Calcudoku

CPE 101: Fundamentals of Computer Science
Winter 2019 - Cal Poly, San Luis Obispo

Purpose

To further your understanding of iteration and using multidimensional lists, as well as implementing an exhaustive search algorithm.

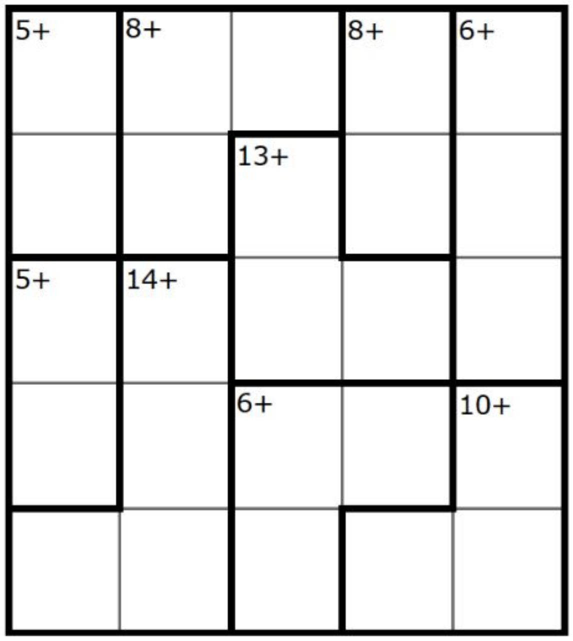
Description

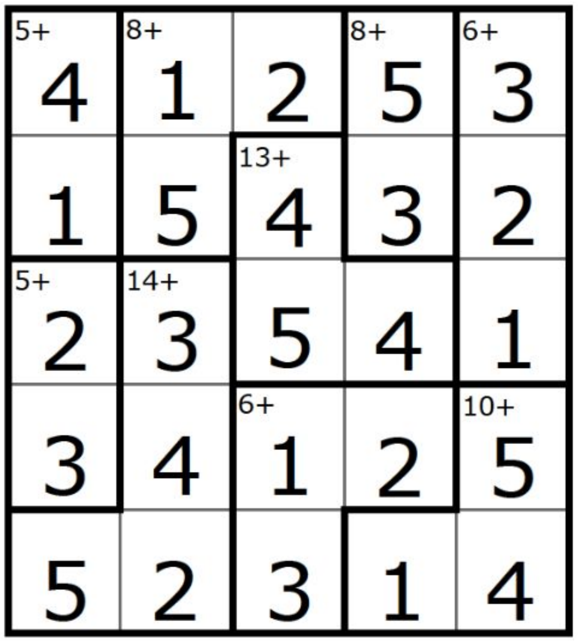
For this program, you will be writing a solver for 5x5 Calcudoku puzzles. A Calcudoku puzzle is an NxN grid where the solution satisfies the following:

- 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 (areas with a bold border) should equal the number shown in the upper left portion of the cage

Puzzle input and output files will be made available on polylearn on next Saturday.

Here is a sample puzzle (left) and its solution (right):





Input

The beginning of a sample input file is shown below.

```
9
5 2 0 5
8 3 1 2 6
...
```

The first line contains the number of cages in the puzzle. After the first line, each subsequent line describes a cage. The first number of a line is the sum of the cage, the second is the number of cells in the cage, and all numbers afterward refer to the positions of the cells that make up the cage. In the puzzle, the cell positions are numbered starting with `0` for the upper left cell and increase from left to right.

Output

Your program should display a solution to the puzzle output in the following format:

```
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
```

Implementation

Solving a Puzzle

Your program will solve puzzles using an exhaustive search (or "brute force") approach, in which it tries (potentially) all possible solutions until it finds the correct one. Your algorithm should perform the following:

1. Initialize all cells to `0`
2. Increment the value in the current cell by `1` (starting from the top-left cell)
    a. If the incremented value is greater than the maximum possible value, set the current cell to `0` and move back to the previous cell
    b. Otherwise, check if the number is valid. If so, continue to the next cell to the right (advancing to the next row when necessary)
3. Repeat Step 2 until the puzzle is fully populated and valid

In order for this algorithm to work, you will need to write functions to test if a puzzle is in a valid state. As you populate the puzzle with numbers, it becomes 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 cage equals or exceeds the required sum

Minimum Required Program Structure

In the function headers below, the parameter `grid` refers to a 2D list of integers representing the cells of the puzzle and `cages` refers to a 2D list of integers representing the values read from input.

`main()`

The `main` function (with optional helper functions) should perform the following steps:

- Store the data from the puzzle input file in a 2D list of integers
- Create a 5x5 grid using a 2D list and fill it with zeros
- Only one `while` loop is recommended to validly populate every cell in the grid

`validate_all(grid, cages)`

Return True if all 3 validation functions below return `True` and `False` otherwise.

`validate_rows(grid)`

Return `True` if all rows contain no duplicate positive numbers and `False` otherwise.

`validate_cols(grid)`

Return `True` if all columns contain no duplicate positive numbers and `False` otherwise. It is recommended that you transpose the grid (convert its rows to columns and columns to rows) and pass this transposition to `validate_rows` to reuse your existing code.

`validate_cages(grid, cages)`

Return `True` if the sum of values in a fully populated cage equals the required sum or the sum of values in a partially populated cage is less than the required sum and `False` otherwise.

Testing

You are required to write at least 3 tests for each function (except `main`) in `tests.py`. Since we are emphasizing test-driven development, you should write tests for each function first. In doing so, you will have a better understanding as to what the functions take as input and produce as output, which makes writing the function definitions easier.

Each puzzle can be found in a separate file:

```
      test1.in, test2.in, test3.in, ...
```

Your program should be run using:

```
      python3 calcudoku.py < test#.in
```

You should compare your output with the corresponding output files using `diff` (without the use of any flags):

```
      test1.out, test2.out, test3.out, ...
```

Submission

Zip `calcudoku.py` and `tests.py` into one zip file whose name contains your cal poly id, and upload it to polylearn.