COMP 480, Spring, 2023
Program #4 (pa4),
Date Assigned: Friday, April 14
Date Due: Friday, May 5, 11:59PM
Possible Points: 100

1 Initial Setup

Both you and your partner will need to clone the starter code for your group using git. The following assumes that you are using VSCode as your IDE. If you are using some other IDE, follow the instructions for that IDE to clone a repository, or just use command line git ("git clone url").

1. You will need your group number in what follows. Get this from the "Programming Partners" file under the "Programming Assignments" tab on Blackboard. This file also has your assigned partner.
2. In VS Code, open the command palette and select the "Git Clone" option.
3. When prompted for the repository URL, enter the following, with X replaced by your group number (e.g. 7 or 12):
   ssh://git@code.sandiego.edu/comp480-sp23-pa4-groupX
4. Choose the "Open Repository" option in the window that pops up in the lower-right corner of the screen. The repository should contain the following files:
   - `pa4.py`: You will write your code in this file. It includes the header for the `solve(size, filename)` function that you must write.
   - `test_pa4.py`: This program will test your `pa4.py` program. See the discussion below on this.
   - `pi.txt` for `i = 1, ....`: Test input files.
   - `piSol.txt` for `i = 1, ...`: Solution to test input files.
5. Each programming team will have it own repository that has been initialized with the same starter code, so when you sync your code, you are sharing with your partners, but with no one else in the class. (I can see your repository also.)
6. Remember that if you close VSCode, then when you reopen it, you should see your repository. But if you don't see it, then just select `File->Open`, and then select the directory containing your repository.
7. I also recommend that you stage changes, commit those changes, and sync the changes periodically as you are working on the program, and certainly when you are done with a session with your programming partner. This ensures that you won't lose any of your work in case your computer gets lost or a file gets accidentally deleted.

2 Program Specifications

In the `solve(size, filename)` function in `pa4.py`, you will solve a Sudoku problem. You may be familiar with this game, but if not (or if you've forgotten because you switched to Wordle), here is an example input of a problem instance:

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

This is a 9×9 grid of cells where some of the cells have been seeded with values that are in the range 1 to 9 (the same 9 as the size of the grid). The problem to solve is then to fill in the remaining cells with values (each in the range 1 to 9) such that

1. Each row contains one and only one instance of each of the numbers 1 through 9.
2. Each column contains one and only one instance of each of the numbers 1 through 9.
3. Each of the nine 3×3 submatrices (outlined by the darker boundary lines) contains one instance of each of the numbers 1 through 9.

Here is the (unique) solution to this problem:

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

For any given problem, there are three possibilities for a solution:

1. There is one unique solution (as for the above example). Most problems you encounter online or in books will have one unqiue solution.

2. There are multiple solutions.

3. There are no solutions.

The above example is a 9×9 Sudoku problem, which is the most common. But you can also have

1. 16×16 problems. Here, there must be 16 different values, and so we start using letters: the values that must appear in each row, column, and submatrix are 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G. (Capitalize the letters.) And the submatrices for a 16×16 problem are 4×4.

2. 25×25 problems. Here, there must be 25 different values, and so we use just letters: the values that must appear in each row, column, and submatrix are A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y. (Capitalize the letters.) And the submatrices for a 25×25 problem are 5×5.

In principle, the problem size can be any square of an integer greater than 1, but for this assignment, the test inputs will only be problems of size 9, 16, and 25.

As said above, you will write the `solve(size, filename)` function in `pa4.py`, to solve a Sudoku problem. The parameters are:

1. `size`: The size of the problem to be solved. This will either 9, 16, or 25. (9 meaning a 9×9 Sudoku problem, and so on.)

2. `filename`: The name of the file containing the input to the problem instance. The format of the file is:

   (a) It is a text file.

   (b) There is one line in the file for each seed value in the input. The format for a line is:

   ```
   row col value
   ```

   where `row` is the row index of the value in the Sudoku grid, `col` is the column index. and `value` is the seed value for that row and column.

   (c) The three values on each line are separated by one or more whitespace characters.

   (d) row and column indices in the Sudoku grid are numbered starting at 1. The indices are $1, 2, \ldots, \text{size}$.

   (e) Because size 16 and 25 problems have non-numeric values, all values should be handled as single characters (so, for example, the character '1' rather than the integer 1).

   (f) The seed values can be specified in any order. (They do not necessarily appear, for example, row by row, or column by column.)

   The input file `in1.txt` in your program repository corresponds to the example problem shown above.

`solve(size, filename)` should return a tuple, which I'll refer to as `solution` here. `solution` should contain:

1. `solution[0]`: The solution to the problem:
   (a) `None` if the problem is infeasible.
   (b) If the problem is feasible, return a nested list containing the cell values for your solution. The row should be the first index of the nested list, and the column second. Remember that in the Sudoku grid, row and column indexing starts at 1, whereas in the list indexing starts at 0. So row index 0 in the list corresponds to row 1 of the Sudoku grid, and so on. (And same with column indexing.) The values in the list should be characters. (So, for example, the character '4' rather than the integer 4.) When an value is a letter, it should be capitalized. Most of the test inputs have unique solutions, but at least one is not unique. In the case of multiple solutions, you just have to return one, and it does not have to match the one in the solution file for the test - the test code checks your solution for correctness.
2. `solution[1]`: The number of nodes that you generated in traversing the recursion tree while computing your solution. This is explained further down in this document. There is no correct answer to this part of your solution, as this value can depend on the implementation, and can even vary from run to run if certain data structures are used that exhibit nondeterministic behavior.

3 Program requirements

- First, let me remind you that the code you turn in should be written by you and your programming partner only. Searching the web for a solution and using all or part of it is considered a violation of academic integrity, as is getting coding help from someone outside of your group. - see the discussion about academic integrity in the syllabus.

- Your program must implement a backtracking solution. (See lectures on Friday, April 14 and Monday, April 17, and the reading assignment from Wednesday, April 5, which asked you to read the backtracking chapter from the supplemental textbook.)

  In this problem, backtracking involves recursively searching all possible assignments of values to all grid cells for a solution, which is an assignment that satisifies the requirements specified above. Recall that the execution of a recursive function can be visualized with a recursion tree. In this case each node of the recursion tree represents a partial or full assignment of values to the cells of the grid. A partial assignment of values to the cells means that some cells in the grid have their value specified, and some not.

The child nodes of a node v in the recursion tree represent value assignments where one new value (in some cell that does not have a value in the assignment in node v) is added to the values specified by the node v. Given a node v, there are different ways to choose children nodes of v – that is, there are different ways to branch on v. Here is how you will do it: you will choose a cell that does not have a value in v, and branch on all possible values that that cell can be given. In the case of 9×9 Sudoku, this means 9 child nodes, one associated with each possible values to give to the chosen cell.

The root node of the recursion tree is the partial assignment associated with the values specified by the problem in the ijnput file.

Leaf nodes in the recursion tree represent full assignments of values to all cells in the grid. Once all values have been assigned, you can check if the assignment is a solution (it meets all of the requirements for a solution).

Naively implementing this recursive approach gives a program that will not terminate in a reasonable amount of time. A typical "evil" 9×9 Sudoku problem has 23 values specified in by the problem, leaving 58 unspecified values. So the depth of the recursion tree is 58, with branching factor 9 – I'll leave the math to you. So this is where backtracking comes in. When you are at a node v and are ready to recursively visit children nodes of v, you only generate nodes associated with values that could lead to a solution. For example, if say you are branching on values to be assigned to the cell in row r and column c, and say the value 4 already appears in row r or column c of the partial assignment in node v, then there is no way that assigning 4 to the cell in c, r can lead to a solution, and so you do not generate that child. This eliminates an entire subtree from consideration.

How clever you are at deciding if a proposed value in a cell can be eliminated will determine how frequently you can cutoff traversing subtrees. Here, you might bring in your knowledge/strategy of playing Sudoku on pencil and paper. But be judicious about which strategies are worth implementing in your code, and look for strategies that you would not do on pencil and paper, but that software can do easily.

With the possibility of cutoffs, now leaf nodes in the recursion tree can represent either full assignments of values to cells, or partial assignments where there are no children nodes with assignments that can lead to a solution.

Using recursion implies that you are doing a depth-first traversal of the recursion tree.

Your code is required to count (and report) the number of nodes in the recursion tree that were generated. This is a measure of how good your code is a evaluating prospective values for being cutoff.

No points are given if you do not implement this strategy.

- Your program must pass all tests when you run the test program `test_pa4.py`. No points are given if your program does not pass all tests.

- Your program must run in the alloted time, as indicated by the `TARGET_RUN_TIME` variable in the test program `test_pa4.py`. No points if you do not meet this target. (If you think you're missing it because you have an old slow computer, let me know and I can test it on mine.)

- Your program should follow the programming style guidelines contained in the "programmingStyleGuidelines.txt" document in blackboard on the "Programming Assignments" tab. A maximum of 10 points will be taken off for failure to follow these guidelines.

- Submit your program by the due date/time. The policy for late programs is in the course syllabus. (As I did for pa2 and pa3, I will reduce the first class late penalty to 2.5%, with subsequent late classes the usual 5%.

4 Program comments/suggestions/hints

- When traversing the recursion tree, it can be tedious to make changes to your data structures when going down the recursion tree, and the undoing when returning back up the tree. I suggest that you define a class to store all of the data you need to represent a single node of recursion tree. Then, when you make a recursive call – that is, when you go down one level in the recursion tree – you make a copy of the data structure for the child node, and you are free to modify that as needed without worrying about undoing any changes you make. Python has a module `copy` that has a function `deepcopy` that returns a deepcopy of its parameter. `deepcopy` actually makes copies of data structures, rather than copying just the references, and it does this recursively through all data structures in the parameter object.

5 Submitting your program

To submit your program for grading, be sure that you've pushed it the repository, and post a note on campuswire, category `pa4_submit`, giving just your group number. (Questions about pa4 should go to category pa4.) Your submission time will be considered to be the time when you last sync your program to the server. So once you've turned in your program, don't do any further sync's.