# Mathdoku Program Overview

Mathdoku: It is a public class used to solve a mathematics-based puzzle knows as mathdoku. This puzzle has N*N puzzle where value of a cell is an integer in range of [1-N]. Same value cannot be repeated in the row and column of the cell. Each cell is part of a grouping which is formed to perform mathematical operation (+, -, *, /, =) and equivalent value of the operation is given for each group. The complexity of the puzzle increases with the size of the grid.

**Strategy and Algorithm used:**

- The approach used is mainly based on backtracking.
- Recursion is used to implement the backtracking for the ease of not remembering past values at each step as stack trace takes care of it.
- Steps followed:
    1. Load the puzzle.
    2. Check if it forms n*n valid grid.
    3. Check if it has valid grouping operators and results.
    4. Once we have confirmed that puzzle is valid. Pick the cell which is empty traversing from row to row in entire grid.
    5. When empty cell is spotted try it for all possible values [1-N]
    6. For each value check:
        - Row validity: if the value is not repeated in cell row
        - Column validity: if the value is not repeated in cell column
        - Group validity: If the value satisfies its group constraints:
            - → "+": check if the value after adding to other cell location doesn't exceed the result
            - → "-": return true if both cells are empty or checks if value - other cell or vice versa is equal to the result
            - → "*": check if the value after multiplying to other cell location doesn't exceed the result
            - → "/": return true if both cells are empty or checks if value / other cell or vice versa is equal to the result
            - → "=": returns true if value = result
    7. Whichever value passes all validity tests set it for the cell and if no value fits backtrack using recursion to find new possible solution
    8. Repeat steps 4, 5, 6 and 7 until there is no empty cell and puzzle is solved.
    9. Keep count of number of failed choices before puzzle was solved.

**2) Global Variables in Mathdoku class:**

a) *puzzleSize* - Int to store size of the puzzle used to build puzzle board
b) *ChoiceCounter* - Int to store counter for number of choices made before reaching the final solution.
c) *IsPuzzleReady* - Boolean to check is all constraints to solve puzzle are fulfilled
d) *IsGridComplete* - Boolean to check if n*n values are available to form the grid
e) *Operators* - String list of operators allowed for the puzzle
f) *GrpVarSet* - Set of strings used to represent the groupings
g) *PuzzleInput* - List of strings used to read and store puzzle
h) *GroupCells* - Map of grouping cells, key: group variable, value: location of cells of that grouping
i) *GroupOperators*- Map of grouping operators, key: group variable, value: operator for that group
j) *GroupEquals* - Map of grouping results, key: group variable, value: result of operation for that group
k) *GroupArr*- 2D String array to store grouping variable for each cell
l) *PuzzleBoard*- 2D Int array which is used to finally solve the puzzle

**3) Public Methods in VertexCluster class:**

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

i) Input: BufferedReader stream
ii) Output: boolean
iii) Functionality: This method is used to read the puzzle and store the data in different objects which can be used throughout the class by other methods.
iv) Implementation:
→ reset all the global variables every time load puzzle is called
→ if stream is null return false
→ handle IOException while reading the input in stream
→ read the input line by line and store it in puzzleInput
→ return false if exception is caught
→ return false if file was empty or else call initializeDS
→ If initializeDS returns true, grid can be formed so it returns true

***b) boolean readyToSolve()***

i) Output: boolean isPuzzleReady
ii) Functionality: This method is used to check if all the constraints needed to solve the puzzle are met. If it returns true, solve can be called to get the final solution.
iii) Implementation:
→ get the set of keys from groupOperators, groupEquals and groupCells

→ all the key sets from above set should contain all grouping variables
→ check if all the operators are within the defined set i.e. +, -, /, *, =
→ only one cell should be allocated to the "=" operator
→ only two cells should be allocated to the "-" and "/" operator
→ if all the above conditions are true set isPuzzleReady to true

***c) boolean solve ()***

***i)*** Output: boolean
ii) Functionality: This method is used to solve the puzzle. Returns true if puzzle is solved and returns false otherwise.
iii) Implementation:
→ check if puzzle is ready to be solved
→ get the location of next empty cell
→ if there is no empty cell that means puzzle is solve return true in that case or else iterate over all possible values [1-N] for the empty cell.
→ call value validation for each cell value if it returns true use recursion to solve the next cells and backtrack if it is false and set cell value to 0
→ in case the value for variable fails call choicesIncrement and return false

***d) String print()***

i) Output: String printPuzzle
ii) Functionality: This method is used to print the current state of the puzzle. It returns grouping variable for the cell that is not solved yet.
iii) Implementation:
→ initialize printPuzzle to empty string
→ check if grid is complete and iterate over the puzzle board
→ if cell value is zero add grouping variable for that
→ add new line at the end of each row and return printPuzzle

***e) int choices()***

i) Output: int choiceCounter
ii) Functionality: This method is returning the number of wrong choices made before returning the output.
iii) Implementation:
→ It's just the wrapper method, computation of choices is done in choicesIncrement

**4) Private Methods in Mathdoku class:**

***a) boolean initializeDS()***

i) Output: boolean

ii) Functionality: This method is called by load puzzle if file has some data. This method checks if data loaded is valid to form puzzle grouping and initializes all the global variables

iii) Implementation:

- → get the number of characters in first row of the puzzleInput
- → if number of rows in puzzleInput is greater than n and then iterate over next n rows and check if they also have n characters and set isGridComplete to true
- → if n rows have n elements then update the groupArr with grouping variables and initialize puzzleBoard to 0.
- → use nested for loops to update values of groupArr,groupCells puzzleBoard and grpVarSet
- → iterate over the rest of the rows to create map of groupOperators and groupEquals
- → if number of rows in puzzleInput are less than set isPuzzleReady to false
- → Returns isGridComplete at the end

***b) boolean rowValidation(int value, int rowNo, int colNo)***

i) input: int value, int rowNo, int colNo

ii) Output: boolean

iii) Functionality: This method gets the prospective value and location for the empty cell and returns true if the value is not already in the row of the cell, otherwise returns false

iv) Implementation:

- → It's just the wrapper method, computation of choices is done in choicesIncrement

***c) boolean colValidation (int value, int rowNo, int colNo)***

i) input: int value, int rowNo, int colNo

ii) Output: boolean

iii) Functionality: This method gets the prospective value and location for the empty cell and returns true if the value is not already in the column of the cell, otherwise returns false

iv) Implementation:

- → iterate over the column and check if value already exists

***d) boolean groupValidation(int value, int rowNo, int colNo)***

i) input: int value, int rowNo, int colNo

ii) Output: boolean

iii) Functionality: This method gets the prospective value and location for the empty cell and applies rules for the grouping of the cell if it satisfies the rule method returns true or returns false otherwise

iv) Implementation:

→ get group variable, operator and result for the given cell
→ if the operator is +, check if the value after adding to other cell location doesn't exceed the result
→ if the operator is -, returns true if both cells are empty or checks if value - other cell or vice versa is equal to the result
→ if the operator is *, check if the value after multiplying to other cell location doesn't exceed the result
→ if the operator is /, returns true if both cells are empty or checks if value / other cell or vice versa is equal to the result
→ if the operator is =, returns true if value = result
→ returns false if either one of the above conditions is not met

***e) boolean valueValidation(int value, int rowNo, int colNo)***

i) input: int value, int rowNo, int colNo
ii) Output: boolean
iii) Functionality: This method is used to validate the given value and its cell location
iv) Implementation:
→ it calls rowValidation, colValidation, groupValidation and
→ returns true if each function validates the new value

***f) int[] nextEmptyCell()***

i) Output: int[] emptyLoc
ii) Functionality: This method is used to find the next empty cell in the puzzle board.
iii) Implementation:
→ it iterates over all values in puzzle board and returns the location of cell which has value = 0
→ returns null if there is no empty cell or puzzle is solved

***g) void choicesIncrement()***

i) Functionality: This method is used to update the choiceCounter for the number of choices being made before reaching the final solution for the puzzle.
ii) Implementation:
→ find the empty cells in the puzzle when the choice was dropped
→ if puzzle is less than 4*4 and half of the cells were empty we increment choiceCounter
→ if puzzle is more than 4*4 and n cells were empty we increment choiceCounter

**Assumptions:**

1. The character for each cell grouping is case sensitive. So, a cell entered as z and another as Z are two different groupings.

2. Load Puzzle method will return true if data exists in the file, it will not validate that whether puzzle will be solved by it or not.
3. Print method will return grouping variable if called before solving the puzzle.

**Test Cases:**

- **<u>Input Validation:</u>**
  - **loadPuzzle method:**
    - Pass stream as null
      - Returns false
    - Pass stream as empty file
      - Returns false
- **<u>Boundary Cases</u>**
  - **loadPuzzle method:**
    - Pass stream as 1*1 puzzle “a\n a 1 =”
      - Returns true
    - Pass stream as 2 * 2 valid puzzle
      - Returns true
    - Pass stream as 9 * 9 valid puzzle
      - Returns true
  - **readyToSolve method:**
    - Call readyToSolve with valid 2*2 puzzle
      - Returns true
    - Call readyToSolve with valid 9*9 puzzle
      - Returns true
  - **Solve method:**
    - Load valid 2*2 puzzle and solve it
      - Returns true
    - Load valid 9*9 puzzle and solve it
      - Returns true
  - **print method:**
    - Call print without loading puzzle
      - Returns empty string
    - Load valid 2*2 puzzle and solve it
      - Returns solved puzzle
    - Load valid 9*9 puzzle and solve it
      - Returns solved puzzle
  - **Choices method:**
    - Call choices without loading puzzle
      - Returns 0
    - Load valid 2*2 puzzle, solve it and call choices
      - Returns int (choice count)

- Load valid 9*9 puzzle, solve it and call choices
  - Returns int (choice count)

❖ **<u>Control Flow</u>**

- **loadPuzzle method:**
  - Pass stream as puzzle with 5 column and 4 rows with grouping
    - Returns false
  - Pass stream as puzzle with 4 columns and 5 rows
    - Returns false
  - Pass stream as puzzle with mixed number of columns and rows
    - Returns false
  - Pass stream as puzzle with equal number of columns and rows
    - Returns true
- **readyToSolve method:**
  - Pass stream where grouping operator and result are not provided for all variables and call readyToSolve
    - Returns false
  - Pass stream where grouping operator and result are not in right order eg- “b + 3” and call readyToSolve
    - Returns false
  - Pass stream where two cells have grouping of “=” and call readyToSolve
    - Returns false
  - Pass stream where three cells have grouping of “-” and call readyToSolve
    - Returns false
  - Pass stream where three cells have grouping of “/” and call readyToSolve
    - Returns false
  - Pass stream where cells have operator as “#” and call readyToSolve
    - Returns false
- **Solve method:**
  - Call solve for invalid cases covered in loadPuzzle and readytosolve control flow
    - Returns false
  - Call solve for valid puzzle
    - Returns true
- **print method:**
  - Call print with invalid grid
    - Returns empty string
  - Load valid grid but invalid grouping
    - Returns puzzle with grouping variables
  - Load valid grid and grouping and call print
    - Returns solved puzzle
- **Choices method:**

        - Load valid puzzle, solve it and call choices
            - Returns int (choice count)
        - Load invalid puzzle, solve it and call choices
            - Returns 0

- **Data flow**
    - Call loadPuzzle, readyToSolve, Solve, print and choices as normal flow
    - Call print, loadPuzzle, print, readyTosolve , solve , print
    - Call loadPuzzle multiple times and call print
    - Call loadPuzzle, readyToSolve, Solve, print and then call these in same order with different data
    - Call choices, loadPuzzle, solve, choices, print

**References:**


- https://www.baeldung.com/j
- https://www.geeksforgeeks.org/
- https://algorithms.tutorialhorizon.com/
- Research paper: *KenKen Puzzle solving with Backtracking, Asanilta Fahda 13513079 Progra Studi Tekni Informatika, Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*


**Submitted By:**


Prabhjot Kaur(B00843735)