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

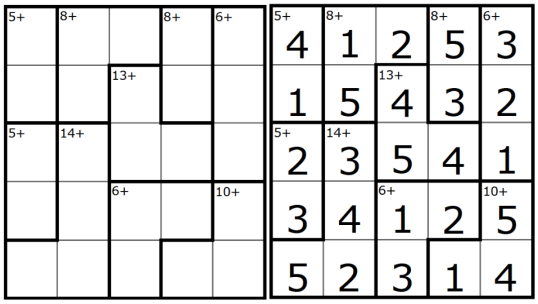
For this program, you will be writing a solver for 5x5 Calcudoku puzzles. A Calcudoku puzzle is an NxN grid where the solution satisfies the following:

- 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 can be downloaded from:

http://users.csc.calpoly.edu/~dkauffma/101/calcudoku.zip

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



Input

A sample input file is shown below.

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

```
6 18 23 24
```

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 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)
    - If the incremented value is greater than the maximum possible value, set the current cell to 0 and move back to the previous cell
    - 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

```
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

```
transpose(grid: List[List[int]]) -> List[List[int]]
```

Return the transposition of the given grid. Assume that the grid has a square number of elements but make no assumptions about its actual size.

```
validate_rows(grid: List[List[int]]) -> bool
```

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

```
validate_cages(grid: List[List[int]], cages: List[List[int]]) -> bool
```

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

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`:

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

Submission

On a CSL server with `calcudoku.py` and `cdtests.py` in your current directory:

```
/home/dkauffma/casey.exe 101 calcudoku
```