COL100 Assignment 8

2^{nd} Semester Semester : 2021-2022

Deadline: 11:59 pm, 28th May, 2022

General Instructions

You should attempt this assignment without taking help from your peers or referring to online resources except for documentation (we will perform a **plagiarism check** amongst all submissions). Any violation of above will be considered a breach of the honor code, and the consequences would range from **zero marks** in the assignment to a **disciplinary committee action**.

Submission Instructions

1. Your code must be in a file named `<EntryNo>-q.py`
2. The evaluation for a lab will be done in the lab only and evaluations not done for the in-lab component for a student in his/her lab slot will be marked 0. It is your duty to get them evaluated.
3. Submit your code in a `.zip` file named in the format `<EntryNo>.zip`. Make sure that when we run unzip `<EntryNo>.zip`, a folder `<EntryNo>` should be produced in the current working directory. For eg. if your entry number is `2021CS5XXXX`, then your zip file would be `2021CS5XXXX.zip` and upon unzipping, it should produce a folder `2021CS5XXXX` containing file `<EntryNo>-q.py`.
4. Your submissions will be **auto-graded**. Make sure that your code follows the specifications (including directory structure, input/output, importing libraries, submission `.zip` file) of the assignment precisely.

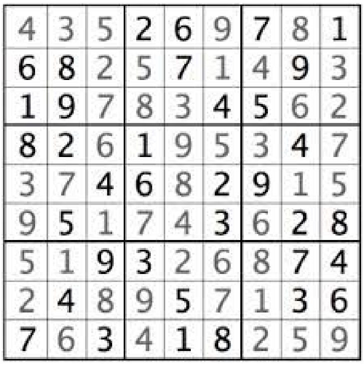
Some Clarifications

1. You must not change the parameters, return type and the names of the functions provided in the skeleton code unless clearly specified.
2. If you still have any more doubts, feel free to shoot them at Piazza.
3. You can only use lists or arrays to store the data in the questions.
4. Remember to **not to import** anything and you can use common inbuilt functions like append/split/len/etc freely.

In this assignment you will be writing a program that will help solving Sudoku. You will be implementing functions that solve some small functionality, which will come together to create a Sudoku solver.

In the previous assignments the In-Lab Component was independent of the rest of the Assignment but here you will have to use the functions done in lab for the other functions therefore please upload them on Gradescope for further use. This assignment may seem slightly daunting at first but the functions are fairly small so please manage your time efficiently as there will be no extensions. Most functions would need less than 10-15 lines of code, if you find yourself writing more than 30-40 lines please rethink as you might be doing something wrong or you might do better in implementing them.

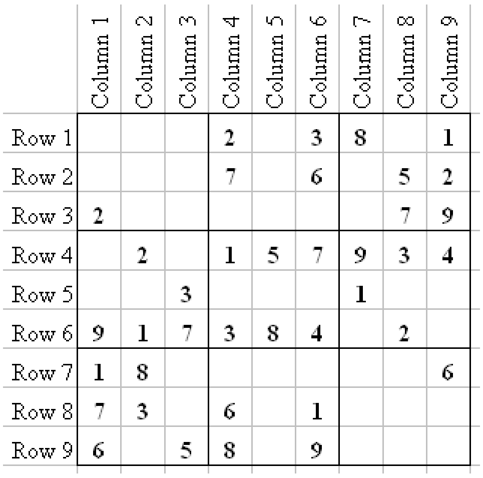
A **Sudoku** is a puzzle with 81 numbers which is the form of a 9×9 grid that is broken into rows, columns and blocks with the constraint that each row, column and block has unique elements from 1 to 9 i.e. no row,column or block has a repeating element. **Each cell has one number inside it and there is only one valid solution for a Sudoku**. Look at Figure 1 for a solved Sudoku.



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

Figure 1: Solved Sudoku

If you look at fig. 1, each row, column and block have 9 values. All of them are unique and non repeating. The task is that you will be given a partially filled Sudoku with only one valid solution. Your task will be to solve the Sudoku using backtracking.



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

Figure 2: Row and Column indexing

Some keywords that we have used -

1. **Row:** It is a row in the grid, it's numbering starts from 1 and goes till 9. Refer to Figure 2 for some more clarity.

2. **Column:** It is a column in the grid, it's numbering starts from 1 and goes till 9. Refer to Figure 2 for some more clarity.

3. **Block:** A Block is a 3×3 grid with 9 numbers. Each number inside the grid is unique. A Sudoku is made of 9 blocks the numbering can be seen in Figure 3. $B1$ denotes the first block.

4. **Position**: Denoted by *pos* in the skeleton code. It defines the position of a cell, it is a pair of two integers i.e. $(row, column)$. It starts with $(1, 1)$ for the first cell and the last cell will have position $(9, 9)$. In fig. 1 the value at $(2, 2)$ is 8.

5. **List:** It is a list of 9 numbers. It can be a block, a row or a column.

A Sudoku is denoted by a list of lists, with each sub-list inside the outer list as a row of the Sudoku. Each position inside the list of list is an integer. **If there is a 0 present it means that position is**

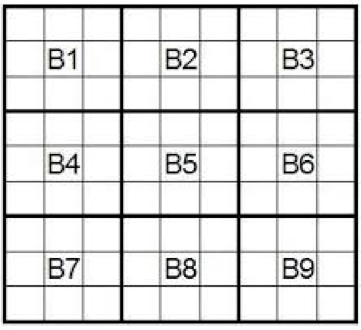


Figure 3: Block indexing

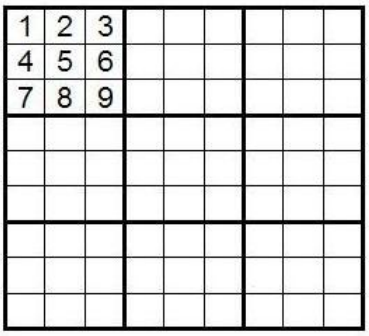


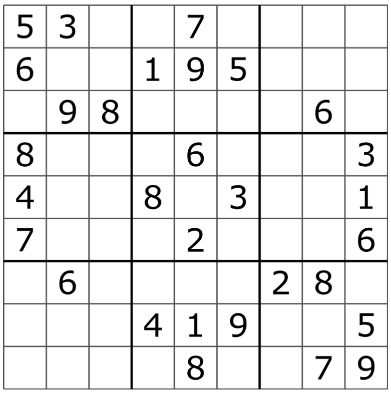
Figure 4: Element indexing inside block

empty.

If you look at **Figure 1** the representation of that Sudoku in python would be the following - `[[4,3,5,2,6,9,7,8,1],[6,8,2,5,7,1,4,9,3],[1,9,7,8,3,4,...],...]` and the representation of **Figure 5** will be - `[[5,3,0,0,7,0,0,0,0],[6,0,0,1,9,5,0,0,0],[0,9,8,0,0,0,...],...]`

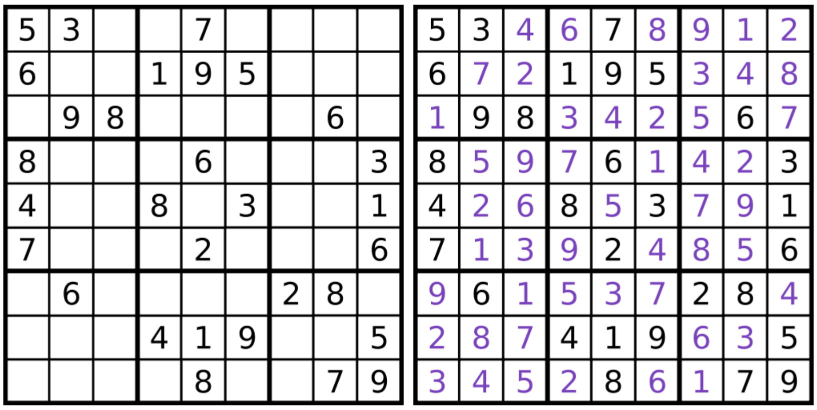
Note: The size of the outer list will be 9 and the size of all the inner lists will be 9 as well.

One way to solve a Sudoku is to list all the possible combinations and then check which of them satisfy the constraints. The other algorithm is using this paradigm called Backtracking. Backtracking is an algorithmic technique for recursively solving problems by attempting to develop a solution progressively,



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

Figure 5: Unsolved Sudoku



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

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

Figure 6: Before and After solving the Sudoku

one piece at a time, and discarding any solutions that do not satisfy the problem's criteria at any point in time. The function `Sudoku_solver()` will use this backtracking algorithm

You only have to implement several functions given in the skeleton code provided to you. Please download that from Gradescope and fill in the functions. Please read the comments of each function very carefully before implementing it. You can't change the name, return type and parameters of the functions.

Important Note:

1. All the indexing is from 1 to 9 but indexing in arrays, tuples and lists is from 0 to 8 so please make sure you keep this in mind while implementing your code.
2. You are **not** supposed to print any thing **inside** the functions.
3. All functions will be graded separately but make sure the basic functions work because the later ones will call them and if they have an error you will fail the later functions as well.
4. For all of the functions **only valid input** will be given, no edge-cases or invalid inputs will be checked.
5. Each function has a level and a list of dependencies which include the functions which have to be implemented **before** that particular function.

Here is the list of the functions along with their descriptions :

Functions to be implemented: The following functions are need to be implemented.

`FUNCTIONS`:

1. `get_block_num(sudoku:List[List[int]], pos:Tuple[int, int]) -> int:`
   This function takes two parameters position and sudoku and returns the block number of the block which contains the position.

   For example- If the position given is `(4,6)` then we return 5 as `(4,6)` is present in block number 5. (See fig. 2 and 3 for indexing reference). For the input `(1,1)` the returned value is 1, for `(3,5)` the returned value is 2 and for `(1,4)` the block is 2. Remember that the position is defined as `(row,column)` and indexing starts from 1.

Hint: Try using modulo operator instead of using if-else-statements

Level: Easy

2. `get_position_inside_block(sudoku:List[List[int]], pos:Tuple[int, int]) -> int`
   This function takes two parameters position and sudoku and will return the index of the position **inside the corresponding block**.

   For example, If the position is given as `(4,6)` then we return 3 as `(4,6)` position is at the index 3 of the block 5 (See fig 2, 3 and 4 for indexing reference). For position `(1,1)` the returned value is 1 as its at the 1^{st} place in the 1^{st} block, similarly `(1,2)` will return 2 and `(1,3)` will return 3. Note that `(1,4)` will again return 1 as its at the 1^{st} position of the 2^{nd} block. Remember that the position is defined as `(row,column)` and indexing starts from 1.

   Dependencies: `get_block_num()`

   Level: Easy

3. `get_block(sudoku:List[List[int]], x:  int) -> List[int]`:
   This function takes an integer argument i and then returns the i^{th} block of the Sudoku. See fig. 1 for block indexing. **Note** that block indexing is from 1 to 9 and not 0-8 thus the input x will be from 1 to 9.

   For example If we call `get_block(4)` in *figure 6* after solving Sudoku then it will return $[8, 5, 9, 4, 2, 6, 7, 1, 3]$ as it is the 4^{th} block of the Sudoku.

   Level: Easy

4. `get_row(sudoku:List[List[int]], x:  int) -> List[int]`:
   This function takes an integer argument i and then returns the i^{th} row. Row indexing have been shown above. **Note** that row indexing is from 1 to 9 and not 0-8 thus the input x will be from 1 to 9.

   For example If we call `get_row(4)` on solved Sudoku of figure 6 then it will return $[8, 5, 9, 7, 6, 1, 4, 2, 3]$ as it the 4^{th} row of the solved Sudoku

   Level: Basic

5. `get_column(sudoku:List[List[int]], x:  int) -> List[int]`:
   This function takes an integer argument i and then returns the i^{th} column. Column indexing have been shown above. **Note** that block indexing is from 1-9 and not 0-8 thus the input x will be from 1 to 9.

   For example if we call `get_column(4)` in figure 6 after solving Sudoku then it will return $[6, 1, 3, 7, 8, 9, 5, 4, 2]$ as it is the 4^{th} column of the Sudoku.

   Level: Basic

6. `find_first_unassigned_position(sudoku :  List[List[int]]) -> Tuple[int, int]`:
   This function returns the first empty position in the Sudoku. If there are more than 1 position which is empty then position with lesser row number should be returned. If two empty positions have same row number then the position with less column number is to be returned. If the Sudoku is completely filled then return `(-1,-1)`. The returned value is the position thus will be a tuple

of integers and will be equal to `(row,column)`

For example, when called on the Sudoku in fig. 5 will return `(1,3)`

Level: Easy

7. `valid_list(lst: List[int])->bool`:
   This function takes a lists as an input and returns true if the given list is valid. The list will be a single block , single row or single column only. A valid list is defined as a list in which all non empty elements doesn't have a repeating element.

   Note: A list may be valid even though it doesn't satisfy the Sudoku or be completely filled. You just have to check whether the provided list have repetitive **non-empty** elements or not. If the list is valid and Sudoku is violated then also you have to return true. Also note that 0 can be present any number of times as the cell containing 0 is empty. You are **NOT** allowed to use dictionaries or hash-maps in this question.

   For example: Valid list when called on `[1,2,3,0,0,1,7,5,0]` will return `False` due to repeating number 1. Whereas when called on `[5,3,0,0,7,0,0,0,0]` will return `True`

   Level: Medium

8. `valid_Sudoku(sudoku:List[List[int]])->bool`:
   This function returns `True` if the whole Sudoku is valid. You will have to check whether each row, column and block has unique non-empty elements. As in the `valid_list()` the Sudoku may have multiple 0s as they just indicate an empty position.

   Dependencies: `valid_list()`

   Level: Medium

9. `get_candidates(sudoku:List[List[int]], pos:Tuple[int, int]) -> List[int]`:
   This function takes the Sudoku and s position as arguments and returns a list of all the possible values that can be assigned at that position so that the Sudoku remains valid at that instant. What this means is that say the block which has this position has the number 5 then 5 is **not** a candidate. If the number 3 does not appear in the row, column or block of the position it is a valid candidate.

   Dependencies: `get_row(), get_column() and get_block()`

   Level: Medium

10. `make_move(sudoku:List[List[int]], pos:Tuple[int, int], num:int) -> List[List[int]]`:
    It takes the Sudoku, a position and a number as a parameter and returns the Sudoku after placing the number at the position.

    Level: Easy

11. `undo_move(sudoku:List[List[int]], pos:Tuple[int, int])`:
    This function fills 0 at position pos in the Sudoku and then returns the modified Sudoku. It will undo the move done and make the position empty again.

    Level: Easy

12. `sudoku_solver(sudoku:  List[List[int]]) -> Tuple[bool, List[List[int]]]`::
    This is the main Sudoku solver. This function solves the given incomplete Sudoku and returns true if a Sudoku can be solved i.e. after filling all the empty positions the Sudoku remains valid and also returns the solved Sudoku.

Algorithm 1 Solve Sudoku

```
Require: sudoku : List[List[int]]
  while Unassigned position exists do
    Get the candidates at that position
    for Each candidate do
      Make move
      Recursively solve for the Sudoku after making move
      if Sudoku not solved then
        Undo Move
      end if
    end for
  end while
```

Look at the above pseudo-code for the algorithm to solve a Sudoku. You have to keep on filling unassigned positions. After finding one you make a guess based on the candidates and the try to solve the Sudoku with this guess. If it is solved you're done but if not you undo the move and make another guess.

Dependencies: `get_candidates(), find_first_unassigned_position(), make_move(), undo_move()`

Level: Hard

You have been given a starter code which has all the functions with their descriptions written in the doc-string along with type-hinting as well for your convenience. The starter code takes in the input from `stdin` and tries to solve the Sudoku using the function `sudoku_solver`. It then prints whether it was able to solve the Sudoku or not. Note that this input/output has been already done, you only need to complete the functions that are given below.

You **cannot** have print statements inside the functions you may print while you are solving the assignment but they have to be removed. Even though we would be importing your functions and checking the returned values for the correctness, make sure you remove all such print statements from your functions before you submit your code, since this can lead to undefined behaviour of the auto-grader.

The In-lab components are -

1. `get_block_num()`
2. `get_block()`
3. `get_row()`

The marks breakdown is given in the table below -

Function	Marks
get_block_num	10
get_position_inside_block	10
get_block	5
get_row	5
get_column	5
find_first_unassigned_position	10
valid_list	10
valid_sudoku	10
get_candidates	15
make_move	5
undo_move	5
sudoku_solver	30
Total Marks	120