



# Práctica obligatoria

## Resolver un Survo- Puzzle

Realizada por:

Eduardo de Diego Lucas

3º Ingeniería en Telecomunicación + ITIS

Estructura de Datos y de la Información  
URJC 2011



## **Estudio del problema y diseño algorítmico**

### **Detección del esquema algorítmico**

**El alumno deberá responder razonadamente a las siguientes cuestiones:**

**1. ¿Por qué el esquema algorítmico más adecuado para solucionar el problema es un esquema basado en exploración de grafos? Razonar por qué otras aproximaciones no serían válidas.**

El esquema algorítmico basado en exploración de grafos es el más adecuado para resolver el problema ya que es el más óptimo desde la perspectiva de cómo organizar los datos y cómo utilizarlos para encontrar la solución real y óptima, la mejor, que es la que buscamos. Si hubiésemos utilizado un esquema lineal o voraz, en el que cada número a elegir debe ir en una posición concreta del survo-puzzle y sólo en esa posición, nunca hubiésemos llegado a encontrar la solución adecuada y por tanto, fracasaríamos en nuestro intento de resolver el problema planteado.

La exploración de grafos ayuda además, a tener la posibilidad de trabajar con diferentes caminos posibles; en realidad, trabajamos con todos los caminos posibles para resolver el problema. Aquí está lo interesante, si alguno de estos caminos es el equivocado, siempre podemos elegir algún otro. Este esquema algorítmico resulta evidente utilizarlo por ejemplo en esta clase de juegos tipo puzzle, donde tenemos que ser capaces de seleccionar el valor adecuado que corresponde a cada posición.

**2. ¿Qué tipo de estrategia de exploración de grafos (vuelta atrás o ramificación y poda) considera más adecuada para resolver su problema concreto? Razone su respuesta y explique las diferencias entre ambas.**

El tipo de estrategia que he utilizado en mi práctica ha sido la de **vuelta atrás** o **"backtracking"**. Con esta estrategia puedo llegar al resultado óptimo viendo todos los caminos posibles, tal y como se ha comentado en la anterior pregunta, y descartar algún camino que no sea factible para la solución adecuada. Siendo un algoritmo recursivo, a medida que se va llamando a la función, se va consiguiendo el camino de exploración óptima para resolver el problema. El recorrido planteado para el sudoku es lineal, fila por fila, por lo que por cada posición se llama recursivamente al algoritmo. Si alguna llamada falla, el control del programa vuelve donde se produjo la última llamada y se prosigue con el camino. Esto nos da una exploración del grafo o el árbol, válidas ambos tipos de datos teóricos, con recorriendo en anchura, con lo que se albergan todas las posibilidades: se ve si en cada posición, todos los números son posibles.

El más adecuado para este problema por tanto es el de vuelta atrás. Tenemos que descartar ramificación y poda ya que consiste en un recorrido del grafo en altura, llega hasta la última posición por cada "rama", camino que se hace y luego si se ve que no es correcto, se "poda", se descarta, ese camino. No es el adecuado en nuestro caso, porque no nos interesa llegar desde el principio al final del problema, sino ir pasando por cada posición para encontrar ya al final, la solución óptima.

### 3. Presentar y explicar el esquema algorítmico de exploración de grafos finalmente seleccionado, describiendo la funcionalidad de los distintos elementos del mismo (funciones auxiliares).

En la anterior pregunta hemos visto finalmente que el esquema elegido es el de **vuelta atrás** o “**backtracking**”. A continuación se presenta y se explica cómo funciona el algoritmo en esta práctica. Comentaremos paso a paso cada función además de el algoritmo en sí.

La implementación del esquema citado en código fuente Pascal es la siguiente (recordemos que se encuentra en el fichero entregado SURVO.PAS):

```

PROCEDURE Resuelve(sumafilas:TFilea;sumacols:TCol;numeros:TValores;VAR todos:TValores;VAR sudokusol:TTablero; VAR
sudoku:TTablero; fila,columna:TElemento;VAR exito:boolean);
VAR
    restantes:TValores;
    sumax,sumay,fsig,csig:TElemento;
    num:TElemento;
BEGIN
    IF(EsSolucion(sudoku,fila)) THEN
        exito:=TRUE
    ELSE BEGIN
        Copiar(restantes,numeros);
        WHILE(NOT(EsConjuntoVacio(restantes)) AND NOT(exito)) DO BEGIN
            DameNumero(restantes,num);
            QuitarNumero(restantes,num);
            CalcularSuma(sumafilas,sumacols,fila,columna,sumax,sumay);
            IF(EsFactible(sudoku,fila,columna,sumax,sumay,num) AND
NOT(EstaEnSudoku(sudoku,num))) THEN BEGIN
                InsertarEnSudoku(sudoku,num,fila,columna);
                QuitarNumero(todos,num);
                CalcularSiguietePos(sudoku,fila,columna,fsig,csig);
                CopiarSudoku(sudokusol,sudoku);
                Resuelve(sumafilas,sumacols,numeros,todos,sudokusol,sudoku,fsig,csig,exito);
                QuitarDeSudoku(sudoku,num);
                InsertarNumero(todos,num);
            END;
        END;
    END;
END;

```

Vamos a comentar a continuación cada función. Empezando con el propio procedimiento RESUELVE:

- **PROCEDURE** Resuelve(sumafilas:TFilea;sumacols:TCol;numeros:TValores;**VAR** todos:TValores;**VAR** sudokusol:TTablero; **VAR** sudoku:TTablero; fila,columna:TElemento;**VAR** exito:**boolean**);

Este procedimiento es el pedido en el enunciado expresamente para la resolución del problema. Contiene todos los elementos necesarios para realizar correctamente el algoritmo. En primer lugar vemos si la estructura sudoku que nos proporcionan es solución: lo veremos más adelante, pero es solución si ha sobrepasado ya las dimensiones del puzzle. Si no es solución, copiamos en cada llamada a RESUELVE un conjunto entero de todos los números disponibles para probar en cada posición todos los números. Comprobamos entonces que no es un conjunto vacío y que no se ha solucionado el puzzle. Entonces para cada número calculamos su correspondiente suma de fila y suma de columna, para comprobar su factibilidad y que por supuesto no esté puesto ya en el sudoku. Una vez hecho esto, lo insertamos, lo quitamos del conjunto de números que tenemos para probar las distintas combinaciones. Copiamos la solución

que tenemos hasta ahora del sudoku y llamamos otra vez a RESUELVE con otro conjunto de números y nueva posición. Si vuelve de esta llamada recursiva, el número se quita del sudoku y se añade de nuevo al conjunto.

- **FUNCTION** EsSolucion(sudoku:TTablero;fila:TElemento):**boolean**; {O(1)}

Esta función se encarga de dar por finalizado el algoritmo. Consiste en obtener la fila en la cual se encuentra el algoritmo RESUELVE: si estamos en una fila mayor que la dimensión del sudoku, entonces el algoritmo puede parar. En nuestro caso si es mayor que cuatro, finaliza. Esta función está en el módulo SURVO.PAS.

- **PROCEDURE** Copiar(**VAR** numeros1:TValores;numeros2:TValores); {O(1)}

Con esta función tenemos la posibilidad de copiar un conjunto de números disponibles para introducirlos en el puzzle en otro conjunto. Esto lo hacemos al principio del algoritmo RESUELVE para tener todos los números para cada posición, una copia auxiliar que nos facilita la resolución. Se encuentra en la unidad TNUMEROS.PAS.

- **FUNCTION** EsConjuntoVacio(numeros:TValores):**boolean**; {O(n)}

Función que nos dice simplemente que si el conjunto dado es vacío o no. Será vacío si la suma de todos sus elementos es cero. Lo encontramos en TNUMEROS.PAS.

- **PROCEDURE** DameNumero(**VAR** numeros:TValores; **VAR** valor:TElemento); {O(n)}

Aquí se procede a seleccionar un número del conjunto de números disponibles para ver después con otra función que explicaremos abajo si es factible y colocarlo en el puzzle. Recorremos el conjunto entero hasta encontrar una posición libre y después se modifica este conjunto, de ahí su complejidad. Lo encontramos en TNUMEROS.PAS.

- **PROCEDURE** QuitarNumero(**VAR** numeros:TValores; num:TElemento); {O(n)}

Con este procedimiento borramos del conjunto dado por la entrada y buscamos en todas las posiciones dónde puede estar el número. Utilizamos este procedimiento para quitar los números que ya han sido utilizados y para no volverlos a tocar. Lo tenemos en TNUMEROS.PAS.

- **PROCEDURE** CalcularSuma(sumafilas:Tfila;sumacols:TCol;fila,columna:TElemento;**VAR** sumax,sumay:TElemento); {O(1)}

Procedimiento que hace uso de los TAD de filas y columnas, los cuales comentaremos en las próximas hojas, y recibe como parámetros la fila y columna dónde buscar los números filas y columnas para encontrar la factibilidad de ese número en esa posición más tarde. Este procedimiento está en TSUMAS.PAS.

- **FUNCTION** EsFactible(sudoku:TTablero;fila,columna,sumax,sumay,num:TElemento):**boolean**; {O(1)}

Función estrella junto a RESUELVE en esta práctica. Con esta función nos garantizamos que un número a colocar en una posición en concreta sea factible o no, respetando el criterio de que debe sumar igual o menos que su respectiva suma de fila o columna. Una vez que pase ese valor, no será factible y por tanto, ese número para esa posición será descartado, ya que no entrará en la condición IF especificada. Se encuentra en SURVO.PAS.

- **FUNCTION** EstaEnSudoku(sudoku:TTablero;num:TElemento):**boolean**; {O(n<sup>2</sup>)}

Esta función nos ayuda a conocer si el número que intentamos poner está ya en el tablero del sudoku. Si está puesto, hemos de descartar la condición, sino, adelante. Su complejidad es alta debido a que tenemos que mirar el tipo de objeto sudoku cada fila y por cada fila, cada columna. Lo podemos encontrar en TSUDOKU.PAS.

- **PROCEDURE** InsertarEnSudoku(**VAR** sudoku:TTablero;num,fil,col:TElemento); {O(1)}

Entrando en la condición de factibilidad, tenemos este procedimiento que introduce el número dado del conjunto en el sudoku según una posición dada por fila y columna. Está en TSUDOKU.PAS.

- **PROCEDURE** CalcularSiguientePos(sudoku:TTablero;fila,columna:TElemento;**VAR** fsig,csig:TElemento); {O(1)}

Con este procedimiento sabemos la siguiente posición en la que corresponde colocar el siguiente valor. Si ya tenemos una posición marcada de antemano, por el archivo de entrada, esa posición la debemos de obviar y pasar a la siguiente, de ahí que debamos introducir por parámetro el sudoku inicial. El procedimiento está en TSUDOKU.PAS.

- **PROCEDURE** CopiarSudoku(**VAR** s1:TTablero;s2:TTablero); {O(1)}

Utilizamos este procedimiento para copiar el sudoku que estamos utilizando como prueba a uno definitivo que utilizaremos como solución final. Está en TSUDOKU.PAS.

Hasta aquí todos los procedimientos y funciones que se utilizan en el algoritmo mostrado. En las siguientes páginas hablaremos de las estructuras de datos utilizados y el por qué de su uso.

### **Particularización del algoritmo sobre el problema concreto**

El alumno deberá responder razonadamente a las siguientes cuestiones:

1. ¿Cuáles son las estructuras de datos más adecuadas para resolver el problema? Exponga y explique las estructuras de datos principales para resolver el problema, así como aquellas estructuras de datos auxiliares necesarias para almacenar la información de cálculos intermedios del algoritmo.

Las estructuras de datos más adecuadas para este problema pueden ser diferentes, según la elección del programador. En mi caso he escogido lo más sencillo, realizar conjuntos de datos estáticos, **arrays**. El caso óptimo sería utilizar una lista o cola dinámica, para mantener los datos que vayamos a utilizar bien ordenados y sin malgastar memoria del programa.

He utilizado las siguientes estructuras:

#### **TYPE {TADNumeros}**

TValores = **ARRAY** [1..N] **OF** TElemento;

Aquí almacenamos los números que nos serán necesarios para la resolución del problema. Los guardamos como conjuntos de tamaño N, el cual está fijado a la dimensión del puzzle, 12.

#### **TYPE {TADSumas}**

TFila = **ARRAY** [1..MAXFILA] **OF** TElemento;

TCol = **ARRAY** [1..MAXCOL] **OF** TElemento;

Con estas dos estructuras realizamos el proceso de almacenar las sumas tanto de las filas como de las columnas dadas por el fichero de entrada. MAXFILA y MAXCOL son las constantes que nos dan información acerca de la dimensión fila x columna del puzzle.

#### **TYPE {TADSudoku}**

TTablero= **ARRAY** [1..MAXFILA,1..MAXCOL] **OF** TElemento;

Esta es la estructura que nos facilita la colocación de los números y después verlos por pantalla. Tiene las dimensiones de fila y columna dadas en el TADSumas.

#### **TYPE {TADElemento}**

TElemento = **integer**;

Nos aseguramos que la estructura más elemental esté encapsulada para no permitir accesos ilegales.

**2,3 y 4. Realice un estudio teórico de la complejidad del algoritmo diseñado. Para ello deberá calcular y presentar separadamente el coste de todas y cada una de las funciones auxiliares implementadas en su algoritmo y, finalmente, calcular y presentar el coste total del algoritmo completo.**

En este apartado vamos a realizar los puntos del 2 al 4: se presentarán las funciones que acompañan a los TAD que se han utilizado y su correspondiente complejidad. Finalmente, calcularemos la complejidad final.

### {TADNumeros}

- **PROCEDURE** CrearConjuntoNumeros(**VAR** numeros:TValores); {O(n)}

Con este procedimiento inicializamos la estructura del conjunto de números. Complejidad O(n) porque recorremos todo el array para rellenarlo de ceros.

- **FUNCTION** EsConjuntoVacio(numeros:TValores):**boolean**; {O(n)}

Se comprueba si el conjunto no tiene ningún elemento. Complejidad O(n) porque debemos recorrer todo el array en el caso peor.

- **PROCEDURE** InsertarNumero(**VAR** numeros:TValores; num:TElemento); {O(n)}

Inserta un elemento en una posición libre. Caso peor, tenemos que recorrer todo el array, por lo que complejidad O(n).

- **PROCEDURE** QuitarNumero(**VAR** numeros:TValores; num:TElemento); {O(n)}

Al revés que la anterior, esta vez, eliminamos algún número. Caso peor, recorrer todo el array: O(n).

- **PROCEDURE** DameNumero(**VAR** numeros:TValores; **VAR** valor:TElemento); {O(n)}

Da el primer elemento del conjunto que esté libre. En el caso peor, recorremos de nuevo todo el array, por lo que complejidad O(n).

- **PROCEDURE** DameNumeroEnPosicion(**VAR** numeros:TValores; pos:TElemento; **VAR** valor:TElemento); {O(1)}

Asigna a una variable dada por la entrada el valor de un elemento en una posición en concreto. Complejidad simple, O(1),

- **PROCEDURE** Copiar(**VAR** numeros1:TValores; numeros2:TValores); {O(1)}

Copia un conjunto a otro. La complejidad es simple también.

### {TADSumas}

- **PROCEDURE** CrearSumaFilaVacio(**VAR** sumafilas:TFila); {O(n)}

- **PROCEDURE** CrearSumaColVacio(**VAR** sumacols:TCol); {O(n)}

Crean estructuras fila y columnas vacías, asignando valores nulos, recorriendo los arrays enteros. Así tienen complejidad O(n).

- **PROCEDURE** InsertarSumaFila(**VAR** sumafilas:TFila; num:TElemento); {O(n)}



- **PROCEDURE** InsertarSumaCol(**VAR** sumacols:TCol; num:TElemento); {O(n)}

Como su nombre propio indica, insertan en ambas estructuras respectivamente valores. Estos procedimientos se llaman desde la función de LeerFichero que está en SURVO.PAS. Recorren todo el array buscando posición vacías para colocar los valores, de ahí su complejidad.

- **PROCEDURE** DameSumaFila(sumafilas:TFila;fila:TElemento;**VAR** suma:TElemento); {O(1)}
- **PROCEDURE** DameSumaCol(sumacols:TCol;col:TElemento;**VAR** suma:TElemento); {O(1)}
- **PROCEDURE** CalcularSuma(sumafilas:TFila;sumacols:TCol;fila,columna:TElemento;**VAR** sumax,sumay:TElemento); {O(1)}

Los dos primeros procedimientos están incluidos en el tercero, que los agrupa. Simplemente guardan en una variable un elemento pedido justo en una posición indicada por fila o columna. La complejidad de todos es simple.

## {TADSudoku}

- **PROCEDURE** CrearSudokuVacio(**VAR** sudoku:TTablero); {O(n<sup>2</sup>)}

Inicializa la estructura del sudoku, asignando valores nulos en todas las posiciones. Se hace llamar desde el programa principal MAIN.PAS. Su complejidad es O(n<sup>2</sup>) debido a que debe recorrer un array por filas y columnas.

- **FUNCTION** DameFila(sudoku:TTablero):TElemento; {O(1)}

Función que se llama desde la EsSolucion para conocer la fila en la que se encuentra el algoritmo en ese momento. Su complejidad es simple, ya que es una asignación.

- **FUNCTION** EstaOcupado(sudoku:TTablero;fila,columna:TElemento):**boolean**; {O(1)}

Nos dice si el sudoku posee en la fila y columna especificada por parámetros un elemento, si está ocupado. Ya que es una búsqueda simple, es complejidad sencilla O(1).

- **FUNCTION** EstaEnSudoku(sudoku:TTablero;num:TElemento):**boolean**; {O(n<sup>2</sup>)}

Puede ser parecida a la anterior, pero esta función dice si un elemento en concreto está en el sudoku, no si una posición está ocupada. Al tener que recorrer todo el sudoku de fila a columna, es complejidad O(n<sup>2</sup>).

- **PROCEDURE** InsertarEnSudoku(**VAR** sudoku:TTablero;num,fila,col:TElemento); {O(1)}

Simplemente consiste en asignar un valor a una posición concreta del sudoku. Complejidad simple.

- **PROCEDURE** EncontrarSumaFila(sudoku:TTablero;fila:TElemento;**VAR** suma:TElemento); {O(n)}
- **PROCEDURE** EncontrarSumaColumna(sudoku:TTablero;columna:TElemento;**VAR** suma:TElemento); {O(n)}

Con estos dos procedimientos calculamos la suma actual de la fila y de la columna que viene referenciada por parámetro de entrada. Es complejidad O(n) ya que tenemos

que ir desde el principio de la fila o columna hasta el final en el caso peor.

- **PROCEDURE** QuitarComunes(**VAR** numeros:TValores;sudoku:TTablero); { $O(n^2)$ }

Este procedimiento nos ayuda a quitar del conjunto de valores dados para introducir, los que ya estén en el sudoku. Al recorrer todo el sudoku, es complejidad alta  $O(n^2)$ .

- **PROCEDURE** QuitarDeSudoku(**VAR** sudoku:TTablero;num:TElemento); { $O(n^2)$ }

Si un valor no vale, lo quitamos del sudoku. Tenemos que recorrer todo el sudoku en el caso peor para encontrarlo.

- **PROCEDURE** CalcularSiguietePos(sudoku:TTablero;fila,columna:TElemento;**VAR** fsig,csig:TElemento); { $O(1)$ }

Este procedimiento calcula la siguiente posición del sudoku para introducir otro número. Al ser una simple asignación, es complejidad sencilla.

- **PROCEDURE** CopiarSudoku(**VAR** s1:TTablero;s2:TTablero); { $O(1)$ }

Copiamos un sudoku a otro. Es una asignación por lo que  $O(1)$ .

- **PROCEDURE** MostrarSudoku(sudoku:TTablero;sumafilas:TFila;sumacols:TCol); { $O(n^2)$ }

Con este procedimiento podremos mostrar la solución obtenida tras las numerosas llamadas recursivas a RESUELVE. Recorre toda la solución, todo el sudoku para mostrar cada posición por pantalla. De ahí su complejidad  $O(n^2)$ .

## {TADElemento}

- **PROCEDURE** Asignar(**VAR** elemento1:TElemento;elemento2:TElemento); { $O(1)$ }
- **FUNCTION** Igualar(elemento1;elemento2:TElemento):**boolean**; { $O(1)$ }
- **PROCEDURE** Aumentar(**VAR** elemento1:TElemento;elemento2:TElemento); { $O(1)$ }

Los tres procedimientos que se han definido en el TAD auxiliar de elemento son simples asignaciones y comprobaciones, por lo que su complejidad es sencilla  $O(1)$ .

Como punto final a este apartado calcularemos la complejidad del algoritmo completo. Tenemos que tener en cuenta las posiciones existentes en el tablero y los números que hemos de coger. Si nos atendemos a lo que nos dice el enunciado, nos encontramos con un fichero de entrada en el que se nos dan fijos tres números en sus respectivas posiciones, por lo que sólo tendremos que posicionar  $12-3 = 9$  números o 9 posiciones según como queramos verlo. Así, el estudio de la complejidad se reduce a los siguientes cálculos:

tenemos nueve posiciones y nueve números a elegir. Por lo tanto, no podemos repetir número pero el orden sí importa, depende de que número hemos sacado, por lo que estamos ante una **permutación sin repetición**. Se calcula como el factorial de ese número; de esta manera, el orden de complejidad de nuestro algoritmo es factorial: de teoría sabemos que la complejidad es  $O(n)$  debido al caso recursivo.

El por qué es factorial es bien sencillo. Para nuestro sudoku, tenemos unos valores que van del 1 al 12. Si hemos quitado 3 desde el principio tenemos 9. Entonces, al comenzar a poner números, en cada iteración, hemos de quitar uno, por lo que:

$$9 \times (9-1) \times (9-2) \times \dots \times (9-8) = 9 \times 8 \times 7 \times \dots \times 1 = 9!$$

Entonces, en nuestro problema en concreto, el coste computacional se eleva a:

**$n! = 9! = 362880$  instrucciones o posibles permutaciones del sudoku**

Sin embargo, si desde el principio no se nos hubiese proporcionado ningún número para ponerlo en el sudoku, tendríamos 12 posibilidades cada vez, entonces:

**$n! = 12! = 479 \cdot 10^6$  instrucciones o posibles permutaciones del sudoku**

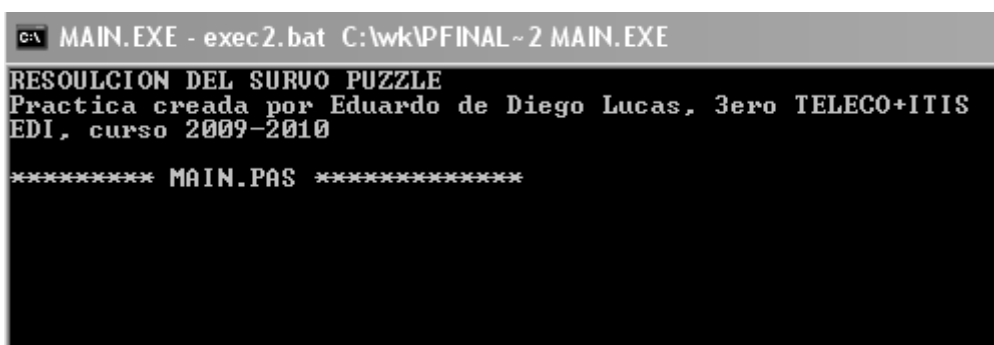
Vemos como la complejidad se reduce drásticamente un orden de magnitud por cada número facilitado. Con esto, podemos decir que para un sudoku de orden  $n$ , tendríamos  $n!$  Posibilidades. El coste se eleva mucho a partir de 12 como hemos comprobado.

## Pruebas, resultados y material de la práctica

### **Pruebas y resultados de la práctica**

En esta parte de la práctica comprobaremos el resultado que nos da la práctica tras ejecutar el programa principal, contenido en el archivo MAIN.EXE.

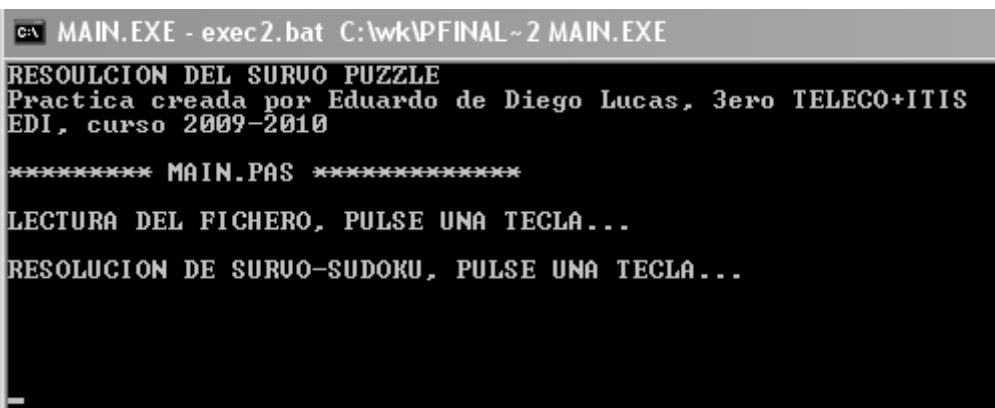
Iremos descubriendo a través de capturas de pantalla el comportamiento del programa:



```
C:\MAIN.EXE - exec2.bat C:\wk\PFINAL~2 MAIN.EXE
RESOLUCION DEL SURVO PUZZLE
Practica creada por Eduardo de Diego Lucas, 3ero TELECO+ITIS
EDI, curso 2009-2010
***** MAIN.PAS *****
```

*Figura 1: ejecución del programa MAIN.EXE. Muestra de la pantalla nada más iniciar.*

En esta primera figura vemos lo que muestra por pantalla el programa nada más ejecutar. Pulsamos una tecla ya que el programa está esperando y aparece lo siguiente:



```
C:\MAIN.EXE - exec2.bat C:\wk\PFINAL~2 MAIN.EXE
RESOLUCION DEL SURVO PUZZLE
Practica creada por Eduardo de Diego Lucas, 3ero TELECO+ITIS
EDI, curso 2009-2010
***** MAIN.PAS *****
LECTURA DEL FICHERO, PULSE UNA TECLA...
RESOLUCION DE SURVO-SUDOKU, PULSE UNA TECLA...
```

*Figura 2: ejecución del programa MAIN.EXE. Archivo leído y se procede a resolver el sudoku..*

Una vez pulsada la tecla, el programa ha leído ya el archivo de entrada por el que se reciben los datos del sudoku a resolver y procede a continuación a encontrar la solución óptima del problema dado con el archivo de entrada.

```

C:\ MAIN.EXE - exec2.bat C:\wk\PFINAL~2 MAIN.EXE
RESOLUCION DEL SURVO PUZZLE
Practica creada por Eduardo de Diego Lucas, 3ero TELECO+ITIS
EDI, curso 2009-2010

***** MAIN.PAS *****

LECTURA DEL FICHERO, PULSE UNA TECLA...
RESOLUCION DE SURVO-SUDOKU, PULSE UNA TECLA...

CONSEGUIDO! Hemos obtenido el siguiente survo puzzle resultado:

-----
  | A | B | C | D |
-----
1 | 12 | 6 | 2 | 10 | [30]
-----
2 | 8 | 1 | 5 | 4 | [18]
-----
3 | 7 | 9 | 3 | 11 | [30]
-----
  | 27 | 16 | 10 | 25 |
-----

***FIN DEL PROGRAMA***

```

Figura 3: ejecución del programa MAIN.EXE. Solución óptima obtenida.

En la figura de arriba podemos apreciar el resultado. El programa ha resuelto con éxito el problema que se proponía en el enunciado. Podemos hacer un zoom a la imagen para observar con más detalle la solución correcta del puzzle y su impresión por pantalla:

```

-----
  | A | B | C | D |
-----
1 | 12 | 6 | 2 | 10 | [30]
-----
2 | 8 | 1 | 5 | 4 | [18]
-----
3 | 7 | 9 | 3 | 11 | [30]
-----
  | 27 | 16 | 10 | 25 |
-----

```

Figura 4: ejecución del programa MAIN.EXE. Zoom del puzzle obtenido..

## **Material de la práctica**

Junto a esta memoria en papel se entrega un CD-ROM de datos donde se almacenan los archivos que son necesarios para ejecutar esta práctica obligatoria. Los archivos y su descripción son los siguientes:



**ENTRADA.TXT**  
Documento de texto  
1 KB

### **ENTRADA.TXT**

Este archivo contiene los datos que el programa principal debe leer para completar el survo-puzzle. Está en formato de texto plano ASCII.



**MAIN.PAS**  
Archivo PAS  
3 KB

### **MAIN.PAS**

Archivo que contiene todas las llamadas a las funciones que hacen capaz la obtención de la solución del sudoku. Aquí se inicializan las variables y se muestra la solución por pantalla.



**SURVO.PAS**  
Archivo PAS  
6 KB

### **SURVO.PAS**

Aquí tenemos todos los subprogramas que son útiles para la resolución del problema, a parte del procedimiento principal

RESUELVE.



**TELEM.PAS**  
Archivo PAS  
2 KB

### **TELEM.PAS**

En este archivo tenemos la estructura para el dato en forma de entero que utilizan las demás estructuras de datos



**TNUMEROS.PAS**  
Archivo PAS  
4 KB

### **TNUMEROS.PAS**

Este archivo contiene la estructura que guarda el conjunto de números para utilizar en la resolución del problema y sus respectivos procedimiento.



**TSUDOKU.PAS**  
Archivo PAS  
6 KB

### **TSUDOKU.PAS**

Con este archivo tenemos la estructura del puzzle-sudoku en forma de filas y columnas además de los correspondientes procedimientos para acceder y obtener de él ciertos elementos que ayudan a RESUELVE en SURVO.PAS.



**TSUMAS.PAS**  
Archivo PAS  
3 KB

### **TSUMAS.PAS**

Archivo que contiene las estructuras para almacenar las sumas de las filas y columnas del sudoku y sus correspondientes procedimientos para acceder a dichas estructuras.