Nombre: Diego Eduardo Garcia Mireles	**Matrícula**: T002994009
Nombre del curso: Estructura de datos	**Nombre del profesor**: FRANCISCO GOMEZ RUBIO
certificado: Estructura de datos	**Actividad**: actividad 3
Fecha: 18/11/2025	
Bibliografía:	

1. Resumen breve

El programa implementa un solver de Sudoku de 9×9 usando backtracking (búsqueda recursiva con retroceso). El tablero se almacena en una matriz int[][] con 0 para casillas vacías. El algoritmo recorre las casillas, intenta valores válidos (1–9) y retrocede cuando una elección causa una contradicción.

2. Código principal: estructura y propósito de cada elemento

A continuación se enumera cada parte del programa y se explica exactamente qué hace y por qué es necesaria.

2.1 static int[][] board

Qué es: la representación del tablero de Sudoku como una matriz de 9 filas por 9 columnas.

Por qué: usar una matriz permite acceder y modificar rápidamente cualquier casilla mediante board[row][col].

Formato: los números 1 a 9 representan celdas con valores fijos o asignados; 0 representa una celda vacía.

Ejemplo (inicial):

{0,0,0,0,4,0,0,0,8},

{4,0,2,0,0,0,0,9,0},

... (y así hasta 9 filas)

2.2 public static void main(String[] args)

Qué hace:

1. Muestra el tablero inicial en consola (llamando a printBoard()).

2. Llama a solve(0) para intentar resolver el Sudoku empezando por la casilla índice 0.

3. Si solve devuelve true, imprime el tablero resuelto; si devuelve false, imprime un mensaje de "No se encontró solución".

Por qué: el main orquesta la ejecución; separa la fase de entrada/visualización de la lógica de resolución.

2.3 static boolean solve(int index)

Qué hace:

- Recibe index, un número entre 0 y 80 que representa la casilla actual (fila*9 + columna). Esto linealiza el recorrido del tablero.

- Si index == 81 indica que ya se procesaron todas las casillas → retorna true (solución completa).

- Calcula la fila (row = index / 9) y la columna (col = index % 9).

- Si la casilla ya tiene un número distinto de 0, avanza a la siguiente casilla (solve(index + 1)).

- Si la casilla está vacía, prueba num desde 1 hasta 9:

  - Llama a isSafe(row, col, num) para comprobar validez.

  - Si isSafe es true, asigna board[row][col] = num y llama recursivamente solve(index + 1).

  - Si la llamada recursiva devuelve true, propaga true hacia arriba (solución encontrada).

  - Si devuelve false, deshace la asignación board[row][col] = 0 y prueba el siguiente número.

- Si ningún número es válido retorna false para forzar retroceso.

Por qué: este método implementa la búsqueda en profundidad con retroceso, que explora combinaciones y elimina (prunes) ramas inválidas.

2.4 static boolean isSafe(int row, int col, int num)

Qué hace:

- Comprueba si num ya está presente en la misma fila (row). Si sí → no es seguro.

- Comprueba si num ya está presente en la misma columna (col). Si sí → no es seguro.

- Calcula la esquina superior izquierda del subcuadro 3×3:

  - startRow = (row / 3) * 3

  - startCol = (col / 3) * 3

  y recorre esas 3 filas y 3 columnas para verificar ausencia de num.

- Devuelve true sólo si num no aparece en fila, columna ni subcuadro.

Por qué: mantener las reglas del Sudoku (no repetir números en fila/columna/cuadro).

2.5 static void printBoard()

Qué hace: imprime el tablero en consola usando . para los ceros y añade divisores cada 3 filas/columnas para facilitar la lectura.

Por qué: hace la salida humana- legible y permite verificar visualmente el resultado.

3. Flujo de ejecución (paso a paso)

1. main imprime el tablero inicial.

2. main llama solve(0).

3. solve recorre celdas en orden lineal (fila 0 col 0 → fila 0 col 1 → ... → fila 8 col 8).

4. Para cada celda vacía intenta valores 1–9 en orden ascendente.

5. Cada asignación válida extiende la búsqueda; cada contradicción provoca retroceso en la pila de llamadas.

6. Si se alcanza index == 81 se ha rellenado todo el tablero sin conflictos: se termina con true.

4. Ejemplo ilustrativo de backtracking (mini-rama)

- Supongamos que en la casilla A (índice 10) se prueba 5 y se asigna porque isSafe devuelve true.

- Más adelante, en la casilla B, ninguno de 1..9 es válido (por restricciones combinadas). solve en B retornará false.

- El programa entonces regresa a la llamada anterior (la que asignó A = 5), restaura A a 0 y prueba el siguiente candidato (6) para A.

- Si algún candidato nuevo en A permite continuar hasta completar el tablero, la solución se encuentra; si no, el algoritmo sigue retrocediendo.

5. Complejidad y comportamiento

- Complejidad en el peor caso: exponencial (O(9^N) donde N es el número de casillas vacías). Sin heurísticas, el solver puede explorar muchas combinaciones.

- Eficiencia práctica: para la mayoría de los Sudokus normales (incluido el provisto), el backtracking simple resuelve el rompecabezas en tiempo razonable.

6. Cómo se obtuvo la solución para este rompecabezas

- El solver comenzó en la primera casilla vacía y fue asignando números válidos según isSafe.

- Cuando una asignación llevó a una contradicción más adelante, se deshizo (backtrack) y se probaron otras opciones.

- Tras explorar las ramas necesarias, se llegó a una asignación completa y consistente para las 81 casillas.

7. Solución final (tablero completo)

Fila por fila:

9 3 6 7 4 2 5 1 8

4 7 2 1 8 5 6 9 3

8 1 5 6 9 3 2 7 4

7 2 8 3 6 1 9 4 5

5 4 9 8 2 7 3 6 1

1 6 3 9 5 4 8 2 7

6 5 7 4 3 9 1 8 2

3 8 1 2 7 6 4 5 9

2 9 4 5 1 8 7 3 6

9. Cómo ejecutar

1. Crear un proyecto Java en NetBeans y pegar el código SudokuSolver en una clase.

2. Ejecutar el proyecto. En la consola aparecerá el tablero inicial y, si hay solución, el tablero resuelto.