

3. Fragmentos de código

3.1. Sudoku.hs

Las librerías que hemos utilizado son: Data.Array, Data.Char, Data.List, System.IO, System.Directory y SudokuAux (es el módulo que hemos realizado y se verá más adelante).

Los tipos los cuales hemos usado son: Sudoku que es un array con una tupla (de enteros) y un entero, Bloque, Fila, Columna, Valor, todos son enteros y por último Posición que es una tupla de Fila y Columna.

```haskell
import Data.Array
import Data.Char
import Data.List
--import Text.Printf
import System.IO
import System.Directory
import SudokuAux


type Sudoku = Array (Int,Int) Int
type Bloque = Int --Array (Int,Int) Int -- Sudoku más pequeño (3x3)
type Fila = Int
type Columna = Int
type Valor = Int
type Posicion = (Fila,Columna)

main :: IO ()
main = do
    putStrLn "\n ¡Bienvenido a la resolución de sudoku! "
    putStrLn "\n ¿Cual de estos sudokus desea resolver? "
    putStrLn "\t1. Dificultad fácil"
    putStrLn "\t2. Dificultad media "
    putStrLn "\t3. Dificultad dificil"
    putStrLn "\t4. Salir"
    putStr "\nEscriba el número asociado a la selección: "
    o <- getLine
    case o of "1" -> lecturaSudoku "sudoku-easy.txt" "sudoku-easy-sol.txt"
              "2" -> lecturaSudoku "sudoku-medium.txt" "sudoku-medium-sol.txt"
              "3" -> lecturaSudoku "sudoku-hard.txt" "sudoku-hard-sol.txt"
              _ -> main
```

El main está formado por varios putStrLn dando la bienvenida y la elección de dificultad del sudoku y un case of que selecciona esa

dificultad: 1 fácil, 2 medio y 3 difícil. En caso de que no se elija ninguno de estos 3 números vuelve a ejecutar la llamada main.

```haskell
lecturaSudoku::String->String->IO ()
lecturaSudoku sudoku res = do
    putStrLn "Comenzamos la carga del sudoku. \n"
    --la <- nos permite extraer el sudoku del IO Sudoku que nos devuelve leerSudoku
    sudo <- leerSudoku sudoku
    sol  <- leerSudoku res
    escrituraSudoku sudo

    --usamos el readfile en lugar de print sudo para que nos imprima el txt que es mas legible que un array
    impresion <- readFile "escritura.txt"
    putStrLn impresion
    --print sol
    comienzo_resolucion sudo sol

comienzo_resolucion::Sudoku -> Sudoku ->  IO ()
comienzo_resolucion sudoku sol = do
    --preguntamos primero por las distintas opciones de resolver el sudoku: automatica o manual
    putStrLn "\n¿De que forma quiere resolver el sudoku?"
    putStrLn "\t1. Manual"
    putStrLn "\t2. Automatica"
    o <- getLine
    case o of "1" -> resolucion_manual sudoku sol
              "2" -> resolucion_automatica sudoku sol
              _ -> comienzo_resolucion sudoku sol
```

LecturaSudoku consiste en generar los sudokus derivados de los archivos .txt de origen. El sudoku que se vaya a usar para resolver será del que se haga una copia lista para ser modificada y además será el que se muestre por pantalla.

Comienzo_resolucion permite elegir la forma de resolver el sudoku. De forma manual y de forma automática. Al principio teníamos pensado que pudiéramos resolver el sudoku de forma automática, pero era más complicado de lo que creíamos y hemos hecho que se muestre la solución por pantalla.

```haskell
resolucion_manual::Sudoku->Sudoku -> IO()
resolucion_manual sudoku sol = do
    --preguntamos por filas
    putStrLn "\nBienvenido a la resolucion manual del sudoku, continuemos."
    putStrLn ""
    sudo<- readFile "escritura.txt"
    putStrLn sudo
    --FUNCION IMPRIMIR SUDOKU
    putStrLn "\n Indique la fila: "
    fila <-darNumero
    --preguntamos por columnas
    putStrLn "\n Indique la columna: "
    columna <- darNumero
    let posicion = (fila, columna)
    --cases de: insertar numero / posibilidades / comprobar numero
    putStrLn "\n¿Que desea realizar en esta posicion? "
    putStrLn "\t1. Insertar nuevo numero"
    putStrLn "\t2. Ver posibilidades de numero"
    putStrLn "\t3. Comprobar valor"
    putStrLn "\t4. Comprobar sudoku"
    putStrLn "\t5. Mas informacion"
    o <- getLine
    case o of "1" -> insertar_numero sudoku sol posicion
              "2" -> verPosibilidades sudoku sol posicion --habria que hacer un sequence
              "3" -> comprobacionValor sudoku sol posicion
              "4" -> comprobacionSudoku sudoku sol
              "5" -> masInformacion sudoku sol posicion
              _ -> resolucion_manual sudoku sol
```

En resolucion_manual hay que indicarle una fila y una columna para saber en qué posición se le insertará el nuevo valor. A parte de insertar un nuevo valor podemos ver los posibles valores de esa posición, comprobar si ese valor es correcto, comprobar si el sudoku se ha completado correctamente y más información como ver que valores tiene un bloque, una fila o una columna.

```haskell
insertar_numero :: Sudoku->Sudoku -> Posicion->IO ()
insertar_numero sudoku sol posicion = do
    putStrLn "\nNuevo valor: \n"
    valor <- darValor
    let nuevoSudo = actualizaSudoku sudoku posicion valor
    escrituraSudoku nuevoSudo
    resolucion_manual nuevoSudo sol

verPosibilidades::Sudoku->Sudoku->Posicion->IO()
verPosibilidades sudoku sol pos = do
    let posib = posibilidades sudoku pos
    print posib
    resolucion_manual sudoku sol

comprobacionValor::Sudoku->Sudoku->Posicion->IO()
comprobacionValor sudoku sol pos = do
    let bool = comprobarValor sudoku sol pos
    if bool then do
        putStrLn "El valor es correcto"
        resolucion_manual sudoku sol
    else do
        putStrLn "El valor no es correcto, vuelve a intentarlo"
        resolucion_manual sudoku sol


comprobacionSudoku::Sudoku->Sudoku->IO()
comprobacionSudoku sudoku sol = do
    let bool = comprobarSudoku sudoku sol
    if bool then do
        putStrLn "El sudoku es correcto.\n HEMOS TERMINADO EL SUDOKU POR TANTO CERRAMOS LA APLICACION"
        --resolucion_manual sudoku sol
    else do
        putStrLn "El sudoku no es correcto, vuelve a intentarlo"
        resolucion_manual sudoku sol
```

Insertar_numero es una función que actualiza la posición que hayamos seleccionado por el nuevo valor en una copia del sudoku de inicio.

VerPosibilidades es una función que muestra los elementos que no se encuentran ni en la misma fila, ni en la misma columna ni en el mismo bloque, por lo que serían los valores que se le pueden dar en esa posición.

ComprobacionValor y comprobacionSudoku son funciones que sirven para verificar si ese valor está en la posición correcta del sudoku y el comprobar si la copia del sudoku es igual que la solución. Llama a las funciones comprobarValor y comprobarSudoku que se encuentra en el otro módulo.

```haskell
masInformacion::Sudoku->Sudoku->Posicion->IO()
masInformacion sudoku sol (x,y) = do

    putStrLn "\n¿Que informacion desea consultar? "
    putStrLn "\t1. Informacion sobre el bloque"
    putStrLn "\t2. Informacion sobre la fila "
    putStrLn "\t3. Informacion sobre la columna"
    putStrLn "\t4. Atras"
    o <- getLine
    case o of "1" -> valorBloque sudoku sol (x,y)
              "2" -> valorFila sudoku sol (x,y)
              "3" -> valorColumna sudoku sol (x,y)
              _   -> resolucion_manual sudoku sol
    resolucion_manual sudoku sol


valorBloque:: Sudoku->Sudoku->Posicion->IO()
valorBloque sudoku sol (x,y) = do
    let bloque      = numBloque (x,y)
    let listaBloque = [1..9] \\ (valoresEnBloque sudoku bloque)
    let frase = "Los valores que faltan en el bloque " ++ (show bloque) ++ " son: " ++ (show listaBloque)
    putStrLn frase
    resolucion_manual sudoku sol

valorFila:: Sudoku->Sudoku->Posicion->IO()
valorFila sudoku sol (x,y) = do
    let listaFila   = [1..9] \\( valoresEnFila sudoku x )
    let frase = "Los valores que faltan en la fila " ++ (show x) ++ " son: " ++ (show listaFila)
    putStrLn frase
    resolucion_manual sudoku sol
```

```haskell
valorColumna:: Sudoku->Sudoku->Posicion->IO()
valorColumna sudoku sol (x,y) = do
    let listaColumna = [1..9] \\ (valoresEnColumna sudoku y)
    let frase = "Los valores que faltan en la columna " ++ (show y) ++ " son: "++(show listaColumna)
    putStrLn frase
    resolucion_manual sudoku sol

resolucion_automatica :: Sudoku -> Sudoku -> IO()
resolucion_automatica sud sol = do
    escrituraSudoku sol
    impresion <- readFile "escritura.txt"
    putStrLn "\nEsta es la solucion del sudoku: \n"
    putStrLn impresion
```

masInformacion es una función que llama a valorBloque en caso de elegir el número 1, valorFila (2) y valorColumna (3). En otro caso vuelve a la funcion resolucion_manual.

ValorBloque, valorFila y valorColumna son funciones que devuelven una lista con los valores de un bloque, de una fila o de una columna.

Resolucion_automatica es una función que imprime el sudoku ya resuelto.

3.2. SudokuAux.hs

Esta clase será tratada como un módulo que importará las funciones necesarias para el correcto funcionamiento de la clase principal.

```haskell
module SudokuAux
    (actualizaSudoku,
    comprobarValor,
    leerSudoku,
    comprobarSudoku,
    cargaSudoku,
    posibilidades,
    darNumero,
    darValor,
    escrituraSudoku,
    valoresEnBloque, valoresEnColumna, valoresEnFila,
    numBloque
    ) where


import Data.Array
import Data.Char
import Data.List
--import Text.Printf
import System.IO
import System.Directory


type Sudoku = Array (Int,Int) Int
type Bloque = Int -- Sudoku más pequeño (3x3)
type Fila = Int
type Columna = Int
type Valor = Int
type Posicion = (Fila,Columna)

cargaSudoku :: String -> Sudoku
cargaSudoku s = listArray ((1,1),(9,9)) (map read (words s))

actualizaSudoku::Sudoku->Posicion->Valor->Sudoku
actualizaSudoku sudoku pos val = sudoku // [(pos,val)]
```

La función carga sudoku creará un sudoku a partir del texto que se encuentra en los ficheros .txt, de esta forma podremos hacer los cambios necesarios tratándolo como un array.

A su vez, actualiza sudoku, nos permite gracias al método (//) de la librería Data.Array, modificar una posición en concreto del sudoku.

```haskell
comprobarValor :: Sudoku -> Sudoku -> Posicion -> Bool
comprobarValor sudoku sol pos = sudoku!pos == sol!pos

comprobarSudoku :: Sudoku -> Sudoku -> Bool
comprobarSudoku sudoku sol = sudoku == sol

leerSudoku :: String -> IO Sudoku
leerSudoku fichero = do
    existe <- doesFileExist fichero
    contenido <- readFile fichero
    let sudoku = cargaSudoku contenido
    return sudoku

--FUNCION AUX QUE NOS PERMITE PASAR UN SUDOKU A UN STRING PARA POSTERIORMENTE
--ESCRIBIRLO EN UN ARCHIVO
sudoku2Lista::Sudoku->[Int]
--tengo que usar el reverse para que en cambio me permita pasarlo bien
sudoku2Lista sudoku = (elems sudoku)

--mediante listaElementos tendremos cada numero con su espacio listo para pasarlo a un unico string
listaElementos::[Int]->[String]
listaElementos xs = [(show x )++ " " | x<-xs]

--queremos que se produzca un salto de linea cada vez que llegue al ultimo elemento de la columna
--por tanto debemos pasar un acc, el max de la columna que usaremos como mod
--nuestro caso base sera cuando terminemos con la lista
--el acc debe empezar siempre en 1 o en el inicio del sudoku
saltoLinea::[String]->Columna->Int-> String
saltoLinea [] _ _= ""
saltoLinea (x:xs) col acc
--  |acc == (fi*col) = ""
  |acc`mod` col == 0 = x ++"\n"++ saltoLinea xs col (acc+1)
  |otherwise = x ++ saltoLinea xs col (acc+1)
```

Los métodos de comprobarValor y comprobarSudoku, sirven para comprobar el valor de una posición en concreto y el del sudoku que se está realizando respectivamente.

El siguiente conjunto de funciones se usarán en combinación para a partir de un sudoku generar una cadena de texto válida para ser impresa en el fichero .txt que usamos para el sudoku que se modifica.

```haskell
--en esta funcion procedemos a copiar el sudoku en un archivo
escrituraSudoku ::Sudoku-> IO()
escrituraSudoku sudoku = do
  let (_,(fi,col)) = bounds sudoku
  let elementos = sudoku2Lista sudoku
  let stringElementos = listaElementos elementos
  let salto = saltoLinea stringElementos col 1
  writeFile "escritura.txt" salto

enteroValido :: String -> Bool
enteroValido xs = elem xs lista
  where lista =["1","2","3","4","5","6","7","8","9"]


darNumero :: IO Fila
darNumero = do
    fi <- getLine
    if enteroValido fi then do
        let fila = read fi :: Fila
        return fila
    else do
        putStrLn "Numero no valido, vuelve a dar un numero"
        darNumero


valorValido :: String -> Bool
valorValido xs = elem xs lista
  where lista =["0","1","2","3","4","5","6","7","8","9"]
```

Escritura sudoku escribirá como mencionamos anteriormente el fichero que usamos siempre para ser modificado.

DarNumero nos servirá para asegurarnos de que el usuario introduzca una fila o una columna que vaya del 1 al 9 haciendo uso de la función enteroValido.

```haskell
darValor :: IO Valor
darValor = do
    putStr "\n Indique un numero: "
    fi <- getLine
    if valorValido fi then do
        let fila = read fi :: Fila
        return fila
    else do
        putStrLn "Valor no valido, vuelve a dar un valor"
        darValor

posibilidades :: Sudoku -> Posicion -> [Valor]
posibilidades sudoku (x,y) = [ z | z <- [1..9] , notElem z (valoresEnFila sudoku x) ,
notElem z (valoresEnColumna sudoku y), notElem z (valoresEnBloque sudoku (numBloque (x,y)))]

--Devuelve una lista con los valores de cada fila para comprobar si son unicos en cada fila
valoresEnFila :: Sudoku -> Fila -> [Valor]
valoresEnFila sudoku fi = [sudoku!(x,y) | x <- [1..fila], y <- [1..col] , x==fi]
  where (_,(fila,col)) = bounds sudoku

--Devuelve una lista con los valores de cada columna para comprobar si son unicos en cada columna
--el filtro actuara de forma que solo sacaremos los valores del bloque en el que nos encontremos
valoresEnColumna :: Sudoku -> Fila -> [Valor]
valoresEnColumna sudoku co = [sudoku!(x,y) | x <- [1..fila], y <- [1..col] , y==co]
  where (_,(fila,col)) = bounds sudoku

--Devuelve una lista con los valores de cada bloque para comprobar si son unicos en cada bloque
valoresEnBloque :: Sudoku -> Bloque -> [Valor]
valoresEnBloque sudoku bloque = [ sudoku!(x,y) | x <- [1..fila], y <- [1..col], numBloque (x,y) == bloque]
  where (_,(fila,col)) = bounds sudoku
```

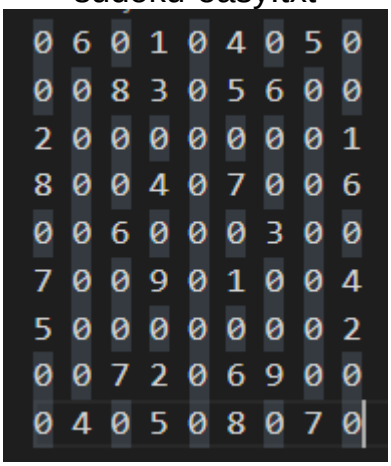
La función de posibilidades será la clave de las interacciones con el usuario en las que nos pregunte por las posibilidades en cierta posición ya que descarta todos los valores que aparezcan en su fila, columna y bloque.

```haskell
-- f calcula el ultimo bloque de la anterior fila y se suma al resultado de c (que divide en 3 ya que cada 3 columnas hay un bloque nuevo).
numBloque :: Posicion -> Bloque
numBloque (x,y)
    | (x >=1 && x<=9) && (y>=1 && y<=9) = f + c
    | otherwise = error "Bloque no valido"
    where f = (x-1) - (mod (x-1) 3)
          c = round $fromIntegral (y+1)/3


--Devuelve una lista con las posiciones vacia para comprobar si una posicion es vacia
posicionVacia :: Sudoku -> [Posicion]
posicionVacia sudoku = [ (x,y) | x <- [1..fila], y <- [1..col], sudoku!(x,y)==0]
  where (_,(fila,col)) = bounds sudoku

--Devuelve si la posicion es vacia
esPosicionVacia :: Sudoku -> Posicion -> Bool
esPosicionVacia sudoku (x,y) = elem (x,y) (posicionVacia sudoku)
```

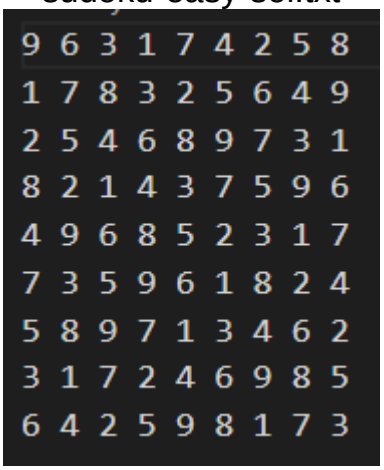
NumBloque nos permite saber en qué bloque nos encontramos según la posición en la que nos encontremos.

Ejemplos de sudoku

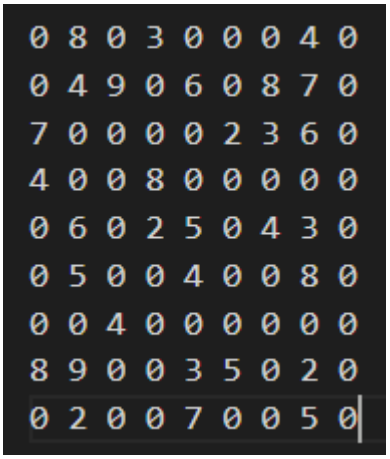
sudoku-easy.txt



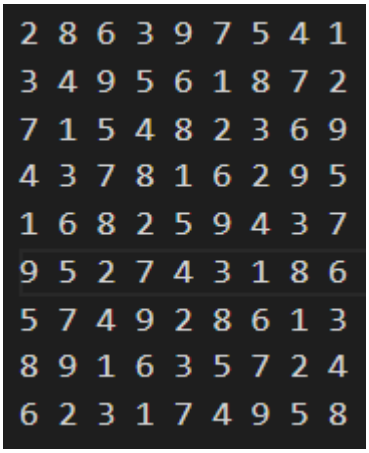
sudoku-easy-sol.txt



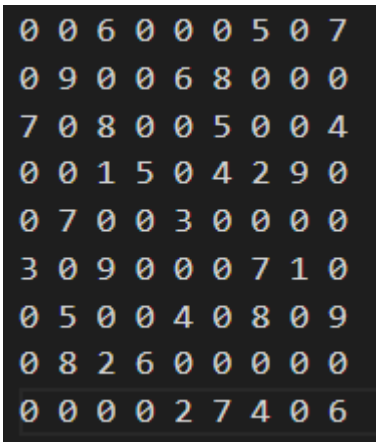
sudoku-medium.txt



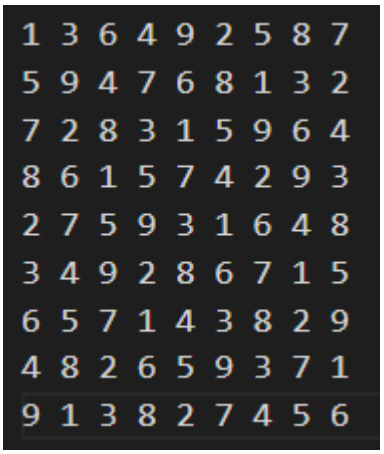
sudoku-medium-sol.txt



sudoku-hard.txt

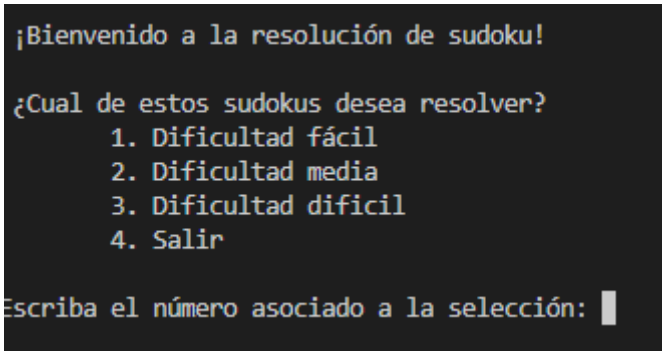


sudoku-hard-sol.txt

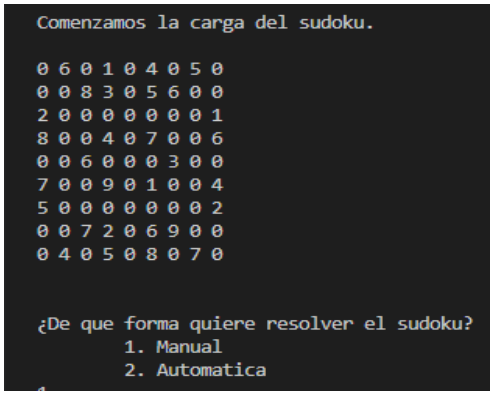


4. Ejecución

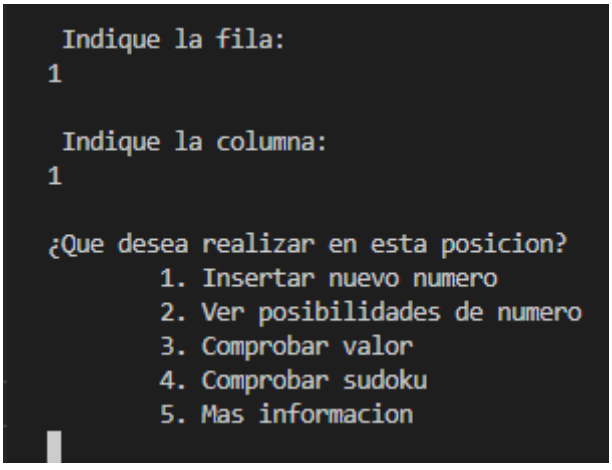
Comenzaremos con la introducción al sudoku preguntando al usuario que dificultad queremos realizar.



En este caso ejecutaremos el sudoku de dificultad fácil, que se mostrará por pantalla y que nos pasará a la siguiente pregunta, ¿queremos resolver el sudoku de forma manual o automática? (la forma automática únicamente nos mostrará el resultado).

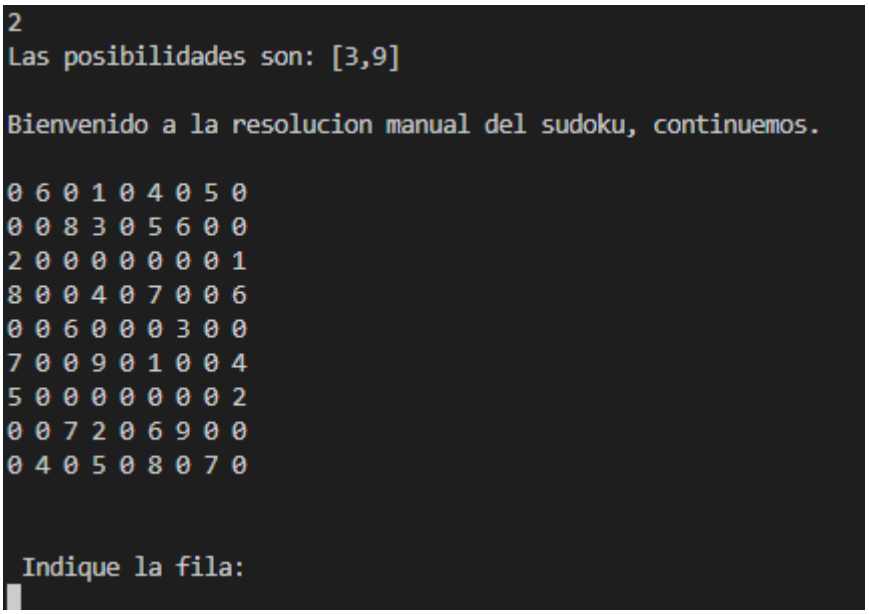


Si continuamos por la resolución manual, tendremos que introducir una fila y una columna (es decir, una posición) sobre la que interactuar.

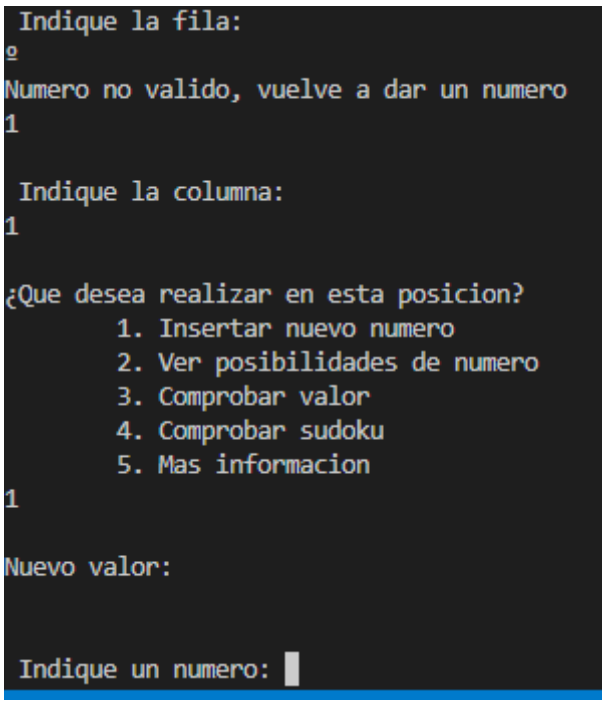


En este caso podemos ver las posibilidades que hay en la posición (1,1) del sudoku.

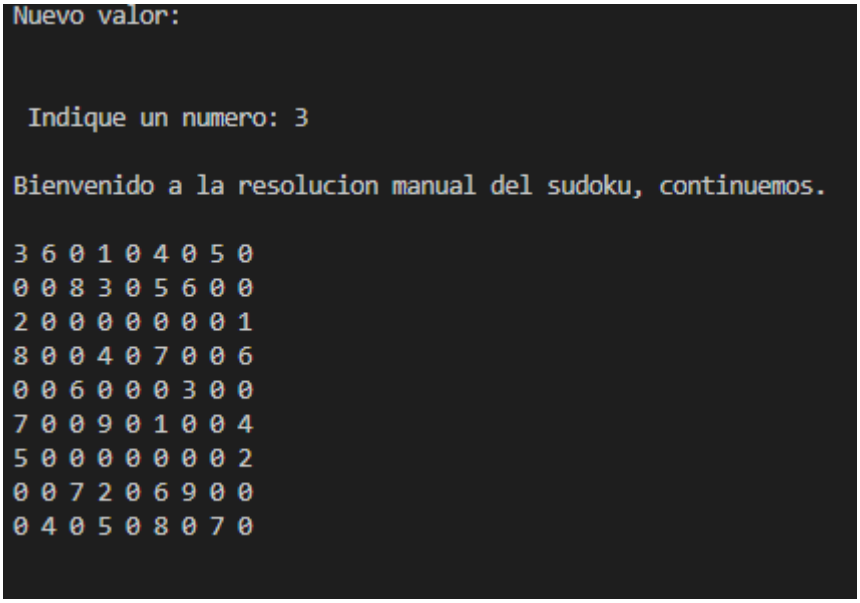
Además, continuaremos con la ejecución del programa para terminar nuestro sudoku.



En el caso de introducir un número no válido en fila o columna, se nos mostrará un mensaje, para continuar como haremos para introducir un nuevo valor.



Como se nos mostró dentro de posibilidades, los números 3 y 9 son los que nos acercarán a nuestra solución. Usaremos en primer lugar el 3.



Si prestamos atención, en la posición (1,1) ahora tenemos el valor 3. Sin embargo, como comprobaremos a continuación, el valor 3 no es el correcto y por tanto procederemos a introducir el valor 9.

```
 Indique la fila:
1

 Indique la columna:
1

¿Que desea realizar en esta posicion?
        1. Insertar nuevo numero
        2. Ver posibilidades de numero
        3. Comprobar valor
        4. Comprobar sudoku
        5. Mas informacion
3
El valor no es correcto, vuelve a intentarlo

Bienvenido a la resolucion manual del sudoku, continuemos.

3 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 0 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0


 Indique la fila:
1

 Indique la columna:
1

¿Que desea realizar en esta posicion?
        1. Insertar nuevo numero
        2. Ver posibilidades de numero
        3. Comprobar valor
        4. Comprobar sudoku
        5. Mas informacion
1

Nuevo valor:


 Indique un numero: 9
```

Nuestro valor debe ser el correcto, y procederemos a comprobarlo.

```
 Indique un numero: 9

Bienvenido a la resolucion manual del sudoku, continuemos.

9 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 0 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0


 Indique la fila:
1

 Indique la columna:
1

¿Que desea realizar en esta posicion?
        1. Insertar nuevo numero
        2. Ver posibilidades de numero
        3. Comprobar valor
        4. Comprobar sudoku
        5. Mas informacion
3
El valor es correcto
```

También podemos consultar otra información interesante como qué valores faltan por colocar en la fila, columna o bloque desde la posición en la que nos encontramos.

```
 Indique la fila:
1

 Indique la columna:
1

¿Que desea realizar en esta posicion?
        1. Insertar nuevo numero
        2. Ver posibilidades de numero
        3. Comprobar valor
        4. Comprobar sudoku
        5. Mas informacion
5

¿Que informacion desea consultar?
        1. Informacion sobre el bloque
        2. Informacion sobre la fila
        3. Informacion sobre la columna
        4. Atras
```

```
1
Los valores que faltan en el bloque 1 son: [1,3,4,5,7]
```

```
2
Los valores que faltan en la fila 1 son: [2,3,7,8]
```

```
3
Los valores que faltan en la columna 1 son: [1,3,4,6]
```

Si queremos ver qué ocurre cuando el sudoku está terminado podemos ejecutar unas líneas que están dentro del archivo comandos.txt.

```
*Main> sudo <- leerSudoku "sudoku-easy-sol.txt"
*Main> sol <- leerSudoku "sudoku-easy-sol.txt"
*Main> comprobarSudoku sudo sol
True
*Main> comprobacionSudoku sudo sol
El sudoku es correcto.
 HEMOS TERMINADO EL SUDOKU POR TANTO CERRAMOS LA APLICACION
*Main>
```

En este archivo encontraremos otros comandos si queremos probar otras funciones ya creadas por nosotros.

5. Uso de librerías

- Data.Array: para todo el tratamiento del tipo sudoku.

- Data.Char: algunos métodos para comprobar si un char era un número o no (terminó sin usarse).
- Data.List: uso principal para un conseguir usar (\\) que nos permite sacar los elementos de la primera lista que no pertenezcan a una segunda lista que se le pasa como parámetro.
- System.IO: librería que nos permite el uso de entrada y salida principalmente.

6. Problemas durante la realización del trabajo

Durante la realización del trabajo uno de los problemas principales a los que nos encontramos fue organizarnos a la hora de cómo queríamos tratar el sudoku, crearlo, modificarlo, guardarlo etc.

Una vez resueltos los problemas organizativos, tuvimos que buscar los métodos adecuados para trabajar con los arrays y además crear una estructura que fuese cómoda y fácil para el usuario.