Lecture Assignment 4

Test Cases:

- **Class Mathdoku()**
  - **Input Validations:**
    - **public Boolean loadPuzzle(BufferedReader stream):**

      i) Stream is null.
      ii) Stream is empty.
      iii) Stream reads value of type other than string.
      iv) Stream reads any special character.
      v) Stream reads space.
      vi) Stream reads line which is less in length compare to size of matrix.

    - **public boolean validate():**

      i) Cells are empty or null.
      ii) Matrix is not according to size n.
      iii) Gaps between cells of group.
      iv) No operator is present in a group.
      v) No result value is present in a group.
      vi) Group with = operator has more than one cell.
      vii) Group with – or / operator has one or more than two cells.
      viii) Group with * or + operator has only one cell.

    - **public boolean solve():**

      i) Cell is empty or group is empty.
      ii) Adds pair of value which conflicts with any value in row or column.
      iii) Adds value to the cell which is more than size of matrix.
      iv) Adds zero or negative value as an input to cell.

    - **public string print():**

      i) Cells are empty or null.
      ii) Cells are not complete.

- **Boundary Cases:**

  - **public Boolean loadPuzzle(BufferedReader stream):**

    i) Size of cells exceeds (n*n).
    ii) Size of cells are less than (n*n).

  - **public boolean solve():**

    i) Pair of x, y should not exceed size n-1.
    ii) Value of pairs should be between 1 and size.

  - **public int choices():**

    i) Choices should be non-negative.

- **Control Flow:**

1. Call loadPuzzle(BufferedReader stream).
   - Reads text file using file reader and stores in buffer.
   - Reads file line by line using readLine().
   - Loads puzzle into matrix of mentioned size n.
2. Call validate().
   - Checks whether the loaded puzzle is valid or invalid.
   - Checks whether the matrix formed is according to mentioned size.
   - Checks whether the groups are connected set of cells without having gap in between cells.
   - Checks that the = operator has only one cell.
   - Checks that the group of – or / operator has only two cells.
   - Checks that the group of + or * operator has atleast two cell.
3. Call solve().
   - Solves the puzzle based on the given operator and its resulting value in a group of cells.
   - Tries different possible pair of values in cells.
   - Backtracks again if there is any conflict or some group is not able to set any pair of values.
   - Calls int choices() if any value need to be reset to zero.
4. Call print().
   - This method will print the puzzle outcome.

- **Data Flow:**

1. Records data of solve() in a new object.
2. Records data of solve() in an already existing object.
3. Call validate() before calling loadPuzzle(BufferedReader stream).
4. Call solve() before calling validate().
5. Call solve() before calling loadPuzzle(BufferedReader stream).
6. Call print() before calling solve().
7. Call choices() before calling solve().

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.

- I have used recursive approach to code this puzzle. I have taken references of other game codes like sudoku, kakuro, etc on internet to understand how to apply the logic of games.

- Strategy of code is that it firstly forms the matrix by reading size of matrix from text file.

- Then it forms group of cells inside the matrix and loads that groups result value and its operator. So loadPuzzle method will load complete puzzle this way by forming groups of cells.
- Then validate method just validates different conditions that puzzle loaded is valid or not.

- Solve method solves the puzzle by trying different possible pairs in cells based on operator provided. I have used recursion specifically for multiplication and addition operations because this both operators can have multiple operands in groups. Whereas subtraction and division operators have only two operands so there is no need of recursion there as it is restricted to a limit.

- I used backtracking algorithm to backtrack the possible pair values in case when the pairs chosen for a specific group is not valid. So it resets values to zero whenever the need arises to backtrack and choice method returns the number of guesses that how many time we backtrack.

- Print method then prints the outcome of puzzle.

- To provide some degree of efficiency I implemented the code in recursive manner and also made the efficient use of variables instead of defining more variables.