Pseudocódigo:

Función main

```
Input:
    -  file containing sudoku board
    -  n for board size (ex. 9 for 9x9)
Output:
    -  txt containing solution

function main(file, n)
    initialize sudokuBoard as matrix of n*n size

    sudokuBoard = board from file
    if not isValidBoard(sudokuBoard, n)
        return error: "Board not valid"
    else
        if solveSudoku(sudokuBoard, n)
            output to file from sudokuBoard
        else
            return error: “Solution not found”
```

Función isSafe

```
Inputs:
    - T for sudoku table
    - row for row the number is in
    - col for column the number is in
    - num for the number we're validating

Output: boolean value

function isSafe(T, row, col, n, num)
    // n = 9 because our grid is 9x9

    for i to n-1
        if (T[row][i] == num)
            return false

    for i to n-1
        if (T[i][col] == num)
            return false
```

```
    // Check the 3x3 subgrid (of our 9x9 sudoku grid)
    startRow = row - (row % 3)
    startCol = col - (col % 3)
    for i from 0 to 2
        for j from 0 to 2
            if T[startRow + i][startCol + j] == num
                return false

    // if all checks pass, the number is safe to place
    return true
```

Función solveSudoku

```
Inputs:
    -  T for the sudoku table
    -  n for the size of table (9 for out 9x9 table)
Output: boolean if the solution has been found

function solveSudoku(T, n)
    row, column = 0
    isEmpty = true

    // Find the next empty cell (value 0)
    for row from 0 to n-1
        for col from 0 to n-1
            if T[row][col] == 0
                isEmpty = false
                Break from inner loop
        if isEmpty == false
            Break from outer loop

    // if no empty cell is found, the board is solved
    if isEmpty
        return true

    // Try placing numbers from 1 to n
    for num from 1 to n
        if isSafe(T, row, col, n, num)
            T[row][col] = num

            // Recursively try to solve the rest of the board
```

```
            if solveSudoku(T, n)
                return true

            // if placing num didn't work, backtrack
            T[row][col] = 0

    // if no number can be placed, the puzzle is unsolvable
    return false
```

isValidBoard function

```
Inputs:
    -  T for sudoku table
    -  n for number for size of board (9, for a 9x9 board)
Output: boolean if board is valid

function isValidBoard(T, n)
    // Check rows and columns
    for i from 0 to n-1
        Create an empty set for rowSet and colSet
        for j from 0 to n-1
            // Check row uniqueness
            if T[i][j] is not 0
                if T[i][j] exists in rowSet
                    return false
                add T[i][j] to rowSet

            // Check column uniqueness
            if T[j][i] is not 0
                if T[j][i] exists in colSet
                    return false
                add T[j][i] to colSet

    // Check 3x3 subgrids
    for row from 0 to n-1, incrementing by 3
        for col from 0 to n-1, incrementing by 3
            Create an empty set gridSet

            // check the subgrid
```

```
            for i from 0 to 2
                for j from 0 to 2
                    num = T[row + i][col + j]
                    if num is not 0
                        if num exists in gridSet
                            return false
                        add num to gridSet

    // if all checks pass, the board is valid
    return true
```

Análisis de complejidad:

Función isValidBoard

for i from 0 to n-1	O(n)
Create an empty set for rowSet and colSet	O(1)
for j from 0 to n-1	O(n), por anidación: $O(n^2)$
if T[i][j] is not 0	O(1)
if T[i][j] exists in rowSet	O(1)
return false	O(1)
add T[i][j] to rowSet	O(1)
if T[j][i] is not 0	O(1)
if T[j][i] exists in colSet	O(1)
return false	O(1)
add T[j][i] to colSet	O(1)
for row from 0 to n-1, incrementing by 3	$O(\frac{n}{\sqrt{n}}) \Rightarrow O\sqrt{n}$ La complejidad computacional es raíz cuadrada ya que se incrementa por 3 el for loop, esto porque hay 3 subcuadrados en el tablero sudoku 9x9, si fuese 16x16 el número sería 4 (raíz de 16).

for col from 0 to n-1, incrementing by 3	$O\sqrt{n}$, por anidación es O(n)
Create an empty set gridSet	O(1)
for i from 0 to 2	$O\sqrt{n}$, por anidación es $O(n^{\frac{3}{2}})$ Raíz cuadrada ya que cada subcuadrado tiene el tamaño de $O\sqrt{n}$ por $O\sqrt{n}$.
for j from 0 to 2	$O\sqrt{n}$, por anidación es $O(n^{2})$
num = T[row + i][col + j]	O(1)
if num is not 0	O(1)
if num exists in gridSet	O(1)
return false	O(1)
add num to gridSet	O(1)
return true	O(1)

Complejidad: $O(n^2) + O(n^2) =$ **$O(n^2)$**

Función solveSudoku:

row, column = 0	O(1)
isEmpty = true	O(1)
for row from 0 to n-1	O(n)
for col from 0 to n-1	O(n), por anidación: $O(n^2)$
if T[row][col] == 0	O(1)
isEmpty = false	O(1)
Break	O(1)
if isEmpty == false	O(1)
Break	O(1)
if isEmpty	O(1)

return true	O(1)
for num from 1 to n	O(n)
if isSafe(T, row, col, n, num)	O(n) [llamada externa], por anidación: $O(n^2)$
T[row][col] = num	O(1)
if solveSudoku(T, n)	$O(k^n)$ [más explicación al final]
return true	O(1)
T[row][col] = 0	O(1)
return false	O(1)

De acuerdo al artículo *Prune-and-Search | A Complexity Analysis Overview* en *GeeksForGeeks*, los algoritmos recursivos que descartan posibilidades (como lo hacemos con la función isSafe) tienen en el peor caso una complejidad $O(k^n)$. En la cual k es una constante de los valores posibles, 9 en el peor caso. Luego, n es el número de celdas por llenar, en el peor caso 81. Dándonos una complejidad en el peor caso de $O(9^{81})$. Pero, debido a pruning (eliminar valores imposibles) la complejidad se reduce conforme el tiempo avanza.

$O(k^m)$ | k = n , m = n * n, n = longitud (o ancho) del tablero sudoku
También expresado como: $O(n^{n*n})$

Complejidad: $\mathbf{O(n^{n*n})}$

Función isSafe:

for i to n-1	O(n)
if (T[row][i] == num)	O(1)
return false	O(1)
for i to n-1	O(n)
if (T[i][col] == num)	O(1)
return false	O(1)
startRow = row - (row % 3)	O(1)
startCol = col - (col % 3)	O(1)
for i from 0 to 2	$O\sqrt{n}$

for j from 0 to 2	$O\sqrt{n}$, por anidación es O(n)
if T[startRow + i][startCol + j] == num	O(1)
return false	O(1)
return true	O(1)

Complejidad: O(n) + O(n) + O(n) = **O(n)**

Función main:

initialize sudokuBoard as matrix of n*n size	O(1)
sudokuBoard = board from file	O(1)
if NOT isValidBoard(sudokuBoard, n)	$O(n^2)$
return error: "Board not valid"	O(1)
else	O(1)
if solveSudoku(sudokuBoard, n)	$O(k^n) = O(n^{n*n})$
output to file from sudokuBoard	O(1)
else	O(1)
return error: “Solution not found”	O(1)

n = |row| o |column| (9 para sudoku 9x9)

Complejidad total: $\mathbf{O(n^{n*n})}$

Recursos:
https://www.geeksforgeeks.org/prune-and-search-a-complexity-analysis-overview/

530070000
600195000
6 0 .

- Buscar un lugar vacío (0)

5 3 0 ...

- Probar valores

1 → 0

isSafe(1)

5,3,1,0,7,0,0,0,0

1 2

isSafe(1) isSafe(2)

X
ya hay en fila

5,3,1,2,7,0,0,0,0

1 2 3 4

isSafe(1) X

isSafe(2) X fila y subcuadro

isSafe(3) X fila

isSafe(4)

5,3,1,2,7,4,0,0,0