CS21 Machine Problem: Sudoku Solver

Carl David B. Ragunton Lab 2 2020-04243

Note: Please input the lines in the program one line at a time. My program cannot handle if the input is inserted all at once.

Link for the Video:

https://drive.google.com/file/d/1CRHYQtFF3cbtDNXcW42W-kCrjxVXzbln/view?usp=sharing

I. Summary

I will only provide a single summary since both of the solvers works the same way. The solvers utilize backtracking in solving the sudoku boards. It checks each element if it is a “0” and tries to solve it if it is a “0”. Otherwise, it goes to the next element. If it has already solved a “0” and the next element will not have an answer, it will go back to its previous state before solving the “0”. It will then try another answer for it until the board is solved.

Before explaining the function used, I will discuss how the input is stored and how it is represented in the program. The input is stored in .data line by line in ASCII format. Since .data is used, the first element will be stored at 0x10010000 and so on. For example, if the input line is 4020, it will be represented as 0x30323034 inside the data. Note that “0” is 0x30 in ASCII and the others have their respective representation as well. Also, when gathering the input, I used 32 bits for each line just to make it more understandable while working on it. The .data serves as our board and we can access the elements by using the correct address for each one (this will be explained more later). The board is stored as ASCII values, thus, it will be stored in a way that left and right are exchanged because MARS is in little endian. But this will not affect the final output since they will be printed in the way they should be.

There is only one function used in the program I made and it is sudoku( ). The sudoku( ) is the function called in the main. This is where each element is checked if it is a “0”. Also, the change in board also happens here. Before changing the “0” in the board, it first checks whether the current num is already in the current row, column or square. If it is already used on those places, the change will not be implemented and it will try to use the next possible num value. Meanwhile, if it is still not used, the current “0” in the board will be replaced by the current num. After that, the function will call itself to try to solve the next “0” in the board. The cycle will continue until the board is solved. Then, the deepest recursion will return 1 to the previous recursion. This will cause a domino effect on all the recursions made and they will end one after the other. Assuming that all the problems that will be given are solvable, the program already has solved board at the .data at this point. The program prints all the lines of the board and it will end.

Shortest Summary:

Get input-> call sudoku-> Check from left to right, top to bottom if element is “0”-> if not “0”, skip; if “0”, check if num is already used in the current row, column or square-> if num is already used, try the next num. If there are no num left, go back to previous state; if still not used, change the element in the board-> recurse sudoku-> wait till finished-> print output

II. Pseudocodes

The pseudocode I made is somewhat in the format of C combined with python language to make it easier for me to implement it on MIPS. Note that there is a more simplified version of it later. The "long" pseudocode will be the one that I will explain in the video to show its implementation on MIPS better.

Here is the long/detailed pseudocode for the 4x4 sudoku solver:

```
sudoku( ){
	sudoku_value=0
	for (row=0 to row<4):						//checks board from top to bottom
		for (col=0 to col<4):					//checks board from left to right
			if (board[row][col]==0x30):			//checks if current element is a "0"
				for (num=0x31 to num<0x35):		//tries to solve using "1" to "4"
					num_already_used =0

					for (ROW=0 to ROW<4):		//checks if num is already in the column
						if (num_already_used==1) break
						if (board[ROW][col]==num) num_already_used =1
					end for loop

					for (COL=0 to COL<4):		//checks if num is already in the row
						if (num_already_used==1) break
						if (board[row][COL]==num) num_already_used =1
					end for loop

					for (ROW =floor(row/2)*2 to ROW <floor(row/2)*2+2; ROW)://checks if num is already in the 2x2 square
						for (COL = floor(col/2)*2 to COL <floor(col/2)*2+2):
							if (num_already_used==1) break
							if (board[ROW][col]==num) num_already_used =1
						end for loop
					end for loop

					if (num_already_used ==0):
						board[row][col]=num		//changes current "0" to num
						sudoku_value=sudoku( )		//recurse to solve next "0"
						if (sudoku_value==1):		//domino effect to end all recursion
							sudoku_value=1
							return sudoku_value
						else: board[row][col]=0x30	//previous change is wrong; go back to last state

				end for loop
				sudoku_value=0				//no values of num can be used; mistake was made
				return sudoku_value

		end for loop
	end for loop
	sudoku_value=1							//finished to change all "0" in board
	return sudoku_value
}
```

Here is the long/detailed pseudocode for the 9x9 sudoku solver. It is almost the same as the 4x4 solver except for the values used.

```
sudoku( ){
	sudoku_value=0
	for (row=0 to row<9):                                   //checks board from top to bottom
		for (col=0 to col<9):                           //checks board from left to right
			if (board[row][col]==0x30):             //checks if current element is a "0"
				for (num=0x31 to num<0x3A):     //tries to solve using "1" to "9"
					num_already_used =0

					for (ROW=0 to ROW<9):   //checks if num is already in the column
						if (num_already_used==1) break
						if (board[ROW][col]==num) num_already_used =1
					end for loop

					for (COL=0 to COL<9):   //checks if num is already in the row
						if (num_already_used==1) break
						if (board[row][COL]==num) num_already_used =1
					end for loop

					for (ROW =floor(row/3)*3 to ROW <floor(row/3)*3+3; ROW)://checks if num is already in the 2x2 square
						for (COL = floor(col/3)*3 to COL <floor(col/3)*3+3):
							if (num_already_used==1) break
							if (board[ROW][col]==num) num_already_used =1
						end for loop
					end for loop

					if (num_already_used ==0):
						board[row][col]=num             //changes current "0" to num
						sudoku_value=sudoku( )          //recurse to solve next "0"
						if (sudoku_value==1):           //domino effect to end all recursion
							sudoku_value=1
							return sudoku_value
						else: board[row][col]=0x30      //previous change is wrong; go back to last state

				end for loop
				sudoku_value=0                  //no values of num can be used; mistake was made
				return sudoku_value

		end for loop
	end for loop
	sudoku_value=1                                          //finished to change all "0" in board
	return sudoku_value
}
```

Here is a more simplified version of the pseudocodes above but I decided to explain the more detailed one in the video because that will make it easier for me to explain how it is implemented as MIPS. That being said, they are still the same except that the pseudocodes above are more detailed than this version.

```
sudoku( ){
	FOR every element in board
		IF current element ==”0”:
			FOR all possible num:
				CHECK if current num is used in column
				CHECK if current num is used in row
				CHECK if current num is used in square
				IF current num not used:
					current element=current num
					CALL sudoku( ) to get sudoku_value
					IF sudoku_value==1: return sudoku_value=1
					ELSE: current element=”0”
			ENDFOR
			return sudoku_value=0
	ENDFOR
	return sudoku_value=1
```

There is also the pseudocode for the main which is just getting the input, calling the function, and printing the board. I did not elaborate in this part because MIPS has its own way of getting the inputs and printing the output.

```
main( ){
	input;			//create board in .data
	sudoku( );
	print;
}
```

Note that the board in this pseudocode stands for arrays inside an array in this format for the 4x4 solver:

board={(1,0,0,0),
	(0,1,0,0),
	(0,0,1,0),
	(0,0,0,1),}

But in MIPS, I stored it in the .data as:

	0	1	2	3
0	0x30	0x30	0x30	0x31
1	0x30	0x30	0x31	0x30
2	0x30	0x31	0x30	0x30
3	0x31	0x30	0x30	0x30

The address can be accessed using this formula: 0x10010000 + (r*32) + c

The format in the 9x9 solver is:
board={(1,0,0,0,0,0,0,0,0),
(0,1,0,0,0,0,0,0,0),
(0,0,1,0,0,0,0,0,0),
(0,0,0,1,0,0,0,0,0),
(0,0,0,0,1,0,0,0,0),
(0,0,0,0,0,1,0,0,0),
(0,0,0,0,0,0,1,0,0),
(0,0,0,0,0,0,0,1,0),
(0,0,0,0,0,0,0,0,1),}

But in MIPS, I stored it as:

	0	1	2	3	4	5	6	7	8
0	0x30	0x30	0x30	0x31	0x30	0x30	0x30	0x30	0x30
1	0x30	0x30	0x31	0x30	0x30	0x30	0x30	0x30	0x30
2	0x30	0x31	0x30	0x30	0x30	0x30	0x30	0x30	0x30
3	0x31	0x30	0x30	0x30	0x30	0x30	0x30	0x30	0x30
4	0x30	0x30	0x30	0x30	0x30	0x30	0x30	0x31	0x30
5	0x30	0x30	0x30	0x30	0x30	0x30	0x31	0x30	0x30
6	0x30	0x30	0x30	0x30	0x30	0x31	0x30	0x30	0x30
7	0x30	0x30	0x30	0x30	0x31	0x30	0x30	0x30	0x30
8	0x30	0x30	0x30	0x30	0x30	0x30	0x30	0x30	0x31

Again, 0x10010000 + (r*32) + c is used to find the address.

III. 4x4 Solver Explanation

```
.eqv    row $s0
.eqv    col $s1
.eqv    num $s2
.eqv    num_already_used $s3
.eqv    sudoku_value $s4
.eqv    ROW $s5
.eqv    COL $s6
```

This part is just renaming some registers to make them easier to work on during coding. Note that I only renamed s registers here because the t registers will not have a specified value on them. They will be used on all kinds of purpose, but I made sure that I changed their values each time they will be used. Notice that these renamed registers are the variables used in the pseudocode in addition to the value of the 2 functions.

```
.data
line1:  .space 32
line2:  .space 32
line3:  .space 32
line4:  .space 32
```

Here is the reserved space in the .data for the input of the sudoku board. line1 is for the 1st row, line2 is for the 2nd row, and so on.

```
.text
main:
        li $v0 8                        #getting input
        la $a0, line1
        li $a1, 5
        syscall

        la $a0, line2
        syscall

        la $a0, line3
        syscall

        la $a0, line4
        syscall
```

This is where we ask for the input from the user and store it to their respective location in the .data. As said in the beginning of this documentation, this program is only able to work if the input is one at a time. If the user decides to put the input all at ones, the program will only read the first line. In other words, only line1 in the .data will have its value changed. Also, the stored values are in ASCII value. After getting the input, the board is now ready to be solved in .data.

```
jal sudoku                      #calling sudoku(board)
```

This calls the function sudoku(board).

```
sudoku:                                 #sudoku( )
        #                               #preamble
        subu $sp, $sp, 32
        sw $ra, 28($sp)
        sw row, 24($sp)
        sw col, 20($sp)
        sw num, 16($sp)
        #

        li row, 0                       #resetting row
        li num, 0x31                    #num="1" in ASCII
        li sudoku_value, 0              #resetting sudoku_value
```

We save certain values to the stack so that we can access them again later when we need them again. Then, we reset the values shown above. This is for when the function is called again.

```
for_row:
        beq row, 4, for_row_x           #end of for_row; row=4
        li col, 0                       #resetting col

for_col:
        beq col, 4, for_col_x           #end of for_col; col=4

        li $t0, 32
        mul $t0, $t0, row
        add $t0, $t0, col               #$t0 = initial board[row][col]
        li $t1, 0x10010000
        add $t1, $t1, $t0               #$t1 = final board[row][col] address
        lb $t2, 0($t1)                  #$t2 = board[row][col] value

        beq $t2, 0x30, if1              #if board[row][col]=="0", start to solve it

        addi col, col, 1                #col++
        j for_col

for_col_x:
        addi row, row, 1                #row++
        j for_row

for_row_x:
        li sudoku_value, 0x1            #last "0" has been solved; will cause a domino effect on the other recursions
        jr $ra
```

Here are the first 2 for loops in the sudoku(board). The for_row label represents the 1st for loop or the loop that uses the row variable. In the same manner, the for_col label is the 2nd for loop and it uses the col variable. Let us first assume that the beq instructions will not work to show how this part works. Inside the for_row label, we set col to be 0; this is to reset c when for every time for_row is used. It will then enter for_col and again, assuming that the next beq instruction does not work, we proceed into finding board[row][col].

To find board[row][col], we use 0x10010000 + (r*32) + c. r in this case is row and c is col. For example, if row=0 and col=0, it will be equal to 0. It will be added to 0x10010000 since this is the start of the .data. If row=0 and col=2, the address of board[0][2] would be 0x10010002.

After getting the address, we check if it is a “0”. If it is, it would go to the next part. Again, let us say that it is not the case and the program will continue. There would be an increment in col and the for loop would run. We are now at the beginning of for_col. This cycle would continue until col=4 and the 1st beq instruction in for_col would run. Once that happens, it would branch to for_col_x which means that for_col has ended.

At this point, row will be incremented and will jump to for_row. This is the outer for loop and would continue until row=4 and the beq instruction on for_row would run. It would branch to for_row_x and would return to the $ra with sudoku_value=1 (this would be explained more later).

Now that we have discussed these for loops, let us now explain the 2nd beq in for_col. As said earlier, we check if board[row][col]==”0”. If it is not, it means that it does not need to be solved and it already has a value. On the other hand, if it is a “0”, the program must solve for its value and will branch to the label if1 which is the first if statement in the pseudocode.

```
if1:
        li num, 0x31                    #resetting num

for_num:
        beq num, 0x35, for_num_x        #end of for_num; num=0x35

NUM_ALREADY_USED:
        li ROW, 0x0                     #reset ROW
        li COL, 0x0                     #reset COL
        li num_already_used, 0          #num_already_used=0
```

The if1 label resets the value of num to 0x31 or “1”; this is for every time for_num would run. Let us assume again that the beq in for_num does not work. We would enter the label NUM_ALREADY_USED which as it name suggests, we are checking whether num is already used on the current row, column or 2x2 square. First comes resetting ROW and COL which are variables that are going to be used here. Also, we first assume that num is still not used so we set num_already_used to 0.

```
for_ROW:                            #checks if num is already in current column
        beq ROW, 4, for_ROW_x       #end of for_ROW; ROW=4

        li $t0, 32
        mul $t0, $t0, ROW
        add $t0, $t0, col           #$t0 = initial board[ROW][col]
        li $t1, 0x10010000
        add $t1, $t1, $t0           #$t1 = final board[ROW][col] address
        lb $t2, 0($t1)              #$t2 = board[ROW][col] value

        beq num_already_used, 1, num_already_used_true  #if num_already_used is 1; can't use this num anymore
        seq num_already_used, $t2, num

        addi ROW, ROW, 1            #ROW++
        j for_ROW

for_ROW_x:
        beq num_already_used, 1, num_already_used_true  #if num_already_used is 1; can't use this num anymore
```

We first check if num is already in the current column using for_ROW. Since we already know how the for loops I make works because we already had 2 of them earlier, we can skip some parts a little. This will traverse the entire current column since we are using the original col from earlier and the new ROW which is the only thing that changes here. For each element visited, we get its value and compare it to num. But before that, we check if num_already_used==1. This would always be false at first try because we assumed it was 0 at start. Here comes the comparing of the element and num and if they are the same we make num_already_used=1. Then, it would loop and if num_already_used is 1, it would branch to num_already_used_true since we already know that it is used. There is no need to continue it. But what if it we detect that num is already in the column on the last iteration, it would no longer go through the necessary beq instruction? It is okay because it will still undergo the beq in the checking of the next area.

```
for_COL:                                #checks if num is already in current row
        beq COL, 4, for_COL_x           #end of for_COL; COL=4

        li $t0, 32
        mul $t0, $t0, row
        add $t0, $t0, COL               #$t0 = initial board[row][COL]
        li $t1, 0x10010000
        add $t1, $t1, $t0               #$t1 = final board[row][COL] address
        lb $t2, 0($t1)                  #$t2 = board[row][COL] value

        beq num_already_used, 1, num_already_used_true  #if num_already_used is 1; can't use this num anymore
        seq num_already_used, $t2, num

        addi COL, COL, 1                #COL++
        j for_COL

for_COL_x:
```

This is exactly the same as the previous one except that we are checking if num is already in the current row.

```
for_COL_x:
        li $t0, 0x2
        div row, $t0
        mflo $t1
        mul $t1, $t1, 2
        addi $t6, $t1, 2                #$t6=floor(row/2)*2+2
        move ROW, $t1                   #ROW=floor(row/2)*2

for_ROW_square:                         #checks if num is already in current 2x2 square
        beq ROW, $t6, for_ROW_square_x  #end of outer loop; ROW=floor(row/2)*2+2

        li $t3, 0x2
        div col, $t3
        mflo $t4
        mul $t4, $t4, 2
        addi $t7, $t4, 2                #$t7=floor(col/2)*2+2
        move COL, $t4                   #COL=floor(col/2)*2

for_COL_square:
        li $t0, 32
        mul $t0, $t0, ROW
        add $t0, $t0, COL               #$t0 = initial board[ROW][COL]
        li $t1, 0x10010000
        add $t1, $t1, $t0               #$t1 = final board[ROW][COL] address
        lb $t2, 0($t1)                  #$t2 = board[ROW][COL] value

        beq num_already_used, 1, num_already_used_true  #if num_already_used is 1; can't use this num anymore
        seq num_already_used, $t2, num

        addi COL, COL, 1                #COL++
        j for_COL_square

for_COL_square_x:
        addi ROW, ROW, 1                #ROW++
        j for_ROW_square

for_ROW_square_x:
        beq num_already_used, 1, num_already_used_true  #if num_already_used is 1; can't use this num anymore
```

After checking the column and row, we are going to see if num is in the 2x2 square. To do this we use a nested loop, the process is still the same other than that. To get the appropriate 2x2 square, we get the floor of both row/2 and col/2. Then, multiply them by 2 to get their starting index. To get the boundary of the square, we just add 2 to what we have just got and we can now traverse the 2x2 square of the current element. Another difference here is that we add another of that beq instruction when the nested for loop finishes. The reason for this is that there is a chance that we will miss the opportunity of going to the appropriate label if the element becomes equal with the num at the last possible part of the square. After all, there is no longer a next part wherein we check for area unlike the previous ones.

Let us first assume that num_already_used=1. That means we can no longer use this num for the current "0" and we would go to num_already_used_true.

```
for_num:
        beq num, 0x35, for_num_x         #end of for_num; num=0x35
```

```
num_already_used_true:
        addi num, num, 1                 #num++
        j for_num
```

```
for_num_x:
        li sudoku_value, 0x0
        jr $ra                           #return 0
```

num will be incremented in the num_already_used_true and it will loop into for_num. If num=0x35 in ASCII or "5", it will end the for loop and will go to for_num_x. It will return to $ra with sudoku_value=0. This means that the previous change we made is invalid and we must go back to the state before we made that change.

Let us now assume that num_already_used =1. We can still use num to replace the "0" and it will go to num_already_used_false.

```
num_already_used_false:
        li $t0, 32
        mul $t0, $t0, row
        add $t0, $t0, col               #$t0 = initial board[row][col]
        li $t1, 0x10010000
        add $t1, $t1, $t0               #$t1 = final board[row][col] address

        sb num, 0($t1)                  #board[row][col] = num

        jal sudoku                      #recurse sudoku(board) to solve the next "0"

        #                               #retrieve values from previous recursion
        lw $ra, 28($sp)
        lw row, 24($sp)
        lw col, 20($sp)
        lw num, 16($sp)
        addu $sp, $sp, 32
        #

        beq sudoku_value, 0x1, sudoku_true#if sudoku is true

        li $t0, 32
        mul $t0, $t0, row
        add $t0, $t0, col               #$t0 = initial board[row][col]
        li $t1, 0x10010000
        add $t1, $t1, $t0               #$t1 = final board[row][col] address

        li $t3, 0x30
        sb $t3, 0($t1)                  #board[r][c] = "0"
```

How the address of the current element has been already explained earlier. The difference this time is that we don't load from it. Instead, we are saving num into that address. This is the part when we are making changes in the board. sudoku( ) will then recurse itself repeating every process we have discussed so far. Basically, once it "solves" a "0", it will go to the next "0" to try and solve it. If it can't be solved, it means that the previous change is wrong and will try another value on it. This will continue until the last element of the board.

But how does it go back to its previous state if a wrong change was made. First, after the recursion, we retrieve the values from the previous recursion. The program checks whether sudoku_value is 0 or 1. In the case that it is 1, it would go to sudoku_true but let us first assume otherwise. If it is 0, we would again get the address of board[row][col] and undo the change we did. We do this by saving 0x30 into it.

```
        li $t3, 0x30
        sb $t3, 0($t1)                  #board[r][c] = "0"

num_already_used_true:
        addi num, num, 1                #num++
        j for_num

sudoku_true:
        li sudoku_value, 0x1
        jr $ra                          #return 1

for_num_x:
        li sudoku_value, 0x0
        jr $ra                          #return 0
```

Note that after undoing the change, it will increment num and will go through the for loop again to continue the cycle.

This time, assume that sudoku_value is 1. Then, the beq instruction in num_already_used_false would work and will branch into sudoku_true. This would return sudoku_value=1 to the previous recursion of sudoku(board).

It means that once the sudoku_value becomes 1, it will be a domino effect to all the previous recursions. All their sudoku_value will also be 1 and will return to the original sudoku( ) call in the main label. But when does the sudoku_value becomes 1?

```
for_row_x:
        li sudoku_value, 0x1            #last "0" has been solved; will cause a domino effect on the other recursions
        jr $ra
```

This is where the sudoku_value becomes 1 first. Once every element in the board is already checked and there are no “0” detected in it anymore. With this explanation, the Shortest Summary at the start should make sense. These explanations should make more sense in the video.

IV. 9x9 Solver Explanation

The process of the 9x9 solver is almost the same as the 4x4 solver. The only difference is some values used, the input, and the output.

```
.data
line1:  .space 32
line2:  .space 32
line3:  .space 32
line4:  .space 32
line5:  .space 32
line6:  .space 32
line7:  .space 32
line8:  .space 32
line9:  .space 32
```

Unlike in 4x4 solver, there are 9 lines reserved in the .data in 9x9 solver.

```
.text
main:
        li $v0 8                        #getting input
        la $a0, line1
        li $a1, 10
        syscall

        la $a0, line2
        syscall

        la $a0, line3
        syscall

        la $a0, line4
        syscall

        la $a0, line5
        syscall

        la $a0, line6
        syscall

        la $a0, line7
        syscall

        la $a0, line8
        syscall

        la $a0, line9
        syscall
```

Of course, the program asks for the input 9 times.

```
for_row:
        beq row, 9, for_row_x           #end of for_row; row=9
        li col, 0                       #resetting col

for_col:
        beq col, 9, for_col_x           #end of for_col; col=9

for_num:
        beq num, 0x3A, for_num_x        #end of for_num; num=0x3A

for_ROW:                                #checks if num is already in current column
        beq ROW, 9, for_ROW_x           #end of for_ROW; ROW=9
```

```
for_COL:                                #checks if num is already in current row
        beq COL, 9, for_COL_x           #end of for_COL; COL=9

for_COL_x:
        li $t0, 0x3
        div row, $t0
        mflo $t1
        mul $t1, $t1, 3
        addi $t6, $t1, 3                #$t6=floor(row/3)*3+3
        move ROW, $t1                   #ROW=floor(row/3)*3

for_ROW_square:                         #checks if num is already in current 3x3 square
        beq ROW, $t6, for_ROW_square_x  #end of outer loop; ROW=floor(row/3)*3+3

        li $t3, 0x3
        div col, $t3
        mflo $t4
        mul $t4, $t4, 3
        addi $t7, $t4, 3                #$t7=floor(col/3)*3+3
        move COL, $t4                   #COL=floor(col/3)*3
```

Several values used in for loops are also changed in order to traverse each element in the 9x9 board. The nested loop in the ok function now checks 3x3 square area instead of 2x2 square. The other conditions were change mostly from row=4 to row=9, col=4 to col=9, ROW=4 to ROW=9, COL=4 to COL=9, num=0x31 to num=0x3A.

```
li $a0, 10                  #printing board    li $a0, 10        li $a0, 10
li $v0, 11                                     li $v0, 11        li $v0, 11
syscall                                        syscall           syscall

li $v0, 4                                      li $v0, 4         li $v0, 4
la $a0, line1                                  la $a0, line4     la $a0, line7
syscall                                        syscall           syscall

li $a0, 10                                     li $a0, 10        li $a0, 10
li $v0, 11                                     li $v0, 11        li $v0, 11
syscall                                        syscall           syscall

li $v0, 4                                      li $v0, 4         li $v0, 4
la $a0, line2                                  la $a0, line5     la $a0, line8
syscall                                        syscall           syscall

li $a0, 10                                     li $a0, 10        li $a0, 10
li $v0, 11                                     li $v0, 11        li $v0, 11
syscall                                        syscall           syscall

li $v0, 4                                      li $v0, 4         li $v0, 4
la $a0, line3                                  la $a0, line6     la $a0, line9
syscall                                        syscall           syscall
```

There are also 9 printing process now. These are all the changes made in the 9x9 solver. All the other parts stay the same and work the same as before.

V. Test Cases

I will give 5 test cases for each of the solvers.
Here are the test cases and the results for the 4x4 solver from an online sudoku generator (https://www.sudokuweb.org/):

Input 1:
3042
0410
4200
0004

Output 1:
3142
2413
4231
1324

Input 2:
0002
0213
0401
1300

Output 2:
3142
4213
2431
1324

Input 3:
3100
0430
1320
4000

Output 3:
3142
2431
1324
4213

Input 4:
0003
1004
4132
0240

Output 4:
2413
1324
4132
3241

Input 5:
0000
4030
0102
0000

Output 5:
1324
4231
3142
2413

Here are the test cases and the results for the 9x9 solver created in an online 9x9 sudoku generator (https://www.sudoku9x9.com/):

Input 1: (the program runs slow in this input, but it works)
000001070
060080001
100900300
031500006
000000040
400800007
080060000
049000210
500034000

Output 1:
893621475
765483921
124975368
231547896
978316542
456892137
382169754
649758213
517234689

Input 2:
007060001
236000080
908500006
092100063
001005890
700000052
600010048
100806200
070409015

Output 2:
457968321
236741589
918532476
592184763
361275894
784693152
625317948
149856237
873429615

Input 3:
000001000
004327000
005000800
100006900
900002050
703000002
070000043
060000000
300190700

Output 3:
637581429
894327516
215649837
152836974
986472351
743915682
579268143
461753298
328194765

Input 4: (the program runs slow in this input, but it works)
030000000
040000090
092005006
070200050
000300008
005009042
006030000
001070000
000801409

Output 4:
538697214
647123895
192485736
379248651
214356978
865719342
926534187
481972563
753861429

Input 5:
400000030
060090000
020407005
200073040
038560100
000000007
000000080
005300000
000120009

Output 5:
417652938
563891274
829437615
251973846
738564192
946218357
192746583
675389421
384125769