Práctica 4. Sudoku problem

Autor: Marcos Hidalgo Baños **Fecha:** 1 de diciembre del 2022

Descripción del problema.

El popular juego Sudoku en su versión tradicional nos presenta ante un tablero parcialmente rellenado, de dimensiones 9x9 y con subtableros de 3x3, en el que tenemos que rellenar cada una de sus celdas con un valor entre 1 y 9 teniendo en cuenta que:

- Un mismo valor solo puede aparecer **una vez por cada fila** del tablero.
- Un mismo valor solo puede aparecer **una vez por cada columna** del tablero.
- Un mismo valor solo puede aparecer **una vez por cada subtablero**.

Código implementado y decisiones de diseño.

En esta práctica se nos encomienda implementar un total de cinco métodos de la clase JAVA **TableroSudoku.java** que serán presentados en este mismo documento. Algunos aspectos generales sobre el desarrollo de estos métodos pueden ser:

➔ Los métodos han sido implementados con la idea presente de optimizar y reducir todo lo posible la complejidad temporal del algoritmo. Esto lo conseguimos evitando el uso de bucles for que recorran cada una de las casillas del tablero (especialmente en la búsqueda de un valor particular) sustituyendolos por bucles while que comprueben el estado de una variable booleana (para salir antes si lo hemos encontrado).

➔ Tal y como se indica en la guía de la práctica, cada vez que en 'resolverTodos' se termine de recorrer la totalidad de casillas vacías y se instancien con un valor, se habrá dado lugar a una nueva solución por lo que tendremos que crear un nuevo objeto 'TableroSudoku'. Esto se debe a que si no lo hacemos, en las posteriores iteraciones también estaremos modificando dicha solución aunque la hayamos introducido en el listado de soluciones.

Método 1. boolean estaEnFila (int fila, int valor);

```java
// Devuelve true si @valor ya esta en la fila @fila.
protected boolean estaEnFila(int fila, int valor) {
    if (fila > FILAS || fila < 0) {
        throw new RuntimeException("Numero de fila invalido");
    }
    int pos = 0;
    while (pos < COLUMNAS && celdas[fila][pos] != valor) {
        pos++;
    }
    return pos < COLUMNAS;
}
```

Método 2. boolean estaEnColumna (int columna, int valor);

```java
// Devuelve true si @valor ya esta en la columna @columna.
protected boolean estaEnColumna(int columna, int valor) {
    if (columna > COLUMNAS || columna < 0) {
        throw new RuntimeException("Numero de columna invalido");
    }
    int pos = 0;
    while (pos < FILAS && celdas[pos][columna] != valor) {
        pos++;
    }
    return pos < FILAS;
}
```

Ambos métodos emplean la misma idea para determinar si un elemento 'valor' está o no en la fila/columna correspondiente, ya que si se encontrara en alguna posición entonces no terminaríamos de iterar sobre todas las posibles filas/columnas.

También se realiza una comprobación previa para determinar si los parámetros proporcionados son correctos para un tablero de Sudoku (dentro de los límites).

Método 3. boolean estaEnSubtablero (int fila, int columna, int valor);

```java
// Devuelve true si @valor ya esta en subtablero al que pertence @fila y @columna.
protected boolean estaEnSubtablero(int fila, int columna, int valor) {
    if (columna > COLUMNAS || fila > FILAS || fila < 0 || columna < 0) {
        throw new RuntimeException("Celda proporcionada invalida");
    }
    boolean esta = false;
    int i = (int) (fila/3)*3, j = (int) (columna/3)*3, f = i, c = j;
    while (i < f+3 && !esta) {
        while (j < c+3 && !esta) {
            esta = celdas[i][j] == valor;
            j++;
        }
        i++;
        j -= 3;
    }
    return esta;
}
```

Debido a que un subtablero en el Sudoku es un recuadro de dimensiones 3x3, deberemos comprobar que en ninguno de los nueve candidatos está el valor en cuestión. Para ello, comenzamos a iterar por la esquina superior izquierda y por filas, realizamos la comprobación para cada una de las columnas (teniendo en cuenta la iteración en la que nos encontramos).

También se vuelve a realizar una comprobación previa para determinar si los parámetros proporcionados son correctos para un tablero de Sudoku (dentro de los límites).

Método 4. boolean sePuedePonerEn(List<TableroSudoku> soluciones, int fila, int columna);

```java
// Devuelve true si se puede colocar el @valor en la @fila y @columna dadas.
protected boolean sePuedePonerEn(int fila, int columna, int valor) {
    return !estaEnFila(fila,valor) && !estaEnColumna(columna,valor) && !estaEnSubtablero(fila,columna,valor);
}
```

Un valor puede colocarse en una fila y columna si no está en la misma fila, columna ni subtablero.

Método 5. void resolverTodos(List<TableroSudoku> soluciones, int fila, int columna);

```
protected void resolverTodos(List<TableroSudoku> soluciones, int fila, int columna) {
    if (estaLibre(fila, columna)) {
        for (int i = 0; i <= MAXVALOR; i++) {
            if (sePuedePonerEn(fila, columna, i)) {
                celdas[fila][columna] = i;
                if (fila == FILAS-1 && columna == COLUMNAS-1) {
                    soluciones.add(new TableroSudoku( uno: this));
                } else if (fila < FILAS-1 && columna == COLUMNAS-1) {
                    resolverTodos(soluciones,  fila: fila+1,  columna: 0);
                } else {
                    resolverTodos(soluciones, fila,  columna: columna+1);
                }
                celdas[fila][columna] = 0;
            }
        }
    } else {
        if (fila == FILAS-1 && columna ==  COLUMNAS-1) {
            soluciones.add(new TableroSudoku( uno: this));
        } else if (fila < FILAS-1 && columna == COLUMNAS-1) {
            resolverTodos(soluciones,  fila: fila+1,  columna: 0);
        } else {
            resolverTodos(soluciones, fila,  columna: columna+1);
        }
    }
}
```

En función de si la casilla está libre, tendremos que o bien comprobar para cada posible valor si se puede poner (y si es así, saber en qué posición lo estamos haciendo para ver si tenemos que seguir probando con otros posibles valores) o bien saltar hasta la siguiente disponible.

Este método es susceptible a ser reescrito de otra manera, o a ser simplificado por otra función que evite la duplicidad del código. No he tomado ese camino por motivos de legibilidad.

Análisis de la complejidad del programa.

La complejidad temporal del algoritmo vendrá dada por las operaciones elementales realizadas. Por este motivo, las principales operaciones implementadas en los métodos ya descritos son:

estaEnFila.

- Comparación lógica para el control de errores → **O (1)**
- Búsqueda del valor por las distintas columnas → **O (n)**

Su complejidad temporal será **O (n)** por ser la operación más costosa, aunque este valor nunca va a ser mayor de nueve por las dimensiones del tablero de Sudoku.

estaEnColumna.

- Comparación lógica para el control de errores → **O (1)**
- Búsqueda del valor por las distintas filas → **O (n)**

Su complejidad temporal será **O (n)** por ser la operación más costosa, aunque este valor nunca va a ser mayor de nueve por las dimensiones del tablero de Sudoku.

estaEnSubtablero.

- Comparación lógica para el control de errores → **O (1)**
- Búsqueda del valor por las casillas del subtablero → **O (n^2)**

La complejidad temporal del método es **O (n^2)** , aunque nunca se va a iterar sobre más de nueve casillas por las propias restricciones del problema (un subtablero es 3x3).

sePuedePonerEn.

Puesto que este método combina los tratados hasta ahora, su complejidad temporal será la mayor de todas ellas, siendo ésta la perteneciente a ‘estaEnSubtablero’.

resolverTodos.

Observación previa. Cuántos más valores iniciales se nos proporcionen, mejor será la complejidad temporal porque evitaremos tener que probar los diferentes candidatos.

- ❖ Comprobar que la casilla está libre → **O (1)**
  - ➢ Probar cada uno de los posibles valores → **O (n)**
    - ■ Comprobar si ese valor se puede poner → **O (n^2)**
    - ■ Avanzar fila/columna o guardar la solución → **O (1)**
- ❖ Si no está libre, avanzar fila/columna o guardar la solución → **O (1)**

Hay que tener en cuenta que el número de valores posibles está limitado por MAXVALOR que en este caso es nueve, por lo que se probarán valores entre uno y este tope.

Por todo esto, la complejidad temporal del algoritmo es **O (n^3)**. Este es un valor muy elevado de complejidad, pero se justifica si sabemos que el problema de la obtención de las soluciones de un Sudoku es uno de los problemas NP-Completos más conocidos.

Con respecto a la complejidad espacial, emplearemos un **listado de soluciones** las cuales serán imágenes (copias) de cada uno de los **tableros** TableroSudoku de dimensiones 9x9. Evidentemente, dependiendo del número de soluciones posibles la longitud del listado variará.

Enlace al repositorio de Github con el código fuente.

github.com/MarkosHB/Analisis-y-diseno-de-Algoritmos/blob/main/Sudoku/TableroSudoku.java

- Marcos Hidalgo Baños -