Final Project CPSC 39 Report

The game that I will be making for my final project is a Hangman game made with Java. My game’s purpose is to provide entertainment for the player. Hangman is a game that has been around for many years now, and is also a very common game. The basis of this game is that words and hints are randomly picked from a HashMap of over 100 different words from various sources such as games, anime, movies etc. The player will be provided a set number of empty spaces in the form of underscores in the console menu of the program. In my game, only 5 attempts are allowed per word before a game is considered over. My game also has a built-in scoreboard which keeps track of how many letters and words the player has guessed correctly.

In terms of algorithms, I have incorporated six different algorithms within my game. They are as follows: random word selection, iterative matching(word guessing/input validation), string comparison(game-win detection), iterative initialization(game initialization), interactive input loop(user input handling) and an incremental update(score tracking) algorithm. The three algorithms that I will be mainly discussing are the input validation, game initialization and the user input handling algorithm.

The first algorithm that I will be discussing is the input validation algorithm. The steps for this algorithm are as follows with a snapshot of the code:

Step 1: Start

* Begin the Hangman game.

Step 2: Prompt input

* Display the current state of the guessed word and remaining attempts.
* Prompt the user to enter a letter.

Step 3: Validate input

* Check if the input is a valid character (a single letter).
  + If invalid:
    - Display "Invalid input. Please enter a single letter."
    - Loop back to prompt for input again.

Step 4: Check guess

* Verify if the guessed letter is present in the selected word:
  + True:
    - Update the guessed word.
    - Display “Good Guess!”
  + False
    - Decrement attempts left.
    - Display “Wrong Guess!”

Step 5: Check win condition

* Verify if the guessed word matches the selected word:
  + True:
    - Display a congratulations message and increment the score.
    - Prompt to play again
  + False
    - If attempts are 0
      * Display "Game over!" with the correct word.
      * Prompt to play again.

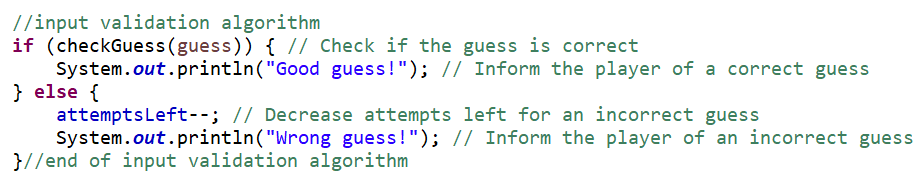
Step 6: Replay decision

* Check if the user wants to play again
  + Yes
    - Reset the game
    - Loop back to prompt input
  + No
    - End the game

Step 7: End

* Exit the Hangman game

Snapshot of the code:



I created this algorithm by using a loop that consists of an “if” and an “else” statement that constantly checks if the guessed letter is correct or not. If a letter is guessed correctly then, the program will print out “Good Guess!” and it will print out “Wrong Guess!” if a wrong letter is guessed. How this is utilized within my Hangman game is that when a letter is guessed correctly, the system will print out “Good Guess!” If a letter is guessed incorrectly, then the number of attempts will be deducted by one and the system will print out “Wrong Guess!”

The Big O time of this algorithm occurs twice within my playGame() method at the line char guess = scanner.next().charAt(0);. The first being at the scanner.next() method. Depending on how long the input is, if the user inputs one character, it takes constant time **O(1)**. But in the worst case, it reads a longer string, which will take linear time in relation to the length of the string. So, the time complexity is **O(n)**, where n is the length of the string entered by the user (in this case, the length of the input word).

The second occurs at the charAt(0) method. This method accesses the first character of the string. Since strings are indexed and the operation is constant time, accessing charAt(0) has a time complexity of **O(1)**. Since the input is captured with scanner.Next() and charAt(0) is called afterward, the overall time complexity of the input handling process would be **O(n)**, where **n** is the length of the string entered by the user.

The second algorithm that I will discuss is the iterative initialization (game initialization). The steps for this algorithm are as follows with a snapshot of the code:

Step 1: Start

* The process begins.

Step 2: Initialize HashMap

* A new HashMap (wordHints) is created

Step 3: Add words and hints

* Multiple words and their corresponding hints are added to the wordHints HashMap.

Step 4: Initialize game variables

* lettersGuessed is set to 0.
* wordsGuessedRight is set to 0.

Step 5: Reset game

* The resetGame() method is called.

Step 6: Select random word

* The selectRandomWord() method is invoked to select a random word from wordHints.

Step 7: Set number of attempts

* attemptsLeft is set to 5.

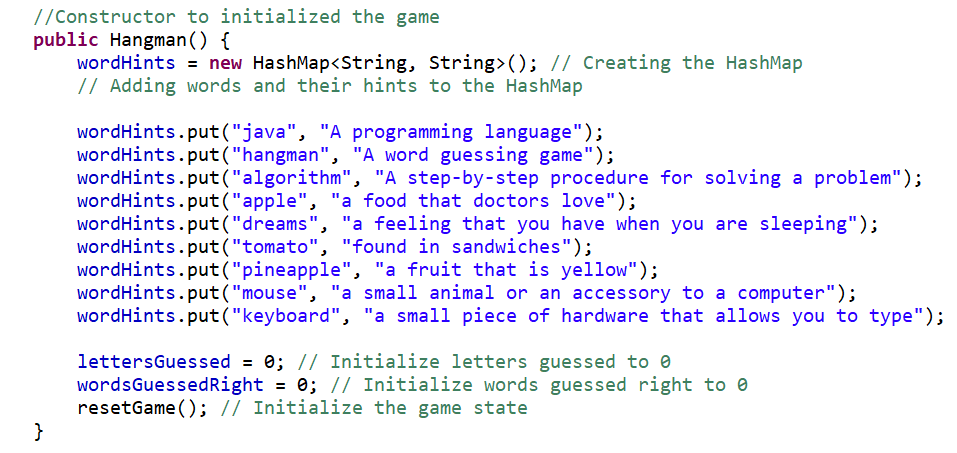
Step 8: Initialize guessed word

* A guessedWord array is initialized, filled with underscores (\_).

Step 9: End of initialization

* The game has been initialized successfully.

Snapshot of the code:



The process in which I created this algorithm is making a constructor called public Hangman and incorporating a HashMap within the constructor. This HashMap has two values which consists of two strings. The first value is the word and the second value is the hint associated with the word.Then I added methods such as lettersGuessed, wordsGuessedRight and set them both equal to zero. I also added a resetGame() method that initializes the game. As for my two values for my HashMap, I currently have over one hundred different words from various sources that range from video games to anime.

The Big O time of this algorithm occurs at two locations in this code: the constructor(Hangman) and the resetGame() method. In the constructor(Hangman), wordHints = new HashMaps<String, String>(); is in constant time **O(1).** Adding entries to the wordHints map: Inserting each entry is **O(1)** because it's just putting a key-value pair into the HashMap. There are **k** words (each insert is **O(1)**), so this part is **O(k)** where **k** is the number of words being added to the map. The parts of my code with (lettersGuessed = 0 and wordsGuessedRight = 0) are both **O(1)**.

The second occurs at the resetGame() method in which the operation selectRandomWord() chooses a random word from the wordHints map. The map has **k** entries, and converting the keys to an array and selecting one randomly is **O(k)** because the keySet().toArray conversion is **O(k)**. The operation guessedWord = new char[selectedWord.length()] initializes an array of length **m** (where **m** is the length of the selected word). This operation is **O(m)**. The loop that initializes the guessedWord array runs **m** times, setting each element of the array to “\_”. Each operation inside the loop is **O(1)** therefore, the total complexity for this part is **O(m)**. The constructor’s initialization of wordHints is **O(k)**, where **k** is the number of words added to the map.The resetGame() method involves: the selectRadomWord() method which is **O(k)**, the array initialization **O(m)** (where **m** is the length of the selected word) and the loop for initializing the guessedWord array **O(m).** In total, the time complexity is **O(k) + O(k) + O(m).** Since both k and m can vary, the overall time complexity can be simplified down to **O(k + m).** **K** is the number of words in the wordHints map (for random word selection) and **m** is the length of the selected word (for initializing the guessed word).

The last algorithm that I will discuss is the interactive input loop(user input handling) algorithm. The steps for this algorithm are as follows with a snapshot of the code:

Step 1: Start

Step 2: Display hint and prompt for input

* Show the hint for the selected word.
* Prompt the user to enter a letter.

Step 3: User inputs a letter

* Input: The letter guessed by the user.

Step 4: Check if guess is correct

* Condition: Is the guessed letter in the word?
  + Yes
    - Update guessed word
      * Update the guessed word with the correct letter.
      * Increment the letters guessed counter.
      * Display "Good guess!"
  + No
    - Decrement attempts left
      * Decrease the remaining attempts for an incorrect guess.
      * Display "Wrong guess!"

Step 5: Check if the game is won

* Condition: Is the guessed word equal to the selected word?
  + Yes
    - Increment words guessed right
      * Increase the count of words guessed right
      * Display “Congratulations!”
      * Break out the loop and move to step 7
  + No
    - Continue the loop (back to step 2)

Step 6: Check if attempts are exhausted

* Condition: Are there any attempts left?
  + No
    - Game over
      * Display "Game Over" and show the correct word.
      * Break out of the loop and move to step 7.

Step 7: Display scoreboard

* Display the current score: number of letters guessed and number of words guessed correctly.

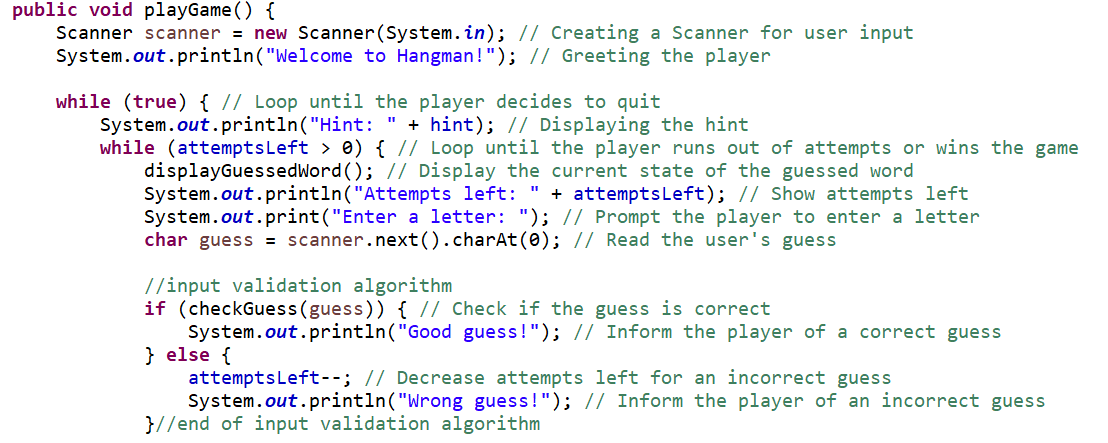
Step 8: Ask if the user wants to play again

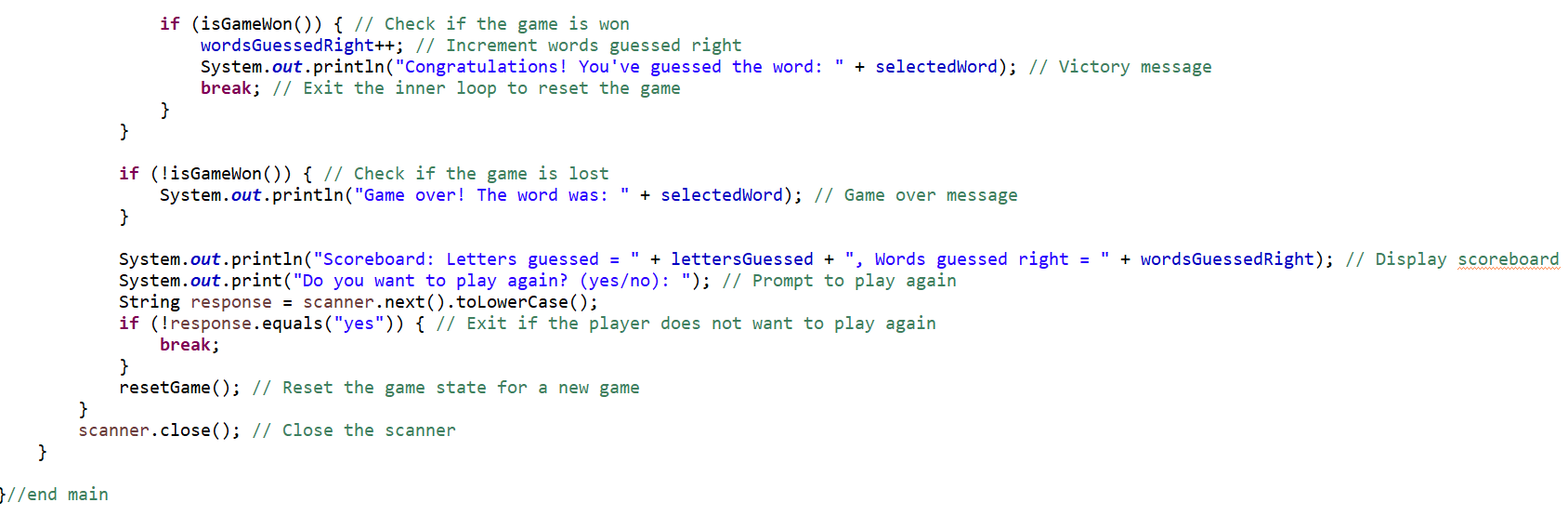
* Input: The user's response ("yes" or "no").

Step 9: Check user response

* Condition: Did the user say "yes"?
  + Yes
    - Reset the game
      * Reset game parameters (attempts, guessed word, etc.).
      * Go back to step 2 to start a new round.
  + No
    - End
      * End the game

Snapshot of the first half of the code:



Second half of the code:

The process in which I created these lines of code is by creating a method called playGame. I then add in a scanner for the user in which they can input into. Then I start off the game by having the program welcome the player by saying “Welcome to Hangman.” Next, I add in a while loop that continuously loops until the player decides to quit the game. Then I add in code that prints out the hint of the associated word. After that I added in another while loop, in which within that loop, the program checks for the amount of guesses the player has remaining. If the number of guesses is greater than zero then, it will loop forever until the player guesses a wrong letter. If an incorrect guess is made, then within the “else” statement, the number of attempts will be deducted by one.

As for the second part of the code, I began by adding an “if” statement that checks if the game is won. If a word is guessed right, then the wordsGuessedRight++ will increment up by one.At the same time the system will print out (“Congratulations!, you’ve guessed the word”+ selectedWord). By adding a break, the program ends the loop and resets the game. Vice versa, if the number of attempts reaches zero, then the “if” statement that I made will end the game and print out “Game over! The word was” + selectedWord). I then added twoSystem.out.println in which the first one adds a scoreboard to the game and the other asks if the player wants to play again. Lastly, I added another “if” statement in which exits the game if the player decides not to keep playing. This then prompts the reseGame() method to reset the game.

The Big O time of this algorithm occurs five times within the playGame() method. The first being the Outer while(true) loop. This loop runs until the user chooses not to play again. The number of iterations of this loop is dependent on the number of times the user wants to play the game. This will not affect the time complexity of a single game because it only runs once per game. The second being in the inner while(attemptsLeft > 0) loop. This loop runs as long as the user has attempts remaining (attemptsLeft > 0) or the game is won. The maximum number of iterations in this loop is equal to the number of attempts allowed, which is a constant (5 in this case). So this loop will run at **most 5 times** in a game. The third occurrence being at the checkGuess method. This method iterates through the selectedWord to check if the guessed letter is in the word. The time complexity of this method is **O(n)**, where n is the length of the selected word. For example, if the word is “hangman”, it would iterate through 7 letters. The fourth occurrence is at the displayGuessedWord() method. This method loops through the guessedWord array, which has the same length as the selected word. Therefore, the time complexity of this method is **O(n)**, where n is the length of the word. The last one is at the isGameWon() method. This method compares the guessedWord array with the selectedWord to determine if the game has been won. The time complexity of this method is **O(n)** because it compares each character in the two strings.

Overall, for each guess, the checkGuess() method runs in **O(n)**, where n is the length of the word. The displayGuessedWord() and isGameWon() methods also run in **O(n)** each. Given that the inner loop can run at most 5 times (due to the number of attempts), the total time complexity for a single game is: **O(n) (for checkGuess) + O(n) (for displayGuessedWord) + O(n) (for isGameWon) = O(n).** Since the inner loop can run at most 5 times, this will not change the overall complexity, which remains **O(n)** for each attempt. The final time complexity of the input handling loop is **O(n)**, where n is the length of the selected word. This complexity accounts for the operations inside the inner loop for each guess, and the maximum number of guesses is bounded by a constant (5 attempts). In the best case scenario, the Big O time would be **O(n)** (when the word is guessed early), In the worst case scenario, the Big O would still be **O(n)** (if all attempts are used and the word is guessed correctly or the player runs out of attempts). Considering my game resets and replays, the outer loop would add a constant factor (since it’s limited to the number of times the user wants to play again), so the time complexity of one complete game still remains **O(n)**.

I used three types of data structures within my game which consist of: HashMap, an array (char[]), and a Stack. I used a HashMap for my game because my game has a bit of randomness to it, in which a HashMap would be the most efficient. HashMaps provide an average O(1) time complexity for insertion and retrieval operations, making it efficient for storing and accessing words and hints. It also allows an easy addition of new words and hints, making the game easily extendable. I’ve incorporated this data structure into my project by having it store two values: words and hints. I initialized this in the Hangman constructor as private HashMap<String, String> wordHints;. This HashMap is populated with words and their hints using the put method. It is then used to randomly select a word and retrieve its hint.

I used an array (char[]) for my game because they are more memory efficient than something like a String object. Another reason is that character arrays allow direct manipulation of individual characters, which is necessary for updating the guessed word state. I initialized this in the resetGame method guessedWord = new char[selectedWord.length()];. I used this array to update with correct guesses and have it be displayed to the player. Using character arrays to represent the selected word and the guessed word state makes the code easy to understand and manipulate.

The last data structure that I used is a Stack. A Stack allows the player to see their history of correctly guessed words in sequential order. This allows the player to keep track of how many total words have been guessed correctly so far. The Stack is declared as a private member of the class, initialized in the constructor as guessedWords = **new** Stack<>();, updated in the playGame() method when a word is guessed correctly, and displayed at the end of each game round. With the addition of a Stack, the player can be immediately provided feedback about the words they have guessed right.

A step in the design process where I encountered an opportunity is when I finished programming the entirety of the code and realized that I should make this Hangman game stand out from the rest. To do this, I added a System.out.println within my playGame method in the while (true) loop. At the end of the while (true) loop, I added code in which the program will show the player how many total letters were guessed and the number of words guessed right so far.

A step in the design process where I encountered an error is when I couldn’t figure out how to make the program ignore spaces between words as additional letters. For example, one of my words is Anakin Skywalker. In the terminal where the player types in a letter, there would be a space in between Anakin and Skywalker. The program thinks that the empty is also missing a letter. After dwelling on the idea for about a day and some help from a colleague of mine, I managed to come up with a method named isGameWon. Within that method, I added a return statement which consisted of **return** String.*valueOf*(guessedWord).equals(selectedWord.replace(" ","\_") );. With this line of code, the program was able to distinguish that the space in between words is not an additional letter.

One change I can make in my next version of the game is to add code that allows the program to read upper case letters as well. In the current version of my game, the program is only able to read lower case letter input. If the player tries to type in a capital A for example, the program will consider it as a wrong guess even though a lower case would work. To do this, I would need to add a .toLowerCase to two lines of code. The first line originally is **char** guess = scanner.next().charAt(0);. I just need to add a .toLowerCase before the equal sign. It would end up looking like this: Character.toLowerCase = scanner.next().charAt(0);. The second line originally is selectedWord = (String) keys[random.nextInt(keys.length)];. Just like the first one, all I need to add is .toLowerCase to this line of code. This is what it would look like after that: selectedWord = (String) keys[random.nextInt(keys.length)]).toLowerCase();.