Project 3 - 16-by-16 Sudoku Solver

Due May 12 by 11:59pm **Points** 30 **Submitting** a file upload **File Types** h and cpp **Available** until May 15 at 12:01am

This assignment was locked May 15 at 12:01am.

ST. CHARLES COMMUNITY COLLEGE

CPT-182 - Programming in C++

Programming Project - 16-by-16 Sudoku Solver (30 Points)

(Number in Question Bank: Assignment **6.1**)

You **must** study **Project Demo 2** before working on this project.

Project Overview

In this project, you are going to use **recursion** to solve a variant of **Sudoku puzzles**. Rather than the standard **9**-by-**9** version, your program will solve **16**-by-**16** (hexadecimal) Sudoku, in which each cell is filled with a hexadecimal digit. There are **16** elements in each row and column, and each grid is **4**-by-**4**. Your program will read a Sudoku puzzle from an input file and output all the solutions of the puzzle to an output file.

Basic Rules of the Sudoku Puzzle

1) The **16** hexadecimal digits are **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **a**, **b**, **c**, **d**, **e**, and **f**. In this project, letter digits (**a** to **f**) are lowercase letters.
2) Hexadecimal Sudoku is solved on a **16**-by-**16** board, which is further divided into **4**-by-**4** grids. Each cell in the table contains one hexadecimal digit, subject to the following rules:
   - Each digit appears **exactly once** in each row.
   - Each digit appears **exactly once** in each column.
   - Each digit appears **exactly once** in each **4**-by-**4** grid.
3) A **sample puzzle** is given below. Your goal is to fill all the blanks with appropriate digits.

b	2		7				f	6	e				3		d
			9	b		2						f	1		6
	a		c			7		4		8				e	
	8		3			1				f	c	a		4	2
		e						1	6	a				3	
3									4		f	8		6	b
			d	2					b			7	4		
	b	8	4	0			d	7		9	3	e			
			f	3	7		4	5			9	0	d	c	
		c	8			a					b	4			
4	9		5	e		d									a
	d				6	c	1						7		
9	1		e	7	4				f			3		d	
	3				9		2		1			5		0	
f		4	b						3		d	1			
5		d				f	e	9				2		b	4

Explanation of the Recursive Solution

To see how the puzzle is solved, consider the leftmost empty cell in the top row (between **2** and **7**). Based on the digits already placed in the same row, column, and grid, we know that the digit in this cell **cannot** be ~~**2**~~, ~~**3**~~, ~~**4**~~, ~~**6**~~, ~~**7**~~, ~~**8**~~, ~~**9**~~, ~~**a**~~, ~~**b**~~, ~~**c**~~, ~~**d**~~, ~~**e**~~, or ~~**f**~~, because these digits already appear in the same row, column, or grid. Therefore, there are **3** candidates to fill this cell, **0**, **1**, and **5**. As a brute force solution, we can first try to place **0** in this cell and then check if it leads to a contradiction or other program later; if so, we can change the digit to **1** and try again; if that does **not** work, then change it to **5** and try again. This general approach is called **depth-first search with backtracking**.

An important observation is that this approach transforms the problem of solving a Sudoku with n empty cells into a problem of solving a Sudoku with $n-1$ empty cells; it reduces the size of the problem, but it does **not** change the logic of the problem. Therefore, this problem is **recursive**.

For Sudoku purists, a puzzle should be designed so that there is only one possible solution, and that solution should be possible to deduce purely with logic, with **no** ~~trial guesses~~ needed. However, many published puzzles have more than one solution and removing an extra digit or two can dramatically increase the number of possible solutions. Therefore, in this project, your program should output all the possible solutions for a given Sudoku puzzle.

The Input File

- The input file is a **plain text file** (filename: `puzzle.txt`).
- There is **only one** Sudoku puzzle stored in the input file. However, the puzzle may have more than one solution.
- The input file contains **16** lines of **16** characters each. A **dot** (`'.'`) is used to indicate a blank cell. Therefore, the **sample puzzle** would be stored in the input file as below:

```
b2.7...f6e...3.d
...9b.2.....f1.6
.a.c..7.4.8...e.
.8.3..1...fca.42
..e.....16a...3.
3.......4.f8.6b
...d2....b..74..
.b840..d7.93e...
...f37.45..90dc.
..c8..a....b4...
49.5e.d........a
.d...6c1.....7..
91.e74...f..3.d.
.3...9.2.1..5.0.
f.4b.....3.d1...
5.d...fe9...2.b4
```

- Please refer to the **sample input files** to better understand the input file format.

The Output File

- The output file is a **plain text file** (filename: `solutions.txt`).
- Your program will write all the solutions of the puzzle in the input file to the output file.
- Each solution should be presented the same as in the input file with all the blanks appropriately filled.
- Each solution needs to be **numbered** (e.g., `"Solution 1"`, `"Solution 2"`, and so on).
- Please refer to the **sample output files** to better understand the output file format.

Other Development Notes

- Your solution **must** be **recursive** to get credit.
- Your recursive method to solve the puzzle will determine which values are legal for a given cell, and then systematically try all possibilities for all blank cells.
- The recursive method to solve the puzzle is **not** going to require ~~a huge amount of code~~, but think carefully about what the parameters of your recursive method should be and how it will determine that a solution has been found. Hint: how many **unfilled** cells are there?

Sample Input and Output Files (Click the Filename to Download)

Input File	Number of Solutions	
puzzle_1.txt (https://drive.google.com/uc?export=download&id=1wCqRM8Rh1kUpfq9g6_-xf055ZmmJN4AZ)	**1** solution	solutions_1.txt
puzzle_2.txt (https://drive.google.com/uc?export=download&id=1vlaiutHCR_BzEpa7cFO7j4XQsB1ljYhe)	**4** solutions	solutions_2.txt
puzzle_3.txt (https://drive.google.com/uc?export=download&id=1w0aAw-KUQGuE0-kp-SE5d3q88zK4kw7n)	**No** solution	solutions_3.txt
puzzle_4.txt (https://drive.google.com/uc?export=download&id=1w9PA3-T8BJdqGiHtxTsUP3-guFVUm8Rs)	**5** solutions	solutions_4.txt
puzzle_5.txt (https://drive.google.com/uc?export=download&id=1vuZ81xeWA3u1f6ZXm4zT724dmSuxAIFp)	**13** solutions	solutions_5.txt
puzzle_6.txt (https://drive.google.com/uc?export=download&id=1vsoavYHXA3jPdNWxzBIbRXWa_C-PIRla)	**95** solutions	solutions_6.txt
puzzle_7.txt (https://drive.google.com/uc?export=download&id=1w3ygdAxOMhJuhSx1T487x0I63XyJ_zDF)	**28** solutions	solutions_7.txt

puzzle_8.txt (https://drive.google.com/uc?export=download&id=1wEJaLp0KHM6-YuoZco5XowbeCtxRSGwa)	**1** solution	solutions_8.txt

Project Submission and Grading (Please Read)

- Please upload all your **.h** (if any) and **.cpp** files (**not** the ~~entire Microsoft Visual Studio project folder~~) on Canvas.
- Before the project deadline, you can submit your work unlimited times. However, only your latest submission will be graded.
- At least **20**% of your code should be **comments**. All variable, function (if any), and class (if any) names should "make good sense". You should let the grader put **least effort** to understand your code. Grader will **take off points**, even if your program passed all test cases, if he/she has to put extra **unnecessary** effort to understand your code.
- Please **save a backup copy** of all your work in your computer hard drive.
- Your program will be graded (tested) using another valid input file (still named **puzzle.txt**) to check whether it can generate the expected (correct) output file (with correct format and correct output values in it). As long as the input file is valid, your program should generate a correct output file. In other words, your program should work for **any** valid input file, **not** just the sample input files provided in the project instructions.
- In this class, you can assume that the input file (input data) is always **valid** and **has correct format**. You do **not** need to deal with ~~invalid input~~ or ~~error handling~~.
- Your work will be graded after the project deadline. All students will receive their project grades at (almost) the same time.