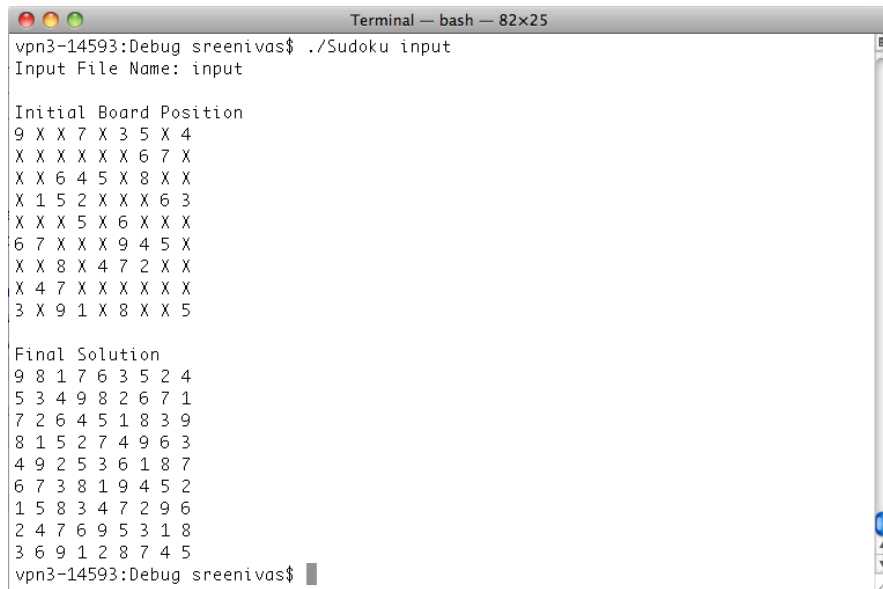


IE523: Financial Computing
Fall, 2016
Programming Assignment 2: Sudoku Solver via
Exhaustive-Search using Recursion
Due Date: 8 September, 2016
©Prof. R.S. Sreenivas

A **Sudoku Puzzle** consists of a 9×9 grid, that is subdivided into nine 3×3 blocks. Each entry in the grid must be filled with a number from $\{1, 2, \dots, 9\}$, where some entries are already filled in. The constraints are that every number should occur exactly once in each row, each column, and each 3×3 block.

Write a C++ program that takes the initial board position as input and presents the solved puzzle as the output. Your program is to take the name of the input file as a command-line input, print out the initial- and final-board positions (cf. figure 4 for an illustration). The input file is formatted as follows



```
vpn3-14593:Debug sreenivas$ ./Sudoku input
Input File Name: input

Initial Board Position
9 X X 7 X 3 5 X 4
X X X X X X 6 7 X
X X 6 4 5 X 8 X X
X 1 5 2 X X X 6 3
X X X 5 X 6 X X X
6 7 X X X 9 4 5 X
X X 8 X 4 7 2 X X
X 4 7 X X X X X X
3 X 9 1 X 8 X X 5

Final Solution
9 8 1 7 6 3 5 2 4
5 3 4 9 8 2 6 7 1
7 2 6 4 5 1 8 3 9
8 1 5 2 7 4 9 6 3
4 9 2 5 3 6 1 8 7
6 7 3 8 1 9 4 5 2
1 5 8 3 4 7 2 9 6
2 4 7 6 9 5 3 1 8
3 6 9 1 2 8 7 4 5
vpn3-14593:Debug sreenivas$
```

Figure 1: Sample output.

– there are 9 rows and 9 numbers, where the number 0 represents the unfilled-value in the initial board. Figure 2 contains an illustrative sample.

The approach you will take for this assignment is to use recursion to implement an exhaustive-search procedure that solves a Sudoku puzzle. That is, you start with the first unfilled-value in the board and pick a valid assignment that satisfies the row-, column- and block-constraints identified above. Following this, you move to the next unfilled position and do the same. If at some point in this process you reach a partially-filled board-position that cannot be filled

```

9 0 0 7 0 3 5 0 4
0 0 0 0 0 0 6 7 0
0 0 6 4 5 0 8 0 0
0 1 5 2 0 0 0 6 3
0 0 0 5 0 6 0 0 0
6 7 0 0 0 9 4 5 0
0 0 8 0 4 7 2 0 0
0 4 7 0 0 0 0 0 0
3 0 9 1 0 8 0 0 5

```

Figure 2: Sample input file called `input`, which was used in the illustrative example of figure 4. The 0's represent the incomplete board positions/values.

further under the three constraints, you backtrack to the last assignment that was made before you got stuck and modify the assigned value accordingly. This can be implemented using recursion as follows:

Boolean **Solve**(row, column)

- 1: Find an $i \geq \text{row}$ and $j \geq \text{column}$ such that `puzzle[i][j] = 0`. If you cannot find such an i and j , then you have solved the puzzle.
- 2: **for** $k \in \{1, 2, \dots, 9\}$ **do**
- 3: `puzzle[i][j] = k`.
- 4: **if** Row-, Column- and Block-Assignment constraints are satisfied by the above assignment, and **Solve**(i, j) returns *true* **then**
- 5: Return *true*.
- 6: **end if**
- 7: **end for**
- 8: { /* If we got here then all assignments made to `puzzle[i][j]` are invalid. So, we reset its value and return *false* */ }
- 9: `puzzle[i][j] = 0`
- 9: Return *false*.

Figure 3: Pseudo-code for the recursive implementation of the exhaustive-search algorithm for the Sudoku puzzle. You solve the puzzle by calling **Solve**(0,0), assuming the indices are in the range $\{0, 1, \dots, 8\}$.

First Part of the Assignment

Your implementation should use a class `Sudoku` with appropriately defined private and public functions that

1. check if the row-, column- and block-assignment constraints are met,

2. reads the incomplete puzzle from an input file that is read at the command-line,
3. prints the puzzle at any point of the search.

along with a recursive procedure that solves the puzzle that was introduced above.

Second Part of the Assignment

The following [blog](#) mentions Sudoku puzzles that can have multiple solutions. I want you to modify your code from the second programming assignment to find *all* solutions to a Sudoku puzzle.

You definitely want to be cautious with this – the empty Sudoku puzzle has theoretically 6,670,903,752,021,072,936,960 solutions. The last thing you want to do is to attempt to print them out! You will find four input files on Compass that identifies four different Sudoku puzzles. Two of them have just one solution, while the other two have multiple solutions. I suggest you run your code on these input files.

Write a C++ program that takes the initial board position as input and presents **all** solutions the puzzle as the output. Your program is to take the name of the input file as a command-line input, print out the initial- and final-board positions (cf. figure 4 for an illustration). The input file format is exactly the same as what was used for the second programming assignment.

The approach you will take for this assignment is to use recursion to modify the exhaustive-search procedure that solved the Sudoku puzzle in the second programming assignment. This is a relatively straightforward thing to do, if you have understood the recursion in the previous programming assignment.

```
Debug -- bash -- 62x38
Ramavarapus-Air:Debug sreenivas$ ./Enumerating\ Sudoku input2
Input File Name: input2

Board Position
9 X 6 X 7 X 4 X 3
X X X 4 X X 2 X X
X 7 X X 2 3 X 1 X
5 X X X X X 1 X X
X 4 X 2 X 8 X 6 X
X X 3 X X X X X 5
X 3 X 7 X X X 5 X
X X 7 X X 5 X X X
4 X 5 X 1 X 7 X 8

Solution #1
Board Position
9 2 6 5 7 1 4 8 3
3 5 1 4 8 6 2 7 9
8 7 4 9 2 3 5 1 6
5 8 2 3 6 7 1 9 4
1 4 9 2 5 8 3 6 7
7 6 3 1 4 9 8 2 5
2 3 8 7 9 4 6 5 1
6 1 7 8 3 5 9 4 2
4 9 5 6 1 2 7 3 8

Solution #2
Board Position
9 2 6 5 7 1 4 8 3
3 5 1 4 8 6 2 7 9
8 7 4 9 2 3 5 1 6
5 8 2 3 6 7 1 9 4
1 4 9 2 5 8 3 6 7
7 6 3 1 9 4 8 2 5
2 3 8 7 4 9 6 5 1
6 1 7 8 3 5 9 4 2
4 9 5 6 1 2 7 3 8
Ramavarapus-Air:Debug sreenivas$
```

Figure 4: Sample output.