Course: DESIGN and ANALYSIS of ALGORITHM

Project Proposal

Hassan Tasdique Abbasi	**45897**
Muhammmad Wasif Qamar	**45815**
Muneeb-ur-Rehman	**48046**
Umer Shahmeer Ahmad	**46194**

Project Name: **TIC-TAC-TOE**

1. Introduction

- **Overview**: This is a Tic-Tac-Toe game implemented in C++ with a WinAPI graphical user interface. The purpose of the project is to apply algorithm design and analysis techniques to a basic game, evaluating time and space complexity for each function.
- **Objective**: The goal is analyze the computational complexity, best/average/worst-case scenarios, and using asymptotic notations for complexity classification.

2. Functions Used:

- `main()`: Initializes the application window and runs the message loop to keep the program active.
- `WndProc(HWND, UINT, WPARAM, LPARAM)`: Handles events like button creation, button clicks, and window closing. This is the core event-handler function.
- `updateBoard(HWND hwnd)`: Updates the Tic-Tac-Toe grid in the GUI by setting text for each button to reflect the current board state.
- `resetBoard()`: Resets the game board and initializes the `square` array with the initial values for a new game.
- `checkwin()`: Checks if there is a winning combination on the board, if the game is a draw, or if it is still ongoing.

3. FlowChart:

4. Function's Analysis:

`main()`:

- **Time Complexity**: `O(1)` — Initializing the window and running the message loop is constant-time work.
- **Space Complexity**: `O(1)` — No dynamic allocation beyond a fixed-size message loop and stack memory for function calls.

`WndProc()`:

- **Time Complexity**: `O(1)` per event handled (though it is repeatedly called for each event).
- **Space Complexity**: `O(1)` — Uses fixed memory for local variables and parameters.

`updateBoard()`:

- **Time Complexity**: `O(9) ≈ O(1)` — Loops through each button (constant 9) and updates text.
- **Space Complexity**: `O(1)` — Uses constant space for temporary variables.

`resetBoard()`:

- **Time Complexity**: `O(9) ≈ O(1)` — Updates 9 elements in the `square` array.
- **Space Complexity**: `O(1)` — Only modifies an existing array without additional allocation.

`checkwin()`:

- **Time Complexity**: `O(8)` — Checks 8 possible winning conditions (constant).
- **Space Complexity**: `O(1)` — Uses fixed space for the `win` array and a few local variables.

5. Complexity Analysis for the Entire Program

- **Overall Time Complexity**: The game operations (such as button clicks and board updates) are all constant time, `O(1)`, due to the fixed board size.
- **Overall Space Complexity**: `O(1)`, as the program only requires fixed memory for the board state and buttons.

6. Best, Average, and Worst Cases

- **Best Case**: Player wins after the minimum moves (5 moves total). `checkwin` runs fewer times, but overall complexity remains `O(1)` due to the constant board size.
- **Average Case**: Player wins after 7 moves, a typical game length. Complexity still `O(1)`.

- **Worst Case**: Game goes to a draw after 9 moves, triggering `checkwin` after each move. Complexity remains `O(1)`.

7. Prior and Posterior Analysis

- **Prior Analysis**: Before implementation, it's clear that a Tic-Tac-Toe game requires minimal computational resources because of the fixed 3x3 grid and limited move options. The main focus was on handling user events and managing GUI components.
- **Posterior Analysis**: After implementation, it's confirmed that all functions operate in constant time, with no dynamic memory allocation or scalability concerns due to the fixed board size.

8. Asymptotic Notations

- Each function operates within **constant time complexity** — `O(1)` — because the number of operations is fixed due to the 3x3 grid size.
- **Space complexity** is also **constant (**`O(1)`**)**, as the program doesn't grow in memory use with game progression.

9. Conclusion

In conclusion, this Tic-Tac-Toe project demonstrates the analysis of a straightforward game algorithm where computational complexity remains minimal due to the fixed 3x3 board size. Each function within the program operates with a constant time and space complexity of **O(1)**, making it highly efficient and predictable. By analyzing each function's best, average, and worst-case scenarios, it is clear that the algorithm is unaffected by variable input sizes, reinforcing its simplicity and efficiency.

The project provides a practical example of applying algorithm analysis techniques, such as prior and posterior analysis, and highlights the importance of understanding asymptotic notations in software development, even in small-scale applications. This analysis can serve as a foundational approach for more complex games or applications where board size or game rules may vary, necessitating more intricate algorithmic strategies.