Project documentation

1. The title of the project is 'sudokuu', realized by Marius Sclearuc (student nr. 100755983), a first-year Data Science student. This document has been created on April 16th, 2023.
2. The project represents a Sudoku Killer game with scala backend and scalafx-based frontend. It can read & write games to JSON files, insert values inside the game grid, display possible sum-splits for a selected region, light up possible values when hovering over a cell, and some other minor functionalities. In short, this game represents a helper for the Sudoku Killer game. Neighboring grid regions have been colored with different colors; hence the game was completed on the 'demanding' difficulty level.
3. Once the user starts the app, the game window automatically pops up. In the middle, it displays a text encouraging the reader to double-click there to open a new game. Alternatively, the user can open the game through the `Game` menu or press "Ctrl + O". Both options work in the same way, opening the file explorer and urging the user to select a .json file.

   In case the user selects a good json-game file, the game loads up, and the grid is display on the left side of the window. A bubble containing all the possible sum-splits for the selected cell appears on the left side of the window. Underneath, a line of buttons is created. The buttons 1-9 are there to insert values inside the selected cell, `Delete` deletes the entry from the selected cell, and `Check` is available to press only when the game grid is full.

   Initially, there is no selected cell. The user can select one by simply clicking on it. Once he has clicked on it, he can:
   - insert/delete a value by pressing one of the buttons below
   - insert a value by pressing one of the 1-9 keys on the keyboard
   - delete a value by pressing backspace/0
   - select the cell on the right/left/above/below by using the respective arrow keys
   - deselect the cell by pressing Esc.

   Every time the user does an action (i.e. selects a new cell, inserts a value), the content of the right-hand-side bubble changes.

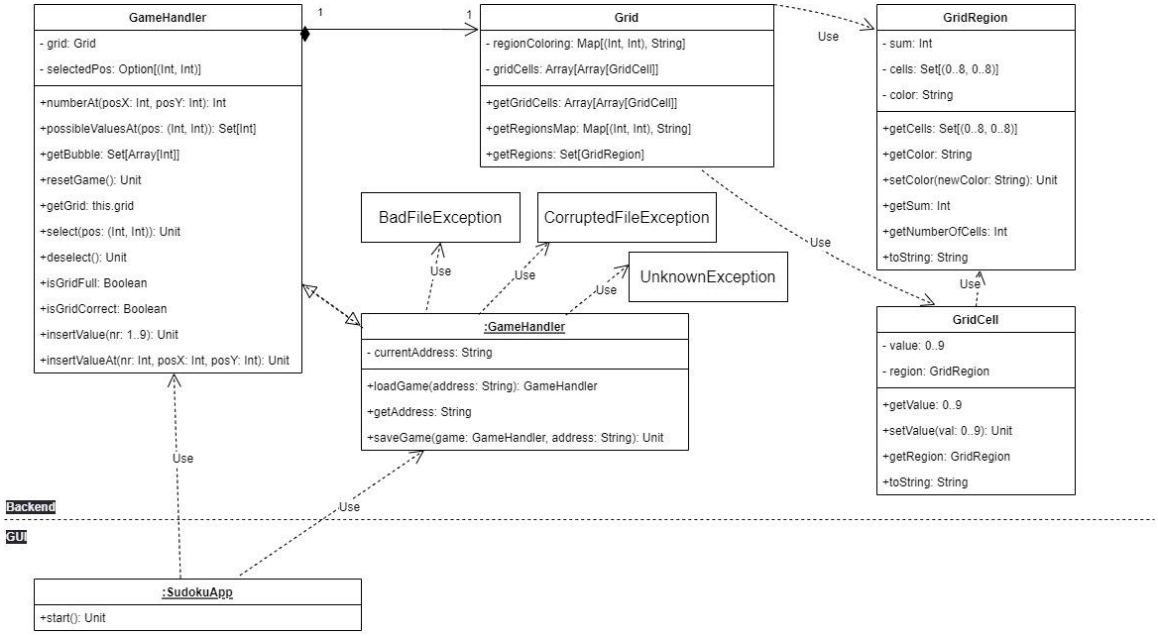
   When the user hovers over a cell, the possible values (i.e., ones that don't violate the rules of killer sudoku) light up in their respective buttons from the row below. When the user hovers over a numbered button (for example, 5), all the cells containing an instance of that number (so all the cells containing nr. 5) light up.

   In case the user gets bored of the game, there's a "Exit" option inside the menu. Similarly, there's a "Reset Game" option and some saving options.

   When the board is full and the user presses `Check`, the app automatically checks if the user's solution satisfies all the required rules. If no, a pop-up window informing of a mistake somewhere appears; if yes, then another pop-up window appears, asking the user how they want to proceed. The user can exit the app, reset the current game, or just press cancel (i.e., no action will be taken

4. The diagram below represents the final class structure. The main class, upon which everything else was built (i.e., depends), is GridRegion. A GridCell depends on the GridRegion, as it points towards which GridRegion contains it. A Grid consists of a 9x9 array of GridCells, but it also contains some GridRegion-related methods. A GameHandler class contains the game Grid (obviously), along with the currently selected position & a variety of methods related to the game flow. It also has a GameHandler companion object, one used to implement the static methods loadGame and saveGame, but also for storing the last-used address of the game. Finally, there's a SudokuApp object extending the scalafx JFXApp3.

   The most important class for the game is GameHandler. It handles all the logic, saves the current game state, etc. The user interacts with the game through the sudokuApp object interface.



   At the time of creating the UML structure & implementing the backend-code, this class structure felt like the right one, as it delimited all the important aspects of a sudoku killer game (i.e., the cells, the grid, and the regions) clearly. However, once I was implementing the front-end, I realized that I overthought the structure, making it too complex and lacking some useful implementations.

   Namely, the whole Grid-GridCell-GridRegion cycle could've been just one big "Grid" class containing a region-to-cells Map[Int, (Int, Int)] and a gridCells: Array[Array[0..9]] as attributes. Moreover, implementing the GameHandler class as an extension of the scalafx Observable trait could've saved so much more code and trouble. Nevertheless, the game is up and running, which I do consider to be a good result.
5. Following are the 6 main algorithms used to create the app:

- The coloring of the GridRegions was done using the greedy algorithm. Below, the algorithm is represented in pseudo-code:

```
def colorGraph(graph: List[(GridRegion, Set[GridRegion])]): // the graph is just the representation of neighbouring edges on
the grid
    initialColors = List(...)                          // helps maintain (some control) over the colors that appear inside
the graph

    take 0th element and set its color to initialColors[0]

    for region in remainingRegions:                            // for all the other regions
        colorsUsedByNow = graph(region).map( x => x.getColor )  // get the color of the neighbours
        freeColors = initialColors.diff(colorsUsedByNow)        // get the colors that haven't been used by neighbours
        if freeColors isEmpty:
            newColor = "#" + randomSixDigitString              // generate a new random colorGraph
            region.setColor(newColor)
            initialColors += newColor
        else:
            region.setColor(initialColors[0])
```

  Implementing this algorithm was quite easy. Moreover, it has the advantage of being quick & effective, albeit sometimes it uses quite a lot of colors.
- To check the contiguousness of a region, a DFS algorithm was implemented. In simple words, the algorithm picks an initial random vertex (i.e., coordinate of the form (Int, In)). It randomly chooses a neighbour of this vertex and "goes" there, adding the initial vertex to a list of "passed" vertices. Next, it once again chooses a random vertex, but this time also making sure this vertex doesn't belong to the "passed" list. It continues hopping from neighbour to neighbour, adding instances to the "passed" list, until there's no vertex remaining in the "unpassed" list. It then compares the list of "passed" vertices with the whole list of vertices. If they're the same, it means the region is contiguous. If not – not contiguous.

  Some other solutions might have been possible, including a simple if-else block, as we're guaranteed that a region has at most 4 cells, hence checking them by hand wouldn't be that tedious. Nevertheless, I wanted to challenge myself a bit by implementing such an algorithm.
- The accounting bubble sum-splits aren't calculated in the most effective way, but for the sake of this program they're working quite fast & well. The underlying algorithm works as follows:
  - given a region r, n is the number of cells it contains
  - possibleSums is a Set[Array[Int]] object, initially containing all the possible n-sums (i.e., all the combinations from 1 + 1 + … + 1 up to 9 + 9 + … + 9, in total 9^n combinations).
  - distinctVals is how many distinct values does this region at least have. If r is in form of an "angle" intersecting 3 different 3x3 squares, then obviously the 2 vertices at the "edge" of the angle can have the same value. distinctVals tries to account for it
  - from possibleSums the algorithm filters all the arrays that have the same sum as the r.getSum
  - it groups them by _.toSet (there can be a lot of permutations for the same sum, we need but 1)
  - once again, filtering. This time the algorithm makes sure that the set (which serves as a key) contains at least distinctVals elements AND that all the values that already appear in this region are part of this set.
  - after that, for each entry remaining, we get the head of the value-Array and take out the values that have already been used.

- the algorithm working for discovering which numbers can fit into a given cell works as follows:
  - initially, it takes all the sum-splits that could appear inside the region containing the give cells and flattens them.
  - it takes out all the numbers that already appear in the same line/column/3x3 square and outputs the difference.
- GameHandler.loadGame():
  The algorithm is very important to the game, as everything loads through it. The implementation itself can be seen inside the code itself, but in short, it works by opening a .json file, checking for the structure, creating the required grid, coloring the regions appropriately using the algorithm described above, and then returning the GameHandler class instance.
- GameHandler.saveGame():
  Pretty much just like the algorithm described above, except in reverse.

6. Only native scala data structures were used.

   Mainly, I used Arrays for storing information where order/number of occurrences mattered (such as each of the accounting bubble's possible sum-splits or the activationTrackingProperty of the sudokuApp interface).

   Similarly, Sets were used wherever order & number of occurrences wasn't an important factor, thus helping reduce the amount of data stored (instances include GridRegion.cells & the accounting bubble's sum-splits).

   To clarify some possible confusion, the accounting bubble is a set, as the order in which the sums appear doesn't matter, whereas the elements of the bubble are Arrays, as the number of occurrences of each element inside the Array matters.

   Maps were used occasionally, mainly in cases where a storage of keys & their respective values was deemed to actually reduce operational time (such as Grid.regionColoring).
7. The program uses no Internet access.

   The format used for storing data is JSON. The file format is represented below:


```
[{
  "value3": 0,
  "regionSum": 22,
  "x3": 2,
  "y0": 5,
  "x0": 0,
  "y1": 4,
  "value0": 0,
  "value2": 0,
  "numberOfCells": 4,
  "x2": 1,
  "y3": 4,
  "value1": 0,
  "y2": 5,
  "x1": 1
}, {
```

```
 "regionSum": 20,
 "y0": 3,
 "value1": 8,
 "x0": 4,
 "y1": 3,
 "value0": 5,
 "value2": 7,
 "numberOfCells": 3,
 "x2": 6,
 "y2": 3,
 "x1": 5
}, ...]
```


4 example files have been included with the source code under the src\test\testFiles directory.

8. A file with automatic unit tests was created under src\test\scala\gameBackendTest. Some of the app's most important algorithms (namely the ones presented above, bar the one returning possible values for a given cell) were tested there. All the errors that occurred have been fixed. Hence, the backend of the program works flawlessly (to the current knowledge).

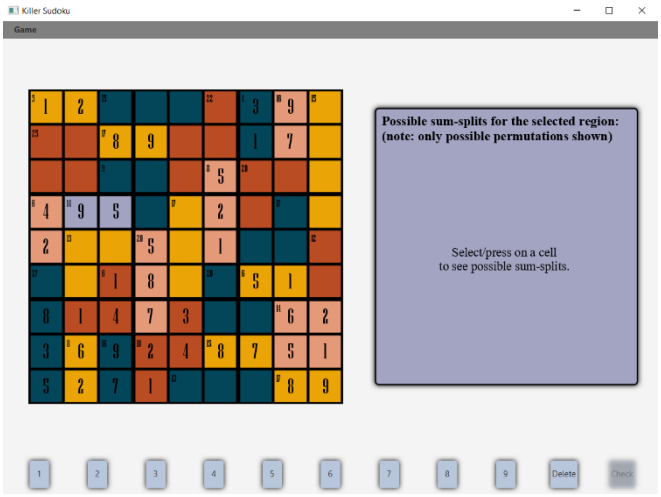
   System testing was done by tinkering with the game myself, but also by some friends who were glad to help my work. No big bugs were found, and everything was patched. Hence, the app's GUI is almost flawless (see the 2nd point of the "known bugs" section).

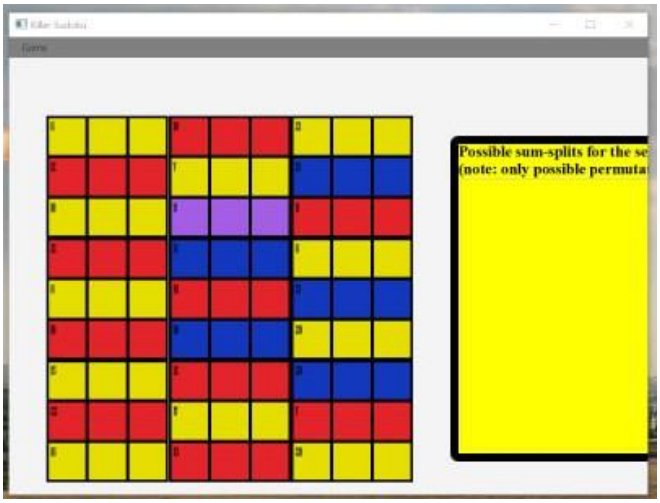
   The first test took quite a while to implement, as, besides the testing itself, I was setting up the "general testing practice" for my program, a set of (unwritten) guidelines I'll be following in order to write down all the other tests. Afterwards, testing has proved itself to be quite an easy thing to implement. A rough estimation on the time spent developing these unit tests is 5h.

   A possible shortcoming of the system testing is that the game has been tested only 2 devices, both with similar screens/aspect ratios. No issues have been spotted on either of them (see the 2nd point of the "known bugs" section).

   The testing process was fairly similar to the one I had planned in my mind before starting to create the app.
9. To my knowledge, the app currently has the following 2 big bugs:
   - Once the sudoku grid is completed and the user presses "Check", a pop-up window inquiring the user for future action pops up. In case the user presses a button, it goes all well. However, when the user tries to press the "X" button in the top-right of the pop-up window, nothing happens. It seems like that button has somehow been deactivated, although it's weird, as other segments of the game use the same method to create pop-up window and their respective "X" button works. I've tried fixing this issue for 1 week now; however, it seems that nothing is working. If I had more time to try and solve this issue, I'd probably dig deeper inside the scalafx Alert class (the one producing this pop-up window) and try to understand which method gets called when the "X" button is pressed and manually override it. All the attempts at doing that have been futile until now.

- In order to avoid issues with window-resizing, I decided to fix the width and height of my app's game window. On my own laptop it has been displayed nicely all the time (see on the right, first pic); however, when I connected my laptop to another screen (one with different resolution, aspect ratio, etc.) and tried moving the app window on that secondary screen, all the elements of the app resized badly (see on the right, second pic). I'm not sure about the source of the issue – maybe it's a scalafx internal issue, as when I've left some freedom for the app window & moved it to the secondary screen, everything came back into place perfectly as soon as I tried resizing. Or it might also be an issue of my app, as all the elements have fixed size and it might cause some display-trouble when moving to other screen. Anyways, if I had more time & more accessible screens with different characteristics, I'd try connecting and displaying my game on all of these screens. I'd try to see if there's a connection between the screen type and the outcome. And, most importantly, I'd take out the fixed size character of my app window, trying various sizes and finding the one that fits best for my game.





Regarding the features, all of them seem to have been implemented and working properly.

10. 3 best sides:
    - Ease-of-use: Being myself a user who doesn't really enjoy taking their hands off the keyboard, I tried to help design the game to have as many keyboard-functionalities as possible. This includes switching cells using arrow keys, easily moving the selected cell right/left/up/down, inserting values using keyboard numbers, and even opening/saving files using shortcuts.
    - Improved accounting-bubble algorithm: The suggested dummy algorithm, albeit being quite simple to implement, suggested a lot of 'dumb' sum-splits. With it being quite the sight for sore eyes, I decided the algorithm needs improvement. Although now it helps the user a lot in solving the game, I do believe that's the sole purpose of the accounting bubble – to be as useful as possible.

- Eye-pleasant color scheme: I've tried to make the game's color scheme as pleasant as possible. Not too bright, not too dark, not too contrasting, not too similar – everything just right.

3 weaknesses:

- Fixed size window – I think this speaks for itself, as a 960x720 window might be too big/small for some screens.
- Persistent bugs – This is also quite obvious. Although these bugs aren't that prolific as to impede the well-functioning of this program, I do consider them an obstacle for the user to enjoy the game.
- Ugly code – As previously mentioned, I would've extended the Observable trait (for the GameHandler class) if I only knew about this before. In not doing so, I had to create a lot of variables that doubled themselves (inside both the GameHandller class instance and the sudokuApp object). This also resulted in the sudokuApp code being over 450 lines long, when in reality I believe around 300-350 lines should be enough for it.

11. Deviations from the plan, realized process and schedule

Around 3 weeks were spent on implementing the backend; afterwards, 2 weeks were spent on the 2nd part of the project, namely the GUI. Finally, about 1 week was spent on creating the unit tests, tinkering with the game myself & with friends, but also on writing down the final report.

Looking back at the schedule outlined in the technical project plan document, it's noticeable that I underestimated the implementation of the backend by ~1.5 weeks. Luckily for me though, the testing part of the game wasn't that painstaking, and so I still managed to finish the project pretty much at the expected time. All in all, I can say that my time-planning wasn't good enough (an important aspect to improve on!), but I still managed to do everything in time.

Another important lesson is that a bottom-up approach isn't necessarily the best one. If I were to have some ideas about scalafx and the way it works, I might have saved considerable amounts of time; nevertheless, my stubbornness to have a "product" (i.e., GameHandler) to work with proved fatal in this case. For future projects, a hybrid between bottom-up and top-down approach will probably be my inclined choice (along with thorough documentation of new interfaces in order not to regret decisions made early in the project!).

12. For the final evaluation, quite a lot of things could be said. I could reiterate for the 100th time how proper research into scalafx and its perks could've saved me so much time & ugly code. Or how the persistent bugs of my app & the fixed size of my window lower the satisfaction level while playing this game. Or how the improved bubble algorithm & the easy-to-use commands helped make the app somewhat enjoyable.

However, I believe I described them in enough detail above. What matters now is that, if I were to restart this project today, I'd probably choose a totally different approach. I wouldn't barge head-in to the backend, disregarding the usability of my app. I would probably choose some interfaces that have better documentation than scalafx (yes, going through that documentation often felt like finding a needle in a haystack, as it had lots of stuff but nothing related to the I encountered). And I'd surely make myself a better, proper development schedule.

Nevertheless, I'm pleased with my progress. On a personal level, I believe I've

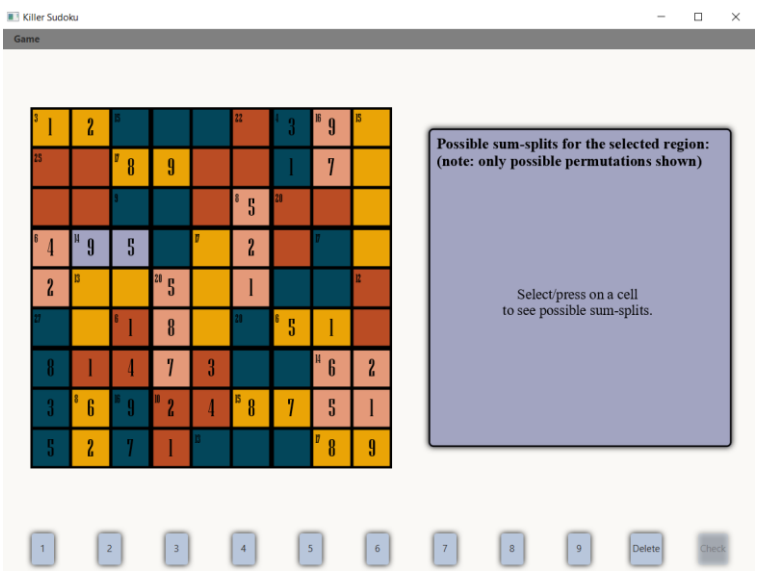
learned much more useful lessons from this project than from any of the other courses I took (maybe it was also because I sometimes felt like the captain of a ship going into a storm with but a compass-tutor to occasionally guide my way and no other real help in there, so I had to get my stuff together and pull through). But one can be sure, never again will I be engaging in such a project all by myself :)

13. References

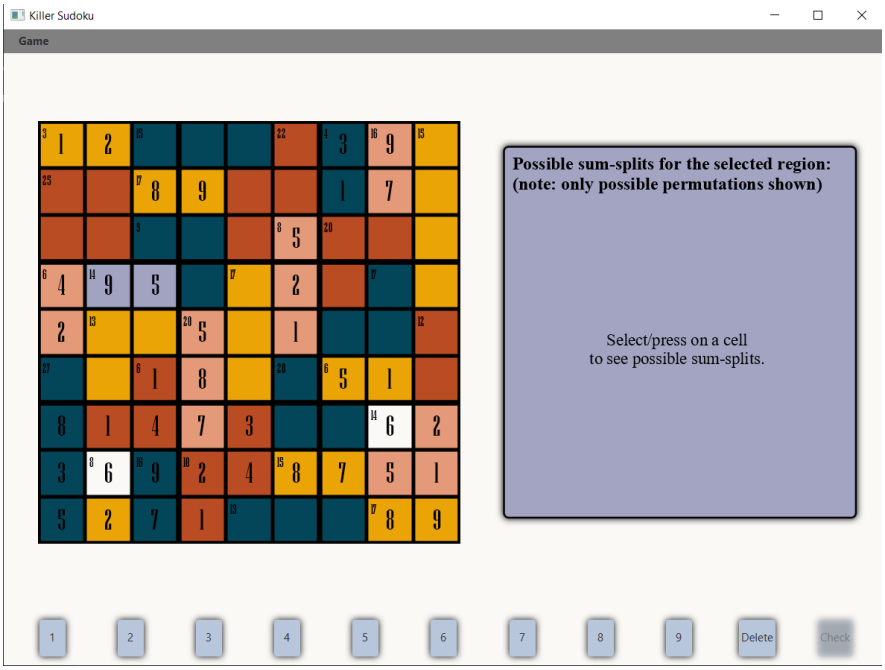
    https://www.scala-lang.org/ - for general scala help
    https://www.scalafx.org/ - for scalafx documentation (quite a bad documentation in my opinion, but sometimes useful nonetheless)
    https://stackoverflow.com/ - mainly for troubleshooting the main scala algorithms (the ones mentioned above)
    https://chat.openai.com/ - for helping me dig through the haystack of documentation scalafx had to offer.
    … - and probably some other minor help.

14. Appendixes

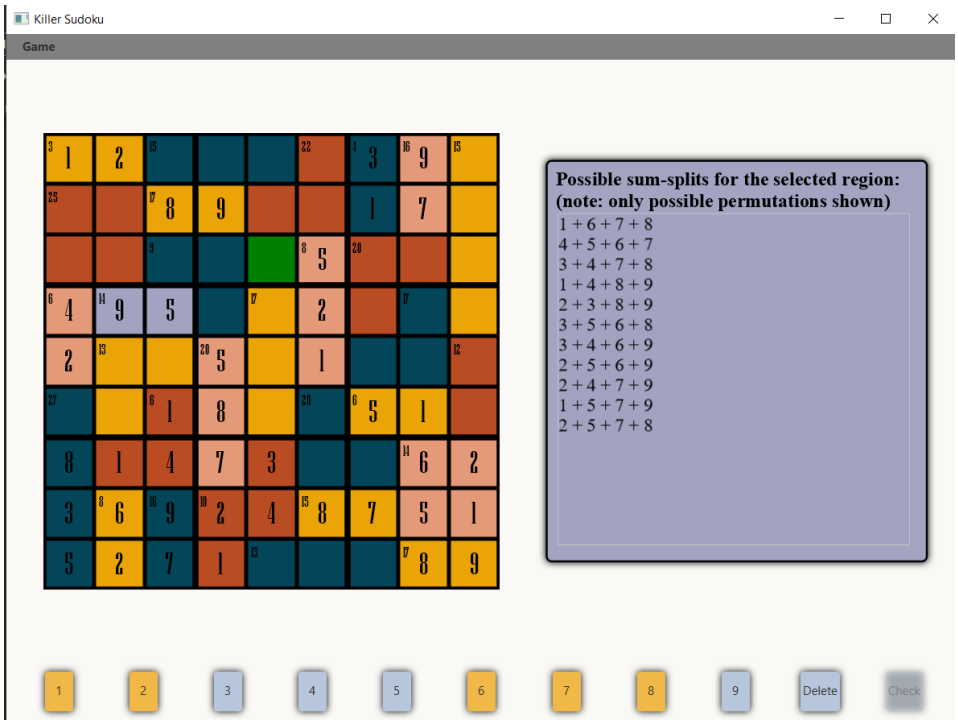
    Source code: https://version.aalto.fi/gitlab/sclearm1/sudokuuu
    Screenshots of the gameplay:



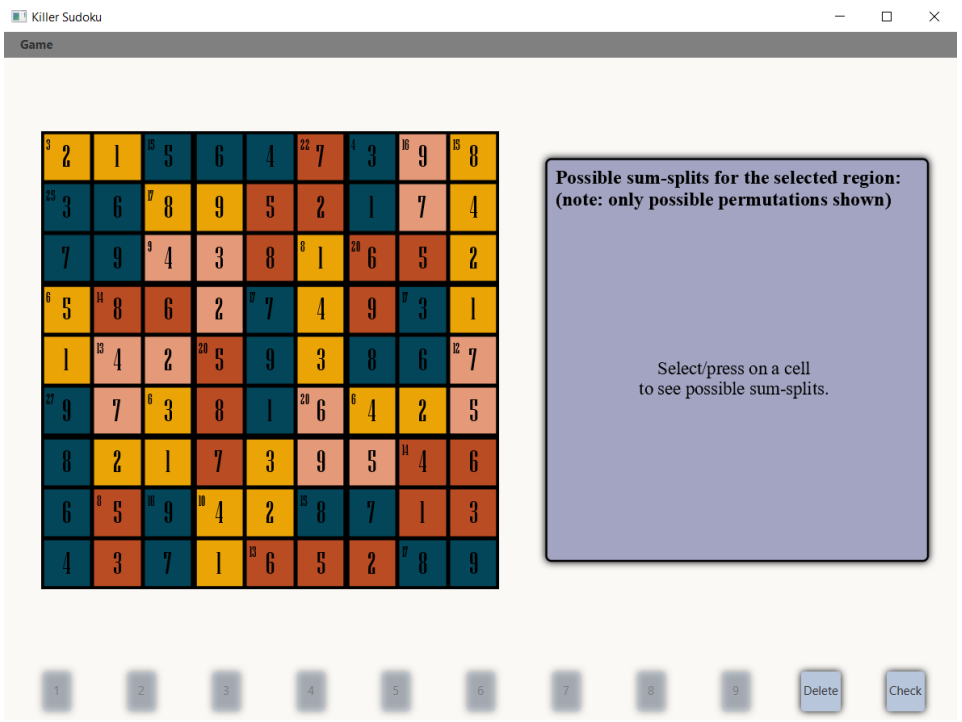
While hovering over number 6 (mouse not shown in the picture):



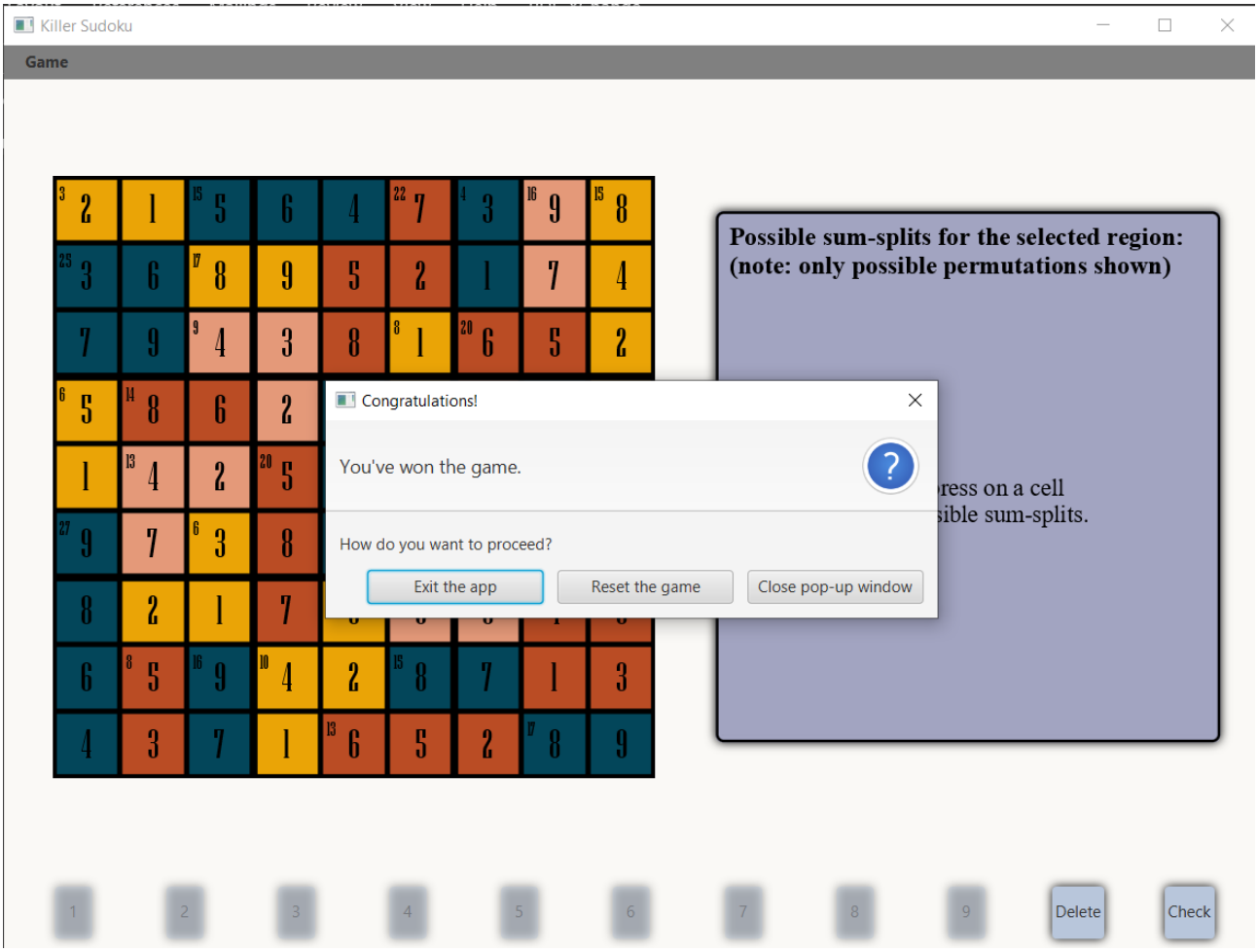
While hovering over the green cell:



With a completed grid:



Upon clicking `Check` (next page):



After clicking the app window's "X" button: