**Self-Assessment**

**Conceptual coverage**

I believe I’ve demonstrated all programming concepts we’ve gone through in Matlab including but not limited to the basic variables, user inputs, and writing styles; and I've also gone above and beyond by integrating these concepts into things such as algorithms and graphical interface.

***Justification***

*Loops*

In the **Main.m** file on line 97, I’ve implemented while loops to repeatedly take user’s input and implemented a *break*feature that safely exits the loop once a level is completed or the user no longer wants to continue the game. For Loops were also used as a part of the maze generation for important aspects like map template generation and the difficulty algorithm.

*Vector*

In **Main.m**, on line 76 and 108, I’ve used vectors to act as an error-handling where only a selected keyboard input will be parsed through the rest of the program.

*Matrices*

Matrices are an important aspect of my code, since it is required for the main map and graphical interface of the game. You can see a map template being generated using Matrices in the file **initializeNewMap.m** inside the *mapArrayTemplate()* function.

*Conditional Statements*

Conditional statements are crucial for most projects, and my program is no exception. I’ve applied the conditional statements to create map boundaries, player boundaries, and player requests. To reduce the use of *if* statements, switch cases were also used when available to improve the readability of the program. An example of switch cases being used can be seen in the **movementHandling.m** file on line 27, where the program determines movement based on the player’s input.

*Functions*

Functions are important to improve readability of a program and also make it modular. Since my game is an infinite-level type of game, it must reliably reproduce the same expected result no matter the occurrence. To demonstrate the importance of functions, I've decided to take a recursive approach in my maze generation. This function found in **InitializeNewMap.m** called *recursiveBacktracking()* calls on itself multiple times to show that re-using the same code in this manner is more viable than having hundreds of the same line of code.

**Value-add to Project**

I have had a substantial contribution to the overall result of my project. Using and referencing short guides from the internet including those from the official Matlab website, I was able to gather principal information about each source to piece together my Project. I’ve also grown a fond connection to the project, and decided to go above and beyond in improving player experience by adding neat features such as *Music*and *Shop interface*.

***Justification***

*Shop*

The shop interface was always a neat idea I wanted to add, where it’ll create a purpose for reaching those higher levels. In the main game, it could always be toggled using ‘Q’ and a working graphical interface could be used to unlock special abilities which could break the game. This shop functionality is found in the **shopFunc.m** file.

*Movement*

Originally, the ‘input()’ function was the only way I knew how to take user input. Obviously this would be really annoying having to click and type in the terminal, but I decided to look on the internet and found a keyboard input method. Overall, I would say moving with a keyboard input is better for the player’s experience compared to a terminal input.

*Interface*

A terminal-based game would appear too boring, so I’ve used the already learnt ‘imshow()’ function with matrices to create a neat graphical interface. With this knowledge, it was already clear how I could continuously use the ‘imshow()’ function to show movement in a game. I’ve also found built-in matlab gui functions such as *questdlg* and *uibutton* to really maximize the user’s experience by making this the default way of taking input, as it's easy and clear to use.

*Music*

Using the *sound()* and *audioplayer()* function, sound effects and background music were added to fill in the silence while you solve a complex puzzle.

**Incremental Development**

I’ve had to explore all the functionality and test cases to find the best way to write and implement my code, and the *Development Process* folder that comes with this submission is evidence of that. Comments were also added to explain necessary aspects of the code.

***Justification***

Every file in the folders and subfolders of the Development *Process* is broken down into what I believe are the 4 main aspects of my game. These include the *Sound* testing, *GUI* testing (aka Box testing), *Input* testing and *Map and Maze Generation*.

For *Sound* Testing, there is only 1 file since there’s no ‘thinking’ involved when applying music, and this file just tests to see how music would sound at different levels, or how to interact and pause/play these sounds. In the end, I simply applied this to make the **soundPlayer.m** function that plays a requested sound.

For *GUI* testing, it was a bit harder, as I had to go through the documentation and how to manipulate the output when a checkbox is ticked, or a button is clicked. Since I knew a bit of how GUI works already as it is fairly similar to *tkinter*, a GUI library for Python, I was able to manipulate callback functions and also design quickly.

For *Input* testing, I found a simple way to take user input simply using the *waitforbuttonpress* and *double(get(gcf, 'CurrentCharacter'))* command. Then, I played around with this through the **keyPrint.m** and **cheatCodeTest.m** files, to later have enough confidence to make a movement game called **movementTesting\_1** and **movementTesting\_2**.

In *Map and Maze Generation,* before planning, I knew I wanted to make an infinite-level type of game with unique levels to complement the movement handling I've learned in the Input testing. The first design was a botwalk as shown in **mapGeneration\_Botwalk.m**. This was when I decided to search the web for an efficient algorithm, and found that *recursive backtrack* was the simplest and easiest to implement, since its main purpose is to reuse the same code. After implementing this correctly (**backtrackingAlgorithm.m**), I developed my own algorithm to spawn players and flags to avoid any ‘free’ levels (**playerFlagSpawn\_math.m**). This new method later became the default way of creating a new level, and was implemented in the main function.

**Testing**

As I fix code as I write, testing with test files has still not yet been my default style, so I forget to do it then and there. However I believe the test cases I’ve provided are enough, to have the game be playable, as bugs look *rare* while reading the code*.*

Most of the bugs found in testing are in the *Development Process* folder, and the rest of my program passes most of the edge cases I could think of.

**Comment and Style**

I believe I've added comments when necessary, and all variables, functions, constants are named properly following the recommended procedure for easy readability.