Ethan Smith

Professor Kanemoto

[CPSC-39-12111](https://mccd.instructure.com/courses/34063)

30 November 2024

Final Programming Project Report

**Overview of the Game:**

The Millionaire Trivia Game is a quiz-based application where players answer multiple-choice questions, aiming to reach one million dollars by correctly answering ten questions in a row. The game is designed to be both entertaining and educational, offering an interactive way to test the players’ knowledge. The game presents questions in random order, ensuring a unique experience with each playthrough, and keeps track of the user’s progress with an updating scoring tower. If ten questions in a row are correctly answered, the user wins. If they get a question wrong, the game tells them what the correct answer was, then exits the program.

**Algorithms:**

**Algorithm 1: Creating and Displaying Answer Buttons**This algorithm handles the creation and display of the answer buttons in the game interface. It begins by creating a JButton for each possible answer, with customized attributes such as background color, font, and border for a visually appealing design. Inside each button, a JLabel is used to allow for text wrapping, ensuring that longer answer choices fit within the button without errors. The label is formatted using HTML to enable the word wrapping functionality. A hover effect is added to the button to improve the interactivity of the game; when the user hovers over the button, its background color changes to give visual feedback. Additionally, an action listener is implemented, so when the button is clicked, the handleQuestionAndAnswer method is triggered, passing the index of the selected answer to process the user's response. This algorithm is efficient and runs in constant time (O(1)), as it performs a fixed number of operations to create and set up each button, regardless of other factors in the game. I used ChatGPT in a sense when I knew what I wanted to implement, but due to my lack of knowledge in javax.swing, didn’t know how to implement it. I also used this for other parts of my GUI design but did not have ChatGPT write the code for the method. After attempting to create the code myself, I would then ask ChatGPT if I had implemented the code correctly, or if there was a way to make it more efficient.

A screen shot of a computer program

Description automatically generated

**Algorithm 2: Handling Questions, Checking Answers, and Game Flow**  
This algorithm controls the game's flow by displaying the current question, verifying the user's answer, and updating the game state accordingly. When the user selects an answer, the method checks if the chosen answer matches the correct one stored in the current question. If the answer is correct, the score increments and a confirmation message is displayed. If the answer is incorrect, the game ends with a message showing the correct answer. Then it checks to see if the player has reached the last level, winning them the $1,000,000. Additionally, the game updates the scoring tower to reflect the user's progress, changing the color of the labels corresponding to the user's score. The method then loads the next question and updates the interface with new question text and answer choices. The game continues until there are no more questions, at which point a message is displayed, and the program exits. This process is recursive, although not traditional recursion, as it repeatedly calls itself with updated data after each user interaction. The time complexity of this algorithm is O(1), as each operation runs in constant time. As previously stated I also used ChatGPT for small parts of this method. Primarily in ensuring the method was recursive, as I previously had separate methods for the question and answer aspects, which would call upon each other. This was not recursive so I asked for assistance in combining the two methods into one, thus calling itself if a correct answer is chosen.

A screen shot of a computer program

Description automatically generated

**Algorithm 3: Setting Up the Scoring Tower**

This algorithm is responsible for dynamically creating and displaying the scoring tower that represents the levels of monetary prizes the player can win as they progress through the game. The tower consists of multiple labels, each representing a different prize level, such as $1,000,000, $500,000, and so on, down to $250. The algorithm begins by defining an array of prize amounts and uses a JPanel with a GridLayout to evenly space out the labels vertically. Each label is customized with specific font settings, colors, and borders to enhance the visual appeal. The labels are then added to the panel, which is positioned on the right side of the screen using a BorderLayout to ensure the layout aligns with the rest of the game interface. The panel is given a fixed width and height. This algorithm enhances the user interface by providing a visual representation of the player’s progress, making it easier to track their winnings. The time complexity of this algorithm is O(N), where N is the number of prize levels, since the algorithm loops through the prize levels array once to create and display the labels. Knowing previous methods, this method was considerably easier to create, and I did not need assistance in doing so.

A screen shot of a computer code

Description automatically generated

**Big O Analysis:**The time complexity of the algorithms in this game is generally efficient. The first algorithm, which handles shuffling the questions, has a time complexity of O(N), where N is the number of questions in the list. This is because the shuffle operation involves iterating through the entire list to randomize the order of the questions. The second algorithm, which checks whether the user’s answer is correct, operates in constant time, O(1), as it simply compares the user’s selected answer to the correct one, a straightforward string comparison. Similarly, the third algorithm, which displays the current question and its possible answers, also operates in constant time, O(1). This is because the algorithm retrieves the next question and updates the display with a fixed number of answer options, which does not depend on the size of the list of questions.

**Data Structures Used:**

LinkedList<Question>:

The game uses a LinkedList to store the questions. This data structure was chosen because it allows for efficient removal of elements after a question has been asked. By using a LinkedList, we avoid the need to shift elements as we would in an ArrayList, which can be costly in terms of time. When a question is displayed and answered, it is removed from the list, ensuring that it won’t be asked again.

ArrayList<Question>:

An ArrayList was used temporarily to shuffle the questions. Since ArrayList allows for quick random access, shuffling a LinkedList can at best be done in O(N) time. After the shuffle, the questions are converted back to a LinkedList for efficient removal, as ArrayLists must shift all elements after removal, making it O(N) time.

Array for Scoring Tower Levels:  
The game uses a String[] array to store the prize levels displayed in the scoring tower. This array allows for constant-time access to each value, making it efficient for managing and displaying the prize tiers. Its fixed size ensures easy retrieval by index, without the complexity of a more elaborate data structure.

**Opportunity:**

One opportunity I encountered was during the randomization of the questions. Initially, I planned to shuffle the questions manually by reordering the list, but I realized that Java’s built-in Collections.shuffle() function provides an efficient and reliable way to shuffle the list. I converted the LinkedList into an ArrayList so that the shuffling would be more efficient (O(1)), then converted it back into a LinkedList for more efficient question removal (O(1)).

**Error Resolution:**

I faced an error when trying to remove questions from the LinkedList. The issue arose because I was trying to remove these elements after they had already been shuffled into the list. The solution was to ensure that questions were shuffled first, then presented to the user, and removed only after they had been asked. This fixed the problem of duplicates and allowed the game to continue without re-asking questions. Another issue I faced was the scoring tower not filling the entirety of the right side of the window. This was due to poor layout management, and the buttons and scoring tower were competing for space. This was resolved by implementing a horizontal split plane, allowing for the scoring tower to fill the lower er right portion of the window.

**Future Improvements:**

For the next version of the game, I would like to fix the scoring tower to occupy both the upper and lower of the east side of the window. I would like to make the GUI more graphically appealing, and lastly I would like to incorporate the three lifelines used in Who Want’s To Be A Millionaire. These would be the 50:50, Phone a Friend and Ask the Audience. These were my priorities to implement next but I underestimated how long making the GUI would take.