Deliverable 1: Project Proposal & Conceptual Design
Data Structures and Algorithms (ES221)

Group Members:
Engineer Muhammad Komail (2023460)
Engineer Irtiza (2023653)
Engineer Shahram Khan Saleh (2023648)

Submission Date: 16 March 2025

Project Proposal & Conceptual Design

Course Details

- **Course Code:** ES221
- **Course Title:** Data Structures and Algorithms
- **Deliverable:** 1 - Project Proposal & Conceptual Design
- **Submission Date:** 16 March 2025
- **Group Members:**
  - Engineer Muhammad Komail (2023460)
  - Engineer Irtiza (2023663)
  - Engineer Shahram Khan Saleh (2023648)

Objective

Students must define their project idea, identify key data structures, and outline the initial algorithmic approach.

1. Project Title & Problem Statement

Project Title: Sudoku Solver Interacting with Compiler Terminals

Problem Statement:

Sudoku is a popular logic-based puzzle game that challenges players to fill a 9×9 grid with digits (1–9) such that each row, column, and 3×3 subgrid contains all digits exactly once. Solving Sudoku manually can be time-consuming and error-prone, especially for complex puzzles. Additionally, players often want tools to generate puzzles, track progress, and receive hints—all within an interactive terminal interface. This project aims to address these challenges by developing a "Sudoku Solver Interacting with Compiler Terminals," a console-based application that:

- Generates solvable Sudoku puzzles of varying difficulty.
- Allows users to input custom puzzles and validates their solvability.
- Provides interactive features like making moves, adding notes, requesting hints, and viewing progress.
- Solves puzzles automatically using efficient algorithms, with options to apply or revert solutions.

The problem is significant because it combines game logic with practical applications of data structures and algorithms, offering an engaging way to explore DSA concepts while providing a useful tool for puzzle enthusiasts.

2. Data Structures Selection

The project implements the following key data structures:

- **Stack:**
  - **Implementation:** Custom template class (`Stack<T>`) using a linked list.
  - **Purpose:** Tracks move history or backtracking steps (e.g., for undo functionality, though currently underutilized).
  - **Justification:** Provides LIFO behavior, ideal for reversing actions or maintaining a history of states.
- **Vector:**
  - **Implementation:** Used in the `Sudoku` class (`vector<vector<Cell>> grid`) and `ProgressTracker` (`vector<ProgressEntry>`).
  - **Purpose:** Represents the 9×9 Sudoku grid and stores progress entries dynamically.
  - **Justification:** Offers dynamic sizing, random access ($O(1)$ for grid operations), and ease of iteration, making it suitable for grid and log storage.
- **Linked List:**
  - **Implementation:** Embedded within the `Stack` class as a singly linked list (`Node` structure).
  - **Purpose:** Supports dynamic memory allocation for stack operations.
  - **Justification:** Efficient for insertions and deletions at the top ($O(1)$), aligning with stack requirements.
- **CustomString:**
  - **Implementation:** A custom string class for managing text data (e.g., progress logs).
  - **Purpose:** Provides string manipulation for logging actions with concatenation operators.
  - **Justification:** Demonstrates manual memory management and string handling, a key DSA learning objective.

3. Algorithmic Approach

The project employs the following main algorithms:

- **Sudoku Generation Algorithm:**
  - **Process:**
    - Fills initial 3×3 subgrids with shuffled numbers (1–9).
    - Uses backtracking to complete the grid into a valid solution.
    - Removes cells based on difficulty level (40 for easy, 50 for medium, 60 for hard).
  - **Time Complexity:** $O(9^m)$, where m is the number of empty cells during solving, though generation is optimized by starting with a partial solution.
- **Backtracking Solver Algorithm (`solveSudoku`):**
  - **Process:**
    - Finds an empty cell using `findEmpty`.
    - Tests numbers 1–9, checking validity with `isValid` (row, column, and 3×3 subgrid checks).
    - Recursively solves the remaining grid; backtracks if no solution is found.
  - **Time Complexity:** $O(9^m)$ in the worst case, where m is the number of empty cells, but practical performance is faster due to early pruning.
- **Validity Checking Algorithm (`isValid`):**
  - **Process:** Checks a number's uniqueness in its row, column, and 3×3 subgrid.
  - **Time Complexity:** $O(1)$ per check (fixed 9 iterations per dimension).
- **Progress Tracking:**
  - **Process:** Logs actions with timestamps and durations using `ProgressTracker`.
  - **Time Complexity:** $O(1)$ per entry addition, $O(n)$ for printing the report, where n is the number of entries.

4. Input & Output Design

Input Design:

- **Source:** User input via compiler terminal (console) using `cin` and `getline`.
- **Types:**
  - Menu selections (e.g., "1" for new game, "3" for making a move).
  - Puzzle creation inputs (row, column, value triples, e.g., "0 0 5").

  - Move inputs (row, column, value, e.g., "3 4 7").
  - Difficulty settings (1–3).
- **Validation:**
  - Ensures inputs are within bounds (0–8 for grid, 1–9 for values).
  - Handles invalid inputs with error messages and re-prompting.

Processing:

- Puzzle generation uses backtracking and cell removal.
- Moves are validated against Sudoku rules (`isValid`).
- Hints and solutions leverage the solver algorithm.
- Progress is tracked with timestamps and action logs.

Output Design:

- **Format:** Console output using `cout`.
- **Examples:**
  - **Grid Display:**

	0	1	2	3	4	5	6	7	8
0	5	3	.	.	7	.	.	.	.
1	6	7	.	.	1	9	5	.	.

  - **Progress Report:**
    * Total time played: 2 minutes 15 seconds
    * Moves made: 25
    * Hints used: 2
    * Mistakes made: 1
    * Detailed Activity Log: [timestamp] - Placed 5 at position 0,0
  - Success/failure messages (e.g., "Move successful", "Puzzle completed!").
- **Features:** Clear formatting with separators (e.g., "—" for subgrids) and user-friendly prompts.

5. Course Concepts Application

This project integrates key DSA concepts from the course:

- **Stacks:** Demonstrates LIFO behavior in the custom `Stack` class, with potential for move history or undo features (currently implemented but underutilized). Reinforces dynamic memory management and linked list operations.
- **Vectors:** Used as the core grid structure, showcasing dynamic arrays and 2D data management. Applied in progress tracking, emphasizing iteration and storage efficiency.
- **Linked Lists:** Embedded in the `Stack` class, teaching pointer manipulation and memory allocation.
- **Backtracking:** The solver algorithm applies recursion and backtracking, a fundamental technique for constraint satisfaction problems like Sudoku.
- **Time Complexity:** Analysis of $O(9^m)$ for solving and $O(1)$ for grid operations reinforces complexity analysis skills.
- **Object-Oriented Programming:** Classes like `Sudoku`, `ProgressTracker`, and `CustomString` demonstrate encapsulation, abstraction, and operator overloading.