From Classic to Strategy: Journey of Games in Python

Name – Yash Mishra
Login – mishr195@purdue.edu
Section Number – Engr 133 LC 05
Team Number - 22

Introduction:

Introduction to the Program and Its Theme:

As a child, I vividly recall being captivated by the pre-installed games on Windows XP. Among these, I found myself particularly intrigued by the computer's ability to competently challenge me, especially in games like chess. This curiosity fueled my interest in understanding how computers operated within games and inspired me to venture into game development through code.

Motivation behind the Project:

The desire to understand the mechanics behind these games and the intriguing aspect of competition against a computer adversary led me to undertake the task of coding two classic games: Tic Tac Toe and 2048. However, beyond mere recreation, my goal extended to exploring the dichotomy between utilizing external modules and constructing games from scratch.

Different Approaches Utilized

Python's early documentation emphasized outputting code solely to the terminal, which intrigued me. Consequently, I accepted the challenge of developing a terminal-based version of Tic Tac Toe. This approach allowed players to engage with the game, either facing the computer or another human, solely through command-line interactions.

In contrast, for the 2048 game, I harnessed the tkinter module in conjunction with the **random** and **copy** libraries. This combination enabled me to create a Graphical User Interface (GUI) version of the game. The GUI presents players with a visually stimulating interface, responding to arrow key inputs and enhancing the interactive gaming experience.

Project Motivation and Connection to Interests

The endeavor to develop the Tic Tac Toe game using recursive techniques and the minimax AI algorithm was driven by a culmination of personal interests, future career aspirations, and a long-standing passion for game development.

Connection to Future Career and Major

- **Aspiring Computer Engineer:**
  Aspiring to become a computer engineer, the utilization of recursion and minimax AI holds tremendous relevance. These concepts are currently at the forefront of technological advancements, particularly in AI and machine learning. Gaining a foundational understanding of these concepts at their grassroots level through game development lays a robust foundation for delving deeper into these topics in future academic pursuits and professional endeavors.

- **Emerging Trends in AI:**
  Understanding the core principles of AI, especially the application of algorithms like minimax, aligns closely with the cutting-edge developments in the field. These techniques are not just confined to game development but are crucial in various fields, including robotics, data analysis, and autonomous systems.

Connection to Passion for Game Development

- **Consistent Interest in Game Development:**
My unwavering interest in game development has been a constant throughout my life. Starting from childhood when I marveled at the interplay between humans and computers in gaming, to present-day endeavors, this passion remains a driving force. The possibility of game development evolving into a potential career path resonates deeply, as it intertwines my interests with my professional aspirations.

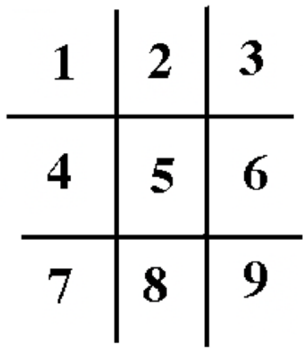
- **Potential Career Prospects:**
Pursuing game development as a potential career avenue aligns with my intrinsic motivation. It not only fuels my creativity but also offers an opportunity to merge technological innovation with entertainment. The Tic Tac Toe project serves as a testament to my commitment and enthusiasm for exploring the realms of game development.

Description of Inputs and Outputs:

Inputs:

- **Tic Tac Toe:**
Player first gets an option between playing against computer or human. If the player selects yes, then the computer starts playing and if the player selects not then it sets up board for two players.

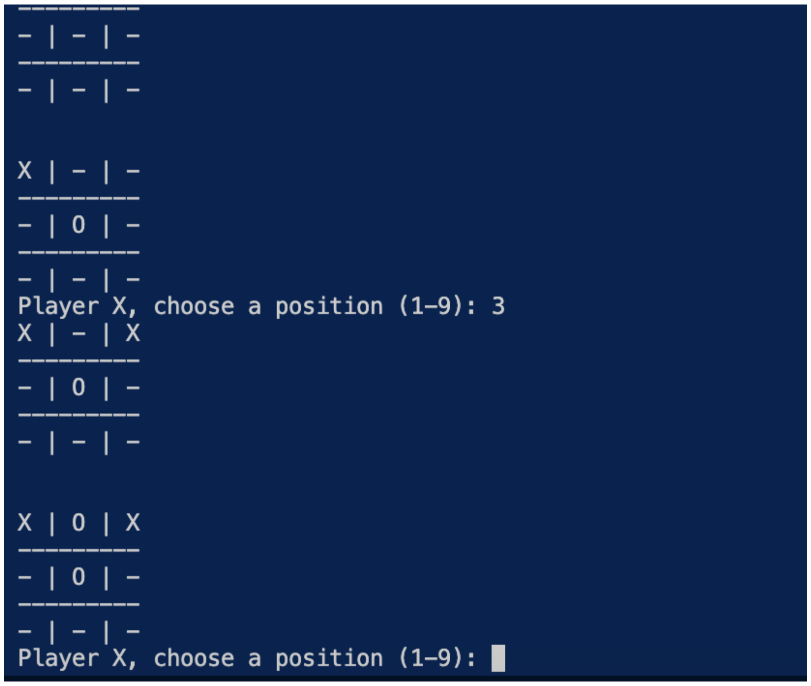
Further, Player inputs for choosing positions on the board (1-9) during their turn. The picture below will highlight the positions respective to the number:



- **2048:** Arrow key inputs to shift tiles in the desired direction and to add simultaneously to reach the goal of 2048.

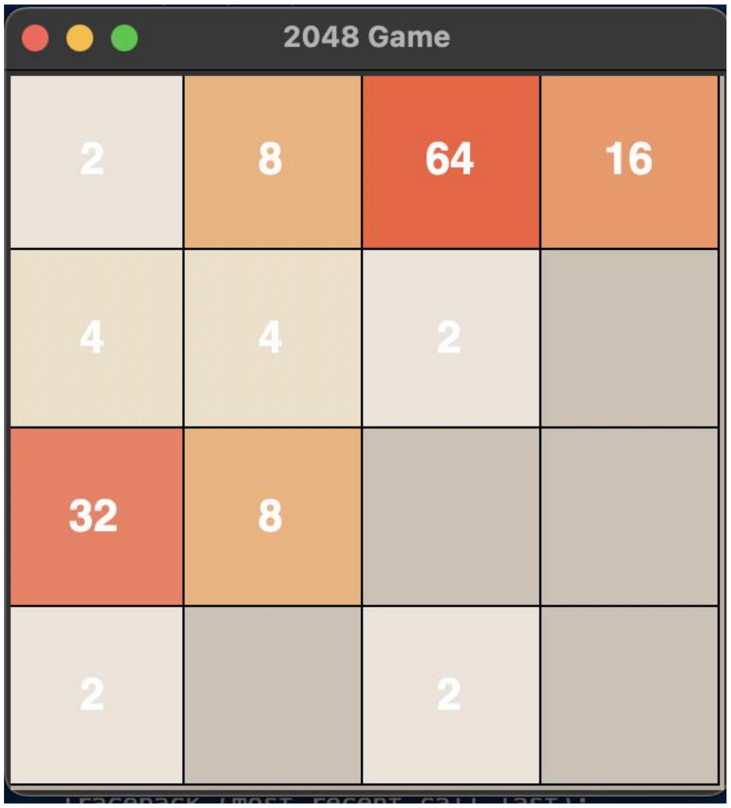
Outputs:

- **Tic Tac Toe:** Terminal-based board displaying player moves and game outcomes.



```
----------
- | - | -
----------
- | - | -


X | - | -
----------
- | O | -
----------
- | - | -
Player X, choose a position (1-9): 3
X | - | X
----------
- | O | -
----------
- | - | -


X | O | X
----------
- | O | -
----------
- | - | -
Player X, choose a position (1-9): 
```

- **2048:** GUI window showcasing the 4x4 grid, updating tile movements, merging, and displaying the current score.



Description of User Defined Function:

2048.py:

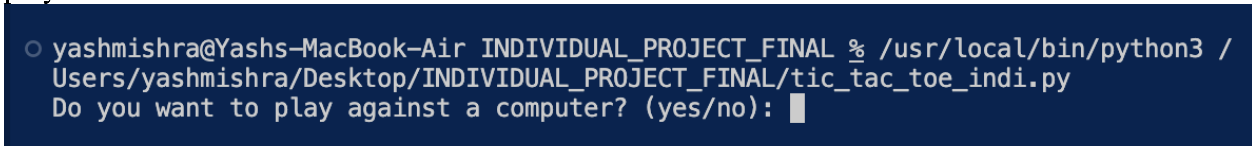
1. initialize_board()
    - Initializes a 4x4 game board with two random tiles.
    - Calls the add_new_tile() function twice to populate the initial board.
2. print_board(board)
    - Prints the current state of the board in the console.
    - Converts the board to a string format for display.
3. add_new_tile(board)
    - Adds a new tile (either 2 or 4) to a random empty cell on the board.
    - Selects a random empty cell and assigns a random value (2 or 4) to it.
4. move(board, direction)
    - Moves the tiles in the specified direction (up, down, left, right).
    - Utilizes the merge_tiles() function to merge adjacent identical tiles after movement.
    - Handles tile movement in the respective direction.
5. merge_tiles(line)
    - Merges adjacent identical tiles in a row or column after a move.
    - Combines identical tiles and returns the modified line.
6. is_game_over(board)
    - Checks if the game is over by assessing if no further moves are possible.
    - Examines the board to identify potential valid moves remaining.
7. game_over()
    - Displays a messagebox asking if the player wants to play again after the game ends.
    - Returns True if the player chooses to play again, else returns False.
8. reset_game()
    - Resets the game by reinitializing the board.
    - Calls initialize_board() and updates the display.
9. update_display()
    - Updates the visual display of the game board using a tkinter canvas.
    - Clears the canvas and redraws the board based on the current state.
    - Checks if the game is over and prompts the player to play again if so.
10. get_tile_color(value)
    - Determines the color representation for a given tile value.
    - Assigns specific colors to different tile values for graphical representation in the GUI.
11. handle_key(event)
    - Listens to keyboard events (arrow keys) and triggers corresponding movements.
    - Calls the move(), add_new_tile(), and update_display() functions based on the direction of the key press.

tic_tac_toe_indi.py:

1. print_board(board)
   - Displays the Tic Tac Toe board on the terminal.
   - Renders the current state of the board using text-based formatting.
2. check_win(player, board)
   - Checks if the specified player has won the game based on the current board state.
   - Examines the board to identify winning combinations.
3. check_tie(board)
   - Determines if the game has reached a tie (draw) based on the current board state.
   - Verifies if there are any empty spaces left on the board.
4. player_input(player, board)
   - Takes input from the player for their move (position on the board).
   - Validates the input and ensures the move is valid before returning it.
5. switch_player(player)
   - Switches the current player between "X" and "O" after each move.
   - Alternates the turns between players during the game.
6. player_game()
   - Manages the gameplay for two human players.
   - Controls the flow of the game, taking inputs from players and determining the game outcome.
7. computer_game()
   - Manages the gameplay for a human player against the computer.
   - Implements an AI-based opponent using the minimax algorithm to make strategic moves for the computer player.
8. computer_move(board, current_player)
   - Calculates the best move for the computer player using the minimax algorithm.
   - Considers possible moves and selects the optimal move to maximize its chances of winning.
9. minimax(board, depth, is_maximizing)
   - Implements the minimax algorithm for decision-making in the game.
   - Recursively evaluates potential moves and determines the best move for the

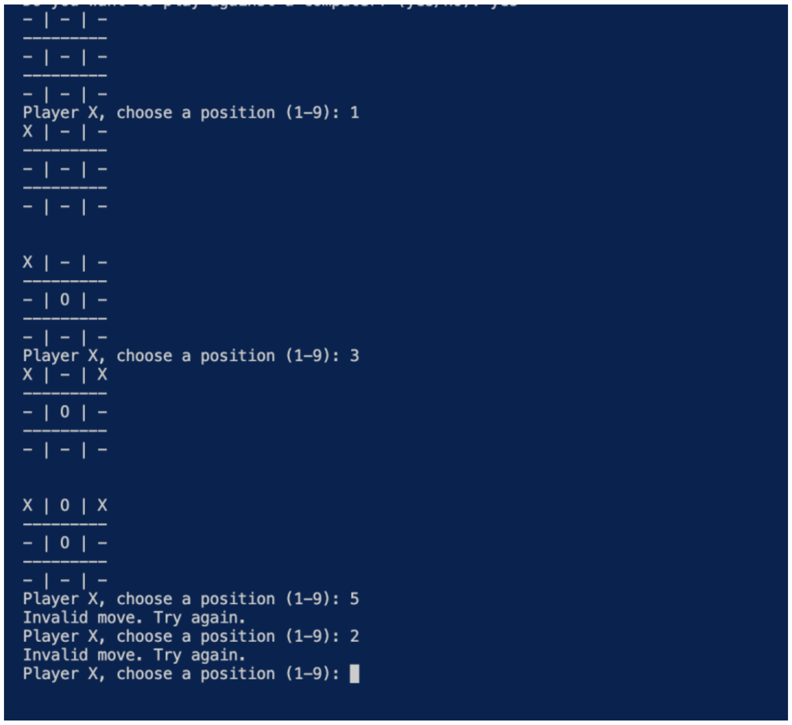
User Manual:

The User has the choice to choose between the two games manually and if he chooses to play tic-tac-toe, the first choice he has to make is if he wants to play against a computer or another player.



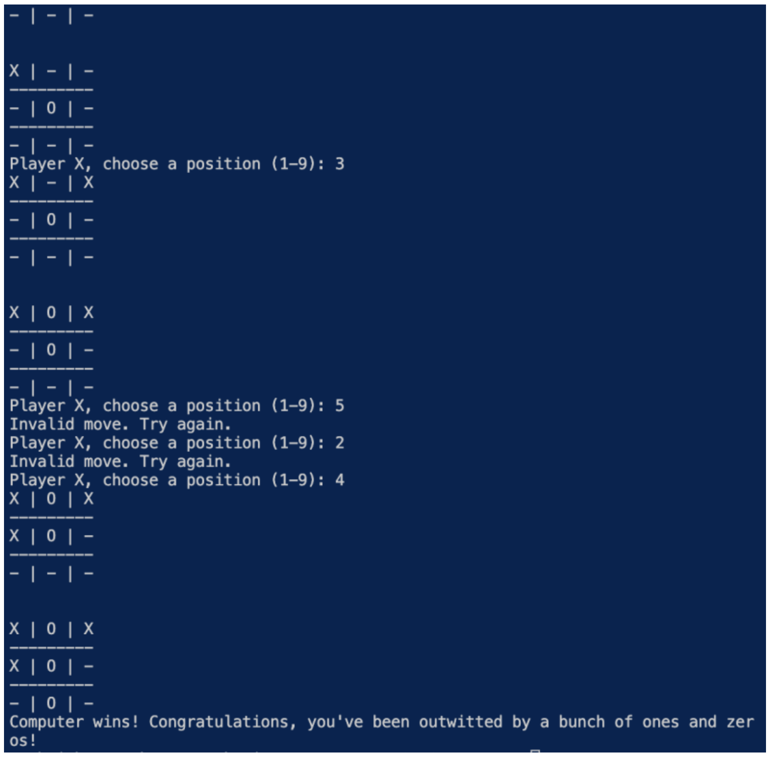
```
○ yashmishra@Yashs-MacBook-Air INDIVIDUAL_PROJECT_FINAL % /usr/local/bin/python3 /
  Users/yashmishra/Desktop/INDIVIDUAL_PROJECT_FINAL/tic_tac_toe_indi.py
  Do you want to play against a computer? (yes/no):
```

If the user selects yes, then the computer starts playing with the user countering every move of the user. If the user tries to play the position that them or the computer has already played, it will show an invalid move and prompt again as shown in the picture below.



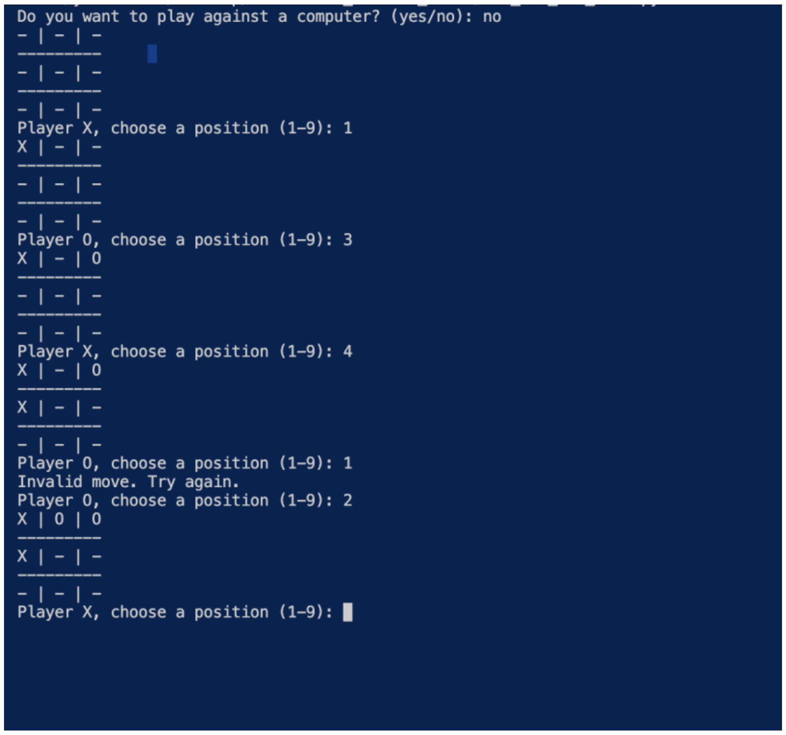
```
- | - | -
---------
- | - | -
---------
- | - | -
Player X, choose a position (1-9): 1
X | - | -
---------
- | - | -
---------
- | - | -


X | - | -
---------
- | O | -
---------
- | - | -
Player X, choose a position (1-9): 3
X | - | X
---------
- | O | -
---------
- | - | -


X | O | X
---------
- | O | -
---------
- | - | -
Player X, choose a position (1-9): 5
Invalid move. Try again.
Player X, choose a position (1-9): 2
Invalid move. Try again.
Player X, choose a position (1-9): 
```

The game will keep on continuing as per the rules until and unless the computer/user wins or draws. And at the end, the code will display who won and end the program.



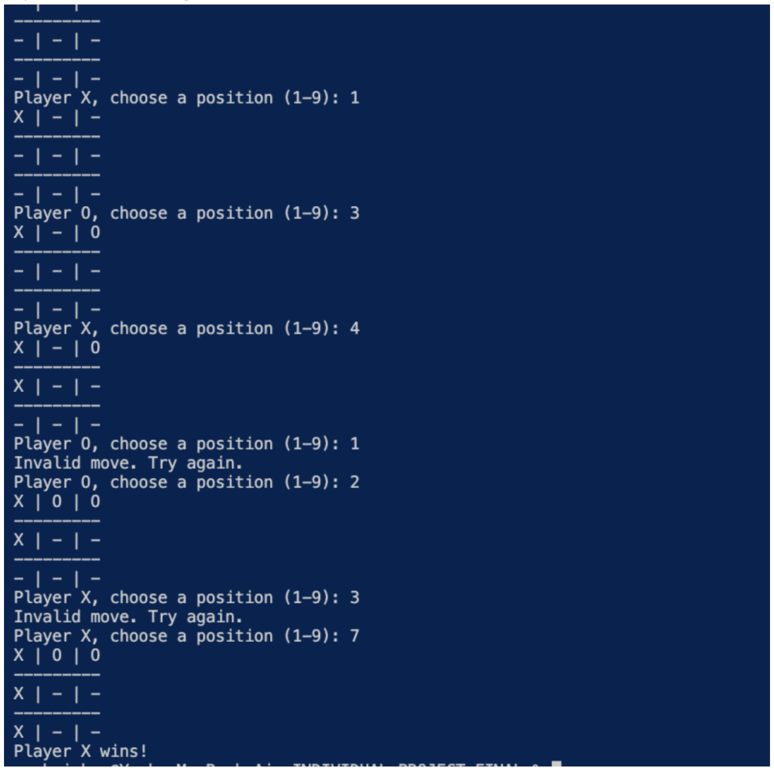
```
- | - | -


X | - | -
---------
- | O | -
---------
- | - | -
Player X, choose a position (1-9): 3
X | - | X
---------
- | O | -
---------
- | - | -


X | O | X
---------
- | O | -
---------
- | - | -
Player X, choose a position (1-9): 5
Invalid move. Try again.
Player X, choose a position (1-9): 2
Invalid move. Try again.
Player X, choose a position (1-9): 4
X | O | X
---------
X | O | -
---------
- | - | -


X | O | X
---------
X | O | -
---------
- | O | -
Computer wins! Congratulations, you've been outwitted by a bunch of ones and zer
os!
```

On the other hand, the user also has a choice to select no if they don't want to play against the computer. If they select no, then the program m immediately designates two players Player X and Player O and then continues to play the game in the same rule. Invalid moves are also pointed out and it allows both the users to try again.



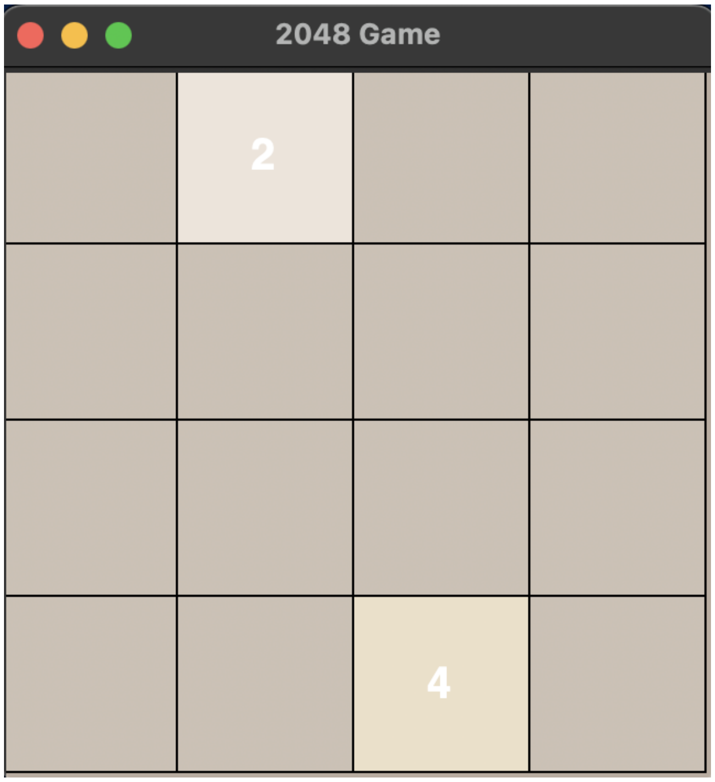
```
Do you want to play against a computer? (yes/no): no
- | - | -
---------
- | - | -
---------
- | - | -
Player X, choose a position (1-9): 1
X | - | -
---------
- | - | -
---------
- | - | -
Player O, choose a position (1-9): 3
X | - | O
---------
- | - | -
---------
- | - | -
Player X, choose a position (1-9): 4
X | - | O
---------
X | - | -
---------
- | - | -
Player O, choose a position (1-9): 1
Invalid move. Try again.
Player O, choose a position (1-9): 2
X | O | O
---------
X | - | -
---------
- | - | -
Player X, choose a position (1-9):
```

The game ends when either of the Player X or Player O wins or it is a draw. The program ends, by showcasing who won or if it is a draw.

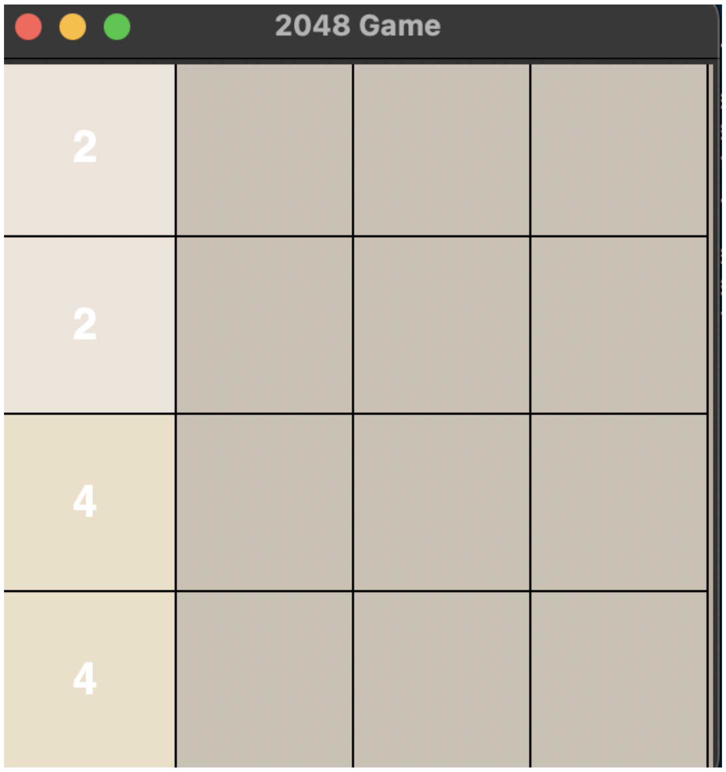


```
---------
- | - | -
---------
- | - | -
Player X, choose a position (1-9): 1
X | - | -
---------
- | - | -
---------
- | - | -
Player O, choose a position (1-9): 3
X | - | O
---------
- | - | -
---------
- | - | -
Player X, choose a position (1-9): 4
X | - | O
---------
X | - | -
---------
- | - | -
Player O, choose a position (1-9): 1
Invalid move. Try again.
Player O, choose a position (1-9): 2
X | O | O
---------
X | - | -
---------
- | - | -
Player X, choose a position (1-9): 3
Invalid move. Try again.
Player X, choose a position (1-9): 7
X | O | O
---------
X | - | -
---------
X | - | -
Player X wins!
```

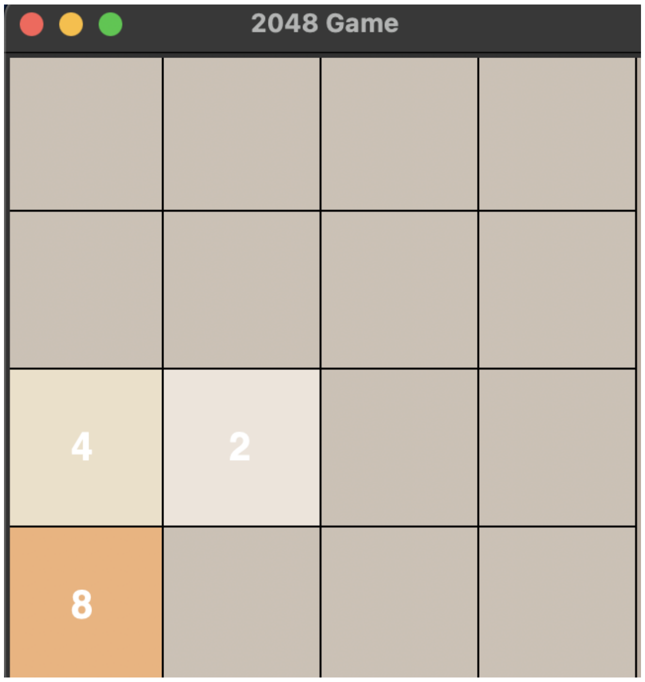
If the User decides to play 2048 instead, then as soon as they run the program the 2048 games pop up with a random allocation of 2's and 4's.



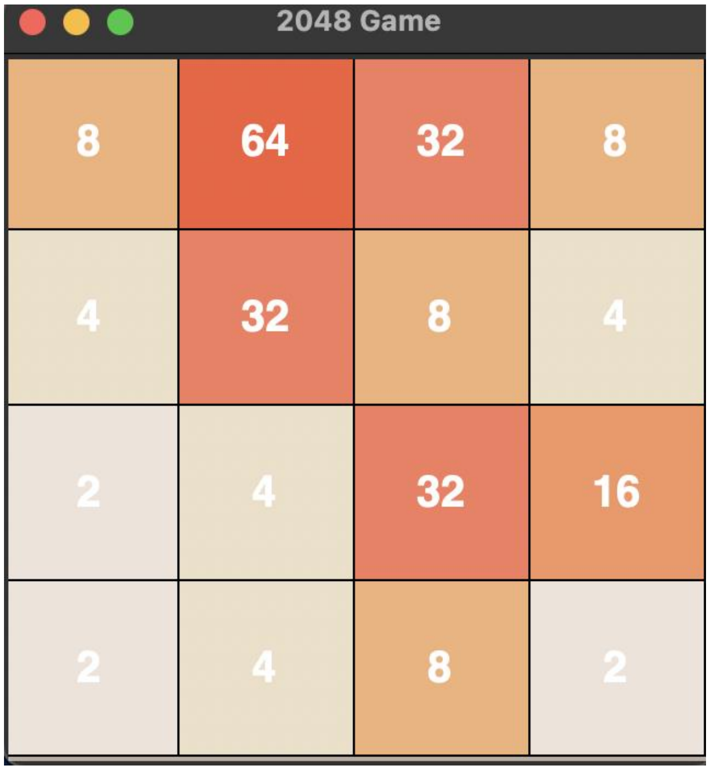
Then the User can use the arrow keys to move and generate more numbers per movement.



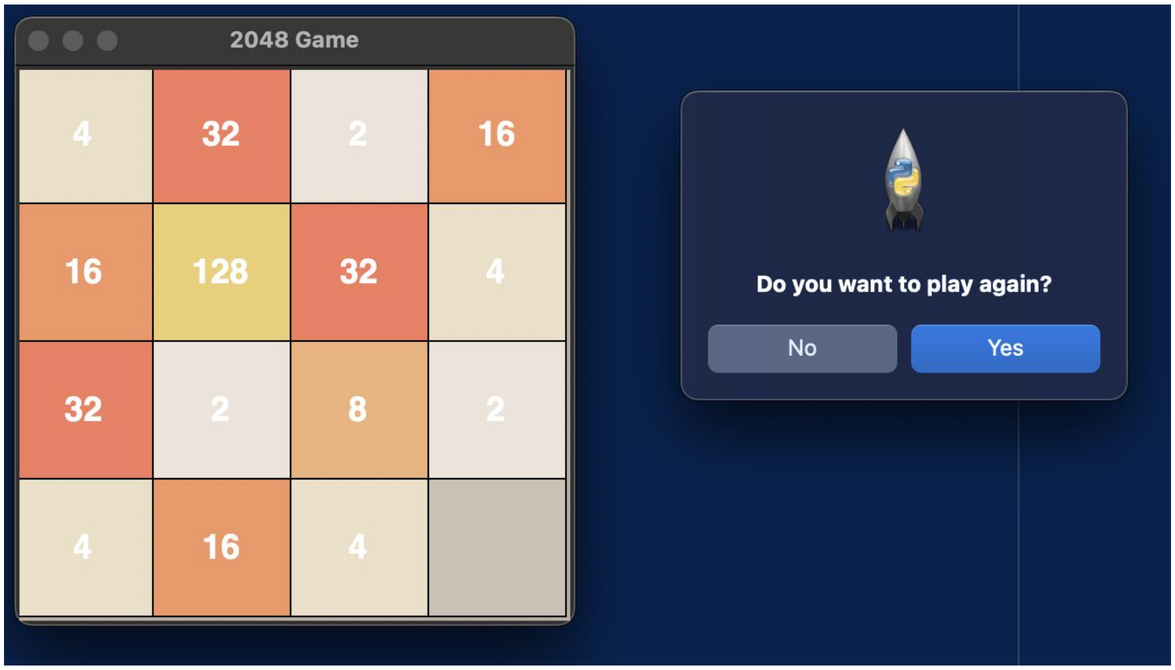
The aim of the game is to move the keys in such a way it adds up and slowly ends up in 2048. In the previous picture, using the downward key will add the similar values together and simultaneously generate more numbers.



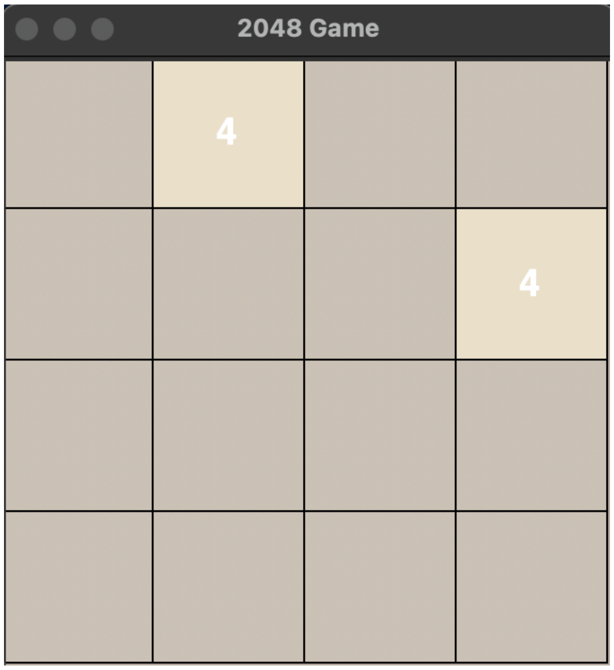
This continues on as you move with your arrow keys to get value of 2,4,8,16,32,64,128,256,512,1024 and finally 2048. The game in between will look something like this.



The game ends when you reach 2048 or when the entire 4 x 4 is filled with values and there is no possible way to reach 2048. At that moment, it displays game over and prompts user if they want to play the game again.



If the user selects yes, then the game restarts again with random allocation of 2's and 4's. Otherwise, the program ends.



That marks the end of the User Manual.

Appendix

Code for 2048 (2048.py):

```
import random
import copy
import tkinter as tk
from tkinter import messagebox

def initialize_board():
    # Initialize a 4x4 board with two random tiles
    board = [[0] * 4 for _ in range(4)]
    add_new_tile(board)
    add_new_tile(board)
    return board

def print_board(board):
    for row in board:
        print(" ".join(map(str, row)))

def add_new_tile(board):
    # Add a new tile (2 or 4) to a random empty cell
    empty_cells = [(i, j) for i in range(4) for j in range(4) if board[i][j] == 0]
    if empty_cells:
        i, j = random.choice(empty_cells)
        board[i][j] = random.choice([2, 4])

def move(board, direction):
    # Move the tiles in the specified direction
    original_board = copy.deepcopy(board)

    if direction == 'up':
        for j in range(4):
            col = [board[i][j] for i in range(4) if board[i][j] != 0]
            col = merge_tiles(col)
            for i in range(4):
                board[i][j] = col[i] if i < len(col) else 0

    elif direction == 'down':
        for j in range(4):
            col = [board[i][j] for i in range(3, -1, -1) if board[i][j] != 0]
            col = merge_tiles(col)
            for i in range(3, -1, -1):
                board[i][j] = col[3 - i] if 3 - i < len(col) else 0

    elif direction == 'left':
        for i in range(4):
```

```python
            row = [board[i][j] for j in range(4) if board[i][j] != 0]
            row = merge_tiles(row)
            for j in range(4):
                board[i][j] = row[j] if j < len(row) else 0

    elif direction == 'right':
        for i in range(4):
            row = [board[i][j] for j in range(3, -1, -1) if board[i][j] != 0]
            row = merge_tiles(row)
            for j in range(3, -1, -1):
                board[i][j] = row[3 - j] if 3 - j < len(row) else 0

    return original_board != board

def merge_tiles(line):
    # Merge adjacent identical tiles in a line
    merged_line = []
    i = 0
    while i < len(line):
        if i < len(line) - 1 and line[i] == line[i + 1]:
            merged_line.append(line[i] * 2)
            i += 2
        else:
            merged_line.append(line[i])
            i += 1
    return merged_line + [0] * (len(line) - len(merged_line))

def is_game_over(board):
    # Check if the game is over (no more moves)
    for i in range(4):
        for j in range(4):
            if board[i][j] == 0 or (i < 3 and board[i][j] == board[i + 1][j]) or \
            (j < 3 and board[i][j] == board[i][j + 1]):
                return False
    return True

def game_over():
    result = messagebox.askquestion("Game Over", "Do you want to play again?")
    return result == 'yes'

def reset_game():
    global board
    board = initialize_board()
    update_display()

def update_display():
    canvas.delete("all")
    for i in range(4):
        for j in range(4):
            value = board[i][j]
            color = get_tile_color(value)
            canvas.create_rectangle(j * 80, i * 80, (j + 1) * 80, (i + 1) * 80, fill=color, outline="black")
            if value != 0:
                canvas.create_text((j + 0.5) * 80, (i + 0.5) * 80, text=str(value), font=('Helvetica', 20, 'bold'))

    if is_game_over(board):
        if game_over():
            reset_game()

def get_tile_color(value):
    colors = {
        0: "#CDC1B4",
        2: "#EEE4DA",
        4: "#EDE0C8",
        8: "#F2B179",
        16: "#F59563",
        32: "#F67C5F",
        64: "#F65E3B",
        128: "#EDCF72",
        256: "#EDCC61",
        512: "#EDC850",
        1024: "#EDC53F",
        2048: "#EDC22E",
    }
    return colors.get(value, "#CDC1B4")


root = tk.Tk()
root.title("2048 Game")


board = initialize_board()

canvas = tk.Canvas(root, width=320, height=320, bg="#BBADA0")
canvas.pack()

def handle_key(event):
    direction = None
    if event.keysym in ['Up', 'Down', 'Left', 'Right']:
        direction = event.keysym.lower()
    if direction:
        if move(board, direction):
            add_new_tile(board)
            update_display()
```

```
root.bind('<Up>', handle_key)
root.bind('<Down>', handle_key)
root.bind('<Left>', handle_key)
root.bind('<Right>', handle_key)

update_display()
root.mainloop()
```

Code for tic-tac-toe (tic_tac_toe_indi.py):

```
def print_board(board):
        for i in range(0, 9, 3):
            print(" | ".join(board[i:i+3]))
            if i < 6:
                print("-" * 9)

def check_win(player, board):
        winning_combinations = [(0, 1, 2), (3, 4, 5), (6, 7, 8),
                                (0, 3, 6), (1, 4, 7), (2, 5, 8),
                                (0, 4, 8), (2, 4, 6)]

        for combo in winning_combinations:
            if all(board[i] == player for i in combo):
                return True
        return False

def check_tie(board):
        return "-" not in board

def player_input(player, board):
        while True:
            try:
                move = int(input(f"Player {player}, choose a position (1-9): "))
                if 1 <= move <= 9 and board[move - 1] == "-":
                    return move - 1
                else:
                    print("Invalid move. Try again.")
            except ValueError:
                print("Invalid input. Enter a number between 1 and 9.")

def switch_player(player):
        return "X" if player == "O" else "O"

def player_game():
        board = ["-" for _ in range(9)]
```

```
    current_player = "X"
    game_running = True

    # Start the game loop
    while game_running:
        print_board(board)
        move = player_input(current_player, board)
        board[move] = current_player

        if check_win(current_player, board):
            print_board(board)
            print(f"Player {current_player} wins!")
            game_running = False
        elif check_tie(board):
            print_board(board)
            print("It's a tie!")
            game_running = False
        else:
            current_player = switch_player(current_player)

def computer_game():
    def computer_move(board, current_player):
        if current_player == "O":
            best_score = -float("inf")
            best_move = None

            for i in range(9):
                if board[i] == "-":
                    board[i] = "O"
                    score = minimax(board, 0, False)
                    board[i] = "-"
                    if score > best_score:
                        best_score = score
                        best_move = i

            return best_move

    def minimax(board, depth, is_maximizing):
        scores = {"X": -1, "O": 1, "tie": 0}
        winner = None

        if check_win("X", board):
            winner = "X"
        elif check_win("O", board):
            winner = "O"
        elif check_tie(board):
            winner = "tie"
        if winner:
            return scores[winner]

        if is_maximizing:
            best_score = -float("inf")
            for i in range(9):
                if board[i] == "-":
                    board[i] = "O"
                    score = minimax(board, depth + 1, False)
                    board[i] = "-"
                    best_score = max(score, best_score)
            return best_score
        else:
            best_score = float("inf")
            for i in range(9):
                if board[i] == "-":
                    board[i] = "X"
                    score = minimax(board, depth + 1, True)
                    board[i] = "-"
                    best_score = min(score, best_score)
            return best_score

    # Initialize game variables
    board = ["-" for _ in range(9)]
    user_choice = "yes"

    computer_playing = user_choice.lower().startswith("y")

    if computer_playing:
        current_player = "X"
        computer_player = "O"
    else:
        current_player = "X"

    game_running = True

    # Start the game loop
    while game_running:
        print_board(board)

        if current_player == computer_player:
            move = computer_move(board, current_player)
            print("")
            print("")
        else:
            move = player_input(current_player, board)

        board[move] = current_player
```

```python
            if check_win(current_player, board):
                print_board(board)
                if computer_playing and current_player == computer_player:
                    print("Computer wins! Congratulations, you've been outwitted by a bunch of ones and zeros!")
                else:
                    print(f"Player {current_player} wins! Savior of the humanity")
                game_running = False
            elif check_tie(board):
                print_board(board)
                print("It's a tie! You can't win, This is the best you will do!!!")
                game_running = False
            else:
                current_player = switch_player(current_player)

if __name__ == "__main__":
        user_choice = input("Do you want to play against a computer? (yes/no): ").lower()
        if user_choice == "yes":
            computer_game()
        elif user_choice == "no":
            player_game()
        else:
            print("Invalid choice. Please enter 'yes' or 'no'.")
```