PAIR PROGRAMMING

PROBLEM STATEMENT : SUDOKO

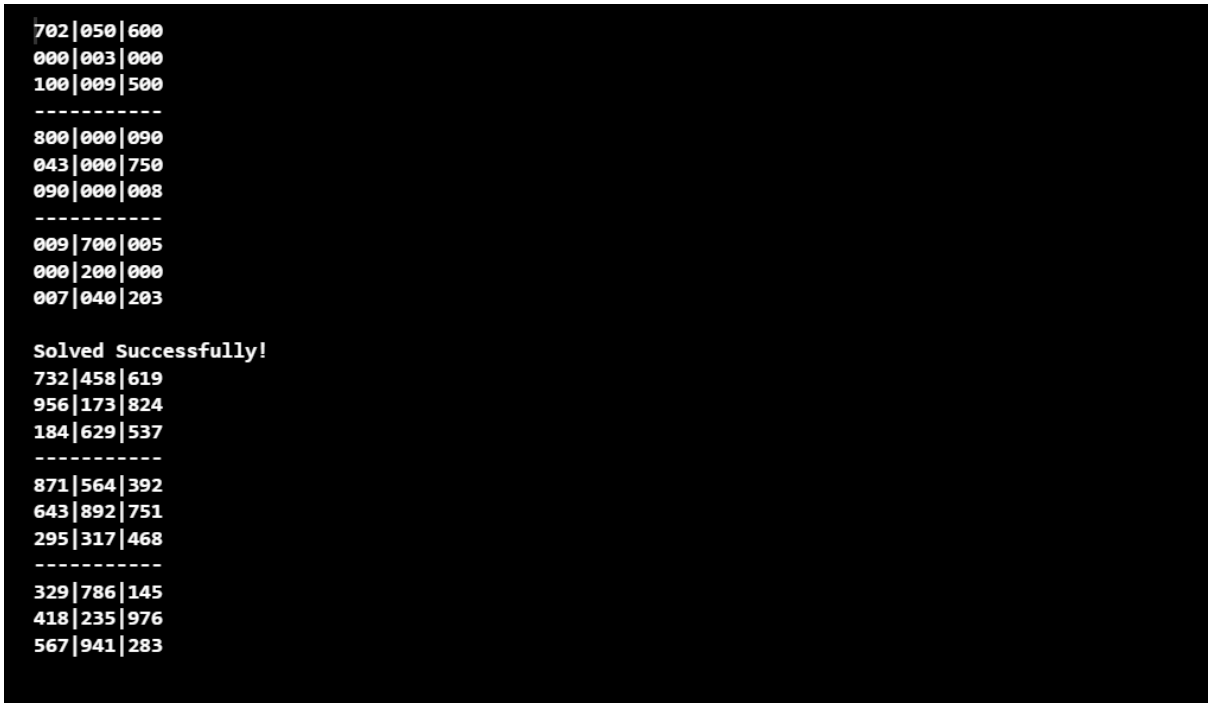
Sud.scala

```
object HelloWorld {
 def isNumberInRow(board:Array[Array[Int]],number:Int,row:Int):Boolean =
 {
  for(i<-0 until 9){
   if(board(row)(i) == number){
    return true
   }
  }
  return false
 }

 def isNumberInColumn(board:Array[Array[Int]],number:Int,column:Int):Boolean =
 {
  for(j<-0 until 9){
   if(board(j)(column) == number){
    return true
   }
  }
  return false
 }

 def isNumberInBox(board:Array[Array[Int]],number:Int,row:Int,column:Int):Boolean =
 {
  var localBoxRow = row - row % 3
  var localBoxColumn =column - column % 3
```

```
  for(m <- localBoxRow until localBoxRow + 3){
    for(n <- localBoxColumn until localBoxColumn + 3){
      if (board(m)(n) == number){
        return true
      }
    }
  }
  return false
}

def isValidPlacement(board:Array[Array[Int]],number:Int,row:Int,column:Int):Boolean =
{
  if(isNumberInRow(board, number, row) == false &&
     isNumberInColumn(board, number, column) == false &&
     isNumberInBox(board, number, row, column) == false){
       return true
     }
  return false
}

def solveBoard(board:Array[Array[Int]]): Boolean =
{
  for(r <- 0 until 9){
    for(c <- 0 until 9){
      if(board(r)(c) == 0){
        for(numberToTry <- 1 to 9){
          if(isValidPlacement(board, numberToTry, r, c) == true){
            board(r)(c) = numberToTry;

            if (solveBoard(board)){
              return true;
```

```
          }
          else{
            board(r)(c) = 0;
          }
        }
       }
       return false
     }
   }
  }
  return true
}

def printBoard(board:Array[Array[Int]]): Unit =
{
  for(a <- 0 until 9){
    if ( a % 3 == 0 && a != 0){
      println("-----------");
    }
    for(b <- 0 until 9){
      if( b % 3 == 0 && b != 0){
        print("|");
      }
      print(board(a)(b))
    }
    println()
  }
}

          def main(args: Array[String]): Unit = {
          var board = Array( Array(7,0,2,0,5,0,6,0,0),
```

```
                    Array(0,0,0,0,0,3,0,0,0),
                    Array(1,0,0,0,0,9,5,0,0),
                    Array(8,0,0,0,0,0,0,9,0),
                    Array(0,4,3,0,0,0,7,5,0),
                    Array(0,9,0,0,0,0,0,0,8),
                    Array(0,0,9,7,0,0,0,0,5),
                    Array(0,0,0,2,0,0,0,0,0),
                    Array(0,0,7,0,4,0,2,0,3));

        printBoard(board);
        println();

        if(solveBoard(board) == true)
  {
     println("Solved Successfully!");
  }
  else
  {
     println("Unsolvable board :(");
  }


         printBoard(board);
        }
}
```



Sud.hs

```
module Main where


import Data.List hiding (lookup)

import Data.Array

import Control.Monad

import Data.Maybe


-- Types

type Digit  = Char

type Square = (Char,Char)

type Unit   = [Square]


-- We represent our grid as an array

type Grid = Array Square [Digit]



-- Setting Up the Problem

rows = "ABCDEFGHI"
```

```
cols = "123456789"
digits = "123456789"
box = (('A','1'),('I','9'))

cross :: String -> String -> [Square]
cross rows cols = [ (r,c) | r <- rows, c <- cols ]

squares :: [Square]
squares = cross rows cols  -- [('A','1'),('A','2'),('A','3'),...]

peers :: Array Square [Square]
peers = array box [(s, set (units!s)) | s <- squares ]
    where
      set = nub . concat

unitlist :: [Unit]
unitlist = [ cross rows [c] | c <- cols ] ++
          [ cross [r] cols | r <- rows ] ++
          [ cross rs cs | rs <- ["ABC","DEF","GHI"],
                    cs <- ["123","456","789"]]


units :: Array Square [Unit]
units = array box [(s, [filter (/= s) u | u <- unitlist, s `elem` u ]) |
                s <- squares]


allPossibilities :: Grid
allPossibilities = array box [ (s,digits) | s <- squares ]

-- Parsing a grid into an Array
```

```
parsegrid     :: String -> Maybe Grid
parsegrid g    = do regularGrid g
                    foldM assign allPossibilities (zip squares g)

  where  regularGrid   :: String -> Maybe String
         regularGrid g  = if all (`elem` "0.-123456789") g
                          then Just g
                          else Nothing

-- Propagating Constraints
assign        :: Grid -> (Square, Digit) -> Maybe Grid
assign g (s,d) = if d `elem` digits
                 -- check that we are assigning a digit and not a '.'
                 then do
                   let ds = g ! s
                       toDump = delete d ds
                   foldM eliminate g (zip (repeat s) toDump)
                 else return g

eliminate     ::  Grid -> (Square, Digit) -> Maybe Grid
eliminate g (s,d) =
  let cell = g ! s in
  if d `notElem` cell then return g -- already eliminated
  -- else d is deleted from s' values
    else do let newCell = delete d cell
                newV = g // [(s,newCell)]
            newV2 <- case newCell of
            -- contradiction : Nothing terminates the computation
                []   -> Nothing
            -- if there is only one value left in s, remove it from peers
                [d'] -> do let peersOfS = peers ! s
```

```
                    foldM eliminate newV (zip peersOfS (repeat d'))
        -- else : return the new grid
            _   -> return newV
        -- Now check the places where d appears in the peers of s
        foldM (locate d) newV2 (units ! s)

locate :: Digit -> Grid -> Unit -> Maybe Grid
locate d g u = case filter ((d `elem`) . (g !)) u of
            []  -> Nothing
            [s] -> assign g (s,d)
            _   -> return g

-- Search
search :: Grid -> Maybe Grid
search g =
  case [(l,(s,xs)) | (s,xs) <- assocs g, let l = length xs, l /= 1] of
        [] -> return g
        ls -> do let (_,(s,ds)) = minimum ls
                 msum [assign g (s,d) >>= search | d <- ds]

solve :: String -> Maybe Grid
solve str = do
   grd <- parsegrid str
   search grd

-- Display solved grid
printGrid :: Grid -> IO ()
printGrid = putStrLn . gridToString

gridToString :: Grid -> String
gridToString g =
```

```
  let l0 = elems g
      -- [("1537"),("4"),...]
      l1 = (map (\s -> " " ++ s ++ " ")) l0
      -- ["1 "," 2 ",...]
      l2 = (map concat . sublist 3) l1
      -- ["1  2  3 "," 4  5  6 ", ...]
      l3 = (sublist 3) l2
      -- [["1  2  3 "," 4  5  6 "," 7  8  9 "],...]
      l4 = (map (concat . intersperse "|")) l3
      -- ["1  2  3 | 4  5  6 | 7  8  9 ",...]
      l5 = (concat . intersperse [line] . sublist 3) l4
  in unlines l5
     where sublist n [] = []
           sublist n xs = ys : sublist n zs
             where (ys,zs) = splitAt n xs
           line = hyphens ++ "+" ++ hyphens ++ "+" ++ hyphens
           hyphens = replicate 9 '-'


main :: IO ()
main = do
    grids <- fmap lines $ readFile "top95.txt"
    mapM_ printGrid $ mapMaybe solve grids
```

```
C:\Users\Akshara S Nair\OneDrive\Desktop\sem 6\PPL\haskel programs\sudoku-norvig>ghci
GHCi, version 9.2.1: https://www.haskell.org/ghc/  :? for help
ghci> :l sudoku.hs
[1 of 1] Compiling Main             ( sudoku.hs, interpreted )
Ok, one module loaded.
ghci> main
 4  1  7 | 3  6  9 | 8  2  5
 6  3  2 | 1  5  8 | 9  4  7
 9  5  8 | 7  2  4 | 3  1  6
---------+---------+---------
 8  2  5 | 4  3  7 | 1  6  9
 7  9  1 | 5  8  6 | 4  3  2
 3  4  6 | 9  1  2 | 7  5  8
---------+---------+---------
 2  8  9 | 6  4  3 | 5  7  1
 5  7  3 | 2  9  1 | 6  8  4
 1  6  4 | 8  7  5 | 2  9  3

 5  2  7 | 3  1  6 | 4  8  9
 8  9  6 | 5  4  2 | 7  3  1
 3  1  4 | 9  8  7 | 5  6  2
---------+---------+---------
 1  7  2 | 4  5  3 | 8  9  6
 6  8  9 | 2  7  1 | 3  5  4
 4  5  3 | 6  9  8 | 2  1  7
---------+---------+---------
 9  4  1 | 8  2  5 | 6  7  3
 7  6  5 | 1  3  4 | 9  2  8
 2  3  8 | 7  6  9 | 1  4  5

 6  1  7 | 4  5  9 | 8  2  3
 2  4  8 | 7  3  6 | 9  1  5
 5  3  9 | 1  2  8 | 4  6  7
---------+---------+---------
 9  8  2 | 5  6  4 | 3  7  1
 3  7  4 | 2  9  1 | 5  8  6
 1  5  6 | 8  7  3 | 2  9  4
---------+---------+---------
 8  2  3 | 6  4  7 | 1  5  9
```