

$$\begin{bmatrix} M & A & \dots & T \\ \vdots & \vdots & \searrow & \vdots \\ R & I & \dots & X \end{bmatrix}$$

Índice.

1. Funciones para la compatibilidad de las matrices.	4
1.1 Función de comprobación del tamaño de dos matrices.	4
Diagrama de flujo de la función.	4
1.2 Función para comprobar si una matriz es cuadrada.	4
Diagrama de flujo de la función.	5
1.3 Función para comprobar si la array de entrada es una matriz válida.	6
Diagrama de flujo de la función.	6
1.4 Función que crea una matriz identidad.	7
Diagrama de flujo de la función.	7
2. Suma y resta de matrices.	8
Diagrama de flujo de la función.	8
3. Trazada de una matriz cuadrada.	10
Diagrama de flujo de la función.	11
4. Multiplicación de dos matrices.	12
Diagrama de flujo de la función.	13
5. Multiplicación de una matriz por un número.	14
Diagrama de flujo de la función.	15
6. Potencia de una matriz.	16
Diagrama de flujo de la función.	17
7. Calcular el determinante de una matriz.	18
7.1. Definir una matriz inferior, función getMinor.	18
Diagrama de flujo de la función.	19
Diagrama de flujo de la función.	21
8. Calcular una sub-matriz.	22
Diagrama de flujo de la función.	22
9. Cálculo de una matriz transpuesta.	23
Diagrama de flujo de la función.	23
10. Calcular la inversa de una matriz.	24
Diagrama de flujo de la función.	25
11. División de matrices.	26
Diagrama de flujo de la función.	27
12. Test de la ortogonalidad de una matriz.	28
Diagrama de flujo de la función.	29
13. Resolución de ecuaciones mediante la regla de cramer.	30
Diagrama de flujo de la función.	31

1. Funciones para la compatibilidad de las matrices.

En este apartado estarán todas las funciones que se usarán en todo el código para prevenir posibles errores en las matrices, estas funciones no pertenecen al esqueleto del proyecto, las he añadido ya que se usan en varias partes, así nos ahorramos repetir código, por ejemplo código para comprobar que son cuadradas o que todos los valores se han definido.

1.1 Función de comprobación del tamaño de dos matrices.

Esta función se encargará de recorrer dos matrices de entrada y comparar cada columna y la cantidad de filas con la otra, si las dos son del mismo tamaño devolverá true, si no false. Para llamarla usaremos **if (!matrizMismoTamano(mat1, mat2))** si devuelve false podemos hacer que haga un **return null** si es para matrices o **Double.NaN** para double.

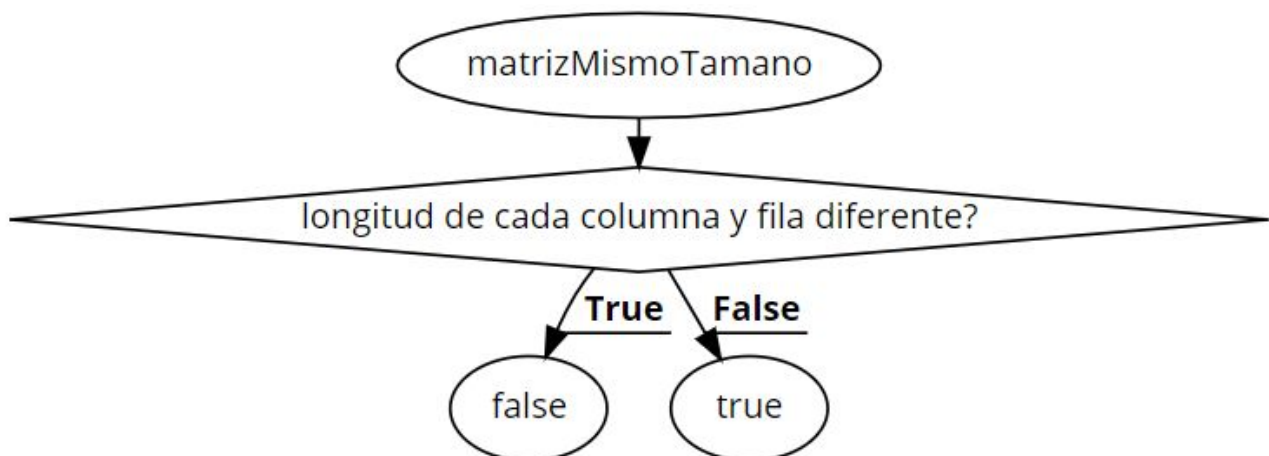
Esta función devolverá un tipo boolean con el resultado de la comparación de las dos matrices. Esta función está compuesta por un **for** el cual va incrementando la variable **contador "i"** para ir incrementando la posición que se compara.

Si alguna posición no concuerda devolverá un **false**, si no se cumple ningún caso al acabar el for devolverá **true**.

```
private static boolean matrizMismoTamano(double[][] mat1, double[][] mat2) {
    for (int i = 0; i < mat1.length; i++) {
        if ((mat1[i].length != mat2[i].length) || (mat1.length != mat2.length)) return false;
    }
    return true;
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **matrizMismoTamano**.



1.2 Función para comprobar si una matriz es cuadrada.

Esta función de tipo boolean se encarga de comprobar si una matriz es cuadrada, lo que quiere decir que todas sus filas y columnas tienen que ser de la misma longitud. Para llamarla usaremos **if (!matrizEsCuadrada(mat)) return null;**

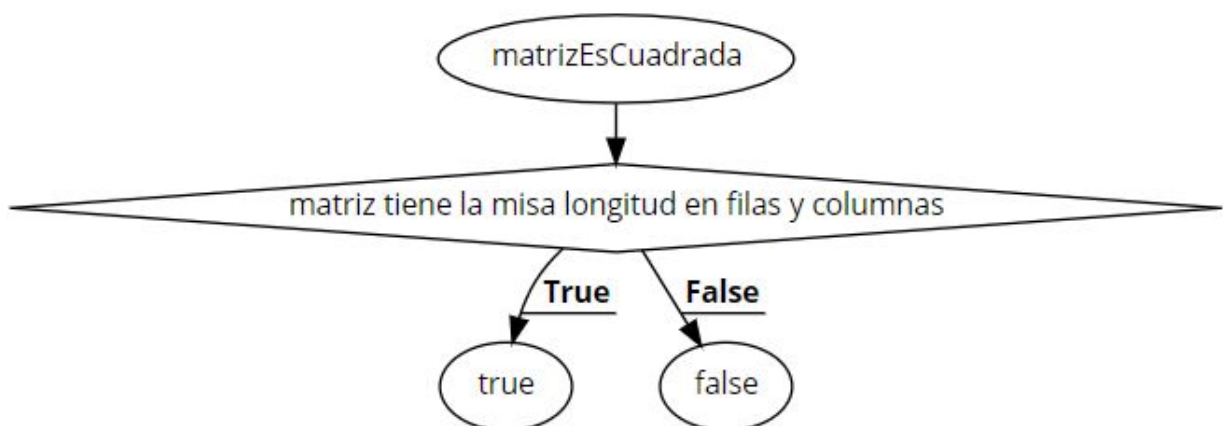
Para comprobar si una matriz cumple con esta característica tenemos que recorrerla comparando sus longitudes, en este caso usaremos un **for**, dentro de este for compararemos la cantidad de filas con la cantidad de columnas (`mat.length != mat[i].length`) y además previniendo que se puede dar el caso de que se introduzca una matriz con todas las filas y columnas iguales menos una de ellas vamos comprobando las columnas de la primera con todas las otras mediante la variable **"i"** (`mat[0].length != mat[i].length`).

Si en alguna de las comparaciones detecta que no coincide la cantidad devolverá un **false**, si no, al acabar el bucle devolverá **true**.

```
private static boolean matrizEsCuadrada(double[][] mat) {
    for (int i = 0; i < mat.length; i++) {
        if ((mat.length != mat[i].length) || (mat[0].length != mat[i].length)) return false;
    }
    return true;
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa a la función **matrizEsCuadrada**.



1.3 Función para comprobar si la array de entrada es una matriz válida.

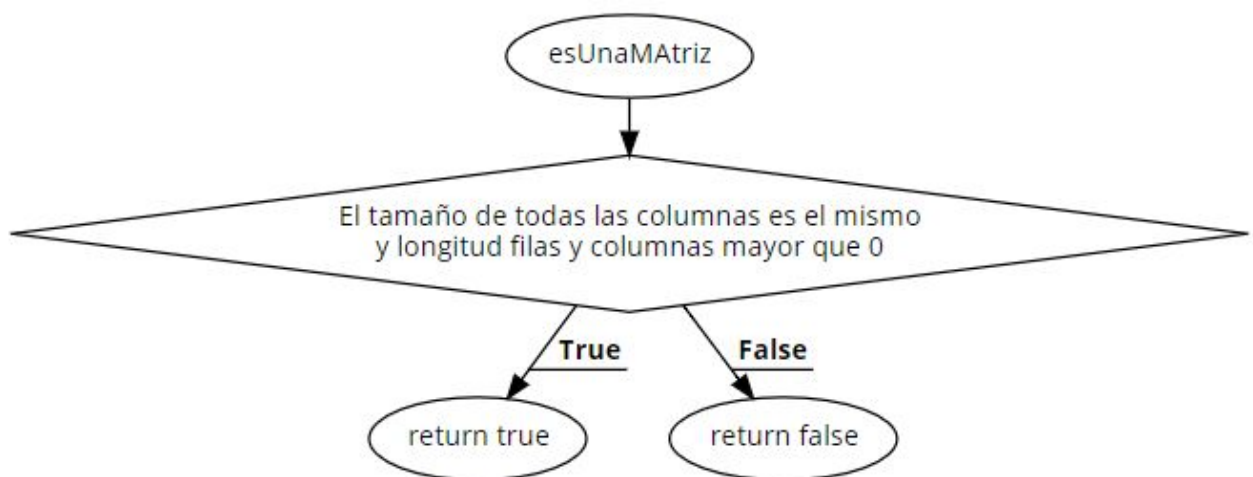
Esta función de tipo boolean comprueba que una array pueda ser llamada una matriz como tal, para esto tiene que cumplir dos condiciones. Que la longitud de todas las columnas sea la misma en todas las filas `if (mat[0].length != mat[i].length)`, y que la matriz no tenga una longitud de filas ni columnas de 0, `(mat[0].length == 0) || (mat[i].length == 0)`.

Si alguna de estas condiciones se cumple hará **return false**. Si no cumple ninguna de las condiciones hará un **return true**. Para llamar a esta función usaremos **if (!esUnaMatriz(mat)) return null;**

```
private static boolean esUnaMatriz(double[][] mat) {
    if (mat.length == 0) return false;
    for (int i = 0; i < mat.length; i++) {
        if ((mat[0].length != mat[i].length) || (mat[i].length == 0)) return false;
    }
    return true;
}
```

Diagrama de flujo de la función.

Este es el diagrama de flujo de la función **esUnaMatriz**.



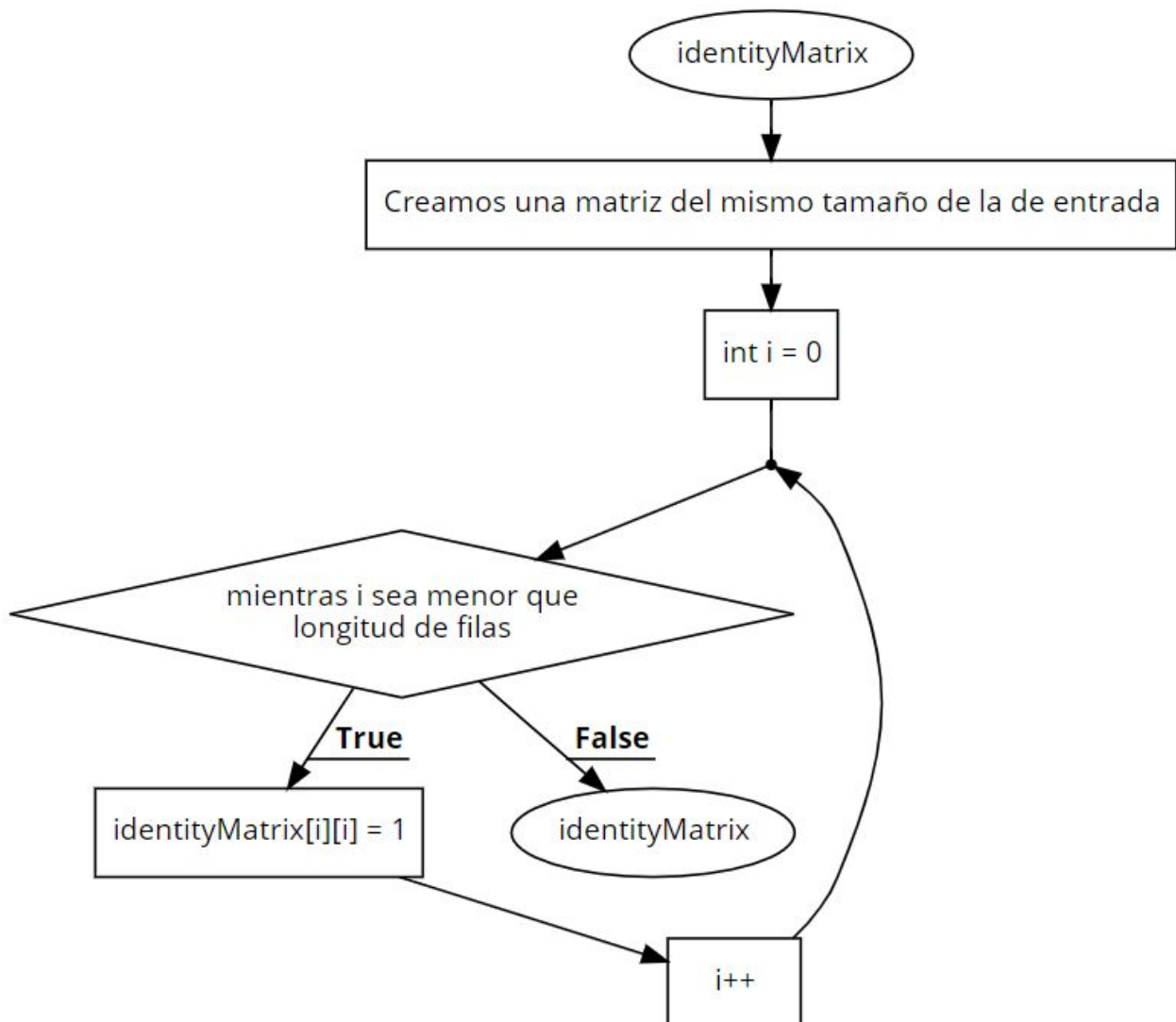
1.4 Función que crea una matriz identidad.

Esta función se encarga de que cuando se le pasa una matriz por parámetro, crear una matriz del mismo tamaño y recorrerla diagonalmente poniendo el número 1 en estas posiciones, luego devolverá la matriz identidad. Esto se usa en diferentes funciones, ya sea para comparar resultados o hacer cálculos.

$$I_1 = [1], I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \dots, I_n = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}.$$

Diagrama de flujo de la función.

Este diagrama representa la función **identityMatrix**.



2. Suma y resta de matrices.

En la suma tenemos que sumar cada posición de la matriz con la posición correspondiente de la otra matriz, una vez hecho, el resultado lo guardamos en una matriz de salida, la cual será devuelta por la función.

	Matriz A				Matriz B				Proceso				Matriz C		
	0	1	2												
0	1	2	2		2	2	2	=	1+2	2+2	2+2	=	3	4	4
1	1	2	3	+	1	1	1	=	1+1	2+3	3+1	=	2	3	4
2	2	1	2		4	4	3		2+4	1+4	2+3		6	5	5

Esta función devuelve una array bidimensional de tipo double y se le introducen por parámetro dos arrays bidimensionales (mat1 y mat2).

Lo primero es hacer las comprobaciones correspondientes a la operación. Para poder hacer una suma las matrices, para la suma tenemos que asegurarnos que las dos arrays sean matrices, para esto usaremos la función [esUnaMatriz](#), también tenemos que comprobar que las dos matrices sean iguales en tamaño, con la función [matrizMismoTamano](#). Si alguna de estas comprueba

La matriz donde se guardará la solución tiene el tamaño de las matrices de entrada ya que tienen que ser iguales.

Para hacer la suma necesitamos un for dentro de otro para recorrer las tres matrices las dos de entrada y guardar el resultado de la suma en la matriz solución, con los contadores **i** y **j** seleccionamos las posiciones, la **i** incrementa tantas veces como longitud de la columna.

```
static double[][] add(double[][] mat1, double[][] mat2) {

    //Comprueba que ambas matrices tengan el mismo tamaño.
    if (!matrizMismoTamano(mat1,mat2)) return null;

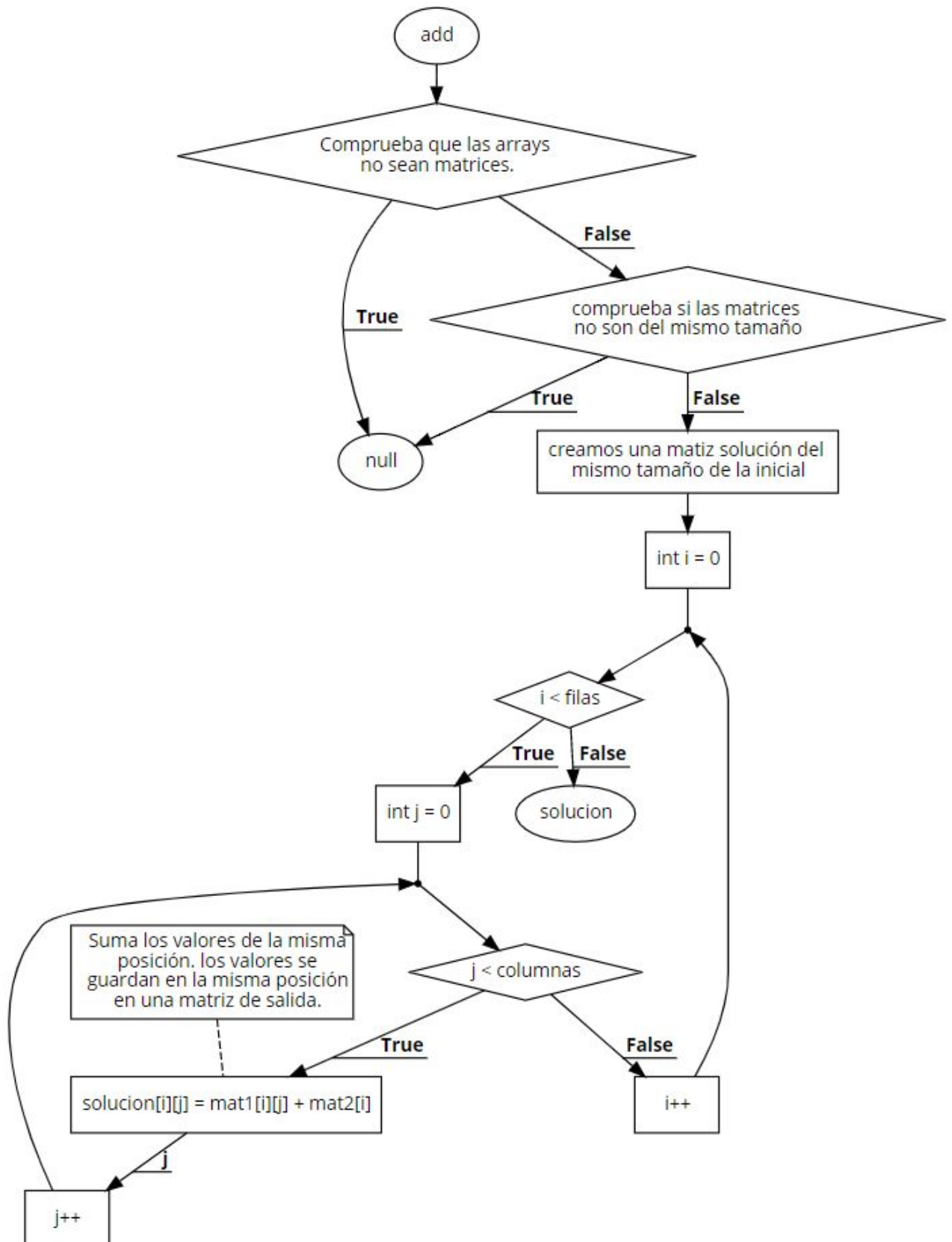
    //Comprueba que ambas matrices sean cuadradas.
    if ((matrizEsCuadrada(mat1)) || (matrizEsCuadrada(mat2)) ) return null;

    double[][] solucion = new double[mat1.length][mat1[0].length];

    /*Suma los valores de la misma posición en las matrices de entrada, los valores se guardan en la misma posición
    en una matriz de salida.*/
    for (int i = 0; i < mat1.length; i++) {
        for (int j = 0; j < mat1[0].length; j++) {
            solucion[i][j] = mat1[i][j] + mat2[i][j];
        }
    }
    return solucion;
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa a la función **add**.



3. Trazada de una matriz cuadrada.

La trazada de una matriz consiste en sumar la diagonal principal de la matriz, para esto hemos de recorrer la matriz incrementando la posición que sumamos.

En esta función tenemos dos casos en los que no se podrá calcular la trazada, uno es si la matriz no es considerada una matriz [esUnaMatriz](#) y el otro es si no es cuadrada [matrizEsCuadrada](#) si uno de estos casos ocurre se hace un **return Double.NaN**, técnicamente se puede calcular la diagonal de una matriz no cuadrada pero no sirve de nada debido a que no se pueden hacer operaciones con dicha trazada.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

$$\text{tr } A = a_{11} + a_{22} + a_{33} + a_{44}$$

Lo primero que haremos es crear una variable para ir guardando el resultado de la suma, esta es **double total = 0;**. A continuación crearemos un bucle **for** que recorrerá la matriz diagonalmente, en mi caso la recorro de la mayor posición hacia el **0,0** ya que la variable **i** empieza valiéndolo el total de la longitud de la matriz menos uno y la voy decrementando, por ejemplo en una matriz de 4x4 iría a estas posiciones: **3,3 - 2,2 - 1,1 - 0,0**. Y finalmente el valor de estas posiciones lo suma a **total**.

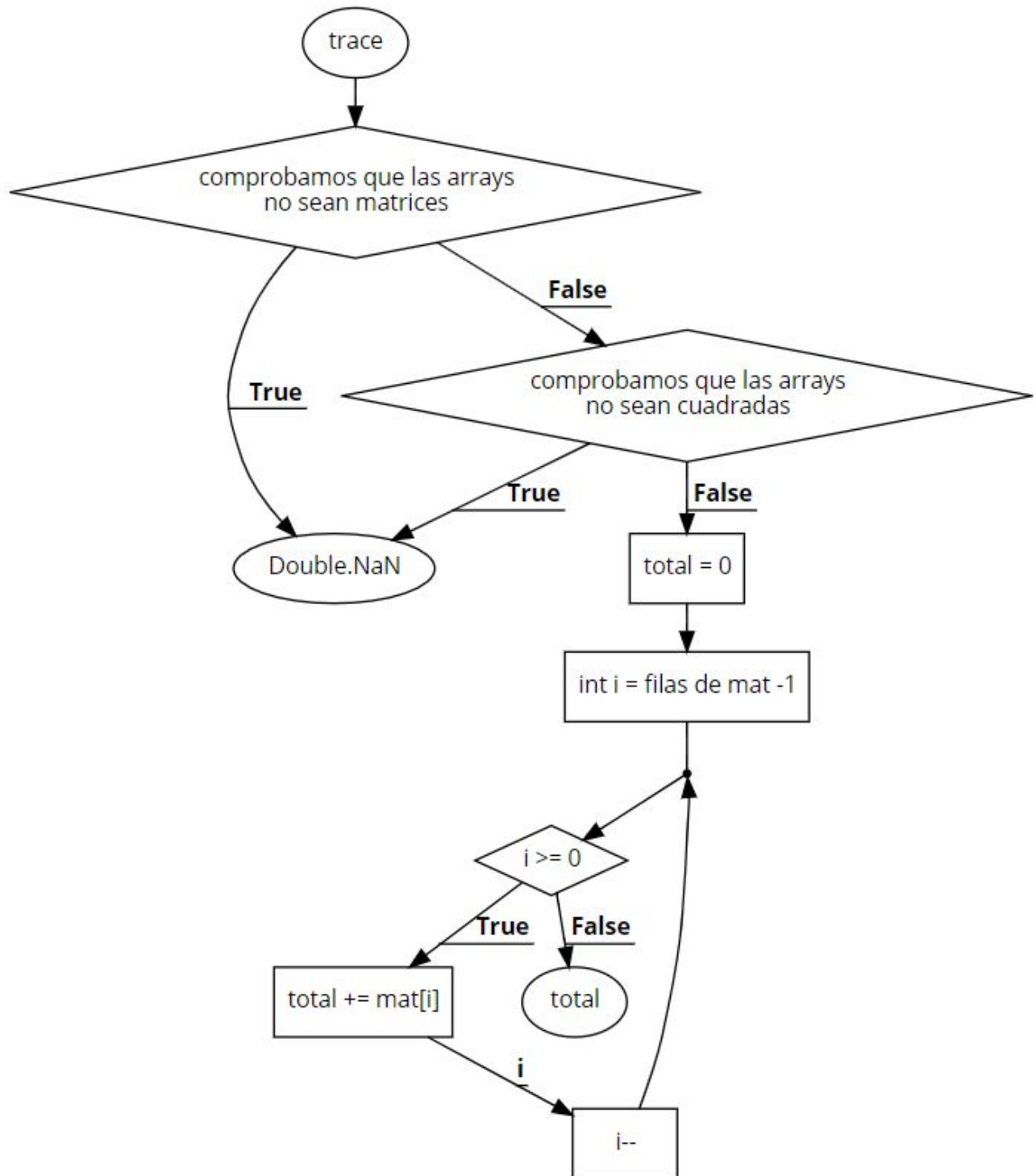
```
static double trace(double[][] mat) {

    if (!esUnaMatriz(mat)) return Double.NaN;
    if (!matrizEsCuadrada(mat)) return Double.NaN;

    double total = 0;
    for (int i = mat.length - 1; i >= 0; i--) {
        total += mat[i][i];
    }
    return total;
}
```

Diagrama de flujo de la función.

Este diagrama de flujo corresponde a la función **trace**. Incluye los casos de error.



4. Multiplicación de dos matrices.

Para Multiplicar dos matrices tenemos que multiplicar las filas de la primera matriz por las columnas de la segunda., luego se suma el resultado de las multiplicaciones y se guarda en la posición correspondiente en una matriz de salida.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 6 & 3 \\ 5 & 2 \\ 4 & 1 \end{pmatrix} = \begin{pmatrix} 1*6 + 2*5 + 3*4 & 1*3 + 2*2 + 3*1 \\ 4*6 + 5*5 + 6*4 & 4*3 + 5*2 + 6*1 \end{pmatrix}$$

Tenemos que comprobar que las dos matrices de entrada sean matrices como tal [esUnaMatriz](#) y también para poder multiplicarlas estas matrices tienen que ser compatibles teniendo la misma cantidad de filas la primera con la misma cantidad de columnas con la segunda, esto lo comprobamos con `if (filas1 != columnas2) return null;`.

Creamos una array del tamaño de las filas de la primera y las columnas de la segunda debido a que la matriz resultante siempre serán esos tamaños `double[][] solucion = new double[filas1][columnas2];`

Para recorrer las dos matrices usaremos un triple bucle de **for** debido a que tenemos que recorrer **mat1** horizontalmente `mat1[i][l]`, **mat2** verticalmente `mat2[l][j]` y el resultado de la multiplicación de las posiciones sumarlo en la posición actual de la matriz **solucion** la cual solo se moverá de posición cuando se han realizado todas las operaciones `solucion[i][j]`.

```
static double[][] mult(double[][] mat1, double[][] mat2) {

    if (!esUnaMatriz(mat1) || !esUnaMatriz(mat2)) return null;

    int filas1 = mat1.length, columnas1 = mat1[0].length;
    int columnas2 = mat2[0].length;

    /*Comprueba que tengan las filas y columnas compatibles*/
    if (filas1 != columnas2) return null;

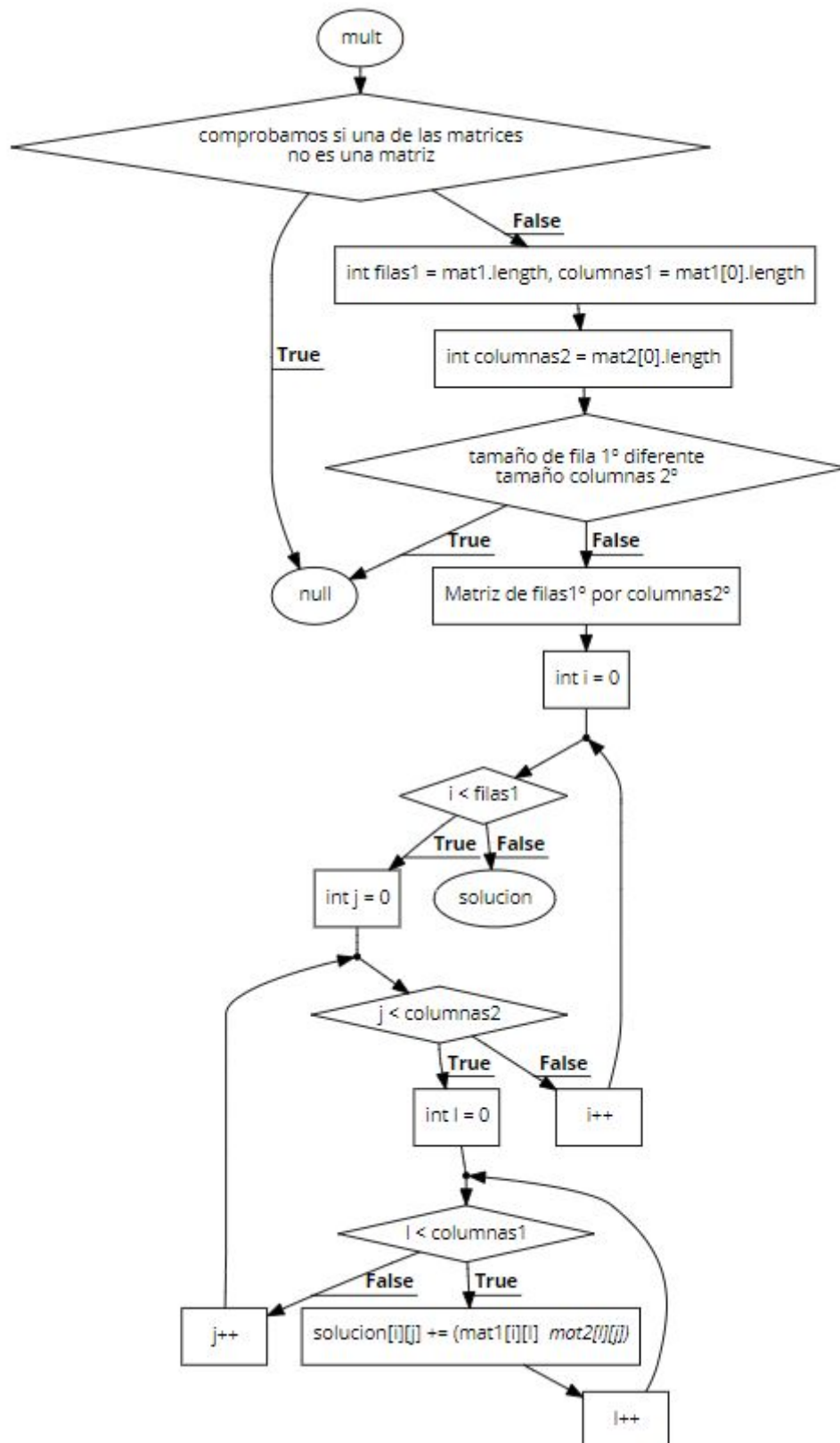
    /*Creamos una array de las filas de la 1ra "mat1[0].length" y las columnas "mat2.length" de la 2da*/
    double[][] solucion = new double[filas1][columnas2];

    for (int i = 0; i < filas1; i++) {
        for (int j = 0; j < columnas2; j++) {
            for (int l = 0; l < columnas1; l++) {
                solucion[i][j] += (mat1[i][l] * mat2[l][j]);
            }
        }
    }

    return solucion;
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **mult**. Incluye la prevención de errores.



5. Multiplicación de una matriz por un número.

Multiplicar una matriz por un número consiste en multiplicar cada posición de la matriz de entrada por el número dado.

$$2 \cdot \begin{pmatrix} 2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 \times 2 & 2 \times 0 & 2 \times 1 \\ 2 \times 3 & 2 \times 0 & 2 \times 0 \\ 2 \times 5 & 2 \times 1 & 2 \times 1 \end{pmatrix} = \begin{pmatrix} 4 & 0 & 2 \\ 6 & 0 & 0 \\ 10 & 2 & 2 \end{pmatrix}$$

Para que la función funcione correctamente tenemos que hacer un par de prevenciones de errores, tenemos que tener en cuenta que el usuario introduzca una matriz válida [esUnaMatriz](#) y que el número por el que se tiene que multiplicar dicha matriz no sea infinito esto lo comprobaremos con `if`

```
(Double.isFinite(n)) return null;.
```

Crearemos una matriz para guardar la solución del tamaño de la matriz de entrada:

```
double[][] solucion = new double[mat.length][mat[0].length];
```

Para llevar a cabo esta operación simplemente necesitamos un doble **for**, uno incrementa posición horizontalmente tantas veces como columnas y el otro verticalmente tantas veces como filas e incrementa cada vez que acaba el total de columnas. La posición correspondiente tiene que ser multiplicada por `n` y guardada en la matriz **solucion**. Esta es la operación correspondiente: `solucion[i][j] = mat[i][j] * n;`

```
static double[][] mult(double[][] mat, double n) {

    if (Double.isFinite(n)) return null;
    if (!esUnaMatriz(mat)) return null;

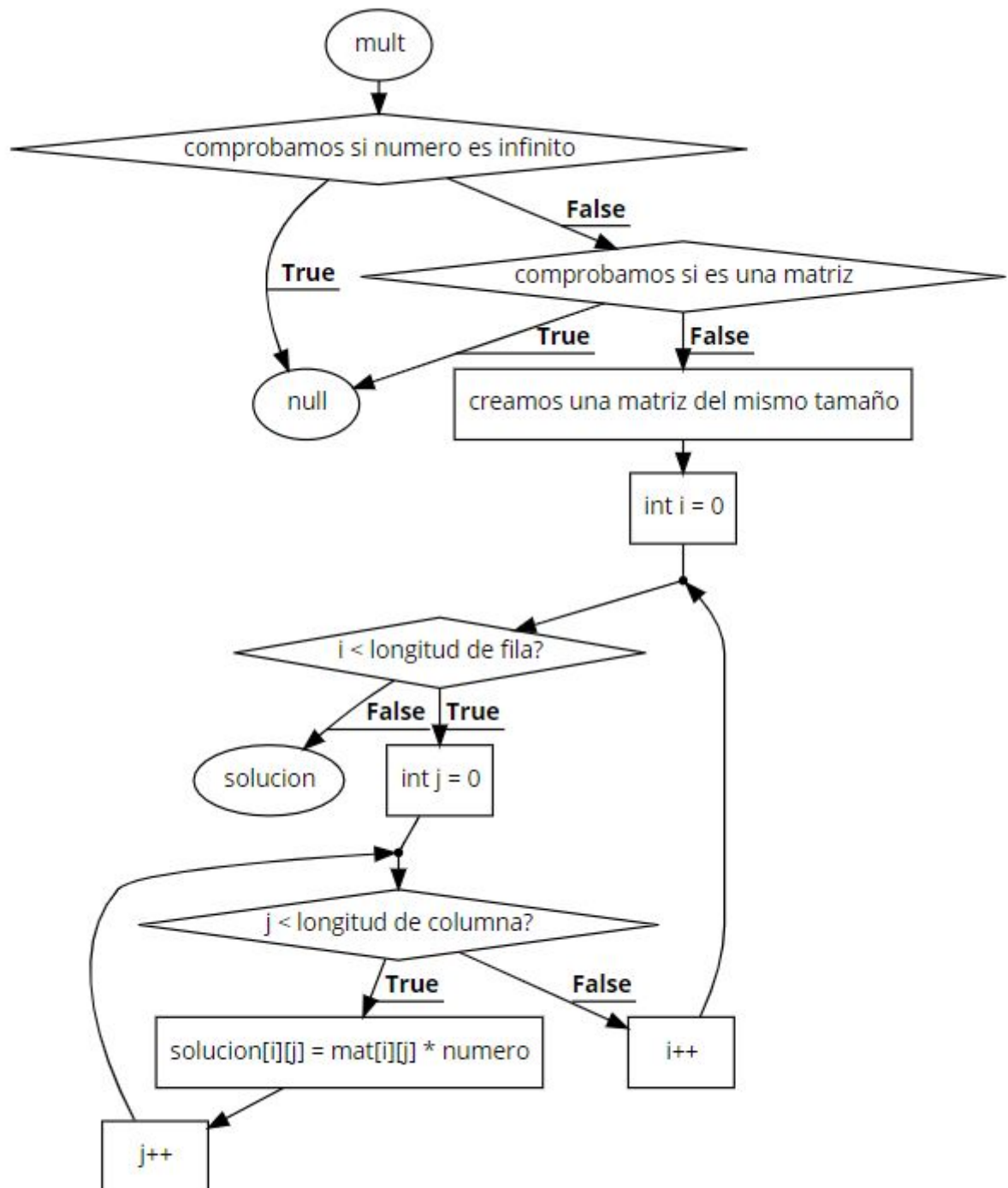
    double[][] solucion = new double[mat.length][mat[0].length];

    for (int i = 0; i < mat.length; i++) {
        for (int j = 0; j < mat[0].length; j++) {
            solucion[i][j] = mat[i][j] * n;
        }
    }

    return solucion;
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **mult**. Incluye la prevención de errores.



6. Potencia de una matriz.

La potencia de una matriz es la multiplicación por sí misma tantas veces como la potencia nos indica. Una vez que se multiplica se tiene que guardar el resultado para multiplicarse por su principal. Por lo tanto hay que guardar la matriz resultante.

$$A^2 = \begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 6 & 1 \end{pmatrix}$$

$$A^3 = A^2 \cdot A = \begin{pmatrix} 1 & 0 \\ 6 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 9 & 1 \end{pmatrix}$$

Para poder hacer los cálculos de **power** tenemos que tener en cuenta unos cuantos casos de error, tenemos que comprobar que [esUnaMatriz](#). También que la matriz sea cuadrada para poder multiplicarla [matrizEsCuadrada](#).

Por último si la potencia a la que se eleva es negativa tenemos que cambiar el signo a la potencia y invertir la matriz

$$\begin{pmatrix} 11 & 3 \\ 7 & 11 \end{pmatrix}^{-2} = \begin{pmatrix} 11/100 & -3/100 \\ -7/100 & 11/100 \end{pmatrix}^2$$

Tenemos que tener en cuenta también que la potencia puede ser 0, si este es el caso simplemente llamamos a la función [identityMatrix](#) y devolvemos el resultado.

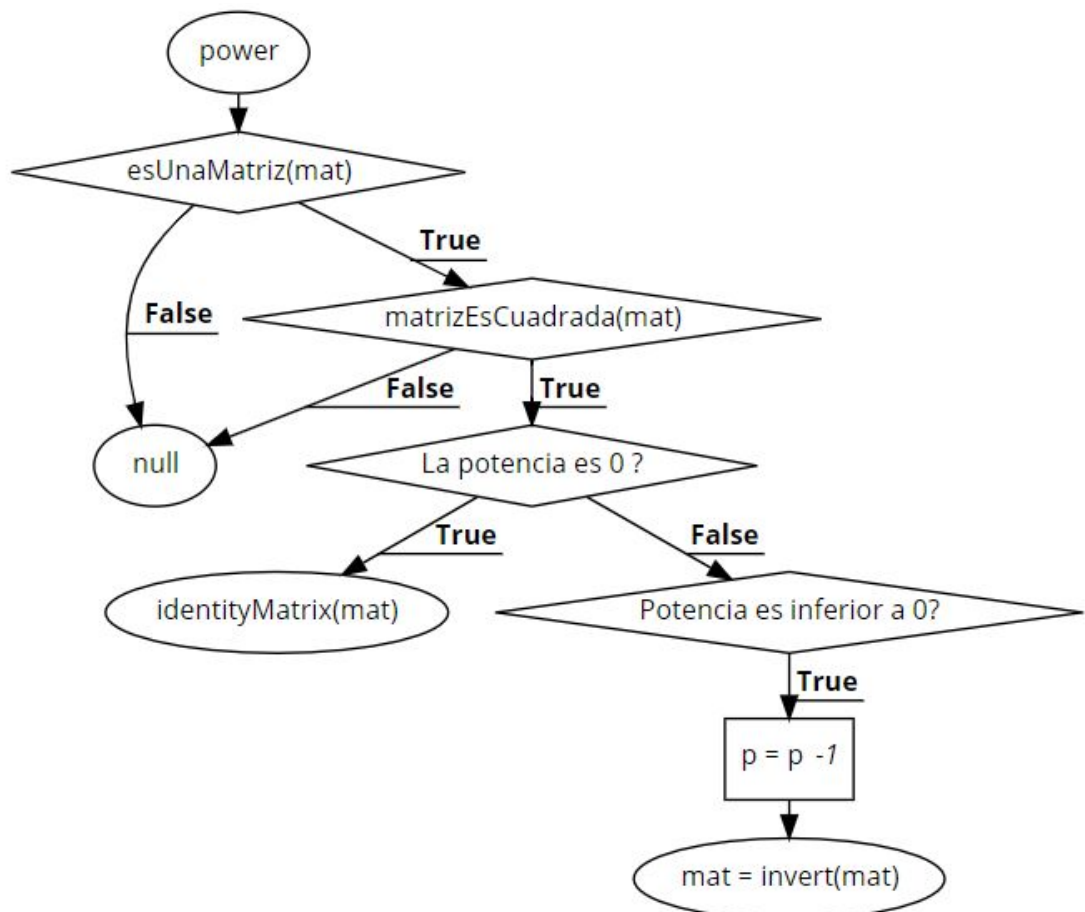
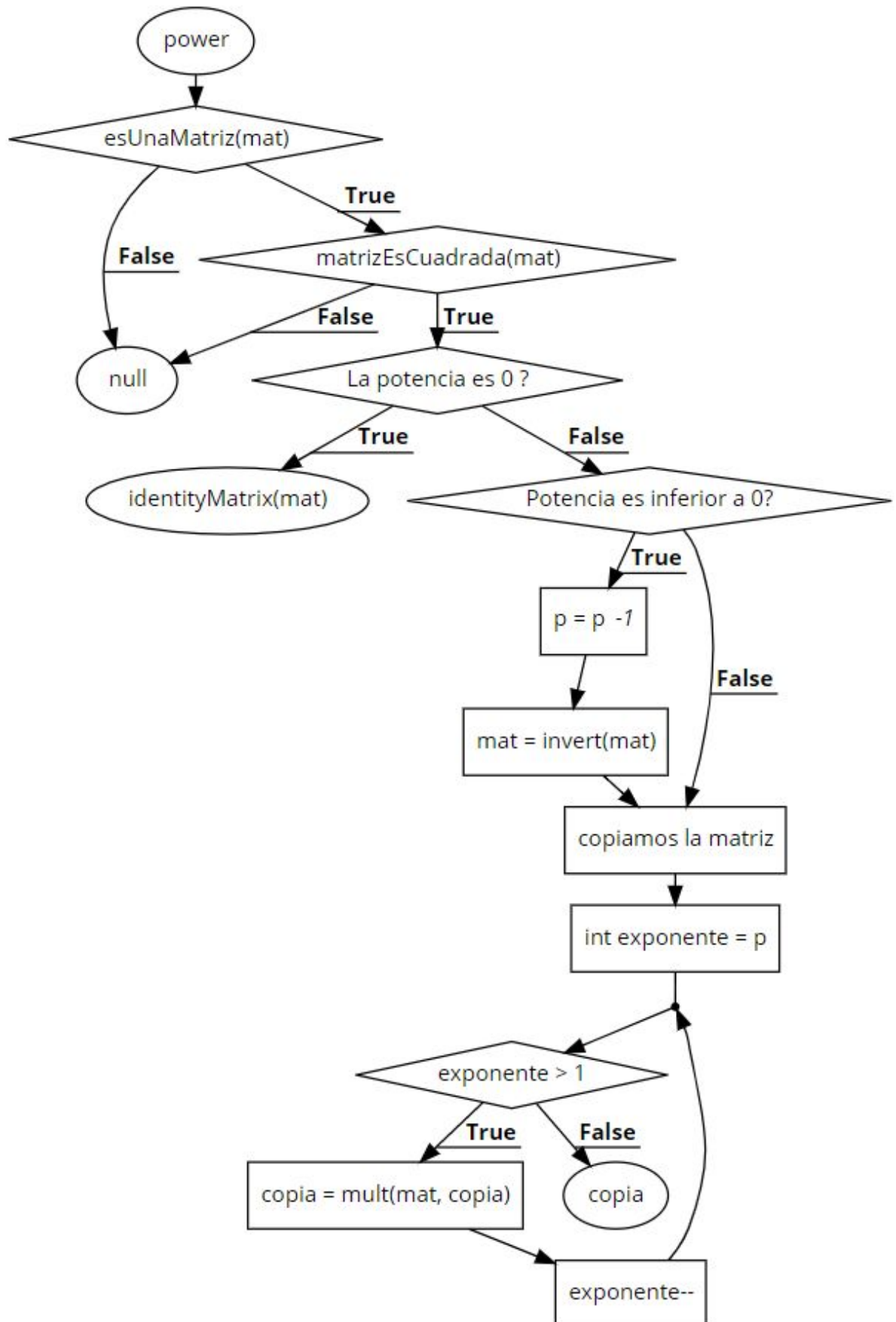


Diagrama de flujo de la función.

Este diagrama de flujo representa la función **power**. Incluye la prevención de errores.



7. Calcular el determinante de una matriz.

Para que esta función sea más óptima tenemos que usar getMinor:

7.1. Definir una matriz inferior, función getMinor.

La función get minor se encarga de hacer una matriz 1x1 más pequeña que la original, para esto necesita que se le pase una matriz y dos coordenadas, lo que hará será elegir la posición y eliminar esa fila y columna.

Por ejemplo si mat es **A**, **x = 2** e **y = 4** se eliminan así.

$$A = \begin{bmatrix} -4 & 2 & 5 & -1 \\ 6 & 0 & 1 & 2 \\ 4 & 3 & 0 & -3 \\ -5 & 1 & -1 & 2 \end{bmatrix} \quad M_{24} = \begin{bmatrix} -4 & 2 & 5 & -1 \\ 6 & 0 & 1 & 2 \\ 4 & 3 & 0 & -3 \\ -5 & 1 & -1 & 2 \end{bmatrix}$$

Tenemos que tener en cuenta que esta acción se tiene que hacer en una matriz llamaremos a la función [esUnaMatriz](#) solo se puede hacer en matrices que sean cuadradas llamamos a la función [matrizEsCuadrada](#) y también tenemos que comprobar que el número que nos dan en **x** e **y** esté delimitado en el rango de tamaño de la matriz.

Una vez se cumplen estas condiciones creamos una **matriz 1x1 inferior** a la de entrada, utilizaremos un doble bucle de **for** para recorrer la matriz horizontal y verticalmente, mientras la recorremos comprobaremos si en la posición que nos encontramos coincide **i** con la **x** y si la **j** coincide con **y**, si no coincide ninguna de las dos copiaremos el número de esa posición en la matriz que hemos creado y avanzaremos una posición en la matriz del resultado para esto hemos creado un contador llamado **filas** y otro **columnas** ya que solo hay que avanzar cuando se copia el número, este proceso se repetirá tantas veces como longitud de la matriz, una vez sea igual se resetean las **columnas** y se incrementan las **filas**.

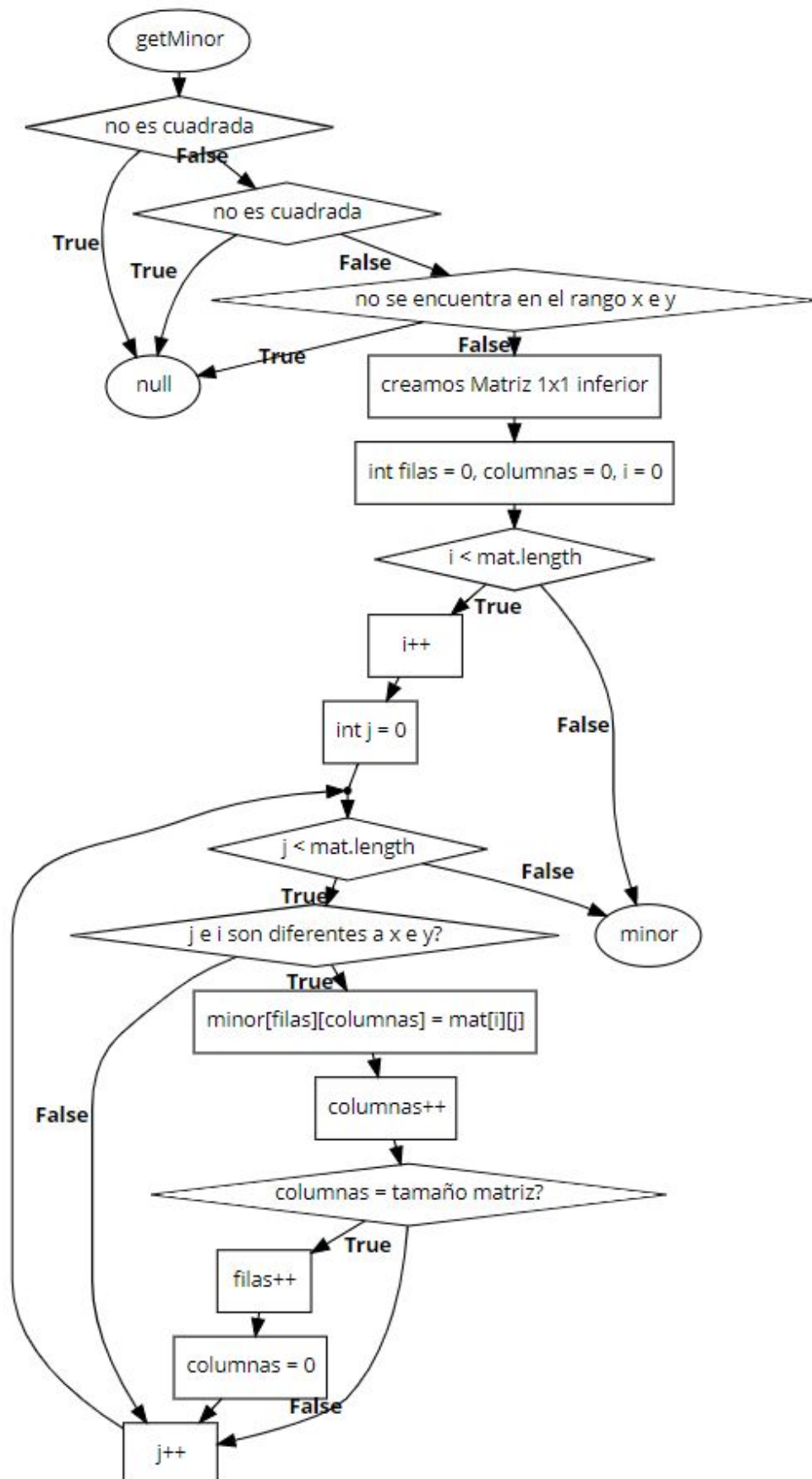
```
static double[][] getMinor(double[][] mat, int x, int y) {
    if (!esUnaMatriz(mat)) return null;
    if (!matrizEsCuadrada(mat)) return null;
    if ((x < 0) || (y < 0) || (x > mat[0].length) || (y > mat.length)) return null;

    /*Crea una array 1x1 inferior a la original*/
    double[][] minor = new double[mat.length - 1][mat[0].length - 1];
    int filas = 0;
    int columnas = 0;

    for (int i = 0; i < mat.length; i++) {
        for (int j = 0; j < mat[0].length; j++) {
            /*Cuando i y j coincidan con las posiciones que no queremos no se copiará
            if ((i != x) && (j != y)) {
                minor[filas][columnas] = mat[i][j];
                columnas++;
                if (columnas == minor.length) {
                    filas++;
                    columnas = 0;
                }
            }
        }
    }
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **getMinor**. Incluye la prevención de errores.



Para calcular el determinante de una matriz tenemos que evaluar la matriz en la que estamos debido a que para sacar el determinante necesitamos tener matrices de 2x2, por lo tanto si tenemos una matriz de 3x3 tendremos que ir haciendo esta matriz más pequeña cada vez “tapando” una fila y columna a partir de un número gracias a la función [getMinor](#), ese número se guarda y se calcula el determinante de la 2x2 haciendo $a_{22} \cdot a_{33} - a_{22} \cdot a_{31}$ y con el número que tapamos antes multiplicamos por el resultado de la operación actual “ $a_{11}(a_{22} \cdot a_{33} - a_{22} \cdot a_{31})$ ”, esto se repite tantas veces como longitud tenga la matriz.

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} =$$

$$= a_{11} \cdot (-1)^{1+1} \cdot \begin{vmatrix} \times & \times & \times \\ \times & a_{22} & a_{23} \\ \times & a_{32} & a_{33} \end{vmatrix}$$

$$+ a_{12} \cdot (-1)^{1+2} \cdot \begin{vmatrix} \times & \times & \times \\ a_{21} & \times & a_{23} \\ a_{31} & \times & a_{33} \end{vmatrix}$$

$$+ a_{13} \cdot (-1)^{1+3} \cdot \begin{vmatrix} \times & \times & \times \\ a_{21} & a_{22} & \times \\ a_{31} & a_{32} & \times \end{vmatrix}$$

Para hacer estos pasos tenemos que tener en cuenta también que la array tiene que ser considerada una matriz con la función [esUnaMatriz](#), tenemos que comprobar si es cuadrada con la función [matrizEsCuadrada](#) y si la matriz es de 1x1 simplemente devolvemos la posición 0,0 de la matriz.

```
static double determinant(double[][] mat) {

    double resultado = 0;
    int indiceColumna = 0;

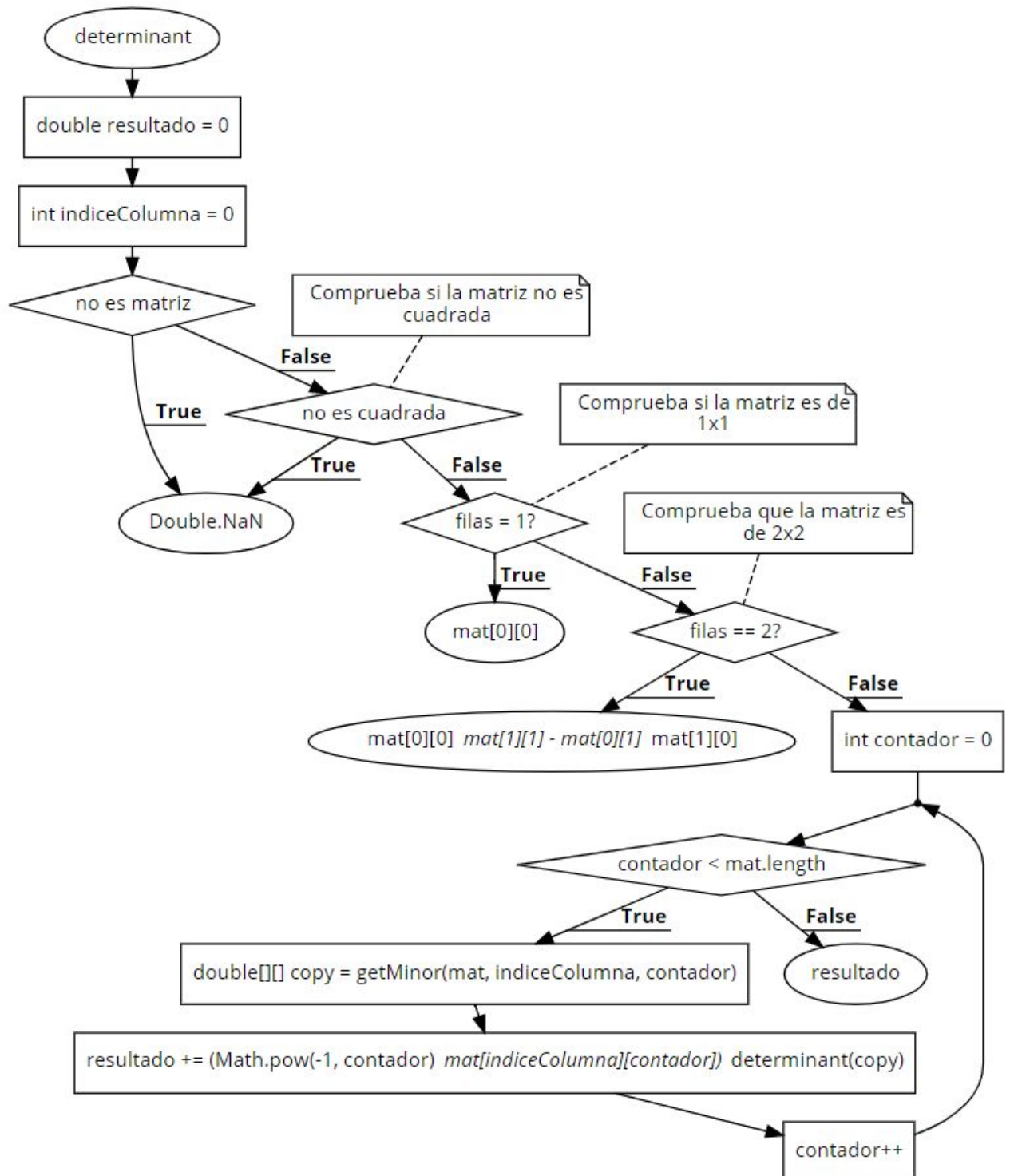
    if (!esUnaMatriz(mat)) return Double.NaN;
    if (!matrizEsCuadrada(mat)) return Double.NaN;
    if (mat.length == 1) return mat[0][0];

    //Comprueba que la matriz es de 2x2
    if (mat.length == 2) return mat[0][0] * mat[1][1] - mat[0][1] * mat[1][0];

    for (int contador = 0; contador < mat.length; contador++) {
        double[][] copy = getMinor(mat, indiceColumna, contador);
        resultado += (Math.pow(-1, contador) * mat[indiceColumna][contador]) * determinant(copy);
    }
    return resultado;
}
```

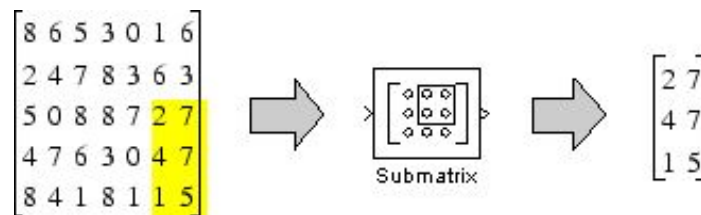
Diagrama de flujo de la función.

Este diagrama de flujo representa la función **determinat**. Incluye la prevención de errores.



8. Calcular una sub-matriz.

Calcular una sub-matriz consiste en seleccionar dos posiciones formadas por **x1,y1** e **x2,y2** una vez tenemos seleccionadas las dos posiciones se define una matriz de punto a punto. Por ejemplo si seleccionamos **posición1(x1=5 e y1=2)** **posición2(x2=5 e y2=4)**.



Tenemos que tener en cuenta que la array de entrada sea una matriz usando la función [esUnaMatriz](#), también las posiciones no pueden ser inferiores a 0 ni mayores que el tamaño de la matriz y por último si x1 o y1 es mayor que x2 o y2 tenemos que cambiar uno por otro.

```
static double[][] submatrix(double[][] mat, int x1, int y1, int x2, int y2) {

    if (!esUnaMatriz(mat)) return null;

    //Se comprueba que x1,x2,y1,y2 estén en el rango de la matriz
    if ((x1 < 0) || (y1 < 0) || (x2 < 0) || (y2 < 0)) return null;
    if ((x1 > mat[0].length-1) || (y1 > mat.length-1) ||
        (x2 > mat[0].length-1) || (y2 > mat.length-1)) return null;

    //comprobamos si x1 es mayor que x2, si es así los cambiamos
    if (x1 > x2) {
        int tempNum = x1;
        x1 = x2;
        x2 = tempNum;
    }

    //comprobamos si y1 es mayor que y2, si es así los cambiamos
    if (y1 > y2) {
        int tempNum = y1;
        y1 = y2;
        y2 = tempNum;
    }

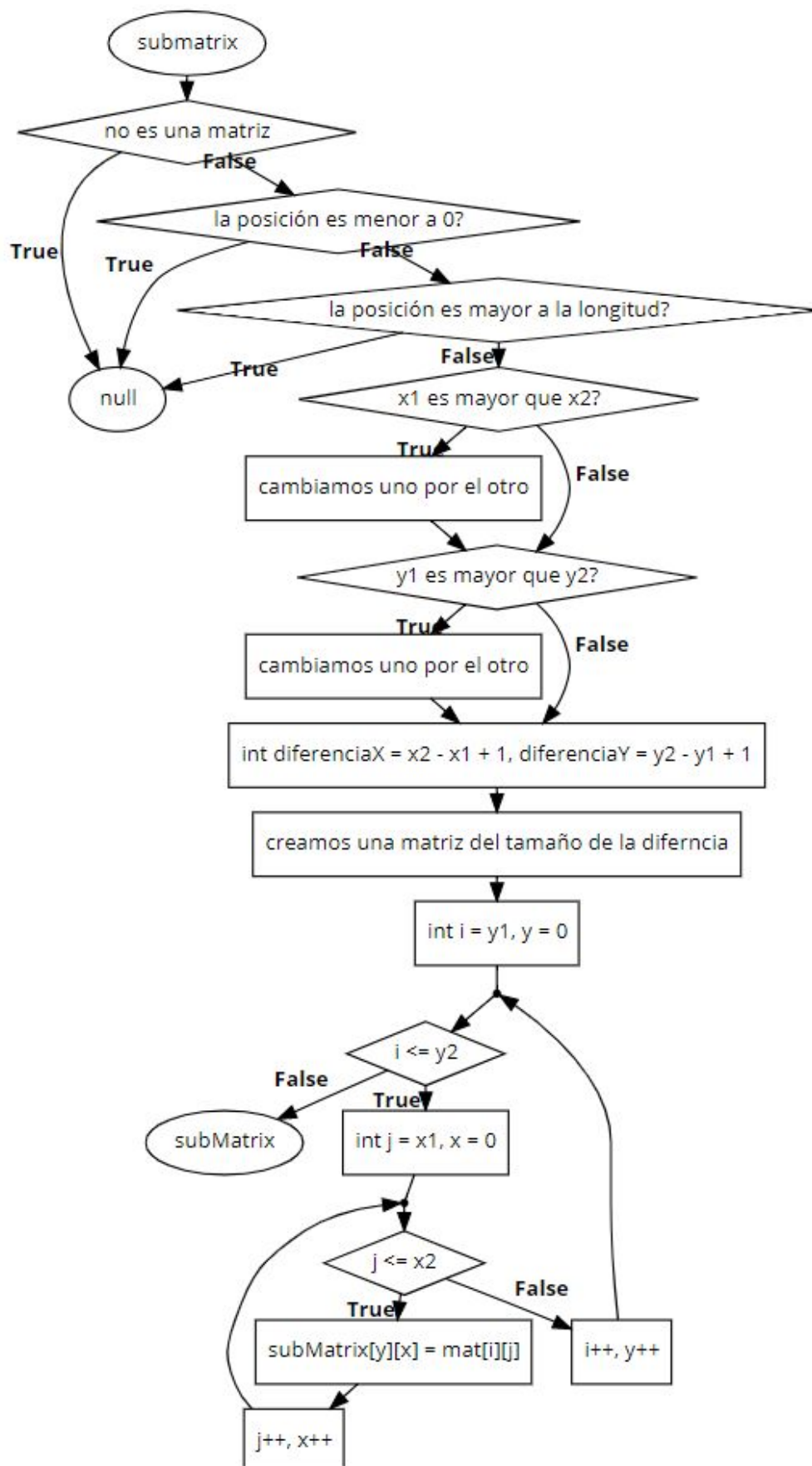
    int diferenciaX = x2 - x1 + 1, diferenciaY = y2 - y1 + 1;
    double[][] subMatrix = new double[diferenciaY][diferenciaX];

    for (int i = y1, y = 0; i <= y2; i++, y++) {
        for (int j = x1, x = 0; j <= x2; j++, x++) {
            subMatrix[y][x] = mat[i][j];
        }
    }

    return subMatrix;
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **submatrix**. Incluye la prevención de errores.



9. Cálculo de una matriz transpuesta.

Para calcular la transposición de una matriz, simplemente tenemos que cambiar las posiciones de fila por posición de columna y viceversa.

$$\begin{bmatrix} 0 & 0 & 4 \\ 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 3 & 2 \\ 0 & 2 & 3 \\ 0 & 3 & 4 \\ 3 & 3 & 1 \end{bmatrix}^t = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 3 & 2 & 3 & 3 \\ 4 & 4 & 0 & 2 & 3 & 4 & 1 \end{bmatrix}$$

Para hacer la transposición tenemos que tener en cuenta también que la array tiene que ser considerada una matriz con la función [esUnaMatriz](#) y si la matriz es de 1x1 directamente devolverse a sí misma.

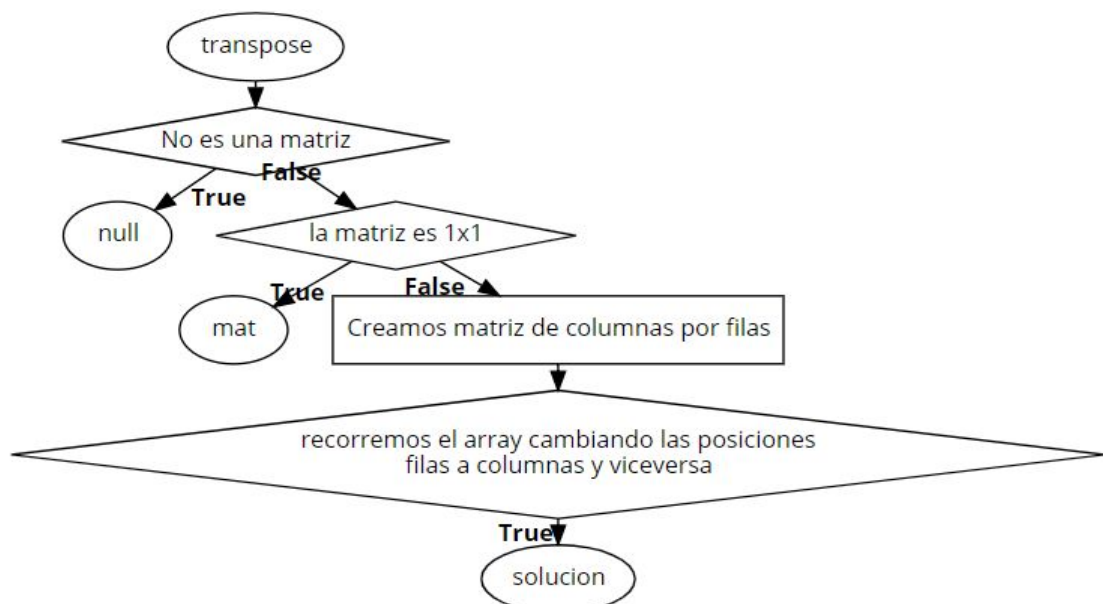
```
if (!esUnaMatriz(mat)) return null;
/*Comprueba que el array es menor que 1, si es así se devuelve a sí misma */
if ((mat.length == 1) && (mat[0].length == 1)) return mat;

double[][] solucion = new double[mat[0].length][mat.length];

for (int i = 0; i < mat.length; i++) {
    for (int j = 0; j < mat[0].length; j++) {
        solucion[j][i] = mat[i][j];
    }
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **transpose**. Incluye la prevención de errores.



10. Calcular la inversa de una matriz.

Para calcular la inversa de una matriz tenemos que transponer dicha matriz, una vez transpuesta calculamos el [determinante](#) de esta y lo dividimos entre 1 este resultado lo guardamos para más tarde. A continuación llamaremos a la función [getMinor](#) para hacer una matriz más pequeña cada vez, get minor necesita una posición para el seleccionar qué fila y columna quitar esta posición se va incrementando para conseguir una matriz inferior cada vez y de esa matriz inferior se calcula el determinante otra vez. Ahora con los determinantes de cada matriz inferior los multiplicamos por el 1/determinante, el resultado es la matriz original invertida.

$$\begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}^{-1} = \frac{1}{4 \times 6 - 7 \times 2} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix}$$

$$= \frac{1}{10} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} 0.6 & -0.7 \\ -0.2 & 0.4 \end{bmatrix}$$

Antes de calcular el determinante tenemos que comprobar que el array tiene que ser considerada una matriz con la función [esUnaMatriz](#) y que es cuadrada [matrizEsCuadrada](#), por último si el determinante de la principal da 0, ya que no se puede operar con el, por que cualquier numero/0 es infinito.

```
static double[][] invert(double[][] mat) {

    if (!esUnaMatriz(mat)) return null;
    if (!matrizEsCuadrada(mat)) return null;

    double[][] matInvertida = new double[mat.length][mat[0].length];
    double[][] matTransposed = transpose(mat);
    double deter = determinant(mat);

    if (deter == 0) {
        return null;
    }

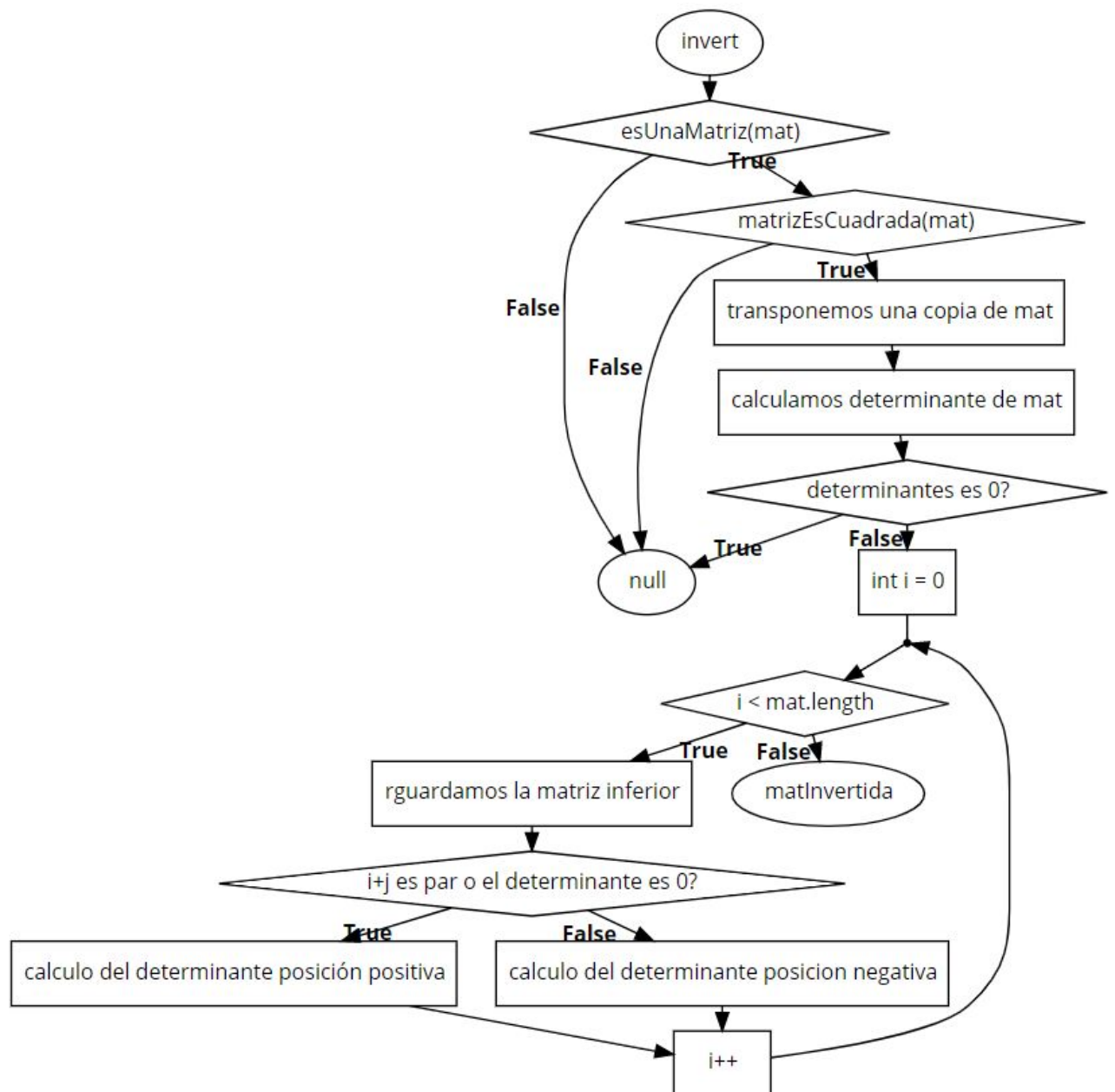
    double determinantePrincipal = 1 / deter;

    for (int i = 0; i < mat.length; i++) {
        for (int j = 0; j < mat[0].length; j++) {
            double[][] copy = getMinor(matTransposed, i, j);
            //comprueba si la suma de la posición de i+j es par o impar, si es par, hará la operación con el signo
            if (((i + j) % 2 == 0) || (determinant(copy) == 0)) {
                matInvertida[i][j] = (determinantePrincipal * (Math.pow(-1, 0) * determinant(copy)));
            } else {
                matInvertida[i][j] = (determinantePrincipal * (Math.pow(-1, 1) * determinant(copy)));
            }
        }
    }

    return matInvertida;
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **invert**. Incluye la prevención de errores.



11. División de matrices.

Para dividir dos matrices tenemos que invertir una de ellas para invertirla usamos la función [invert](#) y multiplicarla por la otra usamos la función [mult](#).

$$\frac{A}{B} = ? \quad \frac{A}{B} = A B^{-1}$$

$$\text{If } A = \begin{bmatrix} 1 & 4 \\ -2 & 3 \end{bmatrix} \quad B = \begin{bmatrix} -2 & 5 \\ 6 & 7 \end{bmatrix} \quad \text{then } B^{-1} = \frac{1}{D} \begin{bmatrix} 7 & -5 \\ -6 & -2 \end{bmatrix}$$

$$D = (-2)(7) - (5)(6) = -44$$

$$\frac{A}{B} = A B^{-1} = \begin{bmatrix} 1 & 4 \\ -2 & 3 \end{bmatrix} \frac{1}{-44} \begin{bmatrix} 7 & -5 \\ -6 & -2 \end{bmatrix}$$

Para poder dividir las ambas tienen que ser consideradas matrices, por lo tanto llamamos a la función [esUnaMatriz](#), la matriz la cual vamos a invertir tiene que ser cuadrada [matrizEsCuadrada](#) y ambas tienen que ser compatibles para poder ser multiplicadas a si que las columnas de una tienen que tener la misma longitud de las filas de la otra.

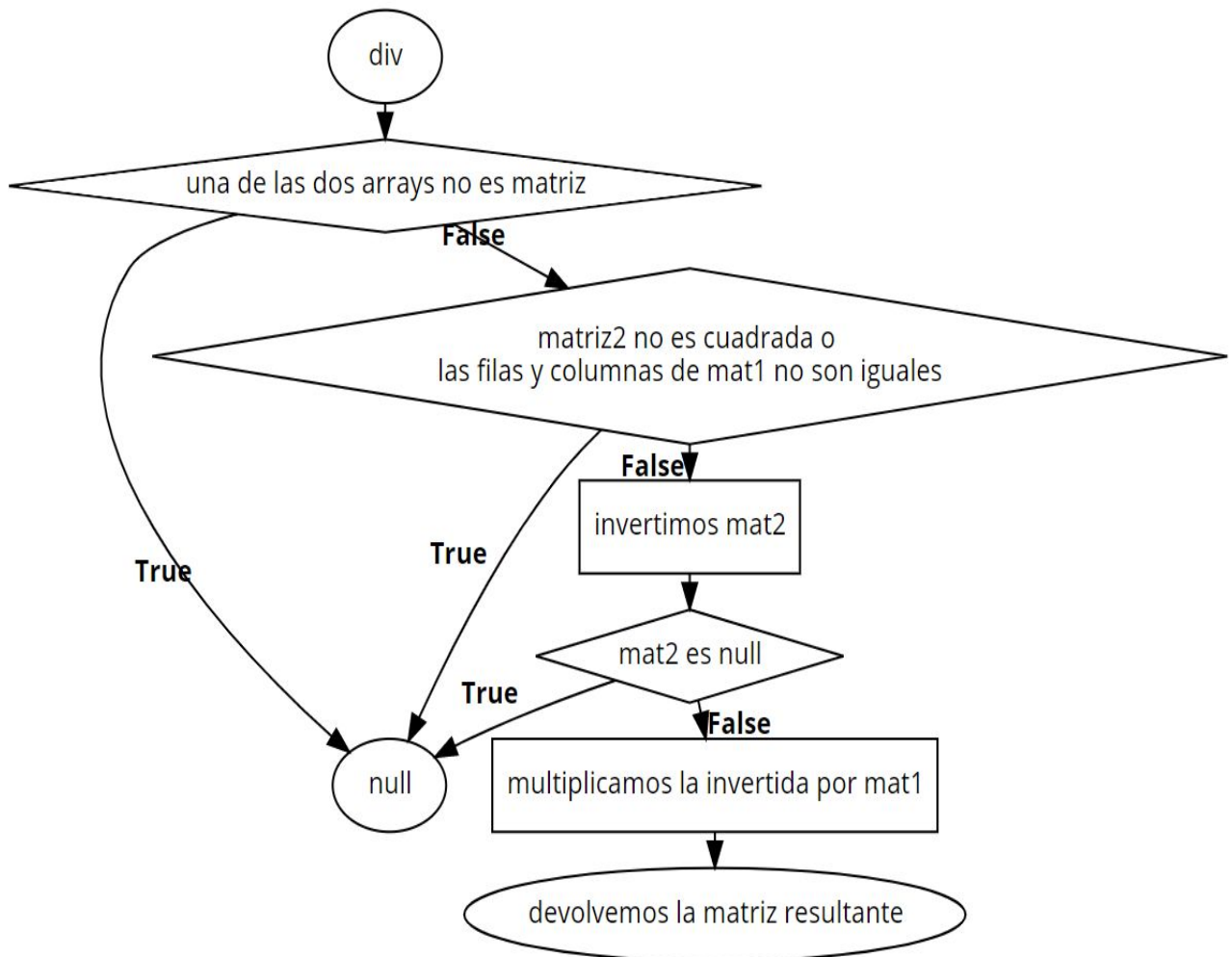
```
static double[][] div(double[][] mat1, double[][] mat2) {

    if ((!esUnaMatriz(mat1)) || (!esUnaMatriz(mat2))) return null;
    if (!matrizEsCuadrada(mat2) || (mat1[0].length != mat2.length)) return null;

    double[][] matInvert = invert(mat2);
    if (matInvert == null) return null;
    matInvert = mult(mat1, matInvert);
    return matInvert;
}
```

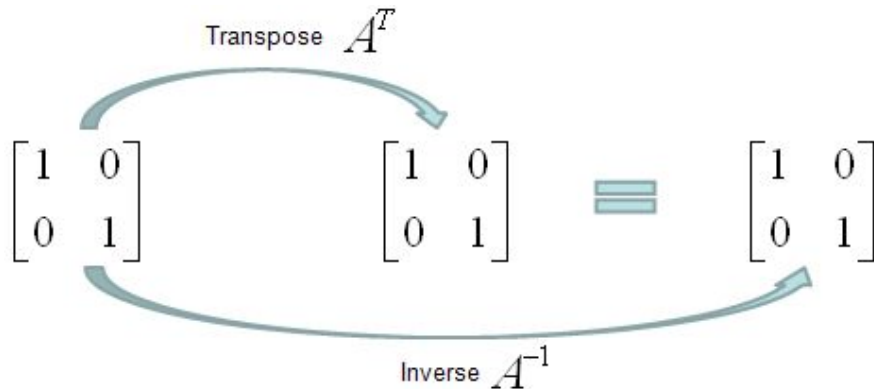
Diagrama de flujo de la función.

Este diagrama de flujo representa la función **div**. Incluye la prevención de errores.



12. Test de la ortogonalidad de una matriz.

Para comprobar si una matriz es ortogonal tenemos que comparar la inversa con la función [invert](#) y la traspuesta de esa matriz con la función [transpose](#), si son iguales entonces la matriz es ortogonal.



$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \xrightarrow{\text{Transpose } A^T} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \xleftarrow{\text{Inverse } A^{-1}}$$

Tenemos que comprobar si la matriz es considerada una matriz como tal con la función [esUnaMatriz](#) y también comprobar si es cuadrada con la función [matrizEsCuadrada](#).

Usaremos la función transpose para [transponer](#) la matriz y usaremos la función [invert](#) para invertir la matriz, una vez hecho los comparamos, si son iguales es ortogonal.

```
static boolean isOrtho(double[][] mat) {

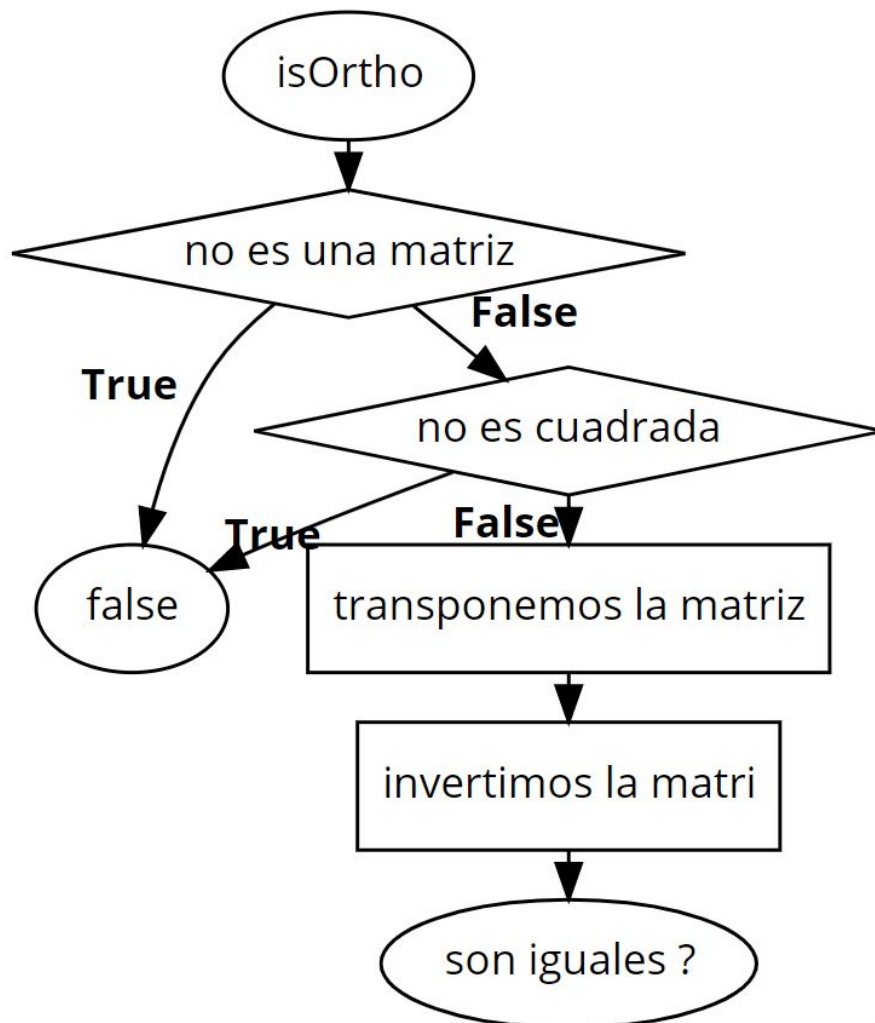
    if (!esUnaMatriz(mat)) return false;
    if (!matrizEsCuadrada(mat)) return false;

    double[][] matrizTranspuesta = transpose(mat);
    double[][] matrizInversa = invert(mat);

    return Arrays.deepEquals(matrizInversa, matrizTranspuesta);
}
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **isOrtho**. Incluye la prevención de errores.



13. Resolución de ecuaciones mediante la regla de cramer.

Cramer es un método para resolver sistemas de ecuaciones lineales el cual consiste en transformar la ecuación en una matriz sin contar el resultado. Dicha matriz será de una fila columna menos que la ecuación inicial, a continuación calculamos el [determinante](#) de la matriz actual y lo guardamos. A continuación iremos sustituyendo las incógnitas por el resultado consecutivamente y calculamos el determinante de cada matriz con las incógnitas sustituidas. el determinante de cada matriz dividido por el determinante de la inicial es la solución de dicha incógnita.

$$\begin{array}{rcl} 2x + 3y - 5z & = & 1 \\ x + y - z & = & 2 \\ 2y + z & = & 8 \end{array} \quad x = \frac{D_x}{D} \quad y = \frac{D_y}{D} \quad z = \frac{D_z}{D}$$

$$D = \begin{vmatrix} 2 & 3 & -5 \\ 1 & 1 & -1 \\ 0 & 2 & 1 \end{vmatrix} = -7 \quad D_y = \begin{vmatrix} 2 & 1 & -5 \\ 1 & 2 & -1 \\ 0 & 8 & 1 \end{vmatrix} = -21$$

$$D_x = \begin{vmatrix} 1 & 3 & -5 \\ 2 & 1 & -1 \\ 8 & 2 & 1 \end{vmatrix} = -7 \quad D_z = \begin{vmatrix} 2 & 3 & 1 \\ 1 & 1 & 2 \\ 0 & 2 & 8 \end{vmatrix} = 14$$

Para poder realizar estas operaciones tenemos que comprobar que la array de entrada es una matriz, para esto usaremos la función [esUnaMatriz](#) y que la matriz tenga una columna más que fila para ser considerada un sistema de ecuaciones representado en matriz.

```
if (!esUnaMatriz(mat)) return null;
// Comprueba que la matriz tenga una columna más que filas.
if (mat.length + 1 != mat[0].length) return null;

//Creamos una matriz de una columna inferior.
double[][] matrizPrincipal = new double[mat.length][mat[0].length - 1];
double[] matrizIncognitas = new double[mat.length];

//crea la matriz de 1 columna inferior.
for (int j = 0; j < matrizPrincipal.length; j++) {
    System.arraycopy(mat[j], srcPos: 0, matrizPrincipal[j], destPos: 0, matrizPrincipal[0].length);
}

double determinantePrincipal = determinant(matrizPrincipal);

for (int i = 0; i < matrizPrincipal[0].length; i++) {
    //reestablece la matriz sin las incognitas cambiadas.
    for (int k = 0; k < matrizPrincipal.length; k++) {
        System.arraycopy(mat[k], srcPos: 0, matrizPrincipal[k], destPos: 0, matrizPrincipal[0].length);
    }
    //Cambia las incognitas por el resultado.
    for (int j = 0; j < matrizPrincipal.length; j++) {
        matrizPrincipal[j][i] = mat[j][mat[0].length - 1];
    }
    //Calcula el determinante de la matriz cambiada.
    double determinanteInferior = determinant(matrizPrincipal);
    matrizIncognitas[i] = determinanteInferior / determinantePrincipal;
}

return matrizIncognitas;
```

Diagrama de flujo de la función.

Este diagrama de flujo representa la función **cramer**. Incluye la prevención de errores

