Game: Sudoku

T1 Planning and Design based on Mathematical and Declarative Programming techniques:

The Sudoku game developed in Python with a graphical user interface (GUI) uses both mathematical principles and software engineering principles. This section discusses the planning and design aspects of the game, focusing on how these principles and techniques are integrated to create a functional and efficient Sudoku solver.

Mathematical Structures:

Following mathematical structures were used:

- **Set Theory:** The Sudoku puzzle relies heavily on the principles of set theory. Each row, column, and 3x3 subgrid must contain all numbers from 1 to 9 without repetition. The best function uses sets to determine possible values for a given cell by excluding values already present in the corresponding row, column, and subgrid.
- **Logic and Constraints:** The game enforces logical constraints to ensure the validity of the puzzle. The valid function checks whether placing a number in a specific position adheres to Sudoku rules. These constraints are checked during both user interactions and the automated solving process.
- **Combinatorics:** Sudoku puzzles involve combinatorial principles to determine the arrangement of numbers. The solver explores different combinations to find a valid solution, making use of the backtracking technique to efficiently navigate through potential configurations.
- **Optimization:** The find_empty function employs a heuristic to improve the efficiency of the backtracking algorithm. By selecting the empty cell with the fewest possible values, it reduces the number of recursive calls needed to find a solution.

Declarative Programming Techniques:

Following declarative programming techniques were used:

Validation Functions: The valid function in the Sudoku game defines the constraints for a valid move in the grid. Instead of detailing how to enforce these constraints, it declaratively specifies what makes a move valid.

State Representation: The Sudoku board is represented as a list of lists (a matrix), which is a declarative way to model the game state. Each cell in the matrix can be accessed and updated based on the rules defined, without needing to manage the underlying memory explicitly.

Recursion: The backtracking algorithm used in the solve and solve_gui methods is inherently declarative. It defines the problem (finding a solution to the Sudoku puzzle) in terms of smaller subproblems (solving the puzzle with one less cell to fill). The recursion handles the control flow implicitly.

Software Engineering Principles

In the making of this game, following software engineering principles were used:

- **Modularity and Code Organization:** The code is organized into classes (Grid and Cube), each responsible for different functionalities of the game. This modularity makes the code more readable, maintainable, and scalable. The Grid class handles the overall Sudoku grid, including drawing the grid, updating the internal model, and solving the puzzle.
- **Encapsulation:** Each class encapsulates its properties and methods, ensuring that the internal state is not directly accessible from outside the class. For instance, the Cube class manages its own value and temporary state, while the Grid class manages the collection of Cube instances and the game logic.
- **User Interaction:** The game uses the Pygame library to create a graphical interface, allowing users to interact with the game through mouse clicks and keyboard inputs. The select, click, and clear methods in the Grid class handle user interactions, providing feedback and updates to the GUI in real-time.
- **Algorithm Design:** The Sudoku solver implements a backtracking algorithm, a classic example of a recursive algorithm used to solve constraint satisfaction problems. The solve and solve_gui methods in the Grid class use this algorithm to find a solution to the puzzle, either automatically or with graphical updates for the user.

The Sudoku game demonstrates the integration of mathematical and declarative programming techniques within the framework of software engineering principles. The modular design, encapsulation, user interaction handling, and efficient solving algorithms collectively contribute to an interactive Sudoku game. The use of set theory,

logical constraints, combinatorics, and optimization principles shows that the mathematical foundation is behind the game's mechanics.

T2 Program Implementation based on Declarative Programming tools and techniques.

Source Code:

```
'''

Sudoku Game:
Programming Language: Python
Interface: GUI based
Student Name: Aisha Abdi
Student ID: 22160571

'''

import pygame
import time
pygame.font.init()

# Class representing the Sudoku grid
class Grid:
    # Initial Sudoku board
    board = [
        [7, 8, 0, 4, 0, 0, 1, 2, 0],
        [6, 0, 0, 0, 7, 5, 0, 0, 9],
        [0, 0, 0, 6, 0, 1, 0, 7, 8],
        [0, 0, 7, 0, 4, 0, 2, 6, 0],
        [0, 0, 1, 0, 5, 0, 9, 3, 0],
        [9, 0, 4, 0, 6, 0, 0, 0, 5],
        [0, 7, 0, 3, 0, 0, 0, 1, 2],
        [1, 2, 0, 0, 0, 7, 4, 0, 0],
        [0, 4, 9, 2, 0, 6, 0, 0, 7]
    ]

    def __init__(self, rows, cols, width, height, win):
        self.rows = rows
        self.cols = cols
        self.cubes = [[Cube(self.board[i][j], i, j, width, height) for j in
range(cols)] for i in range(rows)]
        self.width = width
        self.height = height
```

```
        self.model = None
        self.update_model()
        self.selected = None
        self.win = win

    # Update internal model of the Sudoku board
    def update_model(self):
        self.model = [[self.cubes[i][j].value for j in range(self.cols)] for i
in range(self.rows)]

    # Place a number on the Sudoku grid
    def place(self, val):
        row, col = self.selected
        if self.cubes[row][col].value == 0:
            self.cubes[row][col].set(val)
            self.update_model()

            # Check if the placement is valid and solve the puzzle
            if valid(self.model, val, (row,col)) and self.solve():
                return True
            else:
                self.cubes[row][col].set(0)
                self.cubes[row][col].set_temp(0)
                self.update_model()
                return False

    # Sketch a number on the Sudoku grid
    def sketch(self, val):
        row, col = self.selected
        self.cubes[row][col].set_temp(val)

    # Draw the Sudoku grid
    def draw(self):
        # Draw Grid Lines
        gap = self.width / 9
        for i in range(self.rows+1):
            if i % 3 == 0 and i != 0:
                thick = 4
            else:
                thick = 1
            pygame.draw.line(self.win, (0,0,0), (0, i*gap), (self.width,
i*gap), thick)
            pygame.draw.line(self.win, (0, 0, 0), (i * gap, 0), (i * gap,
self.height), thick)

        # Draw Cubes
        for i in range(self.rows):
            for j in range(self.cols):
```

```
            self.cubes[i][j].draw(self.win)

    # Select a cell on the Sudoku grid
    def select(self, row, col):
        # Reset all other
        for i in range(self.rows):
            for j in range(self.cols):
                self.cubes[i][j].selected = False

        self.cubes[row][col].selected = True
        self.selected = (row, col)

    # Clear the selected cell
    def clear(self):
        row, col = self.selected
        if self.cubes[row][col].value == 0:
            self.cubes[row][col].set_temp(0)

    # Detect mouse click and return cell coordinates
    def click(self, pos):
        if pos[0] < self.width and pos[1] < self.height:
            gap = self.width / 9
            x = pos[0] // gap
            y = pos[1] // gap
            return (int(y),int(x))
        else:
            return None

    # Check if the Sudoku puzzle is finished
    def is_finished(self):
        for i in range(self.rows):
            for j in range(self.cols):
                if self.cubes[i][j].value == 0:
                    return False
        return True

    # Solve the Sudoku puzzle
    def solve(self):
        li,pos,flag = find_empty(self.model)

        if flag == 0:
            return True

        if not pos:
            return False

        row,col = pos
```

```
        for i in li:
            if valid(self.model, i, (row, col)):
                self.model[row][col] = i

                if self.solve():
                    return True

                self.model[row][col] = 0

        return False

    # Solve the Sudoku puzzle with graphical interface
    def solve_gui(self):
        li,pos,flag = find_empty(self.model)

        if flag == 0:
            return True

        if not pos:
            return False

        row,col = pos

        for i in li:
            if valid(self.model, i, (row, col)):
                self.model[row][col] = i
                self.cubes[row][col].set(i)
                self.cubes[row][col].draw_change(self.win, True)
                self.update_model()
                pygame.display.update()
                pygame.time.delay(100)

                if self.solve_gui():
                    return True

                self.model[row][col] = 0
                self.cubes[row][col].set(0)
                self.update_model()
                self.cubes[row][col].draw_change(self.win, False)
                pygame.display.update()
                pygame.time.delay(100)

        return False


class Cube:
    rows = 9
    cols = 9
```

```
    def __init__(self, value, row, col, width, height):
        self.value = value
        self.temp = 0
        self.row = row
        self.col = col
        self.width = width
        self.height = height
        self.selected = False

    def draw(self, win):
        fnt = pygame.font.SysFont("comicsans", 30)

        gap = self.width / 9
        x = self.col * gap
        y = self.row * gap

        if self.temp != 0 and self.value == 0:
            text = fnt.render(str(self.temp), 1, (128,128,128))
            win.blit(text, (x+5, y+5))
        elif not(self.value == 0):
            text = fnt.render(str(self.value), 1, (0, 0, 0))
            win.blit(text, (x + (gap/2 - text.get_width()/2), y + (gap/2 -
text.get_height()/2)))

        if self.selected:
            pygame.draw.rect(win, (255,0,0), (x,y, gap ,gap), 3)

    def draw_change(self, win, g=True):
        fnt = pygame.font.SysFont("comicsans", 40)

        gap = self.width / 9
        x = self.col * gap
        y = self.row * gap

        pygame.draw.rect(win, (255, 255, 255), (x, y, gap, gap), 0)

        text = fnt.render(str(self.value), 1, (0, 0, 0))
        win.blit(text, (x + (gap / 2 - text.get_width() / 2), y + (gap / 2 -
text.get_height() / 2)))
        if g:
            pygame.draw.rect(win, (0, 255, 0), (x, y, gap, gap), 3)
        else:
            pygame.draw.rect(win, (255, 0, 0), (x, y, gap, gap), 3)

    def set(self, val):
        self.value = val

    def set_temp(self, val):
```

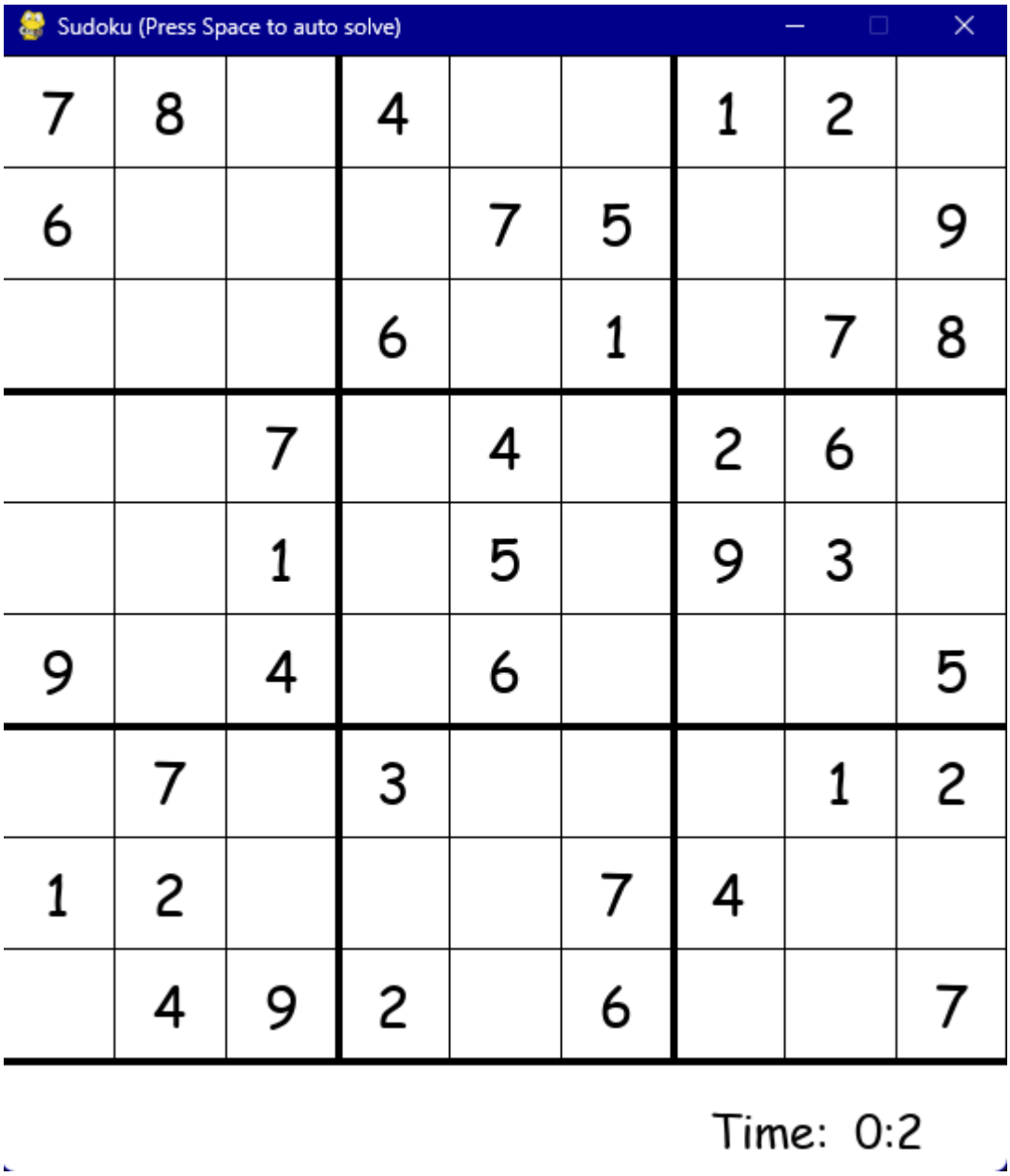
```
        self.temp = val


# Set of all possible values
full = {0,1,2,3,4,5,6,7,8,9}

# Find the best value for a cell
def best(board,i,j):
    excluded = {0}
    for row in range(len(board)):
        excluded.add(board[row][j])

    for col in range(len(board)):
        excluded.add(board[i][col])

    i -= i%3
    j -= j%3

    for row in range(int(len(board)/3)):
        for col in range(int(len(board)/3)):
            excluded.add(board[i+row][j+col])

    left = full.difference(excluded)
    return left

# Find an empty cell in the Sudoku grid
def find_empty(board):
    minv = 10
    minn = set()
    pos = ()
    flag = 0
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == 0:
                flag = 1
                numbers = best(board,i,j)
                if(minv > len(numbers) and len(numbers) > 0):
                    minv = len(numbers)
                    minn = numbers
                    pos = (i,j)

    if(minv == 10):
        return (None,None,flag)

    return (minn,pos,flag)

# Check if a number is valid in a given position
def valid(bo, num, pos):
    # Check row
```

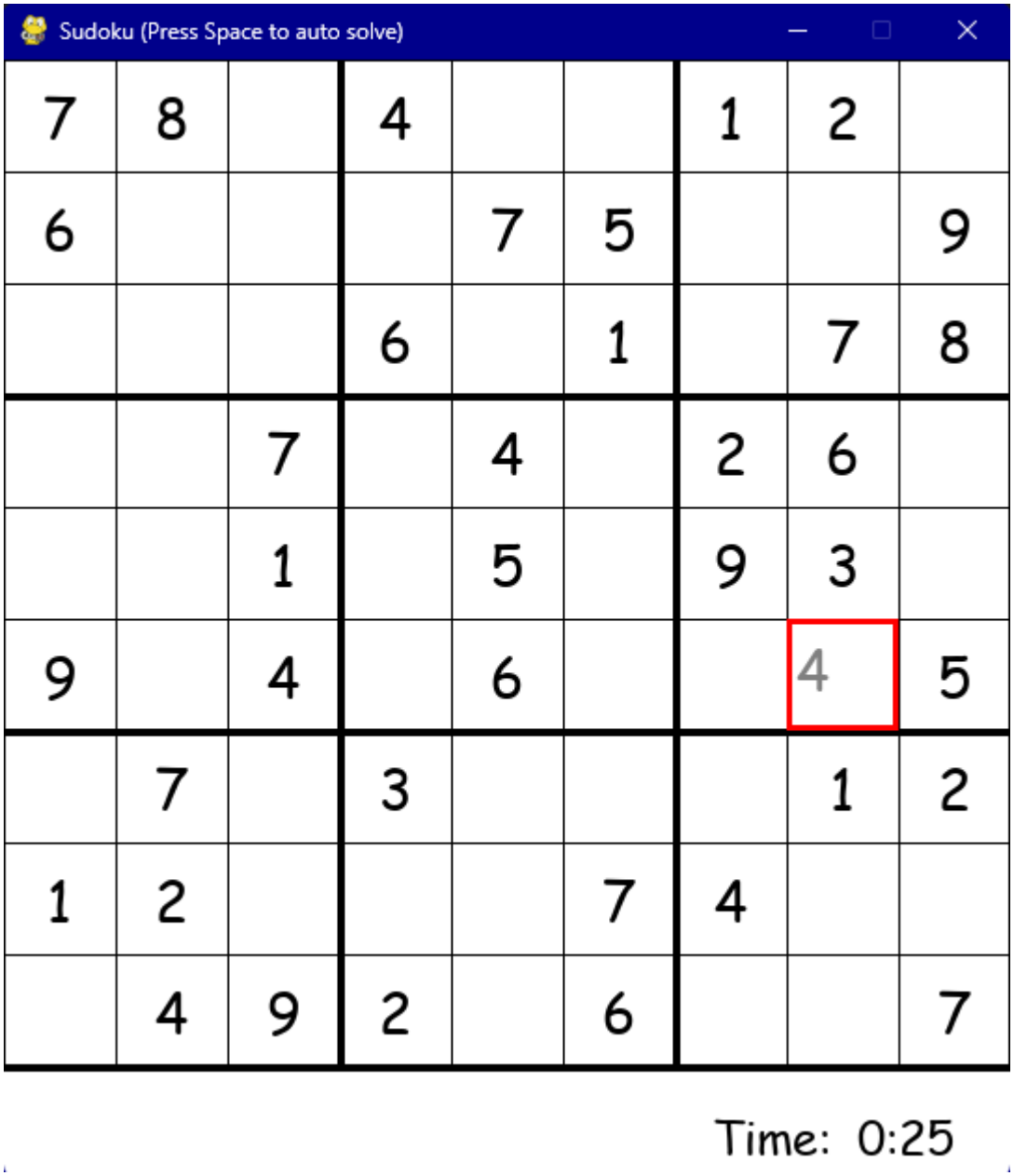
```
    for i in range(len(bo[0])):
        if bo[pos[0]][i] == num and pos[1] != i:
            return False

    # Check column
    for i in range(len(bo)):
        if bo[i][pos[1]] == num and pos[0] != i:
            return False

    # Check box
    box_x = pos[1] // 3
    box_y = pos[0] // 3

    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if bo[i][j] == num and (i,j) != pos:
                return False

    return True


def redraw_window(win, board, time, strikes):
    win.fill((255,255,255))
    # Draw time
    fnt = pygame.font.SysFont("comicsans", 25)
    text = fnt.render("Time: " + format_time(time), 1, (0,0,0))
    win.blit(text, (540 - 160, 560))
    # Draw grid and board
    board.draw()

def format_time(secs):
    sec = secs%60
    minute = secs//60
    hour = minute//60

    mat = " " + str(minute) + ":" + str(sec)
    return mat

def draw_dialog(win, message):
    font = pygame.font.SysFont("inter", 25)
    text = font.render(message, True, (0, 0, 0))
    text_rect = text.get_rect(center=(270, 300))
    pygame.draw.rect(win, (255, 255, 255), (100, 250, 340, 100))
    pygame.draw.rect(win, (0, 0, 0), (100, 250, 340, 100), 2)
    win.blit(text, text_rect)

def main():
    win = pygame.display.set_mode((540,600))
    pygame.display.set_caption("Sudoku (Press Space to auto solve) ")
```

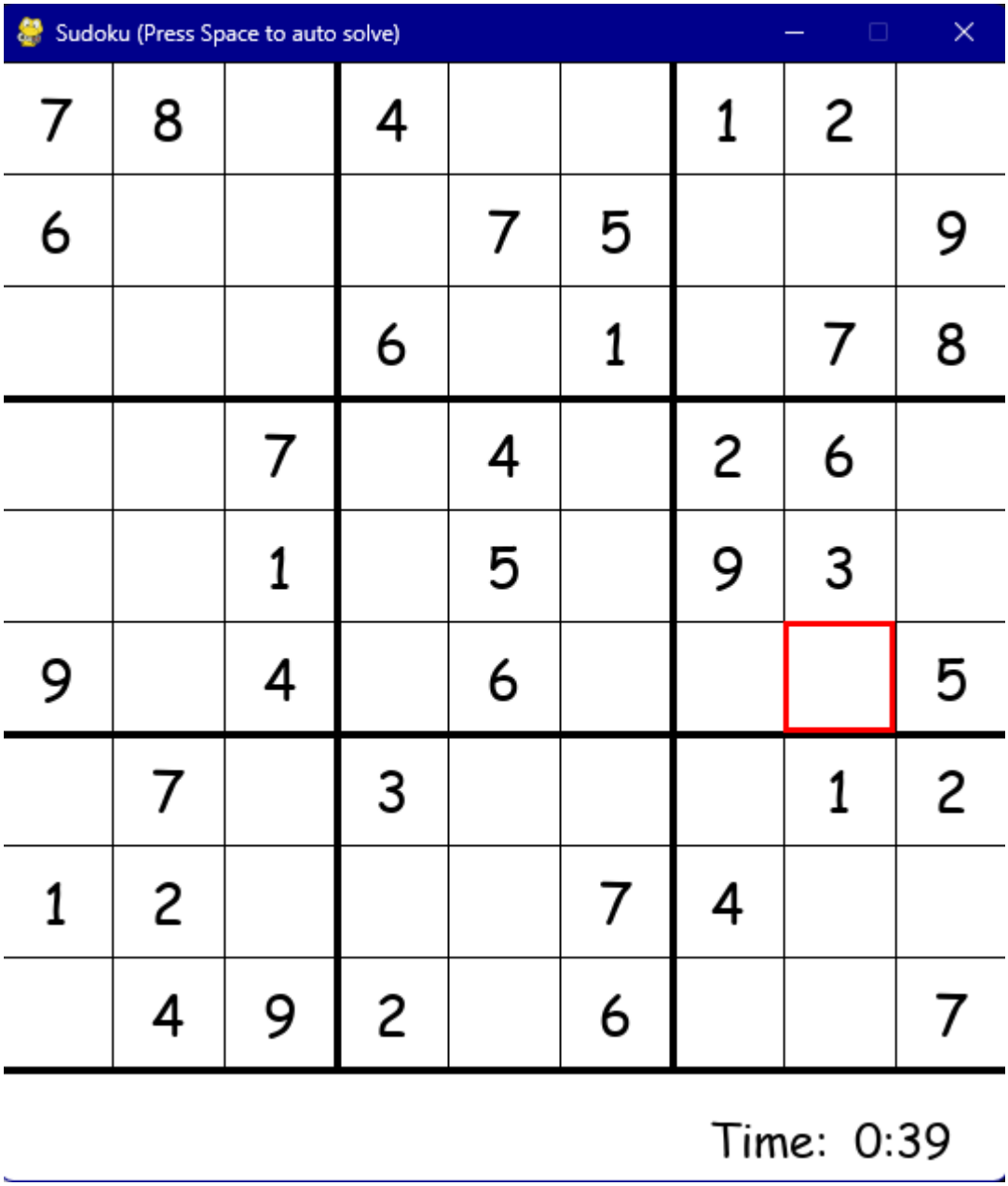
```
    board = Grid(9, 9, 540, 540, win)
    key = None
    run = True
    start = time.time()
    strikes = 0
    solved_by_player = False  # Flag to track if sudoku solved by player
    while run:

        play_time = round(time.time() - start)

        for event in pygame.event.get():

            if event.type == pygame.QUIT:
                run = False
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_1:
                    key = 1
                if event.key == pygame.K_2:
                    key = 2
                if event.key == pygame.K_3:
                    key = 3
                if event.key == pygame.K_4:
                    key = 4
                if event.key == pygame.K_5:
                    key = 5
                if event.key == pygame.K_6:
                    key = 6
                if event.key == pygame.K_7:
                    key = 7
                if event.key == pygame.K_8:
                    key = 8
                if event.key == pygame.K_9:
                    key = 9
                if event.key == pygame.K_DELETE:
                    board.clear()
                    key = None

                if event.key == pygame.K_SPACE:
                    if board.solve_gui():
                        solved_by_player = False
                        draw_dialog(win, "The computer solved the sudoku.")
                        pygame.display.update()
                        time.sleep(2)
                        run = False
                    else:
                        draw_dialog(win, "The sudoku is not solvable!")
                        pygame.display.update()
                        time.sleep(2)
```

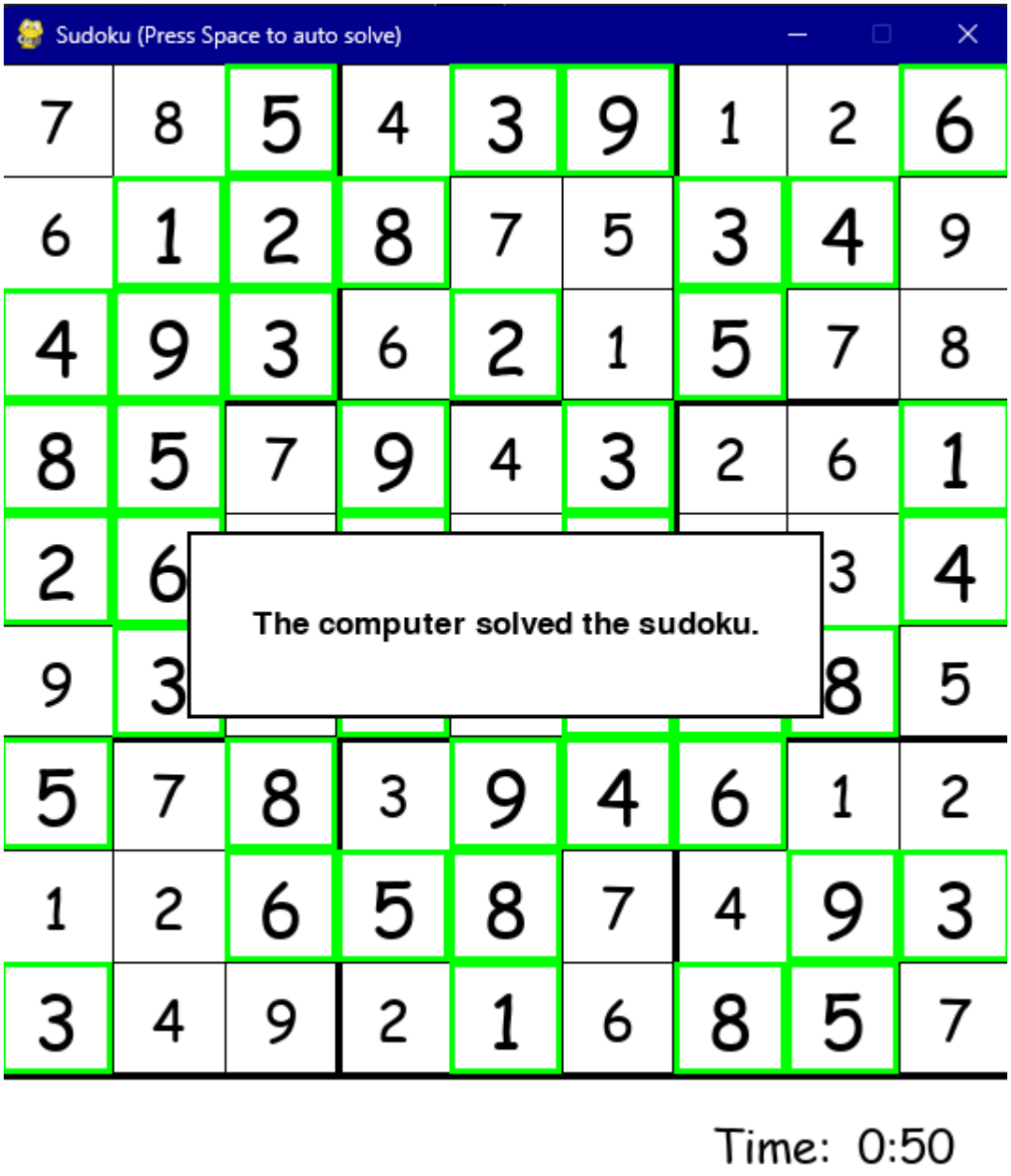
```
                if event.key == pygame.K_RETURN:
                    i, j = board.selected
                    if board.cubes[i][j].temp != 0:
                        if board.place(board.cubes[i][j].temp):
                            print("Success")
                            if board.is_finished():
                                solved_by_player = True
                                draw_dialog(win, "You are a CHAMP!")
                                pygame.display.update()
                                time.sleep(2)
                                run = False
                        else:
                            print("Wrong")
                            strikes += 1
                        key = None

            if event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                clicked = board.click(pos)
                if clicked:
                    board.select(clicked[0], clicked[1])
                    key = None

        if board.selected and key != None:
            board.sketch(key)

        redraw_window(win, board, play_time, strikes)
        pygame.display.update()

    pygame.quit()

main()
```

Output Screenshots:




Sudoku (Press Space to auto solve)
7 8 4 1 2
6 7 5 9
6 1 7 8
7 4 2 6
1 5 9 3
9 4 6 4 5
7 3 1 2
1 2 7 4
4 9 2 6 7
Time: 0:25


Sudoku (Press Space to auto solve)
7 8 4 1 2
6 7 5 9
6 1 7 8
7 4 2 6
1 5 9 3
9 4 6 5
7 3 1 2
1 2 7 4
4 9 2 6 7
Time: 0:39


Sudoku (Press Space to auto solve)
7 8 5 4 3 9 1 2 6
6 1 2 8 7 5 3 4 9
4 9 3 6 2 1 5 7 8
8 5 7 9 4 3 2 6 1
2 6 3 4
9 3 8 5
5 7 8 3 9 4 6 1 2
1 2 6 5 8 7 4 9 3
3 4 9 2 1 6 8 5 7
The computer solved the sudoku.
Time: 0:50

Declarative Implementation of Mastermind in Python:

Data Structures and Type Definitions:

- **Grid Class:** Represents the Sudoku grid and contains methods to manipulate and solve the grid.
- **Cube Class:** Represents an individual cell in the Sudoku grid.

Behaviour Step Implementations:

- **Initialize Pygame and the Grid:** Create the Pygame window and initialize the Grid object with a predefined Sudoku board.
- **Main Loop:** Handle user inputs such as number keys, delete key, space key (to solve), and return key (to place a number).

Areas of Improvement:

- Implement different difficulty levels with varying pre-filled cells.
- Add an option to reset the board to its initial state.
- Provide usage instructions and game rules within the game or as a separate help menu.

T3 Testing and Verification of Programs via appropriate tools and techniques

---

Test Case ID	Test Case Description	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
TC 1	Verify Sudoku grid initialization	1. Launch the Sudoku game. 2. Check the initial state of the grid.	Initial board State	The grid matches the initial board state provided in the code.	As Expected	PASS
TC 2	Verify selecting a cell	1. Launch the Sudoku game. 2. Click on a cell in the grid.	Coordinates of the cell to select	The selected cell is highlighted	As Expected	PASS

		3. Check if the cell is highlighted.				
TC 3	Verify placing a valid number in an empty cell	1. Launch the Sudoku game. 2. Select an empty cell. 3. Enter a valid number using the keyboard. 4. Press 'Enter' to place the number.	Valid number (e.g., 5) and coordinates of the empty cell	The number is placed in the cell.	As Expected	PASS
TC 4	Verify placing an invalid number in an empty cell	1. Launch the Sudoku game. 2. Select an empty cell. 3. Enter an invalid number using the keyboard. 4. Press 'Enter' to place the number.	Invalid number (e.g., 10) and coordinates of the empty cell	The number is not placed in the cell, and an error is indicated.	As Expected	PASS
TC 5	Verify game detects completed puzzle	1. Launch the Sudoku game. 2. Solve the Sudoku puzzle manually or use the auto-solve function. 3. Check if the game detects the puzzle as completed.	Solved Sudoku puzzle	The game displays a "The computer solved the Sudoku!" message.	As Expected	PASS