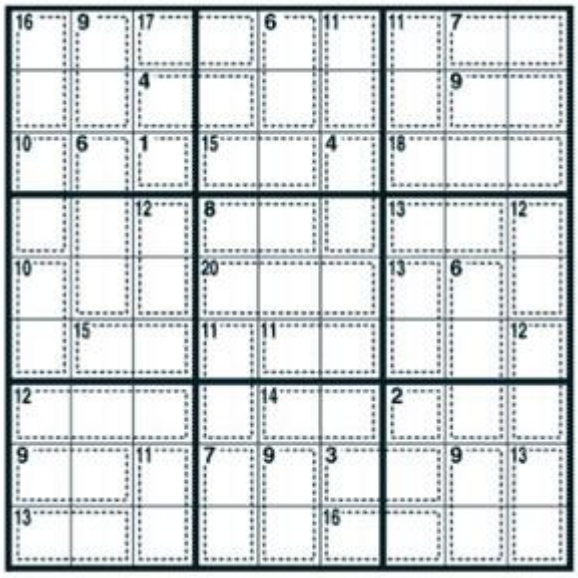
# Reto 2: Sudoku killer

| | |
|---|---|
| Order | 2 |
| Subject | ED |
| Subtipo | Reto |
| Tags | Abstracción TDA |
| Tema | |

## TDA SUDOKU KILLER:



Realizado por José Alberto Hoces Castro y Lorena Cáceres Arias.

## Introducción

En este trabajo vamos a hacer un tipo de dato abstracto para representar un sudoku killer.

Un sudoku killer cualquiera está formado por un tablero de nxn casillas que están agrupadas en una serie de celdas que nos informan de la suma de los valores de dichas casillas. Vamos a intentar hacer una buena representación de este objeto para poder resolverlo.

Este sudoku sigue las mismas normas que los normales, con el añadido de conocer los valores de las sumas de ciertas regiones, a las que en nuestro caso llamaremos celdas. Para resolverlo, primero debemos poder representarlo de forma adecuada, y por ello proponemos esa implementación.

# Representación

Nosotros, la opción por la que hemos optado para su representación es una matriz de vectores, lo que se puede interpretar como una matriz tridimensional. En ella se almacenan los posibles valores que podrían ocupar cada casilla del sudoku, quedando una matriz nxn una vez resuelto (es decir, el vector de posibles soluciones se va actualizando hasta que solo queda una solución posible, por lo que en vez de ser nxnxn sería nxnx1 = nxn). También debemos apoyarnos de un vector de celdas para que quede constancia de los datos de las sumas.

Creemos que esta puede ser una buena representación ya que tener que estar calculando cada vez qué números son posibles para una casilla es muy tedioso, si lo vas almacenando y lo integras ahorramos mucho al no tener que revisarlo cada vez. Las celdas las implementamos como un vector y no como parte de las casillas por comodidad, pues solo se van a usar como una pequeña parte del proceso de comprobar si una casilla admite cierto número.

## Datos auxiliares

Vamos a usar dos Struct auxiliares para que sea más cómodo y comprensible todo lo que vamos a hacer a continuación.

**Coord:**

**int x;**

**int y;**

**Cell:**

**int suma;**

**Vector <Coordenada> v;**

## Representación de la Clase SudokuKiller

Con esto, podemos hacer el sudoku como una matriz de vectores (Vector de Vector de  Vector enteros) y un vector de celdas.

**SudokuKiller:**

**Vector <Vector <Vector <int>>> matrix;**

**Vector <Cell> cells;**

Pre:

- Longitudes de los dos vectores de vectores de matrix tienen que ser valer n que es múltiplo de 3 mayores que 0.
- Los valores de los enteros de la matriz están entre 1 y n
- En el vector de celdas no se pueden repetir coordenadas y deben estar todas.

---

(Reglas del sudoku)

- En el vector de enteros de la matriz (las posibles soluciones) no puede repetir números.
- En cada fila, columna, y bloque de 3x3 (que no se solapan y empezando desde la esquina) no se pueden repetir números
- Se debe respetar que la suma de los enteros de la matriz que están en una celda (los que nos indican las coordenadas de su vector de coordenadas) sea la que indica esa celda.
- Las sumas deben ser plausibles, es decir, que si es un grupo de x casillas, su valor estará entre x y nx. También deben respetar que esa suma se pueda realizar con números válidos, entre 1 y n, siguiendo las reglas anteriores.

## Métodos Privados

- **void Solutions (int x, int y)**:
  - **Brief:** Actualiza las posibles soluciones para una casilla ante la situación actual del tablero

    *La idea es que solutions pueda llamar a check con todos los numeros posibles y anotar los correctos.*

- **Param x:** Coordenada x de la casilla.
- **Param y:** Coordenada y de la casilla.
- **Pre:** 0≤x,y<n.
- **Post:** El objeto se modifica.

- **bool Check( int x , int y , int val)**:
  - **Brief:** Dado una casilla y un número, comprueba si es correcto, teniendo en cuenta la situación actual del sudoku.
    - *Este método se puede implementar comprobando una serie de técnicas que surgen de las reglas del sudoku:*
      - *Comprobar suma (De la celda)*
      - *Comprobar suma, pero solo dentro del bloque (todos suman 45)*
      - *Comprobar fila y columna (Solo puede haber un número de cada)*
      - *Comprobar Bloque (Solo puede haber un número de cada)*
  - **Param x:** Coordenada x de la casilla.
  - **Param y:** Coordenada y de la casilla.
  - **Pre:** 0≤x,y<n , 1≥val≤n.
  - **Return:** True si ese valor es válido y False en caso contrario.
  - **Post:** El objeto no se modifica.

## Métodos Públicos

- **void Solve()**:
  - **Brief:** Resuelve el sudoku, dejando todos los vectores de enteros a dimensión 1, con su solución.
    - *Una forma de hacerlo sería que esto fuese un método iterativo que con ayuda de solutions vaya descartando posibilidades hasta tener el sudoku resuelto.*
  - **Pre:** Que todas las casillas cumplan las reglas del sudoku.
  - **Post:** El objeto se modifica, y queda una matriz nxnx1.
- **Vector<Vector<int>> & Solution (const & SudokuKiller sudoku) const:**

- **Pre:** Se debe haber usado el método solve sobre sudoku antes de usar este método. De lo contrario, devolverá la matriz que se haya construido al usar el constructor.

---

### Constructors:

---

- **SudokuKiller(const Vector<cell> &  cells, int n):**
  - **Brief:** Es el constructor a partir de un vector de celdas (existen sudokus en los que no se da ningún valor de las casillas y solo se especifican las celdas) y la dimensión, ya que aunque se puede determinar buscando el máximo de las coordenadas del vector de coordenadas de cada celda, es más cómodo tener la dimensión de antemano para ahorrar en tiempo.
  - **Param n:** Dimensión de la matriz, la cual se inicializa todo a 0.
  - **Param cells:** Vector de celdas, teniendo cada una la suma de las casillas que le corresponden y un vector de coordenadas con las casillas que le corresponden.
  - **Pre:**
    - La dimensión n debe ser $n \geq 3$ y $n\%3 = 0$
    - El vector de celdas tiene que tener sentido con la matriz. Es decir, que solo aparezca una coordenada cada vez, que estén todas y no haya repeticiones entre las casillas que les corresponden a distintas celdas.
- **SudokuKiller(const Vector<Vector <int>> & m, const Vector<cell> &  cells)**
  - **Brief:** Es el constructor a partir de una matriz con los valores que nos dan de pista y el vector de celdas
  - **Param v:** Matriz (Vector de vectores de enteros) con los números que vienen resueltos. Esta matriz lleva 0 donde no nos dan la solución.
  - **Param cells:** Vector de celdas.
  - **Pre:**
    - La matriz m tiene que ser cuadrada (dimensión nxn), con n múltiplo de 3.
    - Los números colocados, es decir, los distintos a 0, tienen que respetar las reglas del sudoku (ver arriba).

    - El vector de celdas tiene que tener sentido con la matriz. Es decir, que solo aparezca una coordenada cada vez, que estén todas y no haya repeticiones entre las casillas que les corresponden a distintas celdas.

---

### Getters:

---

- **Celda & GetCell (int x, int y) const**:
    - **Brief:** Dadas unas coordenadas nos dice la celda a la que pertenece esa casilla
    - **Param x:** Coordenada x de la casilla.
    - **Param y:** Coordenada y de la casilla.
    - **Pre:** 0≤x,y<n.
    - **Return:** Referencia constante a la celda
    - **Post:** El objeto se modifica.

---

- **int operator[ ][ ] (int x, int y) const**:
    - **Brief:** Dadas las coordenadas, nos devuelve el valor de la casilla con dichas coordenadas.
    - **Param x:** Coordenada x de la casilla.
    - **Param y:** Coordenada y de la casilla.
    - **Pre:** 0≤x,y<n.
    - **Return:** Solo nos devuelve el valor correcto si el método privado solutions ha actualizado las soluciones de tal forma que solo queda un posible valor. Si hay más de un valor, nos devuelve 0.
    - **Post:** El objeto no se modifica.
- **Vector<int> PossibleValues (int x, int y) const:**
    - **Brief:** Dadas las coordenadas, nos devuelve el vector que alberga las posibles soluciones de una casilla (dicho vector es el que está más profundo dentro de nuestra estructura matricial).
    - **Param x:** Coordenada x de la casilla.
    - **Param y:** Coordenada y de la casilla.
    - **Pre:** 0≤x,y<n.

- **Return:** El vector con las posibles soluciones
- **Post:** El objeto no se modifica.

#### Setters:

*No incluimos métodos que modifiquen la clase ya que está planteada para que tras introducir el planteamiento del problema la clase se encargue de resolverlo y dar la solución, no para ser interactivo. Si quisiéramos algo de ese estilo habría que añadirlos, pero no es el objetivo que al que creemos que tenemos que enfrentarnos.*

### Funciones amigas:

- **std::ostream& operator<<(std::ostream& flujo, const SudokuKiller& sudoku)**
  - **Brief:** El operador de flujo para poder imprimir el sudoku. Lo debe hacer como una matriz en la cual en cada casilla en lugar de un número aparece una lista separada por

    1/2/3 | 1/2/3 | 3/6/4

    5 | 9 | 3/4/7

    1/2/8 | 1/2/8 | 3/6/7

  - **Param flujo:** El flujo actual
  - **Param sudoku:** Sudoku a imprimir
  - **Return:** El flujo que entró

# Resumen

### Sugerencia de entrada:

La clase está pensada para que se haga una entrada recopilando los datos, se forme un vector de celdas apropiado y/o una matriz adecuada, y a partir de eso se cree un objeto SudokuKiller el cual se puede resolver a sí mismo cuando haga falta.

### Conclusión:

Vemos cómo sería una posible representación de una clase que resuelva el SudokuKiller. Esperamos haber podido comunicar el porqué de las decisiones que

tomamos con respecto al diseño de la clase y sus respectivos métodos.