

# PRACTICA 1

## Objetivo General

Desarrollar tres programas en C para reforzar el conocimiento de apunadores, cadenas de caracteres, vectores, matrices y matrices cubicas.

## Objetivos Particulares:

- El primer programa va a permitir reforzar el conocimiento sobre apunadores, matrices de cadenas de caracteres y vectores, y despertar la lógica para atacar un problema similar al que se aborde en el segundo examen.
- En el segundo examen se pretende reforzar el conocimiento sobre matrices, ya que pretende desarrollar un algoritmo que resuelva una versión simple de un sudoku de 9 x 9, mediante el algoritmo de Backtracking.
- En el tercer programa se pretende poner a prueba los conocimientos de matrices tridimensionales en C, mediante la generación de varios tableros sudoku de 9 x 9 de varias capas. Aquí se pretende que el programa genere de manera aleatoria cada tablero de sudoku y mencione si se puede resolver o no.

## Documentos para entregar

- Programa en C perfectamente documentado sobre las partes funcionales del programa.
- Informe a mano de como se abordó la problemática de la práctica en cuestión, las secciones que debe tener serán: Caratula, Introducción, Desarrollo y Conclusiones.
- Del punto anterior el desarrollo no recibirá informes con todo el código solo transcribirlo, necesitará diagramas, hojas borrador, pruebas de escritorio, etc.
- Recuerden que, si no se esfuerzan, pierden el miedo a equivocarse y finalmente aprender de ellos al reportarlos, jamás aprenderán a sentir un aprendizaje real.
- La práctica puede realizar de forma individual o en equipo de máximo 3 personas.

## Plazo de entrega

La hora y fecha límite para entregarla será el **martes 25 de febrero** con límite de entrega antes de que empiece la clase (**11:59:59 A.M.**). Los códigos se deben enviar al correo [opelo1209@gmail.com](mailto:opelo1209@gmail.com) y el informe se entrega en físico.

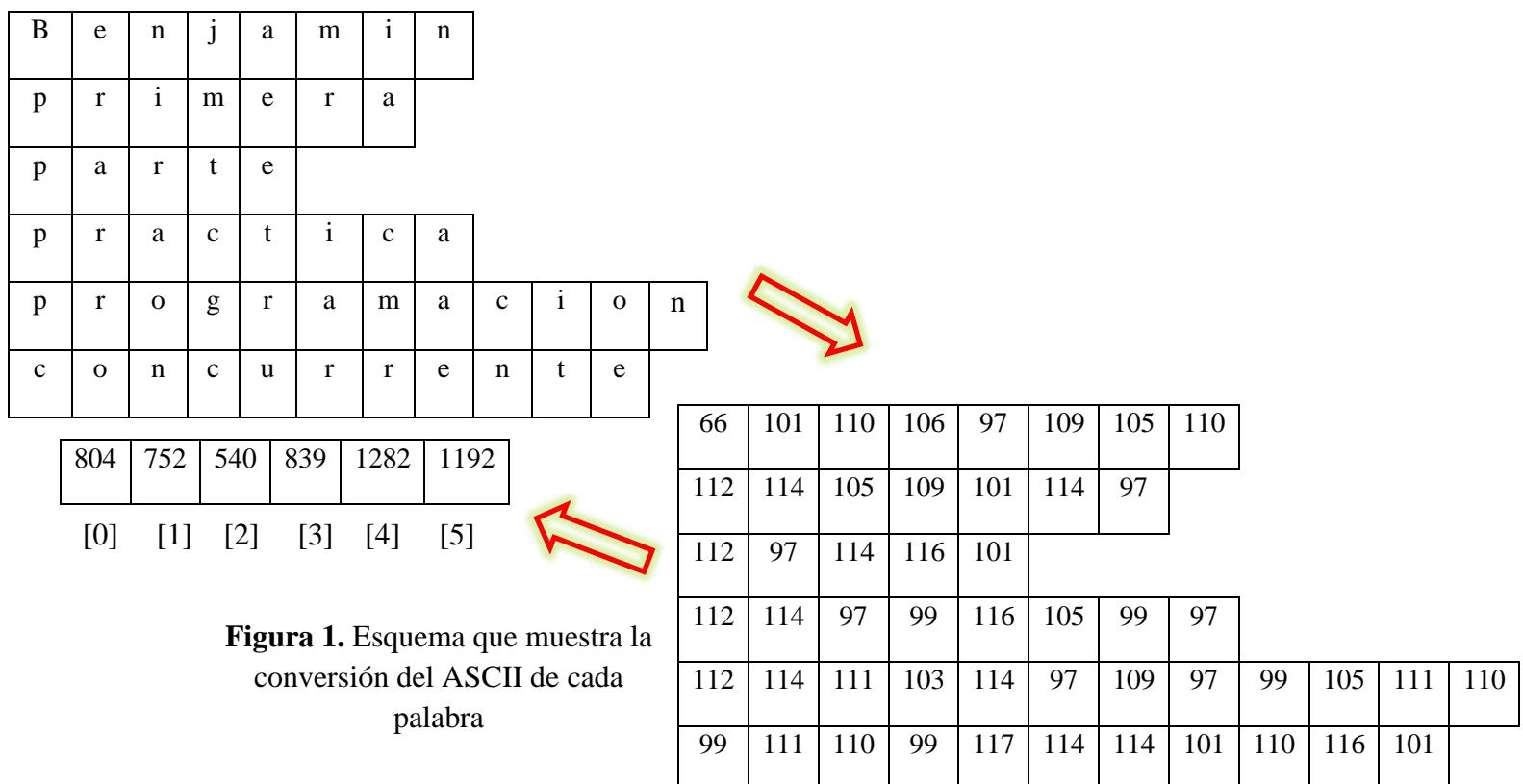
### Especificaciones del programa para entregar:

Primera parte - Ordenar Alfabéticamente una Cadena de caracteres e implementar en listas con vectores.

Los pasos que necesitan implementar son los siguientes:

1. Se pedirá cual es el número de cadenas que se quieran comparar posteriormente se le solicitara al usuario introducir cadenas por cadenas.
2. Una vez que se termine de introducir todas las cadenas se desplegará la lista de nombres en desorden. Para este punto es fácil deducir que se necesitarán una matriz de cadena de caracteres para que, al ingresar las cadenas, estas se guarden en la matriz. Se puede fijar el tamaño máximo de las palabras a máximo 15, usar `scanf("%15s", cadena);`
3. Una forma de ordenar las palabras es en orden alfabético de su primera letra, sin embargo, lo vamos a hacer más interesante, vamos a ordenar en base al código ASCII. En este punto cada palabra se debe transformar a su respectivo código ASCII, guardando el valor acumulativo en un vector de enteros, para entender mejor este punto, pueden visualizar el diagrama de la Figura 1.

Benjamin	primera	parte	practica	programacion	concurrente
[0]	[1]	[2]	[3]	[4]	[5]



**Figura 1.** Esquema que muestra la conversión del ASCII de cada palabra

4. Ordenar con el método de burbuja que se propuso para el examen de la Semana 2 el vector con la suma del ASCII de las palabras. Al mismo tiempo se puede ordenar la matriz de palabras.
5. Finalmente, se desplegará la lista de las palabras en orden de su código ASCII, tal y como muestra en la Figura 2.

```

opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_UAM/Trimestre_25_I/Materia - Programación Concurrente/Prácticas/Práctica1$ gcc ordenaPalabrasAscii.c -o ordenaPalabrasAscii
opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_UAM/Trimestre_25_I/Materia - Programación Concurrente/Prácticas/Práctica1$ ./ordenaPalabrasAscii
Dame el tamaño de las palabras que ingresarás???
6
Ingresa las palabras de longitud m de longitud máxima de 15
Ingresa la 1 palabra
Benjamin
Ingresa la 2 palabra
primera
Ingresa la 3 palabra
parte
Ingresa la 4 palabra
práctica
Ingresa la 5 palabra
programación
Ingresa la 6 palabra
concurrente
Las palabras ingresadas fueron: [ Benjamin ] -> [ primera ] -> [ parte ] -> [ práctica ] -> [ programación ] -> [ concurrente ] -> [ NULL ]
El vector con el ASCII de las palabras es: [ 804 ] -> [ 752 ] -> [ 540 ] -> [ 839 ] -> [ 1282 ] -> [ 1192 ] -> [ NULL ]
Palabras ordenadas por código ASCII: [ parte ] -> [ primera ] -> [ Benjamin ] -> [ práctica ] -> [ concurrente ] -> [ programación ] -> [ NULL ]
Vector ASCII después del ordenamiento: [ 540 ] -> [ 752 ] -> [ 804 ] -> [ 839 ] -> [ 1192 ] -> [ 1282 ] -> [ NULL ]
opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_UAM/Trimestre_25_I/Materia - Programación Con

```

**Figura 2.** Ejecución final de la primera parte de la práctica.

## 6. NOTAS FINALES

- a. La matriz de cadena de caracteres y el vector se debe hacer con memoria dinámica. Y paso de parámetros por referencia, al igual que se explicará en la clase del jueves 20 de febrero.
- b. Para evitar que se ingresen de forma incorrecta casillas de la matriz de cadenas de caracteres, se recomienda que se implemente el siguiente procedimiento para limpiar el buffer:

```

//Procedimiento para limpiar el buffer
void limpiarBuffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF); // Descarta caracteres restantes
}

```

Con este procedimiento al invocarlo después del scanf(),

```
scanf("%10s", Palabras[i]);
limpiarBuffer(); // Evita que la siguiente entrada se salte
```

Se podrán ingresar bien los elementos y se cortarán las palabras que sobrepasen los 15 caracteres como se muestra en la Figura 3.

```
oogramación Concurrente/Prácticas/Práctica1$ ./ordenaPalabrasAscii
Dame el tamaño de las palabras que ingresarás???
3
Ingresa las palabras de longitud m de longitud máxima de 10
Ingresa la 1 palabra
benjamínmoreno
Ingresa la 2 palabra
montiel
Ingresa la 3 palabra
palabra
Las palabras ingresadas fueron: [ benjamínmo ] -> [ montiel ] -> [ palabra ] -> [ N
ULL ]
```

**Figura 3.** Ingreso correcto de las palabras de 10 caracteres.

Segunda parte – Resolver un Sudoku sencillo con el algoritmo de Backtracking

Para esta parte se pretende resolver un tablero de Sudoku mediante un algoritmo de búsqueda informado llamado Backtracking, empecemos por revisar el tablero que pretende resolver en la Figura 4.

5	3			7					
6			1	9	5				
	9	8				6			
8			6				3		
4		8		3			1		
7			2				6		
	6				2	8			
		4	1	9			5		
			8		7	9			

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

**Figura 4.** Configuración inicial y tablero resuelto del Sudoku de prueba.

Como vimos en la Figura 4, se tiene un tablero de Sudoku de 9 x 9, recordar que las reglas son las siguientes:

1. En cualquiera de las filas  $i$ -esimas  $\forall i = 0, 1, 2, \dots, 9$ , deben estar presentes los dígitos del 1 al 9 solo una vez
2. En cualquiera de las columnas  $j$ -esimas  $\forall j = 0, 1, 2, \dots, 9$ , deben estar presentes los dígitos del 1 al 9 solo una vez
3. Se deben formar tableros de 3 x 3 (9 casillas), y en estos no deben existir repeticiones.

Para resolver un Sudoku en C, pueden utilizar un algoritmo de Backtracking, que es una técnica recursiva que intenta colocar números en cada celda y retrocede si se encuentra un error.

Aquí les dejo un algoritmo general para resolver un Sudoku:

1. Usaremos una matriz estática de 9x9 para almacenar el tablero de Sudoku, recordar que esta matriz se debe llenar con la configuración inicial que se propone en la Figura 4, con el siguiente código.

```
int sudoku[N][N] = {
    {5, 3, 0, 0, 7, 0, 0, 0, 0},
    {6, 0, 0, 1, 9, 5, 0, 0, 0},
    {0, 9, 8, 0, 0, 0, 0, 6, 0},
    {8, 0, 0, 0, 6, 0, 0, 0, 3},
    {4, 0, 0, 8, 0, 3, 0, 0, 1},
    {7, 0, 0, 0, 2, 0, 0, 0, 6},
    {0, 6, 0, 0, 0, 0, 2, 8, 0},
    {0, 0, 0, 4, 1, 9, 0, 0, 5},
    {0, 0, 0, 0, 8, 0, 0, 7, 9}
};
```

2. Imprimir este tablero inicial tal y como se muestra en la Figura 5.

```
opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_Concurrente/Practicas/Practical$ gcc sudoku.c -o sudoku
opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_Concurrente/Practicas/Practical$ ./sudoku
◆ Sudoku inicial:
 5 3 0 0 7 0 0 0 0
 6 0 0 1 9 5 0 0 0
 0 9 8 0 0 0 0 6 0
 8 0 0 0 6 0 0 0 3
 4 0 0 8 0 3 0 0 1
 7 0 0 0 2 0 0 0 6
 0 6 0 0 0 0 2 8 0
 0 0 0 4 1 9 0 0 5
 0 0 0 0 8 0 0 7 9
```

**Figura 5.** Despliegue del tablero inicial del Sudoku.

Podemos notar en la Figura 5 que se pone un punto azul muy diferente a la mayoría de los caracteres que hemos puesto en nuestros programas, bien esto representa un símbolo especial Unicode, en la Tabla 1 les dejo algunos que les podrán ser de utilidad.

Descripción	Carácter	Código Unicode	Código UTF-8
<b>Paloma (Check)</b>	✓	U+2705	\xE2\x9C\x85
<b>Paloma simple</b>	✓	U+2714	\xE2\x9C\x94
<b>Cruz (X)</b>	✗	U+274C	\xE2\x9D\x8C
<b>Cruz simple</b>	✗	U+2716	\xE2\x9C\x96
<b>Punto</b>	◆	U+1F539	\xF0\x9F\x94\xB9

**Tabla 1.** Lista de caracteres Unicode útiles.

Lo único que deben hacer para desplegar el símbolo especial es copiar la columna que dice Código UTF-9.

3. El tercer punto es implementar una función recursiva *resolverSudoku(sudoku)* de tipo entero. Esta función es el alma de la resolución del problema propuesto y debe ser implementada forzosamente para obtener calificación aprobatoria en esta parte de la práctica, por ello les doy el pseudocódigo para que lo transcriban en C.

```

FUNCION sesolverSudoku(matriz)
    DEFINIR fila, columna, encontrado COMO ENTERO
    encontrado ← FALSO

    // Buscar la primera celda vacía
    PARA fila ← 0 HASTA 8 HACER
        PARA columna ← 0 HASTA 8 HACER
            SI matriz[fila][columna] = 0 ENTONCES
                encontrado ← VERDADERO
                SALIR BUCLE
            FIN SI
        FIN PARA
        SI encontrado = VERDADERO ENTONCES
            SALIR BUCLE
        FIN SI
    FIN PARA

```

```
// Si no hay celdas vacías, el Sudoku está resuelto
SI encontrado = FALSO ENTONCES
    RETORNAR VERDADERO
FIN SI

// Intentar colocar números del 1 al 9
PARA numero ← 1 HASTA 9 HACER
    SI esSeguro(matriz, fila, columna, numero) ENTONCES
        matriz[fila][columna] ← numero

        // Intentar resolver el resto del Sudoku recursivamente
        SI resolverSudoku(matriz) ENTONCES
            RETORNAR VERDADERO
        FIN SI

        // Retroceder si no funciona (backtracking)
        matriz[fila][columna] ← 0
    FIN SI
FIN PARA

// Si no hay solución válida, retornar falso
RETORNAR FALSO
FIN FUNCION
```

4. En la función entera *esSeguro(matriz,fila,columna,numero)* se debe verificar las restricciones de Sudoku que son las siguientes:
  - a. Un número no debe repetirse en la misma fila.
  - b. Un número no debe repetirse en la misma columna.
  - c. Un número no debe repetirse en la subcuadrícula 3x3.
  - d. Cada una de estas es parte del código que se deja para que se implemente sin dejar de lado que son simples for's, con manejo re de return's, 0 si no es seguro y 1 si lo es.
5. Después de resolver el tablero Sudoku se debe mostrar la solución como se muestra en la Figura 6.

```
urrente/Practicas/Practical$ gcc sudoku.c -o sudoku  
opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_
```

urrente/Practicas/Practical\$ ./sudoku

◆ Sudoku inicial:

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	2	8	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

✓ Solución del Sudoku:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

**Figura 6.** Salida final que se espera de la segunda parte.

Tercera parte – Resolución de Sudoku tridimensional.

Para esta parte de la práctica se pretende resolver un Sudoku tridimensional de 5 capas, 9 filas y 9 columnas, estas últimas representando las dimensiones que se abordaron en la segunda aparte, las cuales pueden ser fijadas por un par de Macros o define's de C de la siguiente forma:

```
#define N 9 // Tamaño del Sudoku
#define Z 5 // Número de capas (tableros)
```

La lógica es muy similar a la que se trabajo en la segunda parte, con la diferencia que ahora los tableros se deben llenar de forma aleatoria, cosa que deben trabajar como un reto personal. Lo demás se puede reutilizar de la segunda parte, ya que al tener los tableros ahora se reduce en implementar un for que se mueva por las capas y resuelva cada tablero Sudoku, algo de la siguiente forma:

```
// Mostrar y resolver cada Sudoku
for (int z = 0; z < Z; z++) {
    printf("\n\xF0\x9F\x94\xB9 Sudoku #%d:\n", z + 1);
    imprimirSudoku(sudoku[z]);

    if (resolverSudoku(sudoku[z])) {
        printf("\n\xE2\x9C\x85 Solución:\n");
        imprimirSudoku(sudoku[z]);
    } else {
        printf("\n\xE2\x9D\x8C No tiene solución.\n");
    }
}
```

Del código anterior es bueno denotar que algunos tableros no podrán ser resueltos, por ello el mensaje de que no tiene solución. La ejecución de esta parte se presenta en la Figura 7.

```

opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_UAM/Trabajo_Concurrente/Practicas/Practica1$ gcc sudoku_3D.c -o sudoku_3D
opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_UAM/Trabajo_Concurrente/Practicas/Practica1$ ./sudoku_3D

◆ Sudoku #1:
0 3 0 0 0 0 5 0 0
0 0 0 0 0 0 9 6 0
0 0 1 0 0 0 0 0 0
0 0 0 0 0 8 0 0 0
0 0 5 0 0 0 0 6 0
0 0 0 0 0 0 0 0 0
0 1 0 8 0 0 4 0 5
0 0 0 0 2 0 0 0 0
0 0 7 0 0 3 0 0 0

✓ Solución:
2 3 4 1 6 9 5 7 8
5 7 8 2 3 4 9 6 1
6 9 1 5 8 7 2 3 4
1 2 3 6 4 8 7 5 9
4 8 5 7 9 2 6 1 3
7 6 9 3 1 5 8 4 2
3 1 2 8 7 6 4 9 5
9 5 6 4 2 1 3 8 7
8 4 7 9 5 3 1 2 6

◆ Sudoku #2:
0 0 0 0 0 0 0 3 8
3 7 6 0 0 0 0 4 0
0 9 1 0 0 0 0 0 7
0 0 0 0 5 2 0 0 0
0 0 0 0 0 0 0 0 0
4 0 0 9 0 0 0 0 0
2 0 0 0 1 0 0 0 0
0 0 0 7 9 0 0 0 0
0 0 0 5 0 0 0 0 0

✓ Solución:
5 2 4 1 6 7 9 3 8
3 7 6 2 8 9 1 4 5
8 9 1 3 4 5 2 6 7
1 3 7 4 5 2 6 8 9
9 5 2 6 3 8 4 7 1
4 6 8 9 7 1 5 2 3
2 4 5 8 1 3 7 9 6
6 1 3 7 9 4 8 5 2
7 8 9 5 2 6 3 1 4

◆ Sudoku #3:
0 8 0 0 0 4 0 0 0
6 0 0 0 0 0 0 0 0
4 0 3 8 0 0 0 0 0
0 0 0 0 0 0 0 0 2
0 0 2 0 0 9 0 0 0
9 0 7 1 4 0 0 0 0
0 0 0 4 0 0 0 0 3
0 0 8 0 0 1 0 6 0
0 0 0 0 7 0 0 0 0

✓ Solución:
1 8 5 2 3 4 6 7 9
6 2 9 7 1 5 3 4 8
4 7 3 8 9 6 1 2 5
5 1 4 6 8 3 7 9 2
8 6 2 5 7 9 4 3 1
9 3 7 1 4 2 5 8 6
7 5 6 4 2 8 9 1 3
3 4 8 9 5 1 2 6 7
2 9 1 3 6 7 8 5 4

```

```

◆ Sudoku #3:
0 8 0 0 0 4 0 0 0
6 0 0 0 0 0 0 0 0
4 0 3 8 0 0 0 0 0
0 0 0 0 0 0 0 0 2
0 0 2 0 0 9 0 0 0
9 0 7 1 4 0 0 0 0
0 0 0 4 0 0 0 0 3
0 0 8 0 0 1 0 6 0
0 0 0 0 0 7 0 0 0

✓ Solución:
1 8 5 2 3 4 6 7 9
6 2 9 7 1 5 3 4 8
4 7 3 8 9 6 1 2 5
5 1 4 6 8 3 7 9 2
8 6 2 5 7 9 4 3 1
9 3 7 1 4 2 5 8 6
7 5 6 4 2 8 9 1 3
3 4 8 9 5 1 2 6 7
2 9 1 3 6 7 8 5 4

◆ Sudoku #4:
0 0 5 0 0 0 0 0 9
0 0 4 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0
0 0 7 0 0 0 0 0 0
8 0 0 0 7 0 0 5 0
0 0 2 0 0 9 0 0 6
5 0 0 0 1 0 3 0 0
0 6 0 0 9 0 8 0 0
0 4 0 0 0 5 0 0 0

✓ Solución:
1 2 5 3 4 6 7 8 9
3 7 4 9 5 8 6 1 2
6 8 9 1 2 7 5 3 4
9 3 7 5 6 1 4 2 8
8 1 6 2 7 4 9 5 3
4 5 2 8 3 9 1 7 6
5 9 8 4 1 2 3 6 7
2 6 1 7 9 3 8 4 5
7 4 3 6 8 5 2 9 1

◆ Sudoku #5:
0 0 0 9 0 0 0 0 0
1 0 0 0 0 0 0 0 8
0 3 0 0 0 0 0 0 0
2 7 0 0 0 4 6 0 5
0 0 0 7 0 0 0 0 3
0 0 0 0 1 0 9 0 0
0 0 0 0 0 8 0 7 0
0 5 0 2 0 0 0 9 0
0 0 0 0 0 0 1 0 0

✓ Solución:
4 2 5 9 8 3 7 6 1
1 6 9 4 2 7 3 5 8
7 3 8 1 5 6 2 4 9
2 7 3 8 9 4 6 1 5
5 9 1 7 6 2 4 8 3
6 8 4 3 1 5 9 2 7
9 1 2 6 3 8 5 7 4
3 5 7 2 4 1 8 9 6
8 4 6 5 7 9 1 3 2
opelo@Abraxas-Omen2:/mnt/e/Documentos_Benja/Trabajo_UAM/Tr
urrente/Prácticas/Práctica1$ |

```

**Figura 7.** Ejecución de la tercera parte de la practica.