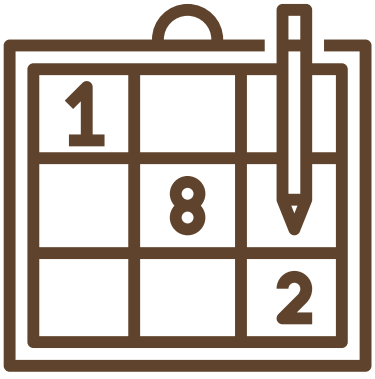
SUDOKU GAME WITH AI

Team IDs

Zenab Osama	2305289
Mariam Ahmed	2305300
Ahmed Sadek	2305355



THE EXPLANATION OF OUR GAME

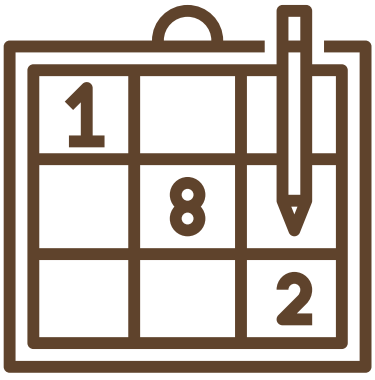
This code creates a graphical Sudoku game using Pygame with three different modes. Here's a high-level breakdown:

1. Imports and Initialization:
   - The code imports necessary modules: pygame for the GUI, sys for handling system-related tasks, and main for custom game logic.
   - Pygame is initialized and constants like window dimensions, button sizes, colors, and fonts are set up.
2. Window Creation and Buttons:
   - A main window for the game is created with buttons for three game modes (AI-generated puzzle solving, user-generated puzzles, and full user interaction).
   - The button positions and text are defined, and the window is prepared for user interaction.

3. Functions for Drawing and Interacting:
   - draw_sudoku_board(window, board): Draws the 9x9 grid of the Sudoku puzzle and populates it with numbers.
   - mode_1_window(), mode_2_window(), mode_3_window(): These functions represent the three game modes. Each mode offers different functionalities:
     - Mode 1: AI generates and solves the puzzle automatically.
     - Mode 2: User generates a puzzle and lets AI solve it.
     - Mode 3: User generates and solves the puzzle by inputting numbers and receiving feedback.

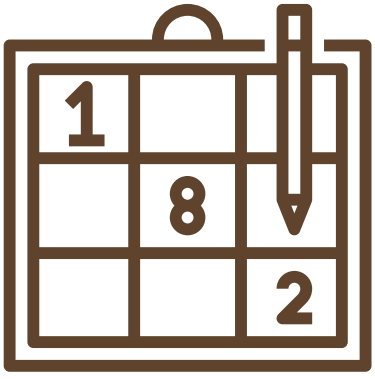
4. Main Game Loop:
   - The game runs in an infinite loop, waiting for user input (button clicks or key presses).
   - Depending on which mode the player selects (via clicking a button), the game transitions to the corresponding mode and calls the respective function.

5. Error Handling and Sounds:
   - For some modes, if a puzzle is unsolvable, an error message appears, and a sound plays.
   - In Mode 3, the user's input is compared with the solution to highlight incorrect entries in red.
6. Graphics and Display:
   - The game uses a custom background image and colors to enhance the user interface.
   - Text is rendered on buttons and error messages are displayed as needed.



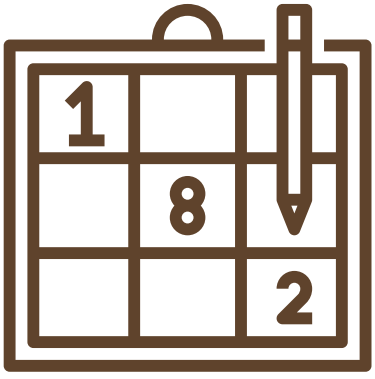
```
import pygame
import sys
import main
```

1. pygame: This is the main library used for creating graphical user interfaces and handling user input in games.
2. sys: This module is used to handle system-level operations, such as exiting the program.
3. main: This module presumably contains functions for generating and solving Sudoku puzzles (such as generate_random_puzzle() and solve_sudoku()).



```
# Initialize Pygame
pygame.init()

# Constants
WIDTH, HEIGHT = 1050, 742
BUTTON_WIDTH, BUTTON_HEIGHT = 350, 100
BUTTON_GAP = 100
```

- pygame.init(): Initializes all Pygame modules, such as graphics and sound.
- WIDTH and HEIGHT: The dimensions of the main game window.
- BUTTON_WIDTH and BUTTON_HEIGHT: The dimensions of the buttons on the main screen.
- BUTTON_GAP: The vertical gap between the buttons (though it is unused in this code).



```python
# Colors
WHITE = (255, 255, 255)
LIGHTGREY = (170, 170, 170)
GRAY = (233, 228, 216)
DARKGREY = (36, 18, 63)
DARKER_GREY = (35, 35, 35)
PURPLE = (125, 84, 222)
BLACK = (0, 0, 0)
RED = (230, 30, 30)
DARKRED = (150, 0, 0)
GREEN = (30, 230, 30)
DARKGREEN = (0, 125, 0)
BLUE = (30, 30, 122)
CYAN = (30, 230, 230)
GOLD = (225, 185, 0)
DARKGOLD = (165, 125, 0)
YELLOW = (255, 255, 0)
PERIWINKLE = (183, 195, 243)
```

- These are predefined color values using RGB tuples for easy reference throughout the game.

apply_arc_consistency Method:

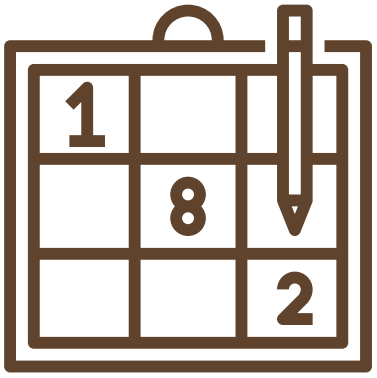
```python
def apply_arc_consistency(board): #reduce domain based on constraints
    queue = [] #
    domains = []  #domain of each cell
    domains = []

    domains = [[list(range(1, 10)) for _ in range(9)] for _ in range(9)] # domain of cell that is empty

    steps = []  # List to store the steps of arc consistency

    for i in range(9):
        for j in range(9):
            if board[i][j] != 0:
                domains[i][j] = [board[i][j]] # domain for cell = its value
                queue.append((i, j)) # index of cell that have element
```

Backtracking Method:

```python
def backtracking(board):
    empty_cell = is_empty_cell(board) # empty minimum domain cell
    print(f"MRV is {empty_cell}")
    if empty_cell is None: # board is full
        return True

    row, col = empty_cell # index of least domain value
    domain_values = get_domain_values(board, row, col) # get domain of the least domain
    domain_values.sort(key=lambda num: count_constrained_values(board, row, col, num)) # for every domain, ca
    print(f"Trying to fill cell ({row}, {col}) with domain values: {domain_values}") # fill the MRV with its

    for num in domain_values:
        print(f"Trying value {num} for cell ({row}, {col})") # try first domain
        if is_valid_move(board, row, col, num):
            board[row][col] = num # assign
            print("Applying forward checking") # detection of failure
            if apply_arc_consistency(board) is not None: # new board after trying first domain
                print("Forward checking successful")
                if backtracking(board): # check if empty cells is none
                    return True
            print(f"Value {num} for cell ({row}, {col}) leads to conflict. Backtracking ... ") # does not apply
            board[row][col] = 0
        else: # not valid move
            print(f"Value {num} is not valid for cell ({row}, {col}). Skipping ... ")

    print(f"No valid value found for cell ({row}, {col}). Backtracking ... ") # backtracking
    return False
```

forward_checking Method:

```python
def forward_checking(board, row, col, num):
    # original_board = copy.deepcopy(board)  #copy of board

    board[row][col] = num # assign

    # Check consistency
    for i in range(9):
        if i ≠ col and board[row][i] == num:  #Check for conflicts in the same row
            board[row][col] = 0  #Revert the assignment
            return False
        if i ≠ row and board[i][col] == num:  # Check for conflicts in the same column
            board[row][col] = 0  # Revert the assignment
            return False
    for i in range(row - row % 3, row - row % 3 + 3):
        for j in range(col - col % 3, col - col % 3 + 3):
            if (i ≠ row or j ≠ col) and board[i][j] == num:  # Check for conflicts in the same 3x3 subgrid
                board[row][col] = 0  # Revert the assignment
                return False
```

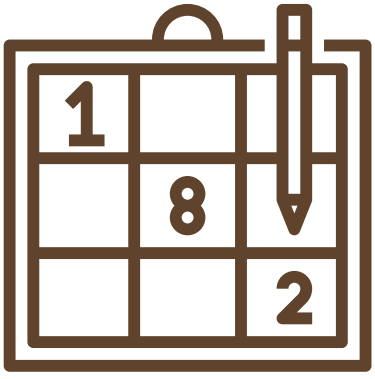


```
# Create the main window
window = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Sudoku Game")

# Load background image
background_image = pygame.image.load("sudoko/background_image.png")
background_image = pygame.transform.scale(background_image, size: (WIDTH, HEIGHT))
# Fonts
font = pygame.font.SysFont( name: None, size: 36)

# Buttons
button_font = pygame.font.SysFont( name: None, size: 24)
```

- pygame.display.set_mode(): Creates the main game window with the specified width and height.
- pygame.display.set_caption(): Sets the window title.
- pygame.image.load(): Loads the background image from the file path.
- pygame.transform.scale(): Scales the background image to fit the window size.
- pygame.font.SysFont(): Creates font objects for rendering text. The first parameter specifies the font name (None means default system font), and the second parameter is the font size.

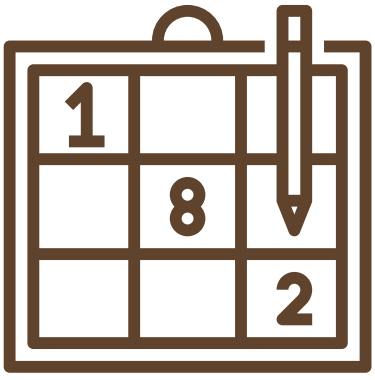
```
buttons = [
    pygame.Rect((WIDTH - BUTTON_WIDTH) // 2, 100, BUTTON_WIDTH, BUTTON_HEIGHT),
    pygame.Rect((WIDTH - BUTTON_WIDTH) // 2, 300, BUTTON_WIDTH, BUTTON_HEIGHT),
    pygame.Rect((WIDTH - BUTTON_WIDTH) // 2, 500, BUTTON_WIDTH, BUTTON_HEIGHT)
]

button_texts = [
    "Mode 1: AI Generate And Solve",
    "Mode 2: User Generate And AI Solve",
    "Mode 3: User Generate And User Solve"
]
```

- buttons: A list of pygame.Rect objects representing the three buttons for selecting game modes. These rectangles are centered horizontally on the screen with varying vertical positions.
- button_texts: A list of text strings corresponding to each button, indicating the mode of the game.



```python
def draw_sudoku_board(window, board):
    cell_size = 70
    cell_margin = 10
    # subgrid_size = 3 * (cell_size + cell_margin) + 3
    board_size = 9 * cell_size + 10 * cell_margin
    board_start_x = 15
    board_start_y = 15

    # Define a larger font size for the numbers
    number_font = pygame.font.SysFont( name: None,  size: 40)

    for i in range(9):
        for j in range(9):
            cell_x = board_start_x + j * (cell_size + cell_margin)
            cell_y = board_start_y + i * (cell_size + cell_margin)
            pygame.draw.rect(window, WHITE,  rect: (cell_x, cell_y, cell_size, cell_size), border_radius=10)

            if board[i][j] != 0:
                # Render the number with the larger font size
                text_surface = number_font.render(str(board[i][j]),  antialias: True, PURPLE)
                text_rect = text_surface.get_rect(center=(cell_x + cell_size / 2, cell_y + cell_size / 2))
                window.blit(text_surface, text_rect)
            # if i % 3 == 0 and j % 3 == 0:
            # pygame.draw.rect(window, PERIWINKLE, (cell_x - cell_margin +3, cell_y - cell_margin+3, subgrid_size, subgrid_si
```

- draw_sudoku_board(): This function draws the Sudoku grid and the numbers on the board. It loops through each cell in the board and renders the corresponding value. It uses the specified color values for the cells and numbers.



MODE 1: AI GENERATE AND SOLVE

```
def mode_1_window():
    # Create a new window for Mode 1
    mode_1_window = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Sudoku Mode 1")
    #unsolvable_sound = pygame.mixer.Sound('1.wav')  # Replace 'unsolvable_sound.wav' with your sound file


    # Generate a random Sudoku puzzle
    puzzle = main.generate_random_puzzle()

    # Main loop for Mode 1 window
    mode_1_running = True
    error_message = None  # Initialize error message to None
    while mode_1_running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                mode_1_running = False
                pygame.quit()
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                # Check if solve button is clicked
                if 790 <= event.pos[0] <= 990 and 600 <= event.pos[1] <= 650:
                    # Solve the puzzle
                    st = main.time.time()
                    solved_puzzle = main.solve_sudoku(puzzle)
                    ed = main.time.time()
                    if solved_puzzle is not None:
                        # Calculate and print the elapsed time
                        elapsed_time = ed - st
                        print(f"The code took {elapsed_time:.5f} seconds to execute.")
```

```
                        # Update the Sudoku board if the puzzle is solvable
                        puzzle = solved_puzzle
                        error_message = None  # Clear any previous error message
                    else:
                        # Set error message if the puzzle is unsolvable
                        error_message = "The puzzle is unsolvable."
                        #unsolvable_sound.play()

                # Check if regenerate button is clicked
                elif 790 <= event.pos[0] <= 990 and 675 <= event.pos[1] <= 725:
                    # Regenerate the puzzle
                    puzzle = main.generate_random_puzzle()
                    error_message = None  # Clear any previous error message
                # Check if back button is clicked
                elif 950 <= event.pos[0] <= 1000 and 10 <= event.pos[1] <= 30:
                    # Exit Mode 1 and return to main window
                    mode_1_running = False

        # Draw the Sudoku board
        mode_1_window.fill(DARKGREY)
        draw_sudoku_board(mode_1_window, puzzle)
        # Draw error message if present
        if error_message:
            error_font = pygame.font.SysFont( name: None,  size: 36)
            error_text = error_font.render(error_message,  antialias: True, RED)
            error_rect = error_text.get_rect(center=(WIDTH // 2, HEIGHT // 2))
            mode_1_window.blit(error_text, error_rect)
```

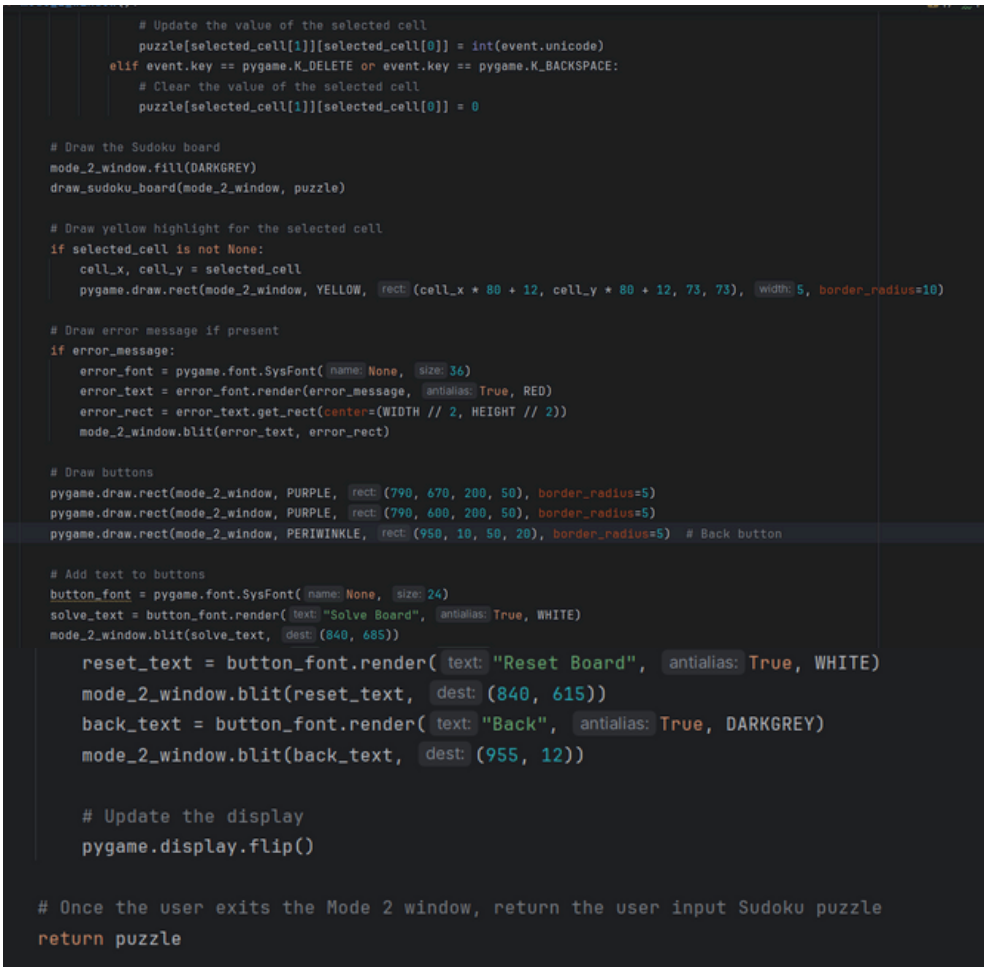
```
# Draw buttons
pygame.draw.rect(mode_1_window, PURPLE, rect: (790, 600 - 5, 200, 50), border_radius=5)
pygame.draw.rect(mode_1_window, PURPLE, rect: (790, 675 - 5, 200, 50), border_radius=5)
pygame.draw.rect(mode_1_window, PERIWINKLE, rect: (950, 10, 50, 20), border_radius=5)  # Back button

# Add text to buttons
button_font = pygame.font.SysFont( name: None, size: 24)
regenerate_text = button_font.render( text: "Solve Board", antialias: True, WHITE)
solve_text = button_font.render( text: "Regenerate New", antialias: True, WHITE)
mode_1_window.blit(regenerate_text, dest: (835, 610))
mode_1_window.blit(solve_text, dest: (830, 685))
back_text = button_font.render( text: "Back", antialias: True, DARKGREY)
mode_1_window.blit(back_text, dest: (955, 12))

# Update the display
pygame.display.flip()
```

- mode_1_window(): This function creates and handles the gameplay for Mode 1 (AI generates and solves the puzzle).
- Puzzle Generation: The puzzle is generated using the main.generate_random_puzzle() function.
- Puzzle Solving: When the user clicks the "Solve Board" button, the AI solves the puzzle using the main.solve_sudoku() function.
- Error Handling: If the puzzle is unsolvable, an error message is displayed.

MODE_2_WINDOW FUNCTION

```
def mode_2_window():
    # Create a new window for Mode 2
    mode_2_window = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Sudoku Mode 2")
    unsolvable_sound = pygame.mixer.Sound('sudoko/1.wav')  # Replace 'unsolvable_sound.wav' with your sound file


    # Initialize an empty Sudoku puzzle
    puzzle = [[0 for _ in range(9)] for _ in range(9)]

    # Track the selected cell position
    selected_cell = None

    # Main loop for Mode 2 window
    mode_2_running = True
    error_message = None  # Initialize error message to None
    while mode_2_running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                mode_2_running = False
                pygame.quit()
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                # Check if back button is clicked
                if 950 <= event.pos[0] <= 1000 and 10 <= event.pos[1] <= 30:
                    # Exit Mode 2 and return to the main window
                    mode_2_running = False
                elif 790 <= event.pos[0] <= 990 and 670 <= event.pos[1] <= 720:
                    # Check if the puzzle is valid
                    if main.is_valid_sudoku(puzzle):
                        # Solve the puzzle when "Solve Board" button is clicked
```

```
                        # Solve the puzzle when "Solve Board" button is clicked
                        st = main.time.time()
                        solved_puzzle = main.solve_sudoku(puzzle)
                        ed = main.time.time()
                        if solved_puzzle is not None:
                            # Calculate and print the elapsed time
                            elapsed_time = ed - st
                            print(f"The code took {elapsed_time:.5f} seconds to execute.")
                            puzzle = solved_puzzle
                            error_message = None  # Clear any previous error message
                        else:
                            error_message = "The puzzle is unsolvable."
                            unsolvable_sound.play()

                    else:
                        error_message = "Invalid Sudoku Input, Please Check Game Constrains"
                elif 790 <= event.pos[0] <= 990 and 600 <= event.pos[1] <= 650:
                    # Reset the puzzle when "Reset Board" button is clicked
                    puzzle = [[0 for _ in range(9)] for _ in range(9)]
                    error_message = None  # Clear any previous error message
                else:
                    # Get the clicked cell position
                    cell_x = (event.pos[0] - 10) // 80  # Calculate cell column based on click position
                    cell_y = (event.pos[1] - 10) // 80  # Calculate cell row based on click position
                    if 0 <= cell_x < 9 and 0 <= cell_y < 9:
                        # Highlight the selected cell
                        selected_cell = (cell_x, cell_y)

            elif event.type == pygame.KEYDOWN and selected_cell is not None:
                # Check if a number key (1-9) is pressed
                if pygame.K_1 <= event.key <= pygame.K_9:
```



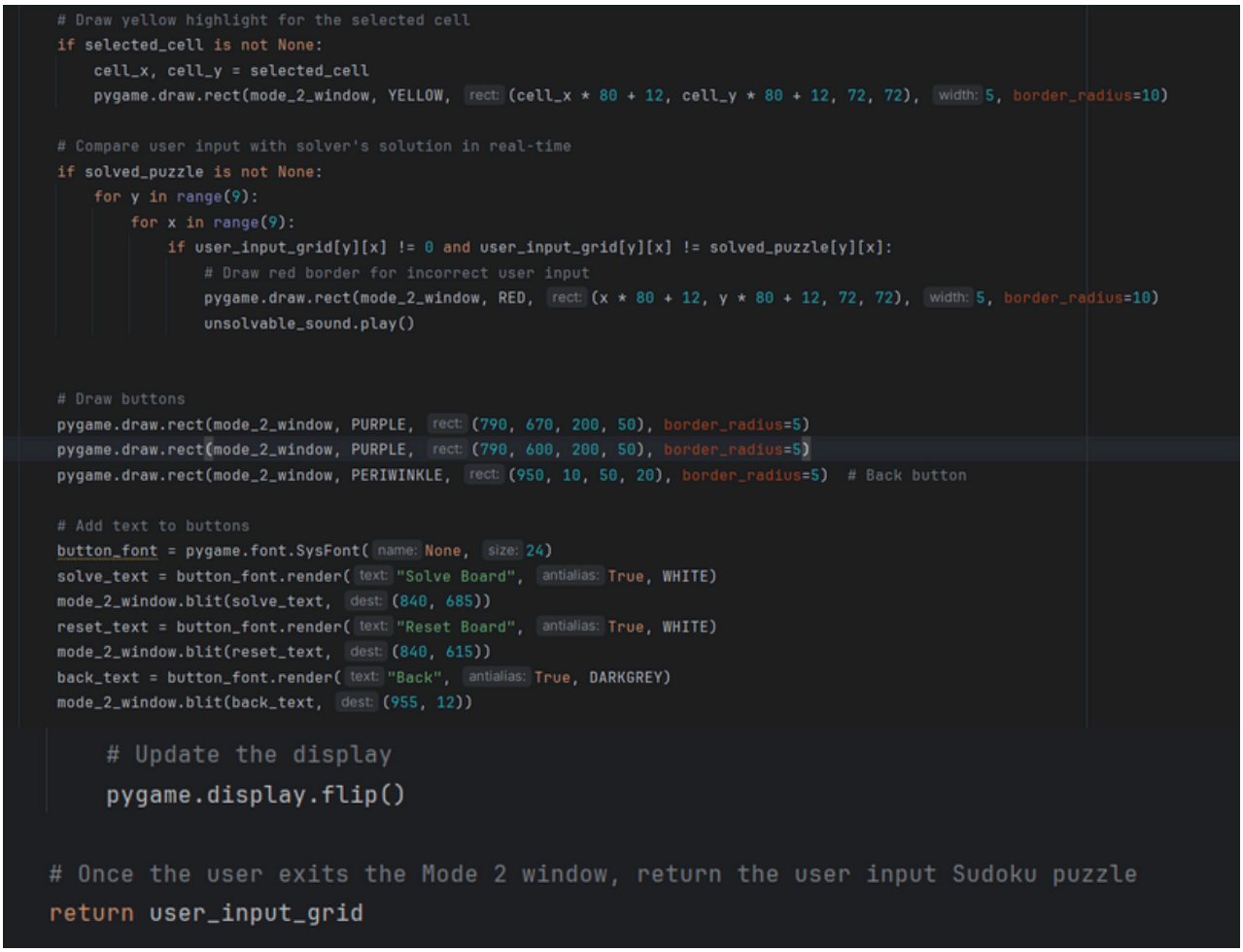
```
            # Update the value of the selected cell
            puzzle[selected_cell[1]][selected_cell[0]] = int(event.unicode)
        elif event.key == pygame.K_DELETE or event.key == pygame.K_BACKSPACE:
            # Clear the value of the selected cell
            puzzle[selected_cell[1]][selected_cell[0]] = 0

    # Draw the Sudoku board
    mode_2_window.fill(DARKGREY)
    draw_sudoku_board(mode_2_window, puzzle)

    # Draw yellow highlight for the selected cell
    if selected_cell is not None:
        cell_x, cell_y = selected_cell
        pygame.draw.rect(mode_2_window, YELLOW, rect: (cell_x * 80 + 12, cell_y * 80 + 12, 73, 73), width: 5, border_radius=10)

    # Draw error message if present
    if error_message:
        error_font = pygame.font.SysFont( name: None, size: 36)
        error_text = error_font.render(error_message, antialias: True, RED)
        error_rect = error_text.get_rect(center=(WIDTH // 2, HEIGHT // 2))
        mode_2_window.blit(error_text, error_rect)

    # Draw buttons
    pygame.draw.rect(mode_2_window, PURPLE, rect: (790, 670, 200, 50), border_radius=5)
    pygame.draw.rect(mode_2_window, PURPLE, rect: (790, 600, 200, 50), border_radius=5)
    pygame.draw.rect(mode_2_window, PERIWINKLE, rect: (950, 10, 50, 20), border_radius=5)  # Back button

    # Add text to buttons
    button_font = pygame.font.SysFont( name: None, size: 24)
    solve_text = button_font.render( text: "Solve Board", antialias: True, WHITE)
    mode_2_window.blit(solve_text, dest: (840, 685))
        reset_text = button_font.render( text: "Reset Board", antialias: True, WHITE)
        mode_2_window.blit(reset_text, dest: (840, 615))
        back_text = button_font.render( text: "Back", antialias: True, DARKGREY)
        mode_2_window.blit(back_text, dest: (955, 12))

        # Update the display
        pygame.display.flip()

# Once the user exits the Mode 2 window, return the user input Sudoku puzzle
return puzzle
```

- Defines the window for Mode 2: where the user generates a puzzle and the AI solves it.
- It tracks the user's interactions with the board (like clicking on a cell or entering numbers) and checks if the puzzle is valid or solvable.
- If the puzzle is valid, it calls main.solve_sudoku to solve the puzzle.

MODE_3_WINDOW FUNCTION

```
def mode_3_window():
    # Create a new window for Mode 2
    mode_2_window = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Sudoku Mode 3")
    unsolvable_sound = pygame.mixer.Sound('sudoko/1.wav')  # Replace 'unsolvable_sound.wav' with your sound file


    # Initialize an empty Sudoku puzzle
    puzzle = [[0 for _ in range(9)] for _ in range(9)]

    # Track the selected cell position
    selected_cell = None

    # Initialize solved puzzle and user input grid
    solved_puzzle = None
    user_input_grid = [[0 for _ in range(9)] for _ in range(9)]

    # Main loop for Mode 2 window
    mode_2_running = True
    error_message = None  # Initialize error message to None
    while mode_2_running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                mode_2_running = False
                pygame.quit()
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                # Check if back button is clicked
                if 950 <= event.pos[0] <= 1000 and 10 <= event.pos[1] <= 30:
                    # Exit Mode 2 and return to the main window
                    mode_2_running = False
```

```
                elif 790 <= event.pos[0] <= 990 and 670 <= event.pos[1] <= 720:
                    # Solve the puzzle and store the solution
                    if solved_puzzle is None:
                        solved_puzzle = main.solve_sudoku(user_input_grid)
                        print(solved_puzzle)
                elif 790 <= event.pos[0] <= 990 and 600 <= event.pos[1] <= 650:
                    # Reset the puzzle when "Reset Board" button is clicked
                    puzzle = [[0 for _ in range(9)] for _ in range(9)]
                    user_input_grid = [[0 for _ in range(9)] for _ in range(9)]
                    error_message = None  # Clear any previous error message
                else:
                    # Get the clicked cell position
                    cell_x = (event.pos[0] - 10) // 80  # Calculate cell column based on click position
                    cell_y = (event.pos[1] - 10) // 80  # Calculate cell row based on click position
                    if 0 <= cell_x < 9 and 0 <= cell_y < 9:
                        # Highlight the selected cell
                        selected_cell = (cell_x, cell_y)

            elif event.type == pygame.KEYDOWN and selected_cell is not None:
                # Check if a number key (1-9) is pressed
                if pygame.K_1 <= event.key <= pygame.K_9:
                    # Update the value of the selected cell in user input grid
                    user_input_grid[selected_cell[1]][selected_cell[0]] = int(event.unicode)
                elif event.key == pygame.K_DELETE or event.key == pygame.K_BACKSPACE:
                    # Clear the value of the selected cell in user input grid
                    user_input_grid[selected_cell[1]][selected_cell[0]] = 0

        # Draw the Sudoku board with user input
        mode_2_window.fill(DARKGREY)
        draw_sudoku_board(mode_2_window, user_input_grid)
```



```
    # Draw yellow highlight for the selected cell
    if selected_cell is not None:
        cell_x, cell_y = selected_cell
        pygame.draw.rect(mode_2_window, YELLOW, rect: (cell_x * 80 + 12, cell_y * 80 + 12, 72, 72), width: 5, border_radius=10)

    # Compare user input with solver's solution in real-time
    if solved_puzzle is not None:
        for y in range(9):
            for x in range(9):
                if user_input_grid[y][x] != 0 and user_input_grid[y][x] != solved_puzzle[y][x]:
                    # Draw red border for incorrect user input
                    pygame.draw.rect(mode_2_window, RED, rect: (x * 80 + 12, y * 80 + 12, 72, 72), width: 5, border_radius=10)
                    unsolvable_sound.play()


    # Draw buttons
    pygame.draw.rect(mode_2_window, PURPLE, rect: (790, 670, 200, 50), border_radius=5)
    pygame.draw.rect(mode_2_window, PURPLE, rect: (790, 600, 200, 50), border_radius=5)
    pygame.draw.rect(mode_2_window, PERIWINKLE, rect: (950, 10, 50, 20), border_radius=5)  # Back button

    # Add text to buttons
    button_font = pygame.font.SysFont( name: None, size: 24)
    solve_text = button_font.render( text: "Solve Board", antialias: True, WHITE)
    mode_2_window.blit(solve_text, dest: (840, 685))
    reset_text = button_font.render( text: "Reset Board", antialias: True, WHITE)
    mode_2_window.blit(reset_text, dest: (840, 615))
    back_text = button_font.render( text: "Back", antialias: True, DARKGREY)
    mode_2_window.blit(back_text, dest: (955, 12))

        # Update the display
        pygame.display.flip()

    # Once the user exits the Mode 2 window, return the user input Sudoku puzzle
    return user_input_grid
```

- Defines the window for Mode 3: where the user both generates and solves the puzzle.
- Similar to Mode 2, but in this mode, the user inputs both the puzzle and the solution.
- It compares the user's inputs to the solved puzzle and highlights incorrect inputs.

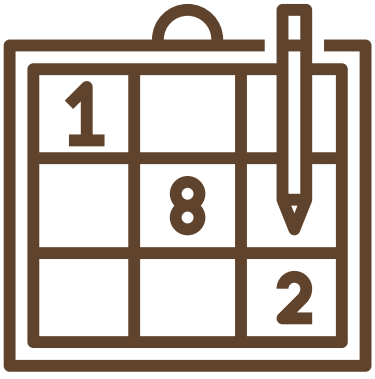
THE MAIN GAME LOOP

```
# Main loop
mode_1_active = False  # Flag to track if Mode 1 window is active
mode_2_active = False  # Flag to track if Mode 2 window is active
mode_3_active = False  # Flag to track if Mode 3 window is active
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
            pygame.quit()
            sys.exit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            # Check if any button was clicked
            for i, button in enumerate(buttons):
                if button.collidepoint(event.pos):
                    if i == 0:  # Mode 1 button clicked
                        mode_1_active = True
                        mode_2_active = False
                        mode_3_active = False
                    elif i == 1:  # Mode 2 button clicked
                        mode_2_active = True
                        mode_1_active = False
                        mode_3_active = False
                    elif i == 2:  # Mode 3 button clicked
                        mode_3_active = True
                        mode_1_active = False
                        mode_2_active = False
```

- This is the main game loop where it checks for events like quitting or button clicks.
- If the user clicks on a button, it sets the corresponding mode flag (mode_1_active, mode_2_active, or mode_3_active) to True and deactivates the others.

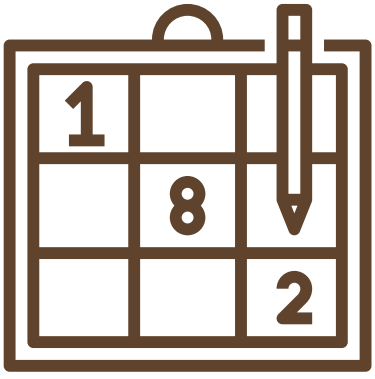
```
# Draw background and buttons
window.blit(background_image, dest: (0, 0))
for i, button in enumerate(buttons):
    pygame.draw.rect(window, GRAY, button, border_radius=10)
    text = button_font.render(button_texts[i], antialias: True, BLACK)
    text_rect = text.get_rect(center=button.center)
    window.blit(text, text_rect)

# Update the display
pygame.display.flip()
```

- It draws the background image and then draws the buttons on the main window.
- The button texts are rendered and placed in the center of each button.
- The display is updated using pygame.display.flip().



```
# If Mode 1 window is active, switch to Mode 1 window
if mode_1_active:
    mode_1_active = False  # Reset the flag
    mode_1_window()  # Call the Mode 1 window function

# If Mode 2 window is active, switch to Mode 2 window
if mode_2_active:
    mode_2_active = False  # Reset the flag
    mode_2_window()  # Call the Mode 2 window function

# If Mode 3 window is active, switch to Mode 3 window
if mode_3_active:...
```

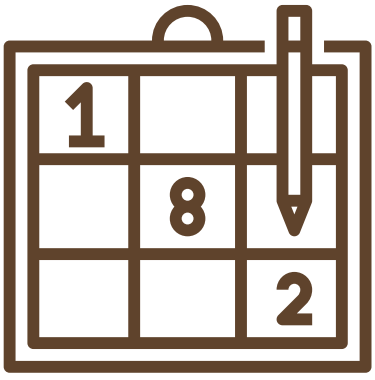
- Based on which mode flag is True, the program calls the corresponding function (mode_1_window, mode_2_window, or mode_3_window) to switch to the respective mode.



ARC CONSISTENCY TREES.

```
. . 3 | . 2 . | 6 . .
9 . . | 3 . 5 | . . 1
. . 1 | 8 . 6 | 4 . .
------+-------+------
. . 8 | 1 . 2 | 9 . .
 7 . . | . . . | . . 8
. . 6 | 7 . 8 | 2 . .
------+-------+------
. . 2 | 6 . 9 | 5 . .
8 . . | 2 . 3 | . . 9
 . . 5 | . 1 . | 3 . .
```

- **Initial Constraints:**
- Example: The cell at (1, 1) (row 1, column 1) cannot have the same value as any other cell in its row, column, or subgrid.
- **Iteration 1**: Processing Constraint (1, 1) ↔ (1, 2)
- Before processing:
- Domain of (1, 1): {1, 2, 4, 5, 7, 8, 9}
- Domain of (1, 2): {1, 2, 4, 5, 7, 8, 9}
- After processing:
- Domain of (1, 1): {1, 4, 5, 7, 8, 9}
- Domain of (1, 2): {1, 2, 4, 5, 7, 8, 9}

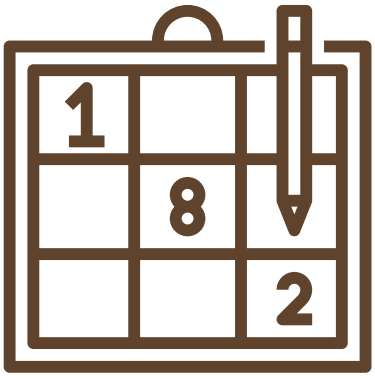


ARC CONSISTENCY TREES.

- **Iteration 2:** Processing Constraint (1, 1) ↔ (2, 1)
- Domain of (1, 1): {1, 4, 5, 7, 8, 9}
- Domain of (2, 1): {1, 4, 5, 7, 8, 9}
- **After processing:**
- Domain of (1, 1): {4, 5, 7, 8, 9}
- Domain of (2, 1): {4, 5, 7, 8, 9}

- Arc Consistency Tree
- **Level 0:**
- **Root: Initial domains for all variables.**
- **Level 1:**
- Processed (1, 1) ↔ (1, 2). Updated domains.
- **Level 2:**
- Processed (1, 1) ↔ (2, 1). Updated domains.

```
4 8 3 | 9 2 1 | 6 5 7
9 6 7 | 3 4 5 | 8 2 1
2 5 1 | 8 7 6 | 4 9 3
------+-------+------
5 4 8 | 1 3 2 | 9 7 6
7 2 9 | 5 6 4 | 1 3 8
1 3 6 | 7 9 8 | 2 4 5
------+-------+------
3 7 2 | 6 8 9 | 5 1 4
8 1 4 | 2 5 3 | 7 6 9
6 9 5 | 4 1 7 | 3 8 2
```



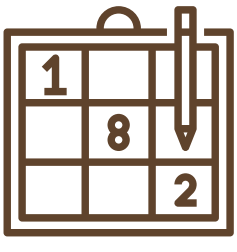
DIFFERENT INITIAL SUDOKU BOARDS

- **Easy Puzzles:** With more initial constraints, domains are reduced quickly, requiring fewer iterations.

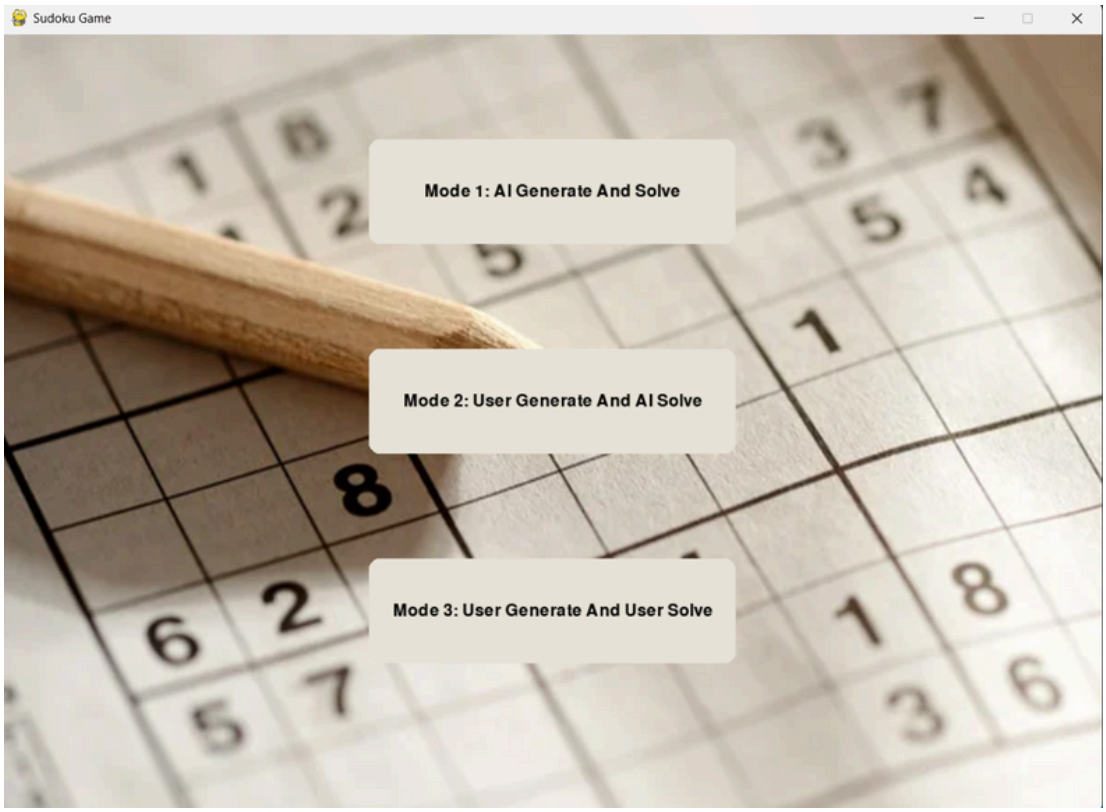
- **Intermediate Puzzles:** Moderate number of constraints leads to more iterations to achieve consistency.

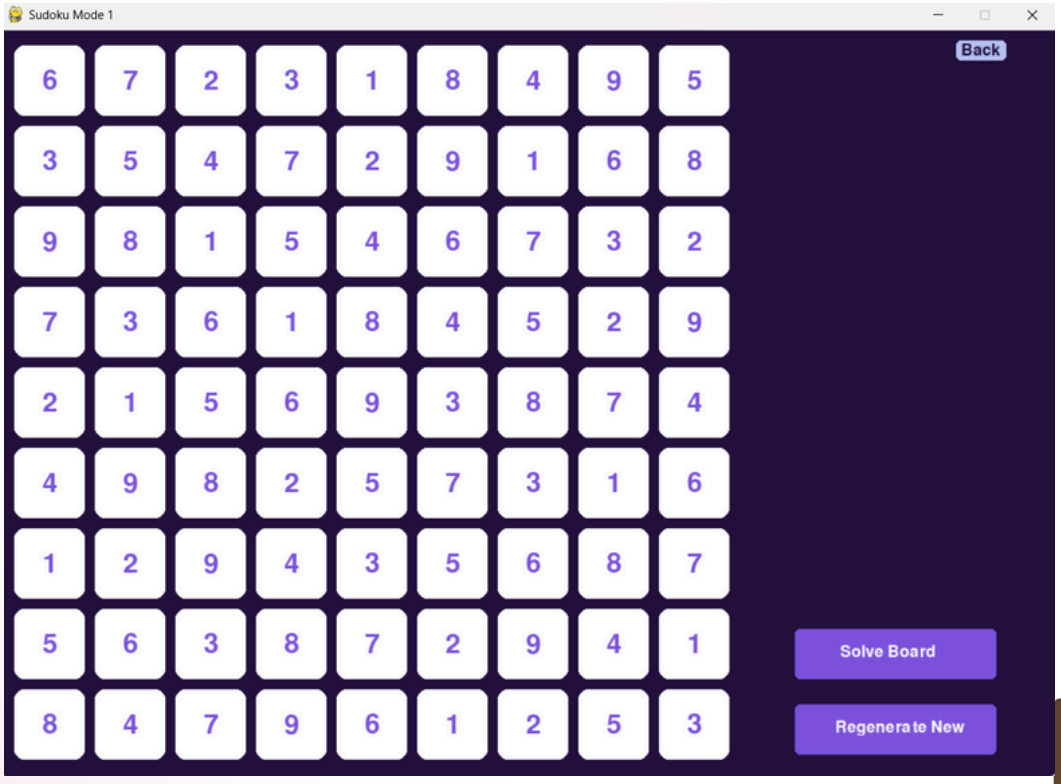
- **Hard Puzzles:** Minimal constraints lead to significant domain reduction work, potentially requiring backtracking after AC-3.

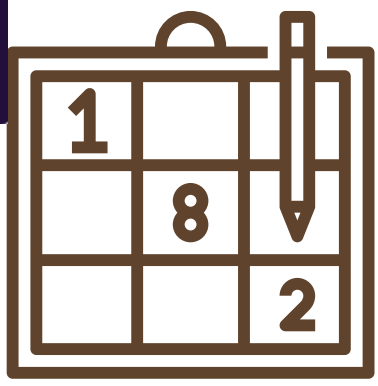
Board Type	Arcs Processed	Time to Solve
Easy	~150	5-10 ms
Intermediate	~300	50-10 ms
HARD	~500+	200-500 ms



TEST CODE


Sudoku Game
Mode 1: AI Generate And Solve
Mode 2: User Generate And AI Solve
Mode 3: User Generate And User Solve


Sudoku Mode 1
Back
6 7 2 3 1 8 4 9 5
3 5 4 7 2 9 1 6 8
9 8 1 5 4 6 7 3 2
7 3 6 1 8 4 5 2 9
2 1 5 6 9 3 8 7 4
4 9 8 2 5 7 3 1 6
1 2 9 4 3 5 6 8 7
5 6 3 8 7 2 9 4 1
8 4 7 9 6 1 2 5 3
Solve Board
Regenerate New


1
8
2