Exercice

Exercice

Programmer un algorithme de résolution de sudoku

Type Sudoku

```
protocol PSudoku : Sequence{

    associatedtype ItSudoku : IteratorProtocol
    associatedtype ItSudokuRow : IteratorProtocol
    associatedtype ItSudokuColumn : IteratorProtocol
    associatedtype ItSudokuRegion : IteratorProtocol
    associatedtype ValuesSet : PSudokuValuesSet

    // initialiser une grille vide

    /// init an empty sudoku grid
    init()

    /// number of values set
    var count : Int { get }

    // remplir une grille avec des valeurs

    /// fill sudoku grid with values given in parameter
    /// - Parameter values: an array of row values, nil means no values
    /// - Precondition: 9 rows and each row must have 9 values, either nil or an int in [1,9]
    /// - Postcondition: a valid sudoku grid else throws an error
    mutating func fillWith(values: [[Int?]]) throws
```

```
/// - Parameters:
///    - value:  value to be put in the grid
///    - at:  position in term of (row, colum) of the value to be changed
/// - Precondition:  1 ≤ value ≤ 9 and 0 <= row, column <= 8 ;
/// there not same value on the row, colum and region
mutating func set(value:Int?, at: (Int,Int)) throws
/// set a value in sudoku grid at a given position
/// - Parameters:
///    - value:  value to be put in the grid
///    - at:  position in term of index of the value to be changed count from left up to right dow
/// - Precondition:  0 ≤ value ≤ 9 and 0 <= index < 81 ;
/// there not same value on the row, colum and region
///
mutating func set(value: Int?, at: Int) throws
/// get value at a given position
/// - Parameters:
///    - at:  position in term of (row, colum) of the value to be changed
/// - Precondition:  0 <= row, column <= 8
/// - Returns:  the value at position given in parameters, nil if no value
func getValue(at: (Int,Int)) -> Int?
/// get value at a given position
/// - Parameters:
///    - at:  position in term of index, count from left up to right down, of the value to be get
/// - Precondition:  0 <= row, column <= 8
/// - Returns:  the value at position given in parameters, nil if no value
func getValue(at: Int) -> Int?
```

Type Sudoku

```swift
 /// true if all values of the grid has been filled
  var isFilled : Bool { get }

  // solve

  /// solve sudoku grid by filling it
  /// - Returns: true if this grid is solvable and a solution has been found
  mutating func solve() -> Bool

  // iterator
  func makeIterator() -> ItSudoku

  func makeItSudoku() -> ItSudoku

  // iterator on row

  func makeItSudoku(row: Int) -> ItSudokuRow

  // iterator on column

  func makeItSudoku(column: Int) -> ItSudokuColumn

  // iterator on region
  func makeItSudoku(region: (Int,Int)) -> ItSudokuRegion
```

Sudoku : solution

```
/// résoudre la grille de sudoku en la remplissant
/// - Returns: vrai si la grille a une solution
///            la grille est alors remplie avec la solution
func solve(grid: SudokuGrid) ⟶ Bool {
  // faire la liste des emplacements à remplir de la grille
  var locs = freeLocations(grid: grid)
  // appel à la fonction récursive qui remplit la grille
  return fillGrid(grid: grid, locs: locs)
}
```

```
func fillGrid(grid: SudokuGrid,
              freeLocs: inout LocationsList) ⟶ Bool{
  si isEmpty(locs) alors return isFilled(grid)
  var solved : Bool = false
  let loc = removeFirst(freeLocs) // récupérer la 1ère
  position
  let possibleValues = freeValues(ofGrid: grid, At: loc)
  var itValues = makeIterator(possibleValues)
  // on va essayer une à une les valeurs possibles
  tantque ((v = next(itValues)) != nil) && !solved do
    set(value: v, at: loc, on: grid) // on essaie la valeur
    // on obtient une nouvelle grille, on la remplit...
    solved = fillGrid(grid: grid, freeLocs: freeLocs)
  fintq
  // si non résolue, la position essayée est à réessayer
  si !solved alors push(freeLocs, loc) }
  return isFilled(grid) // résolue si remplie
}
```