Coursework #2

Module's Information:

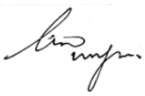
Module	PRG3205 Data Structure and Algorithms		
Session	August 2024		
Programme	BCSI		
Lecturer	**AP. Ts. Dr. Rajermani A/P Thinakaran**		
	Email: rajermani.thina@newinti.edu.my Phone: 06-7982000 - 2209	Room: A3-15	
Coursework Type	Assignment (Max 5 members per group)		
Percentage	40% out of 100%		
Hand-out Date	Week 6 – 24 September 2024		
Due Date	Week 13 – 17 November 2024	Presentation Date	TBC

Students' Declaration:

We declare that:

- ***We understand what is meant by plagiarism***
- ***This assignment is all our own work, and we have acknowledged any use of the published or unpublished works of other people.***
- ***We hold a copy of this assignment which we can produce if the original is lost or damaged.***

[Name/ID] Teo Chung Henn I22022496 [Signature] 

[Name/ID] Liew Wen Yen I22022754 [Signature] 

[Name/ID] Chong Ken Ji I22022311 [Signature] 

[Date] 24 November 2024

Learning Outcomes Assessed:

LO3	Demonstrate lifelong learning skills by understanding the trade-offs between different implementation of data structures and algorithms.

Description of Coursework #2:

The current reality presents a series of challenges that will be difficult to overcome without global collaboration to promote sustainable development from a future-oriented perspective. The United Nations (UN), through the implementation of the Sustainable Development Goals (SDGs) and the 2030 Agenda for Sustainable Development, seeks to create a more equitable environment that can alleviate the existing difficulties in the world today. This is where education plays a fundamental role (Pineda-Martínez et al., 2023) and with the help of offline available technological resources and appropriate pedagogical strategies, the aim is to build an education oriented towards achieving the SDGs to achieve a fairer and more equitable world.

Figure 1 illustrate 17 SDGs and the details can be obtain in the following link https://sdgcompass.org/sdgs/.

Figure 1. 17 Sustainable Development Goals

Serious games are games developed for non-entertaining purposes. Application of serious games in educational context have attracted attention from researchers from many countries as it has been proved that can increase learner motivation (Kara, 2021).

In this assignment, you are required to propose and develop a digital game in serious game context and reflect to SDG 4 – Quality Education. The proposed game should be able to improve the players' knowledge of a particular topic. Remember that your digital game should consist of basic

features which are add, update/edit, search and delete records. In addition, you can include any other relevant features which are able to meet the objective of the game. For some ideas you can refer to some previous work (Yan and Sunderraman 2004; ChePa al et., 2013).

Your system should be developed using any algorithm combinations learned on this course. Justification required why you choose the algorithms by analyzing the performance of the algorithms. The implementation using array or linked list.

Additional Information – Timing of the algorithms

Time each of your algorithms on sorting **random** arrays of size N, 4N and 6N, for some large value of N.

```
double startTime = System.nanoTime();
System.out.println(startTime);
double endTime = System.nanoTime();
System.out.println(endTime);
double tD = endTime - startTime ;
System.out.println(tD);
```

Here's the basic technique for timing code:

The call System.nanoTime() requests the system to use the most accurate timer available. Many systems still use a millisecond clock, which means your nanosecond count may be rounded to the nearest 1000 nanoseconds. If something appears to take zero time, that probably means it took less than a millisecond.

Here are some tips for helping to get reasonable accuracy.

- Close all other applications, so you don't have the computer doing too many other things during the timings.
    - Reason: System.nanoTime() gives "wall clock" time, not just time spent in your program.
- "Warm up" your program by executing all your methods a few times before you start timing.
    - Reason: The compiler may delay actually compiling your code until it is needed, so the first execution may be much slower.
- Call System.gc() before you begin timing.
    - Reason: So, garbage collection doesn't happen during your timing runs.

- Run the code multiple times and take the average.
    - For best accuracy, throw out the best and worst times, and take the average of the remaining times.
- As much as possible, try to include in your timing only the code you want to time.
    - A few assignments, comparisons, etc., aren't worth worrying about.

References

ChePa, N., Alwi, A., Din, A.M. and Safwan, M., 2013, August. The application of neural networks and min-max algorithm in digital congkak. In *Proceedings of the 4th International Conference on Computing and Informatics, ICOCI* (pp. 201328-30).

Kara, N. (2021). A systematic review of the use of serious games in science education. *Contemporary Educational Technology*, *13*(2), ep295.

Pineda-Martínez M, Llanos-Ruiz D, Puente-Torre P, García-Delgado MÁ. Impact of Video Games, Gamification, and Game-Based Learning on Sustainability Education in Higher Education. *Sustainability*. 2023; 15(17):13032. https://doi.org/10.3390/su151713032

Yan Liu, D.G. and Sunderraman, R., 2004. Computer Games Improve Learning Chinese. *2011 2nd International Conference on Education and Management Technology. IPEDR vol.13 (2011) © (2011) IACSIT Press, Singapore*

Submission Guideline:

The assignment should be accompanied by a document of each in which should include the following:

Project Cover Sheet (Assignment 1st page)

- Marking Rubrics (Group & individual)
- Table of contents (with page numbers for easy reference)
- Title
- Abstract
- Introduction

    - Introduction
    - Problem Statement
    - Project Objectives
    - Project Scope
    - Limitation
    - Target Audience
- Literature Review
    - SDG
    - Game
    - Algorithms
- Methodology
- Development
    - Pseudocode to document program logic.
    - Screen Output
    - Other relevant information
- Testing
    - Test plan and Test data to be used with testing.
- Summary
- References

Upload the softcopy of the document and source code at the given Canvas link.
Submit hardcopy of the assignment during the presentation.

Each group is required to do a **20-minute** presentation and demo the developed digital game.
The presentation day and date will be announced later.

Report Format

a) Adhere to the following word processing requirements:
    - All heading should be capitalized, bold and left-aligned.
    - All paragraphs must be fully justified; add space before and after paragraph.

- All text must be typed in font type “Times New Roman”, text size 12.
- Include details for your name, student identification number, and assignment title in your front page.
- Do not include unnecessary information such as logo, date, and assignment title in the header and footer section of each page. You are only allowed to insert page number in the bottom right footer section of each page.

b) Remember to spell check your assignment.

c) Use default margin settings.

End of Coursework #2

PRG3205 Assignment Rubrics
Group Component

Criteria	Performance					Marks Given
	Poor (0 – 2 marks)	**Fair (3 – 4 marks)**	**Good (5 – 6 marks)**	**Very Good (7 – 8 marks)**	**Excellent (9 – 10 marks)**	
Understands the Problem and Requirements	Student's work shows incomplete understanding of problem and/or requirements.	Student's work shows slight understanding of problem and requirements.	Student's work shows understanding of problem and most requirements.	Student's work shows complete understanding of problem and all requirements.	Student recognizes potential conflicts b/t requirements and seeks clarification from client/user.	
Uses Appropriate Algorithms	Student 'hacks out' program with no thought to algorithm design.	Student chooses/ designs algorithm(s) that are incorrect.	Student chooses/ designs algorithm(s) that is/are correct but somewhat inefficient.	Student chooses/ designs efficient algorithm(s).	Student researches tradeoffs of different algorithms making sure the results is accurate and relevant.	
Design of logic (Flowchart / Pseudocode)	No evidence of any logic by student.	Student's work shows incomplete understanding on logic design.	Program has significant logic errors.	Program has slight logic errors that do not significantly affect the results.	Program is logically well designed.	
Designs Appropriate User Interface	Implements very poor I/O functionality.	Only implements basic I/O functionality.	Some concepts of 'user-friendly' I/O used (e.g. prompts on input & labels on output).	Uses well-designed 'user-friendly' I/O interface appropriate for task and client.	User-friendly' I/O interface (with GUI components - optional).	
Tests Program for Correctness	No evidence of any testing by student.	Evidence of only one case tested.	Evidence of a few cases tested.	Evidence of "typical cases tested, but only assuming valid inputs.	'Robust design' with extensive testing.	
Creativity	Zero creativity	Lack of creativity and originality	Lack of originality or Lack of creativity	Creative design but lack of originality.	Creative and original idea	
Documentation	Documentation merely contains the cover page and printout of the program code; no referencing.	Poor layout / flow; Missing some essential components within the documentation; Basic documentation standards not adhered to; No referencing.	Average layout / flow; No missing components of the documentation; Did some referencing but did not adhere to Harvard Name Referencing; Sample outputs available without any explanation.	Good layout / flow; No missing components of the documentation; Good documentation standards; Adhered to Harvard Name Referencing standards but with minor errors / omissions; Sample outputs available with some explanation.	Excellent layout / flow; No missing components of the documentation; Excellent documentation standards; Adhered to Harvard Name Referencing standards with no obvious errors / omissions; Sample outputs available with clear explanation.	
					Total Marks (70 Marks)	

Remarks:

PRG3205 Assignment Rubrics
Individual Component

Name: Teo Chung Henn ID: I22022496

Criteria	Performance					Marks Given
	Poor (0 – 2 marks)	Fair (3 – 4 marks)	Good (5 – 6 marks	Very Good (7 – 8 marks)	Excellent (9 – 10 marks)	
Presentation	Did not turn up for presentation; Did not know how to execute the system.	Barely able to explain the codes / work done; Had difficulty in executing the system.	Able to explain some codes / work done; Able to execute the system; Only able to explain own component of the system / solution.	Provided good explanation of the codes and work done; Able to execute the system; Able to explain how own component of the system works with at least one other component of the system / solution.	Provided excellent explanation of the codes / work done; Able to execute the system; Able to explain how own component of the system works with all other components of the system / solution; Able to show additional concepts / new ideas used in the solution.	
Question and Answer	Unable to provide answers to any of the questions asked.	Able to answer a few questions posed; Mostly inaccurate / illogical answers / explanation provided.	Able to answer some questions posed; Some accurate / logical answers / explanation provided; Reasonable answers given but with some hesitation	Able to answer most questions posed; Mostly accurate / logical answers / explanation provided; Reasonable answers given.	Able to answer all questions posed; Accurate / logical answers / explanation provided; Reasonable answers given with sound arguments and clear discussion.	
Contribution	Did not contribute at all to the project.	Minimal contribution to the project.	Average contribution to the project	Good contribution to the project.	Contributed the most to the project.	
					Total (30 Marks)	

Remarks:

PRG3205 Assignment Rubrics
Individual Component

Name: Liew Wen Yen ID: I22022754

Criteria	Performance					Marks Given
	Poor (0 – 2 marks)	Fair (3 – 4 marks)	Good (5 – 6 marks	Very Good (7 – 8 marks)	Excellent (9 – 10 marks)	
Presentation	Did not turn up for presentation; Did not know how to execute the system.	Barely able to explain the codes / work done; Had difficulty in executing the system.	Able to explain some codes / work done; Able to execute the system; Only able to explain own component of the system / solution.	Provided good explanation of the codes and work done; Able to execute the system; Able to explain how own component of the system works with at least one other component of the system / solution.	Provided excellent explanation of the codes / work done; Able to execute the system; Able to explain how own component of the system works with all other components of the system / solution; Able to show additional concepts / new ideas used in the solution.	
Question and Answer	Unable to provide answers to any of the questions asked.	Able to answer a few questions posed; Mostly inaccurate / illogical answers / explanation provided.	Able to answer some questions posed; Some accurate / logical answers / explanation provided; Reasonable answers given but with some hesitation	Able to answer most questions posed; Mostly accurate / logical answers / explanation provided; Reasonable answers given.	Able to answer all questions posed; Accurate / logical answers / explanation provided; Reasonable answers given with sound arguments and clear discussion.	
Contribution	Did not contribute at all to the project.	Minimal contribution to the project.	Average contribution to the project	Good contribution to the project.	Contributed the most to the project.	
					Total (30 Marks)	

Remarks:

FACULTY OF INFORMATION TECHNOLOGY (FIT)

PRG3205 Assignment Rubrics
Individual Component

Name: Chong Ken Ji ID: I22022311

Criteria	Performance					Marks Given
	Poor (0 – 2 marks)	**Fair (3 – 4 marks)**	**Good (5 – 6 marks**	**Very Good (7 – 8 marks)**	**Excellent (9 – 10 marks)**	
Presentation	Did not turn up for presentation; Did not know how to execute the system.	Barely able to explain the codes / work done; Had difficulty in executing the system.	Able to explain some codes / work done; Able to execute the system; Only able to explain own component of the system / solution.	Provided good explanation of the codes and work done; Able to execute the system; Able to explain how own component of the system works with at least one other component of the system / solution.	Provided excellent explanation of the codes / work done; Able to execute the system; Able to explain how own component of the system works with all other components of the system / solution; Able to show additional concepts / new ideas used in the solution.	
Question and Answer	Unable to provide answers to any of the questions asked.	Able to answer a few questions posed; Mostly inaccurate / illogical answers / explanation provided.	Able to answer some questions posed; Some accurate / logical answers / explanation provided; Reasonable answers given but with some hesitation	Able to answer most questions posed; Mostly accurate / logical answers / explanation provided; Reasonable answers given.	Able to answer all questions posed; Accurate / logical answers / explanation provided; Reasonable answers given with sound arguments and clear discussion.	
Contribution	Did not contribute at all to the project.	Minimal contribution to the project.	Average contribution to the project	Good contribution to the project.	Contributed the most to the project.	
					Total (30 Marks)	

Remarks:

Table of Contents

List of Figures

List of Tables

1. Assignment Abstract

Our project or educational game title – **Hang Em High**. Learning history has always been a struggle for many students. They often see it as boring and hard to remember. That is where our project, Hang Em High, comes in. This serious educational game redesigns the classic Hangman concept to make it a fun way for students to engage with Malaysian history. Players can choose between answering timed multiple-choice questions and guessing words based on hints. Gamification elements such as leaderboards, difficulty levels and streak-based scoring are used to turn traditionally passive learning process into an active and enjoyable experience.

We use multiple data structures like Array Lists, Queues and HashMap to efficiently manage questions, words and leaderboards, while algorithms like Merge Sort and Binary Search are implemented to handle leaderboard sorting and searching. With features like leaderboards, difficulty levels and streak-based scoring, the game keeps players hooked while they learn. What is most important is Hang Em High is built to be accessible, targeting Sustainable Development Goal 4 (SDG 4) which is Quality Education. It is free and runs on basic computers with a browser and internet connection. With the blending of gaming and education, we hope to make an impact by making history more exciting and helping students discover a love for learning.

2. Project Introduction

2.1. Introduction and Project Title

Gaming has always been popular among teenagers. But education can sometimes come off as boring. So, why not merge the two? In recent years, blending education with gaming has gained attention as a fresh way to boost learning outcomes. Games' engaging mechanics and interactive environments can make education more fun and effective. Imagine learning while you play – that's a win-win situation. Educators can create immersive experiences that not only grab students' attention but also help them understand and remember what they learn.

So that brings us to our project – an educational game called **Hang Em High**, inspired by the classic hangman concept. The title Hang Em High plays the traditional Hangman game, where players try to guess words or phrases while avoiding getting hung by making too many incorrect guesses. However, we are taking it even further by introducing another game type. In Hang Em High, players are able to take on two types of challenges: a question-based game and a word-guessing game. In the question-based game, players face multiple-choice questions with a 20-second timer. The score increased with each correct answer while racing against the clock as the hangman looms closer with every mistake made. The word-guessing game, on the other hand, challenges players to guess words based on hints. Similarly, it comes with a limited number of attempts before the hangman is drawn.

Hang Em High also features a leaderboard that tracks scores across all game types and levels. players can now compete to beat high scores on the leaderboard, which adds a little friendly competition. Who doesn't love a good challenge to motivate themselves? At least I am. The combination of gameplay and educational content turns the traditional, sometimes dull learning experience into a fun adventure. Players won't just sit back and absorb information in a passive manner; they will now actively engage with the material while having fun. History often seems boring, but with Hang Em High, we want to make it fun and help teenagers love history because we definitely do not want history to repeat itself.

2.2. Problem Statement

Our problem statement – Students find history subjects boring. They find them dull and challenging to pay attention to in class as well as retaining the information. While there are exceptions who genuinely enjoy history, the majority of our friends do not share this enthusiasm. There is a need for interactive tools or a way to make learning history fun and accessible while improving student's ability to retain information.

2.3. Project Objectives

- Create a Hangman-inspired game that helps players learn about Malaysian history.
- Use gamification techniques to keep players motivated and actively involved in the learning process.
- Use efficient and effective data structures to organize and manage various game elements like questions and scoring.
- Implement a leaderboard that tracks players' scores for each game type and difficulty level to ensure fair competition without mixing scores.
- Use engaging sound effects and visual elements to enhance overall gaming experience.

2.4. Project Scope

- Focused on a range of questions related to Malaysian history.
- Players participate in two types of games: a question-based game with multiple-choice questions and a timed word-guessing game.
- Each game includes three levels of difficulty – **easy** (ages 7-12, Year 4 – Year 6), **medium** (ages 13-15, Form 1 – Form 3) and **hard** (ages 16-18, Form 4 – Form 5).
- Includes a scoreboard to track player scores over time.
- Compatible for desktop browsers (PC and Mac).

2.5. Limitations

- Limited to Malaysian history questions, which may not engage all students.
- Focused on single-player mode.

2.6. Target Audience

- Students at both primary and secondary levels who are currently studying history.
- Educators who are seeking interactive tools to teach history.

3. Literature Review

3.1. Sustainable Development Goals

This project supports **Sustainable Development Goal 4 (SDG 4): Quality Education**. SDG 4 goal is to ensure inclusive and equitable quality education as well as promote lifelong learning opportunities for all. Our game – **Hang Em High** – promotes quality education through an engaging and interactive learning experience. By transforming a subject like history into a fun game, we aim to capture the interest of students who may find conventional teaching methods boring. This game application is especially beneficial for students from low-income backgrounds who have limited resources as it is free to play, requires no internet connection and can be accessed on any PC, whether public or personally owned, even on low-end machines. We seek to create an environment where learning is not just about memorizing facts but also about exploring and discovering knowledge in a way that connects with young minds. So, by using gamification elements on education, we really hope to boost students' ability to learn and remember historical facts and help them develop a deeper appreciation for Malaysian history.

3.2. Game

Research indicates that digital educational games do improve students' outcomes. Serious games that are designed specifically for educational purposes have demonstrated positive effects on intrinsic motivation. These games provide learners with challenges and feedback that foster an interest in the subject matter itself. In contrast, gamification, which incorporates game-like elements (such as rewards and leaderboards) into non-game environments, tends to impact extrinsic motivation more that encourages students to achieve learning outcomes due to external incentives (Ren, Xu and Liu, 2024). For example, interactive simulations such as **Kerbal Space Program** and **SimCity** allow learners to apply theoretical knowledge in practical scenarios, thereby strengthening problem-solving skills and deepening understanding in complex domains. Behaviorist theory supports the notion that the reward mechanism is inherent in these games, such as points and level progression, increasing students' willingness to engage with educational content by associating learning with achievement and enjoyment (Li, Chen and Deng, 2024).

Kahoot, being one of the most popular digital educational games in classrooms worldwide, is known for engaging quiz-based format. It also combines gamification elements such as points, timers and leaderboards with educational content that makes to appealing and effective to enhance student engagement. Based on the information, we are also able to highlight that our simple digital educational game, **Hang Em High**, leverages both intrinsic and extrinsic motivation. The intrinsic motivation is fostered through knowledge-based challenges in Malaysian history. Players are encouraged to explore and understand historical content to play the game. At the same time, extrinsic motivation is supported by game elements like the leaderboard, point system and difficulty levels (easy, medium and hard). They drive players to achieve high scores.

3.3. Data Structure and Algorithm

3.3.1. List

List can be defined as a data type in which the elements are stored in an ordered manner for easier and efficient retrieval of the elements. In our application, lists are used extensively to manage and store collection of items such as questions, answers options, words to guess, and leaderboard entries. Lists are an efficient way to store and retrieve elements in sequence. In Java, a **List** is a generic interface. The two most popular implementations of the List interface are **ArrayList** and **LinkedList**. Below is a simple comparison table comparing ArrayList and LinkedList in Java.

Feature	ArrayList	LinkedList
Definition	A resizable array of data structure. Also called dynamic array. It stores elements in a sequential order and the size can be increased or decreased by adding or removing elements.	A linear data structure in which elements, call nodes, are stored in sequence where each node contains a data field and a reference (pointer) to the next node in the sequence.
Random Access	Fast, O(1) time complexity.	Slow, O(n) time complexity.
Insertion / Deletion	Shifting elements takes O(n).	Faster at head/tail O(1) but O(n) for indexed elements.
Memory Overhead	Less as it only stores elements.	More due to storing pointers.

Ideal Use Cases	Frequency access by index is needed with fewer insertions/deletions.	Frequent insertions and deletions are required, especially in the beginning or end.

Table 3.1: Comparison of ArrayList and LinkedList

In our project, we prefer to use **ArrayList** due to their efficient random-access capability. We use ArrayList because they are ideal for the scenarios below.

- **Shuffle** questions and answers before displaying them to the user.
- **Search** and **sort** leaderboard entries for displaying top scores.

Since LinkedList do not support direct index-based access, they are considered less efficient for these tasks. To reach the nth element, we must traverse from the head node, which takes O(n) time. The difference makes ArrayList a better choice when random access or sorting is needed.

3.3.2. Queue

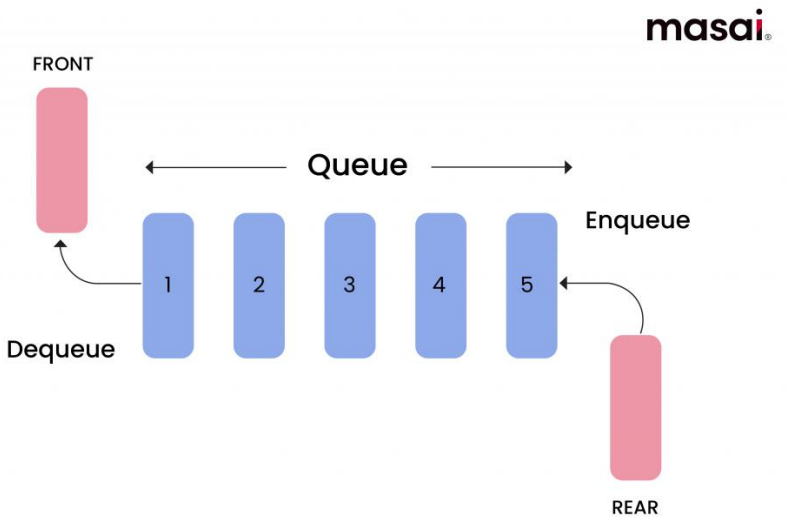


Figure 3.1: Queue Structure Diagram

Queue is a linear data structure that follows the **First In, First Out (FIFO)** principle, meaning that the first element added to the queue will be the first one to be removed. It operates similarly to a real-world queue. You can think of a queue as people standing in line in a supermarket to pay for their groceries. The first person to stand in line is the first one who can pay and then leave the supermarket.

Before we fetch the questions from the backend list to be displayed on the front-end side, we shuffle both the questions and their corresponding answers. Then, we can transform the shuffled list into a queue. The nature of both the question-based and word-guessing challenges requires that questions and words are displayed in a specific order, one after another.

We use a queue for this purpose because the first question added (enqueue) to the queue is the first one to be presented to the player. This ensures a logical progression. After each question is answered, we dequeue (remove) the question from the front. Both enqueue and dequeue operations operate in O(1) complexity. This efficient dequeue operation allows the next question to be available for display, as it moves to the front of the queue. With this approach, we are able to

maintain the order of the questions and also track the number of questions left in the queue by checking its size after each dequeue operation. Using a queue allows us to process the questions sequentially without the need to modify the original list of questions.

On the other hand, a **stack** operates on a **Last In, First Out (LIFO)** basis. This means the most recently added question is the first one to be presented, which is not suitable for our game. Using a stack would disrupt the sequential nature of delivering the question. As a result, players would not be able to receive questions in the order they were intended.

3.3.3. HashMap

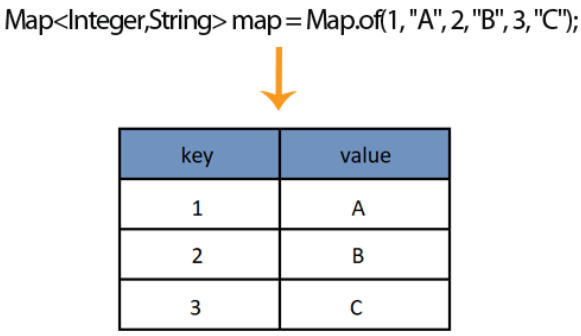


Figure 3.2: HashMap Key-Value Mapping

HashMap is a data structure that is used to store and retrieve values based on unique keys. Each entry in a HashMap consists of a key and a corresponding value. The key is used to retrieve the value from the map. HashMap uses a hash function to compute an index where the key-value pair will be stored. When a key is added to the HashMap, the hash function converts the key into an integer index that corresponds to a specific location in an internal array where the value associated with that key will be stored. This makes HashMap very efficient for quick access and lookups because the value can be retrieved directly from the computed index, rather than having to search through the entire collection. The average complexity for accessing a value by its key is O(1).

First use case is, in the question-based game, we can use a HashMap to store the difficulty levels as keys and their corresponding lists of questions as values. The game features three difficulty

levels: easy, medium and hard. We store all these difficulty levels in a single HashMap. We can then access the list of questions by specifying the key, whether “easy”, “medium” or “hard”. This approach eliminates the need to maintain three separate lists with different names, which can complicate the code and make it harder to manage. Instead, using a single HashMap simplifies data handling and provides quick access to the relevant questions based on the player’s selected difficulty level.

Similarly, word-guessing games can also benefit from a HashMap to manage multiple lists of words categorized by different difficulty levels. Just like the question-based game, we store the difficulty levels as keys and their corresponding lists of words as value.

For every single word, we maintain two HashMap. One HashMap stores the character counts in each word, where each unique character serves as a key and the character’s count as the value. The second HashMap keeps track of how many times each character has been guessed by the player. With this dual use of HashMap, we can verify whether the player has successfully guessed all occurrences of each letter in the word. Once the player’s guesses match the required character counts, we can confirm that the word has been successfully guessed. When the occurrence of each letter has been met by comparing the two HashMap, the game logic ensures that any further attempts to guess that character are marked as incorrect since the player has already exhausted the correct count.

3.3.4. Merge Sort

A sorting algorithm is a method used to rearrange a given array or list of elements in an order. The order can be either ascending or descending based on the requirements. Below are the four common sorting algorithms.

Criteria	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort
Description	Repeatedly swaps adjacent elements if they are in the	Finds the minimum element and swaps it with the first	Builds the sorted array one item at a time, inserting each	Divide and conquer. Divides the array into halves, sorts

	wrong order. Simple but inefficient for large datasets.	element, then the next minimum for the second element and so on.	new element into its proper position. Efficient for small or nearly sorted data.	them, and then merge back together. Uses extra space for merging.
Time Complexity (Best)	O(n)	$O(n^2)$	O(n)	O(n log n)
Time Complexity (Average)	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n log n)
Time Complexity (Worse)	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n log n)
Space Complexity	O(1)	O(1)	O(1)	O(n)
Stability	Yes	No	Yes	Yes
Ease of Implementation	Very simple	Simple	Simple	Complex
Performance for Large Datasets	Poor	Poor	Poor	Good
When to use	Small datasets, nearly sorted data and when simplicity is required.	Small data sets, when memory efficiency is important, simplicity is required, and stability is not the primary concern.	Small datasets, nearly sorted data or when simplicity is required, and memory and space is a concern.	Large datasets, when performance is important, stability is important, and memory overhead is acceptable.

Table 3.2: Comparison of Sorting Algorithms

Leaderboard entries might be large when the game gets popular, and many students sign up to play the educational history game. Therefore, for sorting leaderboard entries, the most suitable algorithm would be **Merge Sort**.

The Merge Sort algorithm provides O(n log n) time complexity, which is much more efficient when comparing to $O(n^2)$ algorithms like Bubble Sort, Selection Sort and Insertion Sort. Besides,

Merge Sort is a stable sorting algorithm, which is important when it comes to sorting scores or rankings. For example, if two players have the same score, Merge Sort ensures that the player who reached the score first will be placed ahead of other, thus preserving the relative order of entries.

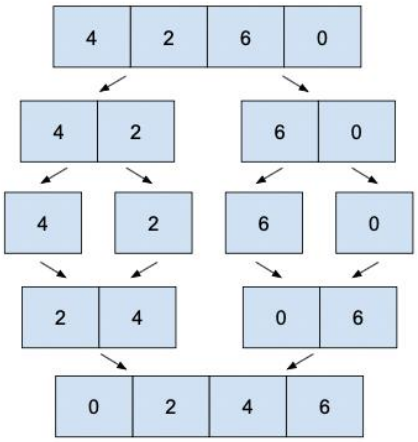


Figure 3.3: Merge Sort Algorithm

The downside of Merge Sort is the O(n) space complexity as it requires extra memory for merging. To merge two sorted sub-arrays into one larger sorted array, a temporary space is required to hold the combined elements before placing them back into the original array. During each merging step, it also needs an additional space to store the intermediate results of combining the elements from the left and right sub-arrays. For leaderboard sorting, we prioritize performance and also scalability. The extra memory usage of O(n) as it is still manageable and acceptable in our case. This concludes that the Merge Sort algorithm is the best choice for sorting our game's leaderboard.

3.3.5. Binary Search

A searching algorithm is a method used to locate specific items within a collection of data like arrays or lists. These algorithms are designed to efficiently navigate through data structures to find the specific elements and information you are looking for. Below are the two most common sorting algorithms.

Linear Search	Binary Search
Check each element in the list one by one.	Repeatedly dividing the sorted list in half. The algorithm starts by checking the middle element and then determines whether to search in the left or right half, and then cutting the search space in half each time.
Input data can be unsorted or sorted	Input data must be in sorted order
Also called sequential search	Also called half-interval search
Time complexity: O(n)	Time complexity: O(log n)
Space complexity: O(1)	Space Complexity: O(1)
Supports multidimensional array	Support only single dimensional array
Simple	More Complex
Slow process	Fast process
Inefficient for large datasets	Efficient for large datasets

Table 3.3: Comparison of Searching Algorithms

Linear search is ideal when we have a smaller number of elements to search through. However, as our game grows in popularity, we might have a large number of leaderboard entries to search through in the possible future. Linear search would become inefficient to search through a large leaderboard as it has O(n) time complexity. **Binary search**, on the other hand, shines and is more suitable for this task when the size of the dataset is huge as it operates with O(log n) time complexity, which is much more efficient.

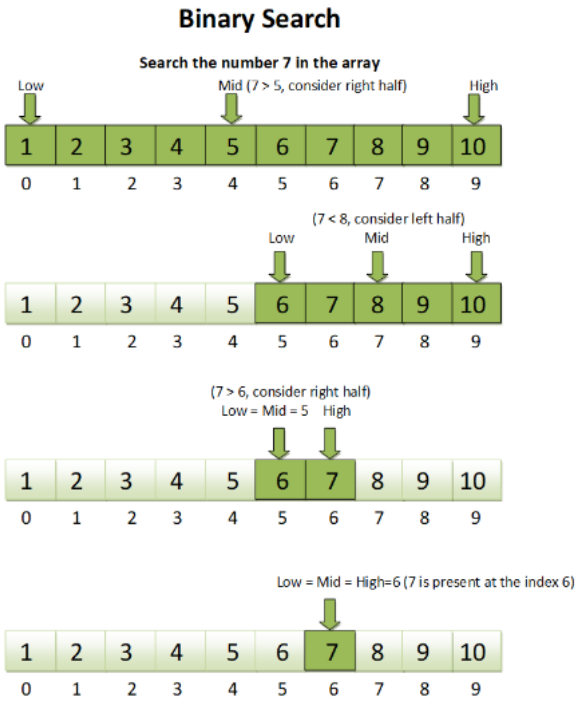


Figure 3.4: Binary Search Algorithm

Our leaderboard entries are stored as a single dimensional list retrieved from the database (binary search supports single dimensional array). Since binary search requires the list to be sorted in advance, we first use our sorting algorithm of choice – **Merge Sort** – to sort the leaderboard entries. After sorting, we can then perform binary search to locate the specific entry within the leaderboard. The combination of Merge Sort (provides fast sorting) and Binary Search (provides fast searching) would provide an optimal and scalable solution for our leaderboard.

4. Methodology

4.1. Technology Stack

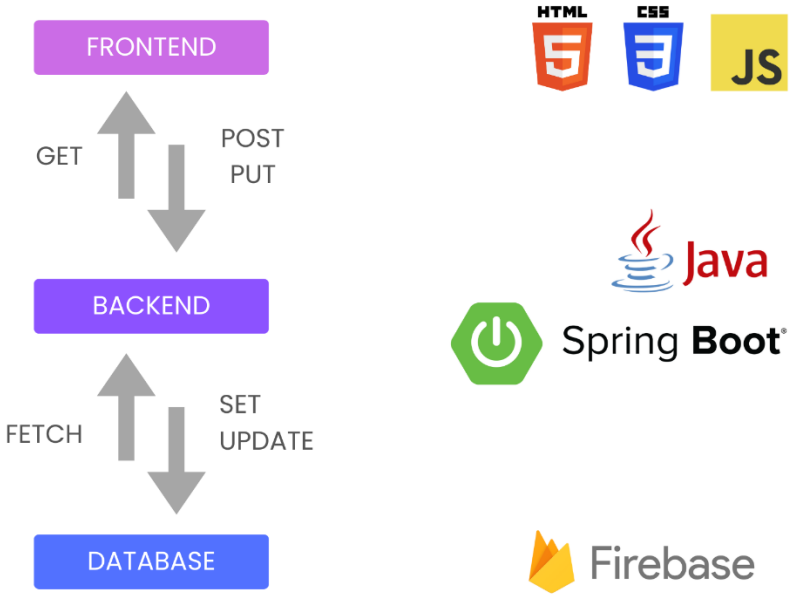


Figure 4.1: Game Technology Stack

Our **Hang Em High** game is developed in **Visual Studio Code** and built using the tech stack:

- Frontend
  - **HTML**, **CSS** and **JavaScript** are used for the user interface (UI). This is the place where the user sees and interacts in our game.
  - The frontend communicates with the backend using HTTP requests (using GET, POST and PUT) to fetch and update data such as questions for question-based games and words for word-guessing games as well as leaderboard entries.
- Backend
  - **Java Spring Boot** is used to implement the backend. This is where all the data structures and algorithms are placed and prepared to be used.

- The backend uses data structures like **ArrayList**, **Queue** and **HashMap** to manage and organize the game data like game questions and leaderboard entries.
- The backend implements **Merge Sort** algorithm to sort leaderboard entries and performs **Binary Search** to look for a specific player entry when requested.
- For example, if users requested to view the leaderboard, the backend would fetch all the leaderboard entries from the database, sort them using Merge Sort, and then respond to the frontend with the sorted list of players' scores to be displayed.

- Database
  - **Firebase** is a BaaS (Backend as a Service) platform provided by Google.
  - We use **Firestore**, which is a cloud-hosted **NoSQL database** offered by Firebase to store our leaderboard data.

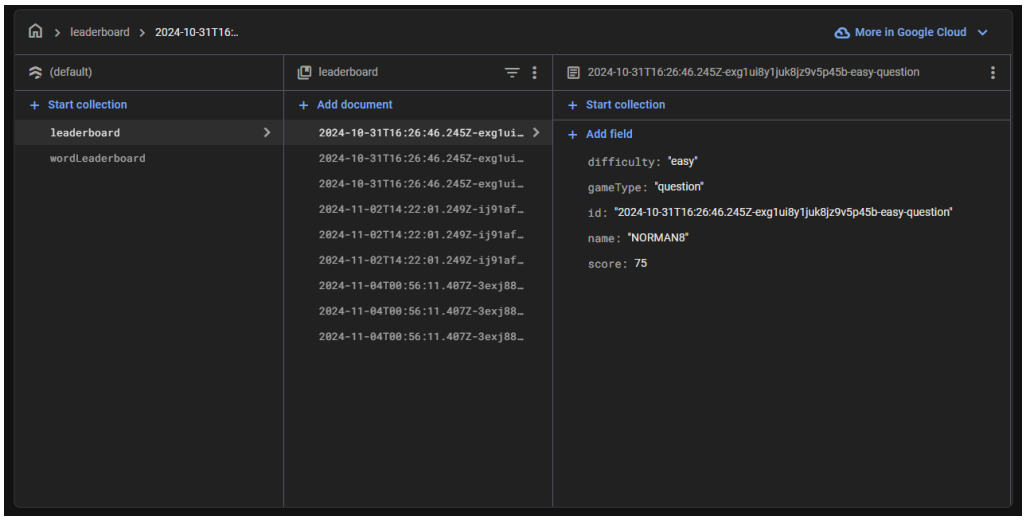


Figure 4.2: Firestore Database Overview

- Each leaderboard entry in the database contains the difficulty, the type of game, a unique identifier for each user, the name of the player, and the score achieved in the game.
- We chose Firebase for our leaderboard database because it provides real-time synchronization, meaning multiple users can compete and interact with the leaderboard with different devices in real time. Firebase is cloud-based. Our team can work from different machines while maintaining access to the same database.

We can all see the leaderboard updates and changes, making the collaborative process so much easier.

4.2. Handling Questions in the Backend

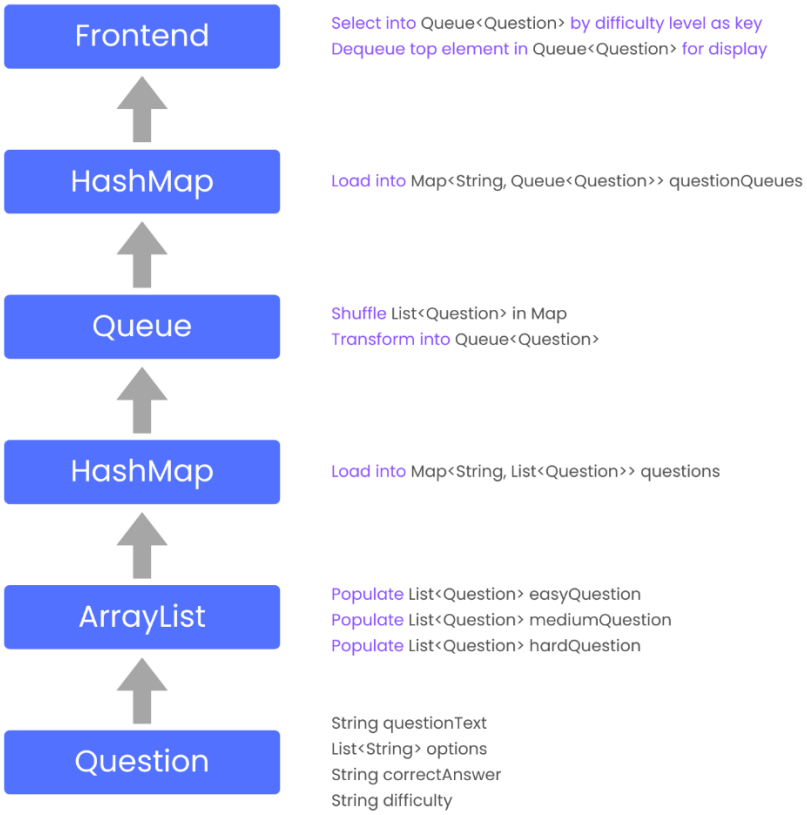


Figure 4.3: Question Processing Workflow

Below are explanations of the process above of transforming questions for question-based game at the backend side to be displayed on the front end in order (bottom-up manner):

1. We define a **Question** model class that contains the fields:
    a. The actual question to be displayed to the user.
    b. A **List** of multiple-choice options for the user to choose from.
    c. The correct answer among the options.

    d. The difficulty level of the question (e.g., easy, medium and hard).
2. Questions are first populated and grouped by difficulty into separate **List**.
    a. The **easyQuestion** list contains all "easy" level questions.
    b. The **mediumQuestion** list contains all "medium" level questions.
    c. The **hardQuestion** list contains all "hard" level questions.
3. We combine the three difficulty question lists into a **HashMap** for easier lookups. The keys are difficulty levels ("easy", "medium", "hard"). The values are lists of questions corresponding to each difficulty level.
4. We shuffle the list of questions within each difficulty in the map to randomize the order of questions and options.
    a. First, we shuffle the **List<Question>** within each difficulty level in the **HashMap**.
    b. Then, for each **Question** in the list, we shuffle the **List<String> options** to ensure the answer positions are randomized.
    c. After shuffling, we transform each **List<Question>** into a **Queue<Question>** to serve questions in a First-In, First-Out (FIFO) manner.
5. The shuffle questions are converted and stored in a new data structure **HashMap Map<String, Queue<Question>> questionQueues**. The keys remain as difficulty levels.
6. When a player selects a difficulty level on the frontend (e.g., easy), the frontend sends a request to the backend with the selected difficulty as a key.
7. The backend retrieves the corresponding **Queue<Question>** from **questionQueues** based on the requested difficulty.
8. Each time the frontend fetches a new question, the backend responds by dequeuing the next question from the **Queue**.
9. The backend continues to dequeue questions until the **Queue** is empty.

4.3. Handling Words in the Backend

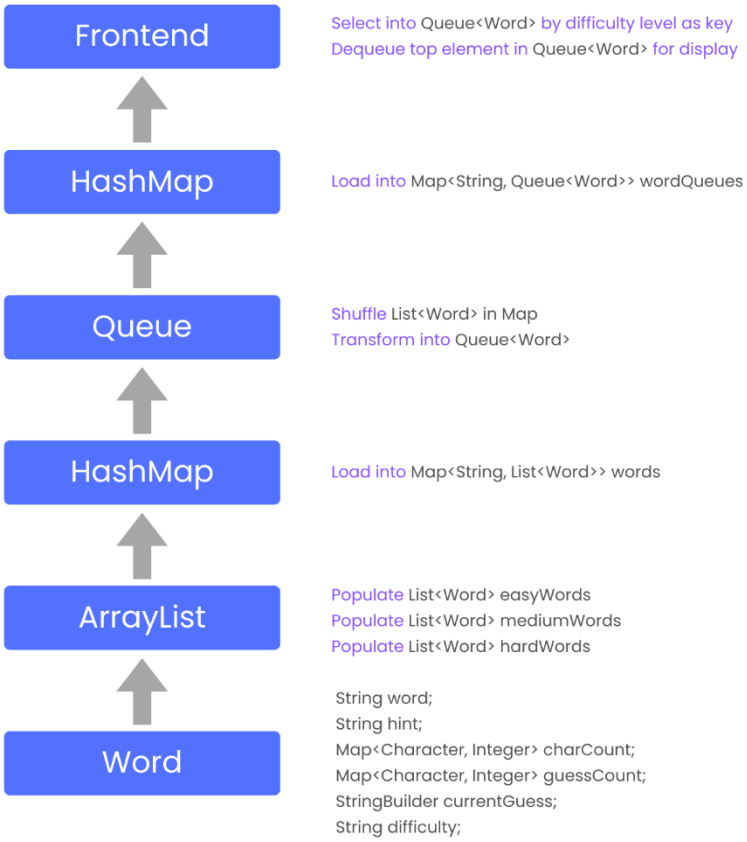


Figure 4.4: Word Processing Workflow

Below are explanations of the process above of transforming words for word-guessing game at the backend side to be displayed on the front end in order (bottom-up manner):

1. Similar to the concept of the question-based game, we define a **Word** model class that contains the fields:
    a. The actual word to be displayed to the user.
    b. A hint to assist player in guessing the word.
    c. **Map<Character, Integer> charCount** counts the occurrence of each character in the word.
    d. **Map<Character, Integer> guessCount** keeps track of the player's guesses, used for comparing against **charCount**.

e. **StringBuilder currentGuess** represents the current state of the player's guessed, displayed as _ _ _ _ to show which letters have been guessed correctly.
f. The difficulty level of the word (e.g., easy, medium and hard).

2. Words are first populated and grouped by difficulty into separate **List**.
    a. The **easyWords** list contains all "easy" level words.
    b. The **mediumWords** list contains all "medium" level words.
    c. The **hardWords** list contains all "hard" level words.
3. We combine the three difficulty word lists into a **HashMap** for easier lookups. The keys are difficulty levels ("easy", "medium", "hard"). The values are lists of words corresponding to each difficulty level.
4. We shuffle the list of words within each difficulty in the map to randomize the order of words.
    a. First, we shuffle the **List<Word>** within each difficulty level in the **HashMap**.
    b. After shuffling, we transform each **List<Word>** into a **Queue<Word>** to serve words in a First-In, First-Out (FIFO) manner.
5. The shuffle words are converted and stored in a new data structure **HashMap Map<String, Queue<Word>> wordQueues**. The keys remain as difficulty levels.
6. When a player selects a difficulty level on the frontend (e.g., easy), the frontend sends a request to the backend with the selected difficulty as a key.
7. The backend retrieves the corresponding **Queue<Word>** from **wordQueues** based on the requested difficulty.
8. Each time the frontend fetches a new word, the backend responds by dequeuing the next word from the **Queue**.
9. The backend continues to dequeue words until the **Queue** is empty.

4.4. Leaderboard Retrieval and Sorting

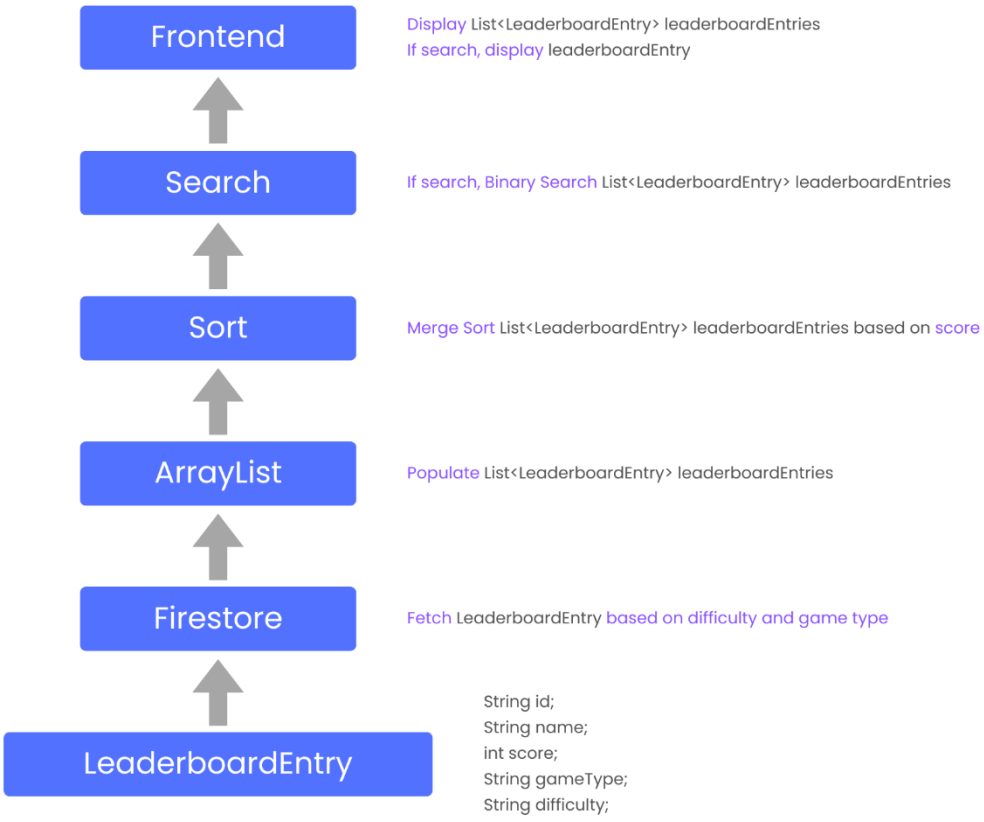


Figure 4.5: Leaderboard Processing Workflow

Below are explanations of the process above of retrieving leaderboard entries to be displayed on the frontend in order (bottom-up manner):

1. We define a **LeaderboardEntry** model class that contains the fields:
    a. A unique identifier for each leaderboard entry.
    b. The name of the player.
    c. The player's score.
    d. The type of game (e.g., question, word)
    e. The difficulty level of the game (e.g., easy, medium and hard).

2. Leaderboard entries are fetched from **Firestore** based on the difficulty level and game type selected by the user on the frontend leaderboard. The backend queries Firestore to retrieve all matching **LeaderboardEntry** documents.
3. The fetched leaderboard entries are populated into a **List**, specifically an **ArrayList** for better performance in sorting and searching operations. The list now contains all the leaderboard entries for a specified difficulty and game type.
4. The entries in the **List** are sorted using **Merge Sort** based on the players' scores in descending order (i.e., highest scores first). By default, it is sorted by score. If needed, for example, players want to search for a specific record by name, the backend will use **Merge Sort by Name** to sort entries **lexicographically** before performing **Binary Search**.
5. If the user performs a search for a specific player, we execute **Binary Search** on the already sorted list.
6. Once sorting or searching is completed, the final list of **LeaderboardEntry** objects is prepared.
    a. If no search was performed, the entire sorted list is sent back to the frontend.
    b. If a specific player search was performed, only the matching entry is returned.
7. The frontend renders the received leaderboard entries to show the players' names and scores in the correct order.

5. Development

5.1. Pseudocode

5.1.1. Main Menu

```
1  START
2      Prompt User to Enter Name
3      Prompt User to Select MenuOption
4      IF MenuOption is "StartGame" THEN
5          StartGame()
6      ELSE IF MenuOption is "Leaderboard" THEN
7          Leaderboard()
8      ELSE
9          About()
10     END IF
11 STOP
```

Figure 5.1: Main Menu Pseudocode

Line	Description
1	Marks the beginning of the game application (main menu).
2	The system prompts the user to input their name.
3	The system presents the user with a menu to choose an option.
4	The actions are "Start Game", "Leaderboard" and "About". • If the user has selected the "Start Game" option, it initiates the START_GAME module. • If the user has selected the "Leaderboard" option, LEADERBOARD module (will be discussed in the leaderboard section). • If the user has selected the "About" option, it initiates the ABOUT module.

Table 5.1: Main Menu Pseudocode Explanation

```
START_GAME MODULE
    Prompt User to Select GameType
    Prompt User to Select Difficulty

    IF GameType is "Question" THEN
        QuestionGame()
    ELSE
        WordGame()
    END IF
STOP
```

Figure 5.2: Start Game Pseudocode

Line	Description
1	This START_GAME module marks the start of the game initiation process.
2	The system asks the user to choose the type of game they want to play. • The options are "Question" for a question-based game and "Word" for a word-guessing game.
3	The system then asks the user to choose the difficulty level for the game. • The level options include "Easy", "Medium" and “Hard".
4	• If the user selects the "Question" game type, it initiates the QUESTION_GAME module. • If the user selects the "Word" game type, it initiates the WORD_GAME module.

Table 5.2: Start Game Pseudocode Explanation

5.1.2. Question-Based Game

```
1  QUESTION_GAME MODULE
2      Initiate Score = 0;
3      Initiate PreviousScore = READ PreviousScore from Database;
4      Initiate Streak = 0;
5      Initiate StreakBonus = 5;
6      Initiate Lives = 6;
7      LOOP
8          Start Timer (20 seconds)
9          FetchQuestion()
10         Display Question
11         Display AnswerOption
12         Prompt user to choose AnswerOption
13
14         IF Timer is 0 THEN
15             Lives = Lives - 1
16             CONTINUE LOOP
17         END IF
18
19         IF RemainingQuestion() is 0 THEN
20             Display "You Passed!"
21             BREAK
22         END IF
23
24         IF AnswerOption is CorrectAnswer THEN
25             Score = Score + 10 + (Streak * StreakBonus)
26             Streak = Streak + 1
27         ELSE
28             Streak = 0
29             Lives = Lives - 1
30         END IF
31
32         IF Lives is 0
33             Display "You Failed!"
34             BREAK
35         END IF
36     END LOOP
37
38     IF Score > PreviousScore THEN
39         WRITE Score to Database
40     END IF
41 STOP
```

Figure 5.3: Question-Based Game Pseudocode

Line	Description
1	QUESTION_GAME module marks the start of beginning of the question-based game.
2	The player's score is set to 0 at the beginning of the game.

3	Fetch the user's previous high score from the database to compare the current score.
4	The streak counter that tracks consecutive correct answers is initialized to zero.
5	The multiplier for the score increase based on the player's streak is set to 5.
6	The player starts with 6 lives.
7	The game continues to run in a loop until the player either passes the game or runs out of lives.
8	A countdown timer of 20 seconds starts for each question. The player must answer before the timer expires.
9	The system initiates the FETCH_QUESTION module and retrieves the next question for the player.
10	The question is displayed to the player.
11	The answer options for the question are presented for the player to choose from.
12	The player is prompted to select an answer.
14	If the timer expires after 20 seconds, the player loses one life. • CONTINUE LOOP skips the rest of the loop and moves on to the next question.
19	The system checks if there are any questions left to answer by calling the REMAINING_QUESTION module. • If there are no more questions left to answer, the player has passed the game. • BREAK indicates the loop ends and the game stops.
24	The system checks if the selected answer matches the correct answer. • If matches the correct answer, the player's score is increased by 10 plus based on their current streak. The longer their streak, the higher the bonus multiplier. • If the answer is wrong, the streak is reset to 0 and the player loses a life.
32	The system checks if the player has run out of lives. • If the player has no lives left, they fail the game. • BREAK indicates that the loop ends and the game stops.
38	Each time the score is updated during the game, we check if the current score exceeds the previous score. • If the current score is greater than the previous score, we write the new score to the database.

Table 5.3: Question-Based Game Pseudocode Explanation

```
FETCH_QUESTION MODULE
    Shuffle the list of questions for each difficulty level
    Convert the shuffled list of questions into a Queue
    Store each difficulty's Queue in a HashMap with difficulty as the key
    GET the difficulty level selected by the user
    Select the Queue corresponding to the selected difficulty from the HashMap
    Dequeue the first question from the selected Queue
    Return the dequeued question
STOP
```

Figure 5.4: Fetch Question from Backend Pseudocode

Line	Description
1	This FETCH_QUESTION module represents the backend process that prepares and sends questions to the frontend.
2	The list of questions for each difficulty level is shuffled randomly to ensure that questions are presented in a random order.
3	Once shuffled, the list of questions is converted into a queue to be processed in a first-in, first-out (FIFO) order.
4	The queues for each difficulty level are stored in a HashMap, where the key is the difficulty level (e.g., easy, medium and hard) and the value is the corresponding queue of questions.
5	The player's selected difficulty level (provided by the frontend) is retrieved.
6	The appropriate queue of questions is retrieved from the HashMap based on the selected difficulty as a key.
7	The first question is dequeued from the queue representing the next question to be asked.
8	The question that has been dequeued is returned to be displayed to the player.

Table 5.4: Fetch Question from Backend Pseudocode Explanation

```
1  REMAINING_QUESTION MODULE
2      GET the difficulty level selected by the user
3      SELECT Queue from HashMap using the difficulty level as the key
4
5      IF Queue is Empty THEN
6          RETURN 0
7      ELSE
8          RETURN Queue size
9      END IF
10 STOP
```

Figure 5.5: Check Remaining Question Pseudocode

Line	Description
1	This REMAINING_QUESTION module is responsible for checking how many questions are left in the selected difficulty level's queue.
2	Fetch the difficulty level that the user is currently playing (e.g., easy, medium and hard).
3	Retrieves the corresponding question queue from a HashMap using the selected difficulty level.
5	Check if the queue is empty. • If the queue is empty (means no questions left), it returns 0 to indicate that there are no more questions for that difficulty level. • If the queue contains questions, it returns the number of questions left in the queue (size of the queue).

Table 5.5: Check Remaining Question Pseudocode Explanation

5.1.3. Word-Guessing Game

```
1  WORD_GAME MODULE
2      Initiate Score = 0;
3      Initiate PreviousScore = READ PreviousScore from Database;
4      Initiate Lives = 6;
5      LOOP
6          Start Timer (count up from 0)
7          FetchWord()
8          Display Word
9          Display A-Z Keyboard Buttons
10
11         UserInput = Listen for Key Press (A-Z)
12
13         FOR Each Letter in Word
14             WHILE Letter is Not Correctly Guessed
15                 UserInput = Listen for Key Press (A-Z)
16
17                 Call CHECK_LETTER(UserInput, CurrentLetter)
18
19                 IF CHECK_LETTER returns True THEN
20                     Reveal Correct Letter in Display
21                 ELSE
22                     Lives = Lives - 1
23                     IF Lives is 0 THEN
24                         Display "You Failed!"
25                         TimeTaken = Get Timer Value (in seconds)
26                         BREAK OUT OF BOTH LOOPS
27                     END IF
28                 END IF
29             END WHILE
30         END FOR
31
32         Score = Score + 20
33
34         IF RemainingWord() is 0 THEN
35             Display "You Passed!"
36             TimeTaken = Get Timer Value (in seconds)
37             BREAK
38         END IF
39     END LOOP
40
41     WeightedScore = CalculateWeightedScore(Score, TimeTaken)
42
43     IF WeightedScore  > PreviousScore THEN
44         WRITE WeightedScore to Database
45     END IF
46 STOP
```

Figure 5.6: Word-Guessing Game Pseudocode

Line	Description

1	The WORD_GAME module marks the start of the beginning of the word-guessing game.
2	The player’s score is set to 0 at the beginning of the game.
3	The system fetches the player 's previous high score from the database to compare the current score.
4	The player starts with 6 lives.
5	The game runs in a loop until the player either successfully guesses all the words or runs out of lives.
6	A count-up timer begins from 0 and tracks the time taken by the player to finish the game.
7	The system initiates the FETCH_WORD module to retrieve the next word for the player to guess.
8	The fetched word is displayed on the screen with hidden letters (e.g., "_ _ _"), indicating how many letters the player needs to guess.
9	A set of A-Z keyboard buttons are rendered on the screen, allowing the player to select letters as their guesses.
11	The game listens for the player to press a key (A-Z) as their input.
15	For each letter in the word, the player must guess the correct letter by pressing a key (A-Z).
17	The system calls the CHECK_LETTER module to check if the UserInput matches any letter in the current word. • If the guessed letter is correct, it is revealed in the display. • If the guessed letter is incorrect, the player loses one life. • If lives reach 0, the loop ends, and the game stops.
32	The player is awarded 20 points for each correctly guessed word.
34	The system checks if there are any words left to guess by calling the REMAINING_WORD module. • If there are no words left, the player has passed the game. • BREAK indicates that the loop ends, and the game stops.
41	After the game ends, the system calculates a weighted score based on the player's total score and the time taken.
43	If this new weighted score is higher than the previous high score, it is updated in the database.

Table 5.6: Word-Guessing Game Pseudocode Explanation

```
FETCH_WORD MODULE
    Shuffle the list of words for each difficulty level
    Convert the shuffled list of words into a Queue
    Store each difficulty's Queue in a HashMap with difficulty as key
    GET the difficulty level selected by user
    Select the Queue corresponding to the selected difficulty from HashMap
    Dequeue the first word from the selected Queue
    Return the dequeued word
STOP
```

Figure 5.7: Fetch Word from Backend Pseudocode

Line	Description
1	This FETCH_WORD module represents the backend process that prepares and sends words to the frontend.
2	The list of words for each difficulty level is shuffled randomly to ensure that words are presented in a random order.
3	Once shuffled, the list of words is converted into a queue to be processed in a first-in, first-out (FIFO) order.
4	The queues for each difficulty level are stored in a HashMap, where the key is the difficulty level (e.g., easy, medium and hard) and the value is the corresponding queue of words.
5	The player's selected difficulty level (provided by the frontend) is retrieved.
6	The appropriate queue of words is retrieved from the HashMap based on the selected difficulty as a key.
7	The first word is dequeued from the queue representing the next word to be asked.
8	The word that has been dequeued is returned to be displayed to the player.

Table 5.7: Fetch Word from Backend Pseudocode Explanation

```
REMAINING_WORD MODULE
    GET the difficulty level selected by the user
    SELECT Queue from HashMap using the difficulty level as the key

    IF Queue is Empty THEN
        RETURN 0
    ELSE
        RETURN Queue size
    END IF
STOP
```

Figure 5.8: Check Remaining Word Pseudocode

Line	Description
1	This REMAINING_WORD module is responsible for checking how many words are left in the selected difficulty level's queue.
2	Fetch the difficulty level that the user is currently playing (e.g., easy, medium and hard).
3	Retrieves the corresponding word queue from a HashMap using the selected difficulty level.
5	Check if the queue is empty. • If the queue is empty (means no words left), it returns 0 to indicate that there are no words for that difficulty level. • If the queue contains words, it returns the number of words left in the queue (size of the queue).

Table 5.8: Check Remaining Word Pseudocode Explanation

```
1  CHECK_LETTER MODULE
2      Initiate charCount  as a HashMap to store word letter counts
3      Initiate GuessCount as a HashMap to track correct guesses
4
5      GET Letter from player input
6
7      IF Letter exists in charCount THEN
8          GET maxOccurrences from charCount for Letter
9          GET currentOccurrences from guessCount for Letter
10
11         IF currentOccurrences < maxOccurrences THEN
12             Increment guessCount for Letter by 1
13             RETURN true  // Letter guessed correctly
14         ELSE
15             RETURN false // Letter already guessed enough times
16     END IF
17     ELSE
18         RETURN false // Letter not found in the word
19     END IF
20 STOP
```

Figure 5.9: Check Guessed Letter if Correct Pseudocode

Line	Description
1	The CHECK_LETTER module is responsible for checking if a player's guessed letter is correct.

2	This module has two HashMap: • charCount stores the total occurrences of each letter in the target word. • guessCount tracks how many times the player has correctly guessed each letter.
5	Retrieves the guessed letter from the player's input.
7	Checks if the guessed letter exists in the charCount HashMap. If the letter exists, • Fetches total times the letter appears in the word from charCount. • Fetches total times the player has guessed this letter from guessCount.
11	If the total times the player has guessed this letter is less than the total times the letter appears in the word, increments the count in guessCount and returns true to indicate a correct guess.
14	If the total times the player has guessed this letter matches the total times the letter appears in the word, returns false to indicate the letter has been guessed enough times.
17	If the guessed letter does not exist in charCount, it returns false to indicate an incorrect guess.

Table 5.9: Check Guessed Letter if Correct Pseudocode Explanation

5.1.4. Leaderboard

```
1  LEADERBOARD MODULE
2      If Search is true
3          FetchEntry()
4      ELSE
5          FetchEntries()
6      END IF
7  STOP
```

Figure 5.10: Leaderboard Pseudocode

Line	Description
1	The LEADERBOARD module is responsible for managing the leaderboard operations.
2	If user wants to find a specific entry, it calls the FETCH_ENTRY module that is responsible for fetching as single leaderboard entry based on the provided name.

4	If a user is not performing a search operation, it calls the FETCH_ENTRY module, which retrieves all leaderboard entries.

Table 5.10: Leaderboard Pseudocode Explanation

```
1  FETCH_ENTRIES MODULE
2      FETCH all leaderboard entries from Firestore
3      CALL MERGE_SORT(entries)
4  STOP
```

Figure 5.11: Fetch Leaderboard Entries from Backend Pseudocode

Line	Description
1	This FETCH_ENTRIES module is responsible for fetching all leaderboard entries from Firestore.
3	After fetching the leaderboard entries, the module uses MERGE_SORT algorithm on the retrieved entries to sort them by score in descending order.

Table 5.11: Fetch Leaderboard Entries from Backend Pseudocode Explanation

```
1  MERGE_SORT(entries)
2      IF length of entries is 1 THEN
3          RETURN entries
4      END IF
5
6      mid = length of entries / 2
7      left = MERGE_SORT(entries[0 to mid])
8      right = MERGE_SORT(entries[mid to end])
9
10     MERGED_LIST = MERGE(left, right)
11
12     RETURN MERGED_LIST
13 STOP
14
15 MERGE(left, right)
16     Initialize mergedList as empty list
17
18     WHILE left is not empty AND right is not empty
19         IF left[0].score > right[0].score THEN
20             append left[0] to mergedList
21             remove left[0]
22         ELSE
23             append right[0] to mergedList
24             remove right[0]
25         END IF
26     END WHILE
27
28     append remaining entries from left to mergedList
29     append remaining entries from right to mergedList
30
31     RETURN mergedList
32 STOP
```

Figure 5.12: Merge Sort Algorithm Pseudocode

Line	Description
1	The MERGE-SORT algorithm is responsible for sorting a list of leaderboard entries based on their scores.
2	If the list only has one entry, it is already sorted, so it returns the list as is.
6	If the list has more than one entry, the algorithm divides the list into two halves.
7	Calculates the midpoint of the list and then recursively calls the MERGE_SORT algorithm on the left and right halves of the list.

10	Once the left and right halves have been sorted through recursive calls, the algorithm merges them back together using the MERGE function.
12	Returns the merged and sorted list.
15	The MERGE function is responsible for merging two sorted lists into one sorted list based on the score in descending order.
16	Initialize a new empty list to hold the combined results.
18	While both the left and right lists are not empty, the algorithm compares the first element of both lists.
19	If the score of the first element in the left list is greater than the score of the first element in the right list, append the element from the left list to the new list and remove it from the left list.
22	If the score of the first element in the right list is greater, it appends the element from the right list to the new list and removes it from the right list.
28	After one of the lists is empty, the remaining entries from the other list are appended to the new list so that all entries are included in the final sorted list.
31	Returns the new list, which now contains all the entries in descending order by score.

Table 5.12: Merge Sort Algorithm Pseudocode Explanation

```
1  FETCH_ENTRY MODULE
2      GET name to search from user
3      FETCH leaderboard entry for name from Firestore
4
5      CALL MERGE_SORT(entries) based on lexicographical order of names
6      CALL BINARY_SEARCH(entries, name)
7  STOP
```

Figure 5.13: Fetch Specific Leaderboard Entry Pseudocode

Line	Description
1	This FETCH_ENTRY module is responsible for retrieving a specific leaderboard entry by name.
2	The module prompts the user for the name they want to search in the leaderboard.
3	The module then fetches all leaderboard entries from Firestore.

5	The fetched entries are sorted by name in lexicographical order (alphabetical order) using the MERGE_SORT algorithm.
6	After sorting, BINARY_SEARCH function is used to find the specific leaderboard entry by the given name.

Table 5.13: Fetch Specific Leaderboard Entry Pseudocode Explanation

```
1  BINARY_SEARCH(entries, name)
2      INIT left = 0
3      INIT right = length of entries - 1
4
5      WHILE left <= right
6          mid = (left + right) / 2
7          IF entries[mid].name == name THEN
8              RETURN entries[mid]
9          ELSE IF entries[mid].name < name THEN
10             left = mid + 1
11         ELSE
12             right = mid - 1
13         END IF
14     END WHILE
15
16     RETURN -1
17 STOP
```

Figure 5.14: Binary Search Algorithm Pseudocode

Line	Description
1	The BINARY_SEARCH function searches for a specific leaderboard entry by name in a sorted list.
2	The left pointer is set to the beginning index of the list (0) and the right pointer is set to the end index of the list (length of the entries – 1).
5	The search is conducted within a loop that continues until the left pointer exceeds the right pointer.
6	The middle index is calculated by averaging the left and right pointers.
7	Compare the name at the middle index with the searched name. • If the names match, it returns the leaderboard entry at the middle index. • If the name at middle index of the list is lexicographically smaller than the searched name, the left pointer is updated to middle index + 1 to search the right half of the list.

	• If the name at middle index is lexicographically larger, the right pointer is updated to middle index – 1 to search the left half of the list.
16	If the name is not found when the left pointer exceeds the right pointer, the function returns -1 to indicate that the name does not exist in the list.

Table 5.14: Binary Search Algorithm Pseudocode Explanation

5.2. Screen Output

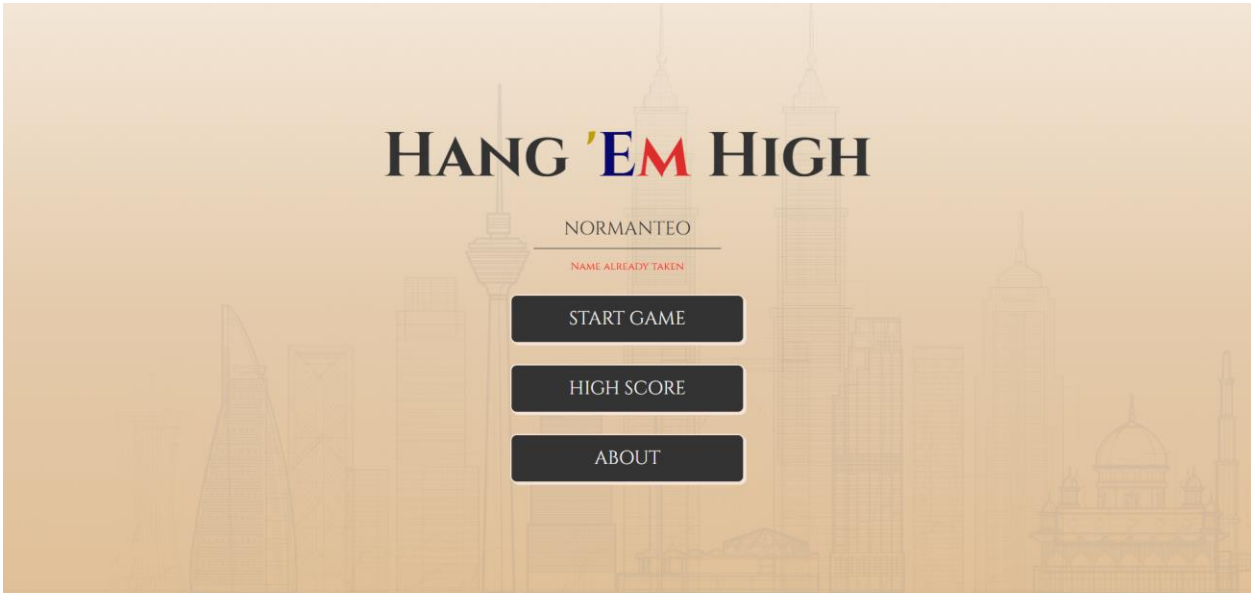


Figure 5.15: Main Menu Page

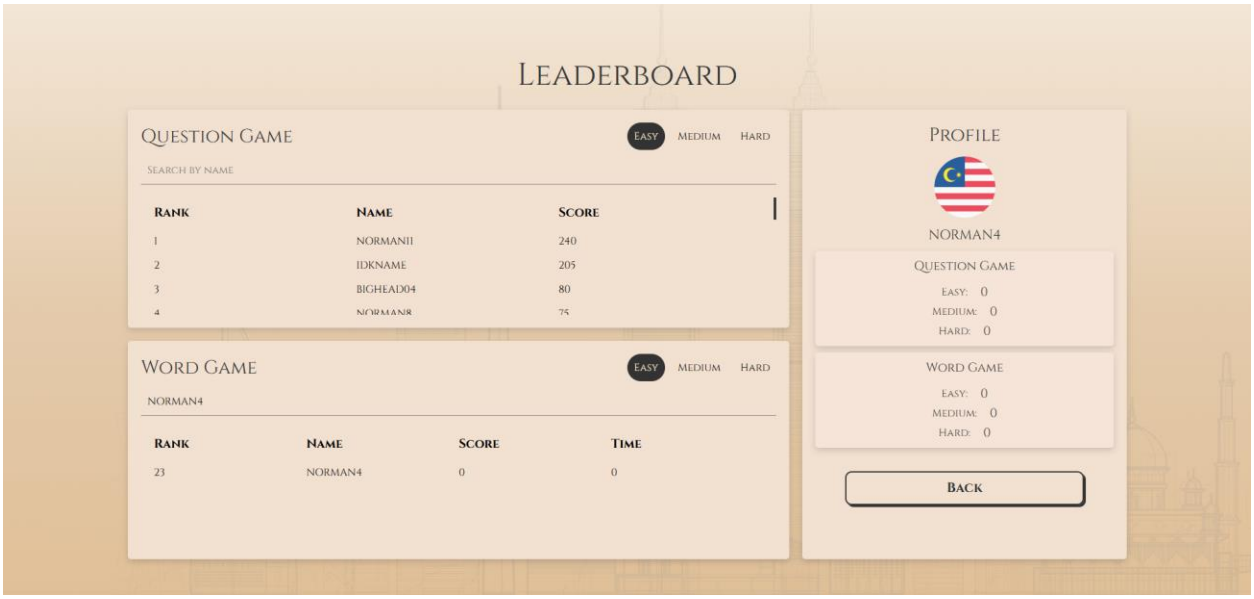


Figure 5.16: Leaderboard Page

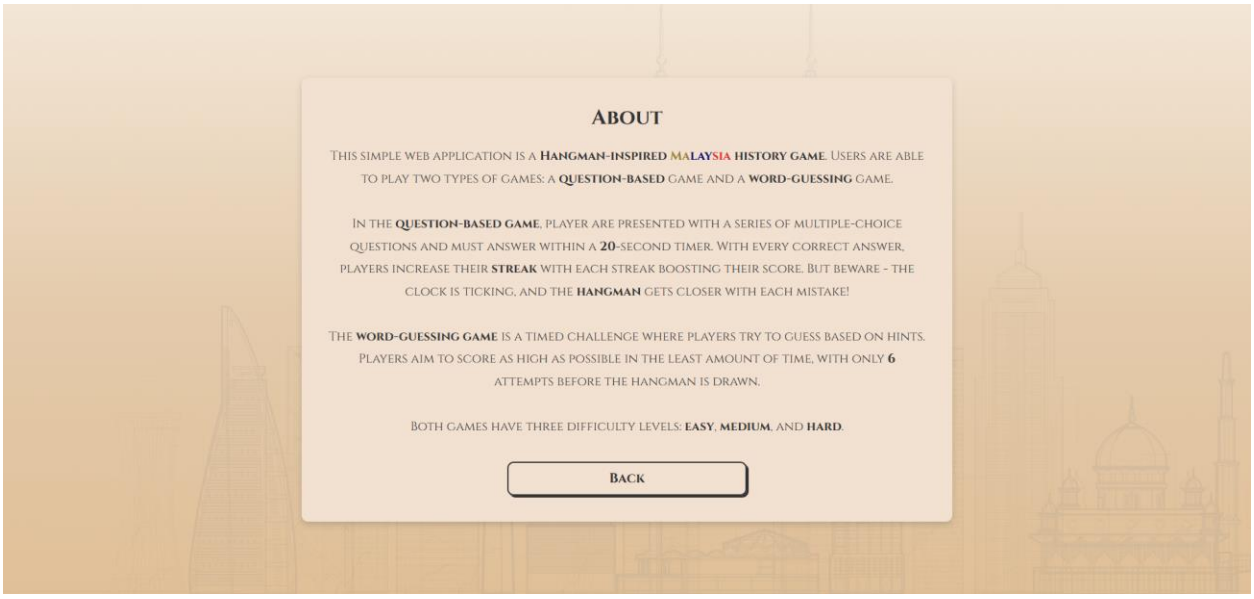


Figure 5.17: About Page

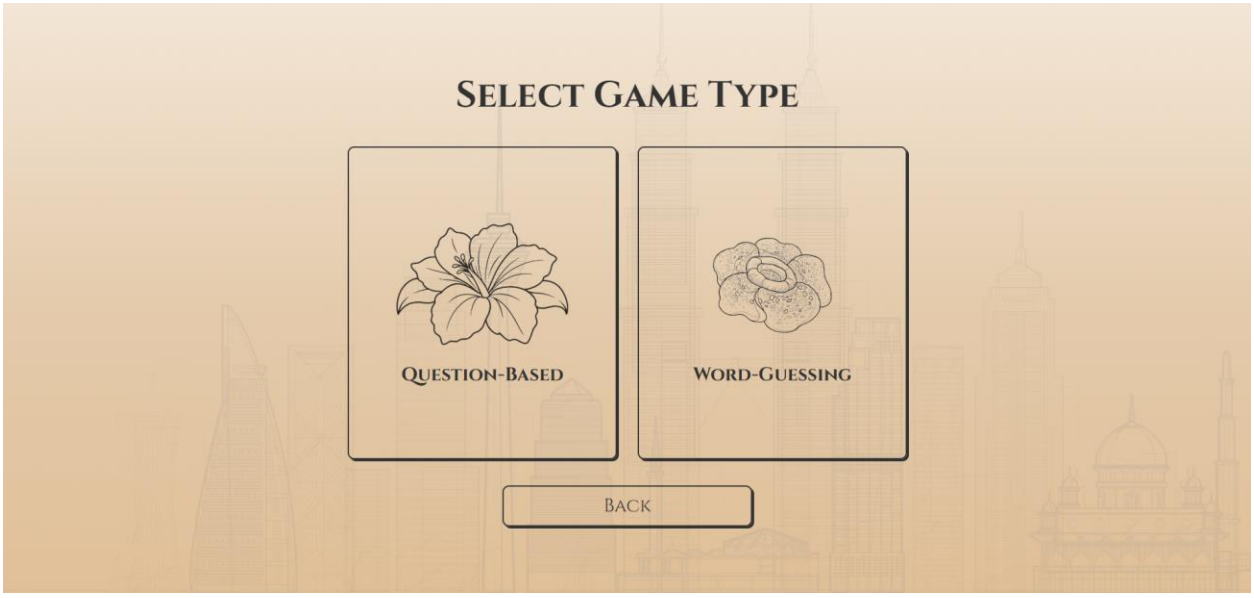


Figure 5.18: Game Selection Page

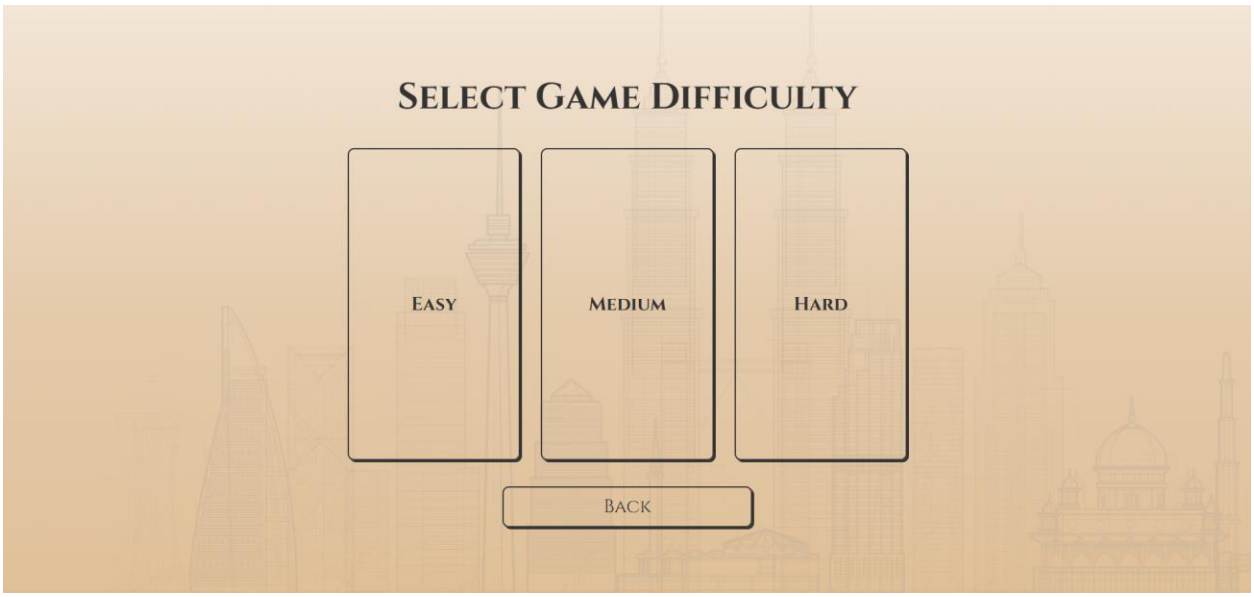


Figure 5.19: Game Difficulty Selection Page

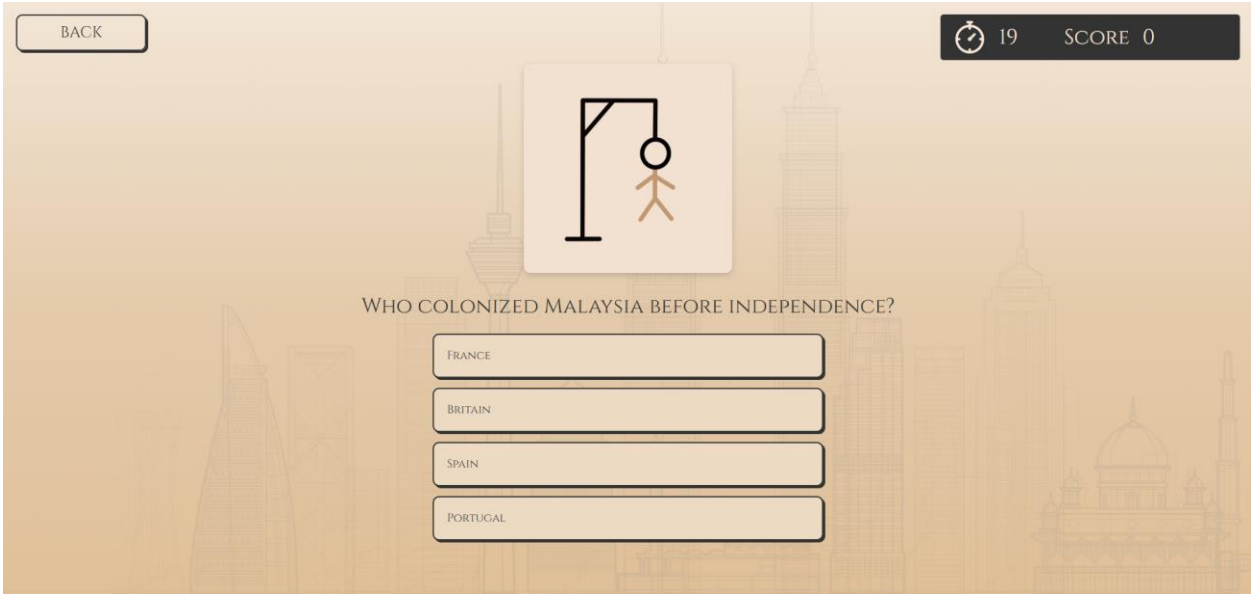


Figure 5.20: Question Game Page

Figure 5.21: Question Game – Game Passed

Figure 5.22: Question Game – Game Over

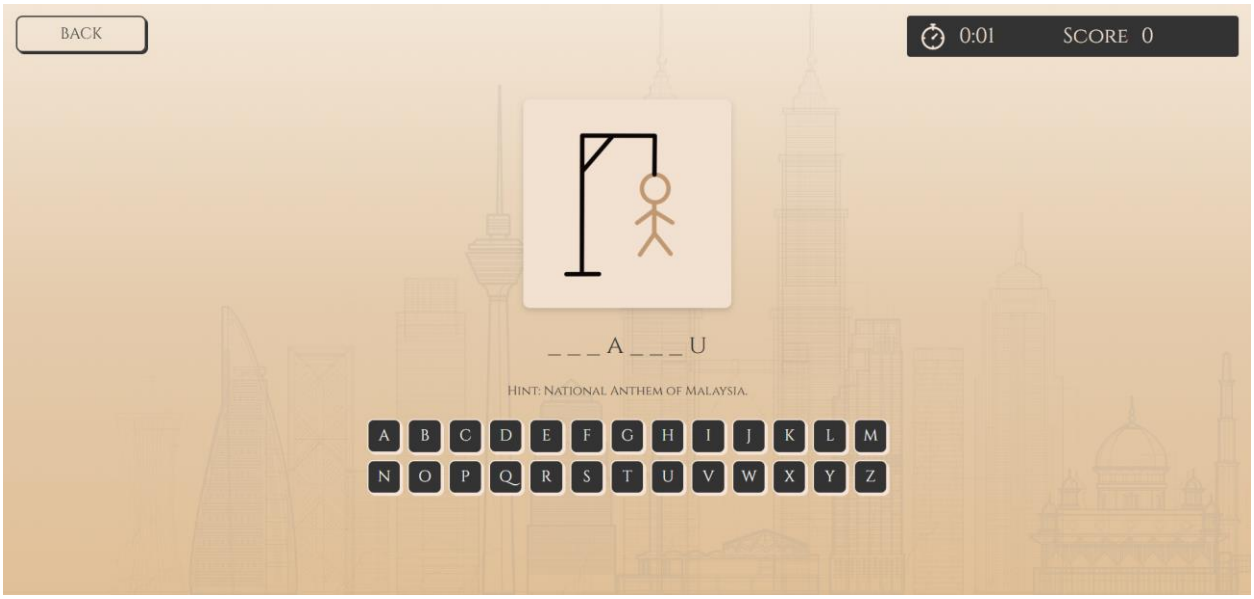


Figure 5.23: Word Guessing Game Page

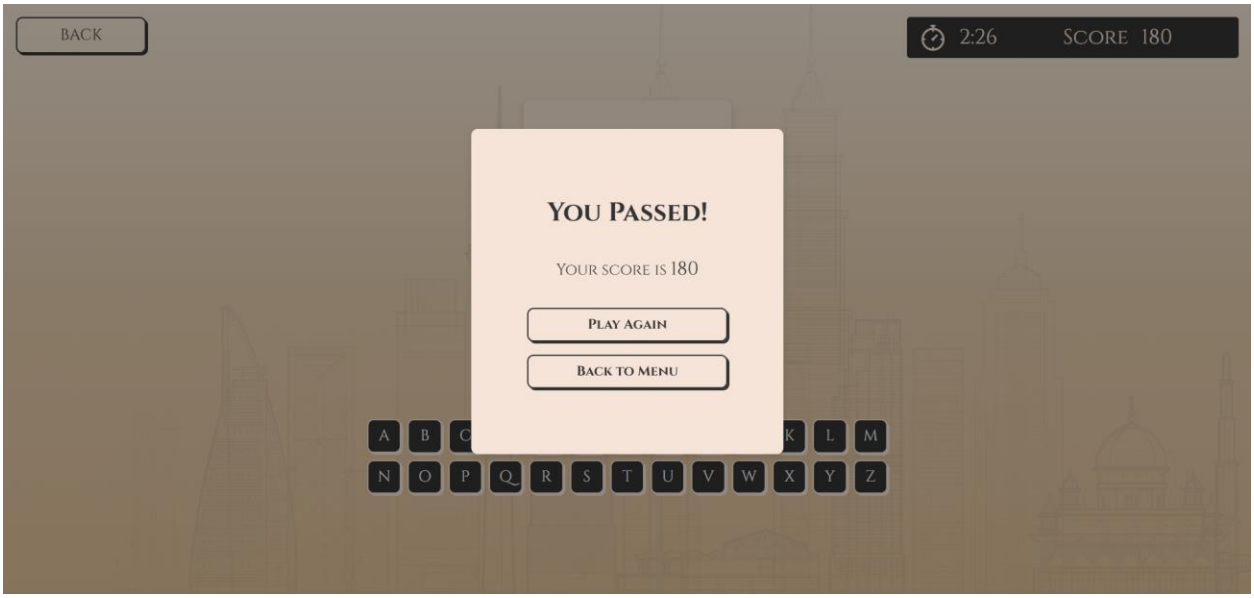


Figure 5.24: Word Guessing – Game Passed

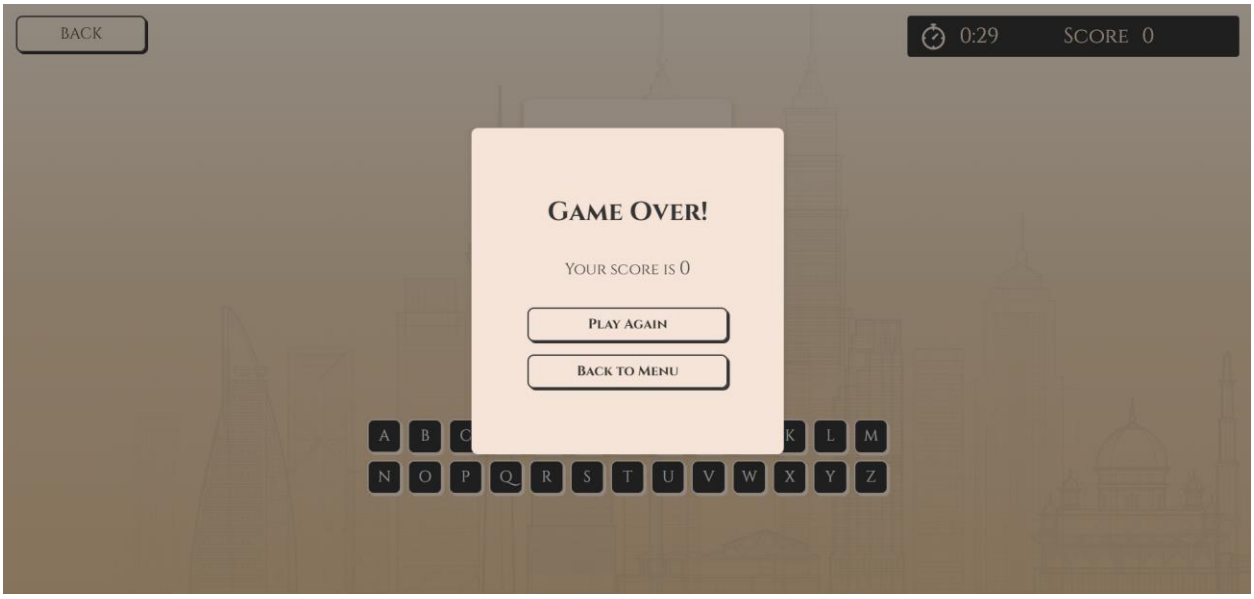


Figure 5.25: Word Guessing – Game Over

5.3. Algorithm Implementation and Timing

Specification of System (The timing of the algorithm can vary depending on the system specifications)	
CPU	12th Gen Intel(R) Core(TM) i5-12400F 2.50 GHz
RAM	16.0 GB (15.9 GB usable)
OS	Windows 11 Home (64-bit operating system)
Storage	2TB HDD 7200 RPM (Project File Location)

Table 5.15: System Specifications for Timing Algorithms

Current State of System (The timing of the algorithm can vary on the current system state and running processes)	
Software In Use	Visual Studio Code, Microsoft Word, Google Chrome (2 tab opened)
Background Tasks	Possible system services and other applications running on the background.
Data in System	We have an initial size of 4 player records with different scores and names for timing the algorithms. N = 4.

Table 5.16: Current State of System for Timing Algorithms

5.3.1. Merge Sort (Sort by Score)

```java
// Merge sort implementation - Score based
private void mergeSort(List<LeaderboardEntry> leaderboard) {
  if (leaderboard.size() <= 1) {
    return;
  }

  int mid = leaderboard.size() / 2;

  List<LeaderboardEntry> left = new ArrayList<>(leaderboard.subList(fromIndex:0, mid));
  List<LeaderboardEntry> right = new ArrayList<>(leaderboard.subList(mid, leaderboard.size()));

  mergeSort(left);
  mergeSort(right);

  merge(leaderboard, left, right);
}
```

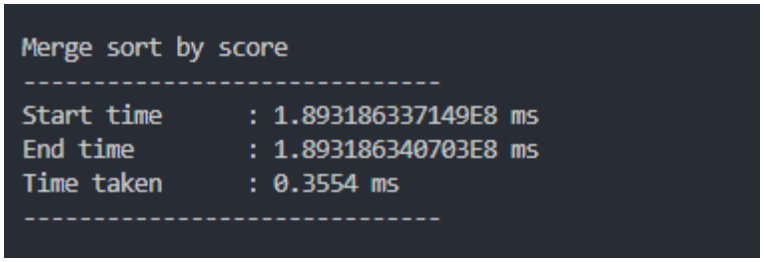
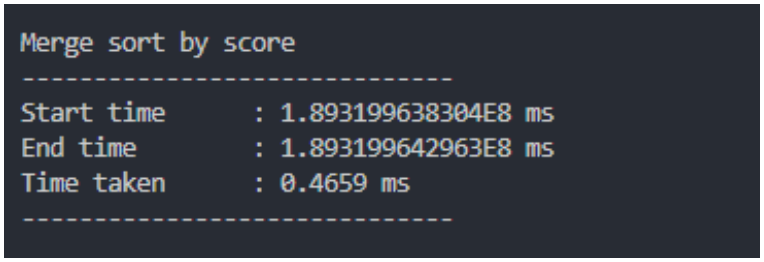
```java
private void merge(
    List<LeaderboardEntry> leaderboard,
    List<LeaderboardEntry> left,
    List<LeaderboardEntry> right) {
  int leftIndex = 0;
  int rightIndex = 0;
  int leaderboardIndex = 0;

  while (leftIndex < left.size() && rightIndex < right.size()) {
    if (left.get(leftIndex).getScore() > right.get(rightIndex).getScore()) {
      leaderboard.set(leaderboardIndex++, left.get(leftIndex++));
    } else {
      leaderboard.set(leaderboardIndex++, right.get(rightIndex++));
    }
  }

  // Copy remaining elements from left
  while (leftIndex < left.size()) {
    leaderboard.set(leaderboardIndex++, left.get(leftIndex++));
  }

  // Copy remaining elements from right
  while (rightIndex < right.size()) {
    leaderboard.set(leaderboardIndex++, right.get(rightIndex++));
  }
}
```

Figure 5.26: Merge Sort by Score Implementation

The merge sort algorithm above is used to sort the leaderboard entries based on the player scores. The method recursively splits the list of entries into two halves, sorts each half, and then merges back together in descending order of score.

```
System.gc();
double startTime = System.nanoTime();
System.out.println(x:"\nMerge sort by score");
System.out.println(x:"------------------------------");
System.out.println("Start time\t: " + (startTime / 1_000_000) + " ms");

mergeSort(leaderboard);

double endTime = System.nanoTime();
System.out.println("End time\t: " + (endTime / 1_000_000) + " ms");

double timeDiff = endTime - startTime;
System.out.println("Time taken\t: " + (timeDiff / 1_000_000) + " ms");
System.out.println(x:"------------------------------\n");

return leaderboard;
```

Figure 5.27: Merge Sort by Score Timing Code

Size Data	N = 4
Attempt 1	 0.3554 milliseconds
Attempt 2	 0.4659 milliseconds

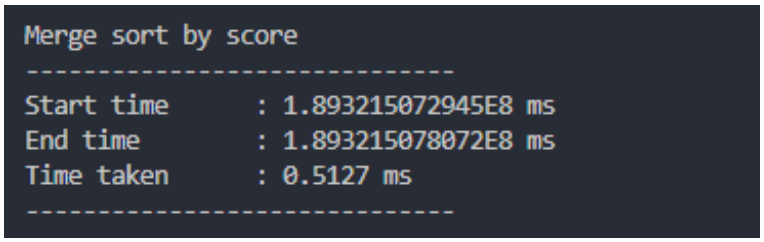
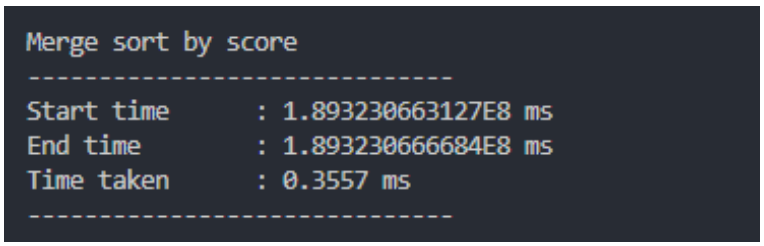
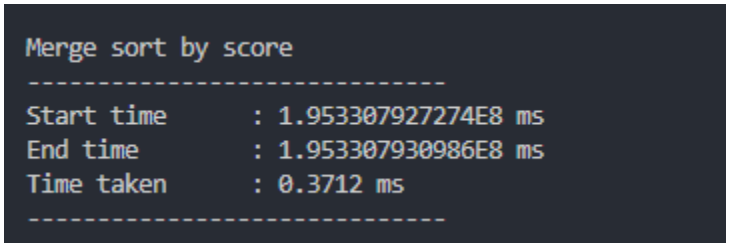
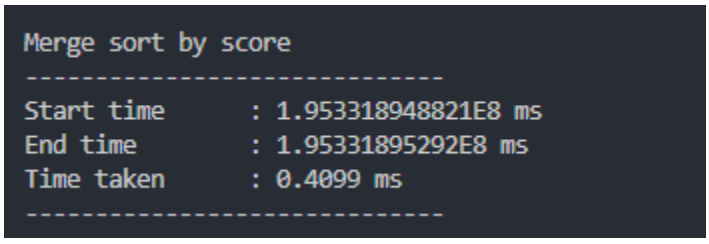
Attempt 3	 0.5127 milliseconds
Attempt 4	 0.3557 milliseconds
Average	(0.3554 + 0.4659 + 0.5127 + 0.3557) / 4 = 0.4224 milliseconds

Table 5.17: Merge Sort by Score Timing (N = 4)

Size Data	4N = 16
Attempt 1	 0.3712 milliseconds
Attempt 2	 0.4099 milliseconds

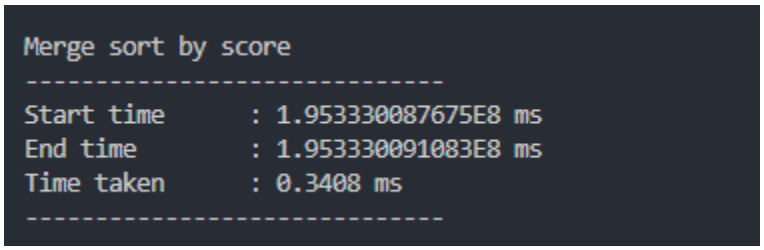
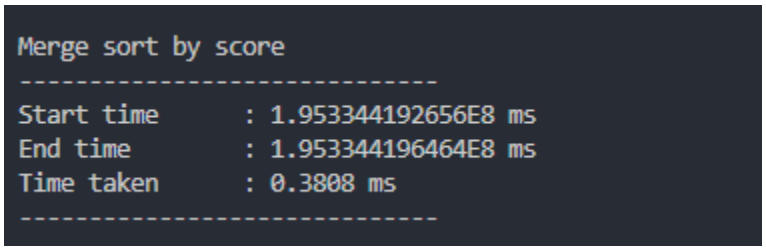
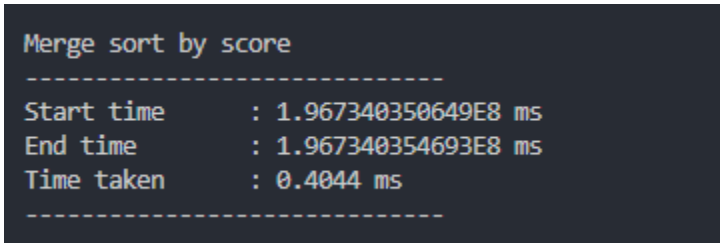
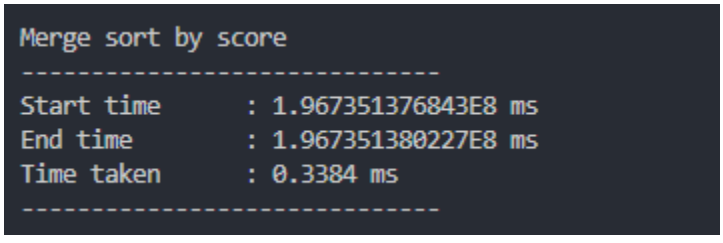
Attempt 3	 0.3408 milliseconds
Attempt 4	 0.3808 milliseconds
Average	(0.3712 + 0.4099 + 0.3408 + 0.3808) / 4 = 0.3757 milliseconds

Table 5.18: Merge Sort by Score Timing (4N = 16)

Size Data	6N = 24
Attempt 1	 0.4044 milliseconds
Attempt 2	 0.3384 milliseconds

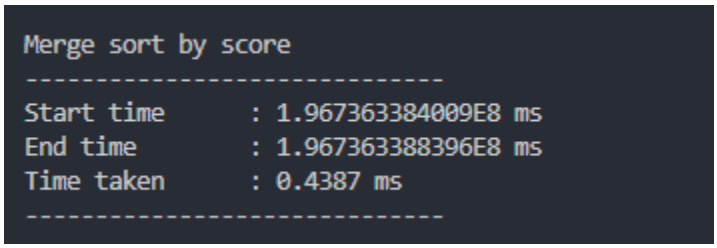
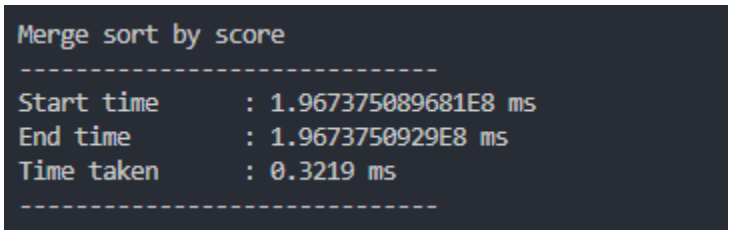
Attempt 3	 0.4387 milliseconds
Attempt 4	 0.3219 milliseconds
Average	(0.4044 + 0.3384 + 0.4387 + 0.3219) / 4 = 0.3759 milliseconds

Table 5.19: Merge Sort by Score Timing (6N = 24)

5.3.2. Merge Sort (Sort by Name)

```
private void mergeSortName(List<LeaderboardEntry> leaderboard) {
  if (leaderboard.size() <= 1) {
    return;
  }

  int mid = leaderboard.size() / 2;

  List<LeaderboardEntry> left = new ArrayList<>(leaderboard.subList(fromIndex:0, mid));
  List<LeaderboardEntry> right = new ArrayList<>(leaderboard.subList(mid, leaderboard.size()));

  mergeSortName(left);
  mergeSortName(right);

  mergeName(leaderboard, left, right);
}
```

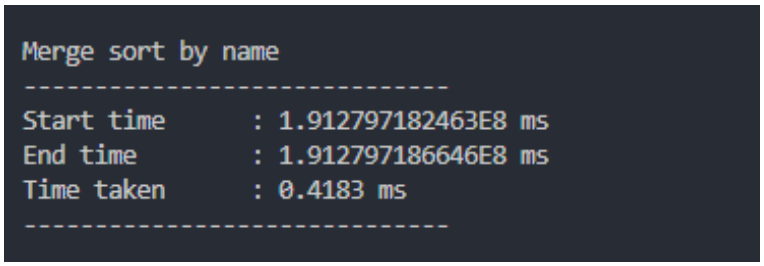
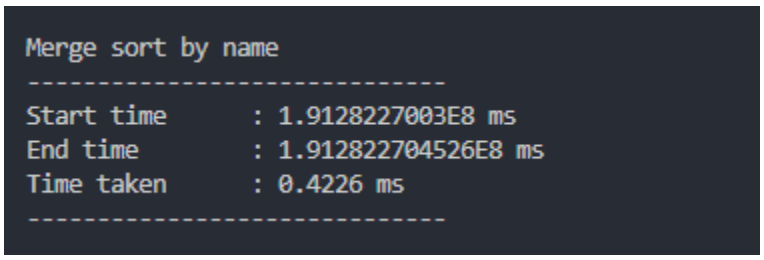
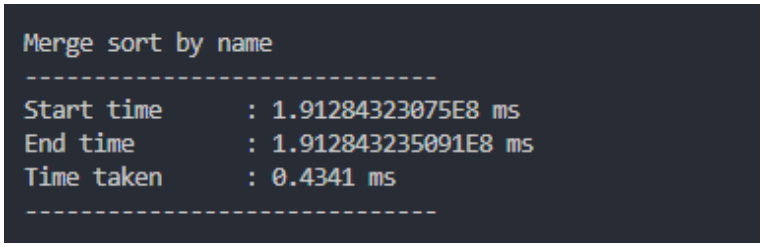
```
private void mergeName(
    List<LeaderboardEntry> leaderboard,
    List<LeaderboardEntry> left,
    List<LeaderboardEntry> right) {
  int leftIndex = 0;
  int rightIndex = 0;
  int leaderboardIndex = 0;

  while (leftIndex < left.size() && rightIndex < right.size()) {
    if (left.get(leftIndex).getName().compareTo(right.get(rightIndex).getName()) < 0) {
      leaderboard.set(leaderboardIndex++, left.get(leftIndex++));
    } else {
      leaderboard.set(leaderboardIndex++, right.get(rightIndex++));
    }
  }

  // Copy remaining elements from left
  while (leftIndex < left.size()) {
    leaderboard.set(leaderboardIndex++, left.get(leftIndex++));
  }

  // Copy remaining elements from right
  while (rightIndex < right.size()) {
    leaderboard.set(leaderboardIndex++, right.get(rightIndex++));
  }
}
```

Figure 5.28: Merge Sort by Name Implementation

The merge sort name algorithm above is used to sort the leaderboard entries based on the player names lexicographically using compareTo. The method recursively splits the list of entries into two halves, sorts each half, and then merges back together in lexicographical order using compareTo method.

```java
System.gc();
double startTime = System.nanoTime();
System.out.println(x:"\nMerge sort by name");
System.out.println(x:"------------------------------");
System.out.println("Start time\t: " + (startTime / 1_000_000) + " ms");
mergeSortName(leaderboard);

double endTime = System.nanoTime();
System.out.println("End time\t: " + (endTime / 1_000_000) + " ms");

double timeDiff = endTime - startTime;
System.out.println("Time taken\t: " + (timeDiff / 1_000_000) + " ms");
System.out.println(x:"------------------------------\n");
```

Figure 5.29: Merge Sort by Name Timing Code

Size Data	N = 4
Attempt 1	 0.4183 milliseconds
Attempt 2	 0.4226 milliseconds
Attempt 3	

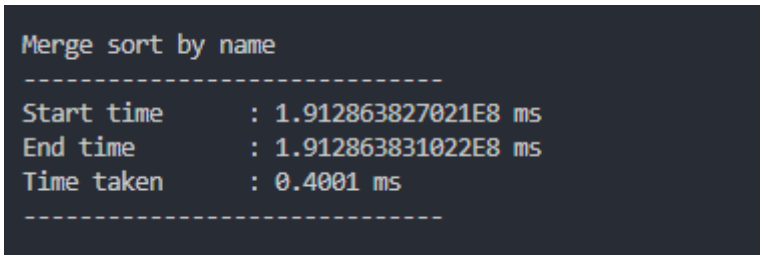
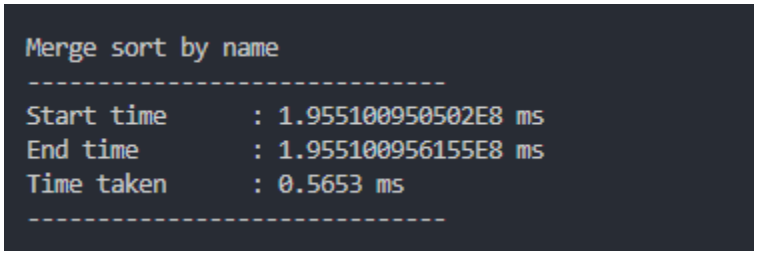
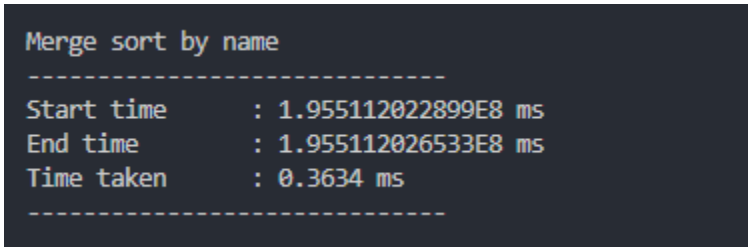
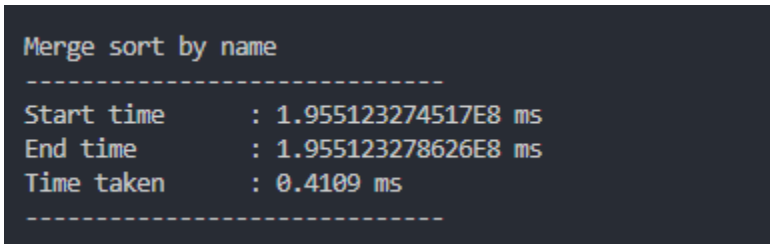
	0.4341 milliseconds
Attempt 4	 0.4001 milliseconds
Average	(0.4183 + 0.4226 + 0.4341 + 0.4001) / 4 = 0.4188 milliseconds

Table 5.20: Merge Sort by Name Timing (N = 4)

Size Data	4N = 16
Attempt 1	 0.5653 milliseconds
Attempt 2	 0.3634 milliseconds
Attempt 3	 0.4109 milliseconds

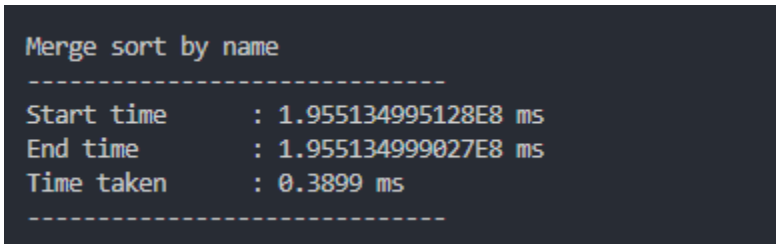
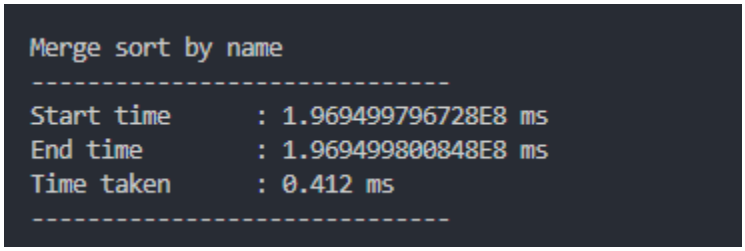
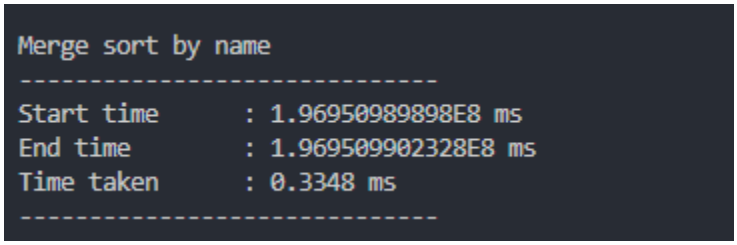
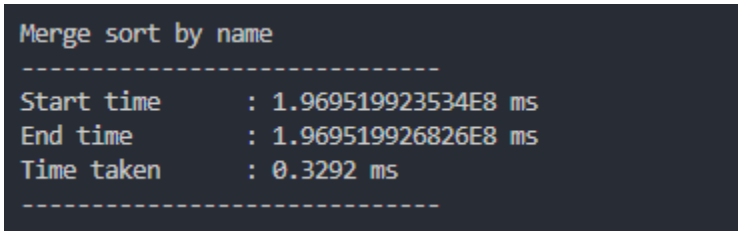
Attempt 4	 0.3899 milliseconds
Average	(0.5653 + 0.3634 + 0.4109 + 0.3899) / 4 = 0.4324 milliseconds

Table 5.21: Merge Sort by Name Timing (4N = 16)

Size Data	6N = 24
Attempt 1	 0.412 milliseconds
Attempt 2	 0.3348 milliseconds
Attempt 3	 0.3292 milliseconds

Attempt 4	Merge sort by name ---------- Start time : 1.969530793414E8 ms End time : 1.969530796725E8 ms Time taken : 0.3311 ms ---------- 0.3311 milliseconds
Average	(0.412 + 0.3348 + 0.3292 + 0.3311) / 4 = 0.3538 milliseconds

Table 5.22: Merge Sort by Name Timing (6N = 24)

5.3.3. Binary Search (Search by Name)

```java
// Binary search by name implementation
public int binarySearchByName(List<LeaderboardEntry> leaderboard, String name) {
  int size = leaderboard.size();
  int low = 0;
  int high = size - 1;

  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (leaderboard.get(mid).getName().equals(name)) {
      return mid;
    } else if (leaderboard.get(mid).getName().compareTo(name) < 0) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }

  return -1;
}
```

Figure 5.30: Binary Search by Name Implementation

The binary search by name algorithm is used to find a specific leaderboard entry by the player's name. It repeatedly divides the list in half, comparing the target name with the middle element. If the name matches, it returns the corresponding entry. Otherwise, it narrows the search to either the left or right half based on the lexicographical comparison.

```
System.gc();
startTime = System.nanoTime();
System.out.println(x:"\nBinary search by name");
System.out.println(x:"------------------------------");
System.out.println("Start time\t: " + (startTime / 1_000_000) + " ms");

int index = binarySearchByName(leaderboard, name);

endTime = System.nanoTime();
System.out.println("End time\t: " + (endTime / 1_000_000) + " ms");

timeDiff = endTime - startTime;
System.out.println("Time taken\t: " + (timeDiff / 1_000_000) + " ms");
System.out.println(x:"------------------------------\n");
```

Figure 5.31: Binary Search by Name Timing Code

Size Data	N = 4

Attempt 1	`Binary search by name`
	`------------------------------`
	`Start time : 1.9178802100 91E8 ms`
	`End time : 1.9178802140 27E8 ms`
	`Time taken : 0.3936 ms`
	`------------------------------`
	0.3936 milliseconds

Attempt 2	`Binary search by name`
	`------------------------------`
	`Start time : 1.9178897451 02E8 ms`
	`End time : 1.9178897493 85E8 ms`
	`Time taken : 0.4283 ms`
	`------------------------------`
	0.4283 milliseconds

Attempt 3	`Binary search by name`
	`------------------------------`
	`Start time : 1.9179044568 69E8 ms`
	`End time : 1.9179044602 8E8 ms`
	`Time taken : 0.3411 ms`
	`------------------------------`
	0.3411 milliseconds

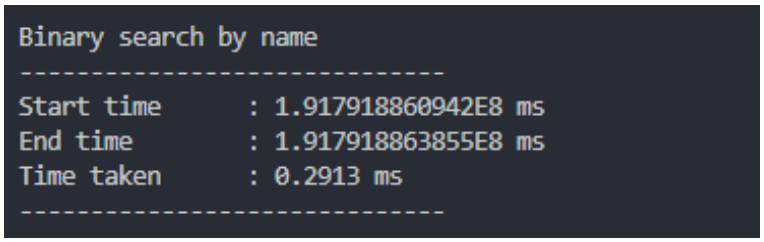
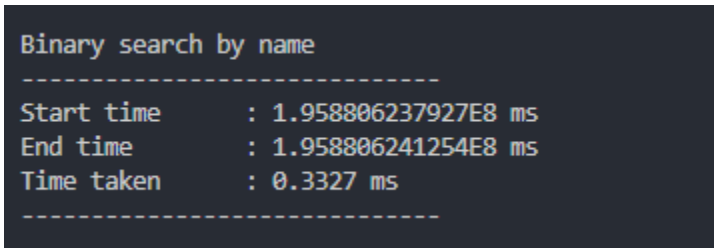
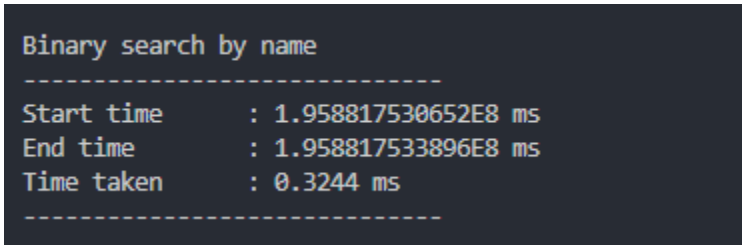
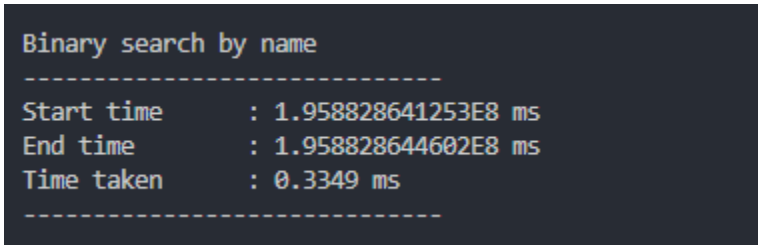
Attempt 4	 0.2913 milliseconds
Average	(0.3936 + 0.4283 + 0.3411 + 0.2913) / 4 = 0.3636 milliseconds

Table 5.23: Binary Search by Name Timing (N = 4)

Size Data	4N = 16
Attempt 1	 0.3327 milliseconds
Attempt 2	 0.3244 milliseconds
Attempt 3	 0.3349 milliseconds

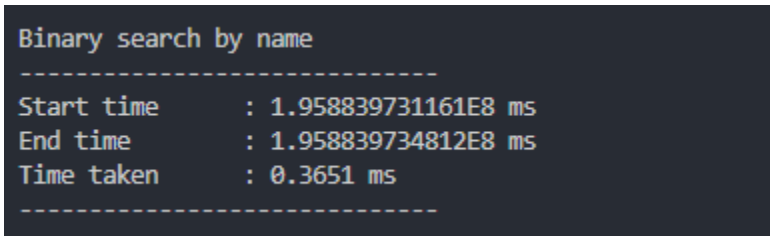
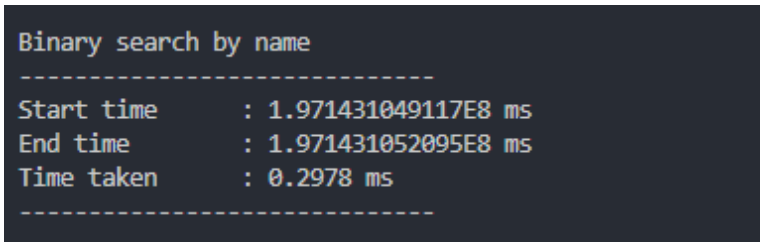
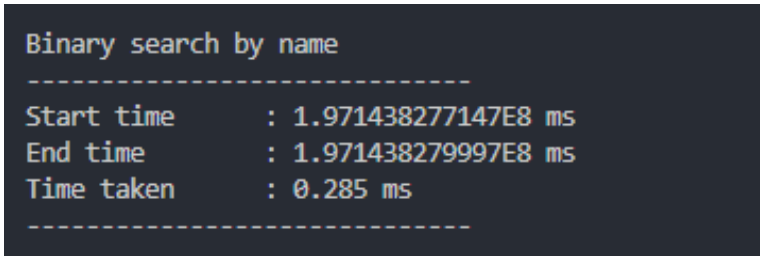
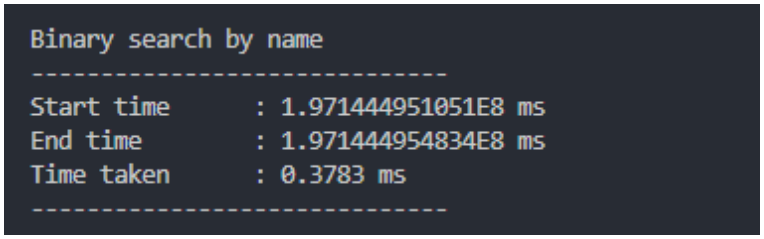
Attempt 4	 0.3651 milliseconds
Average	(0.5653 + 0.3634 + 0.4109 + 0.3899) / 4 = 0.3393 milliseconds

Table 5.24: Binary Search by Name Timing (4N = 16)

Size Data	6N = 24
Attempt 1	 0.2897 milliseconds
Attempt 2	 0.285 milliseconds
Attempt 3	 0.3783 milliseconds

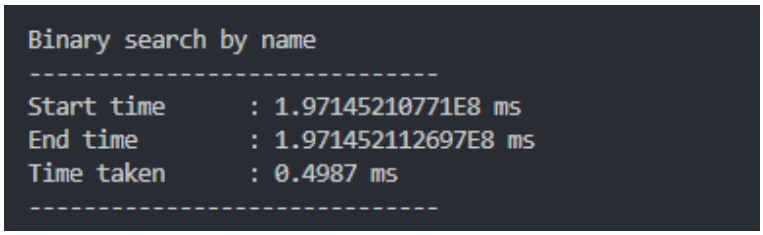
Attempt 4	 0.4987 milliseconds
Average	(0.2897 + 0.285 + 0.3783 + 0.4987) / 4 = 0.3629 milliseconds

Table 5.25: Binary Search by Name Timing (6N = 24)

5.3.4. Analysis of Timing

Merge sort has a time complexity of O(n log n). Theoretically, it will increase as the size of N grows. However, with a relatively small increase in our dataset size, such as from 4 records (N = 4) to 24 records (6N = 24), the timing difference might not be noticeable as the algorithm is efficient with small datasets. An interesting thing we noticed is that in the data above, some larger datasets (e.g., 16 records in merge sort by score) appear to be processed faster than smaller ones like 4 records. This can be caused by a lot of factors. When the algorithm runs multiple times, small fluctuations in timing can occur and could be due to system state at the time of execution (background tasks and processes consuming resources), system load variations that can affect the resource allocation for the algorithm and etc.

6. Testing

6.1. Main Menu

Test Case ID	Description	Test Data	Expected Result	Actual Result	Pass/Fail
TC1	Verify menu navigation	Run the Game	The menu should display option buttons like "Start Game", "Leaderboard", "Settings"	Option buttons displayed correctly	Pass
TC2	Verify "Start Game" button (No name input)	Click the "Start Game" button without inputting a name	The game should display a prompt asking the player to input their name	Name input prompt displayed	Pass
TC3	Verify "Start Game" button (Valid name input)	Input name ("Player1") and click "Start Game"	The game should start and transition to the game selection page.	Transitioned to the game selection page.	Pass
TC4	Verify "Leaderboard" button	Click the "Leaderboard" button	Leaderboard page should open	Leaderboard page opened	Pass
TC5	Verify "About" button	Click the "About" button	About page should open	About page opened	Pass
TC6	Verify name uniqueness (Multiple browsers)	Input name "Player1" in Browser 1, then try to input "Player1" again in Browser 2	Name is taken error message should appear for duplicate name across players on different browsers	Name is taken error message displayed	Pass

Table 6.1: Main Menu Test Case Results

6.2. Game Type Selection

Test Case ID	Description	Test Data	Expected Result	Actual Result	Pass/Fail
TC7	Verify "Question-Based" game selection	Select "Question-Based" from the game type options	The game should transition to the question-based game difficulty page	Transitioned to the question-based game difficulty page	Pass
TC8	Verify "Word-Guessing" game selection	Click the "Word-Guessing" from the game type options	The game should transition to the word-guessing game difficulty page	Transitioned to the word-guessing game difficulty page	Pass
TC9	Verify back button	Click the "Back" button on the game type selection page	The game should return to the main menu	Returned to the main menu	Pass

Table 6.2: Game Selection Test Case Results

6.3. Game Difficulty Selection

Test Case ID	Description	Test Data	Expected Result	Actual Result	Pass/Fail
TC10	Verify "Easy" difficulty selection	Select "Easy" difficulty for selected game type	The game should transition to the easy level game page	Transitioned to the easy level game page	Pass

TC11	Verify "Medium" difficulty selection	Select "Medium" difficulty for selected game type	The game should transition to the medium level game page	Transitioned to the medium level game page	Pass
TC12	Verify "Hard" difficulty selection	Select "Hard" difficulty for selected game type	The game should transition to the hard level game page	Transitioned to the hard level game page	Pass
TC13	Verify back button	Click the "Back" button on the game difficulty selection page	The game should return to the game selection page	Returned to the game selection page	Pass

Table 6.3: Game Difficulty Selection Test Case Results

6.4. Question-Based Game

Test Case ID	Description	Test Data	Expected Result	Actual Result	Pass/Fail
TC14	Verify question display with possible answers	Answer the first question correctly	The question should be displayed along with possible answer options.	The question displayed along with possible answer options.	Pass
TC15	Verify question display for selected difficulty	Select a difficulty level (easy, medium or hard)	The question displayed should match the selected difficulty level.	The question displayed the selected difficulty level words.	Pass
TC16	Verify score increment after correct answer	Answer the first question correctly	The score should increase by 10 points after	The score increases by 10 points after	Pass

			answering correctly.	answering correctly.	
TC17	Verify streak of correct answers	Answer 5 consecutive questions correctly	The score should be increased by 100	The score increased by 100	Pass
TC18	Verify life decrement after incorrect answer	Answer a question incorrectly	One life should be lost, and the hangman should be drawn part by part	Life lost and hangman drawn	Pass
TC19	Verify timer countdown	Start the question timer	The timer should count down from the set time limit 20 seconds	The timer counts down correctly	Pass
TC20	Verify timeout	Let the timer run out before answering	Player should lose one life, and a new question should appear	Player loses one life, and a new question should appear	Pass
TC21	Verify game over condition	Answer all questions incorrectly or lose all lives	A "Game Over" modal window should pop up	Game over modal displayed	Pass
TC22	Verify game passed condition	Answer all 20 questions correctly	A "You Passed" modal window should pop up.	Game passed modal displayed	Pass
TC23	Verify high score update in database	Achieving a higher score than the previous best	New high score should be written to the database if they’re higher than the previous record	New high score updated in database	Pass

TC24	Verify "Play Again" button functionality	Click "Play Again" on the modal window	The game should be reset and allow the user to start the game again from the beginning	Game restarted	Pass
TC25	Verify "Main Menu" button functionality	Click "Main Menu" on the modal window	The game should return to the main menu screen	Returned to main menu	Pass
TC26	Verify "Back" button functionality	Click the "Back" button on the top left of the game page	The game should return to the game difficulty selection page	Returned to the game difficulty selection page	Pass

Table 6.4: Question-Based Test Case Results

6.5. Word-Guessing Game

Test Case ID	Description	Test Data	Expected Result	Actual Result	Pass/Fail
TC27	Verify word display with letter blanks	Start a new word-guessing game and display the first word	The word should display as blanks (e.g., "_ _ _ _") corresponding to its length	The word display as blanks corresponding to its length	Pass
TC28	Verify word matches selected difficulty level	Select a difficulty level (easy, medium or hard)	The displayed word should match the length and complexity of the selected difficulty	The displayed word matches the length and complexity of the selected difficulty	Pass
TC29	Verify timer starts at 0	Start a new game	The timer should initialize at 0 seconds	The timer started at 0	Pass

TC30	Verify timer increment during gameplay	Guess letters of a word	Timer should increment continuously	Timer incremented continuously	Pass
TC31	Verify score increment after each correct word guess	Guess a correct word	The score should increase by 10 points for each word guessed	The score increases by 10 points	Pass
TC32	Verify life decrement after incorrect guess	Guess an incorrect letter	One life should be lost, and a part of the hangman should be drawn	Life lost and hangman drawn	Pass
TC33	Verify game over condition	Exhaust all lives without guessing the full word	A "Game Over" modal window should pop up	A "Game Over" modal window pops up	Pass
TC34	Verify game passed condition	Guess all letters of the word correctly	A "You Passed" modal window should pop up	A "You Passed" modal window pops up	Pass
TC35	Verify high score update in database	Achieving a higher score than the previous best	New high score should be written to the database if they’re higher than the previous record	New high score updated in database	Pass
TC36	Verify "Play Again" button functionality	Click "Play Again" on the modal window	The game should be reset and allow the user to start the game again from the beginning	Game restarted	Pass
TC37	Verify "Main Menu" button functionality	Click "Main Menu" on the modal window	The game should return to the main menu screen	Returned to main menu	Pass

TC38	Verify "Back" button functionality	Click the "Back" button on the top left of the game page	The game should return to the game difficulty selection page	Returned to the game difficulty selection page	Pass

Table 6.5: Word-Guessing Test Case Results

6.6. Leaderboard

Test Case ID	Description	Test Data	Expected Result	Actual Result	Pass/Fail
TC39	Verify profile panel displays player name	Open leaderboard	Profile panel should show the name "Player1" on the right side	Profile panel shows the name "Player1" on the right side	Pass
TC40	Verify all game scores by difficulty on profile panel	Open leaderboard	Scores for each difficulty level (easy, medium, hard) for each game type should be displayed	Scores for each difficulty level for each game type displayed correctly	Pass
TC41	Verify word game leaderboard filtering by difficulty	Select "Easy" difficulty in Word Game Leaderboard	Data retrieved should show only scores for "Easy" difficulty	Data retrieved show only scores for "Easy" difficulty	Pass
TC42	Verify question game leaderboard filtering by difficulty	Select "Hard" difficulty in Question Game Leaderboard	Data retrieved should show only scores for "Hard" difficulty	Data retrieved show only scores for "Hard" difficulty	Pass
TC43	Verify leaderboard sorting by high	Played with 3 accounts: A (90	The leaderboard should display in	The leaderboard displays in	Pass

	scores (Word Game, Medium)	pts), B (80 pts), C (70 pts)	descending order: A, B, C	descending order: A, B, C	
TC44	Verify leaderboard sorting by high scores (Question Game, Hard)	Played with 3 accounts: X (120 pts), Y (100 pts), Z (90 pts)	The leaderboard should display in descending order: X, Y, Z	The leaderboard should display in descending order: X, Y, Z	Pass
TC45	Verify "Back" button functionality	Click the "Back" button on the leaderboard under profile section	The game should be returned to the main menu screen	Returned to main menu	Pass

Table 6.6: Leaderboard Test Case Results

7. Summary

To conclude, we developed a serious educational game called Hang Em High, which is inspired by the classic Hangman game with a focus on Malaysian History. The game is designed using HTML, CSS and JavaScript for the frontend, while the backend API is powered by Java Spring Boot. The project aligns with the Sustainable Development Goal (SDG) of Quality Education. We target students at all levels from primary school Year 4 (Easy) to Form 5 (Hard). Our goal is to make quality education accessible to everyone, especially underserved communities. Players don't need to purchase textbooks; they can simply access the game via any web browser. In the near future, we plan to make the game fully responsive and compatible with all screen sizes so students can use their mobile devices to learn anytime, anywhere.

Returning to one of the primary objectives of this assignment, which is centered around Data Structures and Algorithms, we use several data structures such as ArrayList, Queue and HashMap, along with algorithms such as Binary Search and Merge Sort in the development of this educational game. Data structures make it efficient and simplifies the way of organizing and managing data. For example, imagine trying to handle 100 numeric data points without arrays or lists, we would need 100 separate variables just for that. With data structures like lists, we can store all 100 values in one container. This makes the code cleaner and much more manageable.

Same thing goes to HashMap that played an important role in our word-guessing game. Without HashMap, counting character occurrences would require nested loops to track each character. This low-level implementation increases the complexity. With HashMap, each character is stored as a unique key. We can count and retrieve occurrences in a single pass through the word, which simplifies logic. This project demonstrates how data structures and algorithms helped in developing efficient and scalable applications and in the context of educational tools that are looking to deliver impactful learning experiences to students. Therefore, choose the most appropriate data structures and algorithms based on specific requirements of each situation to maximize efficiency and performance.

8. References

GeeksforGeeks (2015). *Queue Data Structure - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/queue-data-structure/.

Masai School. (2022). *Queue Data Structure - Types, Applications, JavaScript Implementation*. [online] Available at: https://www.masaischool.com/blog/queue-data-structure-types-applications-javascript-implementation/.

W3schools (n.d.). *DSA Queues*. [online] Available at: https://www.w3schools.com/dsa/dsa_data_queues.php.

GeeksforGeeks (2015). *Stack Data Structure - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/stack-data-structure/.

GeeksforGeeks. (2018). *ArrayList vs LinkedList in Java - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/arraylist-vs-linkedlist-java/.

Desir, D. (2021). *Algorithms: Binary Search*. [online] Medium. Available at: https://dolly-desir.medium.com/algorithms-binary-search-2656c7eb5049.

GeeksforGeeks (2019). *Binary Search - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/binary-search/.

W3schools (n.d.). *DSA Binary Search*. [online] Available at: https://www.w3schools.com/dsa/dsa_algo_binarysearch.php.

GeeksforGeeks (2018). *Merge Sort - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/merge-sort/.

W3Schools (n.d.). *DSA Merge Sort*. [online] www.w3schools.com. Available at: https://www.w3schools.com/dsa/dsa_algo_mergesort.php.

GeeksforGeeks (2017). *HashMap in Java with Examples*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/.

W3schools (2019). *Java HashMap*. [online] W3schools.com. Available at: https://www.w3schools.com/java/java_hashmap.asp.

Ren, J., Xu, W. and Liu, Z. (2024). The Impact of Educational Games on Learning Outcomes: Evidence From a Meta-Analysis. *International Journal of Game-Based Learning (IJGBL)*, [online] 14(1), pp.1–25. doi:https://doi.org/10.4018/IJGBL.336478.

Li, Y., Chen, D. and Deng, X. (2024). The impact of digital educational games on student's motivation for learning: The mediating effect of learning engagement and the moderating effect of the digital environment. *PLOS ONE*, 19(1), pp.e0294350–e0294350. doi:https://doi.org/10.1371/journal.pone.0294350.

9. Appendix

9.1. Environment Setup Instructions

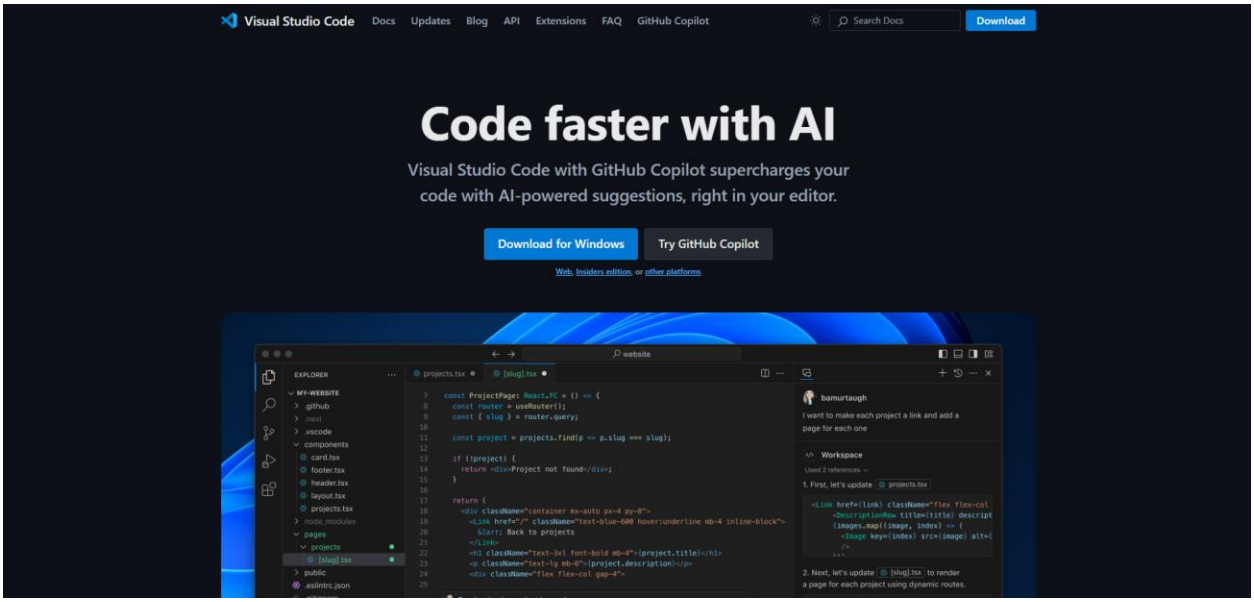


Figure 9.1: Visual Studio Code Official Website

1. Visit the https://code.visualstudio.com/.
2. Download the installer for your system if you haven't already installed it.

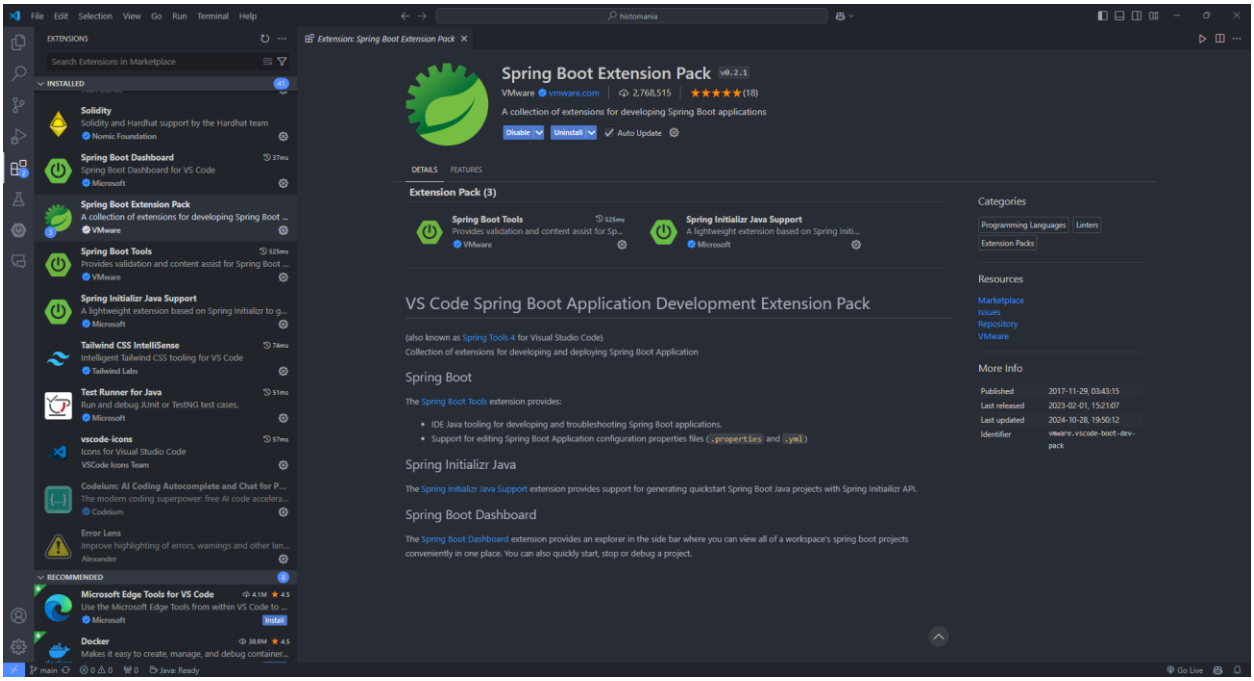


Figure 9.2: Installing Spring Boot Extension Pack in Visual Studio Code

3. Open Visual Studio Code.
4. Go to Extensions Panel (CTRL + SHIFT + X).
5. Search for "Spring Boot Extension Pack" and install it. This enables support for Spring Boot in your development environment.

Figure 9.3: Installing Extension Pack for Java in Visual Studio Code

6. In the Extension Panel, search for "Extension Pack for Java" and install it. This pack includes the necessary tools for Java development in VS Code.

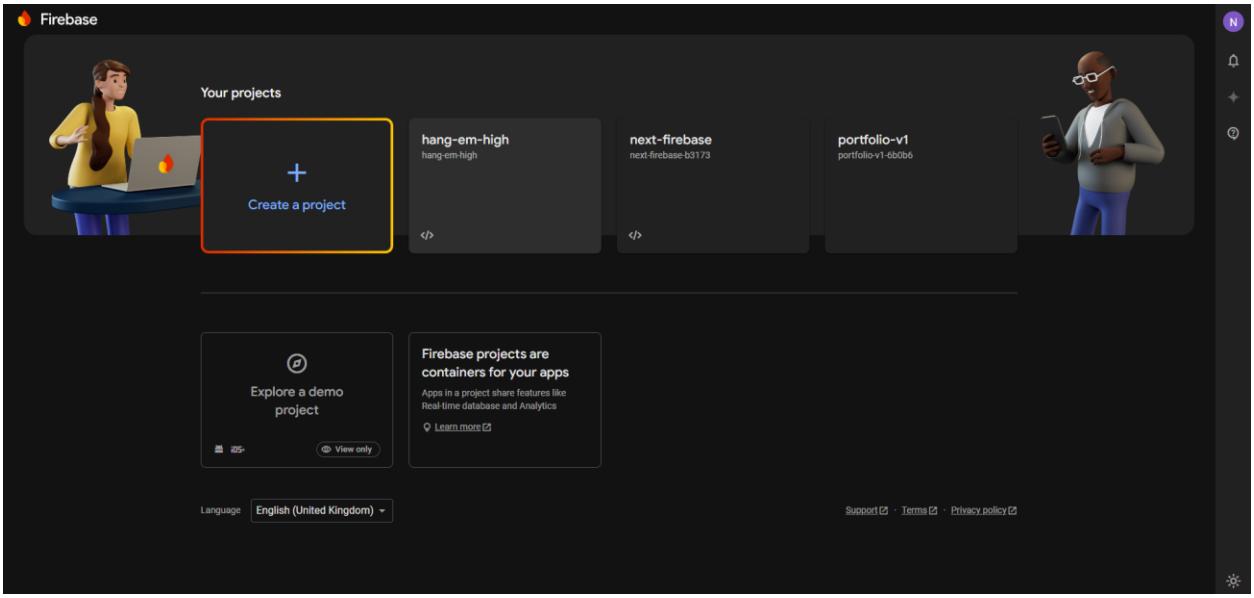


Figure 9.4: Create a New Firebase Project

7. Open your web browser and visit https://console.firebase.google.com/.

8. If you don’t have a Firebase project, click on "Create a Project" and follow the prompts to create one.

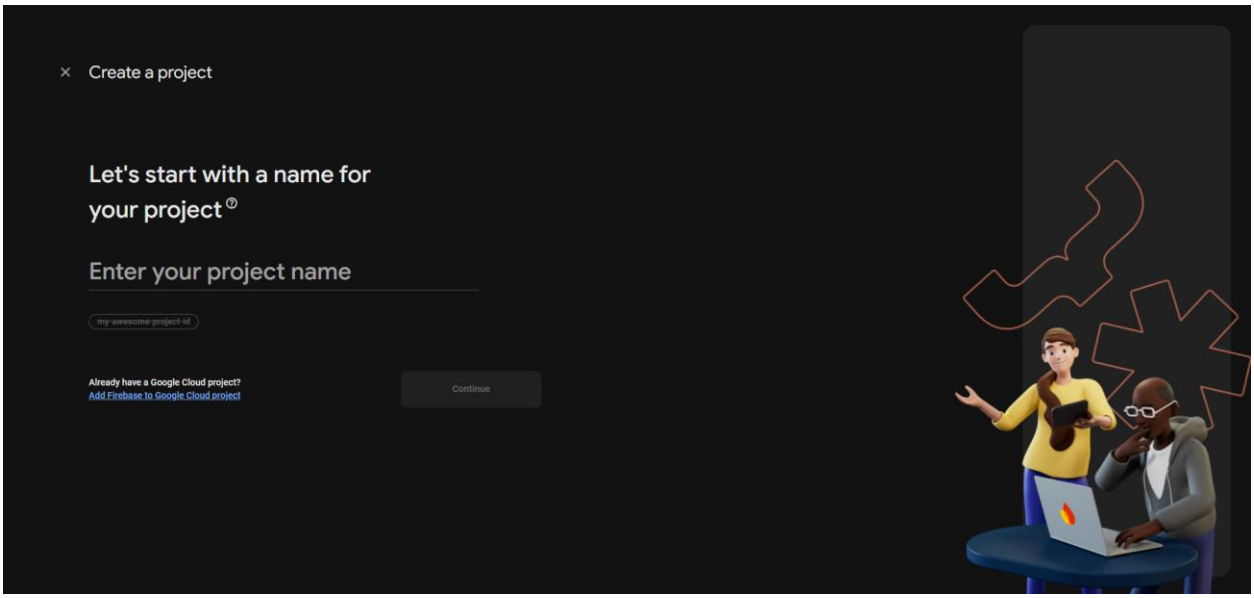


Figure 9.5: Entering Project Name in Firebase

9. Enter your project name and follow the instructions provided by Firebase.

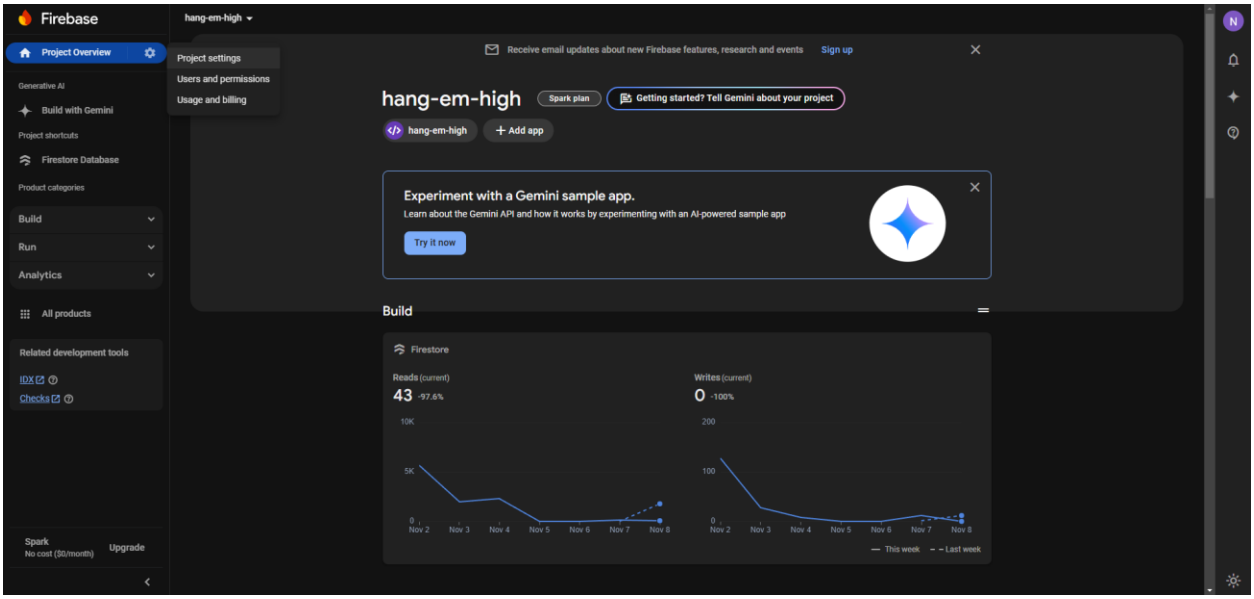


Figure 9.6: Firebase Project Overview Dashboard

10. Once your project is created, you will be taken to the Firebase Dashboard.

11. At the top left, click on the settings icon (gear) and select Project Settings.

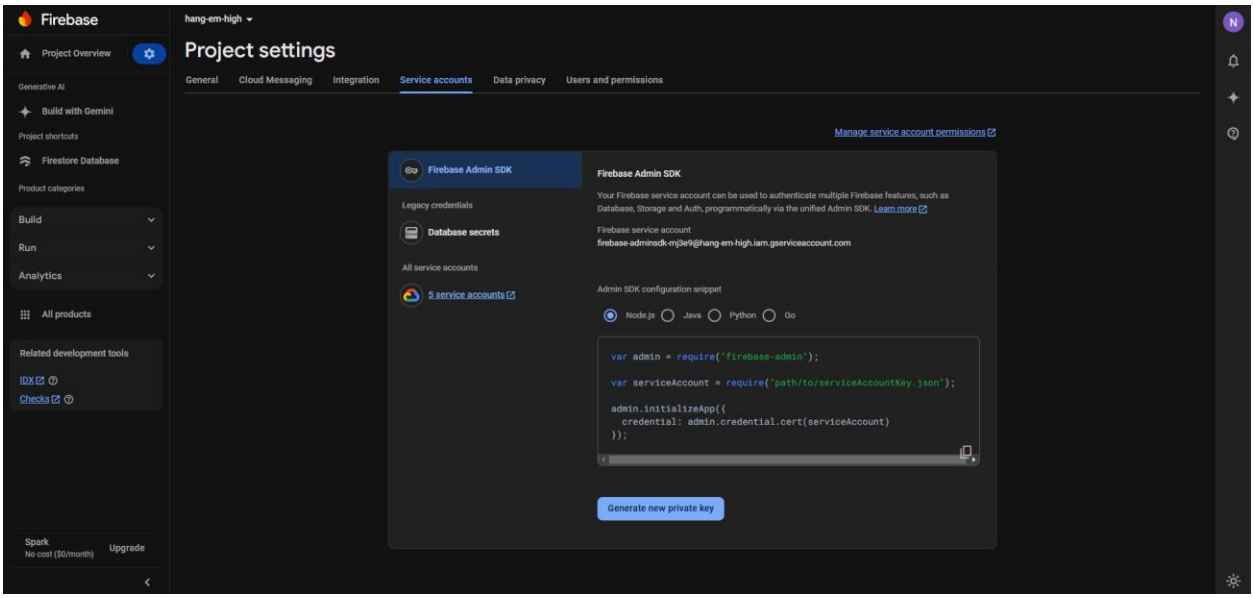


Figure 9.7: Accessing Project Settings in Firebase

12. In the Project Settings page, go to Service Accounts tab.

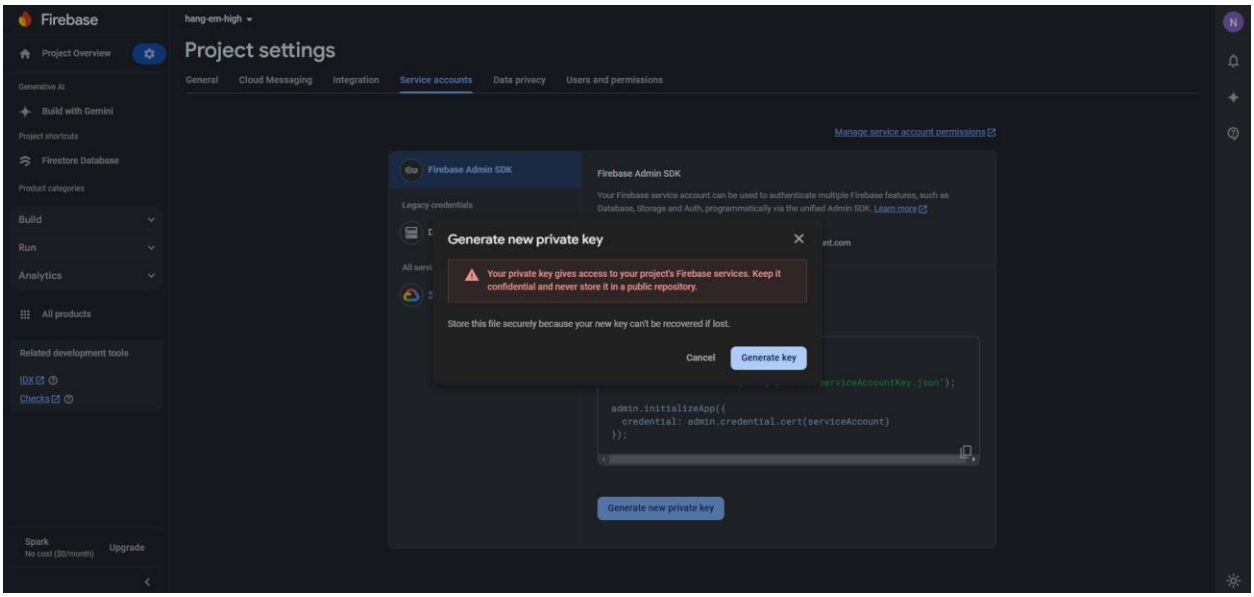


Figure 9.8: Generating a New Private Key in Firebase Project Settings

13. Click on "Generate New Private Key". This will download a .json file.

14. Keep this file safe as it contains sensitive information.

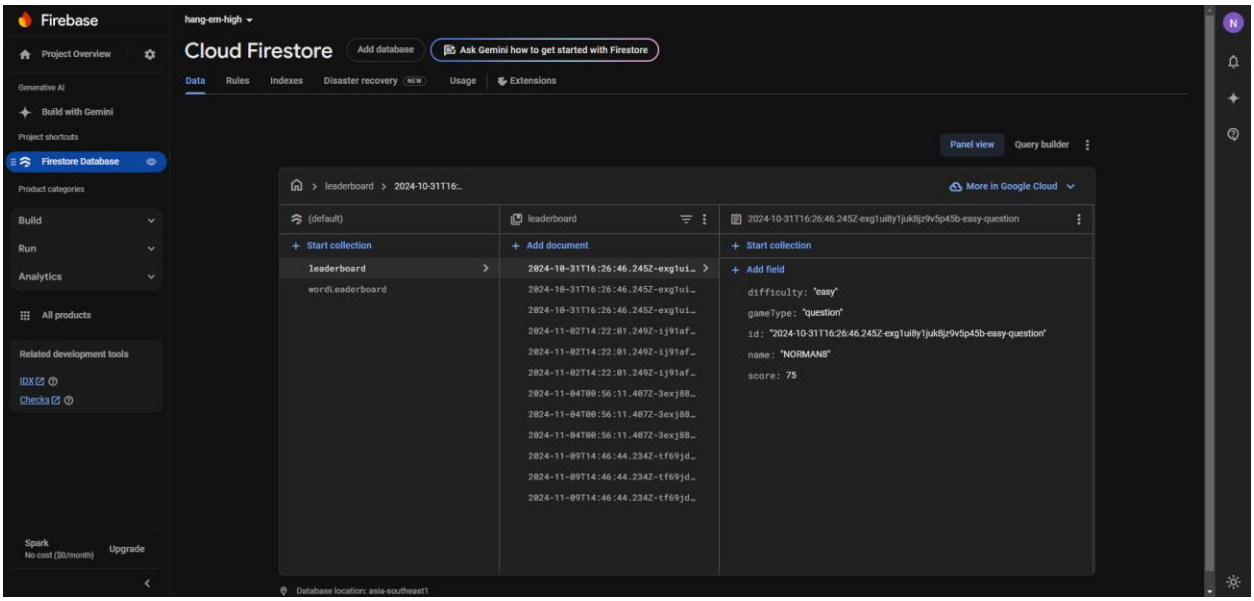


Figure 9.9: Firebase Cloud Firestore

15. In the Firebase Console, search for Firestore Database in the left panel and click on it.
16. If you don’t have a database, click on "Create Database". The default database name for free accounts will usually be "default".
17. Create a collection. You can manually create collections, but the game will automatically create collections as needed once data is available.

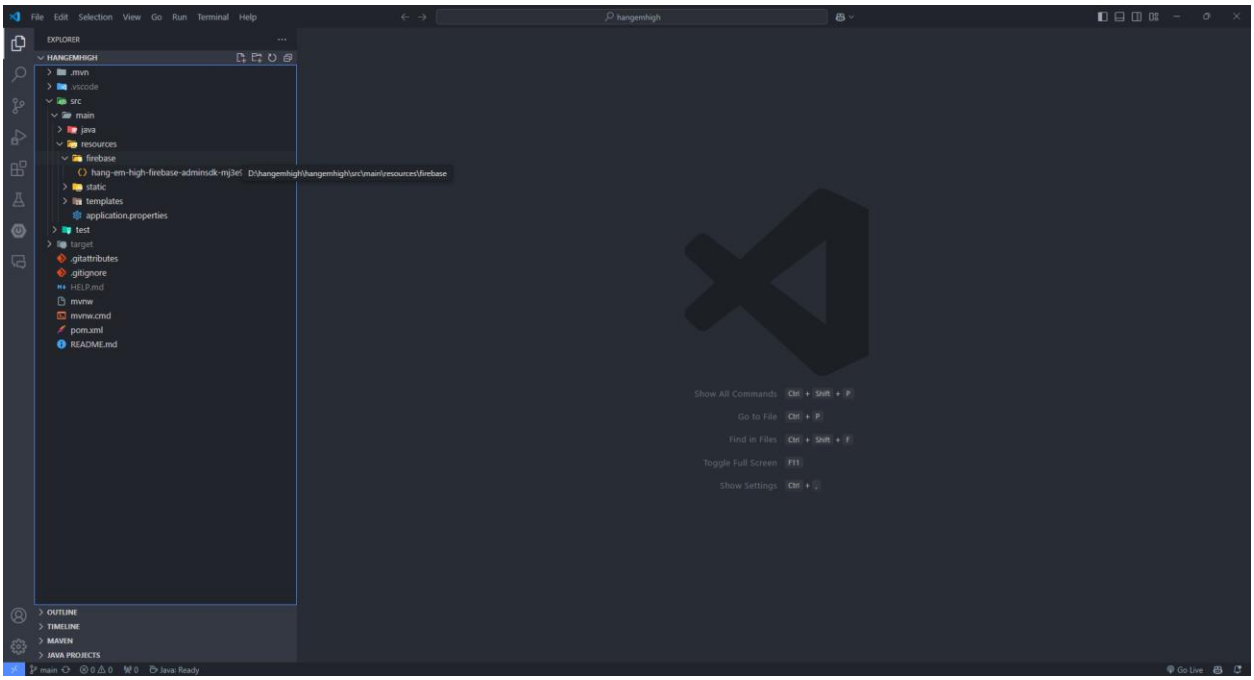


Figure 9.10: Locating Firebase Folder in Visual Studio Code

18. Open Visual Studio Code and navigate to hangemhigh/src/main/resources/firebase folder in the project.
19. Move the .json file you downloaded into this folder.

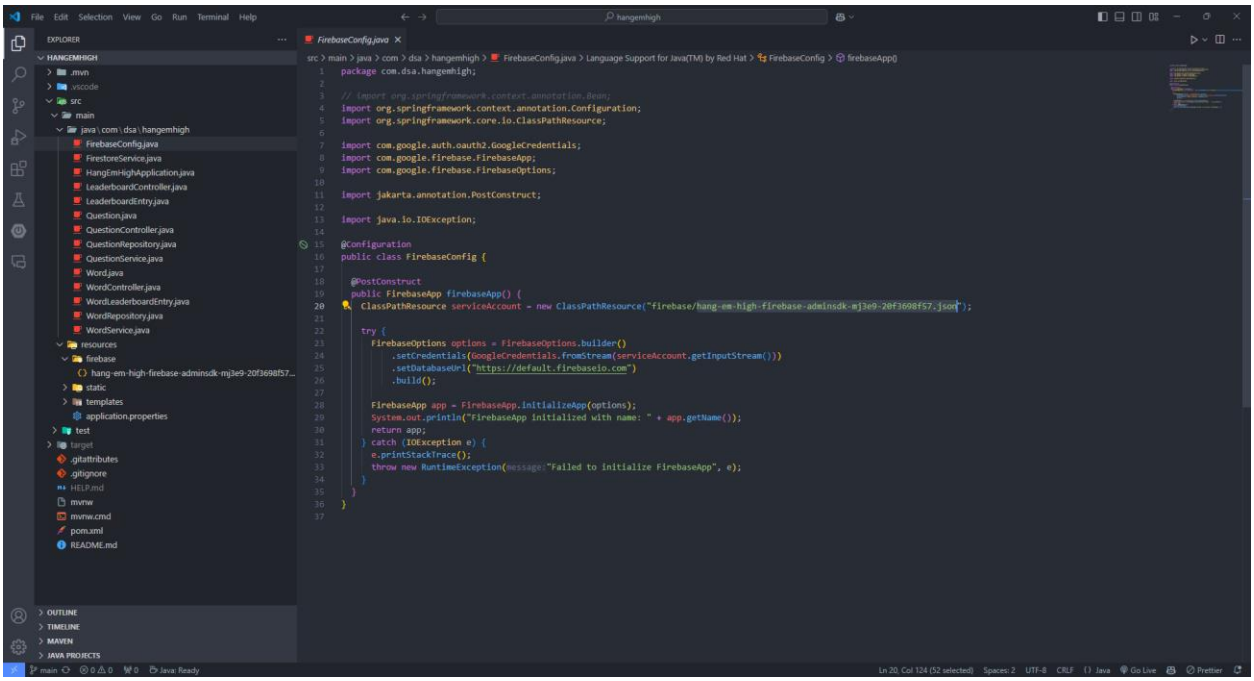


Figure 9.11: Modifying FirebaseCofig.java File

20. Go to hangemhigh/src/main/java/com/dsa/hangemhigh in your project directory.
21. Locate the FirebaseConfig.java file.
22. In this file, look for a line with ClassPathResource. Replace the URL in that line with the path to the .json file you placed in the firebase folder.

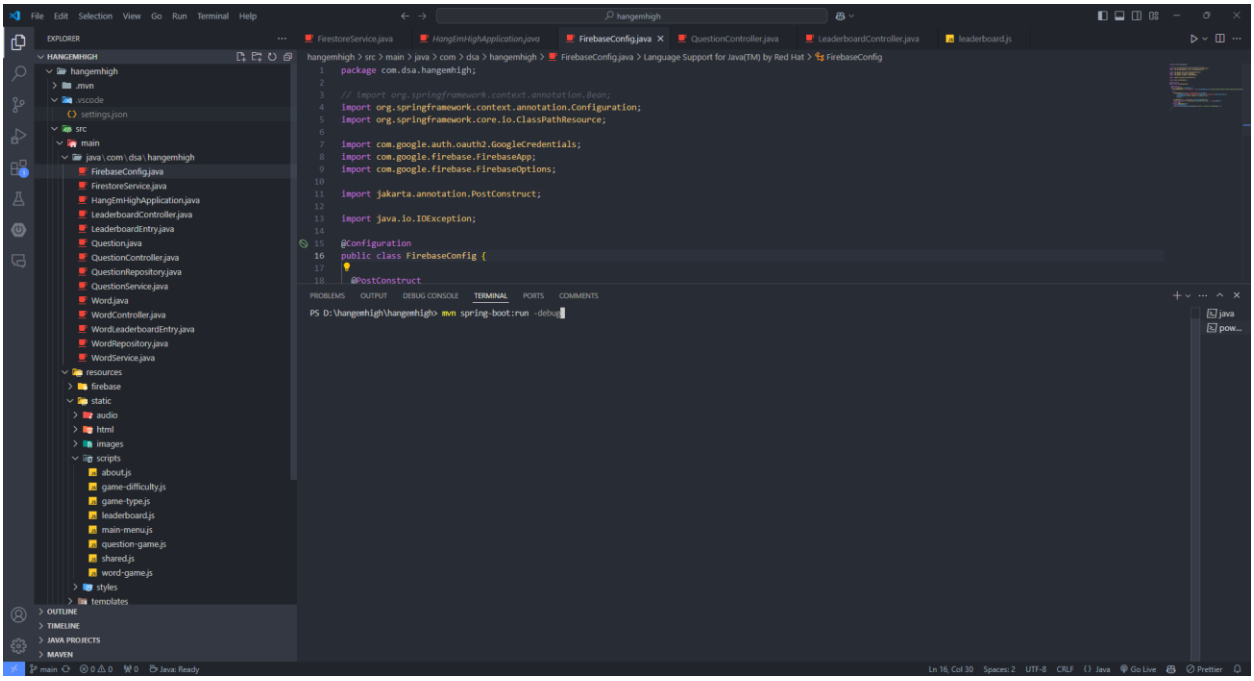


Figure 9.12: Running mvn spring-boot:run -debug in Terminal

23. Open the terminal in VS Code (CTRL + SHIFT + `) and navigate to the root folder of your project.
24. Type the following command: mvn -v. If you see Maven information, it means Maven is already installed. If not, you will need to install Maven.
25. Visit https://maven.apache.org/download.cgi and download the latest Maven version.
26. Follow the Maven installation guide https://phoenixnap.com/kb/install-maven-windows for detailed steps on how to install Maven on your system.
27. If Maven is already installed, run the following Maven command to start the Spring Boot application with debugging enabled: mvn spring-boot:run -debug.

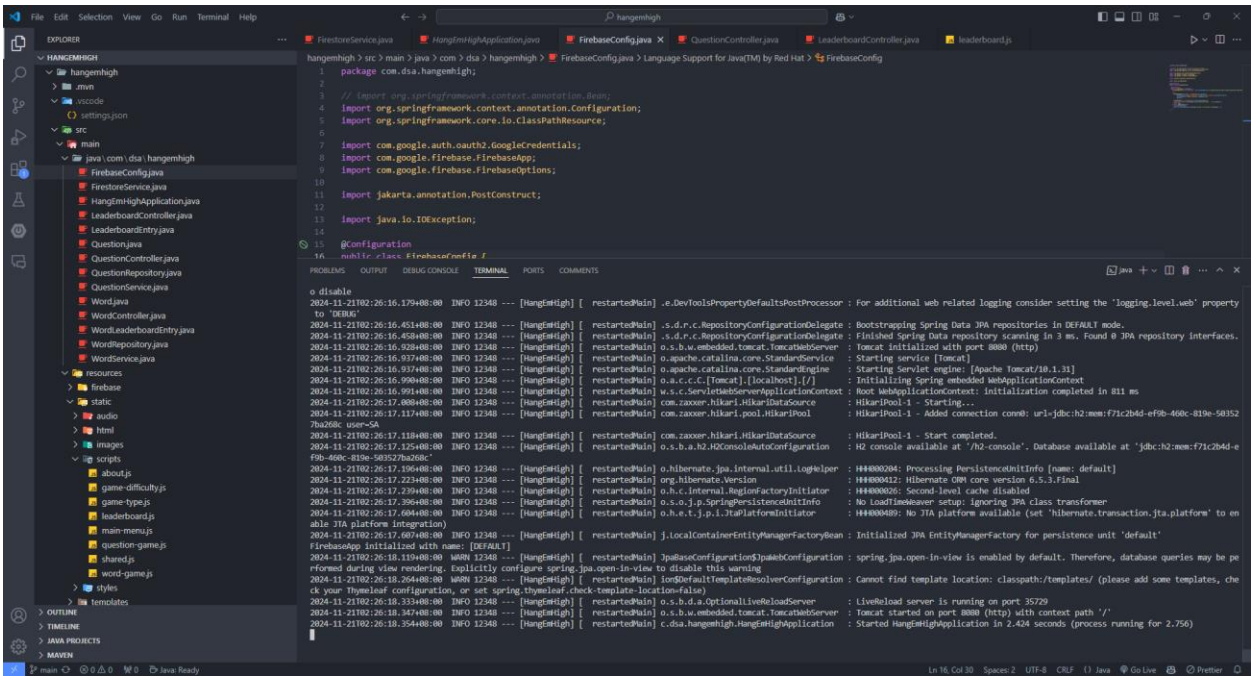


Figure 9.13: Output Message After Running Spring Boot Application

28. This will start the application and show any debug messages in the terminal. If there are no issues, the application should run successfully.
29. Visit http://localhost:8080/html/index.html for the game application after running the command successfully.

9.2. Task Distribution Table

Team	Teo Chung Henn	Liew Wen Yen	Chong Ken Ji
Game Idea		/	
Game Design			/
Game Assets (e.g., Hangman, sounds…)			/
User Interface (HTML / CSS / JS)	/		
Data Structure (Question Game) (Java)		/	
Data Structure (Word Game) (Java)			/
Data Structure (Leaderboard) (Java)	/		
Algorithm (Leaderboard) (Java)	/		
Documentation	/	/	/

Table 9.1: Task Distribution Table