# Project 3: Sudoku

## Abstract

This week, we had an interesting project, making a Sudoku simulation. Running my simulation will generate a 9*9 sudoku on the GUI window with a customized number of starting values in randomly selected location and chosen value. The program is able to solve the sudoku and assert if the solution is correct or sudoku is unsolvable. To accomplish, the Stack data structure was used to store each attempts and to allow backtracking if necessary. I also explored the relationship between number of starting values and solvability, yielding an interesting result in which the sudoku becomes more unsolvable with greater number of starting value.

## Core Data Structures

- `Stack` I used this Last-In-First-Out (LIFO) data structure to store each attempts when solving the sudoku. It stores data in a way resemble to how we stack textbooks - what you lastly put on the top need to be taken out first. The unique allow us to backtracking when we need to go back to the previous attempt and abandon the current one. The `CellStack` I used were implemented in the lab, mainly having two important methods. `push()` method stores an new data in to the `stack`, while `pop()` method removes the last entry of data and returns it.
- `HashMap` I used this dictionary-like data structure to store the `Color`, benefiting from its `O(1)` time complexity when accessing those `Color` when drawing each cell. In my extension, you will see I allowed user to select various color themes to display the sudoku. In each theme, one particular value has one unique color, some in a gradient trend while some implementing a colorful distribution of color. To store these themes, I simply used `HashMap<Integer, Color>` for the program to accessing. For example, when drawing an `Cell` of value 1, the program will call `g.setColor(map.get(1))` to set the color of value 1 before drawing this cell.
- `HashSet` I used this set-like data structure when trying to find the best cell to put attempts. This is like the `Keys` of a `HashMap` object, since it does not allow identical data. I will describe this further below at `getNumberOptions()`.

## Important Tasks

The following methods are crucial to the simulation, thus they are briefly described below.

1. `findBestCell()` This method mainly iterates through the board, and returns the most suitable cell for the `solve()` method to work on. I actually implemented two of this method, another one called `findBestCellNaive()`. Different to the naive one which iterates from the top left corner (0,0) to the bottom right corner (8,8) and looking for the first cell with value 0 (meaning this cell is unsolved), this `findBestCell()` method search for the cell with the fewest number of possible options. To do this, it goes through every cell that is unlocked (unsolved), call the `getNumberOptions()` method on every cell, store what it returns as options. If `options` equals to 0, it will return null to `solve()`, meaning there is one unsolved cell with no possible solution, which forced `solve()` to perform backtracking. If not, it will compare to `min`, if smaller than `min`, it means this cell has the current fewest options. After every free cells are gone through, the cell with fewest options will be return. In this way, the speed of solving increases a lot, which I will demonstrate in the extension part.

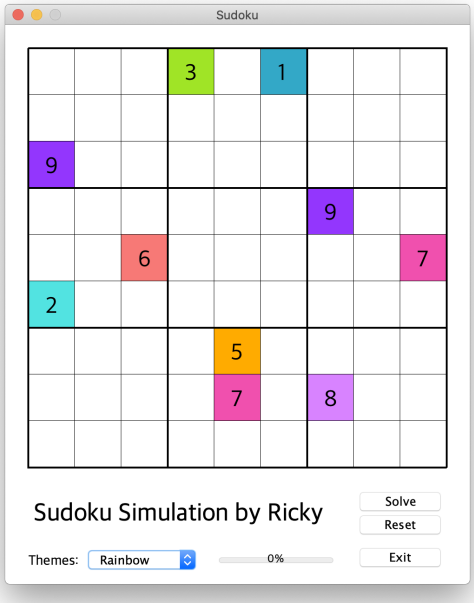
2. `getNumberOptions()` This method is called when the code computes number of possible solution on one free cell. At first, I used a slower algorithm which iterates from 1 to 9 to test if there's same value located in the same row, same column or same box. This method clearly tasks longer time, so I came up with another, using `HashSet`. As I mentioned above, this data structure does not allow same data to be stored twice, and how I benefited from this feature was, I iterated through the row, column and the box where the cell is located, add every value encountered to the `HashSet`. Since same value cannot be stored, it will contain all **unique** values at last. What I need to do is just to subtract 9 by the size of the `HashSet`, which represents the number of values that have not been present in those row, column, and box, which is, the number of possible options.

3. `solve()` This method was the most crucial one when solving the sudoku. Its job can be divided into few parts. The first job is pushing the triable cells into the stack, and assigning that triable cell with the least possible value, found by the `getValidValue()` method. Another job is when no other triable cells exist, popping out the last entry of cell, assign it with a bigger value, also accomplished by the `getValidValue()`, and pushing back to the cell but

now with the bigger value. This method was already extensively described in the project's description, so I will not take this description further.

## Result

Two screenshots consisting of a board prior to solving and after solving are shown below.





I also ran the simulation with various starting values (from 10 to 40) for 10 times. The following table presents the result.

Number of Step Taking to Solve a Sudoku of Various Number of Initial Values

| | 10 | 15 | 20 | 25 | 30 | 35 | 40 |
|---|---|---|---|---|---|---|---|
| **Repetition 1** | 71 | 78 | 61 | 72 | Unsolved | Unsolved | Unsolved |
| **Repetition 2** | 71 | 163 | 61 | Unsolved | Unsolved | Unsolved | Unsolved |
| **Repetition 3** | 71 | 66 | 7748 | Unsolved | Unsolved | Unsolved | Unsolved |
| **Repetition 4** | 74 | 66 | 9858 | Unsolved | 273 | Unsolved | Unsolved |
| **Repetition 5** | 71 | 69 | Unsolved | Unsolved | Unsolved | Unsolved | Unsolved |
| **Repetition 6** | 78 | 82 | 253 | 320 | Unsolved | Unsolved | Unsolved |
| **Repetition 7** | 111 | 66 | 89 | 199 | Unsolved | Unsolved | Unsolved |
| **Repetition 8** | 71 | 482 | Unsolved | Unsolved | Unsolved | Unsolved | Unsolved |
| **Repetition 9** | 76 | 69 | 70 | Unsolved | Unsolved | Unsolved | Unsolved |
| **Repetition 10** | 71 | 79 | Unsolved | Unsolved | Unsolved | Unsolved | Unsolved |

From the table above, we can see that,

1. The greater the number of initial values the sudoku has, the bigger the chances of having no solution is.
2. Although not compared by time taken, the steps used to reach a solution becomes more and more when the number of initial values increase.

# Extension

## Extension 1 - Two findBestCell() Methods

I tried two `findBestCell()` methods here. Both of them mentioned above. The `findBestCellNaive()` simply finds the first cell with value 0, while the `findBestCell()` iterates through the board and calls `getNumberOptions()` on every free cells in order to return the cell with fewest possible options. On one particular board, the `findBestCellNaive()` took 171 steps to reach to the solution, while `findBestCell()` took only 67 steps.

## Extension 2 - Fancier GUI

In order to make the visualization look better, I added

1. **Buttons** for **Reset**, **Solve**, and **Exit**. The **Solve** button starts the solving process of one sudoku board, and the **Reset** button cleans the previous board and set up initial values again which allows a new solve process. The **Exit** button disposes the window and calls `System.exit(0)` to exit.
2. **Progress bar** to indicate the completion of sudoku while solving it in the simulation.
3. **Color Themes** to allow user to select one from all five color themes, which I have named "Ocean", "Peach", "Rainbow", "Modern" and "Material". Every theme is called based on the color palette the theme has, and cells of different values will be drawn with different color. For example, in the "Peach" theme, cells are set to have color ranging from light pink to darker pink, resembling the color of the fruit peach. To allow selecting, I added a `JComboBox<String>` to the `Landscape`. Besides the color of different cells, I also used "Apple SD Gothic Neo" to draw the number of the board, and put black nines to separate among those nine boxes.
4. **Title** of "Sudoku Simulation by Ricky" in a customized font called "Apple SD Gothic Neo", my favorite font.

5. **Statistics** to show the result and attempts taken to reach a solution or a conclusion of no solution.

> 💡 The demonstration of this visuals can be found in the `Project_3_Demo.mp4` in the project folder.

## Extension 4 - Automated Process

In my `Simulation` class, I automated the simulation process and repeated it for 1000 times, for every integer from 10 to 20, which is the number of initial values. The result is summarized into the following table.

| Number of Initial Values | Average Steps Taken |
|---|---|
| **10** | 76 |
| **11** | 160 |
| **12** | 1805 |
| **13** | 4635 |
| **14** | 352 |
| **15** | 367 |
| **16** | 560 |
| **17** | 98 |
| **18** | 344 |
| **19** | 403 |
| **20** | 86 |

## Extension 5 - Read Sudoku That Has Extra White Spaces

I adapted my code so that it can read extra white spaces now. The main difference is, I used regular expression in the `split()` method, so that it splits not only blank spaces but also new line.

## Learning Outcomes

I am quite familiar with the sudoku and stack, so I didn't really learn a lot from this aspect. Instead, I learnt more about various components in Java Swing.

## Acknowledgment

In this project, I didn't ask for anyone for help nor refer to any documents this time.