

SUDOKU SOLVER

=====

Name: MAYUR BHOLE

=====

Roll Number: MT2014065

=====

Problem Statement:

Sudoku is a puzzle game played on a grid that consists of $n^2 \times n^2$ cells, each belonging to three groups: one of nine **rows**, one of nine **columns** and one of nine **boxes**.

A Sudoku puzzle is a Sudoku grid that is partially filled, meaning that a set of fixed cells, whose numerals are given (that cannot be chosen/changed by the solver). The objective of the puzzle Solver is to fill the Sudoku cells by assigning a numeral between 1 and n^2 to each blank cell in such a way that each numeral is unique in each of its three groups (row, column, box).

The objective of this assignment is to write a Sudoku Solver.

You can begin with implementing the following 2 rules.

Rule 1: If a cell can take only one number, assign that number to that cell.

Rule 2: If there is a number which can go into only one cell, then assign that number to that cell.

Abstract:

Sudoku is a logic puzzle that requires the player to fill an $n^2 \times n^2$ grid with the digits 1 to n^2 so that each row, column and $n \times n$ grid contains each digit exactly once. This project is aimed at solving $n^2 \times n^2$ Sudoku puzzles by implementing two basic rules: Rule1 (single candidate) and Rule2 (single position). These rules are able to solve easy and medium level Sudoku puzzles. In order to solve the hard puzzles, the algorithm makes use of backtracking. Sudoku is a NP-Complete problem.

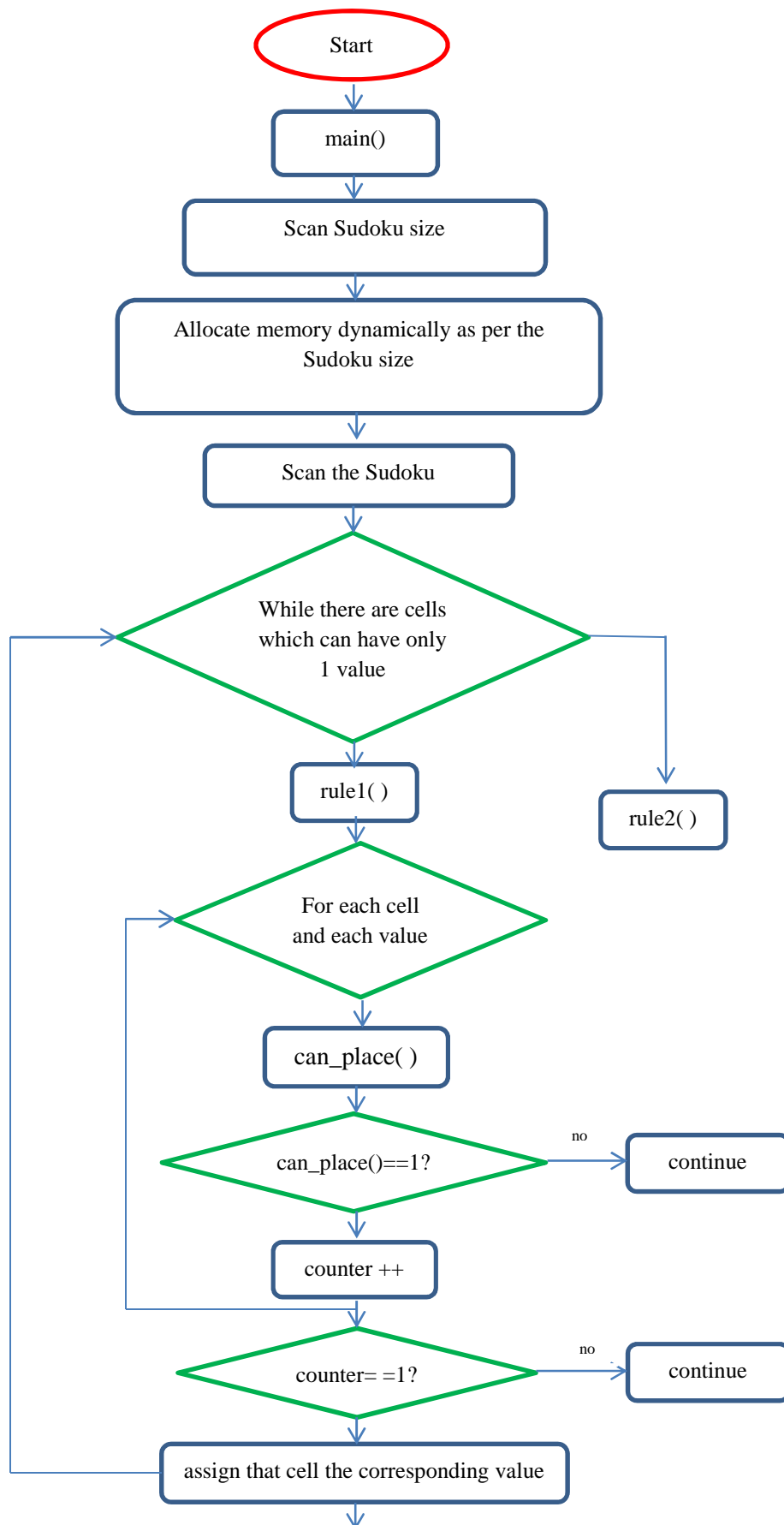
Data Structures & Functions Used:

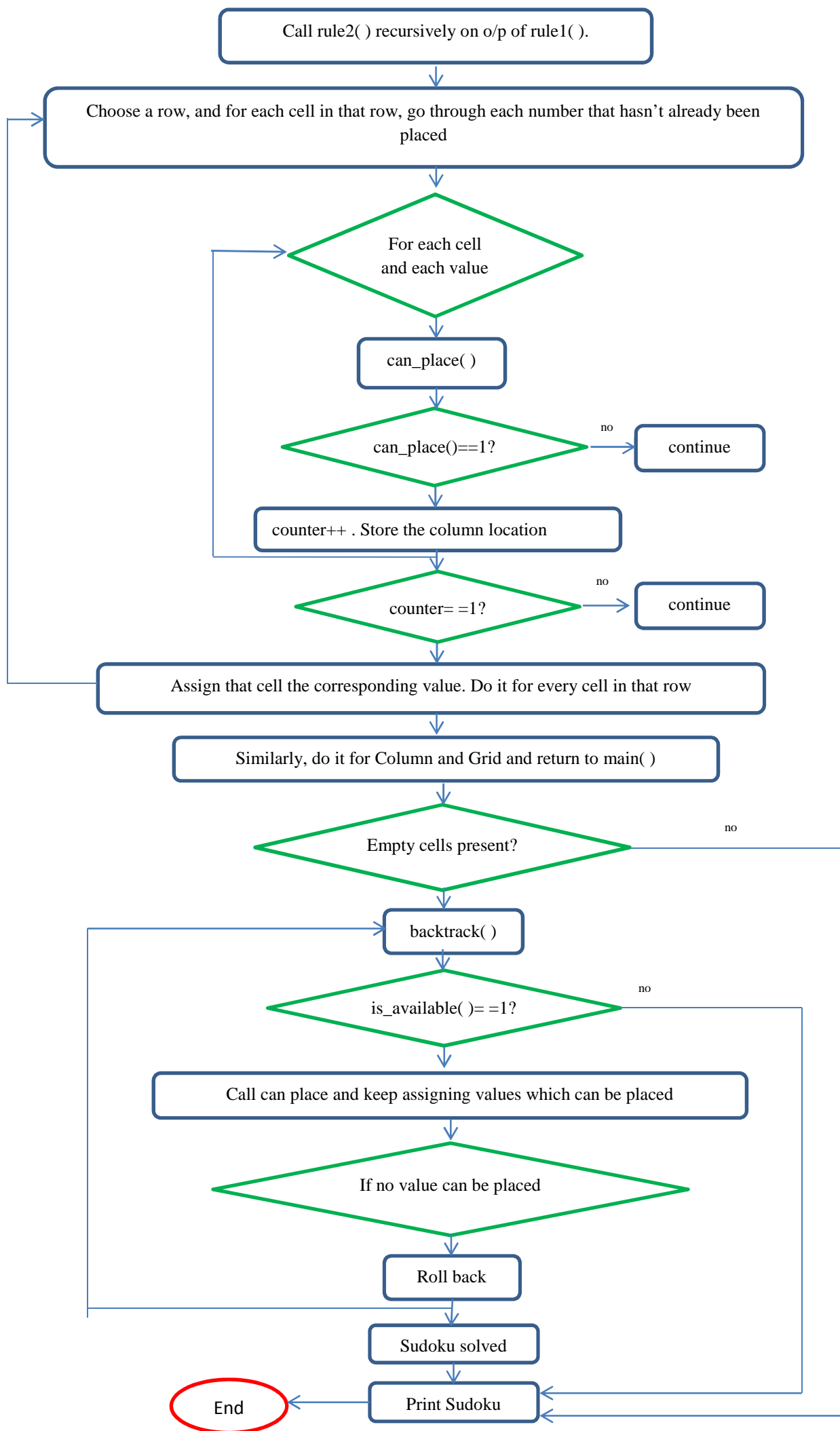
1. **Array:** A 2D array of $(n^2 \times n^2)$ is used. Memory is allocated dynamically using malloc.
2. **can_place():** It checks whether the number can be placed at that particular cell. Returns 1 if the number can be placed, else returns 0.
3. **filled_values():** It counts the number of filled cells in the Sudoku and returns the count.
4. **is_available():** It checks whether the cell is empty or filled. If it is empty then returns 1, else returns 0.
5. **rule1():** It checks for all the possible values which a cell can have, if that particular cell can contain only one value then it assigns that value to that cell.
6. **rule2():** It checks if there is a number which can go into only one cell, if true, then it assigns that number to that cell.
7. **backtrack():** It recursively assigns the possible values to empty cells until there is a situation when none of the values can go in a particular cell.
8. **print_sudoku():** It prints the Sudoku puzzle.

Approach:

1. Dynamically allocate a block of memory ($n^2 \times n^2$) using a double pointer.
2. Scan the input and store it into the dynamically allocated memory.
3. Apply Rule 1 recursively until there is no cell which can have only one possible value.
4. Rule 1: For each empty cell and for each value (1 to n^2), call `can_place()` function to check whether the value can go into that particular cell. If the `can_place()` function returns 1, then increment the counter and also store that value into a temporary variable. If counter value is 1, then the cell can contain only one value. This value is assigned to that particular cell using a temporary variable.
5. Apply Rule 2 recursively on the output of Rule 1.
6. Rule 2: Choose a row and for each cell in that row, go through each of the numbers that hasn't already been placed. Call `can_place()` function. If `can_place()` returns 1, then store the location of column in a temporary variable and increment the counter. After doing it for all cells in that row, check if the value of counter is 1. If yes, then assign the value to that cell. Similarly, do it for column and box.
7. If there are still empty cells even after applying the above rules, then call Backtrack function.
8. `backtrack()`: Call `is_available()`, if it returns 0, then the Sudoku is complete, else if it returns 1, then there are empty cells. Now for every value, call `can_place()`. If that value can be placed in the cell, then assign it to that cell. Recursively make call to `backtrack()`. If at any point of time there is a cell which cannot take any of the values, then roll back.
9. Finally print the Sudoku.

Flowchart:





Output:

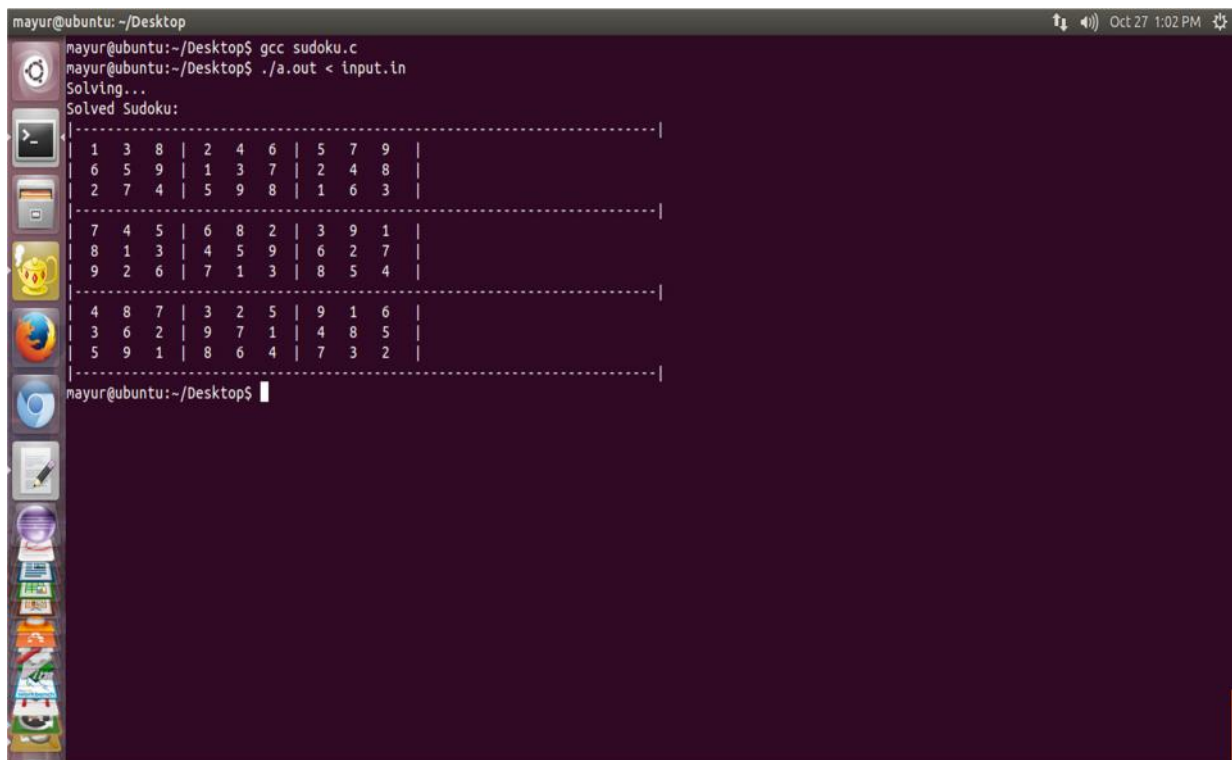
Note: The input has to be given using file redirection. First line has to be Sudoku size. Following that line should be the Sudoku puzzle to be solved

I. 3x3

Input:

```
3
000006000
059000008
200008000
045000000
003000000
006003054
000325006
000000000
000000000
```

Output:



```
mayur@ubuntu: ~/Desktop
mayur@ubuntu:~/Desktop$ gcc sudoku.c
mayur@ubuntu:~/Desktop$ ./a.out < input.in
Solving...
Solved Sudoku:
-----|
1 3 8 | 2 4 6 | 5 7 9 |
6 5 9 | 1 3 7 | 2 4 8 |
2 7 4 | 5 9 8 | 1 6 3 |
-----|
7 4 5 | 6 8 2 | 3 9 1 |
8 1 3 | 4 5 9 | 6 2 7 |
9 2 6 | 7 1 3 | 8 5 4 |
-----|
4 8 7 | 3 2 5 | 9 1 6 |
3 6 2 | 9 7 1 | 4 8 5 |
5 9 1 | 8 6 4 | 7 3 2 |
-----|
mayur@ubuntu:~/Desktop$
```

Execution Time: 0.38 sec

II. 4x4

Input:

4

```
0 16 9 0 11 8 0 10 3 0 7 4 0 15 5 0
11 0 0 12 0 3 5 7 0 15 9 0 1 8 0 16
5 0 0 8 0 0 9 0 16 13 0 12 14 0 10 6
0 6 2 0 16 15 0 13 1 0 14 5 0 9 11 0
8 0 0 6 0 0 4 0 0 10 1 0 3 16 0 12
12 0 0 4 0 0 15 0 0 16 0 6 5 0 2 10
0 2 5 0 12 10 0 1 11 0 4 3 0 13 7 0
10 9 0 1 0 0 16 0 0 2 0 0 4 11 0 8
2 0 6 3 0 0 13 0 0 7 0 0 9 0 16 11
0 13 10 0 5 7 0 15 8 0 16 14 0 6 12 0
14 12 0 7 9 0 10 0 0 5 0 0 13 0 0 15
15 0 16 9 0 2 1 0 0 12 0 0 10 14 0 7
0 3 12 0 1 14 0 8 10 0 15 9 0 7 4 0
7 4 0 2 10 0 11 5 0 3 0 0 15 0 0 14
13 0 11 14 0 16 6 0 5 1 0 0 8 0 0 3
0 1 15 0 13 12 0 3 14 0 2 8 0 10 6 0
```

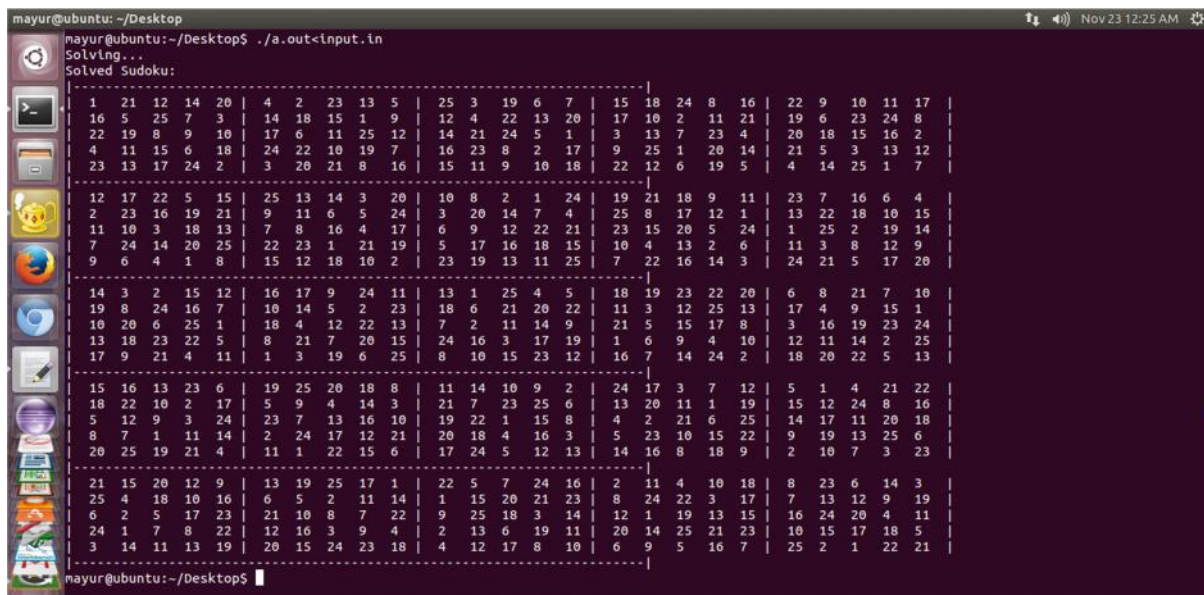
Output:

```
mayur@ubuntu: ~/Desktop
mayur@ubuntu:~/Desktop$ gcc sudoku.c
mayur@ubuntu:~/Desktop$ ./a.out < input.in
Solving...
Solved Sudoku:
 1 16 9 13 | 11 8 14 10 | 3 6 7 4 | 12 15 5 2
11 14 4 12 | 6 3 5 7 | 2 15 9 10 | 1 8 13 16
5 15 7 8 | 4 1 9 2 | 16 13 11 12 | 14 3 10 6
3 6 2 10 | 16 15 12 13 | 1 8 14 5 | 7 9 11 4
-----
 8 7 13 6 | 2 5 4 11 | 9 10 1 15 | 3 16 14 12
12 11 14 4 | 3 13 15 9 | 7 16 8 6 | 5 1 2 10
16 2 5 15 | 12 10 8 1 | 11 14 4 3 | 6 13 7 9
10 9 3 1 | 7 6 16 14 | 12 2 5 13 | 4 11 15 8
-----
 2 8 6 3 | 14 4 13 12 | 15 7 10 1 | 9 5 16 11
4 13 10 11 | 5 7 3 15 | 8 9 16 14 | 2 6 12 1
14 12 1 7 | 9 11 10 16 | 6 5 3 2 | 13 4 8 15
15 5 16 9 | 8 2 1 6 | 4 12 13 11 | 10 14 3 7
-----
 6 3 12 5 | 1 14 2 8 | 10 11 15 9 | 16 7 4 13
7 4 8 2 | 10 9 11 5 | 13 3 6 16 | 15 12 1 14
13 10 11 14 | 15 16 6 4 | 5 1 12 7 | 8 2 9 3
9 1 15 16 | 13 12 7 3 | 14 4 2 8 | 11 10 6 5
```

Execution Time: 0.50 sec

III. 5x5:

Output:



```
mayur@ubuntu: ~/Desktop
mayur@ubuntu:~/Desktop$ ./a.out<input.in
Solving...
Solved Sudoku:
1 21 12 14 20 | 4 2 23 13 5 | 25 3 19 6 7 | 15 18 24 8 16 | 22 9 10 11 17 |
16 5 25 7 3 | 14 18 15 1 9 | 12 4 22 13 20 | 17 10 2 11 21 | 19 6 23 24 8 |
22 19 8 9 10 | 17 6 11 25 12 | 14 21 24 5 1 | 3 13 7 23 4 | 20 18 15 16 2 |
4 11 15 6 18 | 24 22 10 19 7 | 16 23 8 2 17 | 9 25 1 20 14 | 21 5 3 13 12 |
23 13 17 24 2 | 3 20 21 8 16 | 15 11 9 10 18 | 22 12 6 19 5 | 4 14 25 1 7 |
-----
12 17 22 5 15 | 25 13 14 3 20 | 10 8 2 1 24 | 19 21 18 9 11 | 23 7 16 6 4 |
2 23 16 19 21 | 9 11 6 5 24 | 3 20 14 7 4 | 25 8 17 12 1 | 13 22 18 10 15 |
11 10 3 18 13 | 7 8 16 4 17 | 6 9 12 22 21 | 23 15 20 5 24 | 1 25 2 19 14 |
7 24 14 20 25 | 22 23 1 21 19 | 5 17 16 18 15 | 10 4 13 2 6 | 11 3 8 12 9 |
9 6 4 1 8 | 15 12 18 10 2 | 23 19 13 11 25 | 7 22 16 14 3 | 24 21 5 17 20 |
-----
14 3 2 15 12 | 16 17 9 24 11 | 13 1 25 4 5 | 18 19 23 22 20 | 6 8 21 7 10 |
19 8 24 16 7 | 10 14 5 2 23 | 18 6 21 20 22 | 11 3 12 25 13 | 17 4 9 15 1 |
10 20 6 25 1 | 18 4 12 22 13 | 7 2 11 14 9 | 21 5 15 17 8 | 3 16 19 23 24 |
13 18 23 22 5 | 8 21 7 20 15 | 24 16 3 17 19 | 1 6 9 4 10 | 12 11 14 2 25 |
17 9 21 4 11 | 1 3 19 6 25 | 8 10 15 23 12 | 16 7 14 24 2 | 18 20 22 5 13 |
-----
15 16 13 23 6 | 19 25 20 18 8 | 11 14 10 9 2 | 24 17 3 7 12 | 5 1 4 21 22 |
18 22 10 2 17 | 5 9 4 14 3 | 21 7 23 25 6 | 13 20 11 1 19 | 15 12 24 8 16 |
5 12 9 3 24 | 23 7 13 16 10 | 19 22 1 15 8 | 4 2 21 6 25 | 14 17 11 20 18 |
8 7 1 11 14 | 2 24 17 12 21 | 20 18 4 16 3 | 5 23 10 15 22 | 9 19 13 25 6 |
20 25 19 21 4 | 11 1 22 15 6 | 17 24 5 12 13 | 14 16 8 18 9 | 2 10 7 3 23 |
-----
21 15 20 12 9 | 13 19 25 17 1 | 22 5 7 24 16 | 2 11 4 10 18 | 8 23 6 14 3 |
25 4 18 10 16 | 6 5 2 11 14 | 1 15 20 21 23 | 8 24 22 3 17 | 7 13 12 9 19 |
6 2 5 17 23 | 21 10 8 7 22 | 9 25 18 3 14 | 12 1 19 13 15 | 16 24 20 4 11 |
24 1 7 8 22 | 12 16 3 9 4 | 2 13 6 19 11 | 20 14 25 21 23 | 10 15 17 18 5 |
3 14 11 13 19 | 20 15 24 23 18 | 4 12 17 8 10 | 6 9 5 16 7 | 25 2 1 22 21 |
```

Execution Time: approx. 30 min

Conclusion: This algorithm checks all possible solutions to the puzzle until a valid solution is found. Though it is time consuming, the main advantage of using the algorithm is the ability to solve any puzzles and a solution is certainly guaranteed.

Suggestions:

1. Vaneet Goyal (MT2014065)

References:

1. <http://www.geeksforgeeks.org>
2. <https://www.sudokuoftheday.com/techniques/>