LUDO like UCSC

-Take Home Assignment-

Submitted By

K.R.Sandamini

23001722

SCS 1301
Data Structures and Program Design Using C

September 1, 2024

University of Colombo
School of Computing

Table of Contents

1 Introduction **1**

1.1 Overview of Ludo 1

1.2 Purpose of the Simulation 1

2 The structures used to represent the board and pieces **2**

2.1 Types Definition 2

2.2 Constants and Macros 2

2.3 The `Player` Structure 3

2.4 Design Decisions 4

2.4.1 Structure Design Rationale 4

3 Justification for the used structures **5**

3.1 Constants and Macros Justification 5

3.1.1 Color Codes 5

3.1.2 Board Dimensions 5

3.1.3 Home Path Positions 5

3.2 Player Structure Justification 6

4 Functions used in my gameplay logic **7**

4.1 Functions 7

4.1.1 rollDice() 7

4.1.2 initializePlayerStatus() 7

4.1.3 computeNewPosition() 7

4.1.4 capturePiece() 7

4.1.5 yellowPlayerStrategy() 7

4.1.6 bluePlayerStrategy() 7

4.1.7 redPlayerStrategy() 7

4.1.8 greenPlayerStrategy() 7

4.1.9 checkPlayerWin() 7

4.2 Explaining the functions 8

4.2.1 Rolling the dice 8

4.2.2 Identifying the pieces position 8

4.2.3 Describing pieces new positions 8
4.2.4 Capturing the pieces 9
4.2.5 Describing yellow player's behavior 9
4.2.6 Describing blue player's behavior 9
4.2.7 Describing red player's behavior 10
4.2.8 Describing green player's behavior 10
4.2.9 Determine whether the current player has won the game 11

5 The efficiency of the program 12
5.1 Efficiency of `rollDice` Function 12
5.2 Efficiency of `initializePlayerStatus` Function 13
5.3 Efficiency of `computeNewPosition` Function 14
5.4 Efficiency of `capturePiece` Function 16
5.5 Efficiency of `yellowPlayerStrategy` Function 17
5.6 Efficiency of `bluePlayerStrategy` Function 19
5.7 Efficiency of `redPlayerStrategy` Function 20
5.8 Efficiency of `greenPlayerStrategy` Function 22
5.9 Efficiency of `checkPlayerWon` Function 24

6 Conclusion 25

Chapter 1

Introduction

1.1 Overview of Ludo

Ludo is a classic board game that traces its origins to the ancient Indian game Pachisi. The game is designed for two to four players, each of whom is assigned a distinct color: Red (R), Yellow (Y), Green (G), or Blue (B). The objective of Ludo is to move all of one's pieces from the starting base, through the board, and into the home area, while potentially capturing opponents' pieces along the way. The Ludo board features a unique layout consisting of standard cells, color-specific cells, and designated areas such as the base, the starting square 'X', the approach cells, and the home straight.

The game employs a six-sided dice to determine the number of cells a piece can move during each turn. A critical aspect of Ludo is the strategic movement of pieces to either advance towards the home area or to capture opponents' pieces, which sends those pieces back to their respective bases. The interaction between players and the dice introduces a mix of luck and strategy, making Ludo a game of both chance and skill.

1.2 Purpose of the Simulation

The primary purpose of the simulation is to create a programmatic model of the Ludo game that operates autonomously without user intervention. By simulating the game, we aim to analyze and understand the dynamics of Ludo gameplay, evaluate different player strategies, and determine outcomes based on predefined player behaviors. This simulation will allow us to test and refine game strategies, investigate the effectiveness of various player behaviors, and ensure that the game rules are correctly implemented and followed.

This document serves as a guide for the development and evaluation of the Ludo simulation, providing the necessary background, requirements, and implementation details to ensure a successful execution of the project.

Chapter 2

The structures used to represent the board and pieces

2.1 Types Definition

The `types.h` file defines the fundamental data structures and constants used in the Ludo game simulation. This file includes the necessary structures to represent the players, their pieces, and their positions on the Ludo board. Below is a detailed explanation of the constants and the `Player` structure.

2.2 Constants and Macros

- **Color Codes:**
  - `#define YELLOW "\x1b[33m"`: Defines the ANSI escape code for yellow color.
  - `#define BLUE "\x1b[34m"`: Defines the ANSI escape code for blue color.
  - `#define RED "\x1b[31m"`: Defines the ANSI escape code for red color.
  - `#define GREEN "\x1b[32m"`: Defines the ANSI escape code for green color.
  - `#define RESET "\x1b[0m"`: Resets the color formatting in the console output.
- **Board Dimensions:**
  - `#define CELLS 52`: Defines the total number of cells in the standard path on the Ludo board.
  - `#define HOME_PATH 6`: Defines the length of the home path, which is the number of cells from the approach to the home.
  - `#define STANDARD_PATH_SIZE 52`: Alias for the total number of cells in the standard path.

- **Home Path Positions:**
  - `#define YELLOW_HOME_BEGIN 1`: The starting cell for the yellow player's home path.
  - `#define BLUE_HOME_BEGIN 14`: The starting cell for the blue player's home path.
  - `#define RED_HOME_BEGIN 27`: The starting cell for the red player's home path.
  - `#define GREEN_HOME_BEGIN 40`: The starting cell for the green player's home path.

2.3 The `Player` Structure

The `Player` structure is designed to store all the relevant information for each player in the Ludo game. It includes details such as the color of the player, the status and positions of their pieces, and their progress on the home path.

```
typedef struct Player
{
    char color[15];
    char pieces[4][3];
    int loc[4];
    int X;
    int homeStartCell;
    bool inHomePath[4];
    int homePathProgress[4];
} Player;
```

- `char color[15];`: Stores the color of the player as a string (e.g., "YELLOW", "BLUE").
- `char pieces[4][3];`: Stores the identifiers for each of the four pieces belonging to the player. Each piece is represented by a 3-character string (e.g., "R1", "G2").
- `int loc[4];`: An array that stores the current location of each piece on the board. The location is represented as an index corresponding to the cells on the board.
- `int X;`: Stores the starting square index (often the entry point from the base to the standard path).
- `int homeStartCell;`: The starting cell of the player's home path. It is defined using one of the home path position macros (e.g., `YELLOW_HOME_BEGIN`).

- `bool inHomePath[4];`: A boolean array indicating whether each piece has entered the home path.
- `int homePathProgress[4];`: Tracks the progress of each piece in the home path. This is typically an index from 0 to 5, representing the piece's position from the approach to home.

2.4 Design Decisions

2.4.1 Structure Design Rationale

In the current implementation of the Ludo game, only the 'Player' structure is defined, while structures for pieces, board, and game are not explicitly created. This design choice is based on the following considerations:

- **Simplified Design**: By focusing on the 'Player' structure, the implementation remains straightforward. This approach simplifies the management of player-specific data such as color, pieces, and positions, reducing the complexity of the code.
- **Modularity and Flexibility**: The 'Player' structure encapsulates all necessary player-related information, including pieces and movement logic. This modular approach allows for easy updates and maintenance, as player-specific data and behavior are centralized.
- **Reuse of Data**: The current design utilizes arrays and flags within the 'Player' structure to manage piece locations and game states. This avoids redundant structures and keeps the data closely related to the player, improving data coherence and access efficiency.
- **Future Expansion**: Although additional structures for pieces, board, or game might be beneficial for more complex scenarios, the current design prioritizes essential functionality. Future expansions can introduce these structures as needed without significant changes to the existing code.

Chapter 3

Justification for the used structures

3.1 Constants and Macros Justification

3.1.1 Color Codes

The color codes are defined using ANSI escape sequences to enhance console output readability by color-coding text. This aids in visually distinguishing player-related information, enhancing the overall user experience.

3.1.2 Board Dimensions

- `CELLS`: Defines the total number of standard cells on the board, ensuring that any operations involving cell indexing or movement adhere to the board's structure.
- `HOME_PATH`: Specifies the length of the home path, facilitating calculations related to player movement into their home area.
- `STANDARD_PATH_SIZE`: Provides an alias for `CELLS` to ensure consistency and clarity in code, allowing easy updates if board dimensions change.

3.1.3 Home Path Positions

- `YELLOW_HOME_BEGIN`, `BLUE_HOME_BEGIN`, `RED_HOME_BEGIN`, `GREEN_HOME_BEGIN`: These macros define the starting positions for each player's home path. They centralize the home path definitions, simplifying updates and ensuring consistency in referencing home path locations throughout the code.

These constants and macros are employed to ensure clarity, maintainability, and ease of modifications, while supporting clear and consistent gameplay logic.

3.2 Player Structure Justification

The `Player` structure offers several advantages for managing game data:

- `char color[15];`: Stores the player's color, aiding in distinguishing players.
- `char pieces[4][3];`: Holds identifiers for each piece, facilitating easy tracking and reference.
- `int loc[4];`: Tracks each piece's current position on the board, crucial for movement and gameplay logic.
- `int X;`: Defines the entry point from the base to the board, simplifying piece movement calculations.
- `int homeStartCell;`: Marks the start of the home path, ensuring correct path calculations.
- `bool inHomePath[4];`: Indicates if a piece is in the home path, which is vital for applying home path rules.
- `int homePathProgress[4];`: Monitors progress within the home path, allowing accurate tracking of piece advancement.

This structure supports efficient management of player data and gameplay mechanics.

Chapter 4

Functions used in my gameplay logic

4.1 Functions

4.1.1 rollDice()

4.1.2 initializePlayerStatus()

4.1.3 computeNewPosition()

4.1.4 capturePiece()

4.1.5 yellowPlayerStrategy()

4.1.6 bluePlayerStrategy()

4.1.7 redPlayerStrategy()

4.1.8 greenPlayerStrategy()

4.1.9 checkPlayerWin()

4.2 Explaining the functions

4.2.1 Rolling the dice

```
int rollDice()
```

Purpose: This function simulates the roll of a six-sided die. It generates a random integer between 1 and 6, representing the outcome of the dice roll.

Implementation: The function utilizes the `rand()` function to produce a random integer. It then adjusts this integer to fall within the range of 1 to 6 by taking the modulus of 6 and adding 1.

Use Case: The `rollDice` function is essential for determining the number of spaces a player's piece will move on the game board based on the dice roll.

4.2.2 Identifying the pieces position

```
void initializePlayerStatus(Player player)
```

Purpose: This function displays the current status of a player's pieces on the board. It provides information about the location of each piece, whether it is in the base, on the board, or in the home path.

Implementation: The function iterates through the player's pieces and counts how many are in the base and on the board. It then prints the location of each piece:

- Pieces in the base are noted.
- Pieces on the board show their cell location.
- Pieces in the home path display their progress.

Use Case: The `initializePlayerStatus` function helps in visualizing the player's current standing and understanding the position of each piece on the board.

4.2.3 Describing pieces new positions

```
int computeNewPosition(Player *player, int pieceIndex, int diceRoll)
```

Purpose: This function calculates the new position of a player's piece based on the dice roll. It handles different scenarios such as moving from the base, advancing on the board, or progressing within the home path.

Implementation:

- If the piece is in the base, it moves to the starting position.
- If the piece is on the board, the new position is computed by adding the dice roll to the current position and adjusting for circular movement around the board.

- If the piece is in the home path, it updates the progress within the path, ensuring the move does not exceed the home path length.

Use Case: The `computeNewPosition` function is essential for updating a piece's position according to the dice roll and ensuring valid moves within the game rules.

4.2.4 Capturing the pieces

```
void capturePiece(Player *currentPlayer, Player players[], int pieceIndex)
```

Purpose: Handles the logic for capturing an opponent's piece.

Implementation:

- The function checks if any opponent's piece occupies the same position as the current player's piece.
- If a capture is possible, the opponent's piece is sent back to the base.
- The captured piece's status is reset, and a message indicating the capture is displayed.

Use Case: This function adds the competitive element of capturing opponent pieces, a key aspect of Ludo gameplay.

4.2.5 Describing yellow player's behavior

```
int yellowPlayerStrategy(Player *yellowPlayer, Player players[], int diceRoll)
```

Purpose: Defines the strategy for the yellow player, who prioritizes winning by advancing pieces closer to the home.

Implementation:

- The function first checks if any piece can be moved out of the base with a roll of six.
- It then identifies the piece that is nearest to the home path or has the least distance to reach home.
- The selected piece is the one that can move closest to home with the given dice roll.

Use Case: The `yellowPlayerStrategy` function supports a strategy focused on winning by quickly advancing pieces toward the home path.

4.2.6 Describing blue player's behavior

```
int bluePlayerStrategy(Player *bluePlayer, int diceRoll)
```

Purpose: Defines the strategy for the blue player, who uses a random and unpredictable movement strategy.

Implementation:

- The function selects a piece to move based on a cyclic pattern determined by the dice roll value.
- If the selected piece is in the base and the dice roll is a six, the piece is moved out of the base.
- If the piece can legally move to a new position on the board, that piece is chosen.

Use Case: The `bluePlayerStrategy` function embodies a strategy for the blue player that favors unpredictability, making it harder for opponents to anticipate the blue player's moves.

4.2.7 Describing red player's behavior

```
int redPlayerStrategy(Player *redPlayer, Player players[], int diceRoll)
```

Purpose: Defines the strategy for the red player, who is aggressive and prioritizes capturing opponent pieces.

Implementation:

- The function first checks if any piece can be moved out of the base with a roll of six.
- It then evaluates the potential to capture opponent pieces by calculating new positions after the dice roll.
- If a capture is possible, the function returns the index of the piece that can capture.
- If no capture is possible, it selects an alternative piece to move.

Use Case: The `redPlayerStrategy` function implements an aggressive strategy for the red player, focusing on capturing opponent pieces to gain an advantage.

4.2.8 Describing green player's behavior

```
int greenPlayerStrategy(Player *greenPlayer, int diceRoll)
```

Purpose: Defines the strategy for the green player, who aims to create or maintain blocks to hinder opponents.

Implementation:

- The function first checks if any piece can be moved out of the base with a roll of six.
- It then evaluates the potential to create or maintain a block by checking if two pieces can occupy the same position.
- If no block is possible, it selects the first valid piece to move.

Use Case: The `greenPlayerStrategy` function supports a defensive strategy for the green player, focusing on blocking opponents to control the board.

4.2.9 Determine whether the current player has won the game

```
bool checkPlayerWin(Player *currentPlayer)
```

Purpose: Checks if the current player has won the game by moving all their pieces into the home path.

Implementation:

- The function iterates through all of the player's pieces.
- It checks whether each piece has fully completed its progress on the home path.
- If all pieces have completed the home path, the function returns `true`; otherwise, it returns `false`.

Use Case: The `checkPlayerWin` function determines whether a player has won the game, which is a crucial part of ending the game and declaring a winner.

Chapter 5

The efficiency of the program

5.1 Efficiency of `rollDice` Function

```
int rollDice()
{
    return (rand() % 6) + 1;
}
```

Time Complexity: The `rollDice` function operates with a time complexity of $\mathcal{O}(1)$, meaning it executes in constant time. This is because the function performs a small, fixed number of operations regardless of the input size.

Efficiency Considerations:

- **Randomness Quality:** The efficiency of this function is highly dependent on the underlying implementation of the `rand()` function. While `rand()` generates pseudo-random numbers quickly, it may not provide the highest quality of randomness for more complex simulations. For simple use cases like simulating a dice roll, it is generally sufficient.
- **Modulo Operation:** The modulo operation (`% 6`) ensures that the output is within the desired range (1 to 6). This operation is efficient and has a constant time complexity, contributing minimally to the overall execution time.
- **Deterministic Output Range:** The function guarantees a uniform distribution of output across the six possible values (1 through 6), assuming that the `rand()` function itself provides a uniform distribution. This makes it an efficient solution for simulating a fair six-sided die.

5.2 Efficiency of initializePlayerStatus Function

```
void initializePlayerStatus(Player player)
{
    int boardPieceCount = 0;
    int basePieceCount = 0;
    printf("%s Pieces locations\n", player.color);
    printf("================================================================\n");

    for (int i = 0; i < 4; i++)
    {
        if (player.loc[i] == -1)
        {
            basePieceCount++;
        }
        else if (player.inHomePath[i])
        {
            printf(RED "%s ----> %sHomePath[%d] \n" RESET, player.pieces[i], player.color
            ,player.homePathProgress[i]);
            boardPieceCount++;
        }
        else
        {
            printf("%s ----> cell L%d \n", player.pieces[i], player.loc[i]);
            boardPieceCount++;
        }
    }
    printf("\n%s player now has %d/4 pieces on the board and %d/4 pieces on the base.\n",
    printf("================================================================\n");
}
```

Time Complexity: The initializePlayerStatus function operates with a time complexity of $\mathcal{O}(n)$, where n is the number of pieces each player has (in this case, $n = 4$). The function iterates through each piece exactly once, making a constant number of checks and print operations for each piece. Since n is a small constant (4), the function's execution time is predictable and minimal.

Efficiency Considerations:

- **Loop Efficiency:** The function's loop runs for a predetermined number of iterations (4 in this case), performing basic conditional checks and displaying statements. Each iteration consists of checking the condition of one piece (whether it is in the base, on the board, or in the home path) and updating counts accordingly.

- **Conditional Checks:** The conditional checks (`if`, `else if`, `else`) within the loop are efficient because they include simple comparisons and boolean tests. Because the number of checks is limited, the overhead is low.
- **I/O Operations:** The function extensively uses `printf` commands to display the status of the player's pieces. Although `printf` actions are generally efficient, they can be time-consuming compared to other operations, especially if the function is invoked repeatedly or in a tight loop. However, because the function is most commonly used to display status information at specific periods in the game (for example, after a player's turn), the influence on overall performance is minimal.

5.3 Efficiency of `computeNewPosition` Function

```
int computeNewPosition(Player *player, int pieceIndex, int diceRoll)
{
    int currentPos = player->loc[pieceIndex];
    int newPos = currentPos;

    if (player->inHomePath[pieceIndex])
    {
        int updatedProgress = player->homePathProgress[pieceIndex] + diceRoll;
        if (updatedProgress > HOME_PATH)
        {
            return -1; // Cannot move, the move would exceed the home path
        }
        player->homePathProgress[pieceIndex] = updatedProgress;
        return player->homeStartCell; // We return homeStart as a flag that the piece is
        in home path
    }
    else if (currentPos == -1)
    {
        // Moving the piece from the base to the starting cell
        return player->X;
    }
    else
    {
        newPos = (currentPos - 1 + diceRoll) % CELLS + 1;

        // Check if piece can enter the home path
        int distanceToHomeStart = (player->homeStartCell - currentPos + CELLS) % CELLS;
```

```
        if (diceRoll >= distanceToHomeStart)
        {
            player->inHomePath[pieceIndex] = true;
            player->homePathProgress[pieceIndex] = diceRoll - distanceToHomeStart + 1;
            return player->homeStartCell; // We return homeStart as a flag that the
            piece is in home path
        }
    }

    return newPos;
}
```

Time Complexity: The `computeNewPosition` function operates with a time complexity of $\mathcal{O}(1)$. This means that the function executes in constant time regardless of the values of the input parameters. The function consists of a series of conditional checks and arithmetic operations, all of which are performed in a fixed amount of time.

Efficiency Considerations:

- **Conditional Branches:** This function efficiently handles three scenarios: whether the piece is in the home route, in the base, or on the board. Each possibility is handled by a different branch of if-else conditions, allowing the function to swiftly identify the appropriate behavior based on the current state.
- **Arithmetic Operations:** Simple addition, subtraction, and modulus operations are utilized to calculate the new position on the board and progress along the home path. Consistent execution of these processes improves the function's efficiency.
- **Home Path Handling:**If a piece is in the home route, the function examines if a dice roll would lead it to surpass the home path. If the function returns -1, it indicates that the move is invalid. This early exit increases the function's efficiency by eliminating needless computations.
- **Modulus Operation:** Efficiently calculates the new position on the board, ensuring proper wrap-around. The function calculates the distance to the home start cell effectively using modulus arithmetic, resulting in both accuracy and performance.

5.4 Efficiency of capturePiece Function

```
void capturePiece(Player *currentPlayer, Player players[], int pieceIndex)
{
    for (int i = 0; i < 4; i++)
    {
        // Check if the piece can be captured (same location, not in home path)
        if (strcmp(players[i].color, currentPlayer->color) == 0)
            continue;

        for (int j = 0; j < 4; j++)
        {
            if (players[i].loc[j] == currentPlayer->loc[pieceIndex] && !players[i].
            inHomePath[j] && !currentPlayer->inHomePath[pieceIndex])
            {
                printf("%s captures %s's piece %s!\n", currentPlayer->color, players[i].
                color, players[i].pieces[j]);
                // Move the captured piece back to the base
                players[i].loc[j] = -1;
                // Reset the captured piece's status regarding the home path
                players[i].inHomePath[j] = false;
                players[i].homePathProgress[j] = 0;
            }
        }
    }
}
```

Time Complexity: The capturePiece function has a time complexity of $\mathcal{O}(n^2)$, where n is the number of players.

Efficiency Considerations:

- **Conditional Checks:** The function avoids redundant comparisons by checking if the current player is being compared to themself with the strcmp function. This early exit condition prevents redundancy in operations.
- **Nested Loops:** The nested loops may appear inefficient, but given the limited number of iterations in a regular Ludo game, the function executes these checks efficiently. However, this strategy may not scale effectively if the number of players or pieces increases dramatically.
- **Memory Usage:** The function consumes minimum memory, confined to loop counters and temporary variables, resulting in good space efficiency.

- **Print Statements:** The `printf` statements, while useful for debugging and providing user feedback, do not contribute to the overall time complexity of the function as they are executed only when a capture occurs. The time taken by these I/O operations can be considered negligible relative to the overall function execution time.

5.5 Efficiency of `yellowPlayerStrategy` Function

```
int yellowPlayerStrategy(Player *yellowPlayer, Player players[], int diceRoll)
{
    for (int i = 0; i < 4; i++)
    {
        if (yellowPlayer->loc[i] == -1 && diceRoll == 6)
        {
            return i;
        }
    }

    int nearestToHome = -1;
    int smallestDistance = CELLS;

    for (int i = 0; i < 4; i++)
    {
        if (yellowPlayer->loc[i] != -1) // If the piece is not in the base
        {
            int newPosition = computeNewPosition(yellowPlayer, i, diceRoll);
            if (newPosition == -1)
                continue;

            int distanceHome;
            // If the piece is in the home path, calculate the distance to the end of
            the path
            if (yellowPlayer->inHomePath[i])
            {
                distanceHome = HOME_PATH - yellowPlayer->homePathProgress[i];
            }
            else
            {
                distanceHome = (yellowPlayer->homeStartCell - newPosition + CELLS)
                % CELLS;
            }
```

```
            if (distanceHome < smallestDistance)
            {
                nearestToHome = i;
                smallestDistance = distanceHome;
            }
        }
    }

    return nearestToHome;
}
```

Time Complexity: The `yellowPlayerStrategy` function has a time complexity of $\mathcal{O}(n)$, where n is the number of pieces a player has, which is typically 4 in a standard Ludo game. The function performs two distinct loops over the player's pieces:

- **First Loop:** The first loop iterates through all of the player's pieces to check if any piece is in the base (represented by a location of -1) and if a 6 was rolled. This loop has a time complexity of $\mathcal{O}(n)$, and it exits early if a valid piece is found, making it efficient in cases where a 6 is rolled.
- **Second Loop:** The second loop also iterates over the player's pieces, but this time it evaluates the distance of each piece to the home cell. The loop's complexity is again $\mathcal{O}(n)$. During each iteration, the function calls `computeNewPosition`, which is typically a constant time operation, making the overall time complexity for the second loop still $\mathcal{O}(n)$.

Since both loops iterate over the same set of pieces and are not nested, the overall time complexity of the function remains $\mathcal{O}(n)$.

Efficiency Considerations:

- **Early Exit in the First Loop:** The method is designed to return instantly if a piece in the base can be removed by rolling a 6. This avoids superfluous processing and assures that the function can rapidly handle the most basic circumstance.
- **Distance Calculation:**The second loop's logic efficiently calculates each piece's distance from the home route, guaranteeing that the function chooses the piece that is closest to winning. This consists of a simple modulus operation and a subtraction, both of which are efficient processes.
- **Function Calls:** The function calls `computeNewPosition` during each iteration of the second loop. Assuming `computeNewPosition` operates in constant time (as it generally does for small, fixed-size boards like Ludo), this does not significantly impact the function's overall efficiency.

5.6 Efficiency of bluePlayerStrategy Function

```
int bluePlayerStrategy(Player *bluePlayer, int diceRoll)
{
    for (int i = 0; i < 4; i++)
    {
        int pieceToMove = (i + diceRoll) % 4; // Cyclically select a piece index based
        on the dice roll

        // Check if the selected piece is in the base
        if (bluePlayer->loc[pieceToMove] == -1)
        {
            if (diceRoll == 6)
                return pieceToMove;
            continue;
        }

        int newPosition = computeNewPosition(bluePlayer, pieceToMove, diceRoll);
        if (newPosition != -1)
        {
            return pieceToMove;
        }
    }

    return -1;
}
```

Time Complexity: The `bluePlayerStrategy` function has a time complexity of $\mathcal{O}(n)$, where n is the number of pieces a player has (typically 4 in a standard Ludo game). Here's a breakdown of the function's execution:

- **Loop Iteration:** The function contains a single loop that iterates over the player's four pieces. Therefore, the loop executes up to 4 times, giving a linear time complexity of $\mathcal{O}(n)$.
- **Cyclic Piece Selection:** The expression `(i + diceRoll) % 4` is used to select pieces cyclically based on the dice roll. This selection is done in constant time, $\mathcal{O}(1)$, for each iteration.
- **Base Check and Movement Calculation:** Within the loop, the function first checks whether the selected piece is in the base (constant time, $\mathcal{O}(1)$). If the piece is not in the base, the function calls `computeNewPosition`, which typically operates in constant time, $\mathcal{O}(1)$.

- **Early Return:** The function is designed to return immediately when it finds a valid piece to move, which can happen before the loop completes all four iterations. This early exit can optimize performance, particularly when a valid move is found quickly.

Efficiency Considerations:

- **Cyclic Selection of Pieces:** The cyclic selection of pieces depending on dice roll assures that the strategy considers all pieces equally over numerous turns. This cyclic strategy prevents bias against specific pieces while ensuring that all pieces are considered for movement in a balanced manner.
- **Base Check with Dice Roll:** When a 6 is rolled, the function quickly determines if a piece can be removed from the base. Because this is a regular circumstance in Ludo, the function's ability to respond rapidly is critical for sustaining game flow.
- **Function Call to `computeNewPosition`:** The call to `computeNewPosition` during each loop iteration is a critical part of the function's decision-making process. As `computeNewPosition` generally operates in constant time, it does not negatively impact the overall efficiency of the strategy function.

5.7 Efficiency of `redPlayerStrategy` Function

```
int redPlayerStrategy(Player *redPlayer, Player players[], int diceRoll)
{
    int targetIndex = -1;      // Index of the piece that can capture an opponent's piece
    int alternativeIndex = -1; // Index of the fallback piece to move if no capture is
    possible

    // Loop through all four pieces of the aggressive player
    for (int i = 0; i < 4; i++)
    {
        // Check if the piece is in the base (-1 indicates in the base)
        if (redPlayer->loc[i] == -1)
        {
            // If the dice roll is 6, the piece can be moved out of the base
            if (diceRoll == 6)
                return i;
            continue;
        }

        // Calculate the new position of the piece after the dice roll
        int newPosition = computeNewPosition(redPlayer, i, diceRoll);
```

```
        if (newPosition == -1)
            continue;

        // Loop through all competitors to check for possible captures
        for (int j = 0; j < 4; j++)
        {
            // Skip the check for the aggressive player's own pieces
            if (strcmp(players[j].color, redPlayer->color) == 0)
                continue;

            // Loop through all pieces of the current competitor
            for (int k = 0; k < 4; k++)
            {
                // Check if the new position of the aggressive player's piece matches a
                competitor's piece
                // and if neither piece is in their respective home paths
                if (newPosition == players[j].loc[k] && !players[j].inHomePath[k] &&
                !redPlayer->inHomePath[i])
                {
                    targetIndex = i;
                    return targetIndex;
                }
            }
        }

        // If no capture is possible, consider this piece as a alternative option
        if (alternativeIndex == -1)
        {
            alternativeIndex = i;
        }
    }

    return alternativeIndex;
}
```

Time Complexity: The `redPlayerStrategy` function has a time complexity of $\mathcal{O}(n^3)$, where n represents the number of pieces each player controls (typically 4 in a standard Ludo game). Here's a detailed breakdown:

- **Outer Loop:** The function begins by looping over all 4 pieces of the `redPlayer`. This outer loop runs 4 times, contributing $\mathcal{O}(n)$ to the overall time complexity.

- **Capture Check Nested Loop:** Within the outer loop, the function performs a capture check by iterating over all 4 pieces of each of the other 3 players. This involves two nested loops:
  - The first nested loop iterates over each of the other 3 players, contributing an additional $\mathcal{O}(n)$ complexity.
  - The second nested loop iterates over each piece of the selected player, again contributing $\mathcal{O}(n)$ complexity.

  As a result, the nested loops contribute a combined complexity of $\mathcal{O}(n^2)$ to the overall function.
- **Combined Complexity:** Given that the capture check (nested loops) runs within the outer loop, the overall complexity becomes $\mathcal{O}(n) \times \mathcal{O}(n^2) = \mathcal{O}(n^3)$.

Efficiency Considerations:

- **Aggressive Strategy Optimization:** This function focuses on capturing an opponent's piece, which is a key Ludo strategy. It swiftly identifies and returns the piece capable of capturing, allowing for speedy action when a capture is possible.
- **Fallback Strategy:** If capturing is not possible, the function offers a fallback strategy for selecting another piece to move. This ensures that the player's turn is employed efficiently, even when there is no risk of capture.
- **High Complexity Due to Multiple Loops:** The three nested loops required for checking potential captures across all opponents add significant temporal complexity. In instances with additional players or pieces, this could result in performance bottlenecks. However, in a regular Ludo game with four pieces per player, the complexity is reasonable.

5.8 Efficiency of `greenPlayerStrategy` Function

```
int greenPlayerStrategy(Player *greenPlayer, int diceRoll)
{
    int blockCandidateIndex = -1; // Index of the piece that can potentially create or
    maintain a block
    int validMoveIndex = -1;      // Index of the first valid movable piece found

    for (int i = 0; i < 4; i++) // Iterate over all four pieces of the green player
    {
        if (greenPlayer->loc[i] == -1)
        {
```

```
            if (diceRoll == 6)
                return i;
            continue;
        }

        int newPosition = computeNewPosition(greenPlayer, i, diceRoll);
        if (newPosition == -1)
            continue;

        // Check if this move creates or maintains a block
        for (int j = 0; j < 4; j++)
        {
            if (i != j && greenPlayer->loc[j] == newPosition)
            {
                blockCandidateIndex = i;
                break;
            }
        }

        if (blockCandidateIndex != -1)
            break;

        if (validMoveIndex == -1)
        {
            validMoveIndex = i;
        }
    }

    return (blockCandidateIndex != -1) ? blockCandidateIndex : validMoveIndex;
}
```

Time Complexity: The `greenPlayerStrategy` function has a time complexity of $\mathcal{O}(n^2)$, where n represents the number of pieces each player controls (typically 4 in a standard Ludo game). The complexity analysis is as follows:

- **Outer Loop:** The function begins by iterating over all four pieces of the `greenPlayer`. This outer loop runs 4 times, contributing $\mathcal{O}(n)$ to the overall time complexity.
- **Block Check Nested Loop:** Within the outer loop, the function checks whether moving a piece would create or maintain a block. This check involves another loop that iterates over the remaining three pieces of the same player, resulting in $\mathcal{O}(n)$ complexity for this nested loop.

- **Combined Complexity:** Given that the block check runs inside the outer loop, the overall time complexity of the function is $\mathcal{O}(n) \times \mathcal{O}(n) = \mathcal{O}(n^2)$.

Efficiency Considerations:

- **Blocking Strategy:** The function is designed to prioritize creating or maintaining a block, a common defensive tactic in Ludo. A block occurs when two pieces occupy the same cell, preventing opponent pieces from passing. The function quickly returns the index of a piece that can contribute to a block if such a move is found.
- **Valid Move Fallback:** If no block-creating move is possible, the function falls back to returning the index of the first valid movable piece. This ensures that the player's turn is used efficiently, even if a block cannot be formed.

5.9 Efficiency of `checkPlayerWon` Function

```
bool checkPlayerWon(Player *currentPlayer)
{
    for (int i = 0; i < 4; i++)
    {
        if (!currentPlayer->inHomePath[i] || currentPlayer->homePathProgress[i] <
        HOME_PATH)
        {
            return false;
        }
    }
    return true;
}
```

Time Complexity: $\mathcal{O}(n)$, where n is the number of pieces (typically 4). This is because the function iterates through all the player's pieces to check if they have all reached the end of the home path.

Chapter 6

Conclusion

In this Ludo game implementation, we created a robust simulation of the traditional board game, focusing on the distinct strategies of each player type (Red, Green, Yellow, and Blue). The game elements, like as dice rolls, piece movement, capturing, and victory conditions, have been meticulously designed to simulate real-world games. The game's efficient algorithms and well-structured code create a tough and entertaining experience, emphasizing the significance of strategy and decision-making in determining the outcome. This implementation is a comprehensive example of mixing game theory with programming to build a useful and amusing application.