Aditya Challamarad: Sudoku!

Table of Contents

Analysis

1.1 Identification of end-user:

The end-user for my project will be Akshay, the club leader for the after-school puzzle club. Currently, the club consists of about 12 members including Akshay. The club was established in 2021, and they have held competitions in school for puzzles like Chess and the Rubik's cubes. Akshay has recently introduced sudoku as a game at the club. As club members have not played Sudoku before, they will go through weekly practice to master the game. Once the players are advanced enough, they will hold a sudoku tournament. Akshay requires an app for the members to learn and play sudoku on. This means that my project will need to have a range of puzzle difficulties. As the members get more proficient at sudoku, they can attempt harder puzzles.

1.2 Interview (Akshay: club leader):

Date: 23/02/2022

Asking these questions was important as I want my program to meet all the end-user requirements.

Q. What is the current system used?

A. Currently, we are using daily newspapers which come with a sudoku puzzle, and a few other games such as crosswords. One of our members has a subscription to the newspaper and brings it in every week.

Q. How frequently do you play sudoku at the club?

A. The club meets up every Tuesday and Friday after school. As sudoku is a puzzle we have just introduced, we will be prioritizing learning sudoku at least once a week.

Q. What features would you like in the program?

A. We would like the program to give us clues when stuck rather than giving us the entire answer as this will help us improve and continue the question instead of moving on to the next one. Our current system only gives us the answers. We also want a timer to see how long we took to solve each puzzle.

Q. Would you like a login system?

A. Yes, we would require multiple accounts for our members so that each member can solve the different puzzles at their own pace. I would also be able to look at their progression.

This question has made me reconsider my initial idea as now I have to make a secure login system. Akshay and I will be the admins of the login system and the database and will be able to see player progression on the database. All the other members will be the users of the system.

Q. Do you need any statistics displayed during or after a game?

A. Yes, that would be helpful. I want the members to see how long they spent on a given puzzle so that they can track their speed and prioritize increasing it.

Q. Do you want the statistics to be saved?

A. Yes, I want the statistics to be saved so that the members can see their progression throughout the different games.

Q. What computers and operating systems do you plan to use with the new system?

A. The members will be using their laptops which they bring to school. Most of them use Windows while the others use macOS.

I will need to make sure that my program is compatible with both macOS and windows.

Q. What issues do you have with the current system of using newspapers that you would like to see fixed in my program?

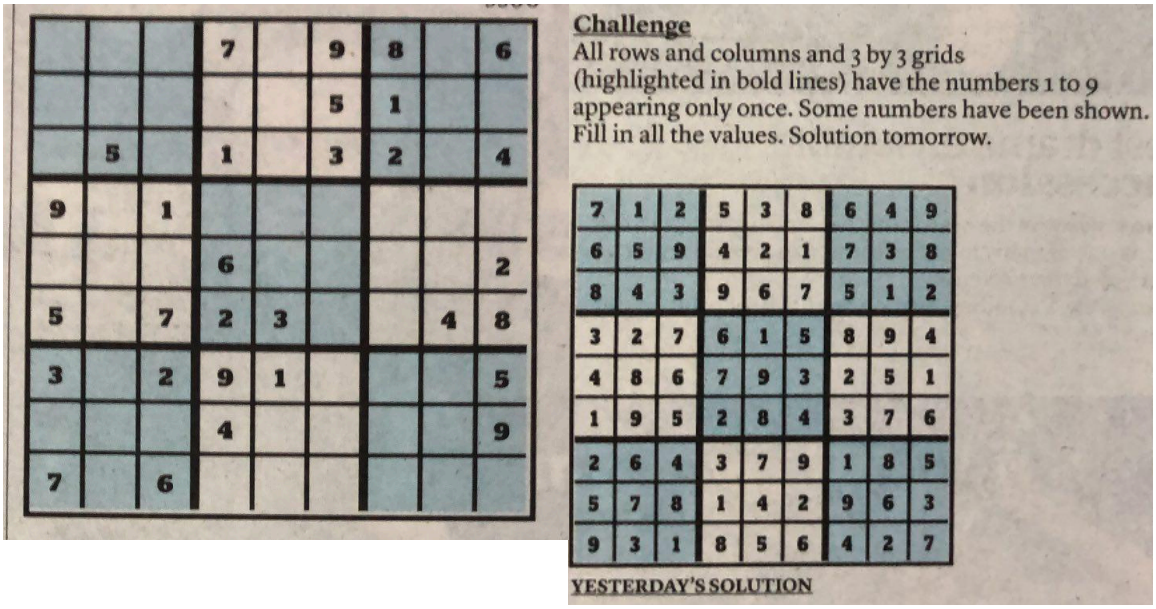
A. As the newspaper's sudoku answer comes the next day, the members must usually wait till the next week's sudoku club meeting to find out if they were correct. The newspaper only comes with 1 sudoku puzzle and I want the members to be able to start a new puzzle whenever they finish the previous one. As only one of our members has access to the newspapers, sharing the puzzle with other members becomes difficult. The puzzles in the newspaper are easy every day and they require harder puzzles to do once they improve.

Asking this question tells me what features are important to the end-users and what features I need to add to my program.

The interview has been valuable as I now know what my end users require in the program and what problems I need to solve for them. The most important feature in my opinion would be the login system so that each member can progress on their own whenever they meet up. Each member will not have to work on the same puzzle at once. This interview has substantially changed my idea of what a user might require from a sudoku app.

1.3 Existing System:

One of the members of the club with a subscription to the daily newspaper brings 3 newspapers to school on the days the club meetings take place. During the lunch break, Akshay, who has access to use printers prints 11 other copies of each of the newspapers which can take up to the entire break. During the meeting, each member tries the solve the same puzzle at once so that times can be tracked easily. A timer on a computer is used to measure the time taken. There is no clue system at this point because clues can be shared only after one person finishes the puzzle and is correct. Clues can be given from that person's answers as no one has the solution to the puzzle until the next day.



Challenge
All rows and columns and 3 by 3 grids (highlighted in bold lines) have the numbers 1 to 9 appearing only once. Some numbers have been shown. Fill in all the values. Solution tomorrow.

7	1	2	5	3	8	6	4	9
6	5	9	4	2	1	7	3	8
8	4	3	9	6	7	5	1	2
3	2	7	6	1	5	8	9	4
4	8	6	7	9	3	2	5	1
1	9	5	2	8	4	3	7	6
2	6	4	3	7	9	1	8	5
5	7	8	1	4	2	9	6	3
9	3	1	8	5	6	4	2	7

YESTERDAY'S SOLUTION

Gulf News newspaper (not the same puzzle)

1.4 IPSO: (newspapers)

Input:

- Members solve the puzzle and make notes if needed on the paper.
- The names and times of each player are also written on the paper.

Processing:

- The next day, one of the group members is tasked with checking if a member solved the puzzle correctly by comparing it with the answer provided for the previous day's question in the newspaper. The fastest time with the correct answer wins.

Storage:

- All the papers are collected and stored physically in a drawer.
- The time to solve for each member is recorded and stored on an excel sheet on a compute

Output:

- At the next meeting, the members find out if their answer was correct or not.
- The winner is announced.

1.5 Problems of the current system: (newspapers)

The major problem with the newspaper sudoku is that the solutions they provide every day are for the previous day's puzzle meaning that players must wait a day to know if their solution was correct. Newspapers only come once a day leading to a very limited number of puzzles. The puzzles are always easy and around the same difficulty making improving in sudoku difficult as there is no increase in difficulty. Paper puzzles cannot be shared easily. Newspapers don't give any clues to the problem meaning that if the player can't solve a problem, they look at the solution and move on to the next problem. Papers can also be torn easily and can be damaged by water. Saving multiple puzzles in paper form can get difficult and can be easy to lose. Players will have a hard time writing on paper if there is no hard surface to place the paper on. Newspapers also don't keep track of time and the number of mistakes made, making it difficult to see if the player is improving.

1.6 Constraints and limitations:

As I have limited knowledge of GUIs, I'll have to learn to make appeasing GUIs in detail to produce the sudoku GUI. Additionally, coming up with a secure and complex hashing algorithm will take time and will not be perfectly secure. Due to the pandemic, my class only has about 2 months to turn in the NEA so I might not be able to implement all my intended features in time, however, I will make sure the program works for its intended use before adding in extra features.
My original plans were to build an AI which solves sudoku, but I have resorted to using a brute force recursion algorithm to solve it because of the difficulty of learning an advanced topic like AI in time. Furthermore, I will have to spend time learning about Database management and manipulation in Python via SQLite3 before adding features like the login system and storing sudoku boards.
My end-user does not have much experience with coding and may not be able to solve any bugs he encounters within my code. He will need to reach out to me to solve the bugs. I will try to solve all the bugs I encounter.
My program will only run-on computers which have Python installed. This system will not be designed to work on mobile devices as they might not have Python on them.
As my end-user and his club will use this on their laptops, I must make sure that my program works on macOS and Windows.

1.7 Existing Solutions:

www.sudoku.com

Sudoku.com is packed with every feature a sudoku player would require, from basics such as difficulty levels to advanced techniques like notetaking. The game has an option to highlight when a number conflicts with a user inputted number and has the option to undo and erase. I plan to have all three of these features. Sudoku.com also comes with a timer that can be paused. I will be implementing a timer so that I can store the times for Akshay to see the members' progressions. Sudoku.com also has android and iOS app versions of the site. Furthermore, Sudoku.com has a note-taking option that helps keep notes of which numbers can go in a cell. As my end clients are beginners at the game and the puzzles will be relatively easy, adding a notes option would be unnecessary and might make the game more complicated to learn. Sudoku.com has an option for clues that display the answer of the cell the user was on when clicked. I intend to have this feature to help the group members learn.

1.8 Objectives and User requirements:

LOGIN

1. Allow users to enter a Username and a password using 2 input boxes (OOP)

2. Allow users to click on a register button to create a new account

    a. If the username or password field is left blank, the user is prompted to fill in both fields.
    b. Each account will require a unique username. If a username already exists in the database, the user will be prompted to enter a different username
    c. The program will only allow passwords with certain complexity to be registered. This will be achieved using regular expressions
        - A password must be at least 8 characters long
        - Password must have at least 1 uppercase and 1 lowercase character
        - Password must have a symbol (!, @, #...)

    d. If a valid username and password are entered and the username does not exist in the database, register is successful
    e. Save passwords using hashing

3. Allow users to click on a login button to access their menu if credentials are correct
    a. If the username or password field is left blank, the user is prompted to fill in both fields.
    b. Correct credentials will allow the user to access the user menu
    c. If the username does not exist, the user is alerted that the name does not exist on the database
    d. If the password does not match, the user is alerted that the password does not match the password associated with that username on the database.

USER MENU

4. Allow users to log out when the log out button is clicked. The user will be taken to the login page.

5. Allow players to see what boards are completed.
    a. Show board numbers that have been completed by the user
    b. Show board numbers that have been completed by the algorithm

6. Allow players to see what boards they have started but haven't completed.

7. Allow players to select a board number.

    a. If the board number does not exist, the player is alerted that the board number does not exist.

8. When the launch button is clicked, launch the game with that BoardNumber.

GAME

9. Allow users to save their boards mid-game.
    a. When the menu button is clicked, save the board before quitting the game
    b. When the program is quit, save the board before quitting.
    - If the user made any progress on the board, this progress will be saved.
    - When that user launches the same board number again, his previously saved changes will be seen. This allows players to save boards at any state.

10. Generate an interactive Board and fill the starting values (in black font) based on the board the user picked in the user menu.

11. Allow users to input values onto the board.
    a. Value is entered by clicking on a cell and clicking a number from 1-9 on the keyboard.
    b. Board will be updated with the new user value (in blue font unless it produces a clash).

12. Allow users to delete USER ENTERED numbers on the board by pressing 0, or the number that already exists in the cell, on the keyboard.

13. Create a hint system for sudoku
    a. Display the correct value for a cell when a cell is clicked on and the letter H/h is clicked on the keyboard.
    b. Only allow for 3 hints in a game.

14. Create a Solve button.
    a. Allow the user to click on a solve button to display the correctly completed board.
    b. This will not be reversible to prevent cheating.

15. Create a system to check for validity using the sudoku game rules
    a. When a number entered by the user is not valid, alert the user by highlighting the user value and the value it clashes with in red.
    b. Highlighting will go away when the clash is resolved.

16. When the GUI board matches the answer for that board, the user has correctly completed the board and is prompted with the text “Correct!”

17. Save the time taken to solve the board after the user has correctly completed the board.

Backend objectives:

18. Store usernames and passwords on a database using hashing on the passwords for security.

19. Create a strategic brute force algorithm called Backtracking to solve sudoku:
    - Find all the x and y values where the board is empty
    - If no spots are empty, the board is solved
    - Try every value from 1 to 9 and if a number is valid edit the board and save the valid value.
    - check if the board is solved by recursively calling the solve board function with the new board as a parameter.
    - Move onto the next empty cell.
    - Remove the algorithm-entered values that fail to satisfy the board validity rules and try again.
    - Repeat until the board is completely solved.

1.9 Potential solutions:

Language:

My 2 main choices are Java and Python. They are the 2 most popular languages and are both free to use. Java is more suited to app and game development than Python. Java is generally much faster and more efficient than Python as it is a compiled language and only needs to be run once to create an executable file. Python is interpreted meaning that it must execute the code every time it is to be run. Although languages like Java are more suited for app development, learning these languages and finishing my NEA in the time constraint of 2 months will prove difficult. I would need to learn libraries such as Applets or JavaFX to create GUIs in Java.

The main advantage of Python is its simplicity, which will let me develop my program faster compared to other languages. Python is portable as it is an interpreted language meaning it can be run on different platforms. Python is an object-oriented language that which It easy for me to reuse my code for elements such as buttons or textboxes in my GUI and login system and allows for easier troubleshooting. I would need libraries such as Tkinter or Pygame to create GUIs in Python. As Python is easy to read, I could easily seek advice from the internet or peers as I can understand their code faster.

After substantial research, I have concluded that I will be using Python for my project as this is the language, I have been taught programming in Python for the past 4 years, and it is the language I’m most familiar with. I have also used Pygame before meaning I will not have to spend time learning a new GUI library. I have done most of my self-learning in Python. Though Java runs programs faster, my program isn't that big and will run in Python just as well.

Platform and developer environments:

The purpose of an Integrated Development Environment is to make coding faster and easier by having all the tools you would require to code in one place. I will be using the PyCharm IDE because I have used it for the past 2 years during my A levels. PyCharm comes with a range of developer tools that will speed up and aid me in programming. I can use the debugging tool to find out if and where my program has bugs. PyCharm allows me to use different syntax highlighting and themes, making it more suited to me and personalized. IDEs can help with syntax making it easier for me to focus on the logic of the code rather than on syntax.

GUI comparisons:

My program requires a GUI library to create an interactive and visually appealing board. There are many libraries I could choose from, each with strengths and weaknesses. I will have to choose between Tkinter, Pygame, and Kivy

Tkinter:

Tkinter is a free and open-source Python library for developing GUI. Tkinter comes as a built-in library in Python and does not need to be downloaded. It is mainly used for developing desktop apps as it does not provide features for developing mobile apps. Tkinter can be run on most operating systems such as Windows, macOS, and Linux. Tkinter is older than Kivy and the GUIs it produces can look outdated. It is mainly used by beginner developers as it is user-friendly.

Kivy:

Kivy is a free and open-source Python library for developing full-fledged multitouch apps. Kivy needs to be installed and does not come with Python. It can be run on computers and smartphones and raspberry pi. Kivy is much more complicated than Tkinter and is widely used to develop dynamic and advanced apps using Python.

Pygame:

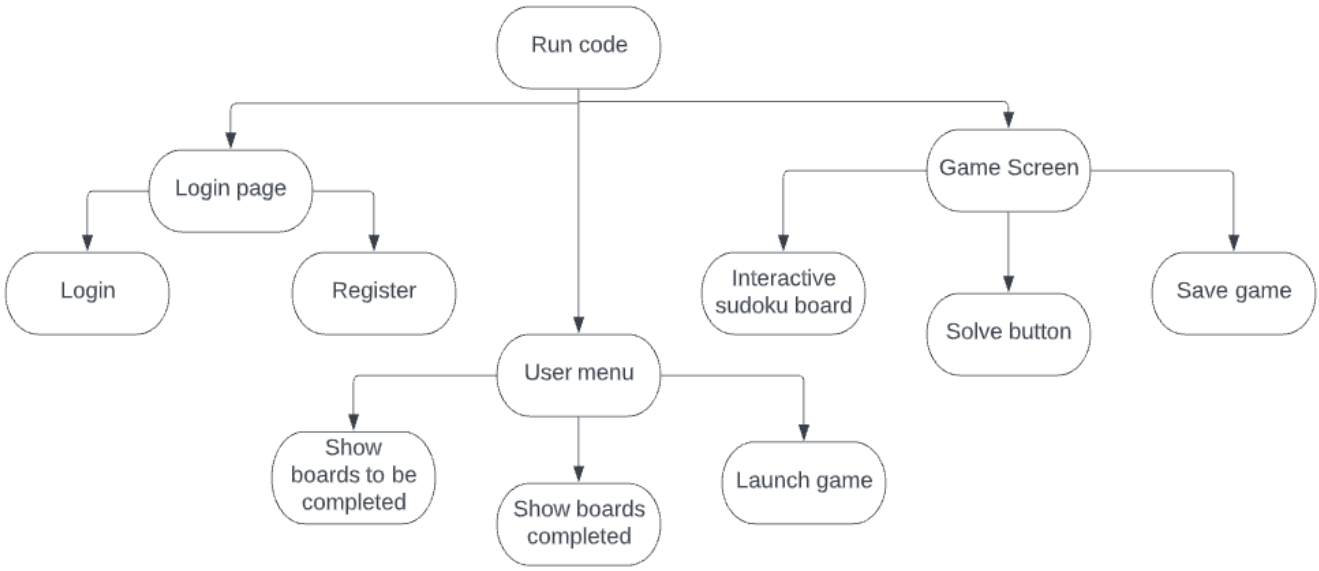
Pygame is also a free and open-source Python library mainly for building games. It can also be used to make game GUIs. Pygame does not come inbuilt with Python and needs to be installed. It can be run on macOS, Windows, and Linux. Pygame is a bit more complicated than Tkinter as it does not come with GUI elements. I will have to create elements such as buttons and text boxes.

Conclusion:

I have chosen to use Pygame. Even though Pygame is not meant for creating GUIs, it is the library I'm most familiar with. I have previously used Pygame to create GUIs and games so I would save time by not having to learn another GUI library. Pygame will allow me to easily have many screens for the login, user menu, and game. Although I will have to create classes for buttons and text boxes, I will only have to create them once using an element class, inheritance, and polymorphism and never worry about them again.

Documented Design

2.1 Top-Down diagram:



Overall system design

Login page:

This is the page launched when the Python file is run. To access the other pages, a user will need to log in. This page also gives a new user the option to create an account. If a user does not remember the password, he is encouraged to contact me to get his password changed as there is no email/2F authentication.

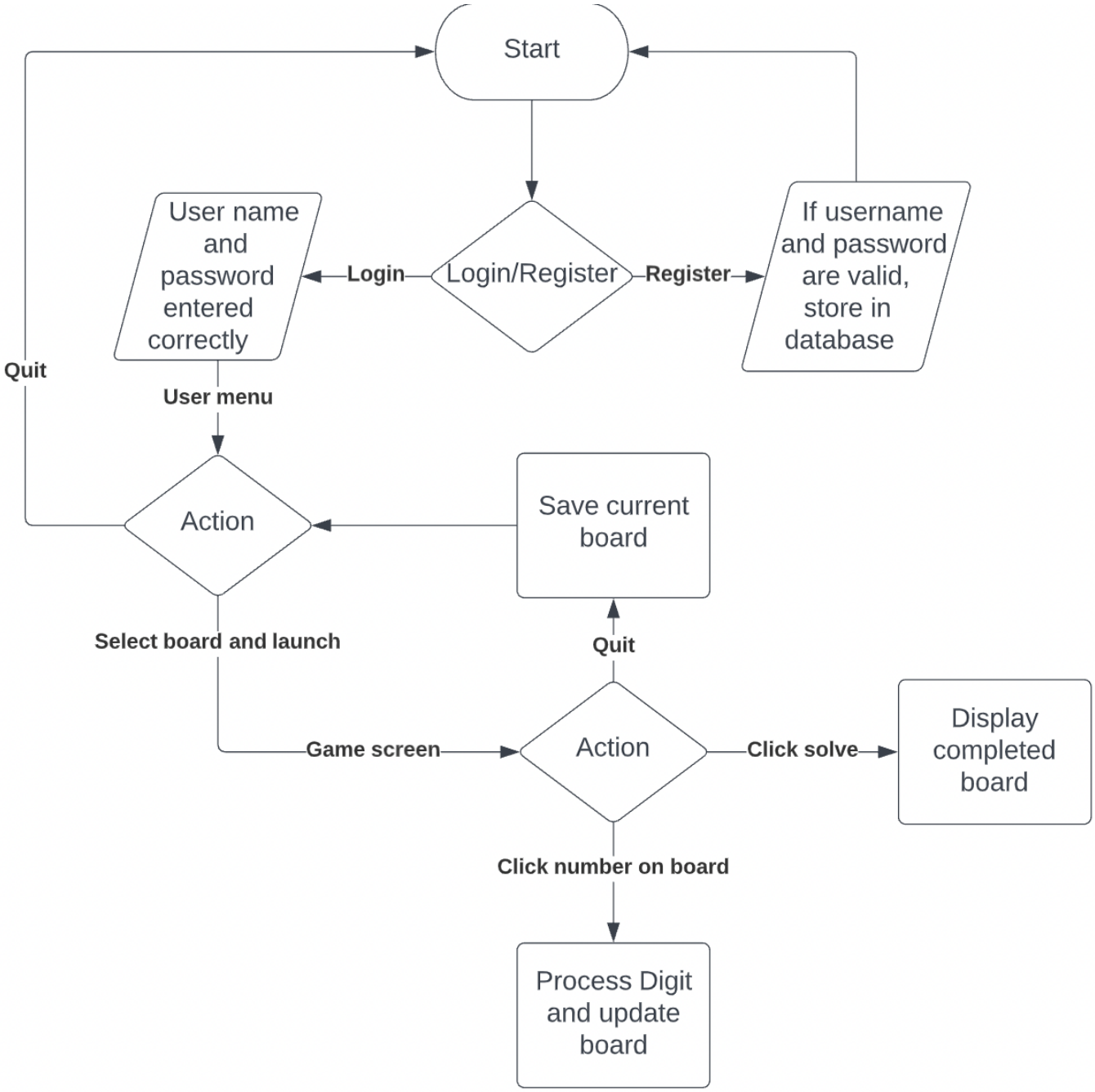
User menu:

This page acts as the user profile and is only accessible through a successful login. The contents of this page are different for every member. Statistics such as boards completed and incomplete boards can be seen. This page is the only way to enter the game screen by writing a board number and clicking a launch button. There is also an option to log out which will bring the user back to the login page.

Game screen:

This page is the main function of my program. It is only accessible once the user launches a board number that exists. Here the user will be able to play sudoku with all the end user's desired features. The timer is started once this screen is launched. There is also a solve button that can be used if a user is stuck. The board will be saved if the program is quit, or the user menu button is clicked.

2.2 Flow chart

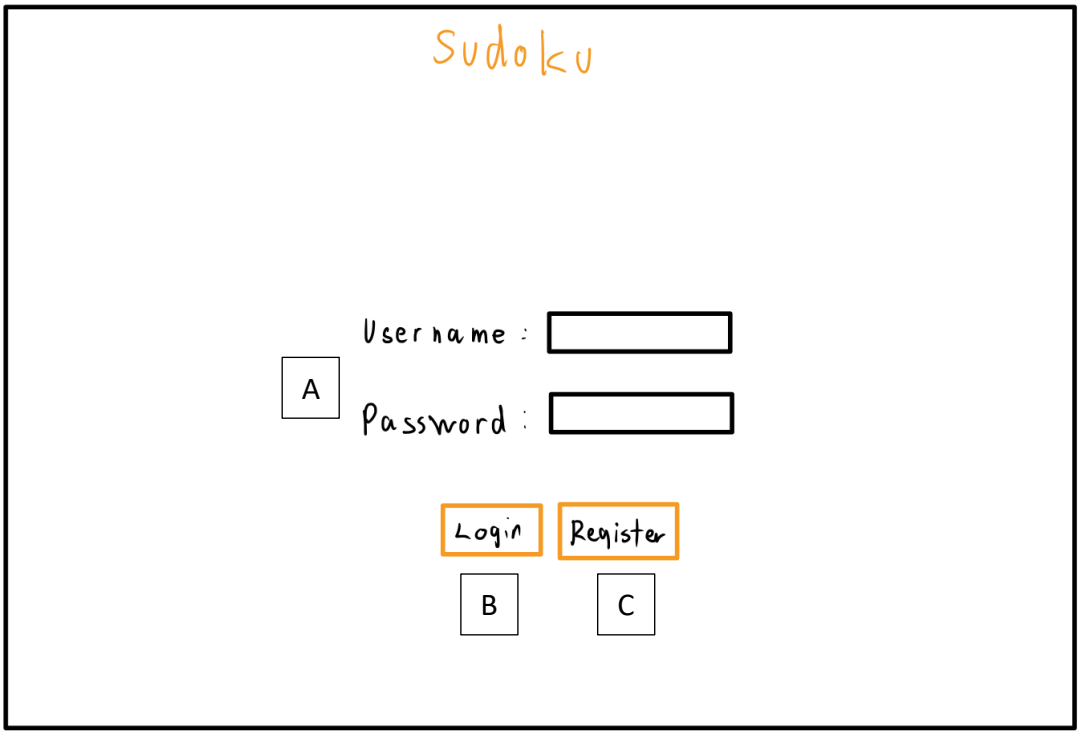


2.3 IPSO:

Input	Login	Username Password Login button
	Register	Username Password Register button
	User Menu	Board Number to launch Launch button Log out button
	Game Screen	Numbers to display in board Solve button User menu button
Processing	Login	Check if fields are empty Check if password is correct
	Register	Check if fields are empty Check if username already exists Check if password is complex enough
	User Menu	Find completed boards on database Find incomplete boards on database Launch board entered
	Game screen	Check if character is valid Check if the number entered produces clashes Check if the game is completed If solve clicked: Display solved board
Storage	Register	Save username on database Save hashed password on database
	Game screen	Save current board if program quit or user menu button clicked Save time and state of board (complete/incomplete) when the user quits the board so that it can be displayed in the user menu
Output	Login	Username not found: Username does not exist Password incorrect: Password incorrect
	Register	Username not unique on database: Username already exists Password not complex: Password is not strong
	User Menu	Display number of boards available Display board numbers of boards that have been: - Completed by user - Completed by solving algorithm - Incomplete
	Game Screen	Output original board in black Output any user-entered values in blue Show clashes in shades of red Output 'Success' after the game is completed without clicking the solve button Output a tutorial on how to use hints and delete numbers

2.4 GUI explanation

These are my initial UI sketches. They will allow me to refer to a blueprint instead of spending time designing the UI while programming the logic. I will try and create my GUI to look very similar to this.



Login screen

A. These are text boxes. Users can enter a username and password and either log in or register.

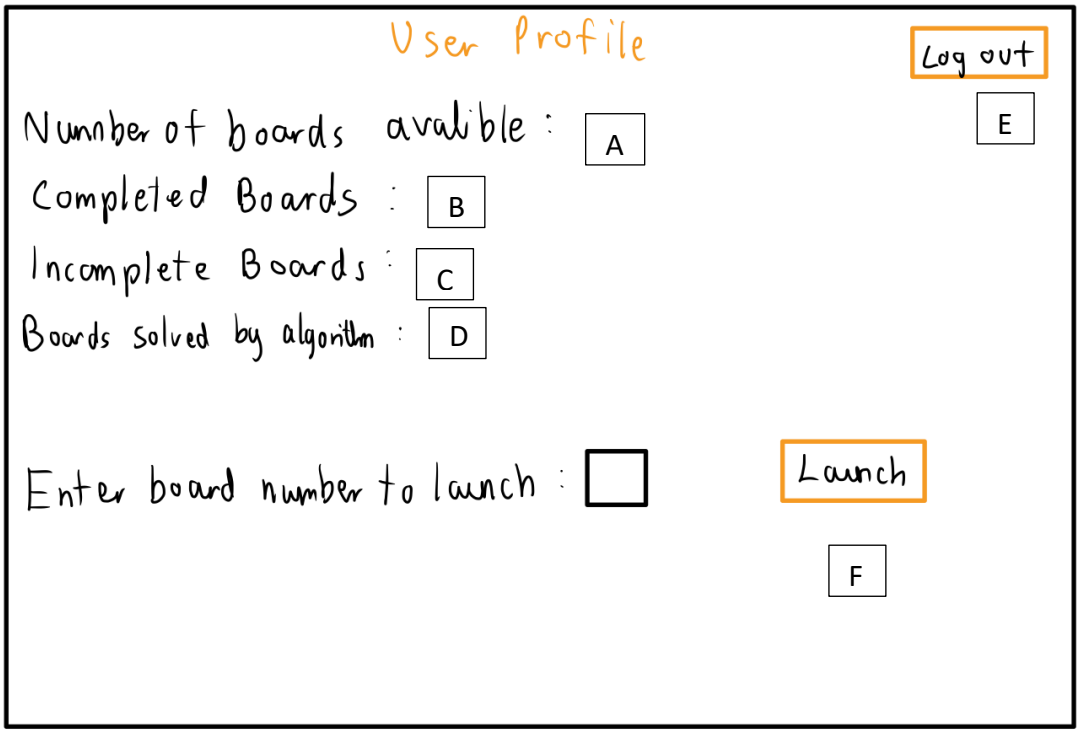
B. If login is clicked, the password they enter is hashed and compared to the hashed password stored on the database to that username.

- If the username and password are correct, the user is taken to the user menu screen
- If the username and password are incorrect, 'Incorrect password' is displayed

C. If register is clicked, the username and the hashed version of the password are stored in the database.

- If the username already exists, 'user already exists' is displayed. I will use the username as a primary key in the database, so I need each username to be unique
- If the username is unique and the password is complex enough, the username and password are saved on the database

User Profile



A. This is the number of boards that can be played now. Akshay and I can add and remove boards easily.

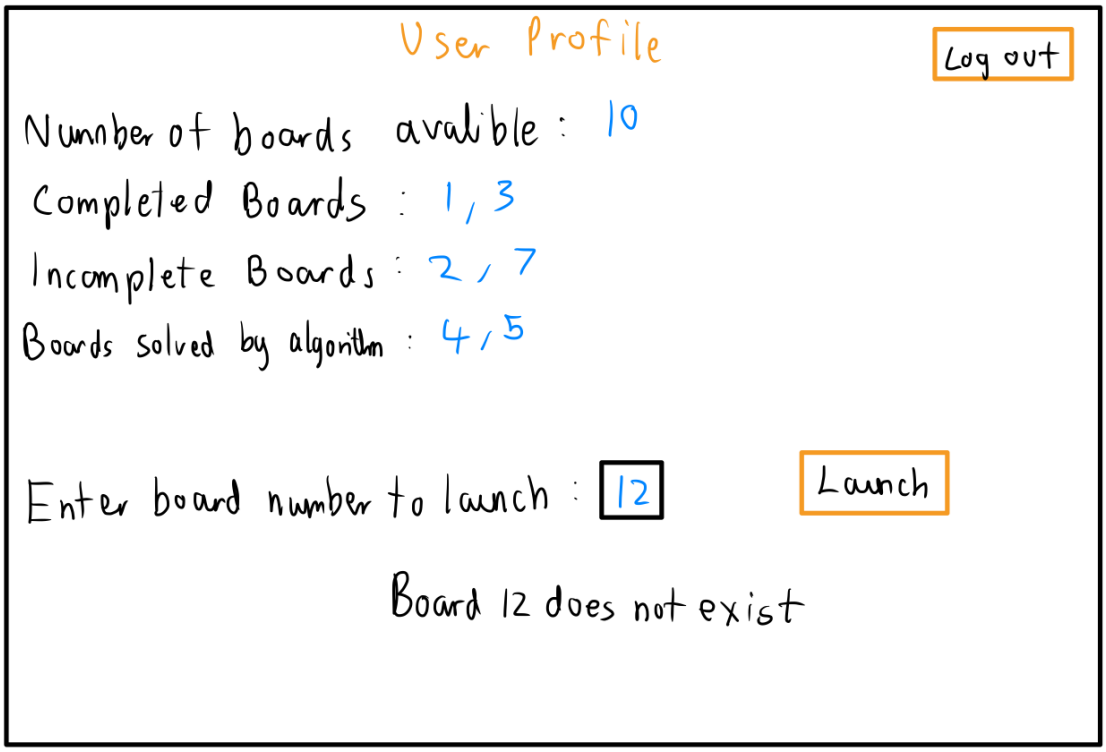
B. These are the boards that have been completed WITHOUT using the solve button.

C. These are the boards that have been started but have not been completed.

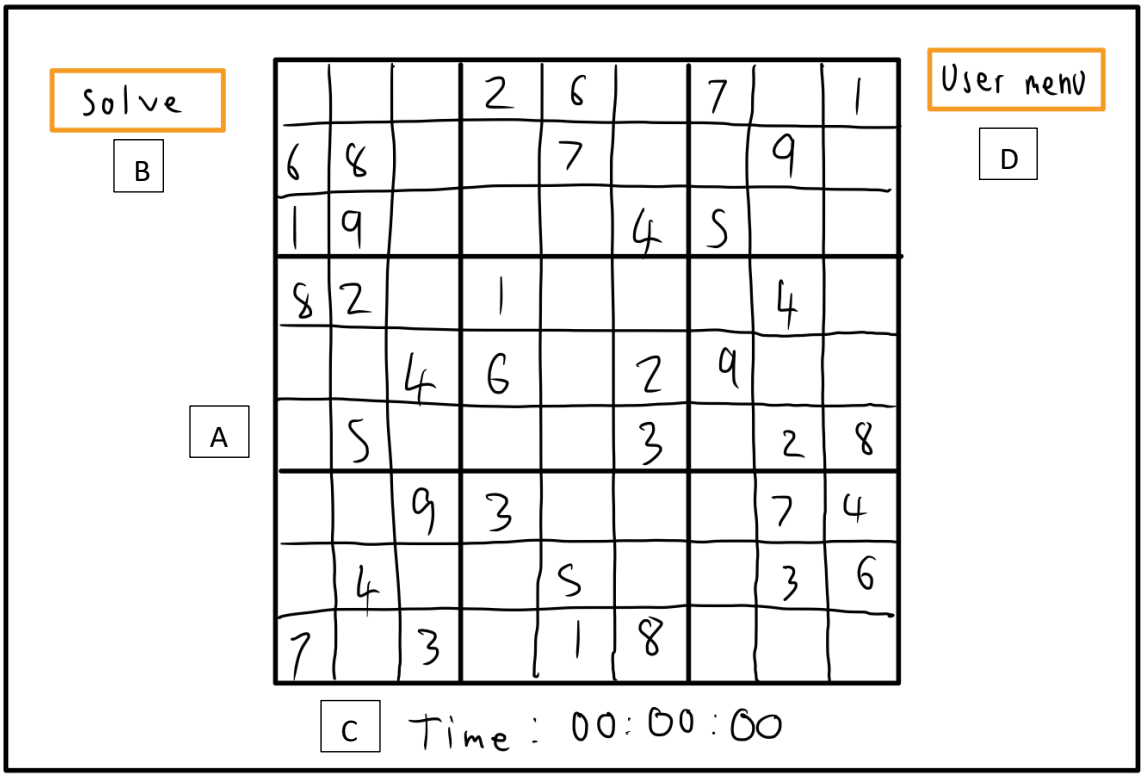
D. These are the boards that have been completed using the solve button

E. Users can log out by clicking this button. They will need to log back in to access a profile again.

F. Users can enter a board number and click launch to Launch the game. If the board number does not exist, 'Board number (number) does not exist.' Is displayed
    a. If the board number has not been launched before, the original board will be launched
    b. If the board number has been launched before, the saved board from when the user launched this board before, is launched.


User Profile
Log out
Number of boards avalible : 10
Completed Boards : 1, 3
Incomplete Boards : 2, 7
Boards solved by algorithm : 4, 5
Enter board number to launch : 12
Launch
Board 12 does not exist

Game screen

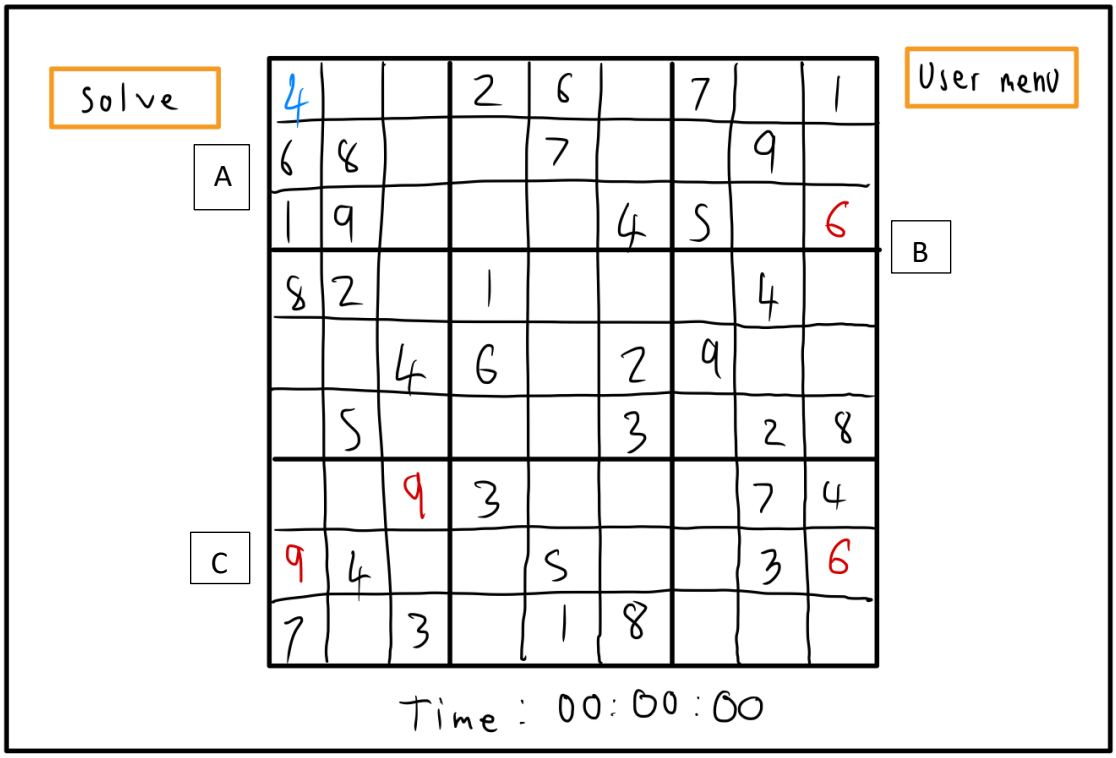


A. This is the interactive sudoku GUI.
   a. If a number between 1 and 9 is clicked in an empty cell, the number is displayed.
      i. If the number does not produce any clashes with other numbers, it is displayed in blue
      ii. If a clash is produced, all the numbers in the clash will be in red
   b. 0 or the key for the number in the cell can be clicked to clear the cell. (If 1 is in a cell and 1 is clicked again, the cell is cleared)
   c. H can be clicked to give a hint (the correct number for that cell). Only 3 hints will be allowed per game.

B. When the solve button is clicked, the completed and correct board is displayed. This board will not count towards the 'completed boards' section in the User Profile.

C. This is the time taken for the board to be completed. It is saved if the user quits the game without completing the board and will resume when the user returns to this board.

D. When this button is clicked, the user will be taken back to their profile and the board will be saved to the database.
   a. If the board is incomplete, the board number they played will be displayed as an incomplete board.
   b. If the board is completed using the solve button, the board number will not be displayed as a completed board.
   c. If the board is completed without using the solve button, the board number will be displayed as a completed board.



A. The number 4 is clicked, and as it does not clash with any other number, its font Is blue.

B. The number 6 is clicked, and as it clashes with another 6 in the same column, its font is red.

C. The number 9 is clicked, and as it clashes with another 9 in the same 3x3 square, its font is red.

2.5 Normalisation

To make my database efficient, I will need to normalize it. I started by making a table of all the data I would need. However, there are many bad practices in this table such as storing multiple values in one cell or having many non-key dependencies such as TimeTaken which in no way, relates to the username.

users:

username	Password	BoardsCompleted	Completed	TimeTaken	Difficulty	SavedBoard
Aditya	ff41ffc558dccf6f14 37d580... (hash)	003020600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300, 003020600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300	True, False	430, NULL	'Easy', 'Easy'	483921657 967345821 251876493 548132976 729564138 136798245 372689514 814253769 695417382 , 123420600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300

1NF

username	Password	BoardsCompleted	Completed	TimeTaken	Difficulty	SavedBoard
Aditya	ff41ffc558dccf6f14 37d580... (hash)	003020600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300	True	430	'Easy'	483921657 967345821 251876493 548132976 729564138 136798245 372689514 814253769 695417382
Aditya	ff41ffc558dccf6f14 37d580... (hash)	003020600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300	False	120	'Easy'	123420600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300

I will need a primary key so that each row can be uniquely identified to take this to its first normal form. I will use a GameID as a primary key.

GameID	username	Password	BoardsCompleted	Completed	TimeTaken	Difficulty	SavedBoard
1	Aditya	ff41ffc558dccf6f1437d580… (hash)	003020600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300	True	430	‘Easy’	483921657 967345821 251876493 548132976 729564138 136798245 372689514 814253769 695417382
2	Aditya	ff41ffc558dccf6f1437d580… (hash)	003020600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300	False	120	‘Easy’	123420600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300

Now each game can be uniquely identified, and the table is in its first normal form.

2NF

To take this board to its second normal form I will need any non-primary-key attribute to be dependent on nothing but the primary key. This means that any column in the table that does not relate to the table's primary key must be removed or put in another table.

I can split the board into 3 separate boards: users, boards, and a link table that will record all the games that have been played.

Users:

username	Password
Aditya	ff41ffc558dccf6f1437d580… (hash)

Passwords only depend on the username so this will be a separate table

defaultBoards:

BoardNumber	Boards	Difficulty
1	003020600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300	‘Easy’

2	003020600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300	'Easy'

This is all the information about each board. I have added a BoardNumber primary key so that the board and all its attributes will be easy to refer to.

I will create a link table between the users and the default boards which will allow me to store data about each game such as time taken and if it's completed and allow me to save boards for each game played. This also allows admins to easily see the progress of each player.

I will not need a game ID primary key as it will be redundant and instead, I will use a composite key between the foreign keys: username and BoardNumber. Together, they will be unique in every row.

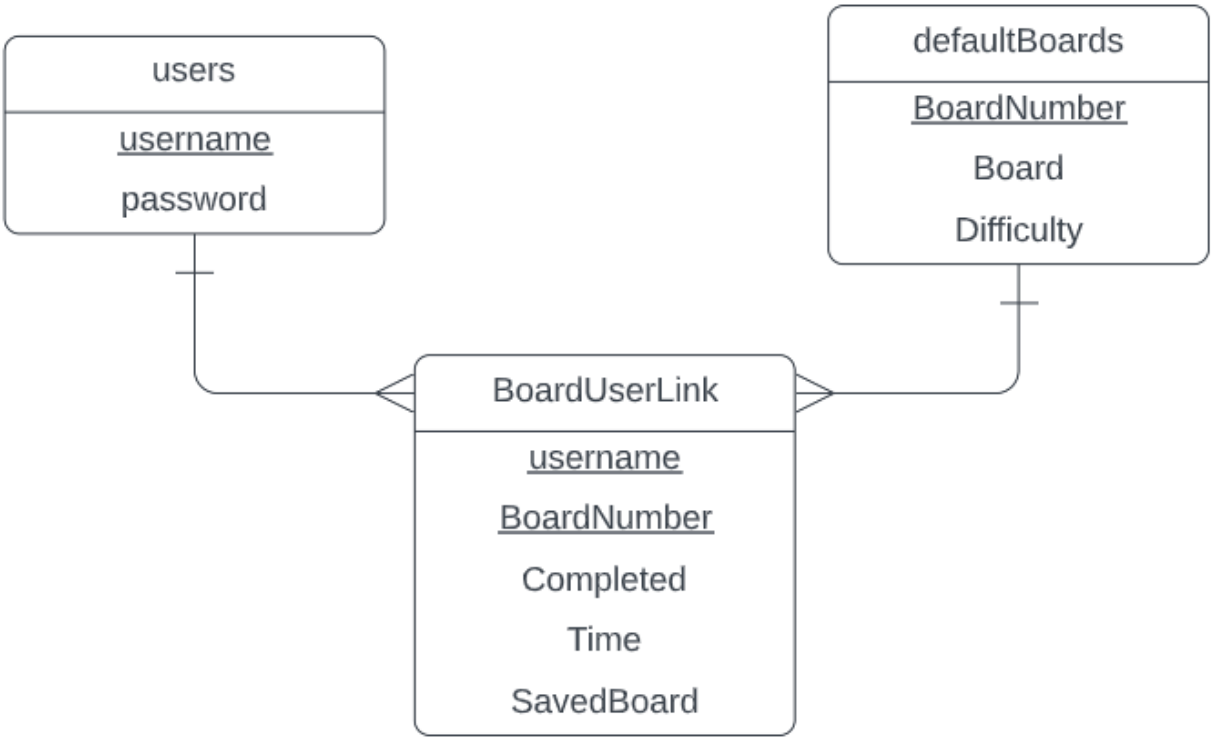
BoardUserLink:

username	BoardNumber	Completed	Time	SavedBoard
Aditya	1	True	430	483921657 967345821 251876493 548132976 729564138 136798245 372689514 814253769 695417382
Aditya	2	False	120	123420600 900305001 001806400 008102900 700000008 006708200 002609500 800203009 005010300

3NF

This database does not have transitive dependencies and thus, is already in third normal form.

2.6 ERD



2.7 Data dictionaries and validation

With the database being normalized, I need to create data dictionaries to help me visualize how data will be stored in the database. I will add the data type, an example, and the validation for the field.

Users:

Field	Type	Validation	Example
username	TEXT Primary Key	NOT NULL	'Aditya'
password	TEXT	NOT NULL	'ff41ffc558dccf6f1437d580…'

defaultBoards:

Field	Type	Validation	Example
BoardNumber	INTEGER Primary Key	NOT NULL	1
Board	TEXT	NOT NULL	'043080250 600000000 000001094 900004070 000608000 010200003 820500000 000000005 034090710'
Difficulty	TEXT		'Easy'

BoardUserLink:

Field	Type	Validation	Example
username	TEXT Foreign key	NOT NULL	‘Aditya’
BoardNumber	INTEGER Foreign key	NOT NULL	1
Completed	TEXT	NOT NULL	‘False’
Time	INTEGER (seconds)		430
SavedBoard	TEXT		‘043080250 612345678 000001094 900004070 000608000 010200003 820500000 000000005 034090710’

2.8 SQL STATEMENTS

This is the code for my initial ideas for the tables

Users table:

```
CREATE TABLE "users" (
	"username"	TEXT NOT NULL,
	"password"	TEXT NOT NULL,
	PRIMARY KEY("username"))
```

DefaultBoards table:

```
CREATE TABLE "defaultBoards" (
	"BoardNumber"	INTEGER NOT NULL,
	"Board"	TEXT NOT NULL,
	"Difficulty"	TEXT,
	PRIMARY KEY("BoardNumber"))
```

BoardUserLink table:

```
CREATE TABLE "BoardUserLink" (
	"username"	TEXT NOT NULL,
	"BoardNumber"	INTEGER NOT NULL,
	"Completed"	TEXT NOT NULL,
	"Time" INTEGER,
	"SavedBoard" TEXT,
	PRIMARY KEY("username", "BoardNumber"),
	FOREIGN KEY("username") REFERENCES "users"("username"),
	FOREIGN KEY("BoardNumber") REFERENCES "defaultBoards"("BoardNumber"))
```

Register system

SELECT username FROM users *Selecting username to check if it exists in the database*

If username and password hit all criteria:

INSERT INTO users(username, password) VALUES ('{u}','{hash(p)}')
inserting the new account into the database. Dbj2 is the hashing algorithm which I will explain later.

Login system

SELECT username FROM users *Selecting username to check if it exists in the database.*

If the username exists:

SELECT password FROM users WHERE username = '{u}' *Selecting password for that username.*
If the username and hashed password match the database data, the login is successful.

User Menu:

SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'True' *Selecting all the boards that have been completed without cheating.*

SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'False' *Selecting all the boards that have not been completed.*

SELECT COUNT(*) FROM defaultBoards *Selecting the number of available boards.*

INSERT INTO "main"."BoardUserLink" ("username", "BoardNumber", "Completed", "Time") VALUES ("{u}", {BoardNumber}, "False", 0) *Inserting the new game into the link table.*

Game Screen:

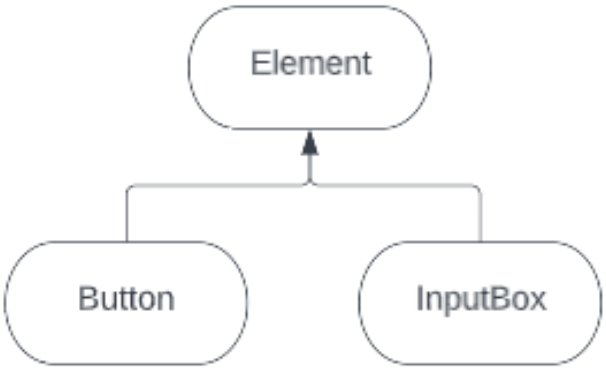
SELECT Time from 'main'.'BoardUserLink' WHERE username = '{u}' AND BoardNumber = {BoardNumber}
Selecting the time already played on this board. It is 0 by default

SELECT Completed from 'main'.'BoardUserLink' WHERE username = '{u}' AND BoardNumber = {BoardNumber} *Selecting Completed to see if the board has already been completed.*

UPDATE 'main'.'BoardUserLink' SET (Completed, Time) = ('True', {timeInSec}) WHERE username = '{u}' and BoardNumber = {BoardNumber} *If board is completed, update the table with Completed being true and the Time taken.*

UPDATE "main"."BoardUserLink" SET (SavedBoard, Time) = ("{boardString}", {timeInSec}) WHERE username = "{u}" AND BoardNumber = {BoardNumber} *Saving the board if the user quits mid game.*

2.9 OOP Design



Element class:

The main purpose of this class is to act as a blueprint for the other elements I will implement such as buttons and text boxes. Most of the parameters needed will be similar for both the buttons and the InputBox class.

Element	
Parameters	x y w h text = ‘’
Functions	Draw(screen) handle_event(event)

The element class takes in 4 parameters: the x coordinate, the y coordinate, the width of the element, and the height of the element. The class also has an optional parameter ‘text’ which is set to an empty string by default indicating that it is optional.

The function draw takes in the parameter ‘screen’ which is the display I will be drawing the element on. I have kept this function empty as I know both the child classes will override this function.

Button class:

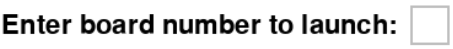
Button INHEREITS Element	
Parameters	super()
Functions	Draw(screen) OVERRIDE handle_event(event) INHERIT

The Button class inherits the element class, so it has the same parameters as the element class. The function handle_event has been inherited from the element class while the draw function has been overridden to match the requirements of a button

InputBox class:

InputBox INHEREITS Element	
Parameters	super() textType = 'text'
Functions	Draw(screen) OVERRIDE handle_event(event) OVERRIDE update(screen)

The InputBox inherits the elements class, so it has inherited all the parameters of elements. This class requires another parameter 'textType' to choose between text, password and int mode. In password mode, all the characters are replaced with '*' for privacy. In int mode, only integers 1-9 are allowed to be typed. The class inherits all the functions from elements and overrides them. An input box requires a prompt so that the user knows what to type in the Input box. Hence, the draw function has been overridden so that it allows for a prompt to be drawn to the left of the input box.



The InputBox class also requires another function called update which updates the size of the textbox if the text in it is too long.

2.10 Data security:

To ensure maximum security, the database will only be accessible to me and the group leader: Akshay. My program does not require many security features apart from hashed passwords so that even if the database is hacked, the hacker will not be able to trace back the hash value to a password.

The hash I plan on using is called SHA-256. This hash function is very secure and is not broken (A hash is considered broken when 2 values generate the same hash value/collide). When a user registers, their password is hashed and saved in the database. The original password will not be saved anywhere meaning that only the user knows the original password. When a user wants to log in, the password they enter to log in will also be hashed and if it matches the hashed password stored on the database, the login is successful.

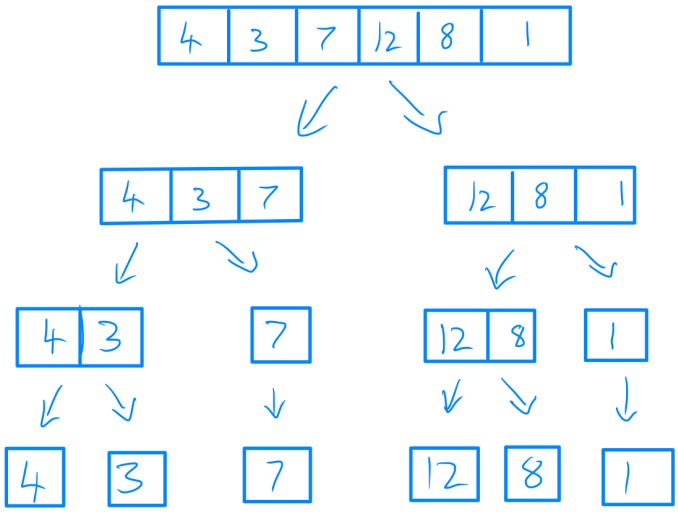
2.11 ALGORITHMS

Merge sort:

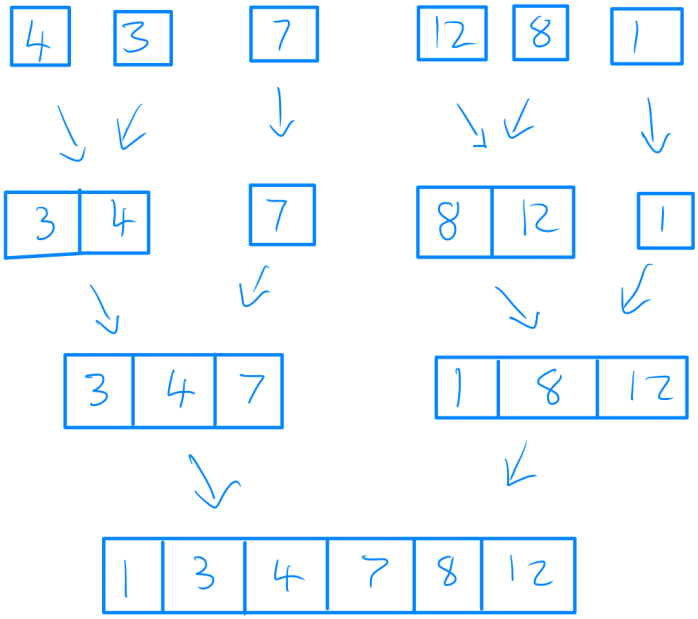
Merge sort is a 'divide and conquer' algorithm which works by recursively breaking down a list until each list contains only 1 element. A list is joined to another list and sorted. These lists keep combining until there is only 1 sorted list remaining.

Steps:

- IF there is only 1 element in the list, it is already sorted. Return.
- If there is more than 1 element in the list, divide the list into 2 recursively until it cannot be divided any further as shown below.



- Merge the smaller lists into a larger sorted list.



Pseudocode:

```
SUBROUTINE merge(arr):
    IF len(arr) > 1 THEN
        lefta ← arr[:LEN(arr) DIV 2]
        righta ← arr[LEN(arr) DIV 2:]
        merge(lefta)
        merge(righta)

        i ← 0 #i acts as a pointer to keep track of the values in the left array
        j ← 0 #j acts as a pointer to keep track of the values in the right array
        k ← 0 #k acts as a pointer in the merged array

        WHILE i < LEN(lefta) AND j < LEN(righta) #while i is lesser than the length of the left array and j
                                                 #is less than the length of the right array

            IF lefta[i] < righta[j] THEN #If value of left array at i is less than the value of right array at j
                arr[k] ← lefta[i] #set array position k to the element at pointer of left array
                i ← i + 1 #increment i by 1

            ELSE
                arr[k] ← righta[j] #if value of left array at I is greater than or equal to the value of
                                   right array at j

                j ← j + 1 #Increment j by 1

             ENDIF
            k ← k + 1 #Increment k by

        ENDWHILE

        WHILE i < LEN(lefta) #while i is less than the length of the left array

            arr[k] ← lefta[i] #set element at position k in array to the element at i in left array

            i ← i + 1 #increment right pointer by 1

            k ← k + 1 #increment k by 1

        ENDWHILE

        WHILE j < LEN(righta) #while j is less than the length of the right array

            arr[k] ← righta[j] #set element at position k in array to the element at j in right array

            j ← j + 1 #increment right pointer by 1

            k ← k + 1 #increment k by 1
         ENDWHILE
         RETURN arr

    ELSE
        RETURN arr # list is already sorted as it only has 1 item in it

    ENDIF
ENDSUBROUTINE
```

Hashing

I am using the djb2 algorithm as it is simple but very efficient and does not produce many collisions, especially when using a random salt along with the password

```
SUBROUTINE hash(password):
    a ← 5381
    # This starting prime number was picked as it showed the least collisions
    FOR x IN password  # for every character in the password
        a ← ((a << 5) + a) + CHAR_TO_CODE(x)
        # a is set to a left shift 5 bits + old a + ASCII value for that character
    ENDFOR
    RETURN hex(a AND 0xFFFFFFFF)  # return the hexadecimal value of a AND -1
ENDSUBROUTINE
```

find_0

Find_0 is a function that finds the next empty cell in a sudoku board.

```
SUBROUTINE find_0(board):
    FOR i ← 1 TO 9 # repeat 9 times
        FOR j ← 1 TO 9 # repeat 9 times
            IF board[i][j] = 0 THEN # if board position at I and j is 0
                RETURN i, j # return the board position as it is empty
            ENDIF
        ENDFOR
    ENDFOR

    RETURN NONE  # if no board position is returned, the board is solved.
ENDSUBROUTINE
```

This function is also used in the backtracking sudoku solver

Check Valid

Sudoku rules:

- Cannot have the same number twice in a row
- Cannot have the same number twice in a column
- Cannot have the same number twice in a 3x3 grid

```
SUBROUTINE check_valid(board, number, position)
# check if a number in a cell is valid based on sudoku rules

    # checks to see if the number is repeated again on the same row
    FOR i ← 1 TO 9 # for every cell in the same row
        IF bo[position[0][i] = number AND NOT(position[1] = i) THEN
        # if the cell is equal to the input number and the position of the cell is not the same as the position of
        # the number to input, return false

            RETURN FALSE
        ENDIF
    ENDFOR

    #checks to see if the number is repeated again in the same column
    FOR j ← 1 TO 9 # for every cell in the same column
        IF bo[j][position[1]] = number AND NOT(position[0] = j) THEN
        # if the cell is equal to the number to input and the position of the cell is not the same as the position
        # of the number to input, return False

            RETURN FALSE
        ENDIF
    ENDFOR

    box_x ← position[1] DIV 3 # box_x is the x coordinate of the 3x3 grid the number to input is in
    box_y ← position[0] DIV 3 # box_y is the y coordinate of the 3x3 grid the number to input is in

    # Checks the 3x3 square
    FOR i ← box_y * 3 TO box_y * 3 + 3 # for every column in the 3x3 grid
        FOR i ← box_x * 3 TO box_x * 3 + 3 # for every row in the 3x3 grid:

            IF board[i][j] = number AND NOT([i,j] = position) THEN
            # if the cell is equal to the number to input and the position of the cell is not the same as the
            # position of the number to input, return False

                RETURN FALSE

            ENDIF
        ENDFOR
    ENDFOR

    RETURN TRUE # if no rules are broken, return True (valid)

ENDSUBROUTINE
```

Backtracking

```
SUBROUTINE BT(board):
    find ← find_0(board) # find is a coordinate if an empty cell is found
    IF find = None THEN # if no empty cell is found
        RETURN board # return board as it is already solved
    ELSE #if an empty spot is found
        row ← find[0] # row is i
        col ← find[1] # column is j
    ENDIF

    FOR i ← 1 TO 9 # for i in the range 1-9 which is every possible number that can be in a cell
        IF check_valid(board, i, (row, col)) THEN # if i is valid with sudoku rules at that empty cell
            board[row][col] ← i #insert i into the board at the position of the empty cell

            IF BT(bo) THEN # if the board is solved
                RETURN bo # return the board
            board[row][col] ← 0 # if it isn't solved, set the board at position of the previous empty cell back
                                # to 0 and try a different number
            ENDIF
        ENDIF
    ENDFOR

    RETURN FALSE # if no solution is found, return FALSE


ENDSUBROUTINE
```

Login

```
SUBROUTINE login(u, p): # login function which takes in username and password
    if LENGTH(u) = 0 OR LENGTH(p) = 0 THEN # if username or password is empty: prompt user to fill fields
        RETURN 'Please complete all fields'
    ENDIF

    UsersOnDb ⟵ SQL('SELECT username FROM users') # selects all the usernames from the table users
    exist ⟵ FALSE
    FOR tuple IN UsersOnDb # for every tuple in the list
        IF u IN tuple THEN # If the username is in the tuple, username exists
            exist ⟵ TRUE
            BREAK # if the username is found, quit the loop
        ENDIF
    ENDFOR

    IF exist = TRUE THEN # if username exists
        passOnDb ⟵ SQL("SELECT password from users WHERE username = (u)'" # Password for that
                                                                        # username is selected
        salt ⟵ SQL("SELECT salt FROM users WHERE username = '(u)" # salt for that username is selected

        IF hash(p + salt) = passOnDb THEN # If hashed password the user enters is the same as the hashed
                                          # password stored on the database, login successful
            RETURN TRUE # password is correct

        ELSE # if hashed pass is not equal to pass on the database
            RETURN 'Incorrect password'

        END IF

    ELSE # if the username does not exist on the database
        RETURN 'Username does not exist'

    ENDIF

ENDSUBROUTINE
```

Register

```
SUBROUTINE register(u, p): # Register function which takes in username and password
    reg ⟵ "^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!#%*?&]{6,20}$"
    # This is the regular expression a password must comply with to be valid
    IF LENGTH(u) = 0 OR LENGTH(p) = 0 THEN # f username and password is empty, user is prompted to fill all
                                           #  the fields
        RETURN 'Please complete all fields'
    ENDIF
    UsersOnDb ⟵ SQL('SELECT username FROM users') # all usernames are selected
    exist <- FALSE
    FOR tuple IN UsersOnDb # for every tuple in the list
        IF u IN tuple THEN # If the username is in the tuple, username exists
            exist ⟵ TRUE
            BREAK # if the username is found, quit the loop
        ENDIF
    ENDFOR
    IF exist = TRUE THEN # if the username exists, prompt user to change the username
        RETURN 'username already exists'
    ELSE # if the username does not already exist
        IF re.search(pattern, p) THEN # if password complies with regular expression
            salt ⟵ ''
            FOR x ⟵ 1 TO 16 # make a 16 character long salt
                char ⟵  CODE_TO_CHAR(random.randint(37, 129)) # pick a ASCII value from 37 and 129
                salt ⟵ salt + char # append the new character to the salt
            ENDFOR
            SQL(f"INSERT INTO users(username, password, salt) VALUES ('{u}','{hash(p + salt)}',
                 '{salt}')") # insert the new username, hashed pass, and salt into the database
            conn.commit() # commit changes to database
            RETURN 'Successfully registered'

        ELSE # if the password does not comply with regular expression
            RETURN 'Password is not complicated enough'
        ENDIF
    ENDIF
ENDSUBROUTINE
```

Solution

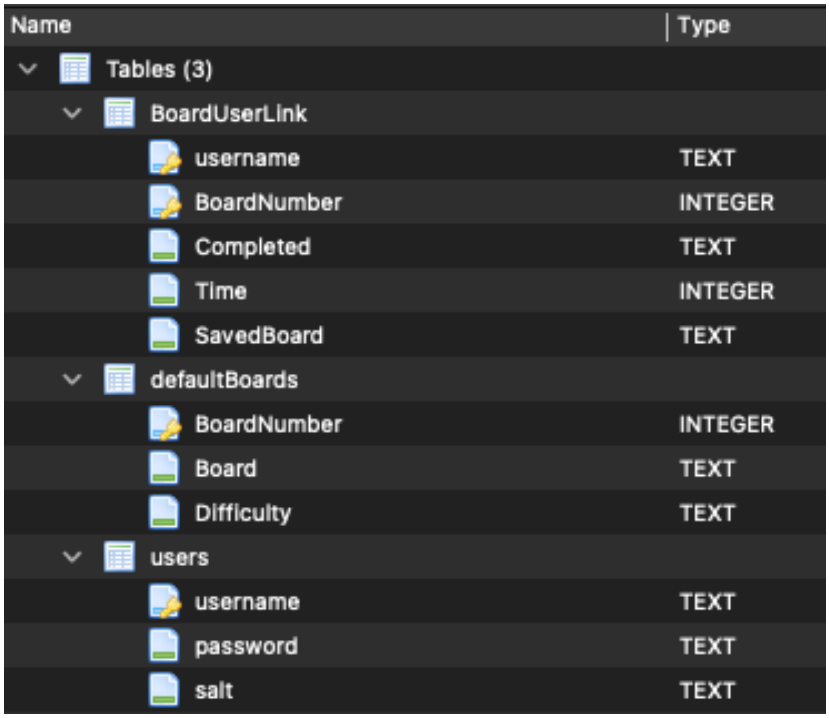
3.1 Evidence of setting up database and tables:

To create and access my database, I will be using SQLite3: a library that allows for communication between Python and databases. The program will have inbuilt create statements so that if a computer does not have the database, it will be created. When a new database is created, I will need to populate the defaultBoards table with the 25 boards by using a list of boards and a loop to insert each of them.

```
conn = sql.connect("data.db")
'''This creates the database if it does not already exist. If it exists, sqlite3 establishes a connection with the database.'''
db = conn.cursor()
db.execute(    USER GENERATED DATA DEFINITION STATEMENTS
    'CREATE TABLE IF NOT EXISTS "users" (\n'  # This command creates a users table if it does not already exist.
    '  "username" TEXT NOT NULL,\n'  # creates a username field that cannot be null
    '  "password" TEXT NOT NULL,\n'  # creates a password field that cannot be null
    '  "salt" TEXT NOT NULL,\n'  # creates a salt field where I will store the random salt of the password
    '  PRIMARY KEY("username")\n")')  # username is the primary key of this table

db.execute(
    'CREATE TABLE IF NOT EXISTS "defaultBoards" (\n'  # This command creates a defaultBoards table if it does not already exist.
    '  "BoardNumber" INTEGER NOT NULL,\n'  # BoardNumber is an integer that cannot be null
    '  "Board"    TEXT NOT NULL,\n'  # Creates Board field which cannot be null
    '  "Difficulty" TEXT,\n'  # creates difficulty field
    '  PRIMARY KEY("BoardNumber"))')  # BoardNumber is the primary key of this table

allBoards =
['041089000\n052003847\n080502039\n205307001\n807054003\n030900576\n600430950\n500076310\n413200700',...]
# a list of every board to store. There is too much to show in the word document as there are 25 boards. I have just shown one board

db.execute(
    'CREATE TABLE IF NOT EXISTS "BoardUserLink" (\n'  # This command creates a BoardUserLink table if it does not already exist.
    '  "username" TEXT NOT NULL,\n'  # creates a username field that cannot be null
    '  "BoardNumber"  INTEGER NOT NULL,\n'  # creates BoardNumber field which cannot be null
    '  "Completed"    TEXT NOT NULL,\n'  # creates Completed field which cannot be null
    '  "Time" INTEGER,\n'  # Time is an integer and will be stored in seconds
    '  "SavedBoard"   TEXT,\n'  # creates SavedBoard field.
    '  PRIMARY KEY("username","BoardNumber"),\n'  # username and BoardNumber make up a composite primary key
    '  FOREIGN KEY("username") REFERENCES "users"("username"),\n'
```

```
    # This makes username from  the users table a foreign key in the boardUserLink table.
    ' FOREIGN KEY("BoardNumber") REFERENCES "defaultBoards"("BoardNumber")\n'')')
    # This makes BoardNumber from the defaultBoards table a foreign key in the boardUserLink table.
count = db.execute("SELECT COUNT(*) FROM defaultBoards").fetchone()[0]  # check to see if defaultBoards is an empty table
if count == 0:  # if table is empty, fill table with boards
    for boardNo in range(1, len(allBoards)+1):  # for every BoardNo in the allBoards list:
        if 0 < boardNo < 6:  # first 5 boards are easy
            difficulty = 'Easy'
        elif 5 < boardNo < 11:  # next 5 boards are medium
            difficulty = 'Medium'
        elif 10 < boardNo < 16:
            difficulty = 'Hard' # next 5 boards are hard
        else:
            difficulty = 'Random'  # last 10 boards are of random difficulties

        db.execute(f"INSERT INTO defaultBoards ('BoardNumber', 'Board', 'Difficulty')
                        VALUES ({boardNo}, '{allBoards[boardNo-1]}', '{difficulty}')")
        # inserting each boardNumber, Board, and the Difficulty on the table
        conn.commit()  # commiting changes to the database
```

I can use software such as a database viewer to see if my commands have created a database. I will be using 'DB browser for SQLite'



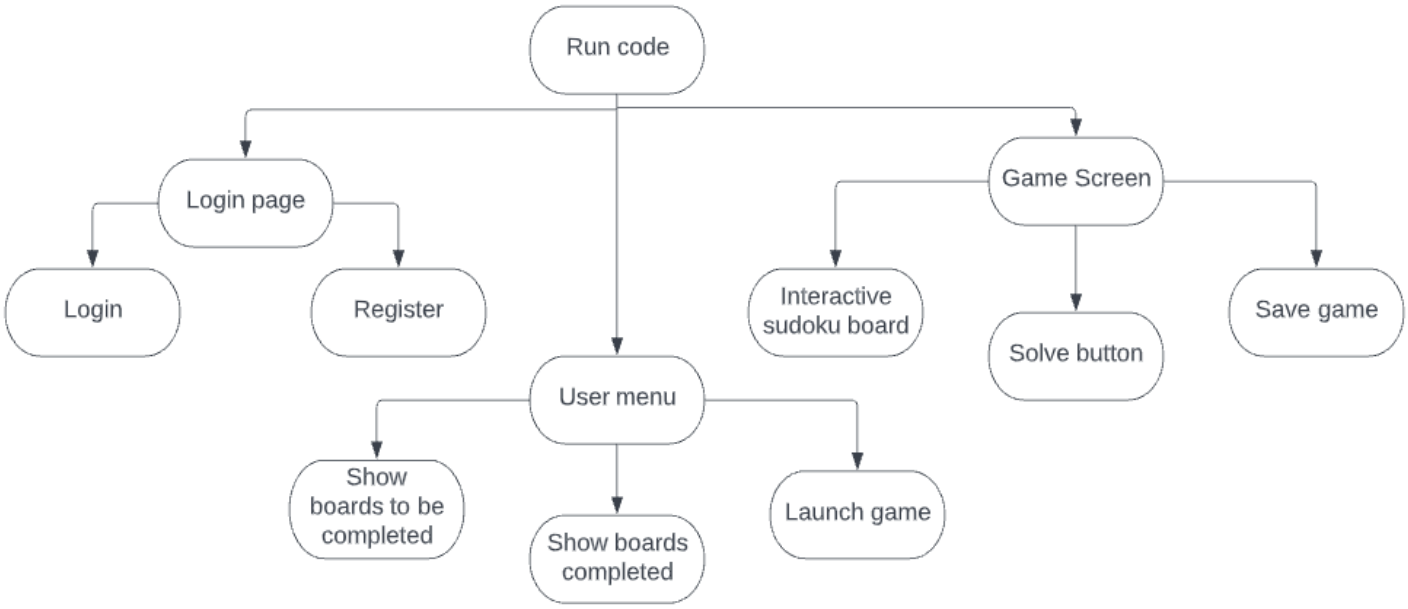
As we can see, the SQL commands have created the following tables.
The program has also filled in the 25 boards along with the BoardNumber, Board, and the Difficulty.

	BoardNumber	Board	Difficulty
	Filter	Filter	Filter
1	1	04108900...	Easy
2	2	08307090...	Easy
3	3	5090020...	Easy
4	4	10370002...	Easy
5	5	0080047...	Easy
6	6	7950480...	Medium
7	7	00601803...	Medium
8	8	79068120...	Medium
9	9	50760140...	Medium
10	10	02604070...	Medium
11	11	0580002...	Hard
12	12	8050000...	Hard
13	13	0043000...	Hard
14	14	3090000...	Hard
15	15	10800000...	Hard
16	16	0030206...	Random
17	17	0000009...	Random
18	18	2000803...	Random
19	19	00190000...	Random
20	20	0009000...	Random
21	21	4800069...	Random
22	22	0430802...	Random
23	23	10092000...	Random
24	24	02081074...	Random
25	25	0300500...	Random

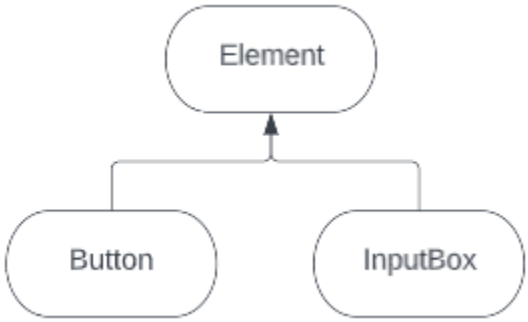
There have been some changes from my initial idea in section 2.10 for a hashing function. In my initial ideas, I planned on using the hashing algorithm SHA-256. I had severely underestimated the complexity of the SHA-256 algorithm and realized that I do not understand much of how it works. So instead, I am using the djb2 hashing algorithm which I can explain. This hash is simple but very effective and does not produce many collisions.

I also did not have a salt in my initial ideas as I did not think it was necessary. However, after researching hashing, I have learned that there are tables online called rainbow tables which store all the hash values of common passwords for many different hashing algorithms. A hacker can just find the password by finding the hash on these tables. A salt is a string of random characters and is concatenated to the end of a password before hashing. This helps as a hacker will not be able to find this password on a rainbow table as it is no longer a common password. Although the salt is saved in the database, it does not help the hacker figure out the password.

3.2 System overview:



3.3 Evidence of built classes:




```
COLOR_ACTIVE = pygame.Color('maroon')
COLOR_PASSIVE = pygame.Color('gray')
FONT = pygame.font.SysFont('Calluna', 32)
```

OOP

```
class Element:  # This class acts as a parent class
    def __init__(self, x, y, w, h, text=''):  # These are all the parameters it takes in
        self.x = x  # x coordinate element starts at
        self.y = y  # y coordinate element starts at
        self.w = w  # width of the element
        self.h = h  # height of the element
        self.text = text  # text in button/prompt of input box
        self.rect = pygame.Rect(x, y, w, h)  # creates a rectangle with the data above
        self.active = False  # if self.active is true, the button/input box has been clicked

    def draw(self, screen):  # this function is empty as it will be overridden in both child classes
        pass

    def handle_event(self, event): # This function draws the element on the board
        if event.type == pygame.MOUSEBUTTONDOWN:  # if event is a mouse click
            if self.rect.collidepoint(event.pos):  # if position of mouse click is within the rectangle, active is true
                self.active = True
            else:  # else, active remains false
                self.active = False

class Button(Element):  # The button class inherits the element class as shown
    def __init__(self, x, y, w, h, text=''):
        super().__init__(x, y, w, h, text)  # All these parameters are handled by the parent class.
        self.COLOR_TEXT = (0, 0, 0)  # color of the text: Black
        self.txt_surface = FONT.render(text, True, self.COLOR_TEXT)
```

```
        if self.txt_surface.get_width() > w:
            # If the length of the text is greater than the user entered width of the button, the width increases to ensure that text is inside
              the button
            self.rect = pygame.Rect(x, y, self.txt_surface.get_width() + 10, h)
        else:
            self.rect = pygame.Rect(x, y, w, h)

        self.COLOR_BUTTON = pygame.Color('burlywood1')  # this is the color of the button: beige

    def draw(self, screen):  # this needs to be overridden as it is unique to buttons
        pygame.draw.rect(screen, self.COLOR_BUTTON, self.rect)
        screen.blit(self.txt_surface, (self.rect.x +5, self.rect.y + 5))  # display text in the middle of button

    #  The Button class completely inherits the handle_event function from the parent class

class InputBox(Element):  # The InputBox class inherits the element class as shown

    def __init__(self, x, y, w, h, text, textType = 'text'):  # The textType parameter is not handeled by the parent class.
        super().__init__(x, y, w, h, text)  # All these parameters are handled by the parent class.
        self.OGw = w  # Original width
        self.color = COLOR_PASSIVE  # color passive is the color of the input box when not clicked on
        self.userInput = ''  # This is what the user will type
        self.txt_surface = FONT.render(self.userInput, True, self.color)
        self.textType = textType  # type of text: text/password/integer
        self.text_surface = FONT.render(self.text, True, (0,0,0))

    def handle_event(self, event):  # The handle_event function is overridden as an input box works in a different way as it has to
                                      handle key presses.
        if event.type == pygame.MOUSEBUTTONDOWN:  # if event is a mouse click
            if self.rect.collidepoint(event.pos):  # if mouse click position is inside rectangle(input box)
                self.active = True  # clicked
            else:
                self.active = False

            if self.active:  # If self.active is true:
                self.color = COLOR_ACTIVE  # if the input box is active, change the color of the box to indicate that it has been clicked on
            else:
                self.color = COLOR_PASSIVE  # If the input box is not active, change the color of the box to indicate that it is not in focus

        if event.type == pygame.KEYDOWN:  # if user clicks a button on keyboard:
            if self.active:  # and if the input box has been clicked on
```

```
        if event.key == pygame.K_BACKSPACE:  # if keypress is a backspace
            self.userInput = self.userInput[:-1]  # remove last letter of userInput, this is a stack structure as FILO
            STACKS FILO
        elif self.textType.lower() == 'int':  # if text type is an integer, only allow numbers 1-9 on the input box
            if event.key in [pygame.K_0, pygame.K_1, pygame.K_2, pygame.K_3, pygame.K_4,
                             pygame.K_5, pygame.K_6, pygame.K_7, pygame.K_8, pygame.K_9]:  # if keypress is any of these keys:
                self.userInput += event.unicode  # add number to user input

        else:  # if text type is a string, add the character to the user input
            self.userInput += event.unicode

        if self.textType.lower() == 'password':  # if the type is password, replace all the characters to display with *
            self.txt_surface = FONT.render('*' * len(self.userInput), True, self.color)

        else:
            self.txt_surface = FONT.render(self.userInput, True, self.color)

    def update(self, screen):  # This function does not exist in the parent class
        # Resize the box if the text is too long.
        screen.fill((255, 255, 255), (self.x - self.text_surface.get_width() - 5, self.y, self.w + self.text_surface.get_width()+10, self.h))
        width = max(self.OGw, self.txt_surface.get_width() + 10)  # pick between whichever is greater: the original width or the width of
                                                                  the text
        self.w = width
        self.rect.w = width

    def draw(self, screen):  # This function is overridden as I will need to add a prompt before the input box
        screen.blit(self.text_surface, (self.rect.x - self.text_surface.get_width() - 5, self.rect.y + 5))  # this renders the text to the left of the
                                                                                                             input box
        screen.blit(self.txt_surface, (self.rect.x + 5, self.rect.y + 5))  # This renders the users text inside the input box

        pygame.draw.rect(screen, self.color, self.rect, 2)  # This draws the input box
```

3.4 Evidence of complete code listings

Login Screen:

```
'''Imports'''

import pygame  # Pygame is the GUI module I chose to use because of my familiarity with it
import time  # Time is used for keeping track of how long the user has been playing a board for
from elements import Button  # Elements is my Python file containing the element classes shown in 3.3
from elements import InputBox
import sudoku_solver  # Sudoku_solver is my Python file containing all the functions used in solving the boards
import re  # RE is a library for regular expressions. I will use this for password complexities
import sqlite3 as sql  # SQLite 3 is used to allow communication between Python and the database
import random  # I will use random to generate a random 16 character salt or password security

'''Login'''

HEIGHT = 550  # Height of the window. Constant value
WIDTH = 900  # Width of the window. Constant value


pygame.init()  # initialise pygame


window = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Sudoku!")


def main():  # This is the start point of the program. This function includes the login system.
    window.fill((255, 255, 255))  # Fill entire screen white
    pygame.font.init()
    font = pygame.font.SysFont('Calluna', 100)
    text = font.render("Sudoku!", True, pygame.Color('burlywood1'))
    window.blit(text, (WIDTH // 2 - text.get_width() // 2, 10))  # blit renders text on screen

    pygame.display.update()
    username = InputBox(400, 250, 200, 32, 'username: ')  # prompt for this Inputbox is username:
    password = InputBox(400, 300, 200, 32, 'password: ', textType='password')  # prompt for this Inputbox is password:
    input_boxes = [username, password]  OOP
    registerButton = Button(500, 350, 100, 32, 'Register')
    loginButton = Button(400, 350, 70, 32, 'Login')

    while True:  # While on the main screen
        for event in pygame.event.get():
        # an event can include anything that happens while on the GUI such as mouse movements and switching tabs
            if event.type == pygame.QUIT:  # if the quit window button is clicked, quit the program
                quit()
```

```
        for box in input_boxes:
            box.handle_event(event)  # Handles any event given to it.

        registerButton.handle_event(event)

        if registerButton.active:  # If register button is clicked:
            registerButton.active = False
            register(username.userInput,
                     password.userInput)  # Passes the username and password to the register function.

        loginButton.handle_event(event)
        if loginButton.active:  # If login button is clicked:
            loginButton.active = False

            if login(username.userInput,
                     password.userInput):
                # Passes the username and password to the login function which returns true if login is successful
                usermenu(username.userInput)  # if login is successful, program passes username to usermenu so that user can access
                                              their profile

    for box in input_boxes:
        box.update(window)  # resizes the input box if text is too long

    for box in input_boxes:
        box.draw(window)  # renders the input boxes to the screen

    registerButton.draw(window)  # renders the buttons to the screen
    loginButton.draw(window)
    pygame.display.update()  # updates display


def login(u, p):  # login function which takes in username and password
    if len(u) == 0 or len(p) == 0:  # if username or password is empty: prompt user to fill fields
        font = pygame.font.SysFont('ComicSans MS', 20)
        text = font.render('Please complete all fields', True, (0, 0, 0))
        window.fill((255, 255, 255), (0, 400, WIDTH, 29)) # clear the previous text
        window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))
        pygame.display.update()
        return
```

```
    db.execute(f"SELECT username FROM users")  # selects all the usernames from the table users
    UsersOnDb = db.fetchall()  # These are all the usernames in a list of tuples
    exists = False
    for tup in UsersOnDb:  # for every tuple in the list
        if u in tup:  # If the username is in the tuple, username exists LINEAR SEARCH
            exists = True
            break  # if the username is found, quit the loop

    if exists:
        passOnDB = db.execute(
            f"SELECT password from users WHERE username = '{u}'").fetchone()[0]  # Password for that username  is selected
        salt = db.execute(f"SELECT salt from users WHERE username = '{u}'").fetchone()[0]

        if hash(p + salt) == passOnDB:
            return True
        # If hashed password the user enters is the same as the hashed password stored on the database, login successful

        else:  # password is incorrect
            font = pygame.font.SysFont('ComicSans MS', 20)
            text = font.render('Incorrect password.', True, (0, 0, 0))
            window.fill((255, 255, 255), (0, 400, WIDTH, 29)) # clear the previous text
            window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))
            pygame.display.update()

    else:  # username does not exist on the database
        font = pygame.font.SysFont('ComicSans MS', 20)
        text = font.render('Username does not exist.', True, (0, 0, 0))
        window.fill((255, 255, 255), (0, 400, WIDTH, 29)) # clear the previous text
        window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))
        pygame.display.update()

def hash(p):  # hash function takes in the password.  HASHING
    a = 5381  # prime hash multiplier
    for x in p:  # for every character in the password,
        a = ((a << 5) + a) + ord(x)
        # a = a left shifted by 5 bits and added to the previous  a. This is added to the ASCII value of the character in password
    return hex(a & 0xFFFFFFFF)  # return the hexadecimal value of a AND -1 (logical AND)

def register(u, p):  # Register function which takes in username and password
    reg = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!#%*?&]{6,20}$"  REGULAR EXPRESSIONS
    # This is the regular expression a password must comply with to be valid
    pattern = re.compile(reg)
```

```
if len(u) == 0 or len(p) == 0:  # If username and password is empty, user is prompted to fill all the fields
    font = pygame.font.SysFont('ComicSans MS', 20)
    text = font.render('Please complete all fields', True, (0, 0, 0))
    window.fill((255, 255, 255), (0, 400, WIDTH, 29)) # clear the previous text
    window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))  # render the text on screen
    pygame.display.update()  # update the display
    return

db.execute(f"SELECT username FROM users")  # all usernames are selected
UsersOnDb = db.fetchall()
exists = False

for tup in UsersOnDb:  # for every tuple in the list
    if u in tup:  # If the username is in the tuple, the username already exists
        exists = True
        break  # if the username is found, quit the for loop

if exists:  # if the username already exists, prompt the user to change the username
    print('username already exists, please try again')
    font = pygame.font.SysFont('ComicSans MS', 20)
    text = font.render('username already exists, please try again.', True, (0, 0, 0))
    window.fill((255, 255, 255), (0, 400, WIDTH, 29)) # clear the previous prompt on the screen
    window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))
    pygame.display.update()

else:  # if the username does not already exist:
    if re.search(pattern, p):  # if password complies with the regular expression
        salt = ''  # A salt is a random string of characters that is concatenated (added) to a password before it is hashed
        for x in range(16):  # generating a salt of length 16
            char = str(chr(random.randint(37, 129)))  # pick a random integer between 37 and 129 and find the ASCII character of it.
            salt += char. # salt = previous salt added to the new random character SALT for HASHING

        db.execute("INSERT INTO users(username, password, salt) VALUES ('{u}','{hash(p+salt)}', '{salt}')")
        # insert the new username, hashed password, and salt into the database. This may seem counterintuitive as I'm giving away
        # half the password to a potential hacker but using a salt means that he will not be able to use a rainbow table to
        # reverse hash the passwords easily.

        conn.commit()  # commit the changes to the database
        font = pygame.font.SysFont('ComicSans MS', 20)
        text = font.render('Successfully registered', True, (0, 0, 0))
        window.fill((255, 255, 255), (0, 400, WIDTH, 29))  # clear the previous prompt on the screen
        window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))  # render the text
```

```
        pygame.display.update()  # update the display

    else:  # if the password does not comply with the regular expression, the user is given the requirements for a strong password
        font = pygame.font.SysFont('ComicSans MS', 17)
        text = font.render("password should have at least: one uppercase letter, one lowercase letter, one number, one
                           symbol(!#$...)", True, (0, 0, 0))  # These are the requirements for the password
        window.fill((255, 255, 255), (0, 400, WIDTH, 29))  # clear the previous text
        window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))
        pygame.display.update()
        return
```

Login screen: Start point of the program. To access the next screens, the user needs to log in successfully		
Procedures	Purpose	Parameters
main	Responsible for rendering all the text, buttons, and input boxes. This function also handles all the events.	
login	This function returns true if the user enters the correct username and password.	u: username p: password
register	This function is responsible for creating new accounts by storing the username and hashed passwords into the database. Only passwords that are complex enough are allowed.	u: username p: password
hash	Converts the password into a hashed password using an algorithm called djb2. I have also used a salt so that hashed passwords will be unique even with the same passwords	p: password

A. The “Sudoku!” text, the input boxes, and the buttons are created by the ‘main’ function using the button and input box classes.

B. When the login button is clicked, the ‘login’ function is run. The function takes in the username and password inputted by the user and gives a Boolean value True if the login is successful.

C. When the register button is clicked, the ‘register’ function is run, The function takes in the username and password inputted by the user and saves the username and a hash for the password (using the ‘hash’ function) on the database.

User Profile:

```
'''User Profile'''
def merge(arr):
    if len(arr) > 1:  # if the length of each sub array is more than one
        lefta = arr[:len(arr)//2]  # split array from the start to the middle point
        righta = arr[len(arr)//2:]  # split array from the middle point to the end

        merge(lefta)  # recurse until each array only consists of 1 element
        merge(righta) # recurse until each array only consists of 1 element RECURSION, MERGE SORT

        i = j = k = 0 #i keeps track of the left most element in left array, j keeps track of left most element in right array, k is the pointer of
                      the merged array
        while i < len(lefta) and j < len(righta):  # while i is less than the length of left array and j is less than the length of the right array
            if lefta[i] < righta[j]:  # if value of left array at I is less than the value of right array at j
                arr[k] = lefta[i]  # set array position k to the element at pointer of left array
```

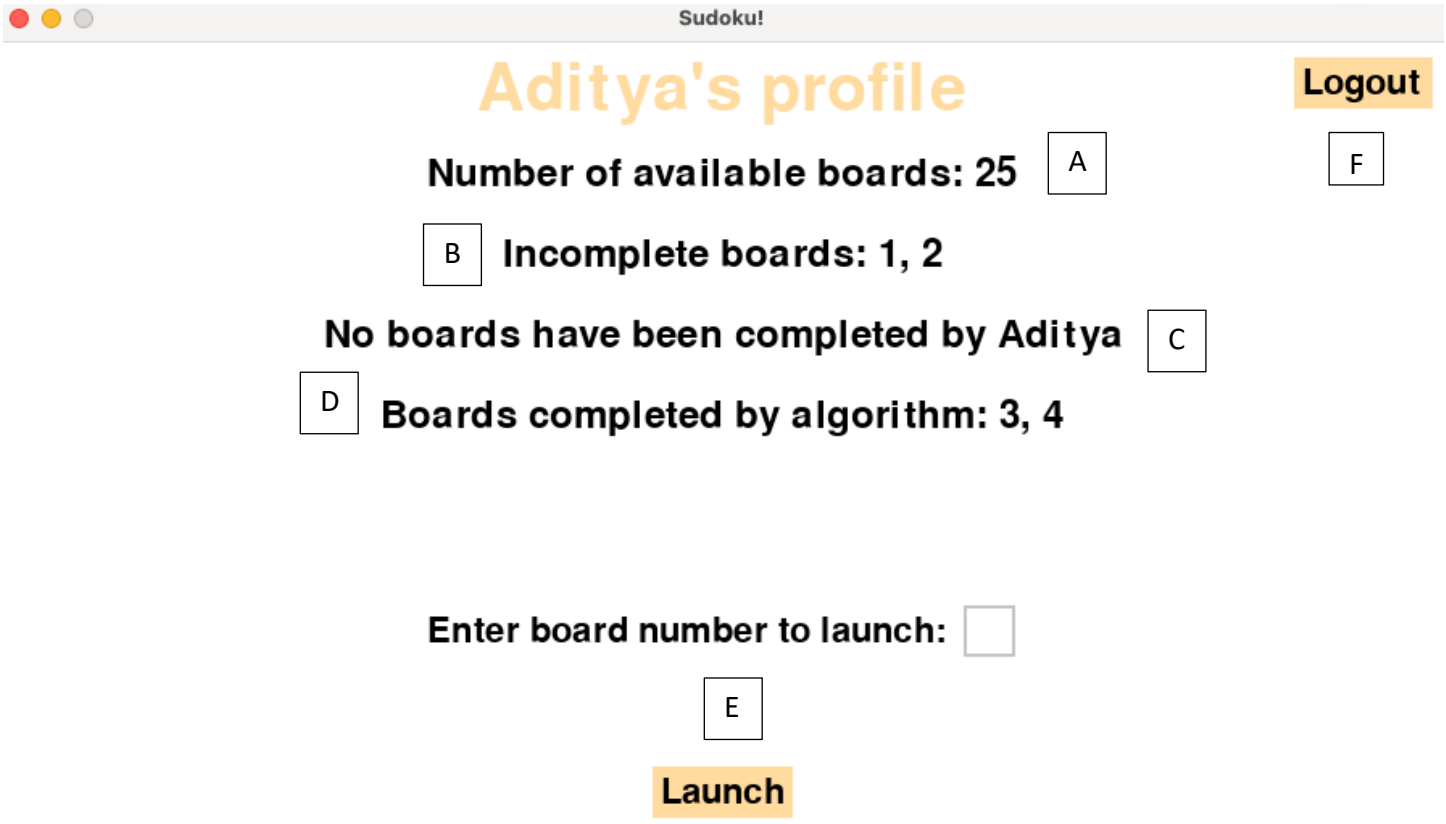
```
            i+=1  # increment left pointer by 1
        else:  # if value of left array at I is greater than or equal to the value of right array at j
            arr[k] = righta[j]  # set array position k to the element at pointer of right array
            j+=1  # increment right pointer by 1

        k+=1  # increment k by one

    # run this while i is still less than the length of the left array
    while i < len(lefta):  # while i is less than the length of the left array
        arr[k] = lefta[i]  # set element at position k in array to the element at i in left array
        i+=1  # increment left pointer by 1
        k+=1  # increment k by 1

    # run this while j is still less than the length of the right array
    while j < len(righta):  # while j is less than the length of the right array
        arr[k] = righta[j]  # set element at position k in array to the element at j in right array
        j+=1  # increment right pointer by 1
        k+=1  # increment k by 1

    return arr  # return the sorted array
  else:  # already sorted
    return arr

def usermenu(u):  # usermenu takes in username of the user that logged in.
  window.fill(BACKGROUND_COLOR)  # entire screen is filled to display users profile
  font = pygame.font.SysFont('Soria', 60)
  text = font.render(f"{u}'s profile", True, pygame.Color('burlywood1'))
  window.blit(text, (WIDTH // 2 - text.get_width() // 2, 10))

  NBoardsCompleted = list(db.execute(
    f"SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'True' ORDER BY
    BoardNumber DESC").fetchall())  # selects all the board numbers which have been completed by the user

  font = pygame.font.SysFont('Soria', 35)
  if len(NBoardsCompleted) == 0:  # If no boards have been completed by user:
    NBoardsCompletedText = font.render(f"No boards have been completed by {u}", True, (0, 0, 0))
    window.blit(NBoardsCompletedText, (WIDTH // 2 - NBoardsCompletedText.get_width() // 2, 170))

  else:  # If boards have been completed by user:
    NBoardsCompleted = [x[0] for x in NBoardsCompleted]  # make a list of the first element from each tuple in NBoardsCompleted
    NBoardsCompleted = merge(NBoardsCompleted)  # sort list in ascending order using merge sort
    NBoardsCompleted = map(str, NBoardsCompleted)  # make every value in the list a string
    NBoardsCompletedText = font.render(f"Boards completed by {u}: {', '.join(NBoardsCompleted)}", True, (0, 0, 0))
    window.blit(NBoardsCompletedText, (WIDTH // 2 - NBoardsCompletedText.get_width() // 2, 170))
```

```
NBoardsIncomplete = list(db.execute(
        f"SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'False' ORDER BY
        BoardNumber DESC").fetchall())  # select all the board numbers which have not been completed by the user
if len(NboardsIncomplete) == 0:  # If no boards have been started:
    NboardsIncompleteText = font.render(f"No boards have been started", True, (0, 0, 0))
    window.blit(NboardsIncompleteText, (WIDTH // 2 - NboardsIncompleteText.get_width() // 2, 120))

else:
    NboardsIncomplete = [x[0] for x in NboardsIncomplete]  # make a list of the first element from each tuple in NboardsIncomplete
    NboardsIncomplete = merge(NboardsIncomplete)  # sort list in ascending order using merge sort
    NboardsIncomplete = map(str, NboardsIncomplete) # make every value in the list a string
    NboardsIncompleteText = font.render(f"Incomplete boards: {', '.join(NboardsIncomplete)}", True, (0, 0, 0))
    window.blit(NboardsIncompleteText, (WIDTH // 2 - NboardsIncompleteText.get_width() // 2, 120))

NumberOfBoards = db.execute("SELECT COUNT(*) FROM defaultBoards").fetchone()[0]
# Select number of rows in defaultBoards
NumberOfBoardsText = font.render(f"Number of available boards: {NumberOfBoards}", True,
                                 (0, 0, 0))  # displays the number of available boards
window.blit(NumberOfBoardsText, (WIDTH // 2 - NumberOfBoardsText.get_width() // 2, 70))

BoardsCompletedAlgo = list(db.execute(
    f"SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'SolvedWithAlgo' ORDER BY
    BoardNumber DESC").fetchall())  # select all boards which have been completed by algorithm

if len(BoardsCompletedAlgo) > 0:  # if any boards have been completed by algorithm:
    BoardsCompletedAlgo = [x[0] for x in BoardsCompletedAlgo]
    # make a list of the first element from each tuple in BoardsCompletedAlgo
    BoardsCompletedAlgo = merge(BoardsCompletedAlgo) # sort list in ascending order using merge sort
    BoardsCompletedAlgo = map(str, BoardsCompletedAlgo)  # make every value of the list a tring
    BoardsCompletedAlgoText = font.render(f"Boards completed by algorithm: {', '.join(BoardsCompletedAlgo)}", True,
                                          (0, 0, 0))  # display the board numbers solved by algorithm
    window.blit(BoardsCompletedAlgoText, (WIDTH // 2 - BoardsCompletedAlgoText.get_width() // 2, 220))  # render text

LaunchNumber = InputBox(600, 350, 32, 32, 'Enter board number to launch: ', textType='int')
# creates an input box for user to enter board number to launch
Launch = Button(WIDTH // 2 - 87 // 2, 450, 87, 32, 'Launch')
pygame.display.update()

LogOut = Button(805, 10, 86, 32, 'Logout')  # creates a log out button.
```

```
while True:  # While on user menu
    for event in pygame.event.get():
    # an event can include anything that happens while on the GUI such as mouse movements and clicking a button
        if event.type == pygame.QUIT:  # if the quit window button is clicked, quit the program
            quit()

        Launch.handle_event(event) # handle all the events
        LaunchNumber.handle_event(event)
        LogOut.handle_event(event)

        if LogOut.active:  # if log out button is clicked, the user is taken back to the login page
            main()

        elif LaunchNumber.userInput != '':  # if launch number field is not empty,
            if 0 < int(LaunchNumber.userInput) <= NumberOfBoards:
            # if the board number the user inputs is an available board number:
                if Launch.active:  # if launch button is clicked:
                    Launch.active = False
                    BoardNumber = LaunchNumber.userInput  # boardNumber to launch is user input number

                    try:  # try launching a board number,
                        # if the board number has already been played by that user,
                        # an error is generated as inserting the same board number and username in another row causes composite key to
                          be non unique
                        board = db.execute(    CROSS TABLE SQL STATEMENTS
                            f'SELECT Board FROM "main"."defaultBoards" WHERE BoardNumber = {BoardNumber}').fetchone()[0]
                            # select the board from default boards where board number is the user inputted board number

                    2D ARRAY
                        board = [list(map(int, line)) for line in board.splitlines()]  # splits the text board into a 2d array.
                        og_board = [[board[x][y] for y in range(len(board[0]))] for x in range(len(board))]
                        # og board is the same as the unedited board
                        db.execute(
                            f'INSERT INTO "main"."BoardUserLink" ("username", "BoardNumber", "Completed", "Time") VALUES
                            ("{u}", {BoardNumber}, "False", 0)')
                            # insert a new row for the game with a uniqe primary key of this username and board number
                        conn.commit()  # commit changes to database
                        gameloop(u, BoardNumber, board, og_board)  # launch game
                    except sqlite3.IntegrityError:  # if the game has already taken place before, the new game cannot be created as the
                    # composite key that will be created already exists in the database causing an integrity error. So launch the saved game

                        board = db.execute(f'SELECT SavedBoard from "main"."BoardUserLink" WHERE username = "{u}" AND
                                           BoardNumber ={BoardNumber}').fetchone()[0]  # select the saved board form this game
```

```
                board = [list(map(int, line)) for line in board.splitlines()]  # splits the text board into a 2d array.

                og_board = db.execute(f'SELECT Board from "main"."defaultBoards" WHERE BoardNumber =
                                      {BoardNumber}').fetchone()[0]

                  # select the default board for that board number and make this the original board
                og_board = [list(map(int, line)) for line in og_board.splitlines()]  # splits the original board into a 2d array.
                gameloop(u, BoardNumber, board, og_board)  # launch saved game

            else:  # if board number that user enters does not exist:
                font = pygame.font.SysFont('ComicSans MS', 20)
                text = font.render(f'Board number {LaunchNumber.userInput} does not exist.', True, (0, 0, 0))
                window.fill((255, 255, 255), (0, 400, WIDTH, 29))  # clear previous text
                window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))
                # render text informing the user that the board number does not exist
                pygame.display.update()  # update the display

        LaunchNumber.update(window)  # update size of input box if text is too long
        LaunchNumber.draw(window)  # render the input box along with its prompt

        Launch.draw(window)  # render launch game button
        LogOut.draw(window)  # render logout button
        pygame.display.update()  # update display
```

User profile: Start point of the program. To access the next screens, the user needs to log in successfully		
Procedures	Purpose	Parameters
usermenu	Responsible for all the text information displayed on the screen. This function also launches the game loop once the user enters a valid board number and clicks on the launch button	u: username



A. This is the number of boards available and is determined by the SQL statement:
SELECT COUNT(*) FROM defaultBoards

B. These are the boards that the user has started but not finished and are determined by the SQL statement:
SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'False'

C. These are the boards that the user has completed without using the solving algorithm and are determined by the SQL statement:
SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'True'

D. These are the boards that have been completed by the solving algorithm and are determined by the SQL statement:
SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'SolvedWithAlgo'

E. This is where the user can enter a board number from the valid boards (1-25) and click launch to launch the game.

F. Clicking this button will log out the user. For this screen to be accessed again, users need to log in successfully.

Game screen:

```
'''Game Screen'''
def gameloop(u, BoardNumber, board, og_board):  # gameloop takes in the username of the user that is logged in, the board
                                                number they input, the board/saved board and the original board
    global hintCount  # hintCount needs to be available to other functions and passing it on makes code messier
    hintCount = 0  # number of hints used per game

    solved_board = sudoku_solver.BT([[og_board[x][y] for y in range(len(board[0]))] for x in range(len(board))])
    # this is the board that is solved using the recursion algorithm and will be used for hints

    start_time = int(time.time())  # timer starts when game is launched Time simulation


    update_gui(board, og_board, 'full')
    solveButton = Button(700, 400, 70, 30, 'Solve')
    solveButton.draw(window)
    MenuButton = Button(720, 20, 130, 30, f"{u}'s menu")
    MenuButton.draw(window)

    pygame.display.update()

    timePlayed = db.execute(
        f"SELECT Time from 'main'.'BoardUserLink' WHERE username = '{u}' AND BoardNumber = {BoardNumber}").fetchone()[0]
        # select the time already played on that game

    timeInSec = 0 + timePlayed  # add the selected time to start time
    timeTaken = convert(timeInSec)  # convert seconds to hr:min:secs

    while True:
        font = pygame.font.SysFont('ComicSans MS', 20)
        completed = db.execute(f"SELECT Completed from 'main'.'BoardUserLink' WHERE username = '{u}' AND BoardNumber
                               = {BoardNumber}").fetchone()[0]  # selects the state of the board in that game

        if completed == 'False':  # if board is completed, total time taken is recorded.
            timeInSec = int(time.time()) - start_time + timePlayed  # time is the time now - the start time + the time already played
            timeTaken = convert(timeInSec)  # seconds is converted into hh:mm:ss format Time simulation

        text = font.render(f'Time: {timeTaken}', True, (0, 0, 0))
        window.fill((255, 255, 255), (100, 510, WIDTH, 29))  # removes previous text
        window.blit(text, (220, 510))  # renders time taken text
        pygame.display.update()  # updates display
```

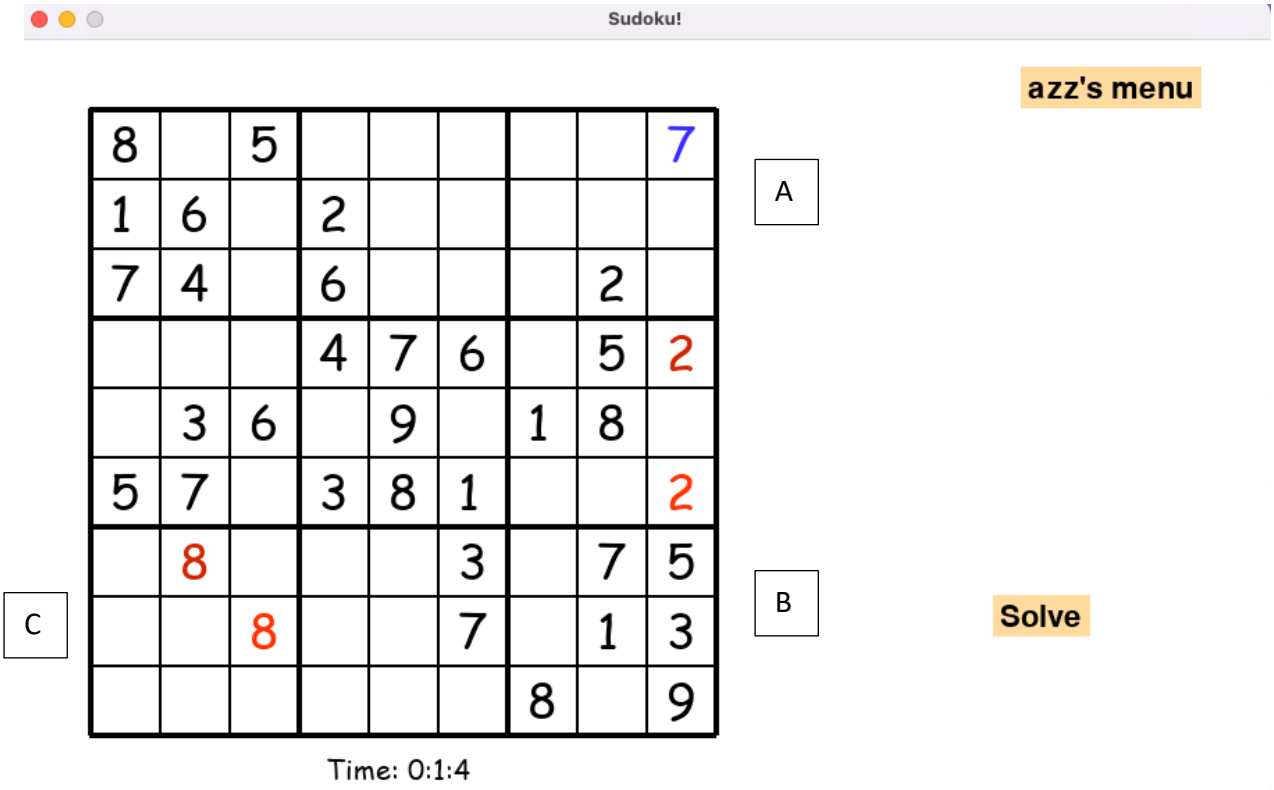
```
if board == solved_board and completed != 'SolvedWithAlgo':  # if board is completed without the use of the algorithm then
    db.execute(f"UPDATE 'main'.'BoardUserLink' SET (Completed, Time) = ('True', {timeInSec}) WHERE username = '{u}'
                    and BoardNumber = {BoardNumber}")  # update the status of the board and the time taken to solve board.
    conn.commit()  # commit changes to database

    font = pygame.font.SysFont('Soria', 40)
    text = font.render("Correct!", True, pygame.Color('burlywood1'))
    window.blit(text, (550, HEIGHT // 2 - text.get_height()))

elif board == solved_board and completed == 'SolvedWithAlgo':  # if board is solved using algorithm:
    font = pygame.font.SysFont('Soria', 40)
    text = font.render("Solved With Algorithm", True, pygame.Color('burlywood1'))
    # inform the user that the board was solved by the algorithm.
    window.blit(text, (550, HEIGHT // 2 - text.get_height()))

for event in pygame.event.get():  # for every event that takes place:
    if event.type == pygame.QUIT:  # if the quit window button is clicked, save the board to the game in the database and quit
        boardString = ''
        for sub_list in board:
            boardString = boardString + (''.join(str(digit) for digit in sub_list)) + '\n'  # convert 2d array into a text board
        db.execute(f'UPDATE "main"."BoardUserLink" SET (SavedBoard, Time) = ("{boardString}", {timeInSec}) WHERE
                        username = "{u}" AND BoardNumber = {BoardNumber}')  # update the board to the saved board
        conn.commit()
        quit()

    if event.type == pygame.MOUSEBUTTONDOWN:  # if the mouse button is clicked:
        pos = pygame.mouse.get_pos()  # get the position of mouse [x,y]
        solveButton.handle_event(event)  # handle all the events for the solve button
        MenuButton.handle_event(event)  # handle all the events for the menu button
        i = pos[1] // 50 - 1  # get position of mouse in terms of 2d grid.  Example: if I is 3, mouse is on 3rd row
        j = pos[0] // 50 - 1 # get position of mouse in terms of 2d grid. Example: if j is 4, mouse is on 4th column

        if solveButton.active:  # if solve button is clicked
            db.execute(
                f"UPDATE 'main'.'BoardUserLink' SET (Completed, Time) = ('SolvedWithAlgo', {timeInSec}) WHERE username =
                    '{u}' and BoardNumber = {BoardNumber}")  # update the board state and the time for that game
            conn.commit()  # commit changes to database
            board = solved_board # board is solved board
            update_gui(board, og_board, 'half')  # update the GUI to display solved board
```

```
            elif MenuButton.active:  # if the menu button is clicked
                boardString = ''
                for sub_list in board:
                    boardString = boardString + (''.join(str(digit) for digit in sub_list)) + '\n' # convert 2d array board to text to save board
                db.execute(
                    f'UPDATE "main"."BoardUserLink" SET (SavedBoard, Time) = ("{boardString}", {timeInSec}) WHERE username =
                    "{u}" AND BoardNumber = {BoardNumber}')  # save the board and the time taken in this session
                conn.commit()  # commit changes to database
                usermenu(u)  # go back to user menu

            elif -1 < i < 9 and -1 < j < 9 and board != solved_board:
                # if user clicks inside the board GUI and the board is not solved yet:
                insert(i, j, board, og_board, solved_board)  # insert a value into  that position

            else:
                pass

def convert(second):  # converts seconds to the format hours: minutes: seconds
    second = second % (24 * 3600) SIMPLE MATHTHEMATICAL CALCULATIONS
    hour = second // 3600
    second %= 3600
    minutes = second // 60
    second %= 60

    return f"{hour}:{minutes}:{second}"

def update_gui(board, og_board, mode):  # update GUI takes in the board, og board, and the mode

    if mode == 'half':  # if mode is half, clear the sudoku board
        pygame.draw.rect(window, BACKGROUND_COLOR, (50, 50, 450, 450))
    elif mode == 'full':  # if mode is full, clear entire screen
        pygame.draw.rect(window, BACKGROUND_COLOR, (0, 0, 900, 550))

    font = pygame.font.SysFont('ComicSans MS', 35)

    for i in range(0, 10):  # draw 10 rows and columns to make the GUI board
        pygame.draw.line(window,
                         color=(0, 0, 0),
                         start_pos=(50 + 50 * i, 50),
                         end_pos=(50 + 50 * i, 500),
                         width=4 if i % 3 == 0 else 2)  # every 3 columns, make the column wider to make a 3x3 grid visible
```

```
    pygame.draw.line(window,
                     color=(0, 0, 0),
                     start_pos=(50, 50 + 50 * i),
                     end_pos=(500, 50 + 50 * i),
                     width=4 if i % 3 == 0 else 2) # every 3 rows, make the row wider to make a 3x3 grid visible

font = pygame.font.SysFont('Comic Sans MS', 35)

for i in range(9):  # for every column
    for j in range(9):  # for every row in column

        if str(board[i][j]) in '123456789':  # if a user enters a value on the board
            if board[i][j] != og_board[i][j]:
                valid = sudoku_solver.check_valid(board, board[i][j], [i, j])  # check if it is a valid input (does not produce any clashes)
                if not valid:
                    fontColor = (255, 0, 0)  # if a clash is produced, make number red
                else:
                    fontColor = (50, 50, 255)  # if no clash is produced, make number blue

            else:  # if board value for that cell is a fixed value
                valid = sudoku_solver.check_valid(board, board[i][j], [i, j])  # check if it is a valid input
                if not valid:
                    fontColor = (200, 0, 0)  # if a clash is produced, make the number dark red to show that it is a fixed value
                else:
                    fontColor = (0, 0, 0)  # if no clash is produced, make number black to show that it is a fixed value

            value = font.render(str(board[i][j]), True, fontColor)
            window.blit(value, ((j + 1) * 50 + 15, (i + 1) * 50))  # render the numbers on the board for each cell
    pygame.display.update()  # update display
```

```
def insert(i, j, board, og_board, solved_board):  # insert is a function for the user to input values onto the board
    global hintCount
    while True:
        for event in pygame.event.get():  # for every event:
            if event.type == pygame.QUIT:
                return

            elif event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()  # get mouse pos
                i = pos[1] // 50 - 1  # get position of mouse in terms of 2d grid
                j = pos[0] // 50 - 1
                if -1 < i < 9 and -1 < j < 9 and board != solved_board:
                    insert(i, j, board, og_board, solved_board) RECURSION
                    # recursion. so that the user can click on another cell after clicking on one cell and still be able to edit it
                else:
                    return   # if mouse click is not on board, return to the game loop


            elif event.type == pygame.KEYDOWN and -1 < i < 9 and -1 < j < 9:  # if event on the board is a key press on the keyboard:

                if og_board[i][j] != 0:  # if board value is not fixed, dont change anything
                    return

                if event.key == 48 or event.key - 48 == board[i][j]:  # if the user presses 0 or the number in the cell, delete the value
                    board[i][j] = 0  # 0 denotes an empty cell
                    update_gui(board, og_board, 'half')  # update the board
                    return

                if 0 < event.key - 48 < 10:  # if key press is the numbers between 0 and 10

                    board[i][j] = event.key - 48  # board value for that cell is the key the user presses
                    update_gui(board, og_board, 'half')  # update the GUI
                    return

                if event.key == 72 or event.key == 104 and hintCount < 3:  # if key press is the letter h/H and user has used less than 3 hints
                    board[i][j] = solved_board[i][j]  # give the correct answer for that cell
                    update_gui(board, og_board, 'half')  # update the GUI
                    hintCount += 1  # increment hint count by 1
                    return
            else:
                pass # if the user does not press a button or a key, don't do anything

main()  # run the main function
```

Game Screen: This screen is the main function of my program and is where the sudoku game will take place. This screen is only accessible through the user menu.		
Global variables:	Hint count: enables program to keep track of how many hints have been used per game	
Procedures	Purpose	Parameters
gameloop	Takes care of most of the events and SQL statements. Keeps track of time played. Creates the buttons and text required on the GUI	u: username BoardNumber Board og_board: original unedited board
convert	Converts time in seconds into time in the format of Hours: minutes: seconds	seconds
update_gui	Responsible for drawing the board and updating it whenever a change is made to it.	Board og_board mode
insert	Allows users to input values into the board and makes sure only valid characters can be entered. Allows users to delete numbers in cells and use their 3 hints.	i j board og_board solved_board
check_valid	explained later	board number position
BT	recursion algorithm to solve board, explained later	board



A. As 7 does not clash with any other 7 in the same row, column, or 3x3 square, it is a valid number and is displayed in blue.

B. As 2 clashes with another 2 in the same column, it is not a valid number and they both are displayed in red

C. As 8 clashes with another 8 in the same 3x3 square, it is not a valid number and they both are displayed in red.

In the image above, 2 different shades of red can be seen. The darker shade represents the fixed/black numbers on the board while the lighter reds are the user inputted numbers.

Sudoku solver

```
'''Sudoku Solver'''

def find_0(board):  # finds all the empty cells in the 2d array
    for i in range(len(board)):  # for every row in the board
        for j in range(9):  # for every column in every row
            if board[i][j] == 0:  # if the cell is empty, return the coordinates of the cell.
                return i, j

    return None  # if there are no more empty cells, return None
```

```
def check_valid(bo, number, pos):  # check if a number in a cell is valid based on sudoku rules

    # checks to see if the number is repeated again on the same row
    for i in range(9):  # for every cell in the same row
        if bo[pos[0]][i] == number and pos[1] != i:
        # if the cell is equal to the number to input and the position of the cell is not the same as the position of the number to input,
         return False as it breaks a rule of soduku
            return False

    # checks to see if the number is repeated again in the same column
    for i in range(9):  # for every cell in the same column
        if bo[i][pos[1]] == number and pos[0] != i:
        # if the cell is equal to the number to input and the position of the cell is not the same as the position of the number to input,
         return False as it breaks a rule of sudoku
            return False

    # checks the 3x3 square
    box_x = pos[1] // 3  # box_x is the x coordinate of the 3x3 grid the number to input is in
    box_y = pos[0] // 3  # box_y is the y coordinate of the 3x3 grid the number to input is in

    for i in range(box_y * 3, box_y * 3 + 3):  # for every column in the 3x3 grid:
        for j in range(box_x * 3, box_x * 3 + 3):  # for every row in the 3x3 grid:
            if bo[i][j] == number and [i, j] != pos:
            # if the cell is equal to the number to input and the position of the cell is not the
             same as the position of the number to input, return False as it breaks a rule of sudoku
                return False
    return True  # if no rules are broken, return True (valid)
```

RECURSION

```
def BT(bo):
# I am using a backtracking algorithm that uses recursion to find a solution to a problem

    find = find_0(bo)  # checks to see if there are any empty spots left
    if not find:
        return bo  # if no empty cells are left, the board is returned as it is completed

    else:
        row, col = find  # the list is split into row and column

    for i in range(1, 10):  # for every valid number that can be inputted:
        if check_valid(bo, i, (row, col)):
                # algorithm tries every number in the empty cell and when it finds a valid number, it
                inserts that number into the cell
            bo[row][col] = i
```

```
        if BT(bo):  # recurses to try and find the next solution to the updated board.
            return bo  # if it solves, the board is returned

        bo[row][col] = 0
            # if no board is returned above, that means that the board becomes invalid while trying
            to solve the rest of the board. Thus, that board value is set to 0 and tried again
    return False  # if no answer is found, return False
```

Sudoku solver: These are the functions that work together to solve a sudoku puzzle

Procedures	Purpose	Parameters
find_0	finds positions of empty cells in a board	board
check_valid	There are 3 main rules in sudoku that determine if a number is valid or not. - A number must only occur once in a row - A number must only occur once in a column - A number must only occur once in the 3x3 grid it is in This function checks if a number in a cell is valid or not. A clash is produced when 2 numbers are not valid. This function is also used in the solving algorithm below	board number position
BT	This is the recursion algorithm that solves a board using refined brute force. The algorithm uses the find_0 function and tries the number 1-9 into a cell and checks if it is valid through the check_valid function. If it is valid, it moves on to the next empty cell and tried the numbers 1-9 there. If a collision if caused in a previous empty cell, it goes back to that cell and tried different numbers until all the cells are filled without validity being broken.	board

Testing

4.1 Testing Strategy

To ensure that all my end user's objectives are met and work properly, I will conduct thorough testing on most objectives. I will be using mostly Blackbox and module testing to make sure that my code works as intended.

Blackbox Testing

Blackbox testing is a form of testing an application/software without knowing the intended functionality of the software. I will be using Blackbox testing to test the functionality of the buttons and textboxes by using typical, erroneous, and extreme data. I will use this to verify that the buttons will open the correct pages and to verify if my application correctly alerts the users if any erroneous/invalid data is entered. For example, leaving a text box empty on the log-in screen and pressing the login button must alert the user that the text boxes are empty.

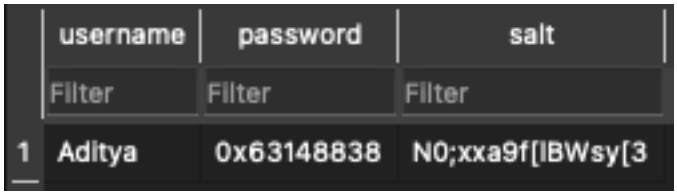
Module Testing

Module testing is a form of testing an application by testing individual subroutines. Instead of testing the entire program at once, the program is tested in small chunks. I will be using this to verify that the login and register functions work as intended. I will also use it to check if my check_valid function on the game screen highlights each number correctly.

4.2 Testing

Test No	1
Objective	2a, 2b, 2c, 2d
Purpose	To ensure that the register button will only accept valid usernames and passwords.
Description	Input data will be inserted into the username and password text boxes and the register button will be clicked.
Input data	**typical:** username: 'Aditya' password: 'Pass#123' **erroneous 1:** username: '' password: '' **erroneous 2:** username: 'Aditya', password: 'Pass#123' **erroneous 3:** username: 'Adi', password: 'pass123'
Expected output	The typical data must pass if the username Aditya does not already exist on the database. Erroneous 1 data must not work as both fields are empty Erroneous 2 data must not work as the username already exists in the database Erroneous 3 data must not work as the password is not complicated enough
pass/fail	**typical:** pass **erroneous 1:** pass **erroneous 2:** pass **erroneous 3:** pass

Typical: Pass



The typical data works as expected. When the username and password were entered into the fields and the register button was clicked, the database was updated, displaying the username and hashed password in the users database. The test passed.

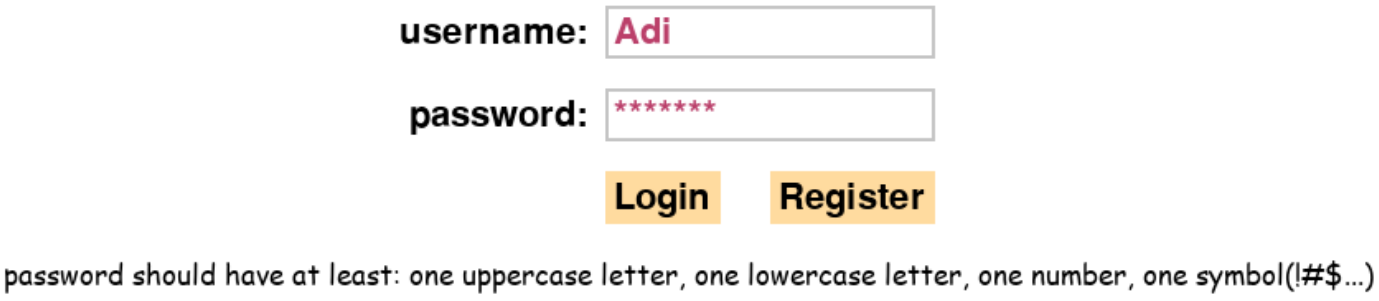
Erroneous 1: Pass

The erroneous 1 data outputs an error as expected. When the username and password fields are empty, the user was prompted to complete all fields. The user data was not saved to the database. The test passed.

Erroneous 2: Pass

The erroneous 2 data outputs an error as expected. As the username already exists in the database, inserting the same username again would cause an integrity error as there are 2 rows of the primary key with the same value. The user is prompted to try again with a new username. The test passed.

Erroneous 3: Pass

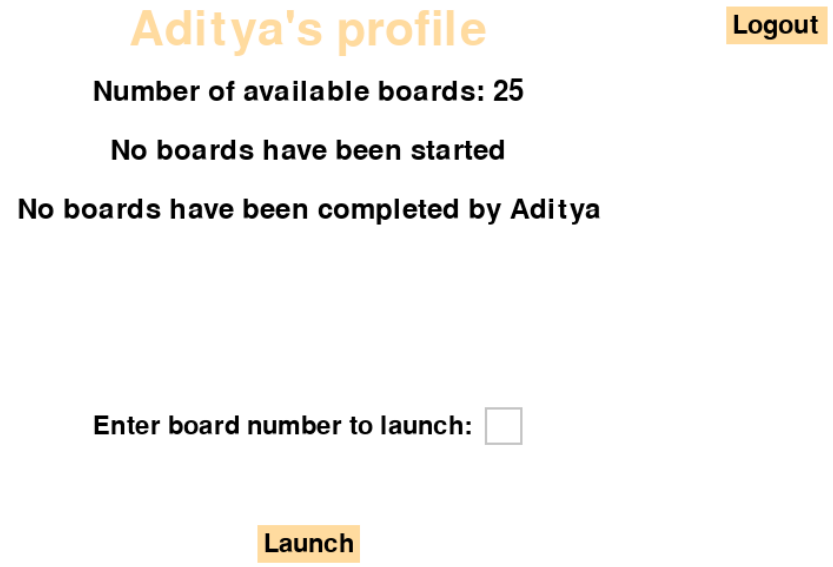


The erroneous data 3 outputs an error as expected. The password 'Pass123' is not complicated enough making it insecure. The user is prompted to try again with a more complicated password and the data is not stored in the database. The test passed.

Test No	2
Objective	3a, 3b, 3c, 3d
Purpose	To ensure that the Login button only accepts valid usernames and their corresponding passwords.
Description	Input data will be inserted into the username and password text boxes and the login button will be clicked
Input data	**typical:** username: 'Aditya' password: 'Pass#123' **erroneous 1:** username: '' password: '' **erroneous 2:** username: 'Pranav', password: 'Pran#321' **erroneous 3:** username: 'Aditya', password: 'Pass#12345'
Expected output	The typical data must pass as the username and password exist in the database as they were created in Test 1 Erroneous 1 data must not work as both fields are empty Erroneous 2 data must not work as the username does not exist in the database Erroneous 3 data must not work as the password is incorrect
pass/fail	**typical**: pass **erroneous 1**: pass **erroneous 2**: pass **erroneous 3**: pass

Typical: Pass

The typical data passed as expected. The username was registered in Test 1. As the user is taken to their profile, we know that the login was successful. The test passed.

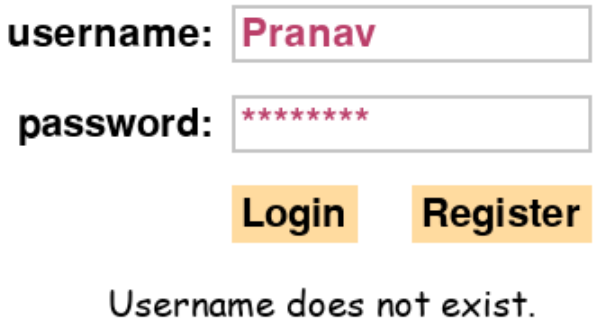


Erroneous 1: Pass

Same as in test 1. The test passed

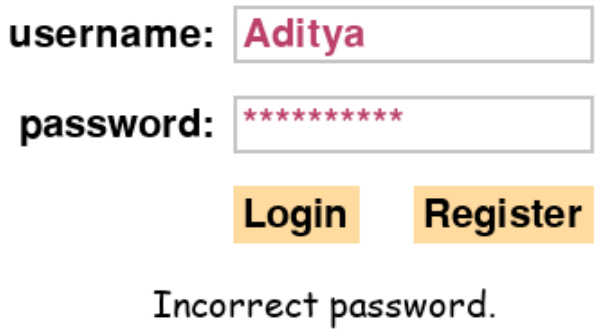
Erroneous 2: Pass

The erroneous data outputs an error as expected. The username 'Pranav' does not exist in the database so it should not log in. The user is prompted that the username does not exist. The test passed.



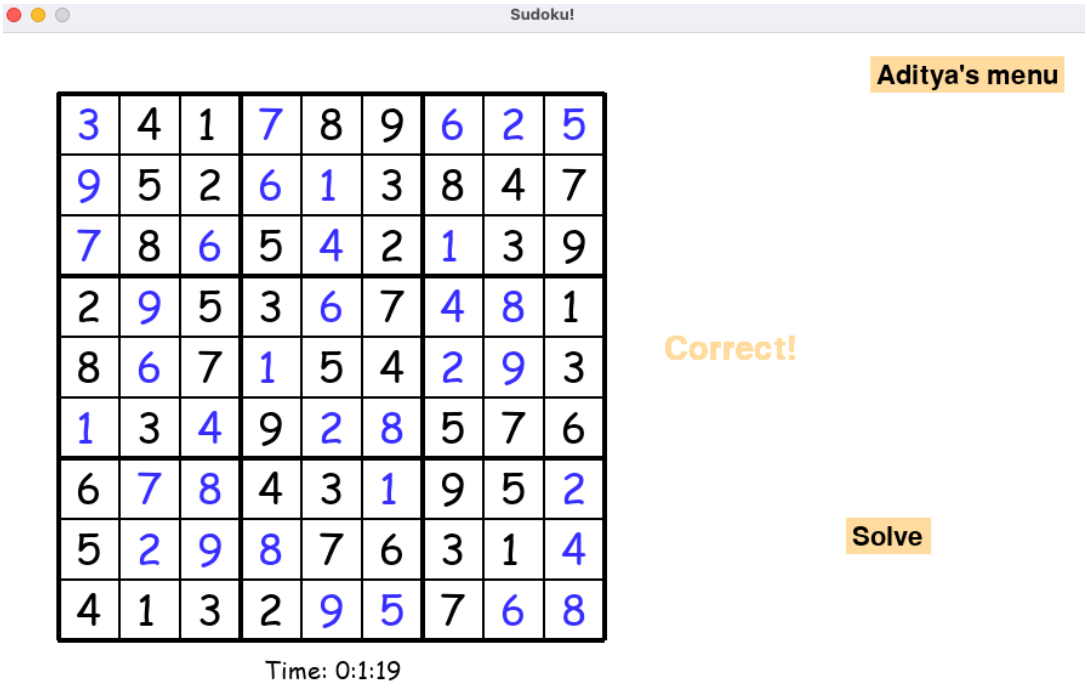
Erroneous 3: Pass

The erroneous data outputs an error as expected. Although the username is in the database, the password 'Pass#12345' does not match the password stored in the database: 'Pass#123'. The user is prompted to try again as the password is incorrect. The test passed.

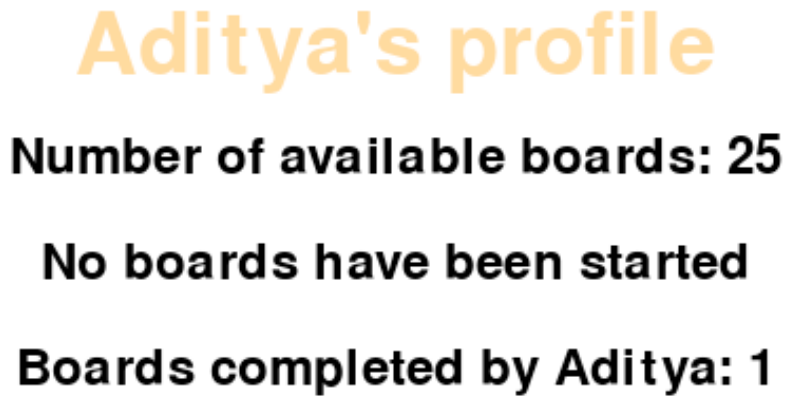


Test No	3
Objective	5a
Purpose	To ensure that completing a board without the algorithm shows that the board has been completed by the user in user menu
Description	A game is started and completed without using the algorithm. Then the user returns to their user menu
Input data	I will use board number 1 in this test. I will then solve the entire board without clicking the solve button.
Expected output	When the board is completed correctly, the board number must appear in the 'Boards completed by user' section
pass/fail	Pass

I solved the entire board correctly:

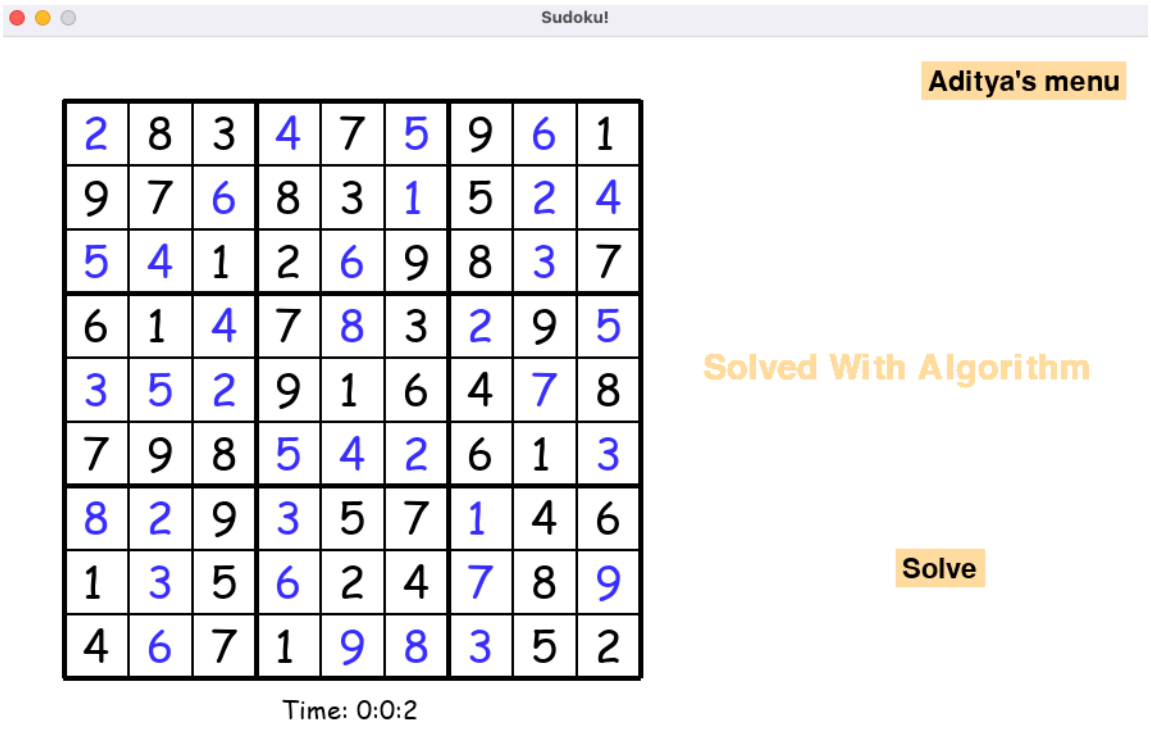


As expected, the board number was displayed in the 'Boards completed by Aditya' section after returning to the user menu. The test passed.

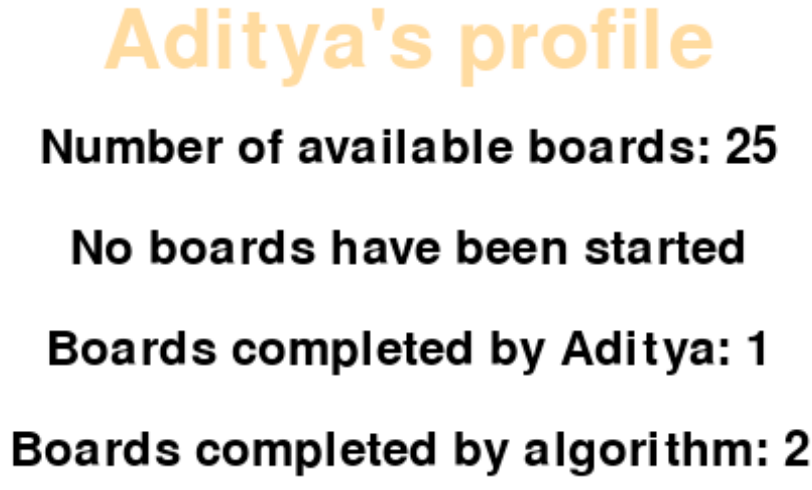


Test No	4
Objective	5b
Purpose	To ensure that completing a board with the algorithm shows that the board has been completed by the algorithm in user menu
Description	A game is started and completed by clicking the solve button. Then the user returns to their user menu
Input data	I will use board number 2 in this test. I will then click the solve button and go back to the user menu screen.
Expected output	When the board is completed using the algorithm, the board number must appear in the 'Boards completed by algorithm' section
pass/fail	Pass

After launching board number 2 and clicking solve, the 'Solved with algorithm' text is displayed.

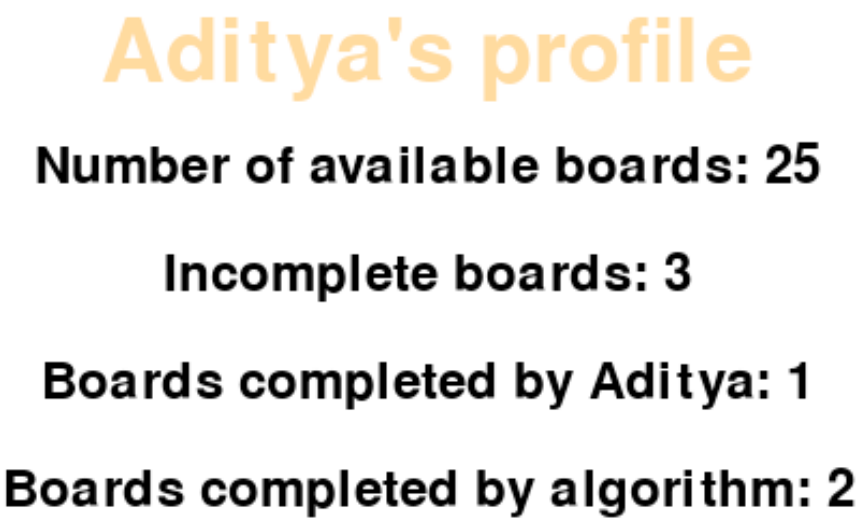


Upon returning to the user menu, the board number was displayed in the 'Boards completed by algorithm' section as expected. The test passed.



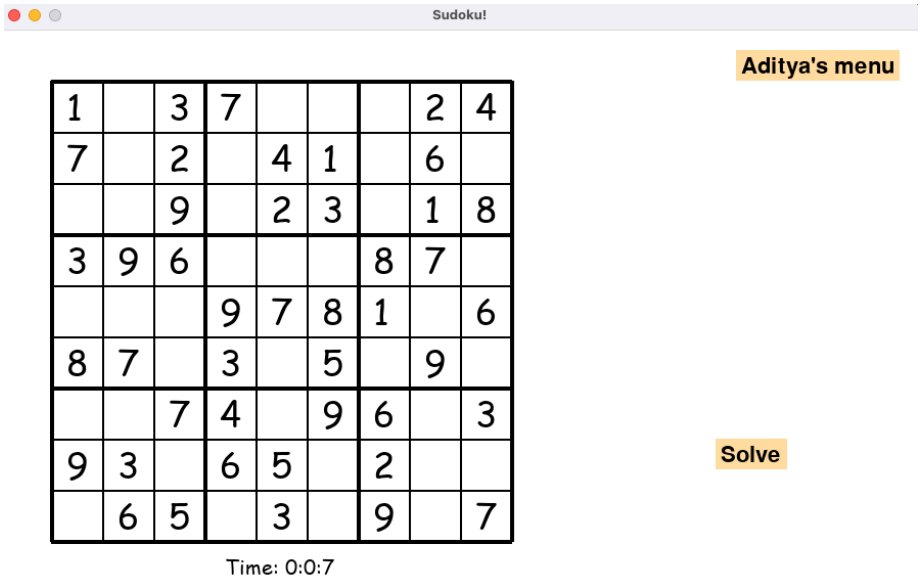
Test No	5
Objective	6
Purpose	To ensure that leaving a board incomplete and returning to the menu shows that the board has not been completed in user menu
Description	A game is started and the user returns back to the user menu without completing the board.
Input data	I will use board number 3 in this test. I will not edit the board and go back to the user menu.
Expected output	When the board is left incomplete, the board number must appear in the 'Incomplete boards' section
pass/fail	Pass

After starting a game with board number 3 and returning to the menu, the board number was displayed in the 'Incomplete boards' section as expected. The test passed.



Test No	6
Objective	7
Purpose	To ensure that the Launch board text box and button only allow valid data inputs.
Description	The input data is inserted in to the 'Enter board number to launch: ' text box and the launch button is clicked
Input data	**Typical:** Board number: '4' **Erroneous:** Board number '' **Extreme 1:** Board number: -1 **Extreme 2:** Board number 120
Expected output	The typical data must work as the board number exists. Erroneous 1 data must not work as the field is left empty. Extreme 1 data must not work as -1 is not a valid board number Extreme 2 data must not work as maximum number of boards is 25 and 120 > 25
pass/fail	**Typical**: pass **Erroneous**: FAIL **Extreme 1**: pass **Extreme 2**: pass

Typical: Pass



The typical data worked as expected.
When the board number '4' was entered into the Input box and the Launch button was clicked, the user was taken to the game screen with board number 4 to play.
The test passed.

Erroneous: Fail

The erroneous data did not work. Although it did not launch a board, the user was not informed that the board number is not valid. The test failed. I will need to edit my code to make sure that when the field is empty and the launch button is clicked, the user must be prompted to enter a board number before proceeding.

Previous code:

```
if LaunchNumber.userInput != '':
  if 0 < int(LaunchNumber.userInput) <= NumberOfBoards:
        if Launch.active:
        I HAVE HIDDEN THIS SECTION AS IT DOES NOT RELATE TO THIS FAIL

  else:  # if board number that user enters does not exist:
    font = pygame.font.SysFont('ComicSans MS', 20)
    text = font.render(f'Board number {LaunchNumber.userInput} does not exist.', True, (0, 0, 0))
    window.fill(BACKGROUND_COLOR, (0, 400, WIDTH, 29))  # clear previous text
    window.blit(text, (WIDTH // 2 - text.get_width() // 2, 400))
    # render text informing the user that the board number does not exist
    pygame.display.update()
```

In my current code, there is an if statement that only accounts for if the user input is not empty. I will need to add an else statement for when the user input is empty.

Updated code:

```
if Launch.active:
    if LaunchNumber.userInput != '':
        if 0 < int(LaunchNumber.userInput) <= NumberOfBoards:
                I HAVE HIDDEN THIS SECTION AS IT DOES NOT RELATE TO THIS FAIL

        else:  # if board number that user enters does not exist:
            font = pygame.font.SysFont('ComicSans MS', 20)
            text = font.render(f'Board number {LaunchNumber.userInput} does not exist.', True, (0, 0, 0))
            window.fill(BACKGROUND_COLOR, (0, 400, WIDTH, 29))  # clear previous text
            window.blit(text, (WIDTH // 2 - text.get_width() // 2,
                               400))  # render text informing the user that the board number does not exist
            pygame.display.update()

    elif LaunchNumber.userInput == '':
        font = pygame.font.SysFont('ComicSans MS', 20)
        text = font.render(f'Please enter a board number', True, (0, 0, 0))
        window.fill(BACKGROUND_COLOR, (0, 400, WIDTH, 29))  # clear previous text
        window.blit(text, (
        WIDTH // 2 - text.get_width() // 2, 400))  # render text alerting the user to fill the text box
        pygame.display.update()
```

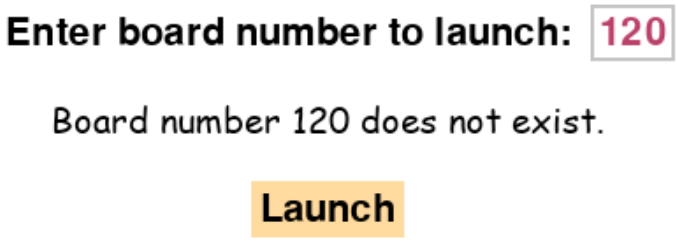
I have fixed my code so that whenever the user clicks the launch button without entering a number in the input box, they are alerted with 'Please enter a board number.' I have also changed the layout of the code so that these alerts only show up when the button is clicked by putting the if statement for the launch button outside the ones for the input box.

Extreme 1: Pass

The extreme data did not work as -1 is not a board number. This was expected as my program limits the inputs on this input box to be the numbers between 1 and 9. The negative sign cannot be inputted by the user.

Extreme 2: Pass

The extreme data outputs an error as expected. I only have 25 boards saved on the database and trying to launch a board number above 25 alerts the user that the board number does not exist.



Test No	7
Objective	11, 12
Purpose	To ensure that the board GUI accepts all the inputs from 1-9 and that these inputs can be deleted while fixed values cannot be deleted
Description	The input data is inserted on to the board. The input data is deleted
Input data	**Typical:** Numbers to input: 1-9 **Erroneous:** Numbers to input: '@', '#'… **Extreme:** Number to input: 10
Expected output	The typical data must work as the board should accept all numbers between 1 and 9 Erroneous data must not work as the board must only accept integers Extreme 1 data must not work as the board must only accept single digit numbers
pass/fail	**Typical**: pass **Erroneous**: pass **Extreme:** pass

Typical: Pass

The typical data worked as expected.
When the numbers 1-9 were inserted on the board, they did not produce any errors apart from clashes due to violating sudoku game rules.
The test passed.

1	1	8	2	3	4	5	6	7
	4		8	9	7			
5		3	9					
7	2			9		3		
	3			6			1	
		6		1			9	4
					4	2		7
			5				4	
						8		1

Erroneous: Pass

The erroneous data did not render on the board as expected. The code only accepts certain ASCII values like: 49-57 (numbers 1-9) and 48 (0). There is no change to the existing board. The test passed.

Extreme: Pass

The extreme data only render renders the first digit of the double-digit number '10' as it only registers single presses of keys. The test passed.

Test No	8
Objective	9a
Purpose	To ensure that starting a board and going back to the menu saves the board correctly.
Description	A game is started, and random numbers are inputted in a few cells. The menu button is clicked
Input data	I will use board number 5 in this test, and I will fill the first row completely.
Expected output	When the board is left incomplete and the user returns to the menu, the board must be saved. The user can continue it whenever.
pass/fail	Pass

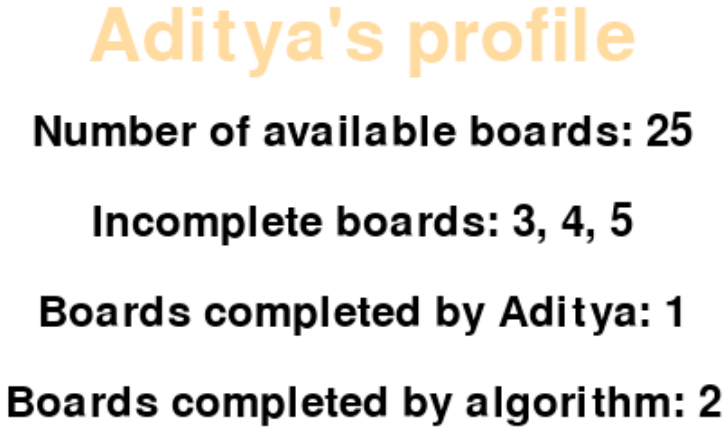
Using board number 5, I have filled the first row as shown:

1	2	8	3	4	4	7	9	6
7	5		9			8		1
	9	4	7	8	1			
				1	8	4		9
3		1	6		9		2	
9	4	6		5		1	3	
	7		4	6	2	3		5
4	1	5		9			7	
2			1				8	4

Time: 0:0:13

The first-row sequence is 128344796.

Upon clicking the menu button, we can see that board number 5 appears in the 'Incomplete boards' section. We can look at the saved board value on the database to see if the board has been saved successfully.



From the database picture above, we can see that the savedBoard value is 128344796, the same as the numbers we inputted during the game. The test has passed.

Test No	9
Objective	9b
Purpose	To ensure that starting a board and quitting the program saves the board.
Description	A game is started and random numbers are inputted in a few cells. The program is quit using the close window button
Input data	I will use board number 6 in this test, and I will fill the first row completely.
Expected output	When the board is left incomplete and the user quits the program and returns back to the menu, the board must be saved.
pass/fail	Pass

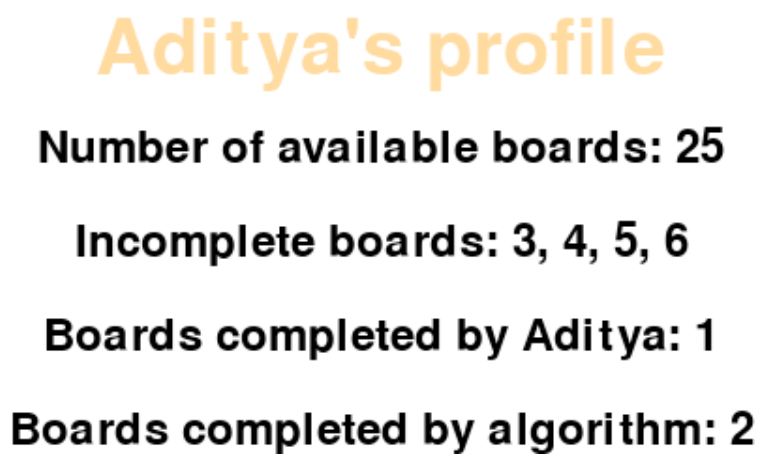
Using board number 6, I have filled the first row as shown:

7	9	5	1	4	8	2	3	4
2		1			9	4		7
	4		7	1			5	
		2			4	7		
			5	3	2			
		6	8			9		
	1			2	3		7	
5		4	1			6		3
			4	6		8	2	1

Time: 0:0:7

The first-row sequence is 795148234.

Upon quitting the game, we can see that the board is saved in the database as the first-row sequence is the same. The number is also displayed in the 'Incomplete boards' section in the user profile.
The test passed.

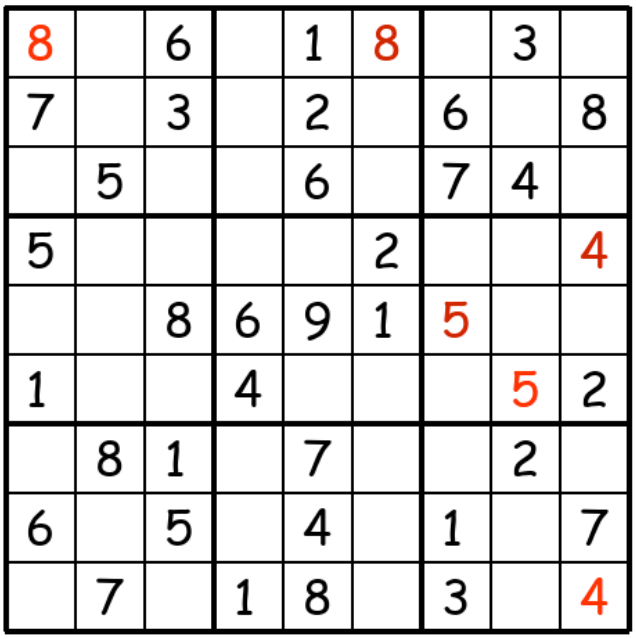


Test No	10
Objective	15
Purpose	To ensure that all clashes on a board are displayed in red and go away when the clash is resolved.
Description	A game is started and a few delibrate clashes are caused. Later, the clashes are resolved by changing the numbers that caused the clash.
Input data	I will be using board number 7 in this test. I will cause a clash in a row, column, and a 3x3 grid later, I will change these numbers so that clash is resolved
Expected output	When clashes re caused, all the numbers involved in the clash turn red when the clashes are resolved, all the numbers must go back to blue/black
pass/fail	Pass

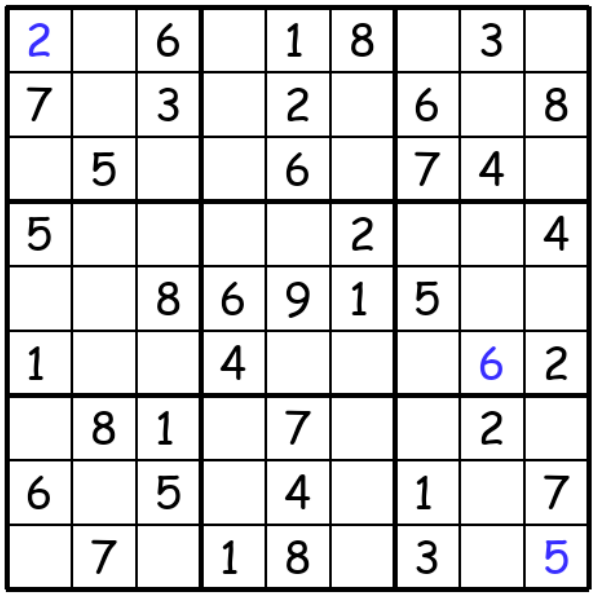
These are the clashes I have caused.
I have realized that differentiating between
Fixed numbers and user-inputted numbers during a
Clash is difficult as both the shades of red are similar
I will fix this by changing the color of the user
Inputted numbers during the clash to a lighter pinkish
Red color.

Previous RGB colors for clash:
Fixed: (128, 0, 0)
User: (255, 0, 0)

New RGB colors for clash:
Fixed: (200, 0, 0)
User: (255, 90, 90)

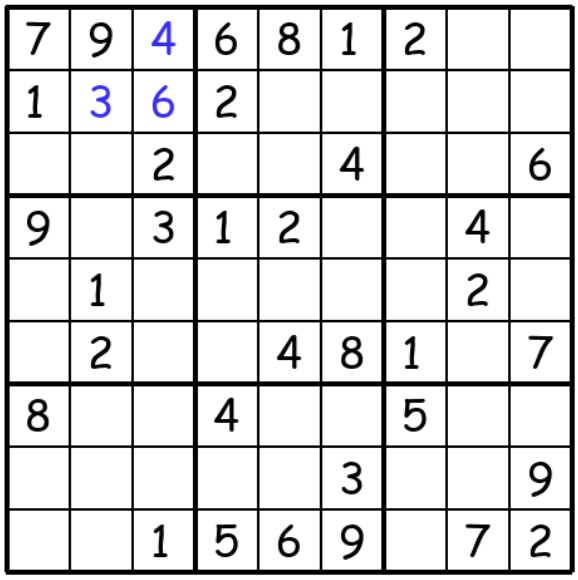


After changing the numbers that caused the clashes,
We can see that the numbers go back to blue and black
indicating that the clash has been resolved.
The test passed.



Test No	11
Objective	13
Purpose	To ensure that the user can only use 3 hints every game by clicking H on the keyboard
Description	A game is started and H is pressed 4 times on the keyboard.
Input data	letter H on keyboard. I will be using board number 8 in this test
Expected output	User clicks H on the keyboard 4 times to get hints. Once the user returns to menu and starts the game again, they must not be able to use any hints
pass/fail	FAIL

After clicking H 3 times on the cells, the correct answer for each cell was displayed correctly.



7	9	4	6	8	1	2		
1	3	6	2					
		2			4			6
9		3	1	2			4	
	1						2	
	2			4	8	1		7
8			4			5		
					3			9
		1	5	6	9		7	2

However, after going back to the user menu and returning to board number 8, I was able to use 3 more hints. I realized that this could be exploited for unlimited hints during a game. The test failed.

I will need to edit my code so that the number of hints used in a game is stored on the database so that the user can only use 3 hints on a board.

Updated code:

I need to update the database create statements as there is a new field in the BoardUserLink table:

```
db.execute(
  'CREATE TABLE IF NOT EXISTS "BoardUserLink" (\n'  # This command creates a BoardUserLink table if it does not already exist.
  '  "username" TEXT NOT NULL,\n'  # creates a username field that cannot be null
  '  "BoardNumber"  INTEGER NOT NULL,\n'  # creates BoardNumber field which cannot be null
  '  "Completed"    TEXT NOT NULL,\n'  # creates Completed field which cannot be null
  '  "Time" INTEGER,\n'  # Time is an integer and will be stored in seconds
  '  "SavedBoard"   TEXT,\n'  # creates SavedBoard field.
  '  "hintCount"    INTEGER,\n'  # creates  hintCount field
```

```
    ' PRIMARY KEY("username","BoardNumber"),\n'
    # username and BoardNumber make up a composite primary key
    ' FOREIGN KEY("username") REFERENCES "users"("username"),\n'
    # This makes username from  the users table a foreign key in the boardUserLink table.
    ' FOREIGN KEY("BoardNumber") REFERENCES "defaultBoards"("BoardNumber")\n' ')')
    # This makes BoardNumber from  the defaultBoards table a foreign key in the boardUserLink table.
```

I also need to update the SQL command for creating a new game, the hintCount when a new game starts must be set to 0 and increment by one every time a hint is used.

```
INSERT INTO "main"."BoardUserLink" ("username", "BoardNumber", "Completed", "Time", "hintCount")
VALUES ("{u}", {BoardNumber}, "False", 0, 0)
```

Clicking on H:

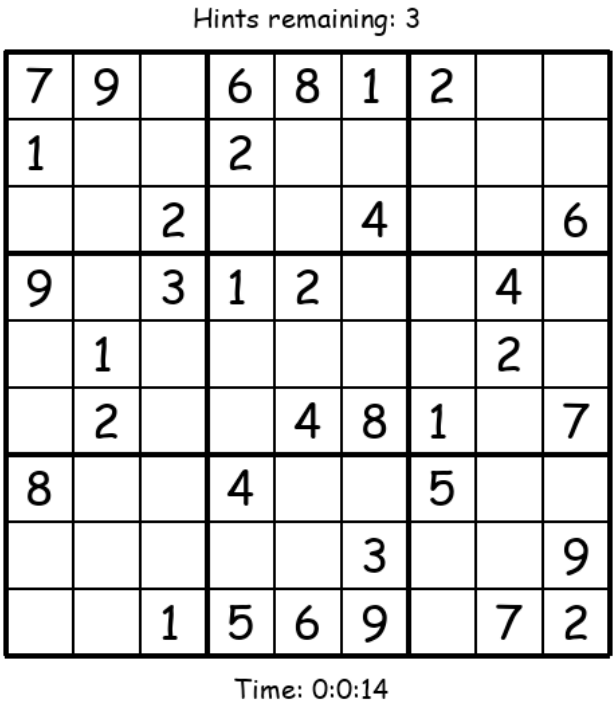
```
if event.key == 72 or event.key == 104: # 72 is the ASCII value for 'h' and 104 is the ASCII value for 'H'
    hintCount = db.execute(
        f"SELECT HintsUsed from 'main'.'BoardUserLink' WHERE username = '{u}' and BoardNumber =
{BoardNumber}").fetchone()[0]  # number of hints used per game

    if hintCount < 3:  # if a keypress is the letter h/H and the user has used less than 3 hints
        board[i][j] = solved_board[i][j]  # give the correct answer for that cell
        update_gui(board, og_board, 'half')
        hintCount += 1  # increment hint count by 1
        db.execute(f"UPDATE 'main'.'BoardUserLink' SET hintCount = {hintCount} WHERE username = '{u}' AND
                        BoardNumber = {BoardNumber}")  # update the hintCount by 1 on database
        conn.commit() # commit changes to database
        return
```

I also want my program to display the number of hints left while playing the game. I will update my code so that it displays the number of hints remaining. The following code is in the gameloop function.

```
hintCount = db.execute(f"SELECT hintCount from 'main'.'BoardUserLink' WHERE username = '{u}' and
                                BoardNumber = {BoardNumber}").fetchone()[0] # This is the number of hints the user used
text = font.render(f'Hints remaining: {3 - hintCount}', True, (0, 0, 0))  # Hints remaining is displayed.
window.fill(BACKGROUND_COLOR, (100, 10, 190+text.get_width() + 5, 29))  # removes previous text
window.blit(text, (190, 10))  # renders time taken text
pygame.display.update()  # updates display
```

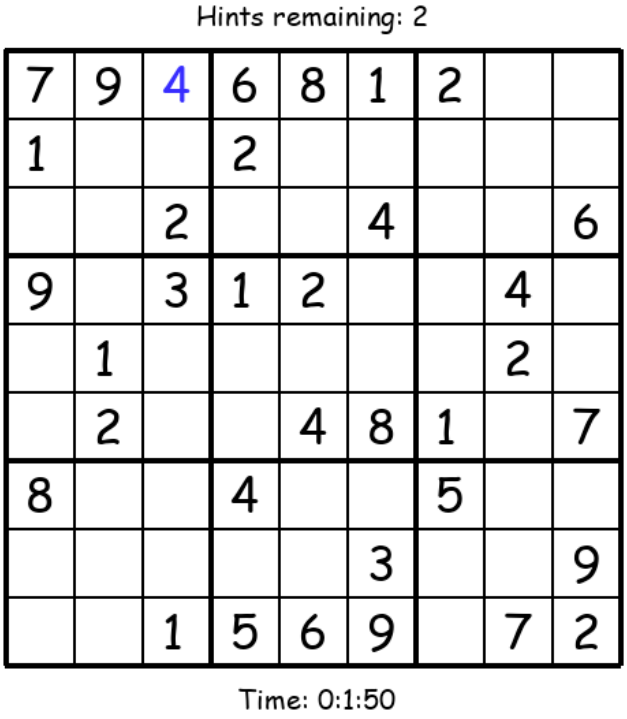
Updated display:

The number of hints a user can use on a board is displayed above the board.

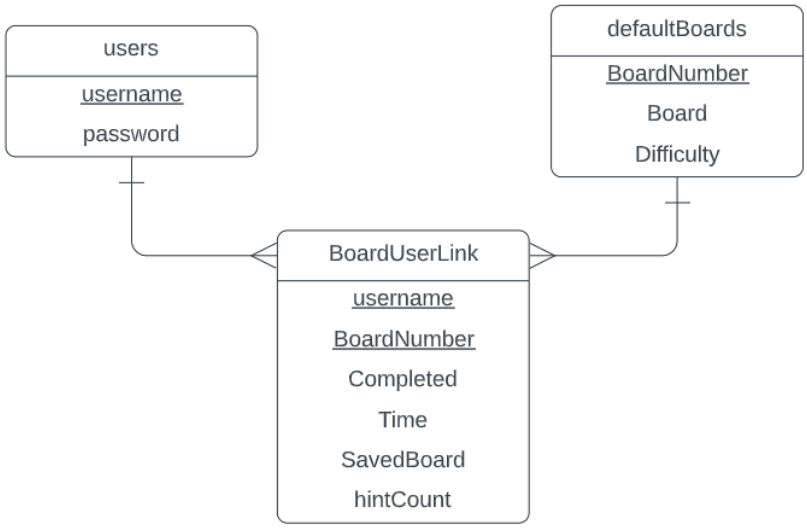


Using a hint will cause the number to decrease by 1. I will click H in the 3rd cell which should bring down the hints remaining counter to 2 and give the correct answer for that cell.
When the user goes back to the user menu and returns to the Board, the program still shows that only 2 hints are remaining.

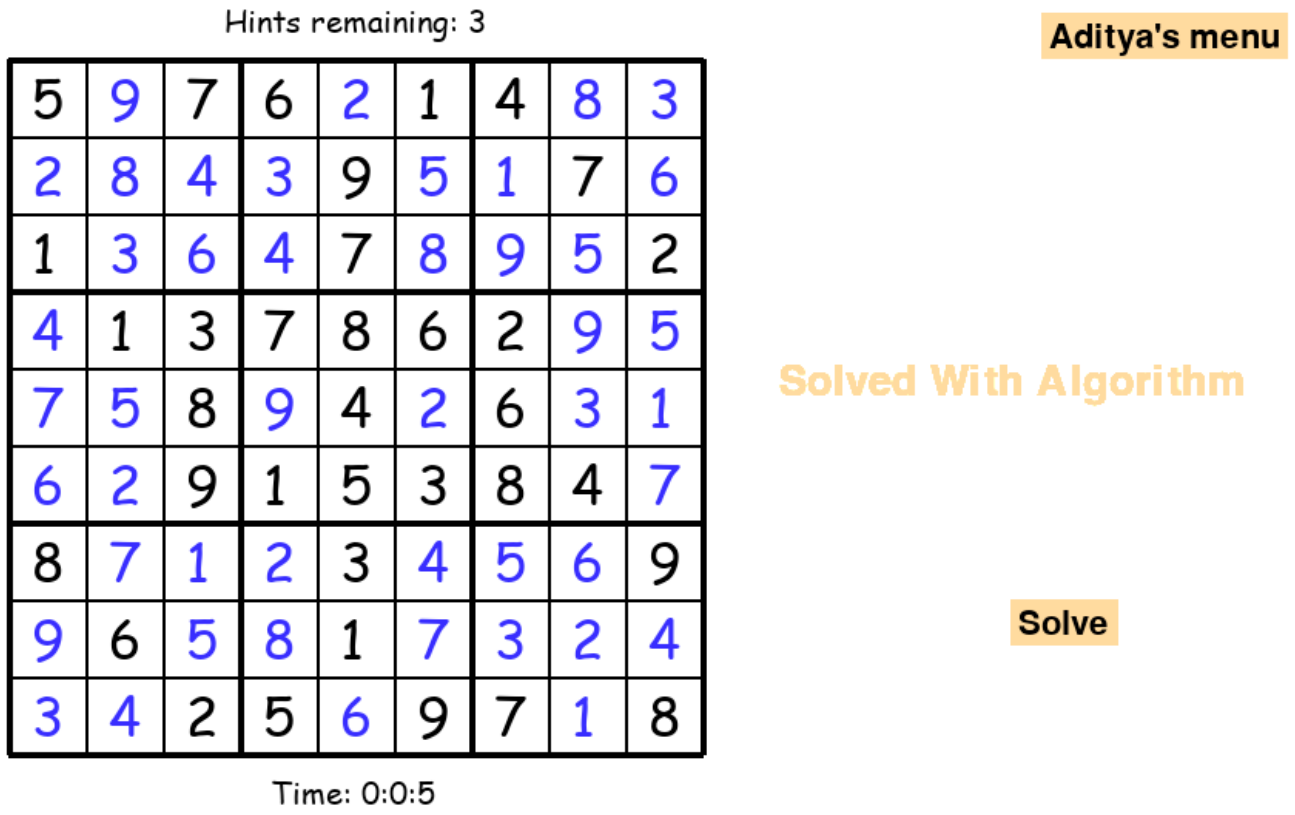
The test will pass now.



Updated ERD:

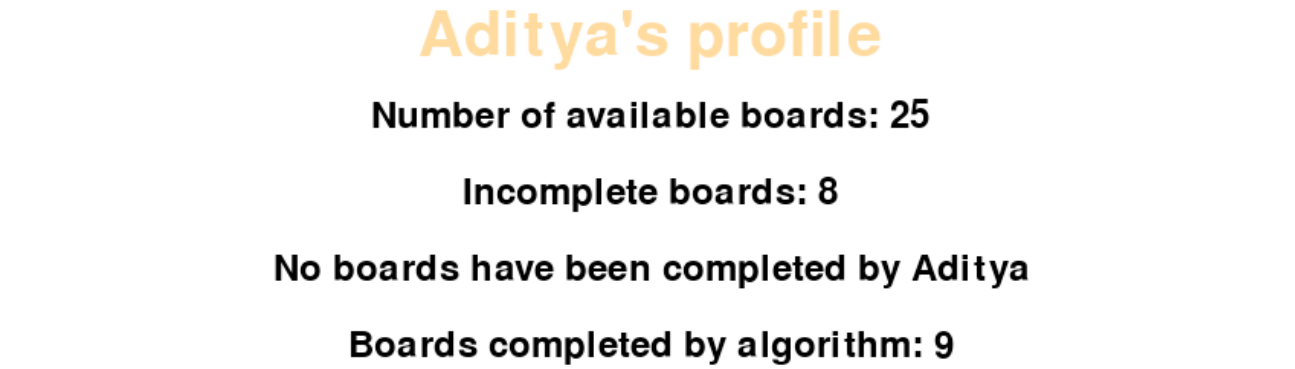


Test No	12
Objective	14
Purpose	To ensure that once the game has been solved by the algorithm, the user cannot edit the board.
Description	To avoid cheating, once the board has been solved by the algorithm, it should be uneditable.
Input data	Solve button is clicked and I try to edit the board. I will be using board number 9.
Expected output	User cannot edit the board after clicking on the solve button and that board number must appear in the 'Solved by algorithm' section in the user menu.
pass/fail	PASS



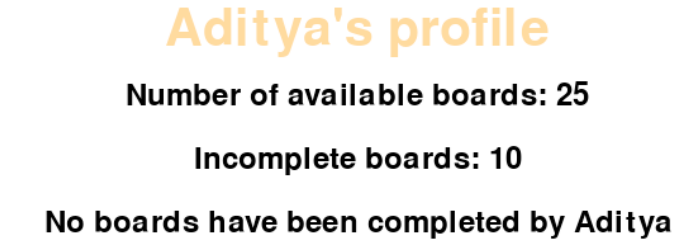
Once the solve button is clicked, the board is unalterable as expected. The time has also frozen indicating that the board is solved. The test passed.

Once the user goes back to their user menu, board number 9 is displayed in the 'Solved by algorithm' section.



Test No	13
Objective	17
Purpose	To ensure that the time I saved when a user does not complete a board. When the user returns to the board, the time just resume. The time must stop counting when the board has been completed either by the user or the algorithm.
Description	For players to be able to track their progress, the time taken must be measured accurately and must be saved even when the board is completed.
Input data	I will play the board for 30 seconds and return back to the menu, wait for a minute and return back to the board. I will then click the solve button. I will use board number 10 for this test.
Expected output	When the user returns to the usermenu/quits the game, the time taken is saved and resumed once the user returns to the board. When the solve button is clicked, the timer must stop.
pass/fail	PASS

After launching board number 10 and trying to solve the board for 30 seconds and returning to the usermenu, I found that board number 10 was in the 'Incomplete boards' section and the time I played for was saved on the database.

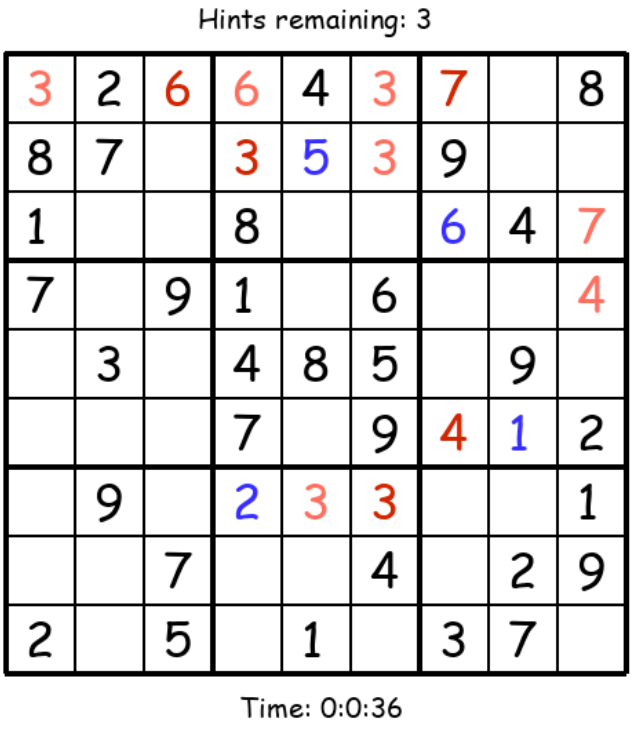


	username	BoardNumber	Completed	Time	SavedBoard	hintCount
	Filter	Filter	Filter	Filter	Filter	Filter
1	Aditya	10	False	30	32664370...	0

30s was saved in the time field as expected.

After waiting for a minute, I launched board number 10 again to see that the timer started from 30 seconds as expected. The timer says 36 seconds as I took 6 seconds to take the screenshot.

The test passed.



Test No	13
Objective	NOT AN OBJECTIVE
Purpose	To ensure that when a board is unsolvable, the program informs the user that the board is unsolvable and does not let the user play on it.
Description	For players to be able to track their progress, the time taken must be measured accurately and must be saved even when the board is completed.
Input data	I will deliberately insert an unsolvable board in the database and launch that board in the program.
Expected output	When the user launches the unsolvable board, the program must not launch the board and say 'This board is unsolvable'
pass/fail	FAIL

My end user: Akshay recently asked me to implement a method of adding more boards. This can easily be done by creating a new row and adding the board in the defaultBoards table. However, I do not have any solution If Akshay accidentally adds an unsolvable board.
I need to implement a feature so that invalid boards cannot be played on.
After some testing, I concluded that my program currently crashes when it encounters an invalid board.
This should not be a difficult fix as all I have to do is:
If the board cannot be solved by the algorithm, the board is not launched and deleted from the database.

Edited code:

```
solved_board = sudoku_solver.BT([[og_board[x][y] for y in range(len(board[0]))] for x in range(len(board))])
# solved_board gives a solved board if it can be solved and False if it cannot be solved
while True:
    if solved_board != False: # if board can be solved:
                I HAVE HIDDEN THIS SECTION AS IT IS NOT RELATED TO THIS TEST
    else: # if the board cannot be solved
        font = pygame.font.SysFont('Soria', 60)  # font size is 60
        text = font.render(f'Board is unsolvable', True, (0, 0, 0))  # initailises the text
        window.fill((255,255,255))  # fills the entire screen white
        window.blit(text, (WIDTH//2 - text.get_width()//2, HEIGHT//2 - text.get_height()//2))
        # renders the text in the middle of the screen
        db.execute(f"DELETE FROM 'main'.'BoardUserLink' WHERE username = '{u}' AND boardNumber =
                        {BoardNumber}")  # deletes the game from the link table
        db.execute(f"DELETE FROM 'main'.'defaultBoards' WHERE boardNumber = {BoardNumber}")
        # deletes the invalid boards from the default boards table so it can never be played again
        conn.commit()  # commit changes to database
        pygame.display.update()  # updates display
        time.sleep(5)  waits 5 seconds on the 'Board cannot be solved' screen and returns back to the user menu.
        usermenu(u)
```

Now whenever an invalid board is launched, the user is alerted that the board is unsolvable, and the board is deleted from the database so it cannot be run again.

Evaluation

5.1 Comparing performance against the objectives

Objective	How objective was met
Login	
1. Allow users to enter a Username and a password using 2 input boxes	FULLY MET As I am using PyCharm and it does not come with a built-in input box, I had to create these input boxes using object-oriented programming. 2 instances of the input box class were created and rendered on the board.
2. Allow users to click on a register button to create a new account 18. Store usernames and passwords on a database using hashing on the passwords for security.	FULLY MET The register button has a register function which is run whenever the button is clicked. The button only functions when both the username and password input boxes are filled, and username is unique/does not already exist. The password is required to be complicated enough which is validated using regular expressions which translate to: - At least 8 characters long - At least 1 capital letter - At least 1 small letter - At least 1 symbol such as # or $ The passwords are stored securely using a hashing algorithm called djb2 and a salt which is added to the end of a password before hashing. This salt greatly improves the security of the system. Hashing can be seen as a sort of one-way encryption. Once a password has been hashed with a good hashing algorithm, it can be very hard to find the original password.
3. Allow users to click on a login button to access their menu if credentials are correct	FULLY MET The login button has its own login function which is run whenever the button is clicked. The button only functions when both the input boxes are filled. If the username does not exist, the user is alerted. If the password does not match, the user is alerted. As the salt is stored in the database, it can be selected and added to the password before hashing to see if it matches the hashed password stored on the database. If it does, then the login is successful.

Objective	How objective was met
User Menu	
4. Allow user to log out when the log out button is clicked. User will be taken to the Main menu page.	FULLY MET the log out button is an instance of the class Button. When it is clicked, the screen is filled white, and the main function is run. To access the next screens, the user needs to log back in successfully. Once logged out, it. Is impossible to access the user menu screen without logging back in.
5. Allow players to see what boards are completed.	FULLY MET A board can be completed in 2 ways: - By the user - By the solving algorithm When a board is solved by the user, 'True' is saved in the database. In the usermenu, the program runs an SQL query to select all the board numbers where completed = True. These board numbers are displayed in the 'Boards completed by user' section When a board is solved by the solving algorithm, 'SolvedByAlgo' is saved in the database. In the usermenu, the program runs an SQL query to select all the boards where completed = 'SolvedByAlgo'. These board numbers are displayed in the 'Completed by algorithm' section.
6. Allow players to see what boards they have started but haven't completed.	FULLY MET When a board is started but hasn't been completed by the user, 'False' is stored in the Completed field in the database. In the usermenu, the program runs an SQL query to select all the board numbers where Completed = False. These board numbers are displayed under the 'Incomplete boards' section
7. Allow players to select a board number. 8. When the launch button is clicked, launch the game with that BoardNumber.	FULLY MET The Launch board input box is an instance of the InputBox class while the launch button is an instance of the button class. When a user inputs a number between 1 and the number of available boards in the launch board input box and the launch button is clicked, the user is taken to the game screen. If the input box has a board number greater than the number of available boards, the user is alerted that the board number does not exist. If the input box is empty and the user clicks the launch button, the user is alerted to fill the input box before launching.

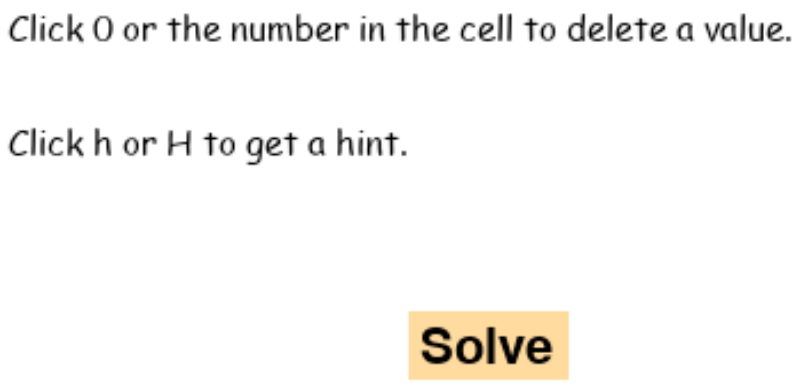
Objective	How objective was met
Game screen	
9. Allow users to save their boards mid game.	PARTIALLY MET There are 3 instances where the board should be saved: - Clicking the user menu - Quitting the program - Program crashes Clicking the user menu saves the board in the database which can be resumed once the user returns to the board Quitting the program also saves the board in the database which can be resumed once the user returns to the board Any progress that the user made on the board previously is can be seen when launched again. As programs can crash unexpectedly and due to many reasons, I have been unable to program it so that the board is saved during any crash. However, I have not noticed any crashes apart from when I am rerunning the Python file while already in the game. This saves the board as NULL on the database and I don't know how to save the board before rerunning the python file. Apart from rerunning the Python file while already in a game, there are no crashes that I know of, and the user's experience should not be affected.
10. Generate an interactive Board and fill the starting values (in black font) of the board based on the puzzle the user picked in the user menu.	FULLY MET When a valid board number is launched, the program runs the update_gui function which uses 2 for loops to create 10 horizontal and vertical lines which form a 9x9 sudoku grid. Every 3 lines, the lines have an increased thickness so that 9 3x3 grids are formed inside the 9x9 grid. The program reads the number 0 in the board 2d list as an empty cell and displays all the other elements in the 2d list as fixed values in black font unless it is involved in a clash in which case it is in a dark red font. These values cannot be edited by the user.

11. Allow user to input values onto the board.	FULLY MET When a user clicks the keyboard buttons 1-9 on an editable cell, the numbers appear in the GUI sudoku board. If they produce a clash, the number is displayed in a light red font. If not, it is displayed in a blue font.
12. Allow users to delete USER ENTERED numbers on the board by pressing 0 on the keyboard.	FULLY MET When a user clicks the keyboard button 0 or the number that is already in the cell, the number in the 2d array is set to 0 which denotes an empty cell. The GUI is updated, and the value in the cell has been removed.
13. Create a hint system for sudoku	FULLY MET When the user clicks H/h on the keyboard after clicking on a cell, the correct value for that cell is displayed. The player only gets to use 3 hints per game and the hints remaining is displayed above the GUI board. The number of hints used per game is saved on the database.
14. Create a Solve button.	FULLY MET When the solve button is clicked, the entire board is updated with the solved board. The board is now unalterable to prevent cheating. If the solve button is clicked, the user will see that board number in the 'Boards solved by algorithm' section in their user menu.
15. Create a system to check for validity using the sudoku game rules	FULLY MET There are 3 major rules in sudoku and when a rule is broken, a clash is produced between 2 or more numbers. The user is alerted that the numbers are invalid as the program highlights both the values in shades of red. When the user resolves the clashes, all the numbers involved in the clash go back to their original color.
16. When the GUI board matches the answer for that board, the user has successfully completed the puzzle and is prompted with the text "Correct!"	FULLY MET When the board matches the solved board and the user has not clicked on the solve button, "Correct!" is displayed on the screen to indicate that the user has completed the board.
17. Save the time taken to solve puzzle after user has correctly completed puzzle.	FULLY MET The time taken is stored whenever the user completes the board either by themselves or with the use of the solve algorithm. The times can be seen in seconds on the database. Additionally, the time taken is also stored whenever an incomplete board is saved by returning to the user menu or quitting the program. The timer will resume counting once the user returns to the same board again.

5.2 Feedback from end-user: Akshay

Objective	Feedback
Login	
18. Allow users to enter a Username and a password using 2 input boxes	I was able to use the user and password fields without any issues.
19. Allow users to click on a register button to create a new account 19. Store usernames and passwords on a database using hashing on the passwords for security.	I was able to create accounts easily and the fact that the passwords are saved on the database without having to worry about people finding the passwords is great.
20. Allow users to click on a login button to access their menu if credentials are correct	I was able to log into the accounts I created without difficulties.
Objective	Feedback
User Menu	
21. Allow users to log out when the log out button is clicked. The user will be taken to the main menu page.	I was able to log out of an account by clicking the log out button.
22. Allow players to see what boards are completed.	I was able to see what boards are completed by clicking the solve button or by solving them myself and going back to the user menu.
23. Allow players to see what boards they have started but haven't completed.	I was able to see what boards I have started but left incomplete in the user menu.
24. Allow players to select a board number. 25. When the launch button is clicked, launch the game with that BoardNumber.	I was able to select board numbers and launch the board numbers I wanted without any errors.

Objective	Feedback
Game screen	
26. Allow users to save their boards mid game.	I had no issues with the board not saving.
27. Generate an interactive Board and fill the starting values (in black font) of the board based on the puzzle the user picked in the user menu.	The board was generated just how I wanted it to however it is hard to differentiate between fixed numbers and my numbers when they both turn red during clashes *I am aware of this problem and have tried to fix it using different colors, but I can't seem to find shades of red that are easily differentiable without changing the color scheme. However, I have made one of the shades a bit lighter.*
28. Allow user to input values onto the board.	I was able to input values I wanted easily.

29. Allow users to delete USER ENTERED numbers on the board by pressing 0 on the keyboard.
 	
30. Create a hint system for sudoku
	I was not able to delete values or use hints because I did not know how to. There should be a tutorial on the screen.
	
	I will add a small tutorial on how to use the delete and hint functionality on the game screen.
	
	Updated code:
	
	`font = pygame.font.SysFont('Comic SansMs', 15)`
	`text = font.render("Click 0 or the number in the cell to delete a value.", True, (0,0,0))`
	`window.blit(text, (525, HEIGHT//2 - text.get_height()//2))`
	`text = font.render("Click h or H to get a hint.", True, (0,0,0))`
	`window.blit(text, (525, HEIGHT//2 - text.get_height()//2 + 50))  # renders text`
	`pygame.display.update()  # updates display`
	
	Updated GUI:
	
	 

31. Create a Solve button.	I could use the solve button without any difficulty
32. Create a system to check for validity using the sudoku game rules	Apart from the hard to differentiate colors, the validity system is well done.
33. When the GUI board matches the answer for that board, the user has successfully completed the puzzle and is prompted with the text “Correct!”	No issues.
34. Save the time taken to solve puzzle after user has correctly completed puzzle.	Time taken is saved as needed.

Feedback from Akshay:

Overall, the program is very easy to use and hits all my requirements for a sudoku app. The game screen looks better than I expected, and I can see the red highlighting clashes feature being very useful. Being able to view the progression of users allows them to improve by focusing on reducing the time taken. I would appreciate a tutorial on how to play sudoku so that new players can get started on your program rather than learning how to play online.

Things I would change if I were to do it again:

The main thing I regret is not splitting up my code into multiple files. I started with the Elements file and the sudoku solver file but after that, all my code was in 1 file. This made it messy and hard to work on. When I realized that, it was too late as it would take a lot of time to split the code into multiple files and link it all up.

Another thing I regret was poor organization in the user menu function. I should have made a function to print out text on the GUI. This code can get messy as seen in the user menu function.

```
font = pygame.font.SysFont('Soria', 35)
if len(NBoardsCompleted) == 0:  # If no boards have been completed by user:
    NBoardsCompletedText = font.render(f"No boards have been completed by {u}", True, (0, 0, 0))
    window.blit(NBoardsCompletedText, (WIDTH // 2 - NBoardsCompletedText.get_width() // 2, 170))

else:  # If boards have been completed by user:
    NBoardsCompleted = [x[0] for x in
                        NBoardsCompleted]  # make a list of the first element from each tuple in NBoardsCompleted
    NBoardsCompleted = map(str, NBoardsCompleted)  # make every value in the list a string
    NBoardsCompletedText = font.render(f"Boards completed by {u}: {', '.join(NBoardsCompleted)}", True,
                                       (0, 0, 0))  # render all the completed boards
    window.blit(NBoardsCompletedText, (WIDTH // 2 - NBoardsCompletedText.get_width() // 2, 170))

NBoardsIncomplete = list(
    db.execute(
        f"SELECT BoardNumber FROM BoardUserLink WHERE username = '{u}' AND Completed = 'False'").fetchall())  # select
if len(NBoardsIncomplete) == 0:  # If no boards have been started:
    NBoardsIncompleteText = font.render(f"No boards have been started", True, (0, 0, 0))
    window.blit(NBoardsIncompleteText, (WIDTH // 2 - NBoardsIncompleteText.get_width() // 2, 120))

else:
    NBoardsIncomplete = [x[0] for x in
                         NBoardsIncomplete]  # make a list of the first element from each tuple in NBoardsIncomplete
    NBoardsIncomplete = map(str, NBoardsIncomplete)  # make every value in the list a string
    NBoardsIncompleteText = font.render(f"Incomplete boards: {', '.join(NBoardsIncomplete)}", True, (0, 0, 0))
    window.blit(NBoardsIncompleteText, (WIDTH // 2 - NBoardsIncompleteText.get_width() // 2, 120))
```