

# CS 21: Machine Problem

## Sudoku Solver

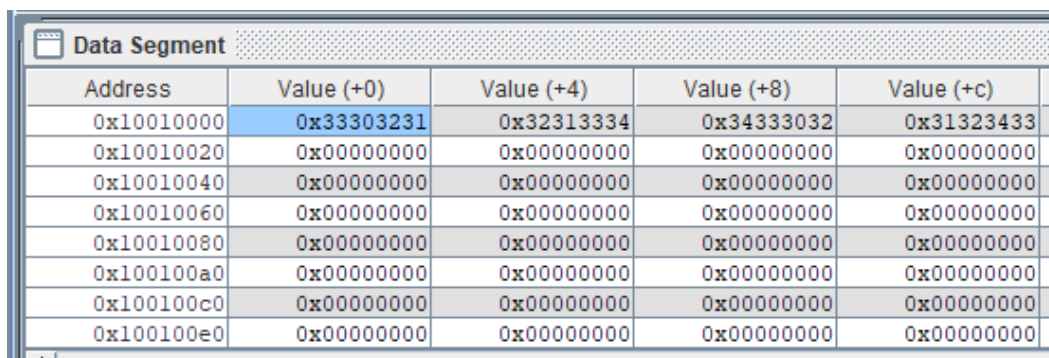
Jaimielle Kyle C. Calderon

2020-05499

## How the solvers work

The task is to solve a given 4X4 (and 9X9) sudoku grid, with the inputs being 4 (9) lines each with 4 (9) digits, and blank spaces are represented by 0s.

The first thing the solvers must do is get the input from the user and store them in a way such that they can be easily accessed by column and by row. The solver uses syscall 8 to read string input, the maximum number of characters is set to 6 for the 4X4 grid (11 for 9X9). **Each line in the input is stored in a different address in the memory**, note that these addresses are contiguous and each buffer size is just 4 (9) so that the value of “\n” wouldn’t be stored. Note that each cell/digit in the grid takes up a byte of storage.



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x33303231	0x32313334	0x34333032	0x31323433
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000

All the cells are stored in a single “array” with their indices given by:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

To access each cell in the array, the solver uses the address of the first buffer (first row) + offset (index). The algorithm used by the solver uses nested loops that requires the indexing to be easily changed by row and by column  $\rightarrow$  `grid[row][column]` (matrix indexing in higher-level languages). Relating this to the indexing above, the solver uses  $\text{index} = 4 * \text{row} + \text{column}$  (this functions the same as `grid[row][column]`). Once the whole grid is stored, the individual chars are converted into ints. Now the solver can call the functions needed in order to complete the grid.

The functions will be explained in more detail in the latter parts of this document, but in a nutshell there are 3 functions: (1) main function, (2) function for checking whether a number is already in a column, a row, or a 2X2 (3X3) division, (3) the function for solving the grid, where the backtracking happens.

Once the solver is done with reading the input, converting it to int, and completing the 4X4 (9X9) grid, the program goes back to main and then the finished grid is printed per line.

The program also utilizes macros for the syscalls (reading input, printing output) and for converting chars to ints.

```

.macro do_syscall(%n)
    li    $v0, %n
    syscall
.end_macro

.macro input_line(%buffer)
    la    $a0, %buffer           # Store here
    li    $a1, 6
    do_syscall(8)                # Read string input
.end_macro

.macro convert_int(%n)
    lb    $t0, buffer1(%n)
    subu  $t0, $t0, 48           # Subtract 48 from ASCII value to convert to int digit
    sb    $t0, buffer1(%n)      # Store the int
.end_macro

.macro printgrid(%n)
    lb    $a0, buffer1(%n)
    do_syscall(1)               # Print int
.end_macro

```

## Algorithm

The algorithm for this problem involves backtracking to arrive at a solution. So, it checks possible paths then goes back one step if the path doesn't lead to the correct solution of the sudoku board.

```

S = 4      // size of the sudoku board
N = sqrt(S)
// Grid contains the board (array)

Checker(r, c, x)    // r: row, c: col, x: number to check
{
    // Checking if there are similar values to x in the corresponding square
    for(i values in range 0 to N, excluding N)
        for(j values in range 0 to N, excluding N)
            if(Grid[(S*(i+(r-r%N)))+(j+(c-c%N))]) is equal to x) then x is invalid -> return 0

    // Checking corresponding row
    for(i values in range 0 to S, excluding S)
        if(Grid[(S * r)+i] is equal to x) then x is invalid -> return 0

    // Checking corresponding column
    for(i values in range 0 to S, excluding S)
        if(Grid[(S * i)+c] is equal to x) then x is invalid -> return 0

    if(function hasn't returned 0 at this point) then x is valid -> return 1
}

```

```

Solver(r, c)
{
    if(r is equal to S-1 and c is equal to S) then base case is reached -> return 1

    if(c is equal to S) then go to next row:
        c = 0
        r = r + 1

    if(Grid[(S * r)+c] is not 0) then go to next cell -> return Solver(r, c + 1)

    // Checking possible values to input to the grid
    for(x values in range 1 to S, including S)
        if(Checker(r,c,x) return value is 1) then x is valid:
            Grid[(S * r)+c] = x
            if(Solver(r,c+1) return value is 1) then current grid is correct -> return 1
            else then need to backtrack:
                Grid[(S * r)+c] = 0
            else make sure cell is empty:
                Grid[(S * r)+c] = 0

    if(function hasn't returned 1 at this point) then current grid is invalid -> return 0
}

```

In the MIPS program, solver is called first inside main after appropriate steps were taken to store the correct digits in the data segment, and checker is called inside solver. Both functions have lines as preamble and end to the functions, their main purpose is to save specific registers, this is necessary especially since the function solver recursively calls itself. The jump instruction is also used heavily in the code, for the loops. Operations such as adding and multiplying are pretty straight forward, they use the corresponding MIPS instructions. If statements use bne, beqz, and such to check cases. And accessing elements of the sudoku board uses the lb instruction with the address of the first buffer (first row) and the offset as the index of the specific element. The empty (0) cells are “changed” by using sw to store the new digit into the grid.

## Functions in the 4x4 Solver



Note that the each cell entry in the grid is accessed from memory using load byte, with the address of the first buffer plus an offset. The offset is the index of the particular cell, and the index is taken from:  $\text{index} = 4 * \text{row} + \text{column}$ . For brevity, in the following sections we refer to `Grid[(4 * r)+c]` as `Grid[r][c]`.

## 1. main

This is the first function in the solver. This is where the input is read using syscall 8, then converted to int, a macro is used for converting these and it is called inside a loop w/c terminates when all the chars are already converted to int. After the conversion, **solver** is called which will start the process of solving the actual sudoku board.

When solver is finished executing, the array for the grid is already modified (solved). The program will move on to printing the grid. To print the grid, the program has two nested loops in order to print 4 integers in each of the 4 lines and making sure to add a newline so that the output will also be composed of 4 lines; syscall 4 is used to print each digit. Each digit is accessed and printed using address of first buffer + index.

After printing the grid, the program will then terminate.

## 2. checker

The purpose of this function is to check whether a number is a valid entry in a specific cell in the board; it returns 1 or 0 — 1 if it is valid, 0 if not. It takes as input the specific number (x), the row (r) and column (c) where it will be potentially placed. It has 3 loops:

### a. square

This checks the 2X2 grid, it uses 2 nested loops. The first loop is `i` from 0 to 2 (will exit the loop when `i=2`) and the inner loop is `j` from 0 to 2 (will exit the loop when `j=2`) and it checks `if (Grid[i+ (r - r%2)][j+(c - c%2)] == x)`. The formula on the index makes sure that it is checking the 2X2 grid associated with the r and c passed as arguments of the function. For example, if `r=0` and `c=1` then this loop will check:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

If the condition in the if statement is true, then the function will return 0, meaning x is not a valid entry to the cell because there's a same digit in the 2X2 grid.

b. row

This checks the row (r input), it uses a loop to look at each cell in the row, the loop is `i` from 0 to 4 (will exit when `i=4`). The condition it checks is whether `grid[r][i] == x` and it will return 0 if this condition is met because that means x is not a valid entry. To visualize, if `r=2`, `c=0` the loop will check:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

c. column

This checks the column (c) and checks each cell in that column. It also uses a loop `i` from 0 to 4 which will terminate when `i=4`. It checks if `grid[i][c]` is equal to the x input to the function. For example, if `c=3`, `r=2` then the loop will look at the following cells:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The function will return 0 if the condition is met, meaning the loop found a cell that already contains x, hence, x is not a valid entry to the board.

After checking, if there are no entries equal to x in the square, the row, and the column associated to where it will be placed, then the function will return 1 — x is a valid entry to the grid.

### 3. solver

This function is where the magic happens, it recursively calls itself until the base case is reached (when the grid is completed). It also returns 1 or 0. The inputs are the row (r) and column (c) of the “current” cell. This function checks 3 cases first before it goes on to trying out different values of x to put into the grid.

- a. check if base case is reached

The base case is reached when `r == 3 && c == 4` because when this is met, this means the grid is already completed.

- b. check if program should move on to the next row

This checks whether `c == 4`, when c is already 4 but r is not equal to 3, this means the solver must move on to the next row to check the next cell, so we

set `c=0` and `r+=1` and move on to the next parts of the function.

c. check if current cell is empty (equal to 0)

This checks if `grid[r][c]==0`, if this condition is true, then the program will move on to the next part of the function, however if the condition is not met, then that means the current cell is already occupied by a digit, so the program needs to move on to the next cell. Hence, `solver` is called again, this time with `r=current r`, `c=c+1`.

After checking the cases above, we can then move on to the loop that checks for different values of `x`. The loop starts with `x=1` and terminates when `x=5`. This means, for each digit in `{1, 2, 3, 4}` it calls the function **checker** (with the arguments set to: `r=current r value`, `c=current c value`, `x=current x value`) to see whether the digit is a valid entry to the board. If it is a valid entry (if checker will return 1), then the empty (0) cell will be changed to the current value of `x`, then the next step is to call **solver** again to check for the next cell (with `r=current r`, `c=c+1`). If this call to `solver` returns 0, then that means the board wouldn't be solved with the current addition to the board, so we backtrack and set the cell to zero again and then check for the next value of `x` in the loop.

However, if the call for checker returns 0, then the program makes sure that the current cell is 0 by setting `grid[r][c]=0`. Since the current value of `x` is not a valid entry so the program will check for the next value of `x`. If `x` reaches 5 then that means the current board will not be solved, so the function returns 0. This return of zero happens when `solver` calls itself (we are assuming inputs from the user are valid/solvable grids), when it is zero then that means the program needs to “undo” some steps that it took, and take a different path (try the next possible value).

These are the steps taken in the function **solver**, the program will repeatedly do this until it finds a path (the correct combination of entries in the board) that will solve/complete the grid. And it will stop calling itself recursively when the base case is reached. The program returns to the main function because it is now ready to print the output.

---

## Sample test cases for the 4x4 Solver





First 4 lines is the given board (manual input/not fed through file), the next 4 lines is the solved board.

```
C:\Users\Kyle\Downloads>java -jar Mars45.jar sm nc 202005499_4.asm
0000
4302
2004
0000
1243
4312
2134
3421
```

```
C:\Users\Kyle\Downloads>java -jar Mars45.jar sm nc 202005499_4.asm
0010
3000
0300
0004
2413
3142
4321
1234
```

```
C:\Users\Kyle\Downloads>java -jar Mars45.jar sm nc 202005499_4.asm
0200
0020
0003
4000
1234
3421
2143
4312
```

Sudoku grids from [http://www.sudoku-download.net/sudoku\\_4x4.php](http://www.sudoku-download.net/sudoku_4x4.php)

## Functions in the 9x9 Solver

The 9X9 solver is almost the same as the 4X4 solver with only little changes.



Note that the each cell entry in the grid is accessed from memory using load byte, with the address of the first buffer plus an offset. The offset is the index of the particular cell, and the index is taken from:  $\text{index} = 9 * \text{row} + \text{column}$ . For brevity, in the following sections we refer to `Grid[(9 * r)+c]` as `Grid[r][c]`.

To visualize, the indexing for the 9X9 grid is:

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

## 1. main

This is the first function in the solver. This is where the input is read using syscall 8, then converted to int, a macro is used for converting these and it is called inside a loop w/c terminates when all the chars are already converted to int. After the conversion, **solver** is called which will start the process of solving the actual sudoku board.

When solver is finished executing, the array for the grid is already modified (solved). The program will move on to printing the grid. To print the grid, the program has two nested loops in order to print 9 integers in each of the 9 lines and making sure to add a newline so that the output will also be composed of 9 lines; syscall 4 is used

to print each digit. Each digit is accessed and printed using address of first buffer + index.

After printing the grid, the program will then terminate.

## 2. checker

The purpose of this function is to check whether a number is a valid entry in a specific cell in the board; it returns 1 or 0 — 1 if it is valid, 0 if not. It takes as input the specific number (x), the row (r) and column (c) where it will be potentially placed. It has 3 loops:

### a. square

This checks the 3X3 grid, it uses 2 nested loops. The first loop is `i` from 0 to 3 (will exit the loop when `i=3`) and the inner loop is `j` from 0 to 3 (will exit the loop when `j=3`) and it checks `if (Grid[i+ (r - r%3)][j+(c - c%3)] == x)`. The formula on the index makes sure that it is checking the 3X3 grid associated with the r and c passed as arguments of the function. For example, if `r=4` and `c=1` then this loop will check:

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

If the condition in the if statement is true, then the function will return 0, meaning x is not a valid entry to the cell because there's a same digit in the 3X3 grid.

### b. row

This checks the row (r input), it uses a loop to look at each cell in the row, the loop is `i` from 0 to 9 (will exit when `i=9`). The condition it checks is whether `grid[r][i] == x` and it will return 0 if this condition is met because that means x is not a valid entry. To visualize, if `r=5`, `c=0` the loop will check:

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

### c. column

This checks the column (c) and checks each cell in that column. It also uses a loop `i` from 0 to 9 which will terminate when `i=9`. It checks if `grid[i][c]` is equal to the x input to the function. For example, if `c=1`, `r=2` then the loop will look at the cells:

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

The function will return 0 if the condition is met, meaning the loop found a cell that already contains x, hence, x is not a valid entry to the board.

After checking, if there are no entries equal to x in the square, the row, and the column associated to where it will be placed, then the function will return 1 — x is a valid entry to the grid.

### 3. solver

This function is where the magic happens, it recursively calls itself until the base case is reached (when the grid is completed). It returns 1 or 0. The inputs are the row (r) and column (c) of the “current” cell. This function checks 3 cases first before it goes on to trying out different values of x to put into the grid.

#### a. check if base case is reached

The base case is reached when `r == 8 && c == 9` because when this is met, this means the grid is already completed.

#### b. check if program should move on to the next row

This checks whether `c == 9`, when c is already 9 but r is not equal to 8, this means the solver must move on to the next row to check the next cell, so we set `c=0` and `r+=1` and move on to the next parts of the function.

#### c. check if current cell is empty (equal to 0)

This checks if `grid[r][c]==0`, if this condition is true, then the program will move on to the next part of the function, however if the condition is not met, then that means the current cell is already occupied by a digit, so the program needs to move on to the next cell. Hence, solver is called again, this time with  $r=\text{current } r$ ,  $c=c+1$ .

After checking the cases above, we can then move on to the loop that checks for different values of  $x$ . The loop starts with `x=1` and terminates when `x=10`. This means, for each digit in  $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$  it calls the function **checker** (with the arguments set to:  $r=\text{current } r$  value,  $c=\text{current } c$  value,  $x=\text{current } x$  value) to see whether the digit is a valid entry to the board. If it is a valid entry (if checker will return 1), then the empty (0) cell will be changed to the current value of  $x$ , then the next step is to call **solver** again to check for the next cell (with  $r=\text{current } r$ ,  $c=c+1$ ). If this call to solver returns 0, then that means the board wouldn't be solved with the current addition to the board, so we backtrack and set the cell to zero again and then check for the next value of  $x$  in the loop.

However, if the call for checker returns 0, then the program makes sure that the current cell is 0 by setting `grid[r][c]=0`. Since the current value of  $x$  is not a valid entry so the program will check for the next value of  $x$ . If  $x$  reaches 10 then that means the current board will not be solved, so the solver function returns 0. This return of zero happens when solver calls itself (we are assuming inputs from the user are valid/solvable grids), when it is zero then that means the program needs to "undo" some steps that it took, and take a different path (try the next possible value).

These are the steps taken in the function **solver**; the program will repeatedly do this until it finds a path (the correct combination of entries in the board) that will solve/complete the grid. And it will stop calling itself recursively when the base case is reached. The program returns to the main function because it is now ready to print the output.

---

## Sample test cases for the 9x9 Solver



First 9 lines is the given board (manual input/not fed through file), the next 9 lines is the solved board.

```
C:\Users\Kyle\Downloads>java -jar Mars45.jar sm nc 202005499_9.asm
800309070
000804000
700600590
069000000
000060001
000000950
003020000
092500000
000000700
826359174
915874362
734612598
169245837
357968241
248731956
573126489
492587613
681493725
```

```
C:\Users\Kyle\Downloads>java -jar Mars45.jar sm nc 202005499_9.asm
250030901
010004000
407000208
005200000
000098100
040003000
000360072
070000003
903000604
258736941
619824357
437915268
395271486
762498135
841653729
184369572
576142893
923587614
```

```
C:\Users\Kyle\Downloads>java -jar Mars45.jar sm nc 202005499_9.asm
000080009
000426130
000901506
200830974
309060080
000294000
056310000
000000807
084052010
0612583749
597426138
438971526
261835974
349167285
875294361
756318492
123649857
984752613
```

Sudoku grids from [http://www.sudoku-download.net/sudoku\\_9x9.php](http://www.sudoku-download.net/sudoku_9x9.php)

---

## Google Drive link for document video

[https://drive.google.com/file/d/1l7OrSR4LB2Kxis9ZZPAPE\\_Arc9qONA7v/view?usp=sharing](https://drive.google.com/file/d/1l7OrSR4LB2Kxis9ZZPAPE_Arc9qONA7v/view?usp=sharing)