Intro

For my project, I decided to create a text-based adventure game in the horror genre. The inspiration behind this idea was my interest in game development, and I wanted to explore how to design an engaging experience using only text. While traditional horror games often rely on visuals and sound for scares, I approached this challenge by focusing on creating a survival horror experience with compelling storytelling. The premise of the game revolves around the player being relentlessly pursued by a creature known as The Watcher. Early in the game, the player encounters a building where they must decide whether to enter. Choosing to enter reveals a series of rooms populated by monsters, and the story begins to unfold, explaining the origin of these creatures and the mysteries of the building. Without giving away too much, the game includes combat mechanics and a variety of player choices that influence the outcome. A key aspect of the game is its replayability, if the player loses or dies, they can restart and make different choices, potentially surviving longer or uncovering new parts of the story. This adds an element of strategy and encourages players to explore different paths. Overall, programming this project was both a challenging and rewarding experience. I enjoyed designing the story, implementing gameplay mechanics, and testing the various decision trees. I believe the game is entertaining not only because of its survival horror elements but also because it offers players the opportunity to immerse themselves in a game where you are being chased and are trying to survive.

Algorithm 1 Combat algorithm:

```
// Combat system Algorithm
public static void combat(Scanner scanner, Player player, Enemy enemy) {
    Random random = new Random();
    int playerHealth = player.getHealth(); // Player's health from the Player class
    int enemyHealth = enemy.getHealth();
    int playerAttack, enemyAttack;

    System.out.println("Combat begins!");

    // Loop for the combat turns
    while (playerHealth > 0 && enemyHealth > 0) {
        // Player's turn
        playerAttack = random.nextInt(20) + 10;  // Example attack range for player
        enemyHealth -= playerAttack;
        System.out.println(player.getName() + " attacks " + enemy.getName() + " for " + playerAttack + " damage!");

        // Check if enemy is defeated
        if (enemyHealth <= 0) {
            System.out.println("You defeated the " + enemy.getName() + "!");
            break;
        }

        // Enemy's turn
        enemyAttack = random.nextInt(15) + 5;  // Example attack range for enemy
        playerHealth -= enemyAttack;
        System.out.println(enemy.getName() + " attacks " + player.getName() + " for " + enemyAttack + " damage!");

        // Check if player is defeated
        if (playerHealth <= 0) {
            System.out.println(player.getName() + " has been defeated by the " + enemy.getName() + "!");
            System.out.println("");
            break;
        }

        // Print the current health status
        System.out.println("Your health: " + playerHealth);
        System.out.println(enemy.getName() + "'s health: " + enemyHealth);

        // Pause for the player to continue
        System.out.println("Press Enter to continue...");
        scanner.nextLine();
    }

    if (playerHealth > 0) {
        System.out.println("You survived the battle!");

    } else {
        System.out.println("You fall to the ground, your vision fading to black.");
        System.out.println("Game over...");
                        System.exit(0);


    }
}
```

a. Initialize Variables

The combat begins by initializing the player's and enemy's health points from their respective objects. Two additional variables (playerAttack and enemyAttack) are declared to store random damage values for each turn. A Random object is created to generate these values.

Big O: O(1)

b. Start Combat Loop

A while loop checks if both the player and the enemy have health points greater than zero. If either health value becomes zero or less, the loop ends, determining the victor.

Big O: O(h) where hhh is the health divided by average damage per turn.

c. Player's Turn
Inside the loop, the player's attack value is calculated using random.nextInt() within a specified range (e.g., 10 to 20). The enemy's health is reduced by the player's attack value, and a message is displayed showing the damage dealt.
Big O:O(1)

d. Check Enemy Defeat
After the player's attack, the algorithm checks if the enemy's health has dropped to zero or below. If true, a victory message is displayed, and the loop ends.
Big O: O(1)

e. Enemy's Turn
If the enemy is still alive, its attack value is calculated similarly to the player's, using random.nextInt() within a different range. The player's health is reduced by this attack value, and the damage is displayed.
Big O: O(1)

f. Check Player Defeat
After the enemy's attack, the algorithm checks if the player's health has dropped to zero or below. If true, a game-over message is displayed, and the loop ends.
Big O: O(1)

g. Display Current Health
If both the player and enemy survive their respective turns, their current health is displayed. The algorithm then waits for the player to press Enter to proceed to the next turn.
Big O: O(1)

h. End Combat
Once the loop exits, a final check determines the outcome of the battle. If the player still has health, they survive and can continue the game; otherwise, the game ends.
Big O: O(1)

Total Time Complexity
The loop iterates based on the total health of both participants, making the overall time complexity O(h), where hhh is the total health divided by the average damage dealt per turn.

The purpose of this algorithm is to show the combat system in the game. It plays a crucial role in my game because it determines how the player interacts with the monsters encountered throughout the entirety of the game. Each combat scenario is turn based , with damage values randomized to create this feeling of unpredictability. If a player defeats an enemy they progress the story, but if they are defeated the game ends, and this encourages replayability.

I designed the combat system with the goal of balancing simplicity and engagement. I implemented a loop that alternated turns between the player and the enemy, while using random values to calculate damage.

This algorithm also includes checks to determine the health of both characters after each turn, ensuring the combat ends when one side is defeated.

Algorithm 2 Inventory algorithm:

```java
package FInal_project;

public class Inventory {
    private Item head;  // Start of the linked list

    // Add item to the inventory
    public void addItem(String itemName) {
        Item newItem = new Item(itemName);
        if (head == null) {
            head = newItem;
        } else {
            Item current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newItem;
        }
        System.out.println(itemName + " has been added to your inventory.");
    }

    // Display items in the inventory
    public void displayItems() {
        if (head == null) {
            System.out.println("Your inventory is empty.");
            return;
        }
        System.out.println("Inventory:");
        Item current = head;
        while (current != null) {
            System.out.println("- " + current.name);
            current = current.next;
        }
    }

    // Use an item from the inventory
    public void useItem(String itemName, Player player) {
        if (itemName == itemName.toUpperCase() || itemName.isEmpty()) {
            System.out.println("Invalid item name. Please select a valid item.");
            return;
        }

        if (head == null) {
            System.out.println("Your inventory is empty.");
            return;
        }

        if (head.name.equals(itemName)) {
            System.out.println("You used " + itemName + ".");
            equipOrRemove(itemName, player);  // Equip if weapon, remove from inventory
            head = head.next;  // Remove the item from inventory
            return;
        }

        Item current = head;
        while (current.next != null) {
            if (current.next.name.equals(itemName)) {
                System.out.println("You used " + itemName + ".");
                equipOrRemove(itemName, player);  // Equip if weapon, remove from inventory
                current.next = current.next.next;  // Remove the item from inventory
                return;
            }
            current = current.next;
        }

        System.out.println(itemName + " is in your inventory.");
    }

    // Equip the weapon or remove item from inventory
    private void equipOrRemove(String itemName, Player player) {
        if (itemName.equalsIgnoreCase("Sharp Knife")) {
            Weapon sharpKnife = new Weapon("Sharp Knife", 30);  // Example damage value
            player.equipWeapon(sharpKnife);  // Equip the weapon to the player
        } else if (itemName.equalsIgnoreCase("Sharp Piece of Wood")) {
            Weapon sharpWood = new Weapon("Sharp Piece of Wood", 10);  // Example damage value
            player.equipWeapon(sharpWood);  // Equip the weapon to the player
        }
        }
    }
```

a. Initialize Linked List

The inventory is implemented as a linked list, with a head pointer representing the start of the list. Each Item object contains an itemName and a next reference pointing to the next item in the inventory.

Big O: O(1)

b. Add Item

To add a new item, a new Item node is created. If the inventory is empty (head == null), the new item becomes the head of the list. Otherwise, the algorithm traverses the list to find the last node and appends the new item there.

Big O: O(n), where n is the number of items in the inventory.

c. Display Items
The algorithm traverses the linked list starting from head, printing each item's name until the end of the list is reached. If the inventory is empty (head == null), it outputs a message indicating the inventory is empty.
Big O: O(n), where n is the number of items.

d. Use Item
To use an item, the algorithm searches the linked list for the specified itemName. If the item is found:

- The corresponding equipOrRemove method is called to handle the item's functionality.
- The item is removed from the list by updating the next pointer of the previous node.
  If the item is not found, an appropriate message is displayed.
  Big O: O(n), where n is the number of items in the inventory.

e. Equip or Remove Item
If the item being used is a weapon, it is equipped to the player via the equipWeapon method. The algorithm supports specific items like "Sharp Knife" or "Sharp Piece of Wood," and other items can be added with additional conditions.
Big O: O(1)

Total Time Complexity

- Adding an Item: O(n)
- Displaying Items: O(n)
- Using an Item: O(n)
- Equip or Remove: O(1)

Overall, the time complexity depends on the number of items in the inventory, with traversal operations contributing O(n).

I made this inventory system because it allows players to collect and manage items they find during the game. Thes items being weapons, playing a critical role in survival, as they can be equipped and used during combat. I created this algorithm by making a custom linked list. Each item is represented as a node in the list, and operations like adding, displaying and removing items involve traversing the list. This approach ensured the system was dynamic and could also handle items efficiently without a fixed limit.

Algorithm 3 Navigation algorithm:

```
// Navigation system to move to the next room
// After combat in room1, ask if the player wants to move to room 2
int roomCounter = 1;  // Start from room 1
while (player.getHealth() > 0) {
    System.out.println("Do you want to move to the next room? (Y/N)");
    String moveChoice = scanner.nextLine().toUpperCase();

    if (moveChoice.equals("Y")) {
        // Determine which story method to call based on the roomCounter
        if (roomCounter == 1) {
            // First room transition
            continueStory2(player, enemies[1]);
        } else if (roomCounter == 2) {
            // Second room transition
            continueStory3(player, enemies);


        } else {
            // No more rooms available
            System.out.println("There are no more rooms to explore. The game ends here.");
            break;
        }
        roomCounter++;  // Move to the next room
    } else {
        // If the player decides to stay
        System.out.println("You decide to stay in the room you are currently in, but feel an ominous presence lurking behind you...");
        System.out.println("They finally catch up to you... and they call themselves the \"Watcher\".");
        scanner.nextLine();  // Wait for the player to read the message
        System.out.println("...");
        System.out.println("Trembling in fear, you decide to fight...");
        scanner.nextLine();  // Wait before starting combat
        combat(scanner, player, enemies[0]);


        if(player.getHealth() > 0) {
            scanner.nextLine();
            System.out.println("You escape the bulding, but these mysteries are far from over...");
            scanner.nextLine();

            System.err.println("Thanks for playing!!!!");
            return;
        }
            else {
                 System.out.println("The Watcher pins you to the ground, its glowing red eyes piercing through your soul.");
                 System.out.println("It leans in close and whispers, 'You will make a perfect subject for my next experiment...'");
                 scanner.nextLine();  // Pause for effect
                 System.out.println("Your vision fades to black as the Watcher begins its dreadful work...");
                 System.out.println("GAME OVER.");
                 return;
            }

        }

        return;

    }
}
```

Steps for the Navigation System Algorithm

a. Initialize Room Counter
The roomCounter variable is initialized to track the player's current room. Starting from room 1, this counter increments as the player progresses to the next room.
Big O: O(1)

b. Navigation Loop
The loop continues as long as the player's health is greater than zero. The player is repeatedly prompted to decide whether to move to the next room or stay.
Big O: O(n), where n is the number of rooms.

c. Player Chooses to Move
If the player decides to move to the next room:

- The algorithm checks the roomCounter to determine which room transition logic to execute.
- If the player is in the last available room, the game ends with a message. Otherwise, the roomCounter increments to reflect the new room.
  Big O: O(1)

d. Player Chooses to Stay
If the player decides to stay in the current room:

- A description of the impending danger is displayed.
- The player is forced into combat with "The Watcher."
- If the player survives the combat, they escape the building; otherwise, the game ends with a death sequence.
  Big O: O(1)

e. Combat Execution
If the player encounters "The Watcher," the combat algorithm is invoked. This ensures a turn-based fight where the outcome determines whether the player survives or the game ends.
Big O: O(h), where hhh is the health divided by average damage per turn.

Total Time Complexity.

The navigation loop iterates through the number of rooms, and within each room, combat might occur. This makes the total time complexity:

- O(n+h), where nis the number of rooms and hhh is the health divided by the average damage dealt per turn.

The navigation system plays a crucial role in progressing the story and gameplay. It allows the player to decide whether to move forward into a new room or stay in their current location. Each decision carries weight, as moving forward advances the story and presents new challenges, while staying in a room leads to a forced encounter with "The Watcher." This mechanic builds suspense and ensures players carefully consider their choices. It also ties directly to the game's survival horror theme, where every decision could lead to unexpected outcomes. When the player decides to move forward, the system transitions to the next room, where unique challenges, monsters, or narrative developments await. The game uses the room counter to track the player’s position, and room-specific methods are called to handle the logic for each space. If the player opts to stay, they face immediate danger, as "The Watcher" forces them into combat. This creates a sense of urgency, encouraging players to make quick but thoughtful decisions as they navigate the game. I designed the navigation system with simplicity and flexibility in mind. At its core, it uses a roomCounter variable to track the player's current position in the game. This numeric tracker determines which room-specific logic or story method to call, ensuring smooth

progression between rooms. A while loop keeps the game running as long as the player's health remains above zero, continuously prompting the player to decide whether to move to the next room or stay. If the player chooses to move forward, the roomCounter increments, and the corresponding room transition logic is executed. Modular methods handle the unique content of each room, making it easy to expand the game by adding new rooms or encounters. If the player stays in the room, the system transitions into a combat sequence with "The Watcher." This integration of navigation and combat mechanics makes the game more cohesive and heightens the stakes for player decisions.

Data Structures

For the data structures I used four, those being arrays, a linked list, strings and custom objects.

I used arrays for fixed-size collections, such as storing enemies in a specific sequence. The array enabled straightforward access to each enemy based on the room number or game sequence. For instance, when transitioning to a new room, the relevant enemy could be retrieved using the room counter as the index.

A linked list allows flexible growth and efficient insertion or deletion of elements without requiring resizing, unlike arrays. This was important since the number of items in the inventory could vary throughout the game. As stated before it was implemented in the inventory class, where each item object represented a node with a name and a reference to the next node. Also items could be dynamically added to the inventory using the additem method, and the linked list was traversed when displaying or using items. This approach ensured the inventory could grow as needed.

I used strings because they were used extensively for textual input, output, and decision-making. They allowed player commands, displaying story elements, and managing descriptions. Strings were used to capture and process player choices (e.g., "Y" or "N" for navigation decisions). They were also central to the storytelling elements, providing descriptions for rooms, items, and enemies

The last data structure I used were custom objects and using classes such as Player, Enemy, and Rooms allowed me to encapsulate related data and methods into meaningful entities. This made the code easier to manage and extend. Each custom object represented a key component of the game, those being players which tracked the players health and equipped weapons, enemies which represented the monsters and properties like name and health, and also rooms which provided descriptions and story logic for each room.

Opportunity Encountered:

I had a couple of opportunities encountered that were more with the story specifically where these options to be made so i just decided to make a switch statement for that. I also decided to just make separate classes that are just called continue story, continue story2, and continue story3 where after each room encounter it would continue the story. Another opportunity I found was making random combat to make sure it wasn't always the player losing. What I mean by this is that the watcher is way stronger than the player and always dies. I realized this isn't fun because

it's just random luck. But then i made it more balanced, it is still a little rng based but there are more likely chances of winning. Overall there weren't many opportunities that came my way but there were many errors.

Errors encounters:

So stated previously in my previous paragraph I did say I encountered a few problems that being i kept on infinite looping my navigation system i had to fix this because there were only three rooms to go to this is why i added the room counter. Another Problem i encountered was sometimes my story didn't continue and I was stuck on this for hours. The way I fixed this was making sure after combat they go to the next room i wanted the player to go, i had to make different static voids to solve this. Another problem was During navigation and combat, player input was not being handled consistently. For example, typing "y" instead of "Y" or "stab" instead of "STAB" caused the game to misinterpret the input and respond with an error. It solved this by adding a toUpperCase() or toLowerCase() normalization to all user inputs before comparing them, ensuring case-insensitive handling. Overall encountering and resolving these errors was a significant part of the development process. Each issue provided an opportunity to refine the game, improve its robustness, and ensure a better player experience. By systematically debugging and testing, I gained a deeper understanding of my code and strengthened its overall design.

Overall/what i would make better:

Overall working on this project was pretty fun and will continue working on more projects like this so i can get better at coding and at game development in general. But there are some improvements I would like to make if I were to work/recreate this game. Those are better combat so that you don't have to rely on rng and add like an attack or defend option, more story/lore , improved navigation system with more rooms, difficulty levels to make the enemies stronger, and much more. These additions would make the game more immersive, strategic, and engaging while appealing to a broader audience. By focusing on expanding the narrative, enhancing gameplay mechanics, and also adding visual and audio elements, the next version would provide a richer and more memorable experience for players.

Overall hope you enjoy the project and I will definitely improve upon it more.