FINAL PROJECT

SUBMISSION DUE DATE: 16/09/2018 23:00 (FINAL DEADLINE! NO EXTENSIONS!)

INTRODUCTION

In the final project you will create an interactive **Sudoku** program with an ILP (Integer Programming) solver component. You will use ILP to both solve Sudoku puzzles and check possible solutions to it. ILP uses mathematical optimization methods to speed up the solution time, compared to the exponential backtracking algorithm from HW3.

The Sudoku program should work in a similar way to the program you implemented in HW3, with user interaction performed through console commands.

IMPORTANT: do not copy-paste text output from this document! Enter it yourself, otherwise some characters will not match the correct output!

HIGH-LEVEL DESCRIPTION

The Sudoku project should have the following functionalities:

- Solving Sudoku puzzles
- Editing and creating new Sudoku puzzles
- Supporting puzzles of any size
- Validating the board
- Saving and loading puzzles from files
- Randomly generating new puzzles
- Finding the number of solutions for a given board
- Auto-filling option for obvious cells (where only a single value is valid)

The program will have two main modes: ***Edit*** and ***Solve***, and an additional ***Init*** mode. In the Solve mode, the user attempts to solve a pre-existing puzzle. In the Edit mode, the user edits existing puzzles or creates new ones, and saves these for opening in the *Solve* mode later on. In the *Init* mode, the user loads a file to enter either *Edit* or *Solve* mode.

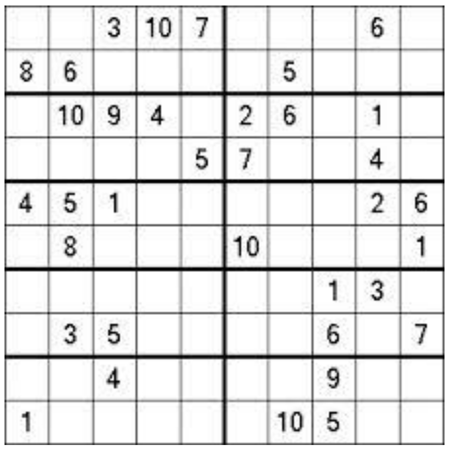
The project consists of 5 main parts:

- Sudoku game logic
- Console user interface
- ILP solver algorithm
- Random puzzle generator
- Exhaustive backtracking solver (for determining the number of different solutions)

The executable will be named "*sudoku-console*". It receives no command-line arguments.

SUDOKU

Sudoku is a number-placement puzzle where the objective is to fill a NxN grid (usually 9x9) with digits so that each column, each row, and each block contain all digits from 1 to N.



		3	10	7				6	
8	6					5			
	10	9	4		2	6		1	
				5	7			4	
4	5	1						2	6
	8				10				1
							1	3	
	3	5					6		7
		4					9		
1						10	5		

The grid consists n x m blocks of size m x n each, for a total of N=mn rows and columns. For example, a 10x10 grid can consist of 5x2 blocks, each of size 2x5 cells (see figure above).

For each cell in the grid, we denote by its neighbors the other cells in its row, column, and block. The main rule is that the value of each cell is different from the values of its neighbors.

A Sudoku puzzle is completed successfully once all N^2 cells are filled with legal values.

Note that the Sudoku puzzle described here is different from the one described in HW3, as its size varies. We will limit both *n* and *m* to 5; however, this is for practical use only and your implementation should not rely on this limitation.

DETAILS

The guidelines below should help you understand how your program should work. The implementation details (such as data structures, modules, etc.) are up to you and will be taken into account for grading.

OVERVIEW

The Sudoku program operates in either "Edit mode" or "Solve mode" (or "Init mode" at the beginning of the program and after a *restart* command).

When the user enters the Solve mode (via the "solve" command), it is given a puzzle to solve. The user may save the puzzle at any time to resume it later.

When the user enters the Edit mode (via the "edit" command), it may freely edit a puzzle and save it to a file. Files can be loaded in either the Edit mode ("edit" command) for updates, or the Solve mode ("solve" command) for solving them.

When saving a file in the Edit mode, the board is validated (i.e., make sure that a solution for this puzzle exists), and all filled cells are marked as "fixed". Such cells cannot be edited by the user in the Solve mode.

At any point in the program the user may use the "exit" command to exit the program. Before exiting, make sure all resources are freed, all files closed, etc. You do not have to exit cleanly on errors.

GAME PARAMETERS

- **Game mode:** either Edit, Solve, or Init. The game starts in: Init. For each command below, we specify whether it changes the mode, and what is the expected behavior of the command for each mode. If it is not specified, then the command behaves the same for Edit and Solve modes. No commands are available in Init mode unless specifically stated otherwise.
- **Mark errors:** either 1 (True) or 0 (False). Determines whether errors in the Sudoku puzzle are displayed. When editing or solving a puzzle, the user is free to enter any input, even erroneous one (according to the rules of Sudoku). An erroneous board is a board with cells which contain illegal values, i.e., the same value more than once in the same row, column, and/or block. However, when this parameter is set to 1, erroneous values are displayed followed by an asterisk. All cells that participate in an error should be marked. The default value is 1. In Edit mode we always mark errors, regardless of the value of this parameter.
- **Undo/Redo list:** for the Edit and Solve modes, the program maintains a *doubly-linked* list of the user's moves, and a pointer to the current move. The user may freely traverse this list (move the pointer) using the "undo" and "redo" commands (which also update the board). When the user makes a new move (via the "set" command – see below), the redo part of the list is cleared, i.e., all items beyond the pointer are removed, and the new move is added to the end of the list and marked as the current move. Note that when starting a new puzzle (by "solve" or "edit") the undo/redo list is **empty**, even if certain cells already contain values. These values are not considered moves and cannot be undone.
- **Game board:** for Edit and Solve modes, the program keeps the current board, i.e., a 2-dimensional array of the board values. The board is printed to the user after every move, as described below. Each board cell can either be "fixed" or not. A fixed cell is a part of the puzzle and may not be changed when solving.

PROGRAM FLOW

When the program starts, it first prints the title "Sudoku\n------\n" (Sudoku in one line and 6 dashes in another). Initially, the game mode is Init, awaiting user commands.

Whenever the user enters an invalid command or a command with a wrong syntax or parameters (for which there is no specific output described below), the program outputs: "ERROR: invalid command\n". You should ignore blank lines (i.e., empty lines or lines that contain only whitespace characters).

All parameters of the command should fit in one line, i.e., a command may not span two lines; these are treated as two different commands. Note that it is OK if the user enters extra parameters for a command! Ignore any parameters beyond those defined. If the command is valid without these extra parameters, then it is a valid command.

While editing or solving a board, the user enters values into cells. The user always has the possibility to enter erroneous values (i.e., a value that already exists in one of the cell's neighbors); however, these cells will be marked if the "Mark errors" parameter is set to 1 (and will always be marked in **Edit** mode). Additionally, puzzles with erroneous values may never be saved in **Edit** mode.

Additionally, in **Solve** mode the user may only enter (or edit) values into cells that are **not** *fixed*, i.e., they were not part of the original puzzle. In **Edit** mode, the user may enter values into any cell. When saving a puzzle to a file in **Edit** mode, all cells containing values are immediately considered *fixed* and saved as such.

BOARD PRINT FORMAT

The board is printed to the screen at the beginning of each user turn (when starting a new puzzle, and after every "set" command). Recall that the board consists of NxN cells, arranged in n-by-m blocks (n rows of blocks, m columns of blocks) of size m-by-n (m rows of cells, n columns of cells).

The board printing format should consist of N+n+1 rows, consisting of two types:

1. Separator row – a series of 4N+n+1 dashes.
2. Cells row – a pipe character '|' starts and ends the row, and separates each set of n cells (i.e., each block). Unlike in HW3, no space separates cells and pipes beyond the space characters defined here. Each cell is represented as 4 characters:
   a. A space character.
   b. Two digits for the cell value (printf format "%2d"). Use two spaces instead of a value for a blank cell.
   c. A dot '.' for a fixed cell, an asterisk '*' for an erroneous cell when in Edit mode or the "Mark errors" parameter is set to 1, or space otherwise. A fixed cell is never marked as erroneous!

The board will be printed in the following format:

1. Separator row
2. Repeat n times for each row of blocks:
    a. Cell row
        i. Repeat m times for each of the row of cells
    b. Separator row

The board example from HW3 is repeated here in the new format:

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

COMMANDS

The interaction should be done via console commands. The user will interact with the program by typing commands from a list provided below, along with desired parameters. We describe below which commands are available in which mode of the program, as well as the specific details and arguments of each command.

The program repeatedly prompts for user commands. Before any command, the program prints "Enter your command:\n" and then awaits a user command. This line is repeated after every user input (valid command, invalid command, blank line, etc.).

The list of commands is provided below. When several outcomes are available for the same command (e.g., several errors hold), follow the first step that applies according to the order written here. In the list below, whenever a command should be treated as an invalid one, you should also print the corresponding error message described previously in this document ("ERROR: invalid command\n").

Note that throughout the project, your program should be able to handle any user input properly and execute the corresponding command or output the correct error message. However, you may treat any input line beyond 256 characters as an invalid command.

1. solve X
    a. Starts a puzzle in ***Solve*** mode, loaded from a file with the name "X", where X includes a full or relative path to the file.
    b. In case the file does not exist or cannot be opened, the program prints "Error: File doesn’t exist or cannot be opened\n" and the command is not executed.
    c. You may assume that the file contains valid data and is correctly formatted.
    d. Note that this command is always available, in *Solve*, *Edit*, and *Init* modes. Any unsaved work is lost.

2. edit [X]
    a. Starts a puzzle in ***Edit*** mode, loaded from a file with the name "X", where X includes a full or relative path to the file.
    b. In case the file does not exist, cannot be created, or cannot be opened, the program prints "Error: File cannot be opened\n" and the command is not executed.
    c. You may assume the file contains valid data and is correctly formatted.
    d. The parameter X is optional. If no parameter is supplied, the program should enter ***Edit*** mode with an empty board.
    e. Recall that in ***Edit*** mode, the value of the "mark errors" parameter is always considered to be 1, ignoring its actual value.
    f. Note that this command is always available, in ***Solve***, ***Edit***, and ***Init*** modes. Any unsaved work is lost.

3. mark_errors X
    a. Sets the "mark errors" setting to X, where X is either 0 or 1.
    b. This command is only available in ***Solve*** mode. Otherwise, treat it as an invalid command.
    c. You may not assume anything regarding the value of X.
    d. If X is not 0 or 1, the program prints "Error: the value should be 0 or 1\n", and the command is not executed.

4. print_board
    a. Prints the board to the user.
    b. This command is only available in ***Edit*** and ***Solve*** modes. Otherwise, treat it as an invalid command.

5. set X Y Z
    a. Sets the value of cell <X,Y> to Z.
    b. This command is only available in ***Edit*** and ***Solve*** modes. Otherwise, treat it as an invalid command.
    c. The values of X, Y, and Z are not guaranteed to be correct (they may not even be integers).

d. If X, Y, or Z are invalid, the program prints: "Error: value not in range 0-N\n", where the size of the current board replaces N (e.g., 1-9), and the command is not executed. The user may empty a cell by setting Z=0.

e. If cell <X,Y> is *fixed* (i.e., part of the puzzle saved in ***Edit*** mode), the program prints "Error: cell is fixed\n", and the command is not executed.

f. Clear any move beyond the current move from the undo/redo list, then add the new move to the end of the list and mark it as the current move.

g. An erroneous input (such as entering 5 when column X already contains 5) **is allowed** but an error should be issued if the "mark errors" parameter is set to 1.

h. This command prints the game board, according to the instructions above.

i. In ***Solve*** mode, after the game board is printed, if this is the last cell to be filled then the board is immediately validated. If the validation fails, we remain in *Solve* mode (the user will have to undo the move to continue solving) and the program prints: "Puzzle solution erroneous\n". If the validation passes, the program prints: "Puzzle solved successfully\n", and the game mode is set to ***Init*** (followed by "Enter your command:\n").

6. validate
   a. Validates the current board using ILP, ensuring it is solvable.
   b. This command is only available in ***Edit*** and ***Solve*** modes. Otherwise, treat it as an invalid command.
   c. If the board is erroneous, the program prints: "Error: board contains erroneous values\n", and the command is not executed.
   d. If the board is found to be solvable, the program prints: "Validation passed: board is solvable\n". Otherwise, the program prints: "Validation failed: board is unsolvable\n".

7. generate X Y
   a. Generates a puzzle by randomly filling X cells with random legal values, running ILP to solve the resulting board, and then clearing all but Y random cells.
   b. This command is only available in ***Edit*** mode. Otherwise, treat it as an invalid command.
   c. The value of X is not guaranteed to be correct (it may even not be an integer).
   d. If X is invalid (not a number, or more than the number of empty cells in the current board), the program prints: "Error: value not in range 0-E\n", where the number of empty cells in the current board replaces E, and the command is not executed.
   e. This command is only available when the board is empty (no cell contains *any* value). If the board isn't empty, the program prints: "Error: board is not empty \n", and the command is not executed.

f. Randomly choose X cells, filling each with a legal random value. Once X cells contain values, run ILP to solve the resulting board. After the board is solved, randomly choose Y cells, and clear the values of all other cells.

g. If one of the X randomly-chosen cells has no legal value available, or the resulting board has no solution (the ILP solver fails), clear the board entirely and repeat the previous step. After 1000 such iterations, the program prints: "Error: puzzle generator failed\n", and the command is not executed.

h. If the puzzle was generated successfully, print the board.

8. undo
   a. Undo previous moves done by the user.
   b. This command is only available in ***Edit*** and ***Solve*** modes. Otherwise, treat it as an invalid command.
   c. Set the current move pointer to the previous move and update the board accordingly. This does not add or remove any item to/from the list.
   d. If there are no moves to undo the program prints: "Error: no moves to undo\n", and the command is not executed.
   e. If there was a move to undo, the program prints the board and then prints: "Undo X,Y: from Z1 to Z2\n", where X,Y is the location of the changed cell, Z1 is its value before the undo, and Z2 is its value after. Empty cells (Z1 or Z2 with a value of 0) should be printed as an underscore '_'.

9. redo
   a. Redo a move previously undone by the user.
   b. This command is only available in ***Edit*** and ***Solve*** modes. Otherwise, treat it as an invalid command.
   c. Set the current move pointer to the next move and update the board accordingly. This does not add or remove any item from the list.
   d. If there are no moves to redo the program prints: "Error: no moves to redo\n", and the command is not executed.
   e. If there was a move to redo, the program prints the board and then prints: "Redo X,Y: from Z1 to Z2\n", where X,Y is the location of the changed cell, Z1 is its value before the redo, and Z2 is its value after. Empty cells (Z1 or Z2 with a value of 0) should be printed as an underscore '_'.

10. save X
    a. Saves the current game board to the specified file, where X includes a full or relative path to the file.
    b. This command is only available in ***Edit*** and ***Solve*** modes. Otherwise, treat it as an invalid command.
    c. In ***Edit*** mode, if the board is erroneous the program prints: "Error: board contains erroneous values\n", and the command is not executed.
    d. In ***Edit*** mode, the board is validated before saving. If the validation passes, then proceed normally (nothing is printed). If the validation fails, the

program prints "Error: board validation failed\n", and the command is not executed.

  e. If the file cannot be created or modified, the program prints the following message: "Error: File cannot be created or modified\n", and the command is not executed.
  f. If no errors occur then the game board is saved to the file and the program prints "Saved to: X\n", where X is the exact filename provided by the user.
  g. File formats for save files are described below.
  h. Saving the puzzle to a file *does not* modify the Redo/Undo list in any way, i.e., a *reset* command (described below) will still revert to the state of the originally loaded file.
  i. In ***Edit*** mode, all cells containing values are marked as "fixed".

11. hint X Y
  a. Give a hint to the user by showing the solution of a single cell X,Y.
  b. This command is only available in ***Solve*** mode. Otherwise, treat it as an invalid command.
  c. You may not assume anything regarding the values of X and Y.
  d. If X or Y are invalid (1-N according to the current board), the program prints: "Error: value not in range 1-N\n", where the size of the current board replaces N (e.g., 1-9), and the command is not executed.
  e. If the board is erroneous, the program prints: "Error: board contains erroneous values\n", and the command is not executed.
  f. If cell <X,Y> is *fixed* (i.e., part of the puzzle saved in *edit* mode), the program prints "Error: cell is fixed\n", and the command is not executed.
  g. If cell <X,Y> already contains a value (and isn't *fixed*), the program prints: "Error: cell already contains a value\n", and the command is not executed.
  h. Run ILP to solve the board. If the board is unsolvable, the program prints "Error: board is unsolvable\n", and the command is not executed. Otherwise, the program prints: "Hint: set cell to Z\n", where Z is the value of cell <X,Y> found by the ILP solution.

12. num_solutions
  a. Print the number of solutions for the current board.
  b. This command is only available in ***Edit*** and ***Solve*** modes. Otherwise, treat it as an invalid command.
  c. If the board is erroneous the program prints: "Error: board contains erroneous values\n", and the command is not executed.
  d. Run an exhaustive backtracking for the current board. The exhaustive backtracking algorithm exhausts all options for the current board, instead of terminating when a solution is found (as in HW3). Once done, the program prints: "Number of solutions: X\n", where X is the number of solutions for the current board returned by the exhaustive backtracking algorithm.
  e. If the board has only a single solution, the program additionally prints: "This is a good board!\n". Otherwise, the program prints: "The puzzle has more

than 1 solution, try to edit it further\n". This is just an informative message to the user and has no additional effect whatsoever.

   f. The exhaustive backtracking algorithm is described below.

13. autofill
    a. Automatically fill "obvious" values – cells which contain a single legal value.
    b. This command is only available in ***Solve*** mode. Otherwise, treat it as an invalid command.
    c. If the board is erroneous the program prints: "Error: board contains erroneous values\n", and the command is not executed.
    d. Go over the board and check the legal values of each empty cell. If a cell <X,Y> has a single legal value, fill it with the value and the program prints: "Cell <X,Y> set to Z\n", where X,Y is the cell index and Z is its value. You should go over the cells from left-to-right, then top-to-bottom.
    e. **Note** that only cells that had a single legal value *before* the command was executed should be filled, e.g., if cell <5,5> has two legal values, but by updating cell <4,5> we eliminate one of the legal values of cell <5,5>, then cell <5,5> should **not** be auto-filled!
    f. After performing this command, additional cells may contain obvious values. However, this command performs only a single iteration and these cells should not be filled.
    g. Once the command is finished, print the board.

14. reset
    a. Undo all moves, reverting the board to its original loaded state.
    b. This command is only available in ***Edit*** and ***Solve*** modes. Otherwise, treat it as an invalid command.
    c. The command goes over the entire undo/redo list and reverts all moves (no output is provided). Once the board is reset, the undo/redo list is cleared entirely, and the program prints: "Board reset\n".

15. exit
    a. Terminates the program. All memory resources must be freed and all open files must be closed. The program prints: "Exiting...\n".
    b. Note that this command is always available, even in ***Solve*** and ***Edit*** modes. Any unsaved work is lost.

INTEGER LINEAR PROGRAMMING (ILP)

Linear programming is a method to achieve the best outcome (maximum or minimum) in a mathematical model whose requirements are represented by linear relationships.

An integer linear program is a mathematical optimization program in which the variables are restricted to be integers, and the objective function and constraints are linear.

An integer linear program in canonical form is expressed as:

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^\mathrm{T}\mathbf{x} \\ \text{subject to} & A\mathbf{x} \le \mathbf{b}, \\ & \mathbf{x} \ge \mathbf{0}, \\ \text{and} & \mathbf{x} \in \mathbb{Z}^n, \end{array}$$

When given a problem, we can use an ILP solver to compute an optimum solution, i.e., assign values to the program variables so as to optimize the objective function while satisfying all the constraints.

In this project, you will use ILP to solve Sudoku puzzles for the user, either for validation or puzzle generation.

Further details on ILP and first steps for expressing Sudoku are provided in Tutorial 3.

The ILP solver we will use is the Gurobi Optimizer, a commercial optimization solver for linear programming. Gurobi supports a variety of programming and modeling languages, including a matrix-oriented interface for C which your program should use.

To use Gurobi, you should first register for an academic license using your TAU email: https://user.gurobi.com/download/licenses/free-academic

Gurobi is installed in *nova* and its code should be executed there. The current version installed in *nova* is Gurobi 5.6.3. You may also install Gurobi on your personal computer; however, there may be differences and you may do so at your own risk.

Our recommendation is to use Gurobi only on the *nova* server.

Gurobi guidelines and example code are provided in moodle.

EXHAUSTIVE BACKTRACKING

The exhaustive backtracking algorithm is similar to the deterministic algorithm; however, it doesn't finish when the last cell is filled with a legal value.

Instead, it marks the board as solved, incrementing a counter, and then continues to increment the value of that cell, or backtrack if necessary in order to retrieve other potential solutions.

Thus, the algorithm exhausts all possible values for all empty cells of the board, counting all solutions found in this process. Once the algorithm backtracks from the 1st empty cell, the algorithm is finished, and the counter contains the number of different solutions available for the current board.

Note that for some boards (according to size and values), the exhaustive backtracking algorithm can be very slow – that is OK.

Additionally, for the project you are required to implement the exhaustive backtracking algorithm with an explicit stack for simulating recursion, rather than through recursive calls.

Any recursive algorithm can be replaced with a stack, by replacing recursive calls with operations on the stack. Whenever a recursive call should be made, we instead add the relevant parameters to the stack (as a single item). Our code then iterates removing an item from the stack and performing the logic for it (which usually adds items to the stack) until the stack is empty.

SAVED GAMES

Each puzzle can be saved at any point, and then later loaded to solve or edit it. To load a puzzle, the user enters the "solve" or "edit" command. To save a puzzle, the user enters the "save" command.

Recall that when saving a puzzle in ***Edit*** mode, all cells with values are marked as "fixed".

FILES FORMAT

The format of the file is as follows (a simple text file):

- The first line contains the block size *m n*. These are two integers, separated by a single space. Recall that the board alphabet size is *N=m x n*.
- Each line contains a single row of the board.
- Each cell contains the cell's value separated with single spaces. Recall that the value in empty cells is 0. If the cell is "fixed", follow its value with a dot '.'.
- An example of a row with 4 values is: *"4 0 1. 0"*.

The save command should follow the above format precisely. Example save files will also be posted on moodle.

When saving a file, the board size and each board row are separated by a newline, and cell values are separated by a single space. However, the program should support loading files with values separated by any number and type of whitespace characters. For example, files may contain a single line of values separated by three tabs, be broken into lines incorrectly, etc.

ERROR HANDLING

Your code should handle all possible errors that may occur (memory allocation problems, erroneous user input, ILP errors, etc.).

File names can be either relative or absolute and can be invalid (the file/path might not exist or could not be opened).

Do not forget to check the following elements:

- The return value of a function. You may excuse yourself from checking the return value of any of the following I/O functions: $printf$, $fprints$, $sprint$, and $perror$.
- Any additional checks, to avoid any kind of segmentation faults, memory leaks, and misleading messages. Catch and handle properly any fault, even if it happened inside a library you use.

You do not need to exit cleanly on errors. Issue an appropriate message before terminating.

SUBMISSION GUIDELINES

The project will be submitted via Moodle. The grading may also involve testing knowledge of the code in a frontal meeting, to be set as necessary. Both students should be familiar with a significant part of the code.

You should submit a zip file named ***id1_id2_finalproject.zip***, where *id1* and *id2* are the IDs of both partners. The zipped file will contain:

- All project-related files (sources, headers, images).
- Your makefile.
- Partners.txt file according to the usual pattern.

CODING

Your code should be partitioned into files (modules) and functions. The design of the program, including interfaces, functions' declarations, and partition into modules – is entirely up to you, and is graded. You should aim for modularity, reuse of code, clarity, and logical partition.

In your implementation, pay careful attention to the use of constant values and proper use of memory. Do not forget to free any memory you allocated. You should especially aim to allocate only *necessary* memory and free objects (memory and files) as soon as possible.

Header and source files should contain a main comment that describes their purpose, implementation, and interface.

Source files should be commented at critical places and at function declarations. Please avoid long lines of code.

Recall to properly document each function. Properly separate public functions (declared in the header and available to other modules) from private (static) functions (declared in the source file only and unavailable to other modules).

COMPILATION

You should create your own makefile, which compiles all relevant parts of your code and creates an executable file named '`sudoku-console`'. Use the flags provided in tutorial 3 and moodle to link Gurobi properly. Your project should pass the compilation test with no errors or warnings, which will be performed by running the **make all** command in a UNIX terminal on **nova**.

GOOD LUCK