Assignment 4

Problem 1: Test cases for Mathdoku class

1. **Input Validation Test Cases:**
   a. **Public boolean loadPuzzle (BufferedReader stream)**
      i. Any other data type passed other than BufferedReader stream
         1. Compile time error
      ii. Null passed
         1. Returns false
      iii. Empty stream passed
         1. Returns false
      iv. Puzzle size not provided in the stream
         1. Returns false
      v. Puzzle size greater than the grouping
         1. Returns false
      vi. Extra leading & trailing spaces in the stream
         1. Returns true
      vii. If empty lines found anywhere in the stream
         1. Returns true
      viii. Grouping less then the size of the puzzle
         1. Returns false
      ix. Grouping more than the size of the puzzle
         1. Returns false
      x. Grouping is not in alphabetical order
         1. Returns true
      xi. More than one space between the constraints passed in the stream
         1. Returns true
      xii. Order of the constraint changed
      (anything other than **group, outcome, operator**)
         1. Returns false
      xiii. Constraint missing anything from **group, outcome, operator**
         1. Returns false
      xiv. Constraint missing at least one space between **group, outcome, operator**
         1. Returns false
      xv. Constraints are not in alphabetical order
         1. Returns true
      xvi. Puzzle size in the stream is <= 1
         1. Returns false
      xvii. Puzzle is a grid of size m x n
         1. Returns false

   b. **Public boolean validate ()**
      i. Puzzle is null
         1. Returns false
      ii. Puzzle groups are null
         1. Returns false
      iii. Every grouping is a connected set of cells
         1. Returns true
      iv. Some groupings are not connected
         1. Returns false

v. Every grouping with the ‘=’ operator has exactly one cell
   1. Returns true

vi. Every grouping with ‘-‘ or ‘/’ operator has exactly two cells
   1. Returns true

vii. Every grouping with ‘+’ or ‘*’ operator has at least two cells
   1. Returns true

viii. All groups are present in the constraints passed
   1. Returns true

ix. Spaces between the groupings passed
   1. Returns false

c. **Public boolean solve ()**

i. Puzzle solved
   1. Returns true

ii. Puzzle not solved
   1. Returns false

iii. Puzzle is null
   1. Returns false

d. **Public String print ()**

i. Puzzle is null
   1. Returns null string

e. **Public int choices ()**

i. Choices is 0
   1. Returns 0

2. **Boundary Tests Cases:**

a. **Public boolean loadPuzzle (BufferedReader stream)**

i. Very long stream is passed
   1. Returns true

ii. Very short stream is passed
   1. Returns true

iii. Puzzle size 2
   1. Returns true

iv. Puzzle size 1
   1. Returns false

v. Huge puzzle is passed as a stream
   1. Won’t be able to solve

b. **Public boolean validate ()**

i. Any group missing in the constraints passed
   1. Returns false

ii. Extra group present in the constraints passed
   1. Returns false

c. **Public boolean solve ()**

i. Puzzle is empty
   1. Returns false

d. **Public String print ()**
(covered in other sections)

e. Public int choices ()
   i. Choices < 0
      1. Returns 0

3. Control Flow Test Cases:

a. Public boolean loadPuzzle (BufferedReader stream)
   Load a solvable puzzle
      1. returns true
   ii. Load an unsolvable puzzle
      1. Returns false

b. Public boolean validate ()
   i. Validate a solvable puzzle
      1. Returns true
   ii. Validate an unsolvable puzzle
      1. Returns false

c. Public boolean solve ()
   i. Solve a solvable puzzle
      1. Returns true
   ii. Solve an unsolvable puzzle
      1. Returns false
   iii. Solve when multiple solutions exist
      1. Returns true

d. Public String print ()
   i. Print a partially solved puzzle
      1. Returns the partially solved puzzle string
   ii. Print an unsolved puzzle
      1. Returns the unsolved puzzle string
   iii. Print a solved puzzle
      1. Returns a solved puzzle string

e. Public int choices ()
   i. Puzzle not solved
      1. Returns 0
   ii. Puzzle solved
      1. Returns value < 0

4. Data Flow Test Cases:

<u>Note:</u>

I am calling validate () method in the solve () method, so if the puzzle is valid then only, I will start solving the puzzle

a. Calling loadPuzzle (BufferedReader stream) twice in a row
b. Calling solve () twice in a row
c. Calling print () twice in a row
d. Calling choices () twice in a row
e. Calling solve () before calling loadPuzzle (BufferedReader stream)
f. Calling print () before calling loadPuzzle (BufferedReader stream)
g. Calling choices () before calling loadPuzzle (BufferedReader stream)
h. Calling print () before calling solve ()

    - i. **Covers test case – calling print () before calling validate ()**
- **i.** Calling choices () before calling solve ()
    - i. **Covers test case – calling choices () before calling validate ()**
- **j.** Calling choices () before calling print ()

Explanation of the solution:

- My approach to solving the Mathdoku puzzle:
    - I am performing all the required validations needed to solve the puzzle
        - If any validation fails then I am returning false
    - Then I am checking whether the puzzle is solved
        - If it is solved then returning true
        - Else moving on with solving the puzzle
    - I am starting with the "=" operator
        - So first I am iterating through the puzzleGroups then,
            - Getting the equal operator's location
            - Getting its outcome
            - Then before moving further I am checking for conflicts in that row & column
                - If conflict found then resetting the puzzle & retuning false
            - If not then I am checking if the outcome is greater than the puzzle size
                - Then resetting the puzzle & returning false
            - Else setting the outcome as cell value for that group
            - Incrementing the choices
            - Adding that group in "=" operator's set

      (**Once all "=" operators' values are set**)
        - Removing the set from the puzzleGroups key set
        - So, now we left with other operator groups
    - Now, I am storing grouping points in a 2D array list
        - Adding all groups locations to the array list
    - Storing all the possible values that could be stored in a cell in an array
    - Going through remaining groupings
        - Storing all the possible pairs in a 2D array list
        - Calling individual methods for remaining 4 operators (+, -, *, /)
        - For add & multiply I am using recursive approach to find the possible pairs
            - Here I am checking for the constraints
            - Then I am trying out all the values passed in the list of possible values
            - If a value fits then adding that value to the solution & removing it from the list
            - The only difference between add & multiply logic is the operation of adding & multiplying
        - For subtract & division, recursion is not used
            - Here I am iterating through the list
            - Then subtracting or dividing the values & comparing with the outcome
            - If satisfies then adding it & its reverse form to the solutions list
            - The logic is same for subtract & divide, only in divide I am performing both divide & modulo
        - Iteratively keeping on performing the above steps to get all the possible pairs for the puzzle
        - After that, I am iterating through the allPossiblePairs array list, checking whether any conflict exists
            - If exists then backtrack

        - If not then setting the value in the puzzle for that group
    - Incrementing the choices

Note:
I am resetting the puzzle & its cell values, so if the puzzle is not solved then it prints the group names instead of values

Steps to make it efficient:

- I started with “=” operator as I just have to set its value
- This fills up some of the cells in the puzzle
- Then I removing those from the main groups, which results in less iterations
- Then for add & multiply I am using recursion to find the possible pairs
- Whereas for subtract & divide I am just iterating the puzzle once & storing all possible pairs in normal & reverse order
- Tried to avoid recursion for any other part to improve the efficiency