School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2021
Assignment 1

Learning Outcomes

In this assignment you will demonstrate your understanding of arrays, strings, functions, and the `typedef` facility. You must *not* make any use of `malloc()` (Chapter 10) or file operations (Chapter 11) in this project. You *may* use `struct` types (Chapter 8) if you wish, but are not required to. (Plus, you might even learn something about sudoku puzzles.)

Sudoku!

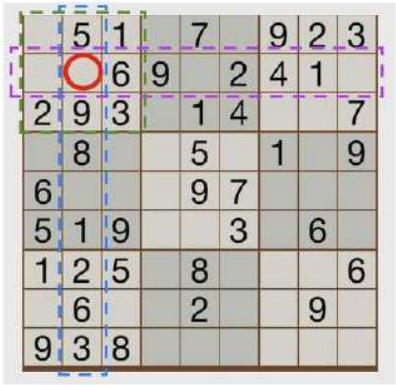

Figure 1: An "Easy" sudoku.

The goal of sudoku is to complete a square grid of $9 \times 9 = 81$ cells using only the numbers from 1 to 9, in a way that has exactly one instance of each digit appearing in each column, in each row, and in each 3×3 sub-square. For example, in Figure 1 the cell marked with the red circle must be a "7", since every other possible value appears in at least one of the three sets (row, purple; column, blue; and sub-square, green) that cell is a member of. All of the other cells in the example can be determined using the same strategy.

Your task in this assignment is to develop a *sudoku assistant*, to help people solve their sudoku puzzles. You will use a one-dimensional C array to store a linearized version of the sudoku matrix (see Figure 4), and will write functions that carry out useful operations, including checking for numbering violations, identifying any "it must be that value" cells, and perhaps even solving some easy puzzles through to completion.

Start by copying the skeleton program `ass1-skel.c` and sample data file `data1.txt` from the LMS Assignment 1 page, and check that you can compile the program via either `grok` or `gcc`. Note that if you plan to use `grok`, you will need to learn how to execute programs in `grok` via the "terminal" interface that it provides. There is a handout linked from the LMS that provides guidance.

The skeleton file provides an empty main program that includes a very important *authorship declaration*. Substantial mark penalties apply if you do not include it in your submission, or do not sign it. The skeleton program also includes two very useful pre-filled "constant" matrices. Make sure that you understand their roles, described below (and in more detail in the Assignment 1 video). Those global matrices are intended to be constant, and you should not change their values in any way.

Stage 1 – Reading and Printing (12/20 marks)

The input to your program will always be a set of 81 one-digit numbers, laid out as a sudoku grid, and with "0" used to represent empty cells. For example, the sudoku in Figure 1 is represented in the test file `data1.txt` in the manner shown on the left of Figure 2. You should read the input using a sequence of 81 calls (in a loop, of course) to `scanf("%d", &val)`, to avoid the need to deal with blank and newline characters.

You are encouraged to use a one-dimensional array of 81 values in your program, rather than a 9×9 two-dimensional array; and to map the sudoku grid into that array in *row-major order*, as shown in Figure 4. Using a one-dimensional arrangement allows you to employ single-nested loops instead of always needing double loops, and to make use of the constant arrays that are part of the

```
0 5 1   0 7 0   9 2 3      . 5 1 | . 7 . | 9 2 3      0 5 1   0 7 0   9 2 3
0 0 6   9 0 2   4 1 0      . . 6 | 9 . 2 | 4 1 .      0 0 6   9 0 2   4 1 0
2 9 3   0 1 4   0 0 7      2 9 3 | . 1 4 | . . 7      2 9 3   0 1 4   0 0 7
                           ------+-------+------
0 8 0   0 5 0   1 0 9      . 8 . | . 5 . | 1 . 9      0 8 0   0 5 0   1 0 9
6 0 0   0 9 7   0 0 0      6 . . | . 9 7 | . . .      6 0 0   0 9 7   0 0 0
5 1 9   0 0 3   0 6 0      5 1 9 | . . 3 | . 6 .      5 1 9   0 0 3   0 6 0
                           ------+-------+------
1 2 5   0 8 0   0 0 6      1 2 5 | . 8 . | . . 6      1 2 5   2 8 0   0 0 6
0 6 0   0 2 0   0 9 0      . 6 . | . 2 . | . 9 .      0 6 0   0 2 0   0 9 0
9 3 8   0 0 0   0 0 0      9 3 8 | . . . | . . .      9 3 8   9 0 2   2 0 9

                           41 cells are unknown
```

Figure 2: Input `data1.txt` (left); corresponding Stage 1 output (center); input `data2.txt` (right).

skeleton program: `c2s[]` (cell-to-set) has eighty-one rows and three columns, with `c2s[c]` listing the ordinal numbers of the three sets that include `c`; and `s2c[]` (set-to-cell) has twenty-seven rows and nine columns, with `s2c[s]` listing the cells that make up set number `s`. For example, in Figure 4 cell number 51 is a member of set number 5 (a row-based set); of set number 15 (a column-based set); and of set number 23 (a square-based set). That means that the three elements in `c2s[51]` will be 5, 15, and 23. In the inverse mapping, `s2c[15]` contains the nine cells in set 15, namely 6, 15, 24, 33, 42, 51, 60, 69, and 78. You'll need to be able to manipulate these two arrays in Stage 2 and Stage 3.

The required output from a Stage 1 program is a neatly-formatted representation of the input data, plus a summary line, see the center part of Figure 2. Note that "`%NDIM`" and "`%NDIG`" and `%(NDIM*NDIG)` operations can be used to identify places where formatting is required to output the required grid.

Stage 2 – Grid Checking (16/20 marks)

Extend your program so that it checks the input grid for mistakes, to ensure that none of the 27 sets in the supplied puzzle have duplicate elements. Report any errors according to their set number, from 0 to 26, and also by the set type ("row", "col", or "sqr") and the ordinal (counting from one) number of that set. For example, on the (incorrect) input file `data2.txt` (see the right side of Figure 2), the Stage 2 output must be as shown in Figure 3. Other examples are available on the LMS. If any errors are detected in the input, your program should exit once it has printed out its Stage 2 diagnostic messages.

```
set  6 (row 7): 2 instances of 2
set  8 (row 9): 2 instances of 2
set  8 (row 9): 3 instances of 9
set 12 (col 4): 2 instances of 9
set 14 (col 6): 2 instances of 2
set 17 (col 9): 2 instances of 9
set 25 (sqr 8): 3 instances of 2
set 26 (sqr 9): 2 instances of 9

7 different sets have violations
8 violations in total
```

Figure 3: Stage 2 output (`data2.txt`).

Stage 3 – Implementing Strategy One (20/20 marks)

Solving a sudoku involves application of several different *strategies*. Your program must implement the simplest of those, let's call it "Strategy One", which is sufficient to solve "Easy"-grade sudokus. (Further strategies must be added before even "Medium"-grade puzzles can be solved. Your program will *not* be able solve all sudokus.) The red circle in Figure 1 is an example where Strategy One can be applied to determine a cell label.

In Strategy One, each cell `c` should be considered in turn. The three sets that cell is a member of (found using array `c2s[c]`) should be examined, accumulating label counts from the (up to) 24 labeled cells in those three sets (found using array `s2c[c]`). Each of those labels is a number between

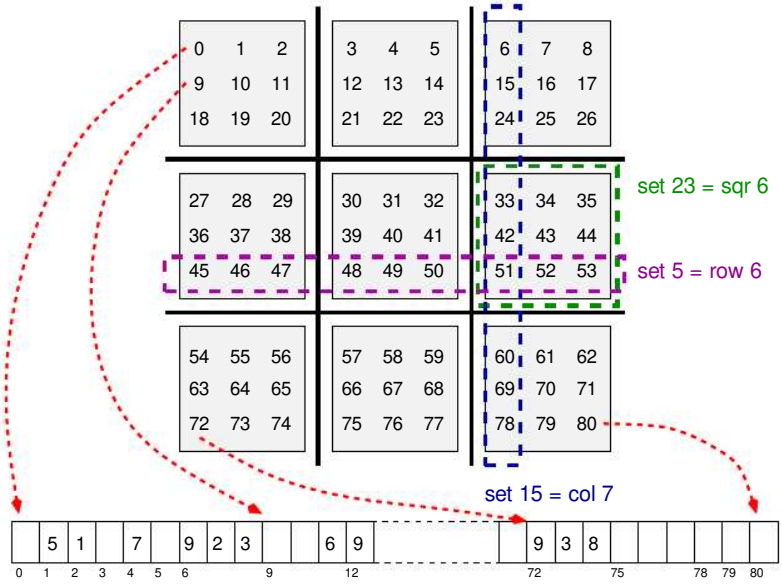

Figure 4: The linearization that is strongly recommended. The two-dimensional grid at the top shows the row-major mapping; the one-dimensional array at the bottom is what actually stores the sudoku grid's values. The example shows the sudoku from Figure 1.

1 and 9. Hence, if there is only one of those nine possible numbers that does not yet occur anywhere in `c`'s three sets, then it must be the label for cell `c`.

To implement this strategy you should iterate over all cells, in row-major order, computing the valid options for each cell. Then go through again, using that information, looking for cells that only have one valid option available, assigning those labels. If any cells did get assigned during that pass, you should recompute all the options, and make another pass. Each pass is shown separately in the output (see Figure 5). Only when a complete pass takes place with no new cells being labeled, or when every cell has been labeled, should your program print the current state of the puzzle, and then end. If the sudoku has been solved, your program should celebrate by also printing "`ta daa!!!`", as shown in Figure 5. The Assignment 1 LMS page has detailed examples showing the output that is required for a range of input files.

If you want a challenge you can also implement "Strategy Two", which determines which values remain possible options for every unlabeled cell `c`, eliminating based on the labeled cells in each of `c`'s three sets; and then checks each of the 27 sets in order. If a set has only one unlabeled cell able to take on one of that set's unused labels, then that label can be assigned to that cell (see the Assignment 1 video). Strategy One and Strategy Two should be alternated, iterating until no further changes occur. This combined approach will be able to solve puzzles beyond the ability of Strategy One alone, but will not earn any more marks. *Only consider implementing Strategy Two when your Strategy One is working.*

```
strategy one
row 2 col 2 must be 7
row 2 col 5 must be 3
row 4 col 6 must be 6
row 5 col 2 must be 4
row 6 col 5 must be 4
row 7 col 6 must be 9

strategy one
row 1 col 6 must be 8
row 2 col 1 must be 8
row 4 col 4 must be 2
row 5 col 3 must be 2
row 9 col 5 must be 6
```

[*and so on, many more lines*]

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

 0 cells are unknown

ta daa!!!
```

Figure 5: Stage 3 (Strategy One) output for file `data1.txt`.

Debugging...

You will probably find it helpful to include a `DEBUG` mode in your program that prints out intermediate data and variable values. Use `#if (DEBUG)` and `#endif` around such blocks of code, and then `#define DEBUG 1` or `#define DEBUG 0` at the top. Turn off the debug mode when making your final submission, but leave the debug code in place. The FAQ page has more information about this.

Boring But Important...

This project is worth 20% of your final mark, and is due at **6:00pm on Friday 17 September**.

Submissions that are made after the deadline will incur penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to obtain a letter from them that describes your illness and their recommendation for treatment. Suitable documentation should be attached to **all** extension requests.

You need to submit your program for assessment **via the LMS Assignment 1** page. Submission is **not possible through grok**. There is a pre-submission test link also provided on the LMS page, so you can try your program out in the test environment. **It does not submit your program for marking either.** Note that this pre-test service will likely overload and fail on the assignment due date, and if that does happen, it will not be a basis for extension requests. *Plan to start early and to finish early!!*

Multiple submissions may be made; only the last submission that you make before the deadline will be marked. If you make any late submission at all, your on-time submissions will be ignored, and if you have not been granted an extension, the late penalty will be applied.

A rubric explaining the marking expectations is linked from the LMS, and you should study it carefully. Marks will be available on the LMS approximately two weeks after submissions close, and feedback will be mailed to your student email account.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** have any "accidents" that allow others to access your work; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrolment terminated for such behavior.

The FAQ page contains a link to a program skeleton that includes an Authorship Declaration that you must "sign" and include at the top of your submitted program. Marks will be deducted (see the rubric linked from the FAQ page) if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action.

Nor should you post your code to any public location (`github`, `codeshare.io`, etc) while the assignment is active or prior to the release of the assignment marks.

And remember, algorithms are fun!