CSCI 3901 Winter 2021

Assignment 4

Due date: 23:59 AST Friday, March 12th, 2021 in Brightspace

Problem 1

Goal

Work with exploring state space and backtracking.

Background

Some games and puzzles like sudoku can be solved with a computer by systematically exploring all possible solutions, called the *state space,* until a correct solution is found. They do this by *backtracking* to a previous state whenever an incorrect solution is found, and then continuing from that point by exploring different possible solutions. In this assignment you will write a program to solve a mathematical puzzle which is closely related to sudoku by backtracking and state space exploration.

You will be solving a puzzle called *mathdoku*[1] (see http://www.mathdoku.com). Figure 1 shows a sample puzzle. In a mathdoku puzzle, you are given a square $n \times n$ grid. Within the grid, each cell is identified as part of some **grouping**. Each grouping is a **connected** set of 1 or more cells and each grouping is given

1. a mathematical operator like + or /, and
2. an integer which is the result of applying the operator to the cells.

The task is to put the integers 1 to n into the cells of the grid so that:

- No integer appears twice in any row
- No integer appears twice in any column
- Applying the operator to the values in a grouping gives the integer assigned to that grouping.

Figure 2 shows a solution to the puzzle in Figure 1.

The possible operators for a cell are + for addition, − for subtraction, ∗ for multiplication, / for division, and = to specify that a grouping (of one cell) has a given value. For the − and / operations, the groupings must have **exactly 2 cells**, and the **larger value always comes first** in the computation. For example, if the operation is / and the values are 2 and 6, the result of that grouping would be

$$6/2 = 3.$$

The results of operations on groupings must always be integers, so 4/3 is **not** a valid assignment.

[1]These puzzles appear under the trade name of KenKen (https://www.kenkenpuzzle.com).

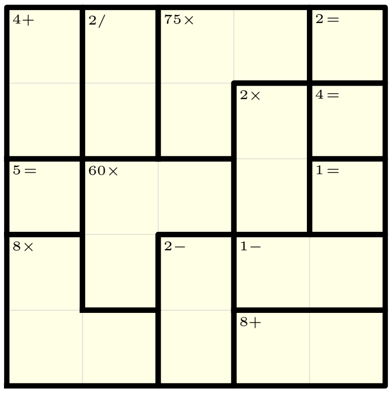

Figure 1: A mathdoku puzzle.

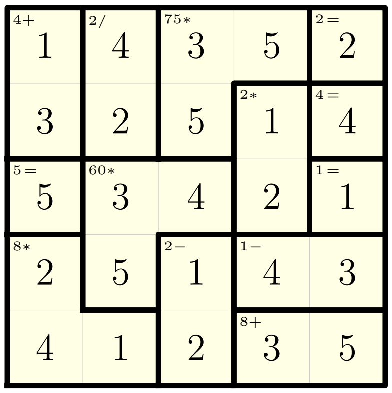

Figure 2: The solution to the puzzle in Figure 1.

Problem

Your task is to create a class called `Mathdoku` that accepts a puzzle and ultimately solves it. The class has at least 5 public methods with the following signatures:

`public boolean loadPuzzle(BufferedReader stream)` Read a puzzle in from a given stream of data. Further description on the puzzle input structure appears below. Return `true` if the puzzle is read successfully. Return `false` if some other error happened.

`public boolean validate()` Determine whether or not the loaded puzzle is valid. Return `true` if the puzzle is valid and can be solved, and return `false` otherwise. Further information on this method appears below.

`public boolean solve()` Do whatever you need to do to find a solution to the puzzle. The method should store the solution in the object so that it can be retrieved later. Return `true` if you solved the puzzle and `false` if you could not solve the puzzle. This method should also keep track of the number of **guesses** you make in the process of solving the puzzle, and then store this in the object to be retrieved later by the `choices` method below.

`public String print()` Return the current puzzle state as a `String` object.

`public int choices()` Return the number of guesses that your program had to make and later undo/backtrack while solving the puzzle.

You get to choose how you will represent the puzzle in your program and how you will proceed to solve the puzzle. The only constraint on your solution is that it should be somewhat **efficient**. In this case efficiency means that you explore **as few solutions as possible** before arriving at a correct one or discovering that there is no solution to the puzzle.

Inputs

The `loadPuzzle` method will accept a description of the puzzle as an input stream. The input stream will be formatted in three sections as follows:

- Section 1: Consists of an integer n which gives the size of the puzzle (an $n \times n$ grid)
- Section 2: Consists of the puzzle square itself. For an $n \times n$ puzzle square, the input will have n lines of input, each being a string of n characters (excluding the end of the line character). For one line, the n characters are the n cells in the line; each cell is a letter that represents the cell grouping to which the cell belongs.
- Section 3: Consists of the constraints for each cell grouping in the puzzle. There will be one line for each cell grouping. That line will have 3 values: the letter representing the grouping, the operation outcome for the grouping, and the operator for the grouping (one of +, -, *, /, or =). The values are separated from one another by **at least** one space.

Example: The puzzle in Figure 1 would be represented as the following input. The constraints are given alphabetically by the grouping name here, but that's not a guarantee in all input.

```
5
abccd
abcef
ghhei
jhkll
jjkmm
a 4 +
b 2 /
c 75 *
d 2 =
e 2 *
f 4 =
g 5 =
h 60 *
i 1 =
j 8 *
k 2 -
l 1 -
m 8 +
```

Input validation

In many programs, such as compilers, there are multiple stages of input validation to check more complex properties of the input. The `validate` method performs additional input validation to check whether the input loaded by the `loadPuzzle` method specifies a valid mathdoku puzzle.

Some conditions you will need to check in your `validate` method (if you already check some of these in your `loadPuzzle`, that is fine) are:

- The cells form an $n \times n$ grid
- Every grouping is a *connected* set of cells — that is, there are no "gaps" between cells in a grouping.
- Every grouping has a value and an operator
- Every grouping with the $=$ operator has exactly one cell
- Every grouping with the $-$ or $/$ operator has exactly two cells
- Every grouping with the $+$ or $*$ operator must have at least two cells

Outputs

The `print` method produces a `String` that represents the current state of the puzzle. The output provides each row of the current puzzle state; listed from top-to-bottom and rows separated by a carriage return (`\n`) character. No space is printed between the columns of the rows. If the cell has a value from 1 to n assigned to it then print that value for the cell. Otherwise, print the grouping letter for the cell.

The following is the returned string for the puzzle in Figure 2, noting that `\n` should be seen as just one character (newline character) in the text below:

```
14352\n32514\n53621\n25143\n41235\n
```

Assumptions

None.

Constraints

- The character for each cell grouping is **case sensitive**. So, a cell entered as `a` and another as `A` are two different groupings.
- You **may** use any data structures from the Java Collection Framework.
- You **may not** use an existing library that already solves this problem.
- If in doubt for testing, I will be running your program on `timberlea.cs.dal.ca`. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- Develop a strategy on how you will solve the puzzle before you start coding your data structure(s) for the puzzle
- Work incrementally. First write the code to read in a puzzle. Test that. Next, write the code to print the current state of the puzzle. Test that. Then, write the code to validate the puzzle. Test that. Last, write your code to solve the puzzle.
- It might be helpful to start by **brute-forcing** the puzzle — specifically, try every possible number in each cell and check to see if it's a solution. You probably won't be able to solve large puzzles this way, so start with something small like a 4×4 grid. Check http://www.mathdoku.com/ for sample puzzles.
- Once you have a working solution, think about ways in which you can make your solution more efficient. For example, if a grouping contains only one cell, do you need to try all possible values in that cell or can you simply **infer** the correct value for that cell? Note that there is no one solution to making your code efficient.
- **Do not change any of the given class names, method names or method signatures.**
- **Do not mark any of variables or methods in your class as static.**

What to submit

- Your `Mathdoku.java` file and any other supporting java files.
  - Do **not** include `package` statements in your java file(s).
- A separate `.pdf` file with
  - A list of your test cases for the problem.
  - An explanation of your solution — specifically, the strategy/algorithm you use to solve the puzzles, as well as what steps you have taken to provide some degree of efficiency.

Marking scheme (out of 25)

- Documentation, program organization, clarity, modularity, style – 4 marks
- List of test cases for the problem - 3 marks
- Explanation of your solution - 3 marks
- Ability to solve puzzles - 12 marks
- Efficiency of your solution - 3 marks