Project 3

Please read the **entire document** before writing any code.

1 Introduction

Please read the **entire document** before writing any code.

1.1 Objectives

For this project, you will write a program requiring use of loops and lists.

1.2 Collaboration

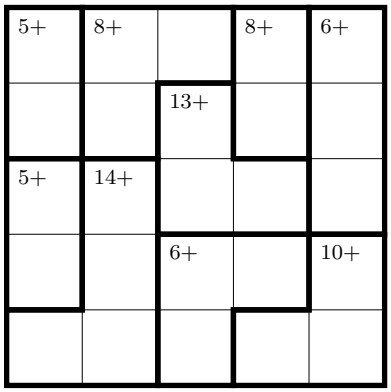
You **may not collaborate** in any way on your project. See the syllabus for more details.

2 Problem Description

For this program, you will be writing a solver for 5×5 Calcudoku puzzles. A Calcudoku puzzle is an $N \times N$ grid where the solution satisfies the following:

- each row can only have the numbers 1 through N with no duplicates
- each column can only have the numbers 1 through N with no duplicates
- the sum of the numbers in a "cage" (area with a bold border) should equal the number shown in the upper left portion of the cage

Here is a sample puzzle:



Here is the solution for the puzzle:

5+ 4	8+ 1	2	8+ 5	6+ 3
1	5	13+ 4	3	2
5+ 2	14+ 3	5	4	1
3	4	6+ 1	2	10+ 5
5	2	3	1	4

The description for each puzzle will be input in the following format:

```
number_of_cages
cage_sum number_of_cells (list of cells)
cage_sum number_of_cells (list of cells)
...
...
...
```

The first line contains the number of cages in the puzzle.

After the first line, each subsequent line describes a cage. The first value in a line is the sum of the values in the cage, the second value is the number of cells in the cage, and the third value onward contains a list of the cells contained on the cage. In the puzzle, the cells are numbered starting with 0 for the upper left cell and increase from left-to-right, and then from top-to-bottom. (In the above puzzle, the squares would be numbered from 0 to 24.)

For the sample puzzle shown above, the input would look like:

```
9
5 2 0 5
8 3 1 2 6
8 2 3 8
6 3 4 9 14
13 3 7 12 13
5 2 10 15
14 4 11 16 20 21
6 3 17 18 22
10 3 19 23 24
```

3 Solving a Puzzle

For this program, you will be solving the puzzle using 'brute-force' where the program will test all possible solutions until it finds the correct one. Your algorithm should perform the following:

- Place a 1 in the current cell (starting with the upper left cell).
- 'Check' if the number is valid
  - If the number is valid, continue to the next cell to the right (advancing to the next row when necessary)
  - If the number is not valid, increment the number and 'check' for validity again

    * If the number is the maximum possible and is still invalid, then 'backtrack' by setting the current cell to 0 and moving back to the previous cell

In order for this algorithm to work, you will need to write a function to test if a puzzle is in a valid state. **You should initialize the values in your puzzle to 0.** As you populate the puzzle, a puzzle is invalid if:

- Duplicates exist in any row or column
- The sum of values in a fully populated cage does not equal the required sum
- The sum of values in a partially populated (i.e., not full) cage exceeds or equals the required sum

In your program, you will be counting the number of times you check if a number is valid, and the number of backtracks.

3.1 Sample Run 1

This is the example from above.

All user input is **bold** in the sample runs for clarity only. Your program is not expected to behave this way.

```
Number of cages: 9
Cage number 0: 5 2 0 5
Cage number 1: 8 3 1 2 6
Cage number 2: 8 2 3 8
Cage number 3: 6 3 4 9 14
Cage number 4: 13 3 7 12 13
Cage number 5: 5 2 10 15
Cage number 6: 14 4 11 16 20 21
Cage number 7: 6 3 17 18 22
Cage number 8: 10 3 19 23 24

--Solution--

4 1 2 5 3
1 5 4 3 2
2 3 5 4 1
3 4 1 2 5
5 2 3 1 4

checks: 2265 backtracks: 438
```

4 Specification: Functions

You are to implement the following functions in `solver_funcs.py`.

```
check_valid(puzzle, cages)
check_cages_valid(puzzle, cages)
check_columns_valid(puzzle)
check_rows_valid(puzzle)
```

Please note:

- Both the puzzle and the cages must be stored as a list of lists.
  - Puzzle is a list of lists; e.g.

```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

  - Cages is a list of lists; e.g.

```
[[5, 2, 0, 5],
 [8, 3, 1, 2, 6],
 [8, 2, 3, 8],
 [6, 3, 4, 9, 14],
 [13, 3, 7, 12, 13],
 [5, 2, 10, 15],
 [14, 4, 11, 16, 20, 21],
 [6, 3, 17, 18, 22],
 [10, 3, 19, 23, 24]]
```

- Feel free to break up your functions into smaller functions (part of your grade may be based on how well you decompose the problem). My `solver_funcs.py` file has seven functions in it. However, you *must* have these four functions and they *must* behave as specified below.

4.1 `check_valid(puzzle, cages)`

The parameters are:

- a list of lists representing the puzzle; and
- a list of lists representing the cages

This function returns `False` if the current puzzle board is invalid—where the invalid criteria are given above—and `True` otherwise. (This will most likely be accomplished by calling the following three functions.)

4.2 `check_cages_valid(puzzle, cages)`

The parameters are:

- a list of lists representing the puzzle; and
- a list of lists representing the cages

This function returns `False` if any of the cages are invalid and `True` otherwise.

4.3 `check_columns_valid(puzzle)`

The parameter is a list of lists representing the puzzle. This function returns `False` if duplicates exists in any column and `True` otherwise.

4.4 `check_rows_valid(puzzle)`

The parameter is a list of lists representing the puzzle. This function returns `False` if duplicates exists in any row and `True` otherwise.

4.5 Function Testing

Test these functions (and any others you choose to write) using the unit testing techniques we have been using all quarter. Your testing of these functions must achieve 100% code coverage (if I haven't already explained what this is, please ask (or wait)). Each test should be written in its own testing function. Be sure to give your tests descriptive names indicating which function is being tested. I provide you with a starting file `funcs_tests.py`. Add additional tests to this file.

5 Specification: Calcudoku Solver

Begin this part with your *fully tested* `solver_funcs.py` file.

Now you must write a `solver.py` that solves Calcudoku puzzles. In `solver.py` you are to implement the following function as well as write your `main`.

5.1 `get_cages()`

This function has no parameters and returns a list of lists representing the cages. The function will prompt the user for a number of cages and will then prompt the user for that many cages. You may assume that all input will be valid. To read in multiple values in one line, you can use `input().split()`. This will return a list of all the values in the input string. Keep in mind that the values will still be strings and you will probably want to convert them into integers.

The function `get_cages` does not require any unit testing, ***but*** it will need to function identically to what is expected (see Section 5.3 for more details).

5.2 `main()`

In the `main` function of your `solver.py`, you must code the algorithm described in Section 3.

5.3 Diff Testing

There are two sample inputs provided in the starting repository. Diff your output against the expected output.

You can (and should) make your own puzzles as tests. You can test them against the provided executable.

6 Grading

Your program grade will be based on:

- thoroughness of your `funcs_tests.py`,
- adherence to the specification,
- the number of unit test cases passed,
- the number of `diff` test cases passed, and
- program style.

Your program will be tested with test cases that you have not seen. Be sure to test your program thoroughly!!! To pass a test case, your output must match mine EXACTLY. Use `diff` to make sure your output matches mine in the test cases given and in test cases that you make up. If there are any differences whatsoever, you will not pass the test case.

7 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 1, as needed, to remember how to push your local code.