EXERCISE NO. 3

SUBMISSION DUE DATE: 5/1/2020 23:00

INTRODUCTION

In this assignment, you are asked to implement the **Sudoku** puzzle game.

Sudoku is a number-placement puzzle where the objective is to fill a 9x9 grid with digits so that each column, each row, and each 3x3 block contains all digits from 1 to 9.

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

For each cell, we denote its neighbors as the other cells in its row, column, and block. The rule is that no cell may not contain the same value as any of its neighbors.

A Sudoku puzzle is completed successfully once the 81st cell is filled with a legal value.

GOAL

In this assignment, you will implement the Sudoku game. You are asked to develop a command-line program which consists of two parts; initialization and the game itself. It receives one command-line parameter "*seed*", as described later (see "Randomization").

In the initialization part, the board is partially filled with random digits (from 1 to 9) in the following way. The program randomly generates a solved board, and then clears all cells except several cells chosen at random. The user chooses the number of these cells.

After the initialization, the game begins. For each move, you should print the game board in a predefined format (described later) and wait for user input. The game continues in this manner until the puzzle is solved and all cells are filled with correct values.

To generate the board, solve the board, and give hints to the user, we will use the **backtracking algorithm**. This algorithm attempts to solve a Sudoku board by filling in its empty cells. It may be possible that there is no solution for a given board. Two variants of the backtracking algorithm will be used, randomized and deterministic.

BACKTRACKING

To solve the Sudoku puzzle, use the recursive Sudoku backtracking algorithm. This algorithm is a brute-force algorithm which attempts all possible combinations of input until reaching a solution or exhausting all options (and thus no solution exists). The randomized version of the algorithm can be used to generate puzzles.

We will first describe the deterministic backtracking algorithm. Then, the changes for the randomized version will be detailed. Note that in both cases, non-empty cells are never changed – we attempt to solve the board at its current state!

The algorithm visits all empty cells from left to right, then top to bottom, filling in digits sequentially (from 1 to 9), or backtracking when the number is found to be not valid.

Place the digit "1" in the first empty cell and check if the placement is legal, i.e., the digit does not appear in the same column, row, or block. If it is legal, then the algorithm advances to the next cell and repeats the process (places the digit "1" in that cell). If it is illegal, increment the value (to "2"), check that the placement is legal, and repeat. Each time a digit cannot be entered, we instead increase the value of that cell by one.

If neither of the 9 digits is allowed in a specific cell, then the algorithm leaves the cell blank and moves back (backtracks) to the previous cell. The value in the previous cell is then incremented by one. If the value of that cell is also 9, then the algorithm leaves that cell blank as well and moves back (backtracks) to the cell before it, until we find a cell where the value can be increased legally and continue the process from there.

The procedure above is repeated until a legal placement for the last (81st) cell is discovered. Thus, all cells are filled, and we have a solution for the given board. However, if we backtrack to the first originally empty cell and discover that neither of the 9 digits is allowed, then no solution for the given board exists.

Note: the backtrack algorithm can be very slow – that is OK. You are **required** use recursion in your implementation. Refer to **Tutorial 2** for further instructions and examples.

For the randomized version of the algorithm, when visiting a cell, we don't place the digit "1" or increment its value by one, but instead list all *legal* digits for that cell and randomly choose one. If no legal digit exists for that cell, then we trackback as in the original algorithm, and choose a random digit that has not been chosen yet for the previous cell, etc.

To ensure consistency of the randomized algorithm, follow these rules:

1. When choosing digits for a cell, we only consider *legal* digits, i.e., we omit the digits of the cell's neighbors (so, even when we first visit a cell, we might consider less than 9 options to choose from).
2. To randomize the chosen digit, put them in order and randomize the index, e.g., if the available digits that have not been chosen previously are 1, 3, 4, 9, then we randomize a number 0-3 and apply the result as the index (index 1 is digit 3, etc.).
3. Don't call *rand()* when only a *single* legal value remains, just use that value.

INITIALIZATION

The game starts by asking the user to enter the number of cells to fill. The program prints the following message and waits for the user's input.

"Please enter the number of cells to fill [0-80]:\n"

The user then enters a number between [0-80]. This is the number of "fixed" cells, cells with values that never change throughout the game.

If the user enters an invalid number, the program prints an error message: "Error: invalid number of cells to fill (should be between 0 and 80)\n", and lets the user try again by printing the same message as above. This phase finishes only upon providing a valid number.

We will use *scanf("%d", …)* to read the input number. You may assume the user enters a single valid integer, or we reach EOF. If we reach EOF, the program prints "Exiting…\n" and terminates.

PUZZLE GENERATION

To generate a random Sudoku puzzle with H hints, you will use the following procedure:

1. Create an empty board
2. Use randomized backtracking to get a random solved board, with no "fixed" cells
3. Repeat H times:
   (a) Randomly select a cell <X, Y>. If cell <X,Y> is "fixed" (i.e., was previously selected), repeat randomizing X,Y until cell <X,Y> is *not* fixed. To ensure consistency, select a cell by randomizing X, then randomizing Y.
   (b) Mark the chosen cell as a "fixed" cell.
4. Clear all cells that are not fixed (NxN – H cells that were not selected).

In addition, you should store the solution (the completed board from step 2), as it is used for giving hints to the user.

RANDOMIZATION

The Sudoku program will receive a single command line parameter – seed. You may assume this parameter is always supplied and that it is a valid integer in the correct range.

At the beginning of your program, add the call: `srand(seed)`, where "seed" is the command-line parameter received. This call seeds the random number generator. By supplying the same seed argument, you can ensure that the series of randomly generated numbers is fixed, which can be useful for testing and comparing to the supplied executable.

Whenever you need a random number in the range 0 to X-1, simply call `rand()%X` to get next random number in the correct range.

THE GAME

After initialization, the game begins where the user attempts to solve the generated puzzle.

The program prints the game board (once, described below), and then repeatedly gets commands from the user as follow:

1. The command includes at least one non-whitespace character.
2. The command and its parameters are pairwise separated by one or more white spaces (not including '\n').
3. You may assume that a command is at most 1024 characters long.
4. A command can contain "extra" parameters – we allow these and ignore any extra parameters. Commands with insufficient parameters are invalid.
5. Commands are case-sensitive.

To read a command line use `fgets(str,n,stream)`, where *str* is a pre-allocated *char** variable, *n* is the maximum number of characters to read (including the string terminator), and *stream* is the file we want to read – ***stdin***.

1. set X Y Z
    a. Sets the value Z in cell <X,Y> (X is the column, Y is the row).
    b. You may assume the values of X, Y, and Z are valid and correct (1-9 for X, Y and 0-9 for Z). A value of 0 for Z means that the cell should be cleared.
    c. If cell <X,Y> has already been filled by the program at initialization, the program prints "Error: cell is fixed\n", and the command is ignored.
    d. If the value Z is invalid for the cell <X,Y> (exists in one of its neighbors), the program prints "Error: value is invalid\n", and the command is ignored.
    e. Otherwise, the program prints the game board again (as described later).
    f. If the game is over, i.e., this is the last empty cell, the program prints "Puzzle solved successfully\n". From this point, all commands except *exit* and *restart* are considered invalid.
2. hint X Y
    a. Give a hint to the user by showing a possible legal value for a single cell.
    b. You may assume that the values X, Y are valid integers in the correct range.
    c. The program prints "Hint: set cell to Z\n", where Z is the value of cell <X,Y> according to the solution of the board (stored during the puzzle generation described above, or "validate" command described below). The hint is given even if it is incorrect for the current state of the board!
3. validate
    a. Validates that the current state of the board is solvable.
    b. Solve the generated board with **deterministic** backtracking.
    c. If no solution is found, the program prints: "Validation failed: board is unsolvable\n".
    d. If a solution is found, update the stored solution to the newly-found solution (so future hints are given from the new solution), and the program prints: "Validation passed: board is solvable\n".

4. restart
    a. Restart the game by starting over with the initialization procedure (followed by solving a new puzzle).
    b. Restart the program by asking the user again for the number of cells to fill, proceeding to the initialization procedure and then a new game.
5. exit
    a. The program prints "Exiting...\n", frees all memory resources and exits.

Special Remarks:

- Columns and rows are **1-based**, i.e., the first column and row are indexed 1.
- If the user enters an invalid line command, i.e., a command that doesn't match any of the commands defined in this section or doesn't match the syntax we defined in any way, the program prints "Error: invalid command\n". You should ignore blank lines and not output anything in this case. A valid command with extra parameters (e.g., "set 2 3 4 5") is considered valid and any extra parameters are ignored.
- If a standard failure or memory error occurs (failure of *malloc*, *scanf*, etc.) beyond what was described, the program prints "Error: <function-name> has failed\n" and exits. No need to exit cleanly. **Note:** you don't need to check if *printf* succeeded.
- If we reach **EOF** the program should exit (as if the user input the "*exit*" command).

GAME BOARD – PRINTING FORMAT

After initialization, and whenever the user successfully applies the "set" command to enter input into a cell (or clear a cell if the input is '0'), the board is printed to the user.

Our printing format will have 13 rows, consisting of two types:

1. Separator row – a series of 34 dashes: "----------\n".
2. Cells row – each cell is represented as two characters: a dot '.' and digit for a fixed cell, and a space and digit for a non-fixed cell. Use space instead of a digit for blank cells. A pipe '|' starts and ends the row and separates each set of 3 digits. A space separates all cells and pipes, for example: "| .1 2 .3 | 4 5 6 | .7 8 9 |\n".

The board will be printed in the following format:

1. Separator row
2. Three rows of cells
3. Separator row
4. Three rows of cells
5. Separator row
6. Three rows of cells
7. Separator row

Note: do not copy-paste the printouts here as that may cause issues. Type them out manually in your source files!

```
---------------------------------
|          |          |    .3    |
|       6  |    .1    |  8  4    |
|    .5  3 | .6  4 .7 | .9       |
---------------------------------
|  5    .8 |    6 .3  |    9 .4  |
| .3    9  |       .8 |  2 .7    |
|  4       | .1 .2    |  5  8    |
---------------------------------
|    .8    |  5 .3    |          |
|  1 .7    |       4  |  3  6    |
|     3 .2 | .9    .1 |       8  |
---------------------------------
```

IMPLEMENTATION OVERVIEW

In this assignment you need to implement different modules which will be combined to form the entire program. Here we describe a recommended set of modules for your implementation.

Notice that these modules were carefully picked and described so it would ease the implementation. If you think that some are useless, or you come up with a better implementation, you may use it but must thoroughly describe the differences and why you chose each. Furthermore, you may extend or split these modules. Feel free to make changes, if you properly justify them.

Regardless of any changes, you must provide full documentation for all the modules and all functions of each module.

The recommended modules for HW3 are:

1. Parser
    a. As the user inputs lines that represents commands, you need to read and interpret them. This should be the responsibility of the Parser module.
    b. It is highly recommended to use the standard function ***strtok***, which can be found in the header file ***string.h***. This function is similar to Java's split method. Use " \t\r\n" (note the space) as a delimiter string. More information can be found online
2. Game
    a. This module encapsulates the Sudoku puzzle game and the Sudoku board.
    b. It will be used to store and manipulate the game status, validate, and set moves, and so on.
3. Solver
    a. This module implements the Backtrack algorithms.
    b. Notice that it is important that the user of this module is not aware of the algorithm behind the module, i.e., it solves the board and the actual implementation may freely change without "breaking" the public interface of the module.
    c. The interface is thus recommended to contain a solver method, and a puzzle generator method (which supplies a puzzle as well as a solution).
4. main.c
    a. The ***main()*** function should be the only function in this module, and it should have no header file.
    b. Any auxiliary functions that you use should reside at the MainAux module.
5. MainAux
    a. All auxiliary functions are placed inside this module. Those are functions that do not belong to any of the other modules.

TESTING

It is recommended to test every module separately before writing your main function. Every module must be relatively independent, such that it operates the same even if the implementations of other modules change (of course, the interface of these modules does not change).

To test every module, you should implement small programs that are called unit tests. Every unit test should cover as much edge cases as possible, this will ease the debugging phase and should make the developing process faster. Unit tests are separate programs that provide "fake" input to functions of a module. By executing calls with various inputs to the functions of the module, the module can be checked independently of the rest of your code.

When changing a module (or one of its dependencies), proper unit tests ensure that we can run the unit tests again and, if all tests pass, then the change is valid, otherwise the tests that fail direct us to the corresponding bug.

EXECUTABLE

As usual, an executable is provided with the solution of the assignment. Use it both to test your code and whenever you are unsure regarding the correct output in certain cases.

Note that the executable uses randomizations in a very specific manner which you should emulate precisely! See Tutorial 2 and above for further clarifications.

PUTTING IT ALL TOGETHER

For HW3 (and the project), you should write a makefile. The makefile should compile your main function along with all parts of your program by simply invoking **“make”. “make clean”** will clean previous compilations. The executable filename should be "sudoku".

CLOSING NOTES AND SPECIAL REMARKS

Use the provided executable to answer any questions regarding input/output, printing format, program behavior, etc. Always match the executable.

Do not duplicate the two backtracking solutions! A correct solution has most of the code shared between deterministic and randomized.

Don't use global variables! Each module should be a black box when used, be defined by its interface and not its implementation (which may change freely without affecting the modules using it), allow creation of several instances, etc. Refer to Tutorial 2. Modules can depend on each other by **interface**, not by **implementation**!

It is recommended to start working on the following modules (as appears in order): Parser, Game, Solver.

Once you have finished implementing these modules, you can start implementing the main function. **Tip:** make sure your code doesn't depend on the size of the board (9) or block (3), but instead uses constants that can be changed and still work. It will help in the future.

SUBMISSION GUIDELINES

Submission is in pairs **only**. One student should submit a zipped file with the name **"id1_id2_assignment3.zip"** where id1 and id2 are the ids of the partners.

The zip file should include all source and header files, along with your makefile. **(note that the unit tests should not be included).**

GOOD LUCK!