RANDOM MAZE GAME

Brampton Manor Academy | 13202

Donatas Jacinavicius

Table of Contents

Analysis

Project background

The aim of my project is to use a variety of different maze generation algorithms in python to create a game, which becomes increasingly more difficult and challenging as the player progresses. The player is placed into a randomly generated maze, they must find a path to the randomly placed treasure inside the maze, and they must do this in the allotted time. The time given to complete the task will be based on the distance of the shortest path to the treasure from the player's location, some extra time will be given to the player. Once the player has reached the treasure, a new maze will be generated, and the treasure placed in another random location. As the player continues to complete mazes the game will become harder, this will be achieved with 4 main techniques:

1. Maze generation algorithm will change to one that creates more challenging mazes
2. Extra time given to find the treasure will be reduced
3. Density of dead ends will be increased
4. Size of mazes will be increased

Current solution

There are many maze games available online. A classic maze game is Pac-man; however, this does not include the random generation of mazes, and merely a predetermined maze. Additionally, I have yet to find a random maze game that uses braiding to adjust the difficulty of the game, and also makes it dynamic to the player's performance. Furthermore, many of these maze games use one algorithm, which means that the gameplay may become stale, whereas my solution will include the use of multiple maze algorithms to ensure the gameplay is engaging even after many attempts of the game.

Research

The main research that I conducted was finding a selection of maze algorithms, some that produce easy mazes, that are heavily biased, and some that produce hard mazes that are almost purely random. I discovered the extremely helpful 'Mazes for Programmers' by Jamis Buck. This book outlines a vast array of maze algorithms, with distinct biases that could be used for a specific purpose.

Research 1.0 Maze Generation –

There are a variety of different maze generation algorithms, each with certain biases that they introduce. It was necessary for me to find a selection of maze generation algorithms that range from easy to hard in terms of how difficult they are to solve, in order to give a gradual increase in difficulty to the player as they progress through the game.

Research 1.1 Binary Tree Algorithm –

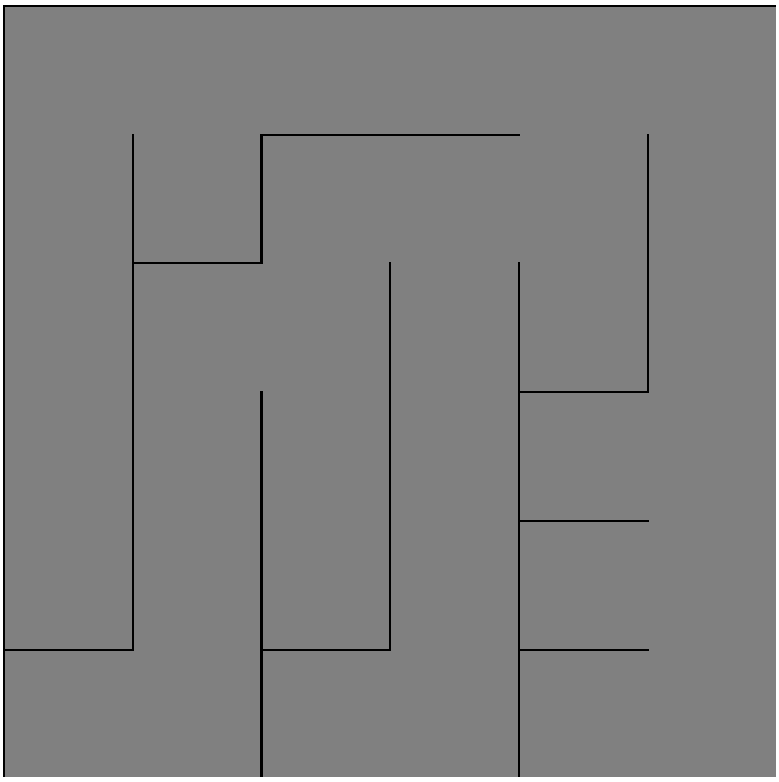


Figure 1 – Binary Tree generated example

Above is an output of my implementation of the Binary Tree algorithm. This algorithm generates a maze by selecting randomly between two directions to carve. In my prototype, these two directions are north and east. As a result, the maze will have an unbroken row in the north, and an unbroken column in the east, this is because on the eastern boundary the algorithm must carve north, and on the northern boundary it must carve east. Additionally, the maze generation must begin from one of the four corners, although the same bias will occur regardless of which corner the maze generation begins in.

A clear bottom-left to top-right bias can be seen in my maze. For example, reaching the top right corner can be achieved by repeatedly moving north and east.

Research 1.2 Sidewinder –

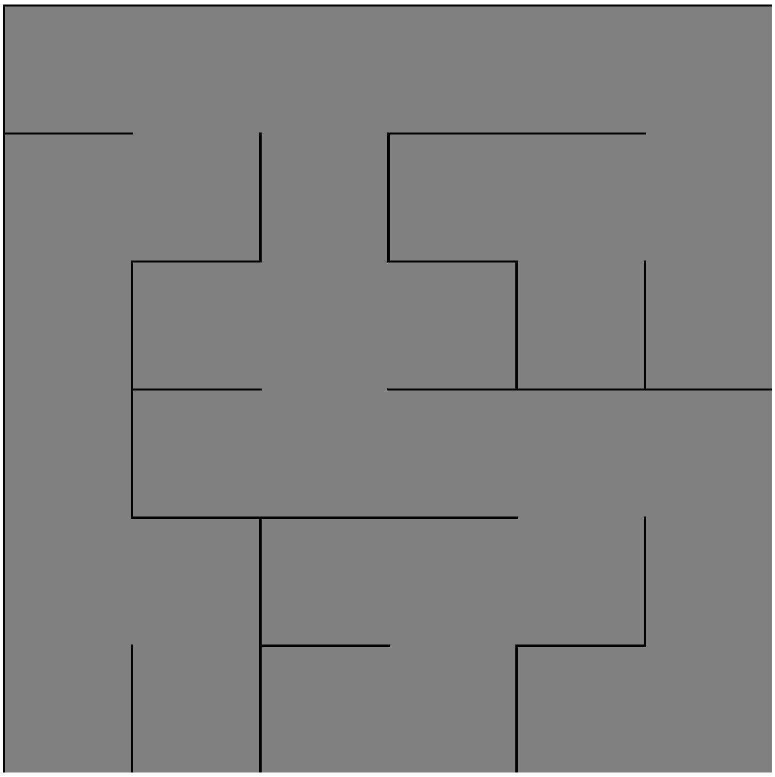


Figure 2 – Sidewinder generated example

Above is an output of my implementation of the Sidewinder algorithm. This algorithm also selects from two directions to carve but does not use this in the same way that Binary Tree does. Starting from the bottom-left block, a random choice is made between carving north and east, if the outcome is to carve east, then the eastern wall of the block is erased, and this block is added to the 'run'. This is repeated until the decision to carve north is reached, then a random block from the current run is selected, and the algorithm carves north from that block into the next row.

Unlike Binary Tree, Sidewinder does not create an unbroken column, it only creates an unbroken row, this being the northern row in my implementation. A bottom to top bias can be seen in the maze, e.g. going from the bottom row to the top row is quite easy, as each row is a horizontal run (or multiple runs) of cells with one of those cells having a passage north.

Research 1.3 Aldous-Broder –

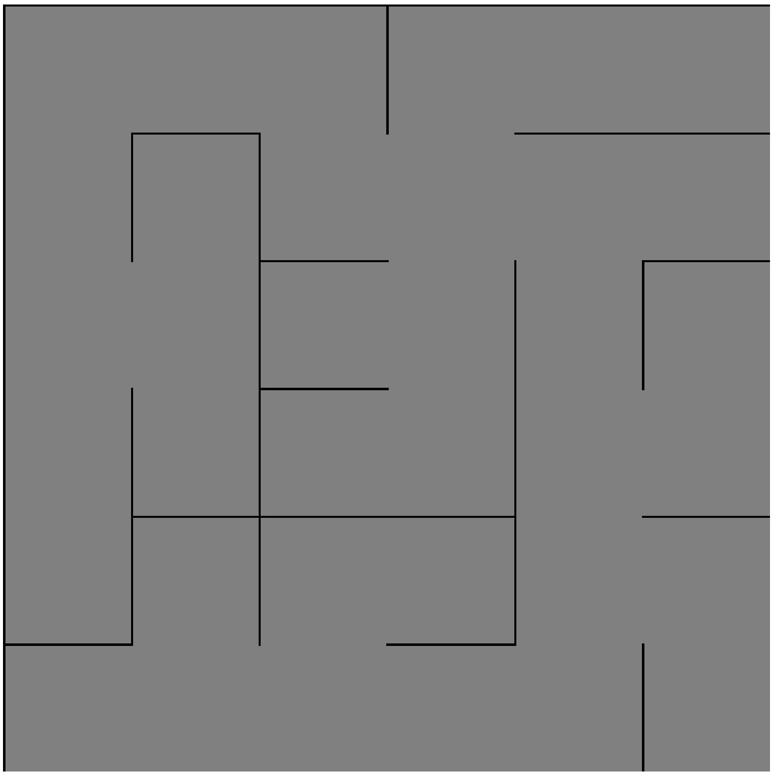


Figure 3 – Aldous-Broder generated example

Above is my implementation of the Aldous-Broder algorithm. This algorithm carves randomly, i.e. the algorithm selects from four different directions to carve. Each time a block is carved from, it is marked as visited. If the block to be carved to next is already a block that has been visited previously, instead the algorithm continues to this block but does not carve to it, a new passage is not made. The algorithm continues this process until all blocks have been visited.

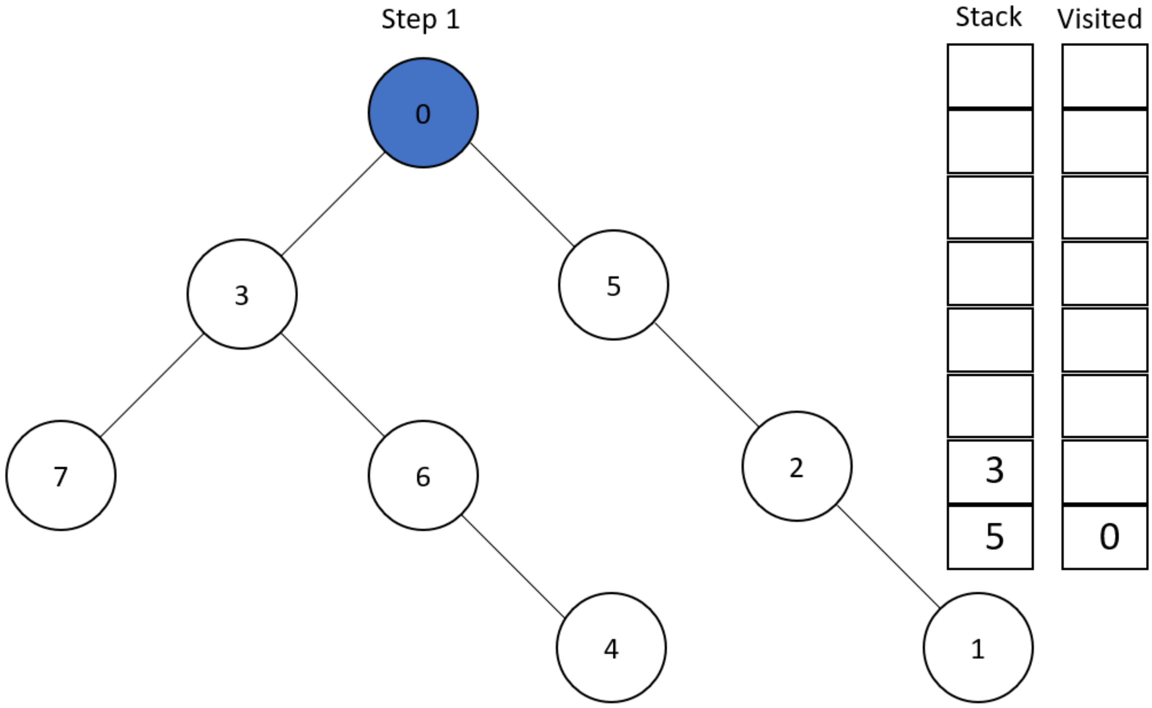
Due to the random nature with which the maze is generated, it has no bias, it is a uniform spanning tree unlike the previous two mazes generated with different algorithms. A consequence of this method is that generation takes far longer than the previous two algorithms. This is because each block may be visited multiple times before all blocks have been visited and the algorithm terminates. This begins to become a problem if numerous mazes need to be generated, or if the grid size of the mazes is large. Generating a 100x100 maze using my implementation took 0.834 seconds, whereas using Binary Tree for the same maze size took 0.014 seconds.

Research 2.0 Searching Algorithms –

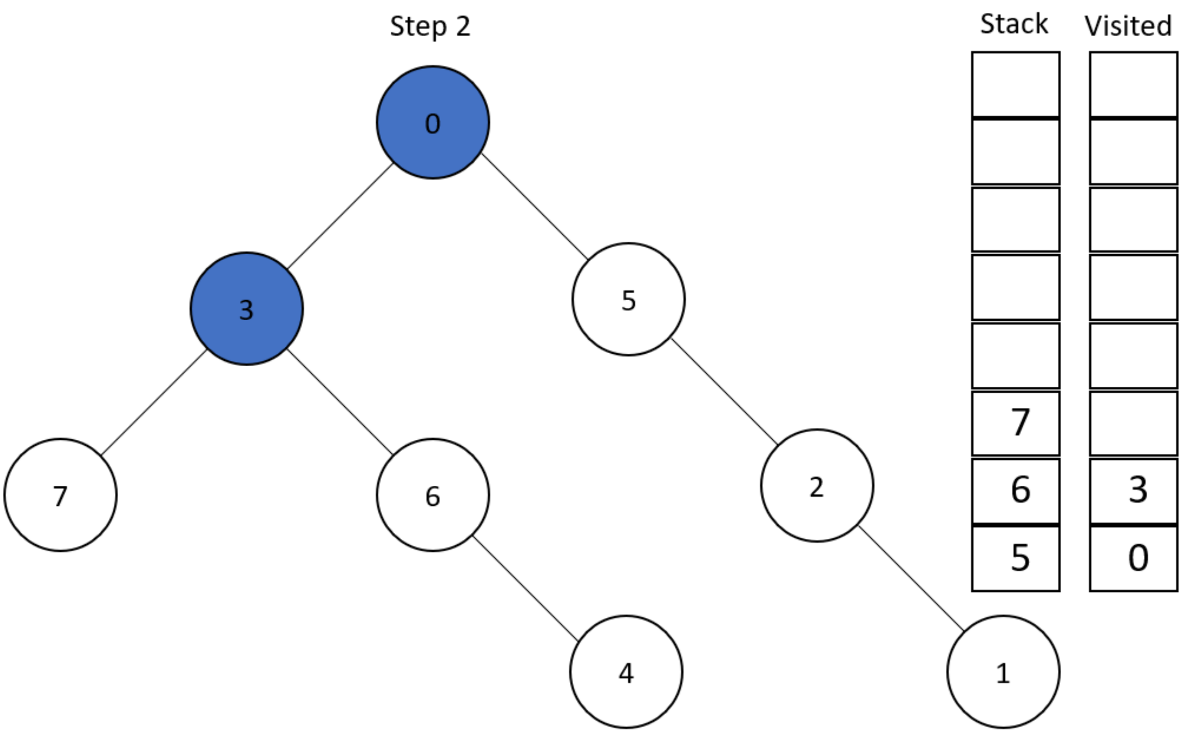
It was necessary to familiarise myself with path finding algorithms and decide on using one that is most suitable to solving mazes. From my Computer Science course I already have knowledge about the Depth-first and Breadth-first searches, as well as Dijkstra's, although further research has led me to understand these algorithms (especially Dijkstra's) in greater detail. All of these searches are algorithms for traversing tree or graph data structures, and therefore can be applied to mazes. This is because mazes are essentially trees, with each block in the maze being a node and each passage between adjacent blocks being an edge.

Research 2.1 Depth-first search –

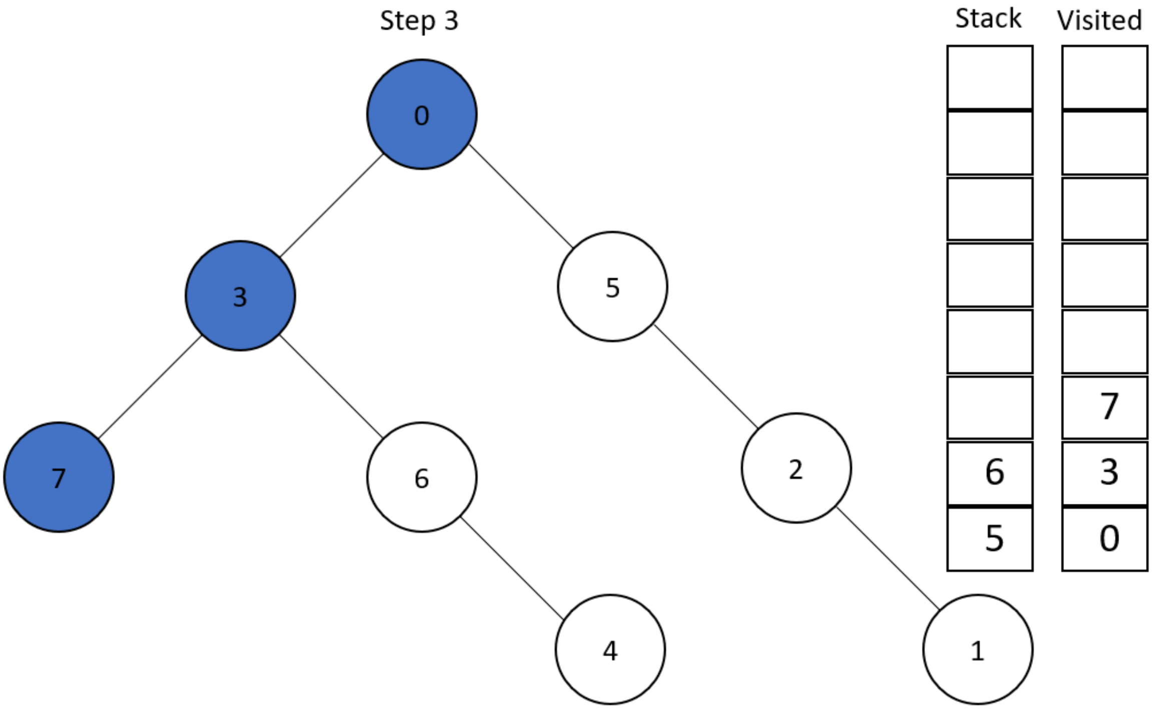
The Depth-first search starts at a root node and traverses along a branch until it can no longer continue down the branch. The algorithm then traverses back until it finds an unexplored branch, and then repeats the process. I have provided an illustration of how this algorithm traverses a graph using a stack.



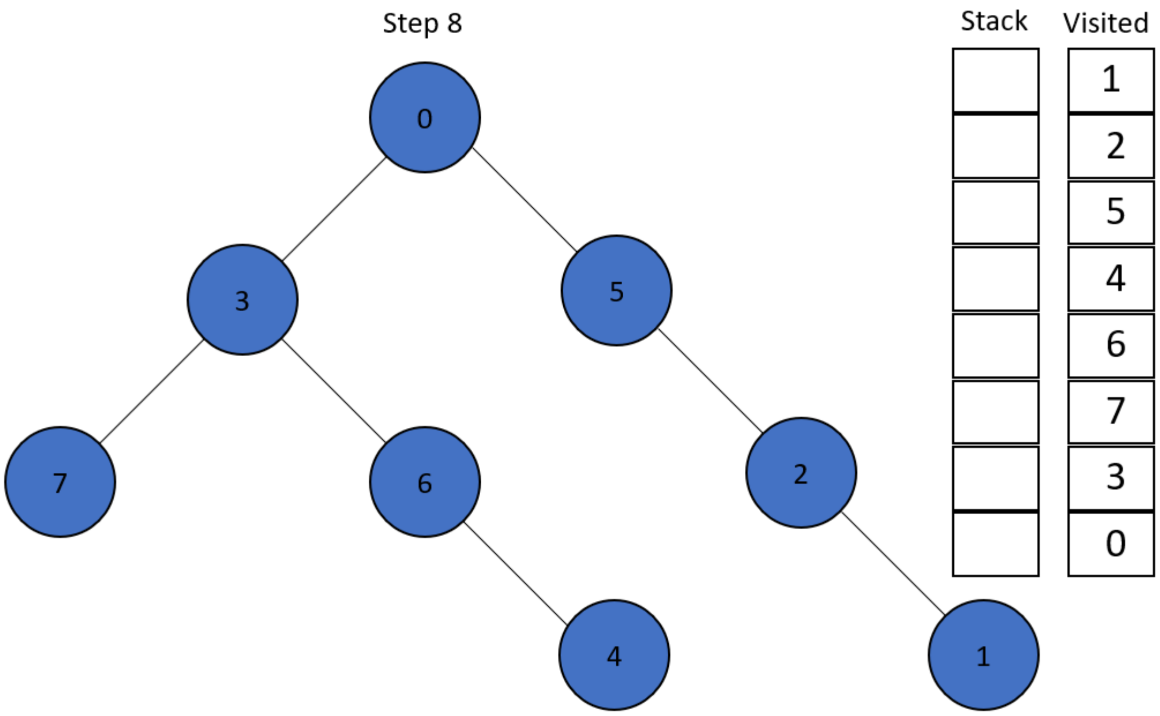
1. The algorithm begins on node 0 and adds that to the visited list. Any adjacent nodes to node 0 are added to the stack.


Step 2

2. The next step is to move to the node at the top of the stack and pop it out of the stack. In this case that node is 3. All adjacent nodes to 3 are pushed into the stack and node 3 is added to visited.


Step 3

3. The previous step is repeated, 7 is popped out of the stack and added to visited. There are no adjacent nodes and therefore nothing is pushed into the stack. The algorithm will continue this recursive action until the stack is empty, unless a goal node is specified, e.g. terminate upon reaching node 6.



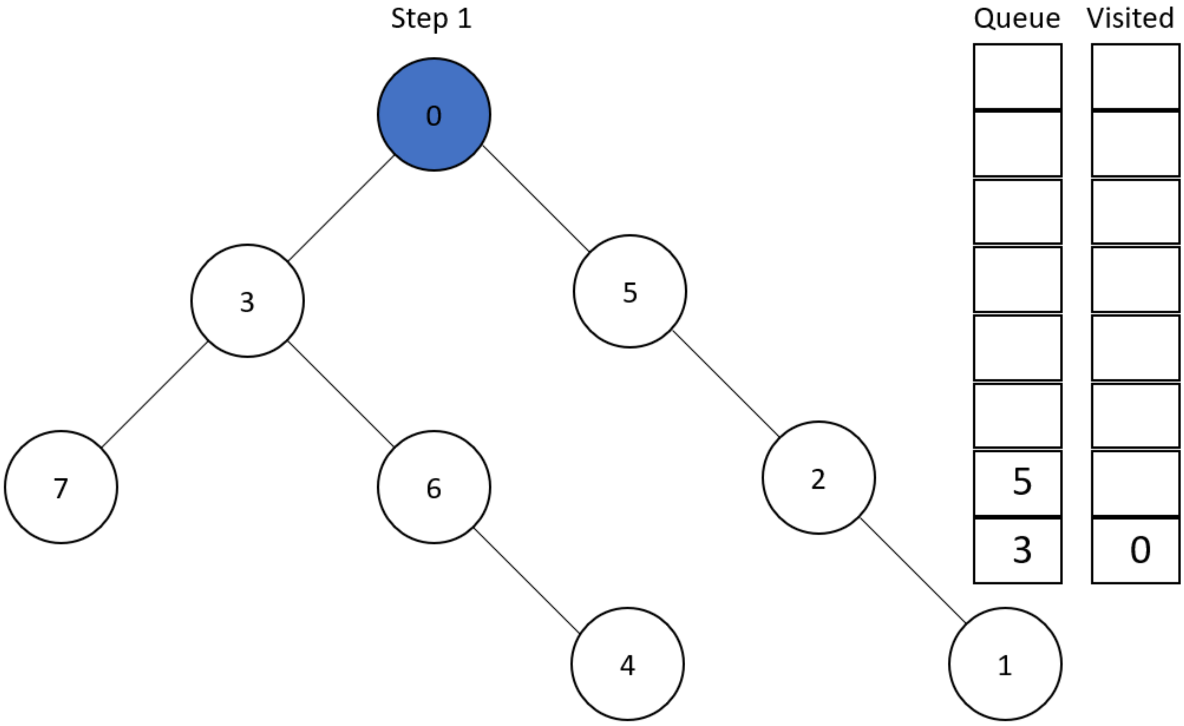
4. At step 8 this particular tree is fully searched. Reading visited from bottom to top will provide the order of the traversal.

With this method the guaranteed shortest path between two nodes can be found, but this is only the case if the graph contains no cycles. This is true for perfect mazes, as they contain no cycles, however, I plan to use braiding in this project which is the removal of dead ends in mazes, this results in mazes with cycles and therefore DFS may not be the best searching algorithm. The advantage is that it requires the less memory than alternatives, as the algorithm only needs to store a stack of nodes from the root node to the current node.

Research 2.2 Breadth-first search –

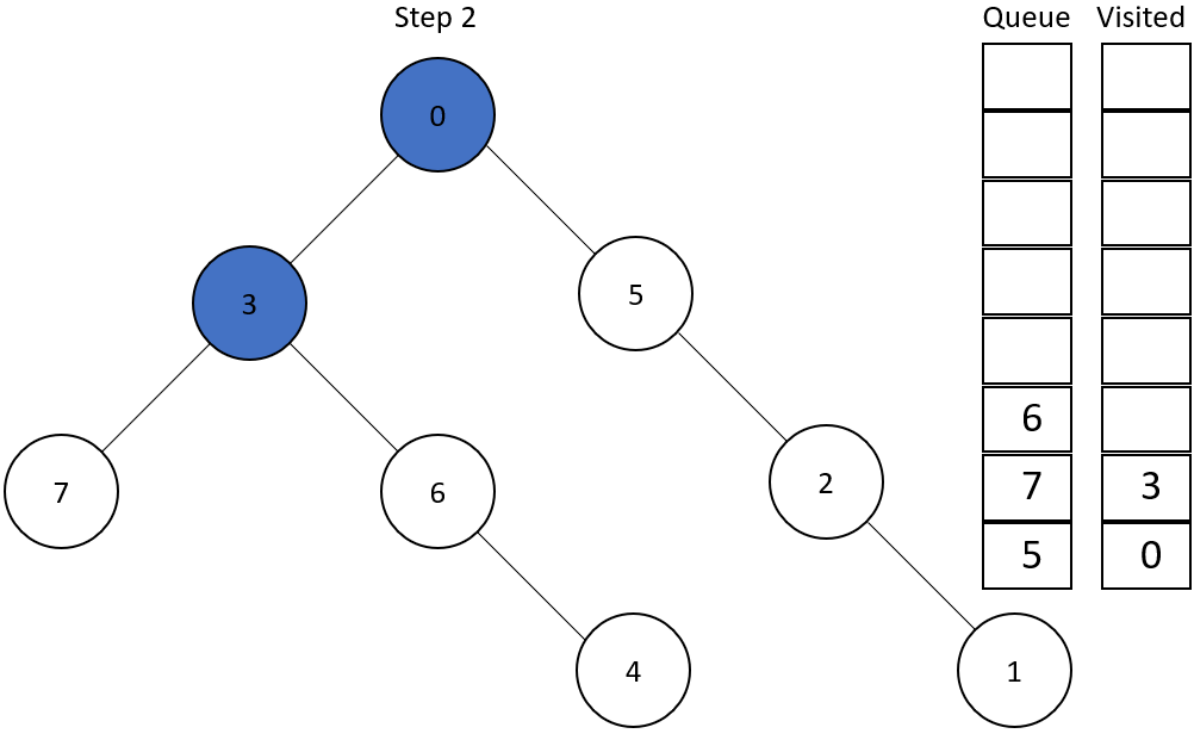
Breadth-first search is similar to Depth-first search with the key difference being that BFS uses a queue instead of a stack. This difference in data structure leads to the traversal searching through in level order, i.e. searching the highest layer first, then the second layer before moving on to the third and so on. I have provided an illustration of BFS below.

Step 1

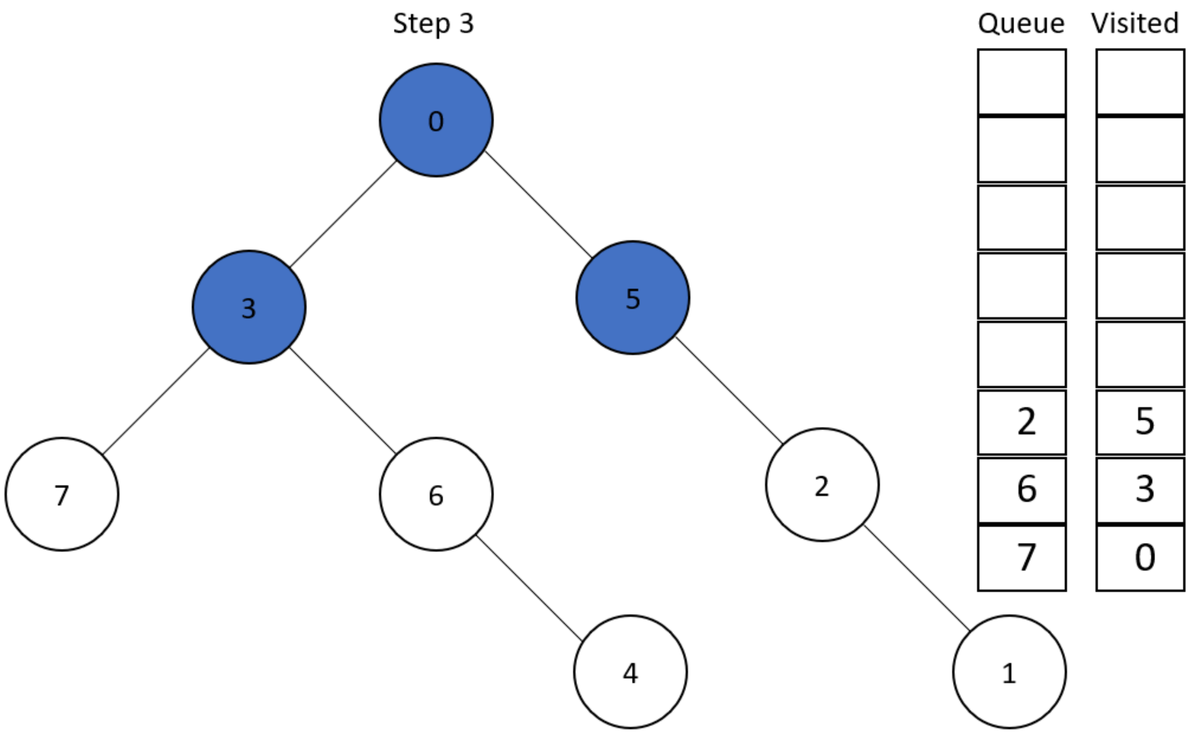


1. The root node 0 is searched, and all adjacent nodes are added to the queue

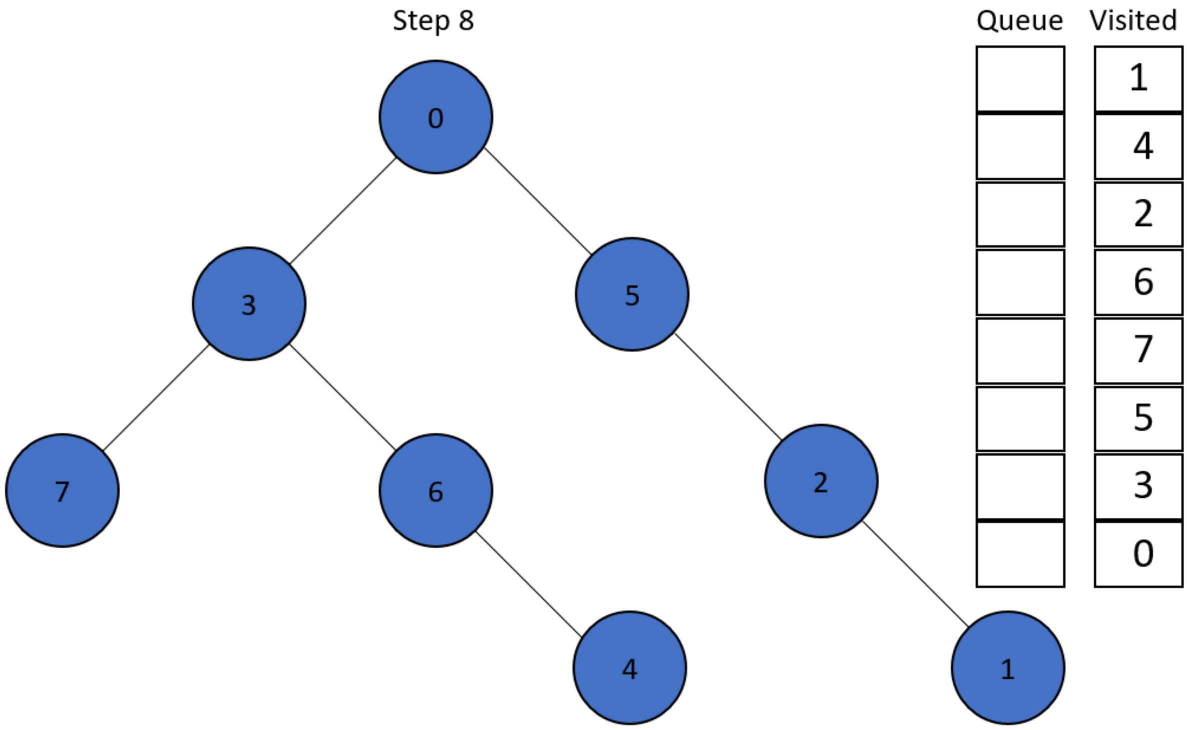
Step 2



2. The first node in the queue is removed from the queue and visited, adjacent nodes to node 3 are added to the queue



3. Step 2 is repeated, in this case 5 is taken from the queue and visited, with its adjacent node 2 being added to the queue.



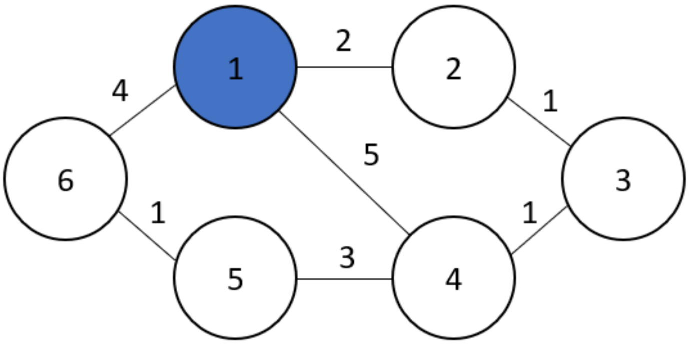
4. At step 8 this tree traversal is complete, the visited list read from bottom to top shows the order of traversal.

From these two illustrations, it can be seen that DFS and BFS traverse a graph at the same speed, both requiring 8 steps to finish the search, with each step being a removal of an item from a stack/queue, and the addition of the adjacent nodes to the stack/queue. Either one may be faster than the other based on the location of the item being searched for. For example, if the item being searched for is close to the root, BFS is faster, and if the item is close to the leaves, DFS is faster. This is true if the search terminates once the item is found. Additionally, if a graph is wider than it is tall, i.e. the amount of nodes on a level grows faster than the amount of levels, BFS will require more memory, as the max queue size is based on the widest layer, while the max stack size is based on the number of layers. The opposite is true for DFS, requiring more memory if a graph is taller than it is wide.

Research 2.3 Dijkstra's -

Dijkstra's algorithm creates a tree of shortest paths from the root node, to all other nodes in the graph. Dijkstra's algorithm is useful because it is able to find the shortest path on weighted graphs, unlike the BFS and DFS. Below I have provided an illustration of the algorithm. Initially, the distance of all nodes in the graph is set as infinity, aside from the root node (1) which begins at distance 0. All nodes begin as unvisited.

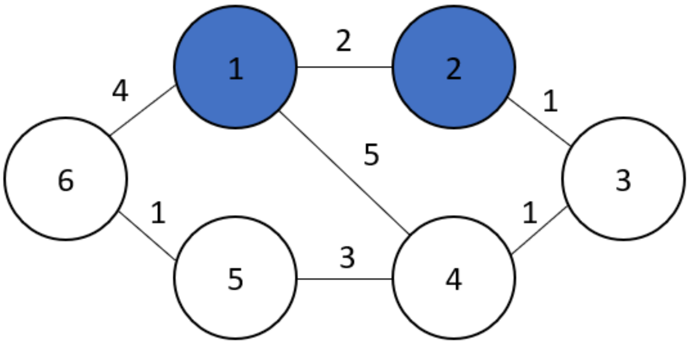
Step 1



Visited	Node	Distance
1	1	0
0	2	0+2(<∞)
0	3	∞
0	4	0+5(<∞)
0	5	∞
0	6	0+4(<∞)

1. Node 1 is marked as visited. For each adjacent node to the current node, the distance of the current node to the root node is added to the distance between the adjacent node and current node. If this sum is less than the current distance marked for the adjacent node, the distance is set as the sum. The algorithm then visits the node with the lowest distance, this being node 2 and repeats step 1.

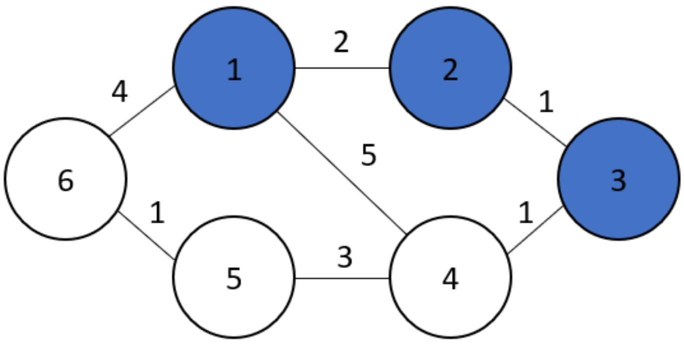
Step 2



Visited	Node	Distance
1	1	0
1	2	2
0	3	2+1(<∞)
0	4	5
0	5	∞
0	6	4

2. Step 1 is repeated for node 2

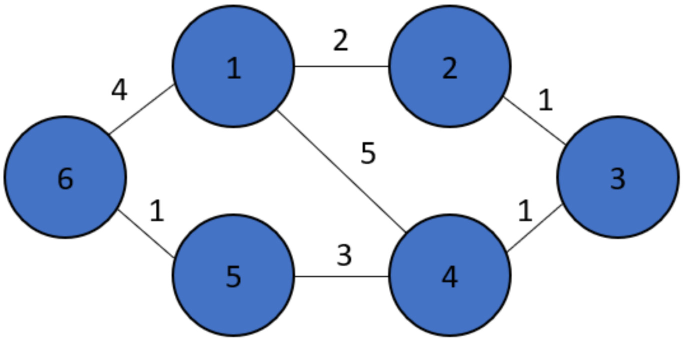
Step 3



Visited	Node	Distance
1	1	0
1	2	2
1	3	3
0	4	3+1(<5)
0	5	∞
0	6	4

3. Step 1 is repeated for node 3, notice that the distance of 5 to node 4 is overwritten with a distance of 4, as the distance between node 3 and the root node, added to the distance between node 3 and node 4 is less than the current distance from node 4 to the root node.

Step 6



Visited	Node	Distance
1	1	0
1	2	2
1	3	3
1	4	4
1	5	5
1	6	4

4. At step 6 the algorithm terminates, having produced the shortest distance of all nodes to the root node.

While Dijkstra’s algorithm takes longer to traverse a graph than DFS and BFS, it is able to calculate the distance between any two nodes while also considering the weights of connections between nodes. BFS can be slightly altered to find the shortest distance by incrementing a ‘step’ variable after each level, terminating the search once the required item is found. The value of ‘step’ will be the length of the shortest path. However, this method does not allow for the calculation of the shortest path in a weighted graph, which is a requirement as blocks that modify player speed must be taken into account when calculating distance, since they will ultimately affect the time required to reach the goal.

Research 3.1 Braiding –

Braiding was another technique I had learnt from Buck's book, it is essentially the joining of dead ends, which results in fewer dead ends in the maze. A maze can be 'braided', i.e. it can have a certain proportion of its dead ends removed. A higher proportion would result in more river-like, winding mazes. A lower proportion would result in mazes that are significantly harder to solve, this is because the algorithms I have chosen to use all produce perfect mazes, so if no dead ends are removed, there is only one path for the player to take to reach the treasure. Below I have provided a comparison between a non-braided and braided maze using my implementation of the braiding algorithm

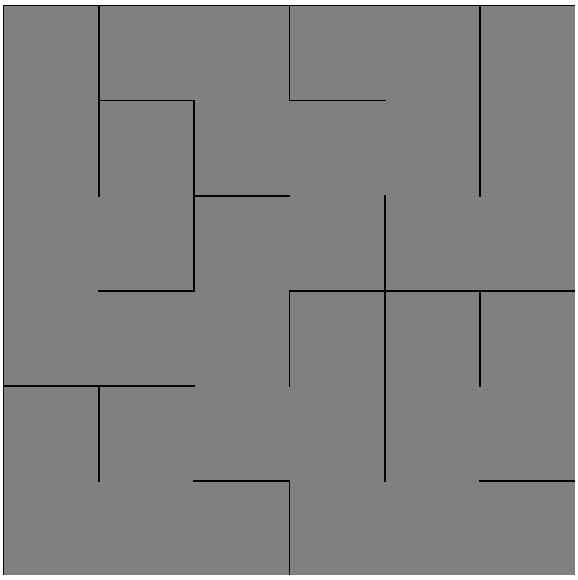


Figure 1 - Aldous-Broder non-braided

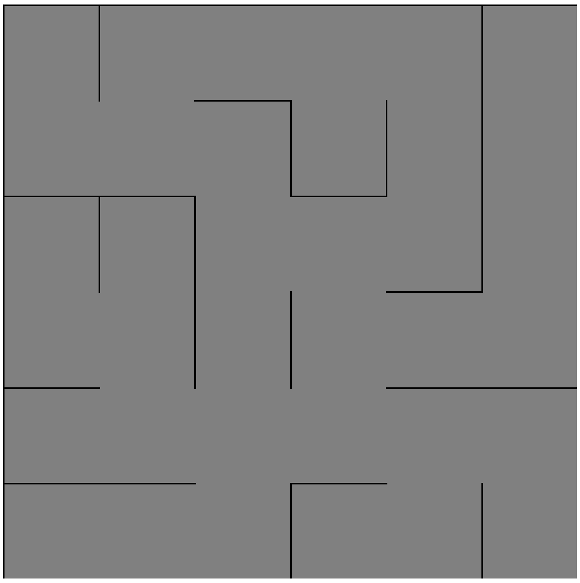


Figure 2 - Aldous-Broder 50% braided

As discussed previously, all maze algorithms I am using produce perfect mazes, which are graphs with no cycles, i.e. there exists only one path between two blocks in the maze. This can be seen demonstrated in Figure 1. Figure 2 shows a maze also generated with the Aldous-Broder algorithm, but every second block has had its dead end removed (if it had one) by the braiding algorithm, which results in approximately half the dead ends being removed in the maze. This results in some pairs of blocks having two or more paths between them. Through my own testing with this algorithm, mazes that are non-braided are much harder to solve, especially when the maze size increases far beyond what is shown in Figures 1 and 2.

Target audience

The game I am creating is mainly designed for recreational purposes, to try and beat the high score of other players. More specifically it is aimed at people interested in maze algorithms, as the game dynamically changes between different maze generation algorithms and modifies the effect of braiding based on player performance.

Methods and sources

The foundation of this game is based upon maze algorithms. I will use three maze algorithms, ordered from easiest to hardest, this includes:

1. Binary Tree
2. Sidewinder
3. Aldous-Broder

The Binary Tree algorithm produces a heavy bias, making it the setting for the easiest difficulty in this game, the Sidewinder algorithm reduces that bias by a half, from a bottom-left to top-right diagonal bias to a bottom to top vertical bias, this will be used for medium difficulty. The hardest difficulty will use the Aldous-Broder algorithm, which has no bias and is completely random; it is however slightly more computationally expensive than the previous two algorithms, as the randomness of the algorithm causes it to not be fully efficient - some blocks in generation may be visited more than once. To add more complexity to the game, upon generation blocks will have a small chance to either modify the player to move faster when standing on them, or slower. These blocks will be referred to as ‘modified’, and their generation will be accomplished with a ‘Modifier’ class.

Braiding will also be used as a measure of difficulty, a more braided maze will be an easier one, and vice versa. Dijkstra’s algorithm will be used to calculate the shortest distance between the treasure and the player in each stage of the game and blocks that alter player speed will be weighted such that Dijkstra’s algorithm takes into account the different amount of time required to traverse them. The time calculated with this method will have an extra variable ‘Ease’ added to it. In this case, Ease is a measure of how much extra time the player is given, and therefore it is a measure of difficulty, which will be changed as the player begins to perform better or worse.

The generation of mazes will include the use of object-oriented programming. The maze will consist of a grid of blocks, each block will be an object of the ‘Block’ class. Initially I expect the class will include the following attributes:

1. North – Tracks the state of the northern wall of the block (1 exists, 0 does not)
2. East – Tracks the state of the eastern wall of the block (1 exists, 0 does not)
3. South – Tracks the state of the southern wall of the block (1 exists, 0 does not)
4. West – Tracks the state of the western wall of the block (1 exists, 0 does not)
5. Visited – Required for Aldous-Broder to keep track of which blocks have already been visited, as a different interaction must occur based on if the block is visited or unvisited.
6. Distance – Used by Dijkstra’s algorithm to find the shortest path from one point in the maze (player) to another point in the maze (treasure). Specifically, this represents the distance from this node to the root node.
7. SpeedMultiplier – Used to determine whether the block spawns a modifier, and if so what type, that changes player speed when the player traverses through the block

The braiding algorithm may be used after the maze has been generated, in order to remove a proportion of dead ends present in the maze.

Through my own testing, I have found that larger mazes are significantly harder to solve than smaller mazes, and so grid size will also be a measure of difficulty.

Objectives

1. Player prompted with menu, with the following options:
    1.1. Start a new game
    1.2. View top 10 high scores
    1.3. Exit the game
2. When player chooses to start a new game execute the following operations:
    2.1. Set starting variables for game:
        2.1.1. Grid size
        2.1.2. Lives
        2.1.3. Braid amount
        2.1.4. (Input) Player name
    2.2. Initialise Player and Treasure objects and generate random locations
    2.3. Generate a maze, apply braiding
    2.4. Calculate shortest path between player and treasure
    2.5. Multiply shortest path by length of one block, and divide by player speed for time, adding the variable 'Ease' to this time
    2.6. Launch the playable pygame window, displaying the following:
        2.6.1. Generated maze including speed modifiers on blocks
        2.6.2. Time remaining for this level
        2.6.3. Lives remaining
        2.6.4. Player and treasure sprites
    2.7. Player movement is allowed via arrow keys to traverse maze, collision with walls and modifiers on blocks are handled by pygame
3. Upon player reaching treasure:
    3.1. Score is increased by 1
    3.2. Every tenth completion 1 life point is awarded to the player (originally begins at 3)
    3.3. Grid size is increased
    3.4. Braid amount is reduced
    3.5. Ease is reduced
    3.6. New treasure location generated; player location remains same
    3.7. Objective 2.3 onwards to 2.7 is completed
4. Upon player running out of time:
    4.1. Score is decreased by 1
    4.2. 1 life point is deducted
    4.3. Grid size is decreased
    4.4. Braid amount is increased
    4.5. Ease is increased
    4.6. Objective 3.6 onwards to 3.7 completed
5. Upon life total reaching 0:
    5.1. Playable environment finishes
    5.2. High score written to csv file
    5.3. Objective 1 is completed
6. Maze generation:
    6.1. Implement the following algorithms:
        6.1.1. Binary Tree
        6.1.2. Sidewinder

6.1.3. Aldous-Broder

6.1.4. Braiding

6.1.5. Dijkstra’s

6.2. Algorithm used is dictated as follows:

6.2.1. If player score < 5, Binary Tree

6.2.2. If player score >= 5 and < 10, Sidewinder

6.2.3. If player score >= 10, Aldous-Broder

6.3. Relevant algorithm is used along with grid size to generate maze

6.4. During generation, each block has a small chance of spawning a modifier on the block, either increasing or decreasing player movement speed while on the block

Proposed solution

Proposed solution 1.0 Data dictionary –

The variables and their starting values I can see being used in the solution are shown below. I have also included what data type they will be and their purpose.

I have decided to separate this section into the variables used for maze generation and those used for the game logic of the solution, as some variables are used in both parts of the solution but serve different purposes. Any variables used in classes or their methods will be discussed in the design section alongside the classes.

Proposed solution 1.1 Maze generation –

Variable Name	Data Type	Example Value	Purpose
GridSize	Integer	5	Determines the size of the grid in maze generation
LOB	List	[] containing GridSize*GridSize amount of Block objects	List of all Block objects in the generated maze, allows algorithms to find adjacent blocks via the index of the block in the LOB. Also allows algorithms to easily loop through each block in the maze.
Index	Integer	0	Stores index of current block and is incremented each time a block is visited. Used in conjunction with LOB to access Block objects
Row	Integer	0	Stores row of the current block being visited by the algorithm
Column	Integer	0	Stores column of the current block being visited by the algorithm
Decider	Integer	Random integer in	Random integer that determines the direction that an algorithm carves. The

		range 0-1 (0-3 in Aldous-Broder)	range is 0-1 (north, east) for Binary Tree and Sidewinder, and is 0-3 (north, east, south, west) for Aldous-Broder
DeciderSpeed	Integer	Random integer in range 1-20	Random integer that determines whether or not a block will spawn a modifier. If the value is 1, a slow modifier is spawned, a value of 20 spawns a fast modifier
Run	List	[]	Used only in the Sidewinder algorithm to maintain a list of the current run
DeciderRun	Integer	Random integer in range 0-(length of Run)	Random integer that determines which block in the current run will be carved north from to the next row. Used only in Sidewinder
IndexRun	Integer	Run[DeciderRun]	Stores the index of the block in the current run that will be carved north from. Used only in Sidewinder
UnvisitedCount	Integer	GridSize*GridSize	Stores the current number of unvisited cells. Used in Aldous-Broder and Dijkstra’s algorithm
AdjacentIndex	Integer	0	Stores index of the block that is being carved to; this will overwrite index in the next iteration of the loop. Used in Aldous-Broder
NewRow	Integer	0	Same purpose as AdjacentIndex, except this relates to the row of the block being carved to
NewColumn	Integer	0	Same purpose as AdjacentIndex, except this relates to the column of the block being carved to
BraidSeverity	Float	0.1	Determines the density of dead ends in the final maze. A value of 1 means no dead ends have been removed, 0 means all dead ends have been removed. Used in the braiding algorithm
Ends	List	[]	Contains all directions which have a wall for the current block. If the length of this list is 3, the block is a dead end. Used in the braiding algorithm
Choice	Character	‘N’, ’E’, ‘S’, or ‘W’	Used to determine which wall of a dead end will be removed. Used in the braiding algorithm
sRow	Integer	GridSize // 2	Represents the current player row at the start of new maze generation. Used in Dijkstra’s algorithm where this

			represents the row of the root node. Begins as centre row
sColumn	Integer	GridSize // 2	Represents the current player column at the start of new maze generation. Used in Dijkstra's algorithm where this represents the column of the root node. Begins as centre column
gRow	Integer	Random integer in range 0-GridSize	This is the row of the goal node, when the current row is equal to gRow and the current column is equal to the gColumn, Dijkstra's terminates and returns the distance to this goal node
gColumn	Integer	Random integer in range 0-GridSize	This is the column of the goal node
VisitedCount	Integer	1	Stores the number of visited blocks. Used in Dijkstra's algorithm
Frontier	List	[]	Used in Dijkstra's algorithm. Contains the list of blocks in the most recent level of the traversal, specifically, the blocks visited in the last iteration of the loop
ToVisit	Nested list	[]	This is a list containing elements that are also lists. The first element of a nested list is the index of an unvisited block with an edge (broken wall) to a visited block. The distance of this visited block to the starting block (sRow, sColumn) is stored in the second element of the nested list. The number of lists in ToVisit will be equal to the number of unvisited blocks that also have an edge to a visited block
Weight	Float	0.5	The cost of visiting the current block. This is a value of 1 if the block contains no modifiers. A value of 0.5 is assigned if the block has a fast modifier, and 2 if the block has a slow modifier. The increase or reduction in weight is based on the percentage difference in time between going through the modifier than without. Fast modifiers double player speed, Slow modifiers half player speed.
hStart	Integer	50	This is used to create wall and modifier objects. This stores the horizontal co-

			ordinate of the top left corner of a modifier or wall sprite while its being instantiated.
vStart	Integer	200	This is used to create wall and modifier objects. This stores the vertical co-ordinate of the top left corner of a modifier or wall sprite while its being instantiated.

Proposed Solution 1.2 Game Logic –

Variable Name	Data Type	Starting Value	Purpose
c	Integer	50	This is a constant, it is used to scale different objects in the game, as well as player speed. This ensures everything is in proportion.
Name	String	Player input	Stores player name to be used in highscores
Choice	Character	‘1’, ‘2’, or ‘3’	Stores the player selection in the first menu shown to the player
CsvFile	TextIOWrapper	Contents of ‘highscores.txt’	Used as an identifier for the opened highscores file. This is used to display the highscores and write to the file.
SpamReader	DictReader	Contents of ‘highscores.txt’ but each row mapped to dictionaries	Used to separate the data read from highscores.txt. Maps each row to a dictionary, elements in dictionary are separated where the delimiter of ‘,’ is present in highscores.txt. Allows for separation of name, score and date fields.
Counter	Integer	1	Incremented each time a new row of the highscores.txt file is printed on screen. Rows are stopped from being displayed once this counter reaches 10.
Highscores	Nested list	[]	Contains list of lists. Each nested list represents a row in the original highscores.txt file, e.g. Highscores[0] is the first row of highscores.txt. The first, second and third

			elements of each nested list are score, name, and date, respectively.
GridSize	Integer	5	Used as a measure of difficulty. Increased by 1 on a successful level completion, decreased by 1 if unsuccessful. The maximum is c/2.5, minimum is 2
BraidSeverity	Float	0.1	Used as a measure of difficulty. Increased by 0.1 on a successful level completion, decreased by 0.1 if unsuccessful. The maximum is 1, minimum is 0.1
Difficulty	String	‘Easy’	Used as a measure of difficulty. Determines which maze generation algorithm is used. Easy = Binary Tree, Medium = Sidewinder, Hard = Aldous-Broder. Difficulty increases for every 5 score gained, up to a maximum of Hard
FPS	Integer	60	Determines the number of frames in a second of game time. This value is used in calculations to work out the number of seconds given to a player after Dijkstra’s returns the shortest path
Frame	Integer	0	This is the current frame of the game, 0 at the beginning, 5 seconds into the game this value is 300 and so on. Also used in calculations similar to FPS
Clock	Clock		This is a pygame clock object. At the end of each main game loop Clock.tick(FPS) is executed, this starts a new frame in the game
Width	Integer	GridSize * c	This is the width of the game window, determined by GridSize and c

Height	Integer	GridSize * c	This is the height of the game window, determined by GridSize and c
Display	pygame.Surface		This is a pygame surface object, it is the main screen which all sprites are drawn onto
PlayerL	Player		This is the player object, this allows for player movement through the methods of the Player class, and stores information about the players location, sprite, speed etc.
TreasureL	Treasure		This is the treasure object, the purpose of creating a class for the treasure is so that a sprite can be created, allowing for pygame to detect collisions between treasure and player.
sPath	Float	12.5	This is the shortest path calculated by Dijkstra's algorithm. This is used in time calculations.
TimeFrames	Float	(sPath * c) / PlayerL.Speed	This is the minimum number of frames that is required to reach the treasure from the location of the player at the start of the current maze generation. This is later used to calculate the number of seconds given to player. This is a float because it involved sPath in its calculation, which may be a float value.
TimeSeconds	Float	TimeFrames / 60	Same as TimeFrames, except this is a conversion into seconds
Ease	Float	5.0	This is the number of additional seconds given to the player (in addition to TimeSeconds). This is a measure of difficulty and decreases by 0.5 for every maze completion, increases by 0.5 for unsuccessful levels. Max is 5, minimum is 1.

TargetFrame	Float	(Frame + TimeFrames) + Ease * FPS	This is the frame that the treasure object must collide with the player object at the latest to be a successful maze completion. Once Frame is greater than this value, the level has been failed.
SecondsRemaining	Integer	Rounded down ((TargetFrame – Frame) / FPS)	This is the number of seconds left until the player loses this level.
CollisionFrame	Integer	0	This is the frame that the player object and treasure object collide. This is an integer and not a float like TimeFrames because it is independent of sPath.
CollisionCheck	Boolean	False	This is set to true when a new level needs to be started, i.e. when the player and treasure objects collide.
Completed	Integer	12	This is the amount of successfully completed maze levels. Every 10 completions award the player 1 life point, and the value of Completed never decreases
Score	Integer	5	This increases the same way Completed does, however, this decreases by 1 when the player is unsuccessful in completing a level. This is used instead of Completed when writing to the highscores.txt file.
Lives	Integer	3	This is the amount of lives the player has remaining. Lives decreases by 1 on every unsuccessful level completion. Once this reaches 0, the game stops, the score is automatically added to highscores.txt and the selection menu is displayed once again.
Running	Boolean	True	This determines whether the game is over or still running. Once the player runs out of

			lives, this is set to False, which then stops the main game loop.

Design

High level overview

High level overview 1.0.0 Classes –

I plan for the following 5 classes to be used in the solution:

1. Block
2. Player
3. Wall
4. Modifier
5. Treasure

An overview of the attributes and methods of each class are discussed below, in addition to the purpose of the class. Any specifics about the methods will be discussed in the pseudocode section.

High level overview 1.1.0 Block class –

The block class will be used in the generation of mazes. Each square in a generated maze will be represented by a 'Block' object. The block class does not contain any methods, this class is in place only to store information about squares in the maze efficiently.

High level overview 1.1.1 Block class attributes –

Attribute Name	Data Type	Example Value	Purpose
Row	Integer	-1	Stores the row in the maze that the block occupies
Column	Integer	-1	Stores the column in the maze that the block occupies
North	Integer	1	Stores the state of the northern wall of the block. A value of 1 means a wall exists, 0 means no wall exists in that direction
South	Integer	1	Stores the state of the southern wall of the block. A value of 1 means a wall exists, 0 means no wall exists in that direction

West	Integer	1	Stores the state of the western wall of the block. A value of 1 means a wall exists, 0 means no wall exists in that direction
East	Integer	1	Stores the state of the eastern wall of the block. A value of 1 means a wall exists, 0 means no wall exists in that direction
Visited	Integer	0	Stores whether or not the block has been visited before. Used both in Aldous-Broder and in Dijkstra's algorithm
Distance	Float	0	Stores the distance from the root node and this node. Used in Dijkstra's algorithm
SpeedMultiplier	Float	1	Stores the multiplier that is applied to player speed when the player collides with this block.

High level overview 1.2.0 Player class –

The player class will be used to store information about the player, such as their position and speed. It will also be used to update player position and check for collisions with other objects in each frame by using methods.

High level overview 1.2.1 Player class attributes –

Attribute Name	Data Type	Example Value	Purpose
image	pygame.Surface		Used to create a rectangular surface in pygame, this is the player sprite
rect	pygame.Rect		This is a pygame object that is used to store rectangular co-ordinates, this is used to find and store the x and y co-ordinates of the player sprite
rect.x	Integer	55	This is an attribute of the rect attribute (which is an object). Represents the x co-ordinate of the top left corner of the player sprite
rect.y	Integer	115	This is an attribute of the rect attribute (which is an object). Represents the y co-ordinate

			of the top left corner of the player sprite
xMoved	Float	4	This is the distance that the player moves in the horizontal direction in one frame
yMoved	Float	-4	This is the distance that the player moves in the vertical direction in one frame
Speed	Float	4	This is the current player speed in any direction, this value is used in calculations for xMoved and yMoved
Modified	String	‘Faster’	This stores the current state of player speed modification, i.e. if a player is currently under the effect of a modifier

High level overview 1.2.2 Player class methods –

Method Name	Parameters	Purpose
RecordMove	self, xChange, yChange	This method enables player movement. When a key is pressed down this method will be called and will add or subtract Speed from xMoved or yMoved, dependent on which movement key is pressed.
UpdatePos	self, Walls, Treasure, Modifiers, Collision, sPath, Score, GridSize, BraidSeverity, TreasureL, PlayerL	This is the main method of the game aspect of the solution. This method is called every frame. This checks for collisions between the player and walls, as well as collisions between the player and treasure. If a collision between player and treasure is detected, Score is increased by 1, a new maze is generated and maze generation settings are changed (e.g. GridSize, BraidSeverity).

High level overview 1.3.0 Treasure class –

This class will serve a similar purpose to the player class, except its for the treasure placed in the maze. This will be the goal for the player to reach.

High level overview 1.3.1 Treasure class attributes –

Attribute Name	Data Type	Example Value	Purpose
Row	Integer	2	This is the current row the treasure is in; this is used in Dijkstra's algorithm as the row of the goal block
Column	Integer	3	This is the current column the treasure is in; this is used in Dijkstra's algorithm as the column of the goal block
image	pygame.Surface		This is the treasure sprite, used to load in the png file of a treasure chest
rect	pygame.Rect		This is a pygame object that is used to store rectangular co-ordinates, used to find and store the x and y co-ordinates of the treasure sprite
rect.x	Integer	216	This is an attribute of the rect attribute (which is an object). Represents the x co-ordinate of the top left corner of the treasure sprite
rect.y	Integer	16	This is an attribute of the rect attribute (which is an object). Represents the y co-ordinate of the top left corner of the treasure sprite

High level overview 1.3.2 Treasure class methods –

Method Name	Parameters	Purpose
UpdatePosT	self, GridSize, OldRow, OldColumn	This method generates a new random row and column for the treasure after each level. The corresponding rect.x and rect.y values are also calculated so that the treasure sprite is displayed correctly in the centre of whichever block it is randomly generated in. OldRow and OldColumn are required as inputs so that the treasure can be guaranteed to spawn on a new block.

High level overview 1.4.0 Wall class –

This class will be used to create sprites for each wall in the maze, and to enable collision detection between objects of this class and the player object.

High level overview 1.4.1 Wall class attributes –

Attribute Name	Data Type	Example Value	Purpose
image	pygame.Surface		This is the wall sprite, used to create a rectangular surface in pygame and is required for collision detection
rect	pygame.Rect		This is a pygame object that is used to store rectangular co-ordinates, used to find the x and y co-ordinates of the wall sprite
Width	Integer	50	This is the width of the wall sprite
Height	Integer	50	This is the height of the wall sprite
rect.x	Integer	250	This is an attribute of the rect attribute (which is an object). Represents the x co-ordinate of the top left corner of the wall sprite
rect.y	Integer	100	This is an attribute of the rect attribute (which is an object). Represents the y co-ordinate of the top left corner of the wall sprite

High level overview 1.5.0 Modifier class –

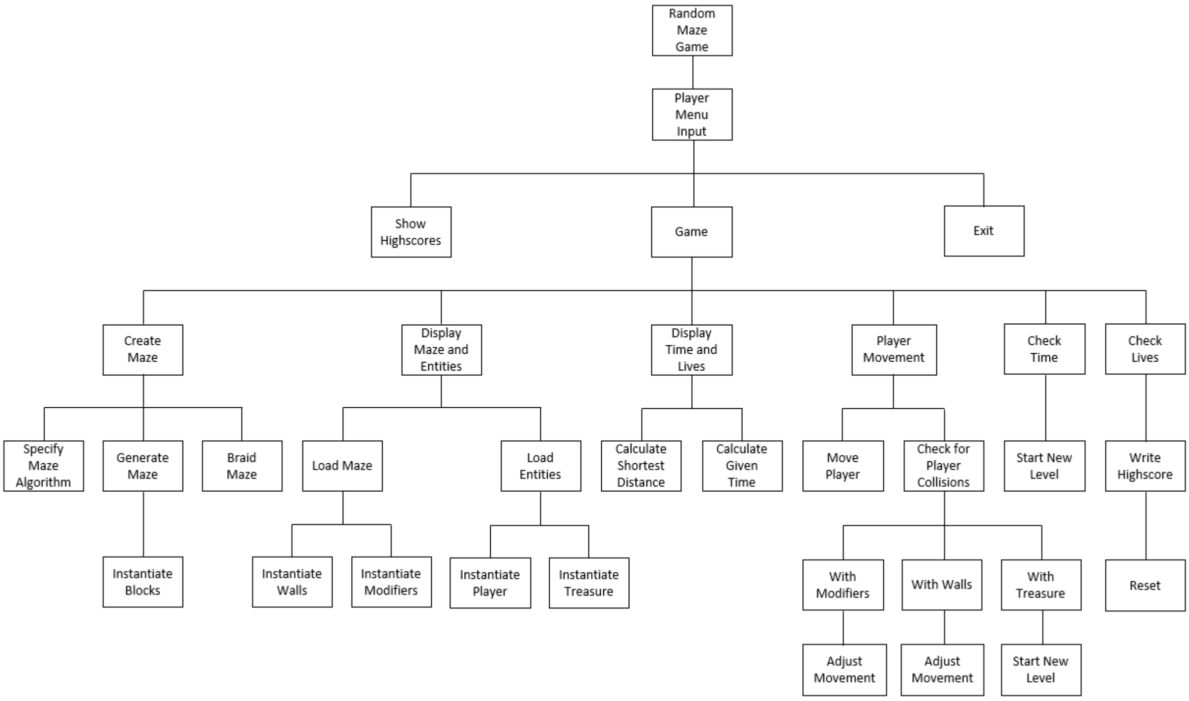
This class has almost an identical purpose to both the wall and treasure classes. That purpose being to easily generate multiple sprites at many locations in the maze and allow for pygame to detect collisions between the player sprite and the sprites of any objects of this class.

High level overview 1.5.1 Modifier class attributes –

Attribute Name	Data Type	Example Value	Purpose
image	pygame.Surface		This is the modifier sprite, used to create a rectangular surface in pygame and is required for collision detection
rect	pygame.Rect		This is a pygame object that is used to store rectangular co-

			ordinates, used to find the x and y co-ordinates of the modifier sprite
rect.x	Integer	105	This is an attribute of the rect attribute (which is an object). Represents the x co-ordinate of the top left corner of the modifier sprite
rect.y	Integer	5	This is an attribute of the rect attribute (which is an object). Represents the x co-ordinate of the top left corner of the modifier sprite
Type	String	‘Fast’	This used to store the type of the modifier. Either ‘Fast’ or ‘Slow’. This determines the image fill; fast modifiers are coloured blue, slow modifiers are coloured brown.

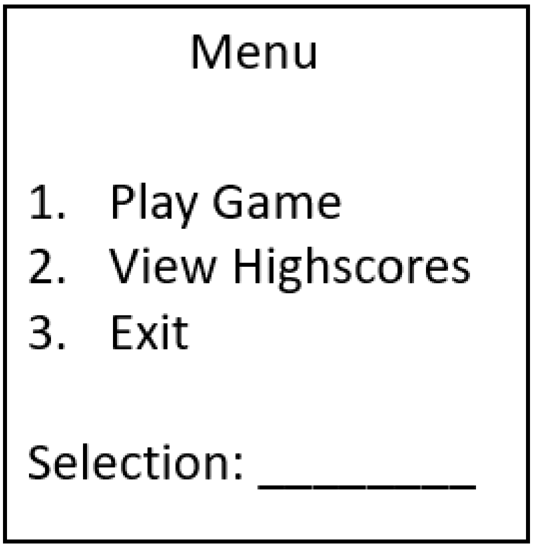
High level overview 2.0.0 Hierarchy diagram –



Sketches

Sketches 1.0 Selection Menu –

This is the first interaction with the user, they will need to select one of these 3 options, by typing the corresponding number in the selection prompt.



Menu

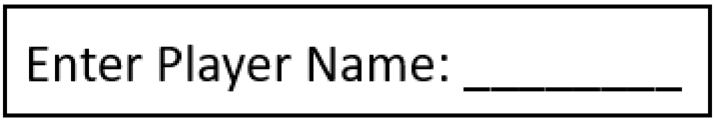
1. Play Game
2. View Highscores
3. Exit

Selection: ________

Figure 1 – Menu Selection Prompt

Sketches 1.1 Play Game –

Upon selecting option 1, the player will be prompted for their name, which will be later used to record their score in the highscores csv file.



Enter Player Name: ________

Figure 2 – Player Name Prompt

After entering the player name, the pygame playable window will appear, with the first maze of grid size 5 loaded, generated with the Binary Tree algorithm (as starting score is 0),

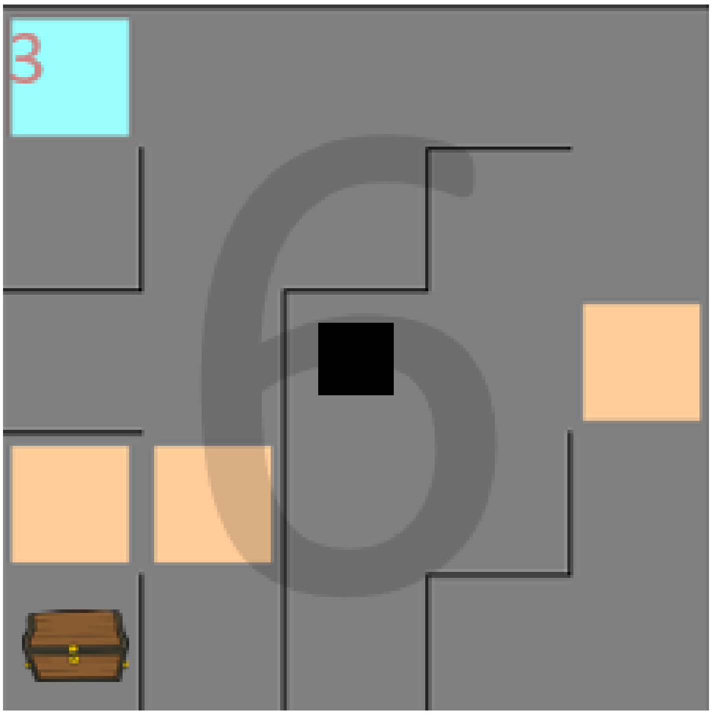


Figure 3 – Example sketch of playable environment

and with significant braiding. The treasure sprite, player sprite and any speed modifier sprites on blocks will be loaded in.

Figure 3 is a sketch of the design I am aiming for in my final implementation, the player is the solid black square, the goal is the treasure chest. The blue patch represents a 'Fast' modifier, one that increases player speed, the brown patch represents a 'Slow' modifier, one that decreases player speed. In the background, the timer is displayed in a transparent black font. In the upper left corner the remaining lives are displayed in a transparent red font.

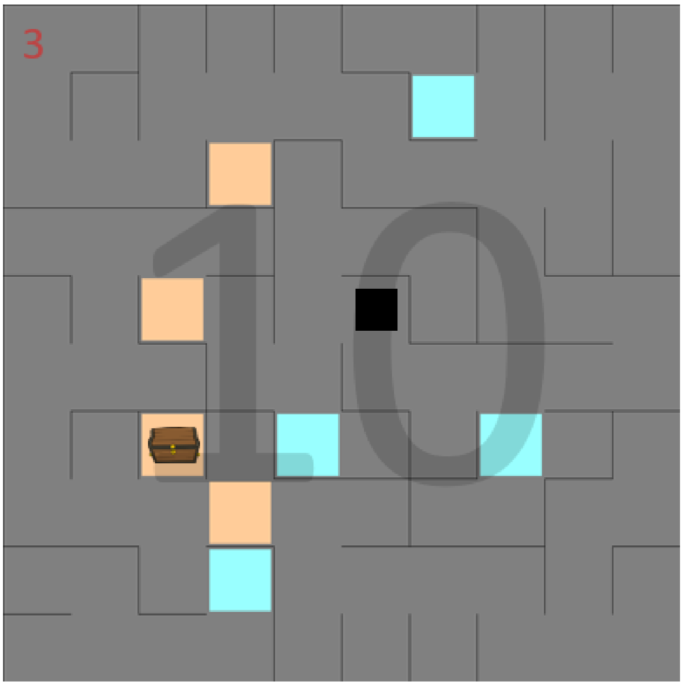


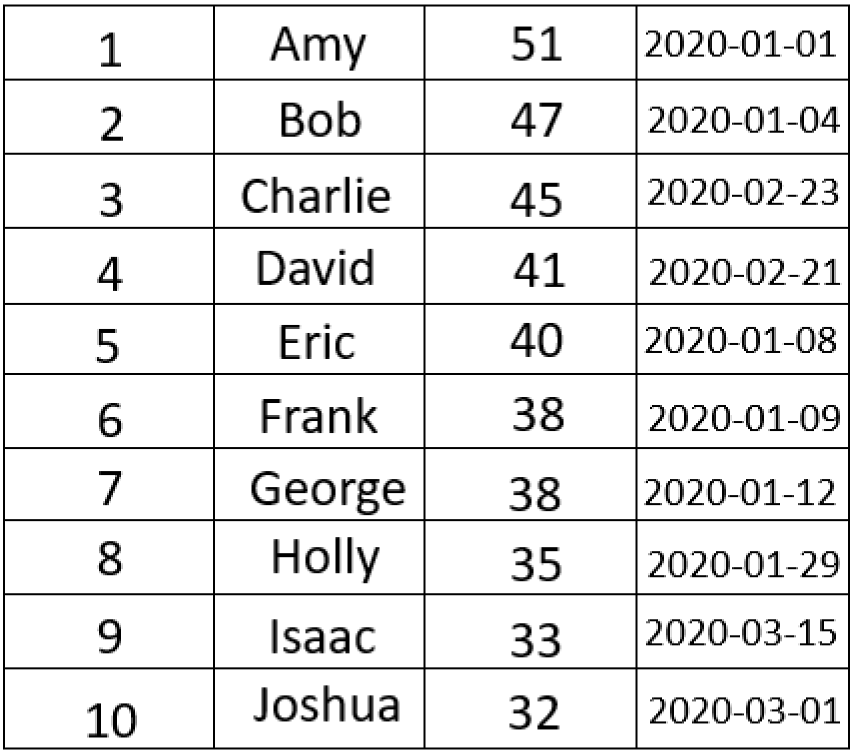
Figure 4 – Example sketch of harder playable environment

As the player accumulates score, the mazes generated will get progressively harder to solve by changing the maze algorithm used, the size of the maze, the amount of braiding, and the amount of extra time given to the player, some of the later mazes generated may look similar to what is shown above in Figure 4.

Once the player has run out of lives, the playable environment will pause, the score will be added to the highscores csv file, and the menu shown in Figure 1 will be displayed again

Sketches 1.2 View Highscores –

If the ‘View Highscores’ option is selected in the menu shown in Figure 1, the csv file is read and the first 10 entries in the file are displayed (the csv file is sorted by highest score).



1	Amy	51	2020-01-01
2	Bob	47	2020-01-04
3	Charlie	45	2020-02-23
4	David	41	2020-02-21
5	Eric	40	2020-01-08
6	Frank	38	2020-01-09
7	George	38	2020-01-12
8	Holly	35	2020-01-29
9	Isaac	33	2020-03-15
10	Joshua	32	2020-03-01

The highscores are sorted by highest score. In cases where the score is the same such as position 6 and 7, the oldest score comes first .

Pseudocode

Pseudocode 1.0 Maze generation –

This part of the pseudocode section (1.x) is dedicated to maze generation and manipulation algorithms, this includes:

1. Binary Tree
2. Sidewinder
3. Aldous-Broder
4. Braiding
5. Dijkstra’s algorithm
6. LoadMaze

The value of decider determines which direction to carve. Values of 0,1,2, and 3 are mapped to north, east, south, west respectively.

Pseudocode 1.1 Binary Tree –

This is the first maze generation algorithm that is used, and takes the following parameters:

Parameter Name	Data Type	Purpose
GridSize	Integer	Defines the size of the maze that is being generated
LOB	List	This is a list of Block objects. This is used to maintain information about the edges of all blocks in the maze.

```
BEGIN
     Set Index to -1
     FOR Row in range 0 to GridSize DO
          FOR Column in range 0 to GridSize DO
               Index = Index + 1
               LOB[Index].Row = Row
               LOB[Index].Column = Column
               Decider = random integer 0 to 1
               IF Row = 0 and Column = GridSize - 1 THEN
                    skip this iteration
               ELSEIF Row = 0 THEN
                    Decider = 1
               ELSEIF Column = GridSize - 1 THEN
                    Decider = 0
               ENDIF
               IF Decider = 0 THEN
                    LOB[Index].North = 0
                    LOB[Index - GridSize].South = 0
               ELSEIF Decider = 1 THEN
                    LOB[Index].East = 0
                    LOB[Index + 1].West = 0
               ENDIF
               DeciderSpeed = random integer 0 to 20
               IF DeciderSpeed  < 3 THEN
                    LOB[Index].SpeedMultiplier = 0.5
               ELSEIF DeciderSpeed > 18 THEN
                    LOB[Index].SpeedMultiplier = 2
               ENDIF
          ENDLOOP
     ENDLOOP
     RETURN LOB
END
```

Pseudocode 1.2 Sidewinder –

This is the second maze generation algorithm that is used, and takes the following parameters:

Parameter Name	Data Type	Purpose
GridSize	Integer	Defines the size of the maze that is being generated

LOB	List	This is a list of Block objects. This is used to maintain information about the edges of all blocks in the maze.

```
BEGIN
     Set Index to -1
     Set Run to []
     FOR Row in range 0 to GridSize DO
          FOR Column in range 0 to GridSize DO
               Index = Index + 1
               append Index to Run
               LOB[Index].Row = Row
               LOB[Index].Column = Column
               Decider = random integer 0 to 1
               IF Row = 0 and Column = GridSize – 1 THEN
                    skip this iteration
               ELSEIF Row = 0 THEN
                    Decider = 1
               ELSEIF Column = GridSize – 1 THEN
                    Decider = 0
               ENDIF
               IF Decider = 1 THEN
                    LOB[Index].East = 0
                    LOB[Index + 1].West = 0
                    IF Index + 1 not in Run THEN
                         append Index + 1 to Run
                    ENDIF
               ELSEIF Decider = 0 THEN
                    DeciderRun = random integer 0 to (length
                    of Run – 1)
                    IndexRun = Run[DeciderRun]
                    LOB[IndexRun].North = 0
                    LOB[IndexRun – GridSize].South = 0
                    Set Run to []
               ENDIF
               DeciderSpeed = random integer 1 to 20
               IF DeciderSpeed < 3 THEN
                    LOB[Index].SpeedMultiplier = 0.5
               ELSEIF DeciderSpeed > 18 THEN
                    LOB[Index].SpeedMultiplier = 2
               ENDIF
          ENDLOOP
     ENDLOOP
     FOR Index in range 0 to GridSize DO
          LOB[Index].North = 1
          LOB[Index + (GridSize**2 – GridSize)].South = 1
     ENDLOOP
     RETURN LOB
END
```

Pseudocode 1.3 Aldous-Broder –

This is the third maze generation algorithm that is used, and takes the following parameters:

Parameter Name	Data Type	Purpose
GridSize	Integer	Defines the size of the maze that is being generated
LOB	List	This is a list of Block objects. This is used to maintain information about the edges of all blocks in the maze.

```
BEGIN
    FOR Block in LOB DO
        Block.Visited = 0
    ENDLOOP
    UnvisitedCount = GridSize ** 2
    Index = 0
    Row = Column = 0
    WHILE UnvisitedCount > 0 DO
        Decider = random integer 0 to 3
        UnvisitedCount = UnvisitedCount - 1
        DeciderSpeed = random integer 1 to 20
        IF DeciderSpeed < 3 THEN
            LOB[Index].SpeedMultiplier = 0.5
        ELSELIF DeciderSpeed > 18 THEN
            LOB[Index].SpeedMultiplier = 2
        ENDIF
        IF Row = 0 THEN
            IF Column = 0 THEN
                Decider = random integer 1 to 2
            ELSEIF Column = GridSize - 1 THEN
                Decider = random integer 2 to 3
            ELSE
                Decider = random integer 1 to 3
            ENDIF
        ELSEIF Row = GridSize - 1 THEN
            IF Column = 0 THEN
                Decider = random integer 0 to 1
            ELSEIF Column = GridSize - 1 THEN
                WHILE Decider in [1, 2] DO
                    Decider = random integer 0 to 3
                ENDLOOP
            ELSE
                WHILE Decider = 2 DO
                    Decider = random integer 0 to 3
                ENDLOOP
            ENDIF
        ELSEIF Column = 0 THEN
            Decider = random integer 0 to 2
        ELSEIF Column = GridSize - 1 THEN
```

```
                    Decider = random integer 0 to 3
                ENDIF
                AdjacentIndex = 0
                IF Decider = 0 THEN
                    AdjacentIndex = Index - GridSize
                    NewRow = Row - 1
                    IF LOB[AdjacentIndex].Visited = 0 THEN
                        LOB[Index].North = 0
                        LOB[AdjacentIndex].South = 0
                    ENDIF
                    Row = NewRow
                ELSEIF Decider = 1 THEN
                    AdjacentIndex = Index + 1
                    NewColumn = Column + 1
                    IF LOB[AdjacentIndex].Visited = 0 THEN
                        LOB[Index].East = 0
                        LOB[AdjacentIndex].West = 0
                    ENDIF
                    Column = NewColumn
                ELSEIF Decider = 2 THEN
                    AdjacentIndex = Index + GridSize
                    NewRow = Row + 1
                    IF LOB[AdjacentIndex].Visited = 0 THEN
                        LOB[Index].South = 0
                        LOB[AdjacentIndex].North = 0
                    ENDIF
                    Row = NewRow
                ELSEIF Decider = 3 THEN
                    AdjacentIndex = Index - 1
                    NewColumn = Column - 1
                    IF LOB[AdjacentIndex].Visited = 0 THEN
                        LOB[Index].West = 0
                        LOB[AdjacentIndex].East = 0
                    ENDIF
                    Column = NewColumn
                ENDIF
                Index = AdjacentIndex
            ENDLOOP
            RETURN LOB
        END
```

Pseudocode 1.4 Braiding –

This algorithm is the removal of dead ends in an already generated maze, creates multiple paths from one block to another, and takes the following parameters:

Parameter Name	Data Type	Purpose
GridSize	Integer	Defines the size of the maze that is being generated

LOB	List	This is a list of Block objects. This is used to maintain information about the edges of all blocks in the maze.
BraidSeverity	Float	Determines the proportion of dead ends that is removed. Lower value means more dead ends removed. This is inversely proportional to the number of dead ends.

```
BEGIN
    Decider = 1 / BraidSeverity
    FOR Index in range 0 to GridSize ** 2 DO
        Ends = []
        IF (Index + 1) % Decider = 0 THEN
            skip this iteration
        ENDIF
        IF LOB[Index].North = 1 THEN
            append 'N' to Ends
        ENDIF
        IF LOB[Index].East = 1 THEN
            append 'E' to Ends
        ENDIF
        IF LOB[Index].South = 1 THEN
            append 'S' to Ends
        ENDIF
        IF LOB[Index].West = 1 THEN
            append 'W' to Ends
        ENDIF

        IF LOB[Index].Row = 0 THEN
            remove 'N' from Ends
        ENDIF
        IF LOB[Index].Row = GridSize - 1 THEN
            remove 'S' from Ends
        ENDIF
        IF LOB[Index].Column = 0 THEN
            remove 'W' from Ends
        ENDIF
        IF LOB[Index].Column = GridSize - 1 THEN
            remove 'E' from Ends
        ENDIF

        IF length of Ends = 3 THEN
            Choice = random choice from Ends
        ELSE
            skip this iteration
        ENDIF

        IF Choice = 'N' THEN
            LOB[Index].North = 0
            LOB[Index - GridSize].South = 0
```

```
          ELSEIF Choice = ‘E’ THEN
               LOB[Index].East = 0
               LOB[Index + 1].West = 0
          ELSEIF Choice = ‘S’ THEN
               LOB[Index].South = 0
               LOB[Index + GridSize].North = 0
          ELSEIF Choice = ‘W’ THEN
               LOB[Index].West = 0
               LOB[Index – 1].East = 0
          ENDIF
     ENDLOOP
     RETURN LOB
END
```

Pseudocode 1.5 Dijkstra’s algorithm –

This algorithm returns the shortest distance between any two given blocks in a maze. In the case of my solution, the start block is the location of the player and the goal node is the location of the treasure. The algorithm considers speed modifiers on blocks in the form of weights. Dijkstra’s takes the following parameters:

Parameter Name	Data Type	Purpose
GridSize	Integer	Defines the size of the maze that is being generated
LOB	List	This is a list of Block objects. This is used to maintain information about the edges of all blocks in the maze.
sRow	Integer	The starting row, i.e. the row of the player
sColumn	Integer	The starting column, i.e. the column of the player
gRow	Integer	The goal row, i.e. the row of the treasure
gColumn	Integer	The goal column, i.e. the row of the treasure

```
BEGIN
     FOR Block in LOB DO
          Block.Visited = 0
          Block.Distance = (GridSize ** 2) * 2
     ENDLOOP
     VisitedCount = 1
     Index = (sRow * GridSize) + sColumn
     Frontier = []
     append Index to Frontier
     LOB[Index].Distance = 0
     LOB[Index].Visited = 1
     WHILE VisitedCount < length of LOB DO
          ToVisit = []
          FOR Index in Frontier DO
               IF LOB[Index].North = 0 THEN
                    append [Index – GridSize,
                    LOB[Index].Distance] to ToVisit
               ENDIF
               IF LOB[Index].East = 0 THEN
                    append [Index + 1, LOB[Index].Distance] to
                    ToVisit
               ENDIF
               IF LOB[Index].South = 0 THEN
                    append [Index + GridSize,
                    LOB[Index].Distance] to ToVisit
               ENDIF
               IF LOB[Index].West = 0 THEN
                    append [Index - 1, LOB[Index].Distance] to
                    ToVisit
               ENDIF
          ENDLOOP

          Frontier = []
          FOR IndexList in ToVisit DO
               IF LOB[IndexList[0]].SpeedMultiplier = 1 THEN
                    Weight = 1
                    IF LOB[IndexList[0]].Distance >
                    (IndexList[1] + Weight) THEN
                         append IndexList[0] to Frontier
                         LOB[IndexList[0]].Distance =
                         IndexList[1] + Weight
                         IF LOB[IndexList[0]].Visited = 0 THEN
                              VisitedCount = VisitedCount + 1
                              LOB[IndexList[0]].Visited = 1
                         ENDIF
                         IF LOB[IndexList[0]].Row = gRow and
                         LOB[IndexList[0]].Column = gColumn
                         THEN
                              RETURN
                              LOB[IndexList[0]].Distance
                         ENDIF
                    ENDIF
               ENDIF
          ENDLOOP

          FOR IndexList in ToVisit DO
               IF LOB[IndexList[0]].SpeedMultiplier = 0.5 THEN
                    Weight = 0.5
                    IF LOB[IndexList[0]].Distance >
                    (IndexList[1] + Weight) THEN
                         append IndexList[0] to Frontier
                         LOB[IndexList[0]].Distance =
                         IndexList[1] + Weight
                         IF LOB[IndexList[0]].Visited = 0 THEN
                              VisitedCount = VisitedCount + 1
                              LOB[IndexList[0]].Visited = 1
                         ENDIF
```

```
                         IF LOB[IndexList[0]].Row = gRow and
                         LOB[IndexList[0]].Column = gColumn
                         THEN
                              RETURN
                              LOB[IndexList[0]].Distance
                         ENDIF
                    ENDIF
               ENDIF
          ENDLOOP

          FOR IndexList in ToVisit DO
               IF LOB[IndexList[0]].SpeedMultiplier = 2 THEN
                    Weight = 0.5
                    IF LOB[IndexList[0]].Distance >
                    (IndexList[1] + Weight) THEN
                         append IndexList[0] to Frontier
                         LOB[IndexList[0]].Distance =
                         IndexList[1] + Weight
                         IF LOB[IndexList[0]].Visited = 0 THEN
                              VisitedCount = VisitedCount + 1
                              LOB[IndexList[0]].Visited = 1
                         ENDIF
                         IF LOB[IndexList[0]].Row = gRow and
                         LOB[IndexList[0]].Column = gColumn
                         THEN
                              RETURN
                              LOB[IndexList[0]].Distance
                         ENDIF
                    ENDIF
               ENDIF
          ENDLOOP
     ENDLOOP
END
```

The reason for the repetition of code in the last 3 nested loops is to ensure that the smallest weight connections are considered first.

Pseudocode 1.6 LoadMaze –

This procedure takes the following parameters:

Parameter Name	Data Type	Purpose
GridSize	Integer	Defines the size of the maze that is being generated
LOB	List	This is a list of Block objects. This is used to maintain information about the edges of all blocks in the maze.
c	Integer	This is the constant that is used to scale all sprites in the game so that they are the correct size relative to each other

hStart	Integer	This is the horizontal co-ordinate of the top left corner of the object sprite
vStart	Integer	This is the vertical co-ordinate of the top left corner of the object sprite

```
BEGIN
     FOR Index in range 0 to GridSize ** 2 THEN
          IF LOB[Index].North = 1 THEN
               Instantiate Wall(c + 1, 1, hStart, vStart)
          ENDIF
          IF LOB[Index].West = 1 THEN
               Instantiate Wall(1, c, hStart, vStart)
          ENDIF
          IF Index >= ((GridSize ** 2) - GridSize) THEN
               Instantiate Wall(c + 1, 1, hStart, vStart + c)
          ENDIF
          IF LOB[Index].SpeedMultiplier = 0.5 THEN
               Instantiate Modifier(40, hStart + 5, vStart +
               5, 'Slow')
          ELSEIF LOB[Index].SpeedMultiplier = 2 THEN
               Instantiate Modifier(40, hStart + 5, vStart +
               5, 'Fast')
          ENDIF
          hStart = hStart + c
          IF (Index + 1) MOD GridSize = 0 THEN
               Instantiate Wall(1, c, hStart, vStart)
               hStart = 0
               vStart = vStart + c
          ENDIF
     ENDLOOP
END
```

Pseudocode 2.0 Game Logic –

This part of the pseudocode section (2.x) is dedicated to the procedures and functions of the game logic of this project excluding classes, this includes:

1. Menu
2. ShowHighscores
3. AddScore
4. Game

Pseudocode 2.1 Menu –

This is the first thing the player sees when launching the program. This is a simple function that gives the player an option and returns the player input. This function takes no parameters

```
BEGIN
     OUTPUT "----------MENU----------"
     OUTPUT "1. Play Game"
     OUTPUT "2. View Highscores"
     OUTPUT "3. Exit"
     INPUT Choice
     RETURN Choice
END
```

Pseudocode 2.2 ShowHighscores –

This is run if the player chooses option 2 in the menu. This procedure returns the top 10 highscores. This procedure takes no parameters

```
BEGIN
     WITH open('highscores.txt') as CsvFile DO
          SpamReader = DictReader(CsvFile)
          Counter = 1
          FOR Row in SpamReader DO
               OUTPUT (Counter, Row['Name'], ':',
               Row['Score'], ':', Row['Date'])
               IF Counter = 10 THEN
                    ENDLOOP
               ENDIF
               Counter += 1
          ENDLOOP
END
```

Pseudocode 2.3 AddScore –

This is run once the player's lives count drops to 0. This read the highscores.txt file, adds the score achieved by player and sorts the list. This sorted list then overrides the contents of highscores.txt. This procedure takes the following parameters:

Parameter Name	Data Type	Purpose
Name	String	Player name that is written to highscores.txt
Score	Integer	The achieved player score that is written to highscores.txt

```
BEGIN
     WITH open('highscores.txt') as CsvFile DO
          SpamReader = DictReader(CsvFile)
          Highscores = [[int(Score), Name, str(date.today())]]
          FOR Row in SpamReader DO
               IF Row['Score'] = 'Score' THEN
                    skip this iteration
               ENDIF
```

```
                append [int(Row['Score']), Row['Name'],
                str(Row['Date'])] to Highscores
           ENDLOOP
           Sort Highscores
      WITH open('highscores.txt') as CsvFile DO
           write 'Score,Name,Date' to CsvFile
           CsvFile start new line
           FOR Entry in Highscores DO
                write str(Entry[0]) to CsvFile
                write ',' to CsvFile
                write Entry[1] to CsvFile
                write ',' to CsvFile
                write str(Entry[2]) to CsvFile
                CsvFile start new line
           ENDLOOP
END
```

Pseudocode 2.4 Game –

This procedure takes no parameters.

```
BEGIN
     GridSize = 5
     BraidSeverity = 0.1
     Difficulty = 'Easy'
     LOB = [Block() for block in range(GridSize ** 2)]
     IF Difficulty = 'Easy' THEN
          LOB = BinaryTree(GridSize, LOB)
     ELSEIF Difficulty = 'Medium' THEN
          LOB = Sidewinder(GridSize, LOB)
     ELSEIF Difficulty = 'Hard' THEN
          LOB = AldousBroder(GridSize, LOB)
     ENDIF
     LOB = Braid(GridSize, LOB, BraidSeverity)
     LoadMaze(c, hStart, vStart, LOB, GridSize)
     INPUT Name
     FPS = 60
     Frame = 0
     Clock = pygame.Clock
     Width = Height = GridSize * c
     Display = pygame.Display(Width, Height)
     Instantiate Player as PlayerL
     Instantiate Treasure as TreasureL
     PlayerL.x = (GridSize * c / 2) + 1
     PlayerL.y = (GridSize * c / 2) + 1

     Row = Column = GridSize // 2
     sPath = Dijkstra(LOB, GridSize, Row, Column,
     TreasureL.Row, TreasureL.Column)
     TimeFrames = (sPath * c) / PlayerL.Speed
```

```
TimeSeconds = TimeFrames / FPS
Ease = 5
TargetFrame = (Frame + TimeFrames) + Ease * FPS
SecondsRemaining = str((TargetFrame – Frame) / FPS)
CollisionFrame = 0
COllisionCheck = False
Score = Completed = 0
Lives = 3

Running = True
WHILE Running DO
     FOR Event in pygame.event.get() DO
          IF Event.type = KeyDown THEN
               IF Event.Key = KeyUp THEN
                    PlayerL.RecordMove(0, -PlayerL.Speed)
               ELSEIF Event.Key = KeyRight THEN
                    PlayerL.RecordMove(PlayerL.Speed, 0)
               ELSEIF Event.Key = KeyDown THEN
                    PlayerL.RecordMove(0, PlayerL.Speed)
               ELSEIF Event.Key = KeyLeft THEN
                    PlayerL.RecordMove(-PlayerL.Speed, 0)
               ENDIF
          ELSEIF Event.type = KeyUp THEN
               IF Event.Key = KeyUp THEN
                    PlayerL.RecordMove(0, PlayerL.Speed)
               ELSEIF Event.Key = KeyRight THEN
                    PlayerL.RecordMove(-PlayerL.Speed, 0)
               ELSEIF Event.Key = KeyDown THEN
                    PlayerL.RecordMove(0, -PlayerL.Speed)
               ELSEIF Event.Key = KeyLeft THEN
                    PlayerL.RecordMove(PlayerL.Speed, 0)
               ENDIF
          ENDIF
     ENDLOOP

     Score, CollisionCheck, sPath, GridSize,
     BraidSeverity = PlayerL.UpdatePos(LOW, LOT, LOM,
     CollisionCheck, sPath, Score, GridSize,
     BraidSeverity, TreasureL, PlayerL)

     IF CollisionCheck = True THEN
          Completed = Completed + 1
          IF Completed % 10 = 0 THEN
               Lives = Lives + 1
          ENDIF
          CollisionFrame = Frame
          TimeFrames = (sPath * c) / PlayerL.Speed
          TimeSeconds = TimeFrames % FPS
          TargetFrame = (Frame + TimeFrames) + Ease * FPS
          IF Ease > 1 THEN
               Ease = Ease – 0.5
          ENDIF
     Width = Height = GridSize * c
     Display = pygame.Display(Width, Height)
     CollisionCheck = False
     IF (Frame – CollisionFrame) % FPS = 0 THEN
          SecondsRemaining = str(math.floor((TargetFrame
          – Frame) / FPS)
     ENDIF
     IF TargetFrame < Frame THEN
          Lives = Lives - 1
          IF Lives = 0 THEN
               Running = False
          ENDIF
          CollisionCheck = True
          Score, CollisionCheck, sPath, GridSize,
          BraidSeverity, = PlayerL.UpdatePos(LOW, LOT,
          LOM, CollisionCheck, sPath, Score, GridSize,
          BraidSeverity, TreasureL, PlayerL)
          CollisionFrame = Frame
          TimeFrames = (sPath * c) / PlayerL.Speed
          TimeSeconds = TimeFrames % FPS
          TargetFrame = (Frame + TimeFrames) + Ease * FPS
          IF Ease < 3 THEN
               Ease = Ease + 0.5
          ENDIF
          Width = Height = GridSize * c
          Display = pygame.Display(Width, Height)
          CollisionCheck = False
     ENDIF

     Display.fill(White)
     Draw LOW
     Draw LOM
     IF length of SecondsRemaining >= 2 THEN
          TFont = pygame.font(‘calibri’, (c * GridSize))
     ELSE
          TFont = pygame.font(‘calibri’, (c * GridSize) +
          c)
     ENDIF
     Timer = TFont.Draw(SecondsRemaining, Black)
     IF length of SecondsRemaining >= 2 THEN
          Display.blit(Timer, (0,0))
     ELSE
          Display.blit(Timer, ((c * GridSize / 4,0)))
     ENDIF
     LFont = pygame.font(‘calibri’, (c*2))
     LivesCounter = LFont.Draw(str(Lives), Red)
     Display.blit(LivesCounter, (0,0))
     Draw LOT
     Draw LOS
     Frame += 1
```

```
          Clock.tick(FPS)
     ENDLOOP
     Empty LOS
     Empty LOW
     Empty LOT
     Empty LOM
     AddScore(Name, Score)
END
```

Pseudocode 3.0 Classes –

This part of the pseudocode section (3.x) is dedicated to classes and their methods. The classes covered here include:

1. Block
2. Player
3. Treasure
4. Wall
5. Modifier

Pseudocode 3.1 Block class –

The initialisation takes no parameters and uses no local variables.

```
BEGIN
     CLASS Block
          DEFINE __init__(self)
               self.Row = -1
               self.Column = -1
               self.North = 1
               self.South = 1
               self.West = 1
               self.East = 1
               self.Visited = 0
               self.Distance = 0
               self.SpeedMultiplier = 1
     ENDCLASS
END
```

Pseudocode 3.2 Player class –

The Player class uses the following parameters:

Parameter Name	Data Type	Purpose	Used in
xChange	Float	This is the amount the player moves in the horizontal direction per frame due to player input	RecordMove

yChange	Float	This is the amount the player moves in the vertical direction per frame due to player input	RecordMove
Walls	List	This is the list of walls, used to check for collision between player and walls	UpdatePos
Treasure	List	This is the list of treasures, used to check for collision between player and treasure	UpdatePos
Modifiers	List	This is the list of modifiers, used to check for collision between player and modifiers	UpdatePos
Collision	Boolean	This is the true or false value that allows the function to determine whether the time ran out or whether the player reached the treasure in time	UpdatePos
sPath	Float	This is the shortest path from the player to the treasure	UpdatePos
Score	Integer	This is the score of the player	UpdatePos
GridSize	Integer	Defines the size of the maze that is being generated	UpdatePos
BraidSeverity	Float	Determines the proportion of dead ends that is removed. Lower value means more dead ends removed. This is inversely proportional to the number of dead ends	UpdatePos
TreasureL	Treasure	This is the treasure object. This is required so that the UpdatePosT procedure can be run once a new maze is generated, so that a new treasure location can be generated	UpdatePos

The Player class uses the following local variables:

Variable Name	Data Type	Purpose	Used in
ModifierSum	Float	This is the cumulative effect of all modifiers that the player is currently colliding with. This effect is therefore the overall modification on the player and is used to determine whether the player will slow down, stay the same or speed up.	UpdatePos
SpeedModifier	String	The value of this is dependent on ModifierSum. If ModifierSum is positive, this is ‘Fast’. If ModifierSum is 0, this is ‘Normal’. If ModifierSum is	UpdatePos

		negative, this is ‘Slow’. This is then directly used to change player speed	
PlayerRow	Integer	This stores the row the player is currently in; it is calculated using c and self.rect.x. This is then passed into Dijkstra as the starting row	UpdatePos
PlayerColumn	Integer	This stores the column the player is currently in; it is calculated using c and self.rect.y. This is then passed into Dijkstra as the starting column	UpdatePos

```
BEGIN
    LOS = pygame.sprite.Group()
    CLASS Player(pygameSprite)
        DEFINE __init__(self)
            self.image = pygame.Surface([c / 3, c / 3])
            self.rect = self.image.get_rect()
            self.xMoved = 0
            self.yMoved = 0
            self.Speed = c/12.5
            self.Modified = ‘’
            ADD self to LOS
        ENDDEF
        DEFINE RecordMove(self, xChange, yChange)
            self.xMoved = self.xMoved + xChange
            self.yMoved = self.yMoved + yChange
        ENDDEF

        DEFINE UpdatePos(self, Walls, Treasure, Modifiers,
            Collision, sPath, Score, GridSize,
            BraidSeverity, TreasureL, PlayerL)
            ModifierSum = 0
            SpeedModifier = ‘’
            FOR Modifier collision with self DO
                IF Modifier.Type = ‘Fast’ THEN
                    ModifierSum = ModifierSum + 1
                ELSEIF Modifier.Type = ‘Slow’ THEN
                    ModifierSum = ModifierSum - 1
                ENDIF
            ENDLOOP
            IF ModifierSum > 0 THEN
                SpeedModifier = ‘Fast’
            ELSEIF ModifierSum < 0 THEN
                SpeedModifier = ‘Slow’
            ELSEIF ModifierSum = 0 THEN
                SpeedModifier = ‘Normal’
            ENDIF
            IF SpeedModifier = ‘Fast’ and self.Modified =
            ‘’ THEN
```

```
     self.RecordMove(self.xMoved, self.yMoved)
     self.Modified = ‘Faster’
     self.Speed = self.Speed * 2
ELSEIF SpeedModifier = ‘Slow’ and self.Modified
= ‘’ THEN
     self.RecordMove(-(self.xMoved/2),-
     (self.yMoved/2))
     self.Modified = ‘Slower’
     self.Speed = self.Speed / 2
ENDIF
IF SpeedModifier = ‘Normal’ THEN
     IF self.Modified = ‘Slower’ THEN
          self.RecordMove(self.xMoved,
          self.yMoved)
          self.Speed = self.Speed * 2
     ELSEIF self.Modified = ‘Faster’ THEN
          Self.RecordMove(-self.xMoved/2,-
          self.yMoved/2)
          Self.Speed = self.Speed / 2
     ENDIF
ENDIF

self.rect.x = self.rect.x + self.xMoved
FOR Wall collision with self DO
     IF self.xMoved > 0 THEN
          self.rect.right = Wall.rect.left
     ELSE
          Self.rect.left = Wall.rect.right
     ENDIF
ENDLOOP

self.rect.y = self.rect.y + self.yMoved
FOR Wall collision with self DO
     IF self.yMoved > 0 THEN
          Self.rect.bottom = Wall.rect.top
     ELSE
          Self.rect.top = Wall.rect.bottom
     ENDIF
ENDLOOP

CollisionList = FOR Treasure collision with
self
IF length of CollisionList > 0 or Collision =
True THEN
     IF length of CollisionList > 0 THEN
          Score = Score + 1
          IF not GridSize <= c/2.5 THEN
               GridSize = GridSize + 1
          ENDIF
          IF BraidSeverity < 1 THEN
```

```
        BraidSeverity = BraidSeverity + 0.1
    ENDIF
ELSE
    Score = Score - 1
    IF GridSize > 2 THEN
        GridSize = GridSize - 1
    ENDIF
    IF BraidSeverity > 0.1 THEN
        BraidSeverity = BraidSeverity - 0.1
    ENDIF
ENDIF
IF Collision = True THEN
    IF PlayerL.rect.x > c THEN
        PlayerL.rect.x  = PlayerL.rect.x - c
    ELSEIF PlayerL.rect.y > c THEN
        PlayerL.rect.y = PlayerL.rect.y - c
    ENDIF
ENDIF
TreasureL.UpdatePosT(GridSize, TreasureL.Row, TreasureL.Column)
PlayerColumn = math.floor((PlayerL.rect.x + (c/6)) / c)
PlayerRow = math.floor ((PlayerL.rect.y + (c/6)) / c)
Empty list Walls
Empty list Modifiers
LOB = [Block for block in range(GridSize ** 2)]

IF Score < 5 THEN
    LOB = BinaryTree(GridSize, LOB)
ELSEIF Score >= 5 and Score < 10 THEN
    LOB = Sidewinder(GridSize, LOB)
ELSEIF Score >= 10 THEN
    LOB = AldousBroder(GridSize, LOB)
ENDIF
LOB = Braid(GridSize, LOB, BraidSeverity)
LoadMaze(c, hStart, vStart, LOB, GridSize)
sPath = Dijkstra(LOB, GridSize, PlayerRow, PlayerColumn, TreasureL.Row, TreasureL.Column)
IF IndexError THEN
    RETURN self.UpdatePos(Walls, Treasure, Modifiers, Collision, sPath, Score, GridSize, BraidSeverity, TreasureL, PlayerL)
ENDIF
```

```
                ENDIF
                RETURN Score, Collision, sPath, GridSize,
                BraidSeverity
        ENDDEF
END
```

Pseudocode 3.3 Treasure class –

The Treasure class uses the following parameters:

Parameter Name	Data Type	Purpose	Used in
GridSize	Integer	Defines the size of the maze that is being generated	UpdatePosT
OldRow	Integer	Stores the old row of the treasure, this is saved to ensure that the new treasure does not spawn in the same location	UpdatePosT
OldColumn	Integer	Stores the old column of the treasure, this is saved to ensure that the new treasure does not spawn in the same location	UpdatePosT

The Treasure class uses the following local variables:

Variable Name	Data Type	Purpose	Used in
Image	pygame.Surface	This is used to load in the treasure chest png file	__init__

```
BEGIN
    CLASS Treasure(pygameSprite)
        LOT = pygame.sprite.Group()
        DEFINE __init__(self)
            Image = Load("treasure.png")
            self.Row = random integer 0 to GridSize - 1
            self.Column = random integer 0 to GridSize - 1
            self.image = scale(Image, (c // 3, c // 3))
            self.rect = self.image.get_rect()
            self.rect.x = (self.Column * c) + c // 3
            self.rect.y = (self.Row * c) + c // 3
            ADD self to LOT
        ENDDEF

        DEFINE UpdatePosT(self, GridSize, OldRow, OldColumn)
            self.Row = random integer 0 to GridSize - 1
            self.Column = random integer 0 to GridSize - 1
            WHILE self.Row = OldRow DO
```

```
                    Self.Row = random integer 0 to GridSize -
                    1
               ENDLOOP
               WHILE self.Column = OldColumn DO
                    self.Column = random integer 0 to GridSize
                    - 1
               ENDLOOP
          ENDDEF
END
```

Pseudocode 3.4 Wall class –

The wall class uses the following parameters:

Parameter Name	Data Type	Purpose	Used in
Width	Integer	This is the width of the wall sprite. This may be large or small dependent on whether the wall is a vertical wall or a horizontal wall	__init__
Height	Integer	This is the height of the wall sprite. This may be large or small dependent on whether the wall is a vertical wall or a horizontal wall	__init__
xVertice	Integer	This is used to define self.rect.x, which is the horizontal co-ordinate of the top left corner of the sprite	__init__
yVertice	Integer	This is used to define self.rect.y, which is the vertical co-ordinate of the top left corner of the sprite	__init__

The wall class uses no local variables

```
BEGIN
     CLASS Wall(pygameSprite)
          LOW = pygame.sprite.Group()
          DEFINE __init__(self, width, height, xVertice,
          yVertice)
               self.image = pygame.Surface([width, height])
               self.Width = width
               self.Height = height
               self.rect = self.image.get_rect()
               self.rect.x = xVertice
               self.rect.y = yVertice
               ADD self to LOW
          ENDDEF
END
```

Pseudocode 3.5 Modifier class –

The modifier class uses the following parameters:

Parameter Name	Data Type	Purpose	Used in
Size	Integer	This is the width/height of the modifier sprite. The sprite is a square so the width = height.	__init__
xVertice	Integer	This functions the same as in the wall class	__init__
yVertice	Integer	This functions the same as in the wall class	__init__
Type	String	This defines the type of modifier the object is, either a fast or slow one.	__init__

The modifier class uses no local variables

```
BEGIN
     CLASS Modifier(pygameSprite)
          LOM  = pygame.sprite.Group()
          DEFINE __init__(self, size, xVertice, yVertice,
          type)
               self.image = pygame.Surface([size, size])
               self.Type = type
               IF type = 'Slow' THEN
                    self.image.fill(Brown)
               ELSEIF type = 'Fast' THEN
                    self.image.fill(Blue)
               ENDIF
               self.rect = self.image.get_rect()
               self.rect.x = xVertice
               self.rect.y = yVertice
               ADD self to LOM
          ENDDEF
END
```

Libraries

Library	Reason for using
Random	Used to generate random integers in maze generation algorithms. These random integers are used to determine which direction to carve, as well as deciding whether or not a block will spawn a modifier on it.
pygame	Used to create and display the playable environment, allows for the creation of player, treasure, wall and modifier sprites. Also allows collision to be implemented between the player, treasure, wall and modifier objects. Calculates time given to the player via frames, tracks lives and displays these two variables.

csv	Used to track highscores. Highscores are written to a csv file and read from a csv file if the player chooses this option.
datetime	Used to find the current date, and this date is then written to the highscores file alongside the player name and score.
math	Used for rounding purposes

Technical Solution

System overview

In this section I will describe the purpose and function of the classes, functions, methods and procedures shown in the pseudocode section.

System overview 1.0 Maze generation –

Each maze generation algorithm contains a possibility of spawning modifiers. The functionality of this is identical in each algorithm. DeciderSpeed is a random integer 1 to 20. If this value is less than 3, a slow modifier is created, and a fast one is created for a value greater than 18. This presents a 10% chance of a slow modifier spawning, and a 10% chance of a fast modifier spawning on each block of the maze. The modifier objects are not actually created in these functions, they are created alongside walls later in the program, the co-ordinates of those modifier objects are based on the row and column of they block they are in.

Additionally, for each maze generation function, the walls between blocks are changed by altering the attributes of the Block objects in the LOB (list of blocks). This LOB list is then returned at the end of the function, which describes the maze that has been generated.

System overview 1.1 Binary Tree –

This is the first maze generation algorithm that is used in the game. The algorithm runs on two loops. The first loop goes through each Row in the range of 0 to GridSize; the second loop goes through each Column in the range of 0 to GridSize. With these two loops running together, every Row and Column combination can be processed. A counter called 'Index' is increased by 1 for each Row and Column combination. The value of this Index at any particular iteration describes the Index of the block that occupies that row and column, this index can then be used to find the adjacent blocks to any block in the maze.

The top left block is index 0, the index follows a left to right, top to bottom pattern, similar to reading a book. This is useful because finding adjacent blocks is made easy. For instance, the block to the right of the current one with an index of Index, has an index of Index + 1. A block above the current block with index of Index, has an index of Index – GridSize.

The variable of Decider determines which direction to carve. In this algorithm, every block is visited and from each block the algorithm carves either north or east. A Decider of 0 indicates to carve north, and a value of 1 indicates to carve east. This value must be validated on the northern boundary and eastern boundary, where only 1 option is possible. On the eastern boundary, i.e. Column = GridSize – 1, the Decider must have a value of 0, and the opposite is true for the northern boundary.

System overview 1.2 Sidewinder –

This is the second maze generation algorithm that is used in the game. The algorithm also runs through two loops exactly like Binary Tree. This, alongside with the Index variable, functions the exact same in both of these functions.

The difference between the two functions is that this one uses another variable, 'Run', to make the final maze less biased. Each time a block is carved east from, the index of that block is appended to Run. Once the decider happens to become 0, a random index from Run is chosen, and the function carves north from that block. The difference here is seen on the eastern boundary where although the decider must be equal to 0 as in Binary Tree, this does not guarantee that a northern wall is broken on the eastern boundary. This avoids creating an unbroken column on the eastern boundary, making the maze harder to traverse. For a visual example, view Research 1.2 and Research 1.1 to view the difference.

System overview 1.3 AldousBroder –

This is the third and final maze generation algorithm that is used in the game. This is quite different from the last two algorithms. There are two main differences:

The first difference is that this does not loop through each column and each row. The method of visiting blocks is slightly different. The algorithm begins on any block, and the block that is visited after the current one is the one that is carved to, whereas previously Index + 1 would be the index of the next block carved to. This achieved through a while loop which terminates once all blocks in the maze have been visited. As a result of this, some blocks may be visited multiple times, making this algorithm take far longer, although it is much closer to being random than the previous two algorithms.

The second difference is that this algorithm carves in four directions as opposed to two. The value of decider can take on four different values, each value for each direction that it can carve. When the algorithm chooses a new block to carve to, it will only destroy the wall between the current block and new block if the new block has not been visited yet. This avoids creating too many passages through the maze. The drawback of this is that the time required to complete maze generation is greater than the other two algorithms and increases in proportion to GridSize at a greater rate. Exactly GridSize * GridSize number of blocks need to be visited in Binary Tree and Sidewinder, while Aldous Broder will almost always visit the same block many times over.

System overview 1.4 Braiding –

The braiding function takes a maze (LOB) as an input and returns a maze with less dead ends. The proportion of dead ends removed is inversely proportional to BraidSeverity. This is implemented with a Decider variable. The Decider is the inverse of BraidSeverity, (1 / BraidSeverity) if the index of a block is divisible by Decider, then the block does not have its dead end removed (if it has one in the first place). For instance, a BraidSeverity of 0.5 would result in a Decider of 2. Exactly half of the blocks in the maze will therefore have their dead end removed if they have one. The value of BraidSeverity is the approximate proportion of remaining dead ends in the maze once the braiding algorithm is complete. It is approximate because each block that is chosen to have its dead end removed may not have a dead end in the first place. It could be that a BraidSeverity of 0.5 results in no dead ends being removed (if the dead ends are all on even indexed blocks) although this is extremely unlikely. On larger mazes, this algorithm becomes more consistent.

This algorithm loops through each index in the range of 0 to GridSize * GridSize. Each index that is not divisible by decider, the number of walls that still exist in that block are counted. If this value is 3, that means the block is a dead end. A random direction in the Ends list is selected, and this wall is removed once boundary conditions are considered, e.g. a block on the northern boundary that has 3 ends cannot have its northern wall removed, as that is the edge of the maze.

System overview 1.5 Dijkstra's –

The function begins by setting the Visited attribute of each block to 0 and the Distance attribute of each block to (GridSize * GridSize) * 2. The distance of (GridSize * GridSize) * 2 is the maximum possible distance that two blocks can be from each other in a square maze of size GridSize (Brackets are doubled because slow modifiers half player speed), which is why it is the default distance for every block.

The initial index is the index of the player sprite at the start of the level. This index is calculated by multiplying PlayerRow by GridSize and adding PlayerColumn. The index is added to the Frontier list. The Frontier list is a list of indexes that are on the ends of the search, i.e. they are blocks that were most recently visited. The while loop runs until the VisitedCount is not less than the length of LOB (total amount of blocks), or until the function returns a value for sPath.

The loop begins by creating an empty list ToVisit. Each Index in Frontier is considered, any block that has a connection to the block at Index is added to ToVisit, alongside with the distance of the block at Index from the starting block. This is appended to ToVisit in the form of a list, where the first element is the index of the block to be visited, and the second element is the Distance attribute of the block at Index in Frontier that is being considered. After this, the Frontier list is emptied.

A new loop is run, and each IndexList in ToVisit is considered. If the block with an Index of IndexList[0] has a Distance attribute greater than the distance of (IndexList[1] + Weight), then the index of this block (IndexList[0]) is appended to Frontier, and the Distance attribute of this block is changed to (IndexList[1] + Weight). If this block has not been visited before the current iteration, VisitedCount is increased by 1. Additionally, if this block has Row and Column attributes equal to the goal row and goal column, then Dijkstra's stops and returns the Distance attribute of this block, since this distance is the distance from the player to the treasure.

The loop described in the above paragraph is actually run 3 times. The first time it is run it only considers connections with the smallest weight (weight = 0.5), this is blocks that have a fast modifier spawned in them (SpeedMultiplier = 2). The second time it is run it only considers connections with medium weight (weight = 1), this is blocks with no modifier.

Finally, the third loop checks all connections with the highest weight (weight = 2), this is blocks that have a slow modifier spawned in them (SpeedMultiplier = 0.5). This is done because Dijkstra's algorithm must consider the connections with the smallest weight/cost first.

Finally, the end of the function returns the Distance attribute of the block at index (gRow * GridSize) + gColumn in LOB. This ultimately should never be run, as the previous 3 loops return this value if the index is ever checked. This is mostly just a redundancy in the code in case the previous 3 return statements do not return something.

System overview 1.6 LoadMaze –

This procedure instantiates wall and modifier objects but does not save them directly or return them. This is because the instantiation of these objects places them in their respective pygame group lists already. This being LOM for modifiers and LOW for walls. They are only required in these lists for the purpose of player collision and can be found via their index in the list.

This is run as soon as a new maze is generated, to instantiate every wall and modifier that will be required to be displayed shortly after.

System overview 2.0 Game logic –

This section is dedicated to elaborating on the functions and procedures shown in Pseudocode 2.x.

System overview 2.1 Menu –

This function simply prints a menu with 3 options, play game, view highscores and exit. The function returns the player choice. This is the first function that is run in the program.

System overview 2.2 ShowHighscores –

This procedure is run if the player selects option 2 from the menu, the purpose of this procedure is to output the 10 highest scores in the highscores.txt file.

No sorting is done in this procedure, as the highscore.txt file is already sorted in the AddScore procedure. The algorithm simply reads the first 10 rows after the initial row, and outputs them in an ordered list.

System overview 2.3 AddScore –

This procedure reads the csv file highscores.txt using DictReader which splits each row into a dictionary based on where the delimiter ',' is placed in the row. The list highscores is created, which is a list of lists. The score of the player, the name of the player and the date/time is appended to the Highscores list as a list, with the elements in order as listed previously. Then, every row that is read from the csv file is also appended to Highscores. Finally, the data in the highscores.txt file is overwritten with the current Highscores list, with each nested list being a new row in the file.

System overview 2.4 Game –

This procedure is the main game and is called if the player selects option 1 in the menu. This calls all other functions and provides a framework, while also handling player movement and player statistics such as lives and score. All of the pygame drawing of sprites also happens in this procedure.

The first step is to create all the variables required before the main game loop begins, this includes:

1. GridSize
2. BraidSeverity
3. Difficulty
4. LOB
5. FPS
6. Frame
7. Clock
8. Width
9. Height
10. Display
11. PlayerL
12. TreasureL
13. Row
14. Column
15. sPath
16. TimeFrames
17. TimeSeconds
18. Ease
19. TargetFrame
20. SecondsRemaining
21. CollisionFrame
22. CollisionCheck
23. Score
24. Completed
25. Lives
26. Running

The function and purpose of all of these variables is discussed in Proposed Solution 1.2.

Next, the main game loop is begun which is repeated FPS times per second, which runs until Running is equal to False. At the beginning of the loop, any key inputs from the player are checked for, if any are found, the required amount of player speed is added or subtracted from the players horizontal and/or vertical velocity.

Next, the function UpdatePos is called. This function handles player collision with modifiers, walls and adjusts player velocity accordingly. It also checks for player collision with the treasure, where a new maze is generated if a collision is detected. UpdatePos also returns the shortest path from player to treasure when a new maze is generated.

UpdatePos returns CollisionCheck, if this is True then player statistics and maze generation settings are adjusted, e.g. Completed, Lives, Ease etc. After this, an if statement checks if the SecondsRemaining variable needs to be adjusted. This is adjusted if the current number of remaining frames is divisible by FPS, meaning an integer number of seconds is left, and updates the visible timer on screen.

Another if statement checks whether the current frame is greater than the TargetFrame (TargetFrame is frame at which players time runs out), if so, then the player has lost the level and player statistics and maze generation settings are adjusted. UpdatePos is called once again, but the value of CollisionCheck passed into it is set to True, meaning a new maze will be generated and all the necessary calculations are made even without a player and treasure collision. If Lives is equal to 0, Running is set to False

The remaining code in the loop mostly handles drawing, this includes:

1. Walls
2. Modifiers
3. Timer
4. LivesCounter
5. Treasure
6. Player

The final line in the loop makes the pygame Clock object move on to the next frame (another iteration of the loop).

Once Running is False, the loop no longer runs again and LOS, LOW, LOT and LOM are emptied, allowing a new game to start if the player wishes to play a new game.

System overview 3.0 Class methods –

This section is dedicated to elaborating on the methods used in classes shown in Pseudocode 3.x, this includes:

1. RecordMove
2. UpdatePos
3. UpdatePosT

System overview 3.1 RecordMove –

This method is called when a player key press is detected in the main game loop. This method simply adds the specified change in x and change in y to the players distance moved in one frame (self.xMoved and self.yMoved). It is also called if a collision with a modifier is detected, altering player distance moved in one frame to be consistent with the modifier that has been collided with.

System overview 3.2 UpdatePos –

This method handles player collision, and the generation of new mazes once the player either runs out of time or collides with the treasure. Initially, collisions with modifiers are checked, and the player speed is adjusted if any collisions are made. Next, collisions with walls are checked and player position (self.rect) is changed if any collisions are made. Modifier collisions must be considered before wall collisions because otherwise, a collision with a wall may not be detected, but once the modifier collision is accounted for, the player will collide with a wall, causing the player and wall sprite to overlap in this frame. Next, collisions with the treasure are checked, if a collision is found, Score is increased by 1, GridSize is increased by 1 if it is not greater than c/2.5 (this value was chosen as a value any higher would cause the maze to be larger than the screen allows to be displayed with the given c), and BraidSeverity is increased by 0.1 if it is less than 1.

Next, UpdatePosT is run to generate a new position for the treasure. Walls (LOW) and Modifiers (LOM) are emptied and a new maze is generated. This maze then has braiding applied, the wall and modifier objects are loaded with LoadMaze, and finally Dijkstra is called to calculate the new shortest path from the player to the treasure.

UpdatePos takes Collision as a parameter, this is set to True in the main game loop if the player has run out of time, otherwise it is False. In this method, the if statement that creates a new level (generates new maze, new objects etc) as described in the above paragraph, has two conditions (will trigger if either condition is True), the first condition was already discussed in the first paragraph and is when the player has collided with the treasure. The second condition is if Collision is equal to True. If this is the case, the player statistics and maze generation settings are altered in the opposite way as described in the first paragraph where the first condition was true. Additionally, since the size of the maze is decreasing, there is a possibility that the player location may be outside the maze once the new maze is generated. Because of this, the player position must be altered to ensure the player does not go out of bounds. The maze size both vertically and horizontally decreases by c, so the player position is also decreased by c. This ensures the player is always kept within the maze, even though maze size may be changing.

System overview 3.3 UpdatePosT –

This method deals with the random generation of the treasure row and column. A new row and column are generated continuously until they are not equal to the old row and old column of the treasure. The treasure location (self.rect) is then altered in accordance to the new row and new column of the treasure.

Key code

Key code 1.0 Maze generation –

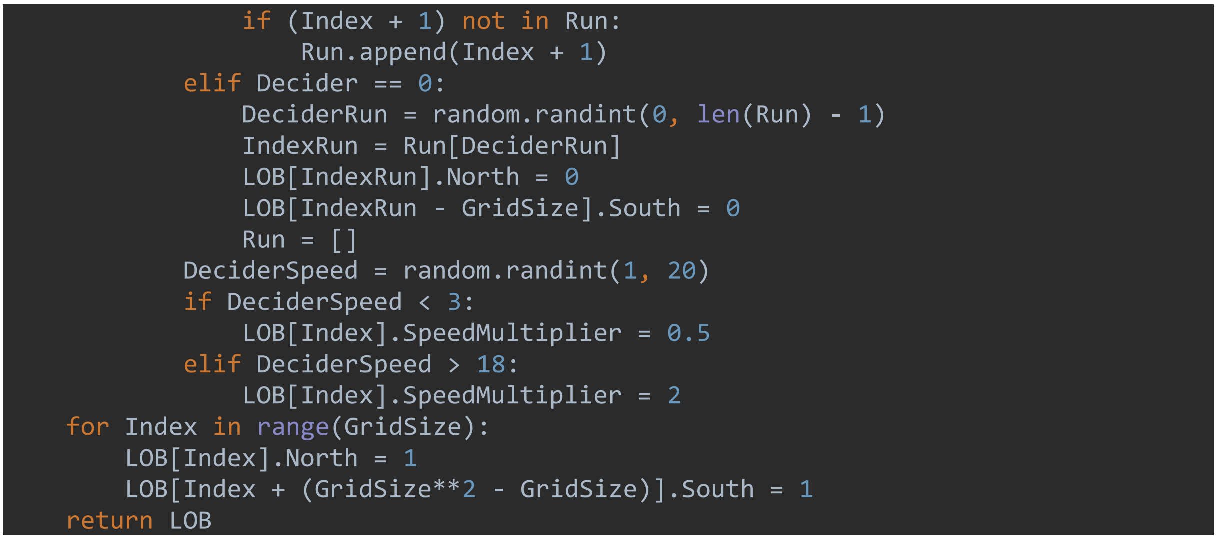
Below I have displayed the finished code for all the pseudocode shown in Pseudocode 1.x

Key code 1.1 Binary Tree –

```
def BinaryTree(GridSize, LOB):
    Index = -1
    for Row in range(GridSize):
        for Column in range(GridSize):
            Index += 1
            LOB[Index].Row = Row
            LOB[Index].Column = Column
            Decider = random.randint(0, 1)  # 0 is north, 1 is east
            if Row == 0 and Column == GridSize - 1:
                continue
            elif Row == 0:
                Decider = 1
            elif Column == GridSize - 1:
                Decider = 0
            if Decider == 0:
                LOB[Index].North = 0
                LOB[Index - GridSize].South = 0
            elif Decider == 1:
                LOB[Index].East = 0
                LOB[Index + 1].West = 0
            DeciderSpeed = random.randint(1, 20)
            if DeciderSpeed < 3:
                LOB[Index].SpeedMultiplier = 0.5
            elif DeciderSpeed > 18:
                LOB[Index].SpeedMultiplier = 2
    return LOB
```

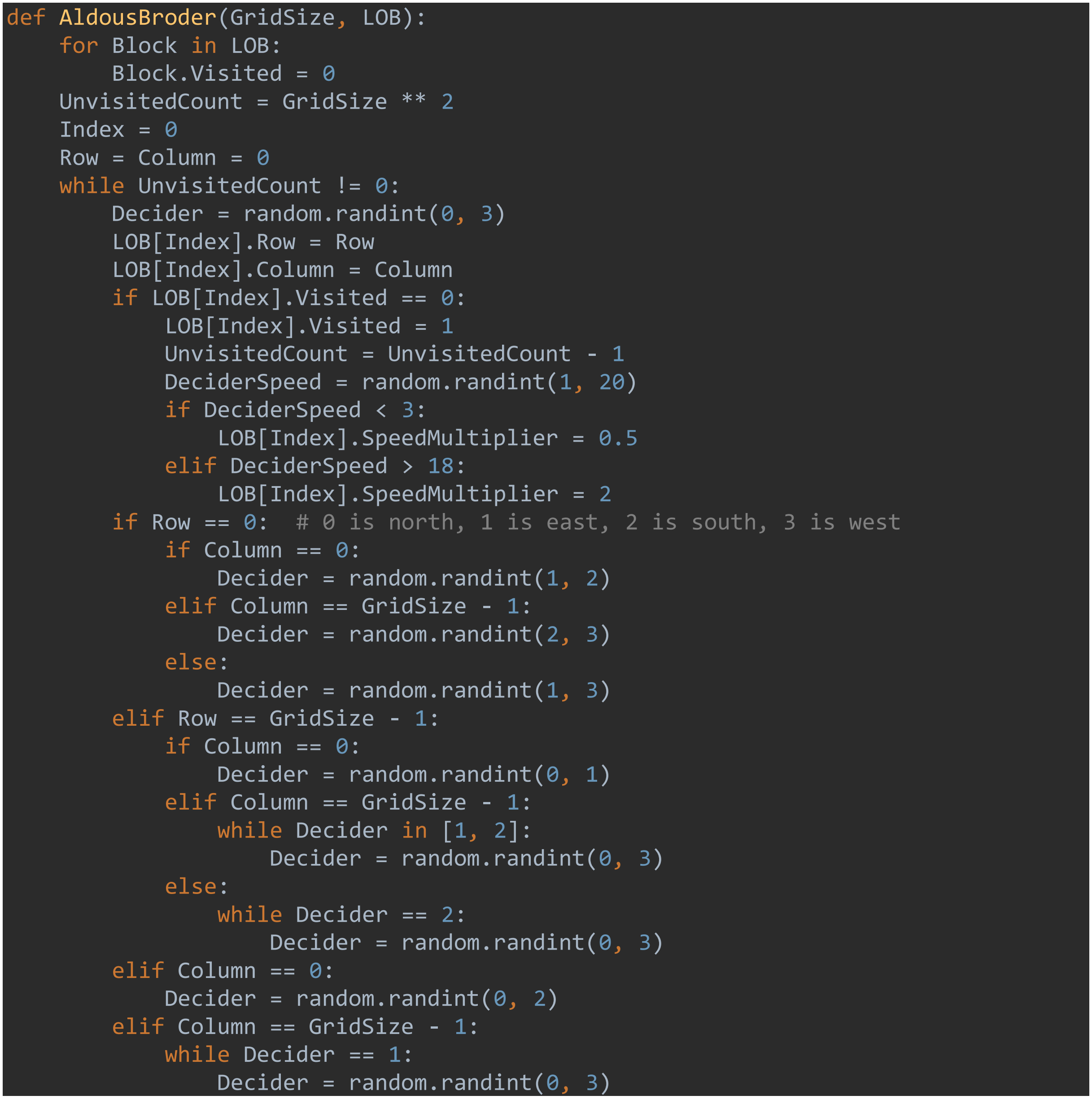
Key code 1.2 Sidewinder –

```
def Sidewinder(GridSize, LOB):
    Index = -1
    Run = []
    for Row in range(GridSize):
        for Column in range(GridSize):
            Index += 1
            Run.append(Index)
            LOB[Index].Row = Row
            LOB[Index].Column = Column
            Decider = random.randint(0, 1)  # 0 is north, 1 is east
            if Row == 0 and Column == GridSize - 1:
                continue
            elif Row == 0:
                Decider = 1
            elif Column == GridSize - 1:
                Decider = 0
            if Decider == 1:
                LOB[Index].East = 0
                LOB[Index + 1].West = 0
```



```
                if (Index + 1) not in Run:
                    Run.append(Index + 1)
            elif Decider == 0:
                DeciderRun = random.randint(0, len(Run) - 1)
                IndexRun = Run[DeciderRun]
                LOB[IndexRun].North = 0
                LOB[IndexRun - GridSize].South = 0
                Run = []
            DeciderSpeed = random.randint(1, 20)
            if DeciderSpeed < 3:
                LOB[Index].SpeedMultiplier = 0.5
            elif DeciderSpeed > 18:
                LOB[Index].SpeedMultiplier = 2
    for Index in range(GridSize):
        LOB[Index].North = 1
        LOB[Index + (GridSize**2 - GridSize)].South = 1
    return LOB
```

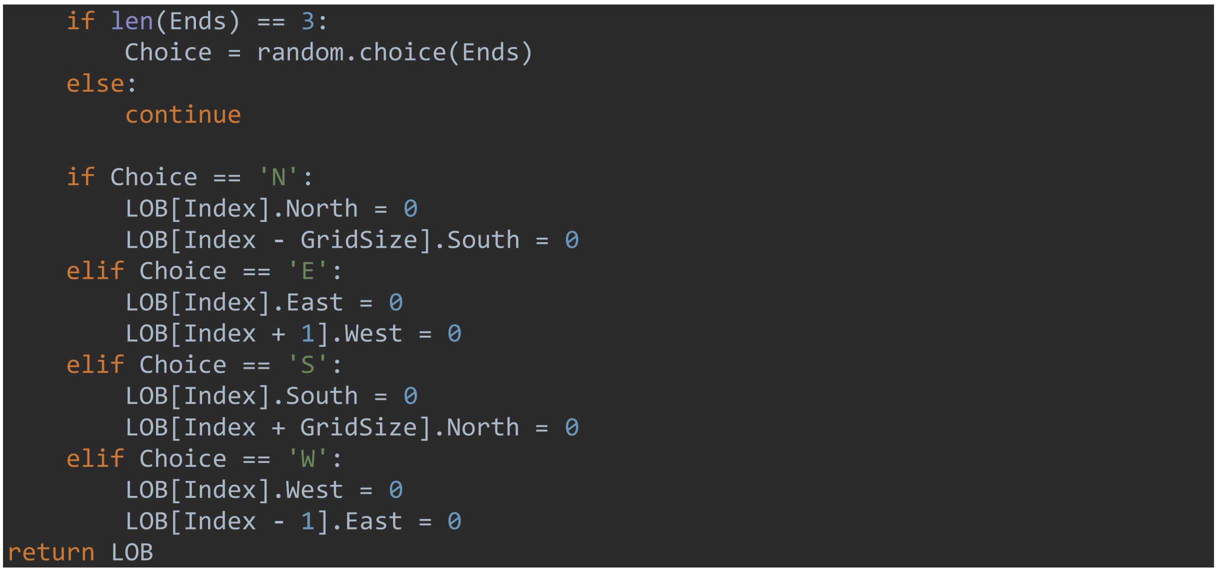
Key code 1.3 Aldous Broder –



```
def AldousBroder(GridSize, LOB):
    for Block in LOB:
        Block.Visited = 0
    UnvisitedCount = GridSize ** 2
    Index = 0
    Row = Column = 0
    while UnvisitedCount != 0:
        Decider = random.randint(0, 3)
        LOB[Index].Row = Row
        LOB[Index].Column = Column
        if LOB[Index].Visited == 0:
            LOB[Index].Visited = 1
            UnvisitedCount = UnvisitedCount - 1
            DeciderSpeed = random.randint(1, 20)
            if DeciderSpeed < 3:
                LOB[Index].SpeedMultiplier = 0.5
            elif DeciderSpeed > 18:
                LOB[Index].SpeedMultiplier = 2
        if Row == 0:  # 0 is north, 1 is east, 2 is south, 3 is west
            if Column == 0:
                Decider = random.randint(1, 2)
            elif Column == GridSize - 1:
                Decider = random.randint(2, 3)
            else:
                Decider = random.randint(1, 3)
        elif Row == GridSize - 1:
            if Column == 0:
                Decider = random.randint(0, 1)
            elif Column == GridSize - 1:
                while Decider in [1, 2]:
                    Decider = random.randint(0, 3)
            else:
                while Decider == 2:
                    Decider = random.randint(0, 3)
        elif Column == 0:
            Decider = random.randint(0, 2)
        elif Column == GridSize - 1:
            while Decider == 1:
                Decider = random.randint(0, 3)
        AdjacentIndex = 0
        if Decider == 0:
            AdjacentIndex = Index - GridSize
            NewRow = Row - 1
            if LOB[AdjacentIndex].Visited == 0:
                LOB[Index].North = 0
                LOB[AdjacentIndex].South = 0
            Row = NewRow
        elif Decider == 1:
            AdjacentIndex = Index + 1
            NewColumn = Column + 1
            if LOB[AdjacentIndex].Visited == 0:
                LOB[Index].East = 0
                LOB[AdjacentIndex].West = 0
            Column = NewColumn
        elif Decider == 2:
            AdjacentIndex = Index + GridSize
            NewRow = Row + 1
            if LOB[AdjacentIndex].Visited == 0:
                LOB[Index].South = 0
                LOB[AdjacentIndex].North = 0
            Row = NewRow
        elif Decider == 3:
            AdjacentIndex = Index - 1
            NewColumn = Column - 1
            if LOB[AdjacentIndex].Visited == 0:
                LOB[Index].West = 0
                LOB[AdjacentIndex].East = 0
            Column = NewColumn
        Index = AdjacentIndex
    return LOB
```

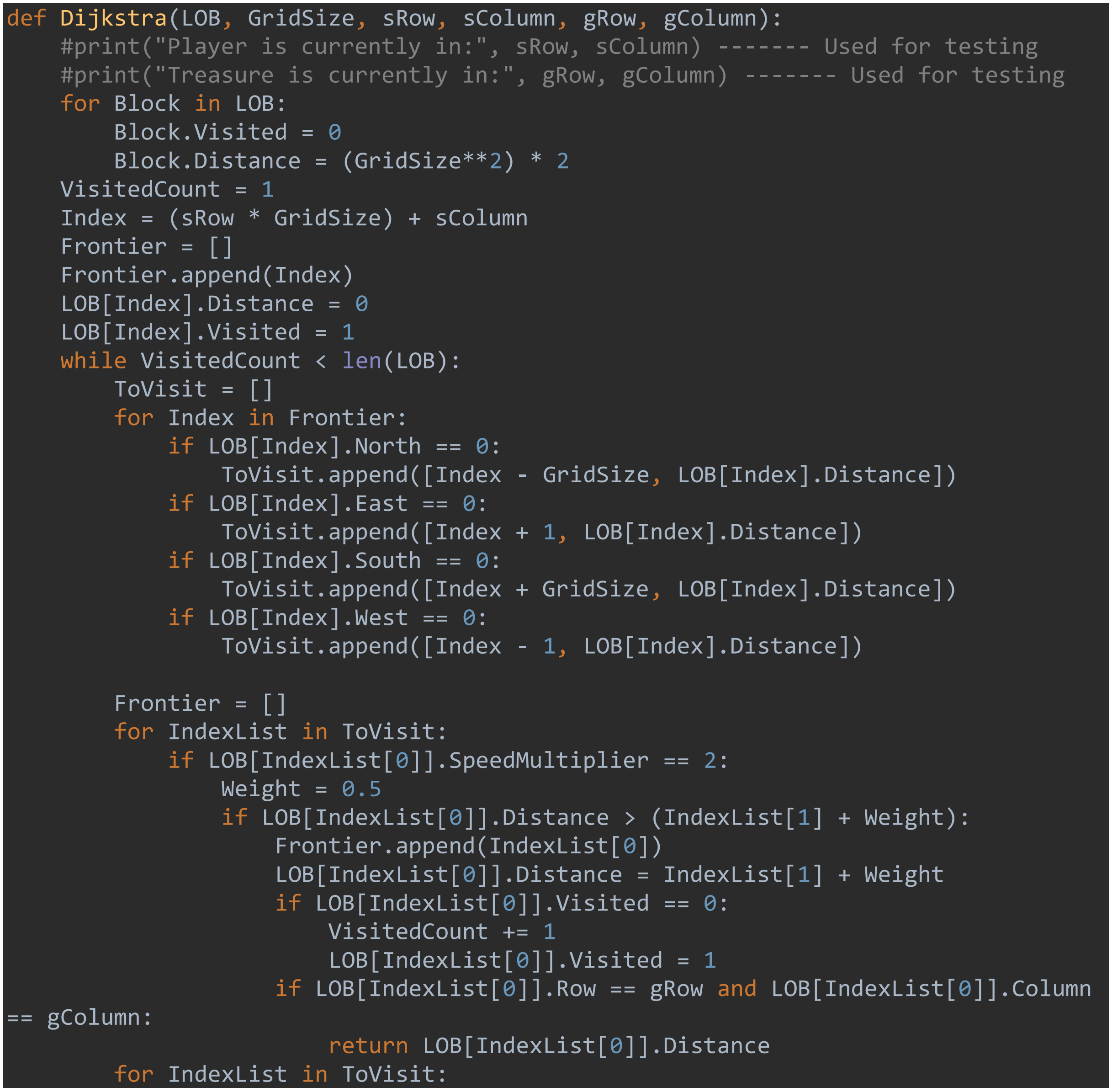
Key code 1.4 Braiding –

```
Decider = 1 / BraidSeverity

for Index in range(0, GridSize**2):
    Ends = []
    if Index % Decider == 0:
        continue
    if LOB[Index].North == 1:
        Ends.append('N')
    if LOB[Index].East == 1:
        Ends.append('E')
    if LOB[Index].South == 1:
        Ends.append('S')
    if LOB[Index].West == 1:
        Ends.append('W')

    if LOB[Index].Row == 0:
        Ends.remove('N')
    if LOB[Index].Row == GridSize - 1:
        Ends.remove('S')
    if LOB[Index].Column == 0:
        Ends.remove('W')
    if LOB[Index].Column == GridSize - 1:
        Ends.remove('E')
```



```
    if len(Ends) == 3:
        Choice = random.choice(Ends)
    else:
        continue

    if Choice == 'N':
        LOB[Index].North = 0
        LOB[Index - GridSize].South = 0
    elif Choice == 'E':
        LOB[Index].East = 0
        LOB[Index + 1].West = 0
    elif Choice == 'S':
        LOB[Index].South = 0
        LOB[Index + GridSize].North = 0
    elif Choice == 'W':
        LOB[Index].West = 0
        LOB[Index - 1].East = 0
return LOB
```

Key code 1.5 Dijkstra's –



```
def Dijkstra(LOB, GridSize, sRow, sColumn, gRow, gColumn):
    #print("Player is currently in:", sRow, sColumn) ------- Used for testing
    #print("Treasure is currently in:", gRow, gColumn) ------- Used for testing
    for Block in LOB:
        Block.Visited = 0
        Block.Distance = (GridSize**2) * 2
    VisitedCount = 1
    Index = (sRow * GridSize) + sColumn
    Frontier = []
    Frontier.append(Index)
    LOB[Index].Distance = 0
    LOB[Index].Visited = 1
    while VisitedCount < len(LOB):
        ToVisit = []
        for Index in Frontier:
            if LOB[Index].North == 0:
                ToVisit.append([Index - GridSize, LOB[Index].Distance])
            if LOB[Index].East == 0:
                ToVisit.append([Index + 1, LOB[Index].Distance])
            if LOB[Index].South == 0:
                ToVisit.append([Index + GridSize, LOB[Index].Distance])
            if LOB[Index].West == 0:
                ToVisit.append([Index - 1, LOB[Index].Distance])

        Frontier = []
        for IndexList in ToVisit:
            if LOB[IndexList[0]].SpeedMultiplier == 2:
                Weight = 0.5
                if LOB[IndexList[0]].Distance > (IndexList[1] + Weight):
                    Frontier.append(IndexList[0])
                    LOB[IndexList[0]].Distance = IndexList[1] + Weight
                    if LOB[IndexList[0]].Visited == 0:
                        VisitedCount += 1
                        LOB[IndexList[0]].Visited = 1
                    if LOB[IndexList[0]].Row == gRow and LOB[IndexList[0]].Column
== gColumn:
                        return LOB[IndexList[0]].Distance
        for IndexList in ToVisit:
            if LOB[IndexList[0]].SpeedMultiplier == 1:
                Weight = 1
                if LOB[IndexList[0]].Distance > (IndexList[1] + Weight):
                    Frontier.append(IndexList[0])
                    LOB[IndexList[0]].Distance = IndexList[1] + Weight
                    if LOB[IndexList[0]].Visited == 0:
                        VisitedCount += 1
                        LOB[IndexList[0]].Visited = 1
                    if LOB[IndexList[0]].Row == gRow and LOB[IndexList[0]].Column
== gColumn:
                        return LOB[IndexList[0]].Distance
        for IndexList in ToVisit:
            if LOB[IndexList[0]].SpeedMultiplier == 0.5:
                Weight = 2
                if LOB[IndexList[0]].Distance > (IndexList[1] + Weight):
                    Frontier.append(IndexList[0])
                    LOB[IndexList[0]].Distance = IndexList[1] + Weight
                    if LOB[IndexList[0]].Visited == 0:
                        VisitedCount += 1
                        LOB[IndexList[0]].Visited = 1
                    if LOB[IndexList[0]].Row == gRow and LOB[IndexList[0]].Column
== gColumn:
                        return LOB[IndexList[0]].Distance
    return LOB[(gRow * GridSize) + gColumn].Distance
```

Key code 1.6 LoadMaze –

```
def LoadMaze(c, hStart, vStart, LOB, GridSize):
    for Index in range(0, GridSize ** 2):
        if LOB[Index].North == 1:
            WallObj = Wall(c + 1, 1, hStart, vStart)
        if LOB[Index].West == 1:
            WallObj = Wall(1, c, hStart, vStart)
        if Index >= (GridSize**2 - GridSize):
            WallObj = Wall(c + 1, 1, hStart, vStart + c)
        if LOB[Index].SpeedMultiplier == 0.5:
            ModifierObj = Modifier(40, hStart + 5, vStart + 5, 'Slow')
        elif LOB[Index].SpeedMultiplier == 2:
            ModifierObj = Modifier(40, hStart + 5, vStart + 5, 'Fast')
        hStart += c
        if (Index + 1) % GridSize == 0:
            WallObj = Wall(1, c, hStart, vStart)
            hStart = 0
            vStart += c
```

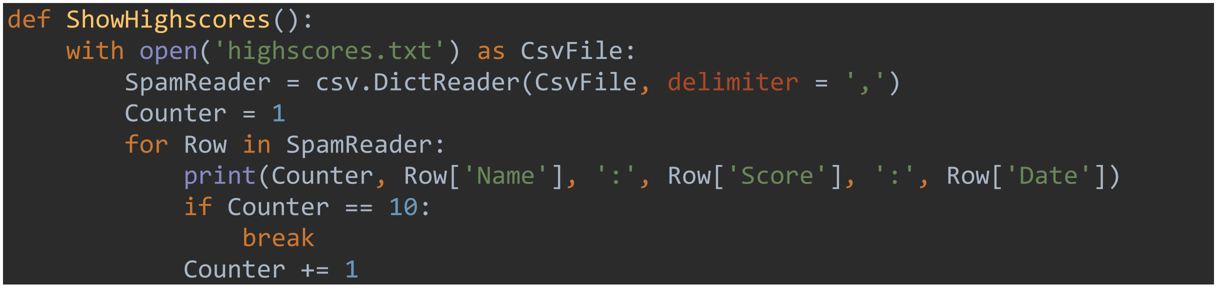
Key code 2.0 Game logic –

Below I have displayed the finished code for all the pseudocode shown in Pseudocode 2.x

Key code 2.1 Menu –

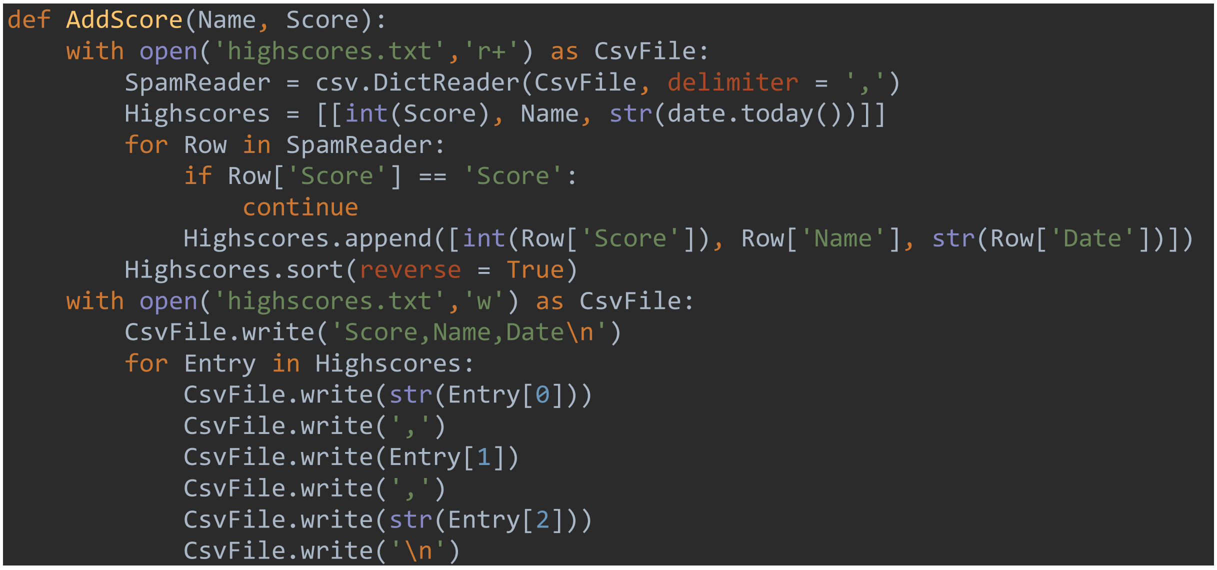
```
def Menu():
    print("----------MENU----------")
    print("1. Play Game")
    print("2. View Highscores")
    print("3. Exit")
    Choice = input("Select option using number")
    return Choice
```

Key code 2.2 ShowHighscores –



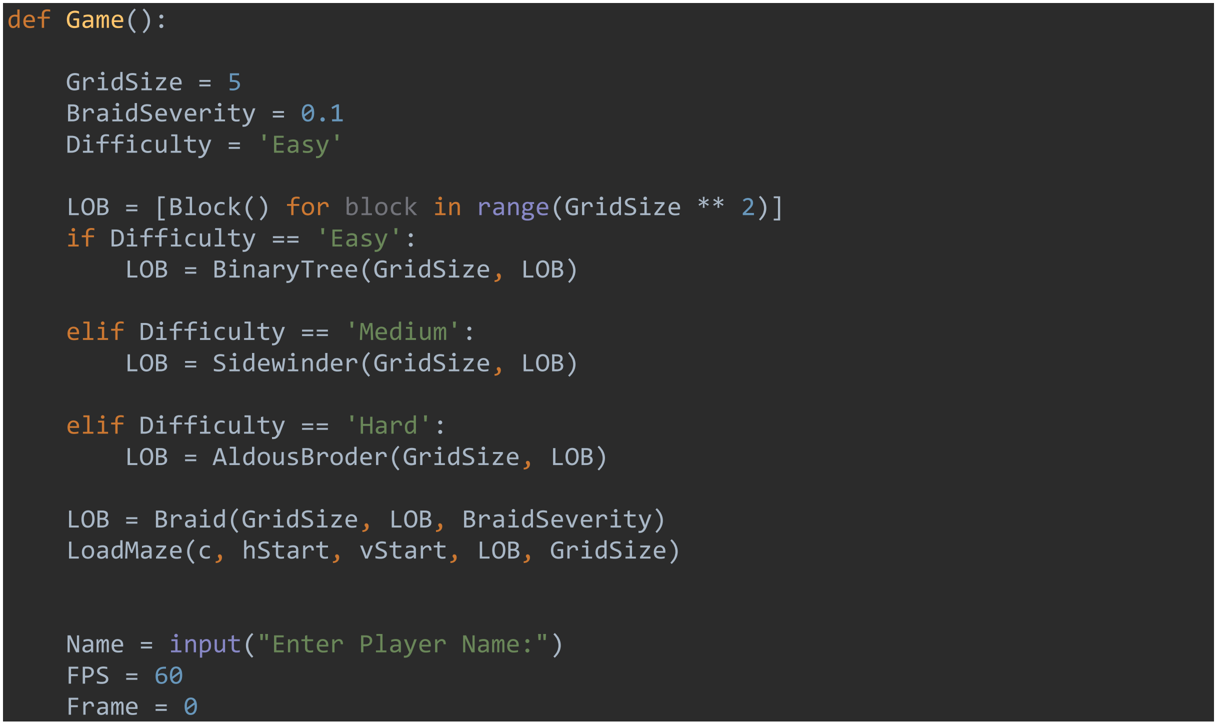
```
def ShowHighscores():
    with open('highscores.txt') as CsvFile:
        SpamReader = csv.DictReader(CsvFile, delimiter = ',')
        Counter = 1
        for Row in SpamReader:
            print(Counter, Row['Name'], ':', Row['Score'], ':', Row['Date'])
            if Counter == 10:
                break
            Counter += 1
```

Key code 2.3 AddScore –

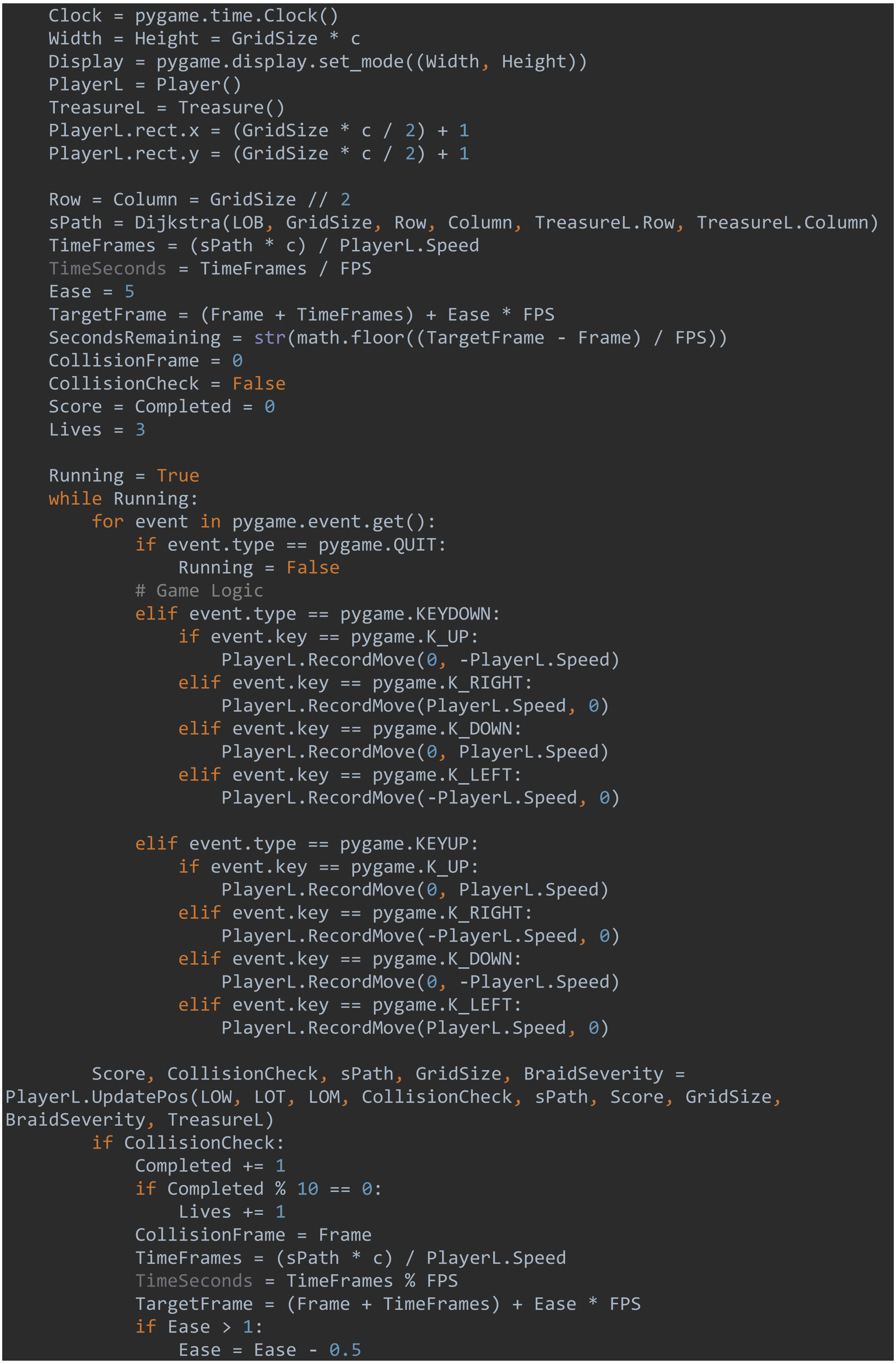


```
def AddScore(Name, Score):
    with open('highscores.txt','r+') as CsvFile:
        SpamReader = csv.DictReader(CsvFile, delimiter = ',')
        Highscores = [[int(Score), Name, str(date.today())]]
        for Row in SpamReader:
            if Row['Score'] == 'Score':
                continue
            Highscores.append([int(Row['Score']), Row['Name'], str(Row['Date'])])
        Highscores.sort(reverse = True)
    with open('highscores.txt','w') as CsvFile:
        CsvFile.write('Score,Name,Date\n')
        for Entry in Highscores:
            CsvFile.write(str(Entry[0]))
            CsvFile.write(',')
            CsvFile.write(Entry[1])
            CsvFile.write(',')
            CsvFile.write(str(Entry[2]))
            CsvFile.write('\n')
```

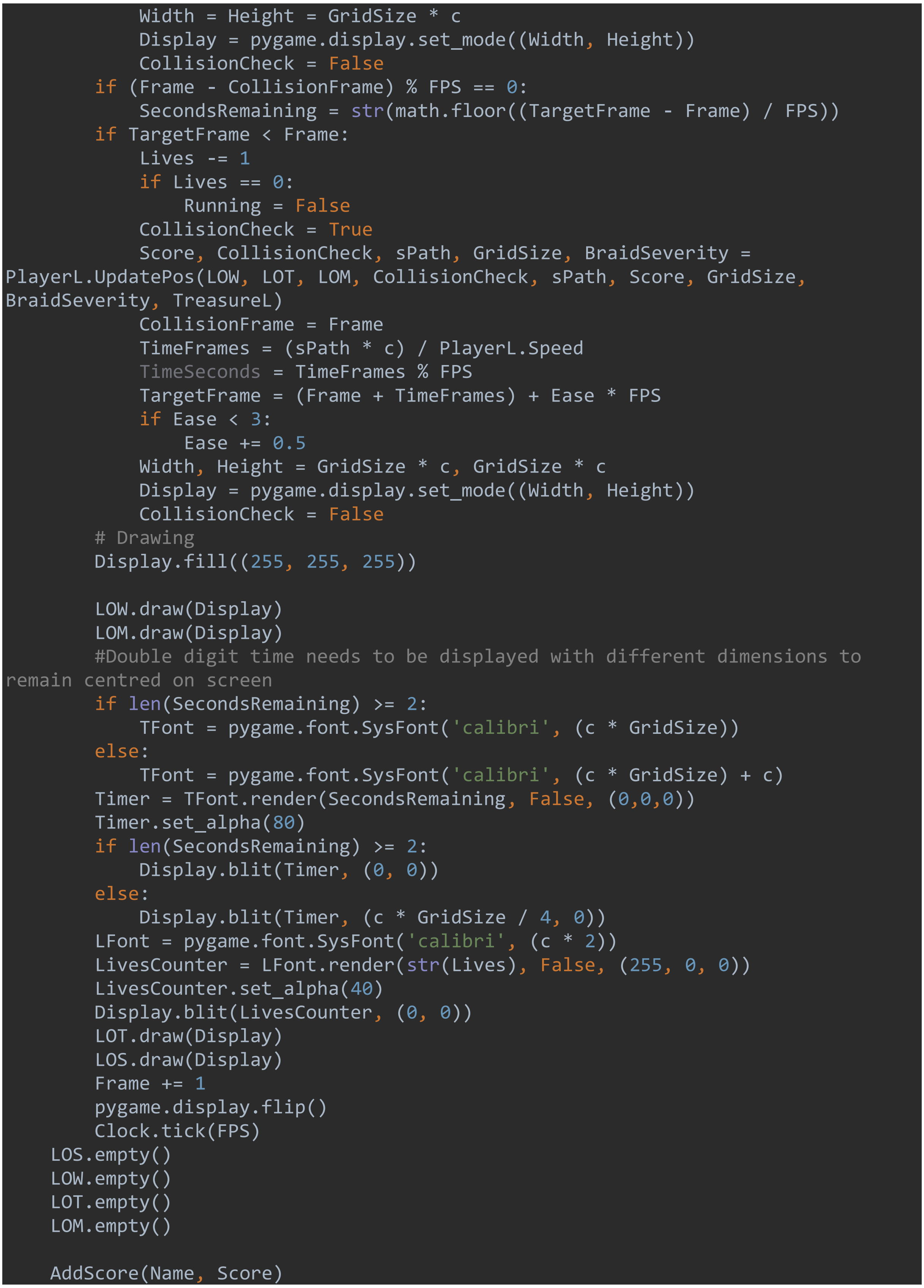
Key code 2.4 Game –



```
def Game():

    GridSize = 5
    BraidSeverity = 0.1
    Difficulty = 'Easy'

    LOB = [Block() for block in range(GridSize ** 2)]
    if Difficulty == 'Easy':
        LOB = BinaryTree(GridSize, LOB)

    elif Difficulty == 'Medium':
        LOB = Sidewinder(GridSize, LOB)

    elif Difficulty == 'Hard':
        LOB = AldousBroder(GridSize, LOB)

    LOB = Braid(GridSize, LOB, BraidSeverity)
    LoadMaze(c, hStart, vStart, LOB, GridSize)


    Name = input("Enter Player Name:")
    FPS = 60
    Frame = 0
```



```
    Clock = pygame.time.Clock()
    Width = Height = GridSize * c
    Display = pygame.display.set_mode((Width, Height))
    PlayerL = Player()
    TreasureL = Treasure()
    PlayerL.rect.x = (GridSize * c / 2) + 1
    PlayerL.rect.y = (GridSize * c / 2) + 1

    Row = Column = GridSize // 2
    sPath = Dijkstra(LOB, GridSize, Row, Column, TreasureL.Row, TreasureL.Column)
    TimeFrames = (sPath * c) / PlayerL.Speed
    TimeSeconds = TimeFrames / FPS
    Ease = 5
    TargetFrame = (Frame + TimeFrames) + Ease * FPS
    SecondsRemaining = str(math.floor((TargetFrame - Frame) / FPS))
    CollisionFrame = 0
    CollisionCheck = False
    Score = Completed = 0
    Lives = 3

    Running = True
    while Running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                Running = False
            # Game Logic
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_UP:
                    PlayerL.RecordMove(0, -PlayerL.Speed)
                elif event.key == pygame.K_RIGHT:
                    PlayerL.RecordMove(PlayerL.Speed, 0)
                elif event.key == pygame.K_DOWN:
                    PlayerL.RecordMove(0, PlayerL.Speed)
                elif event.key == pygame.K_LEFT:
                    PlayerL.RecordMove(-PlayerL.Speed, 0)

            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_UP:
                    PlayerL.RecordMove(0, PlayerL.Speed)
                elif event.key == pygame.K_RIGHT:
                    PlayerL.RecordMove(-PlayerL.Speed, 0)
                elif event.key == pygame.K_DOWN:
                    PlayerL.RecordMove(0, -PlayerL.Speed)
                elif event.key == pygame.K_LEFT:
                    PlayerL.RecordMove(PlayerL.Speed, 0)

        Score, CollisionCheck, sPath, GridSize, BraidSeverity =
PlayerL.UpdatePos(LOW, LOT, LOM, CollisionCheck, sPath, Score, GridSize,
BraidSeverity, TreasureL)
        if CollisionCheck:
            Completed += 1
            if Completed % 10 == 0:
                Lives += 1
            CollisionFrame = Frame
            TimeFrames = (sPath * c) / PlayerL.Speed
            TimeSeconds = TimeFrames % FPS
            TargetFrame = (Frame + TimeFrames) + Ease * FPS
            if Ease > 1:
                Ease = Ease - 0.5
```



```
            Width = Height = GridSize * c
            Display = pygame.display.set_mode((Width, Height))
            CollisionCheck = False
        if (Frame - CollisionFrame) % FPS == 0:
            SecondsRemaining = str(math.floor((TargetFrame - Frame) / FPS))
        if TargetFrame < Frame:
            Lives -= 1
            if Lives == 0:
                Running = False
            CollisionCheck = True
            Score, CollisionCheck, sPath, GridSize, BraidSeverity =
PlayerL.UpdatePos(LOW, LOT, LOM, CollisionCheck, sPath, Score, GridSize,
BraidSeverity, TreasureL)
            CollisionFrame = Frame
            TimeFrames = (sPath * c) / PlayerL.Speed
            TimeSeconds = TimeFrames % FPS
            TargetFrame = (Frame + TimeFrames) + Ease * FPS
            if Ease < 3:
                Ease += 0.5
            Width, Height = GridSize * c, GridSize * c
            Display = pygame.display.set_mode((Width, Height))
            CollisionCheck = False
        # Drawing
        Display.fill((255, 255, 255))

        LOW.draw(Display)
        LOM.draw(Display)
        #Double digit time needs to be displayed with different dimensions to
remain centred on screen
        if len(SecondsRemaining) >= 2:
            TFont = pygame.font.SysFont('calibri', (c * GridSize))
        else:
            TFont = pygame.font.SysFont('calibri', (c * GridSize) + c)
        Timer = TFont.render(SecondsRemaining, False, (0,0,0))
        Timer.set_alpha(80)
        if len(SecondsRemaining) >= 2:
            Display.blit(Timer, (0, 0))
        else:
            Display.blit(Timer, (c * GridSize / 4, 0))
        LFont = pygame.font.SysFont('calibri', (c * 2))
        LivesCounter = LFont.render(str(Lives), False, (255, 0, 0))
        LivesCounter.set_alpha(40)
        Display.blit(LivesCounter, (0, 0))
        LOT.draw(Display)
        LOS.draw(Display)
        Frame += 1
        pygame.display.flip()
        Clock.tick(FPS)
    LOS.empty()
    LOW.empty()
    LOT.empty()
    LOM.empty()

    AddScore(Name, Score)
```

Key code 3.0 Classes –

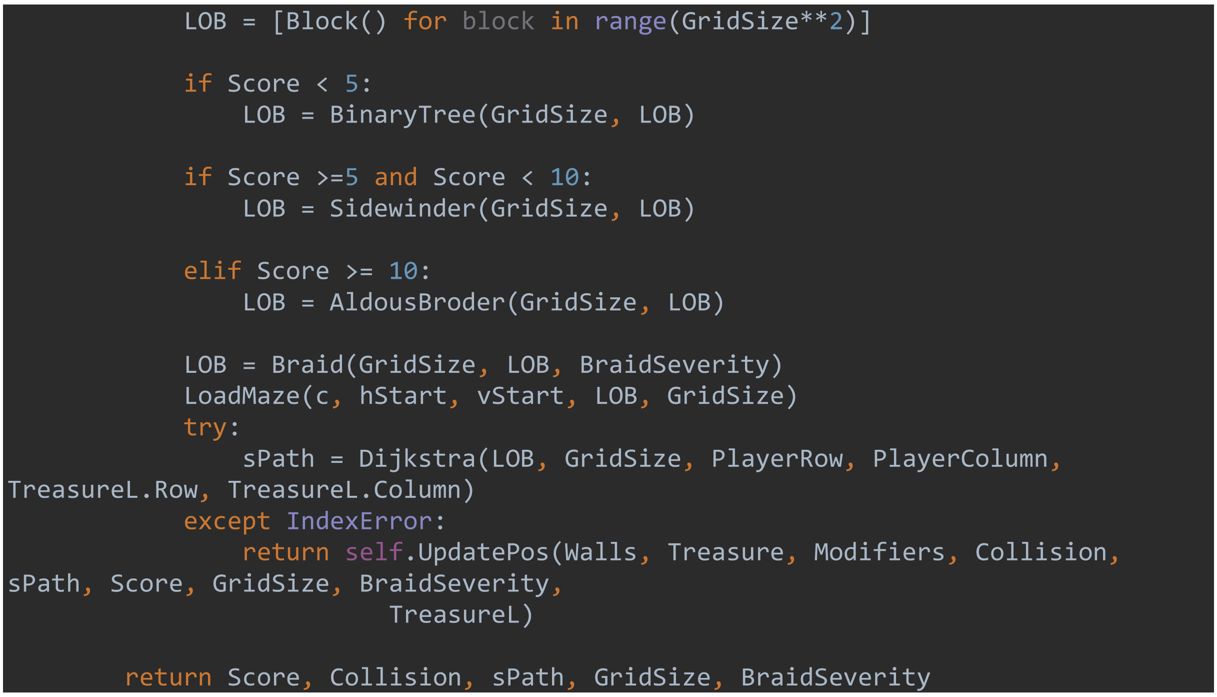
Below I have displayed the finished code for all the pseudocode shown in Pseudocode 3.x

Key code 3.1 Block class –

```
class Block:
    def __init__(self):
        self.Row = -1
        self.Column = -1
        self.North = 1
        self.South = 1
        self.West = 1
        self.East = 1
        self.Visited = 0
        self.Distance = 0
        self.SpeedMultiplier = 1
```

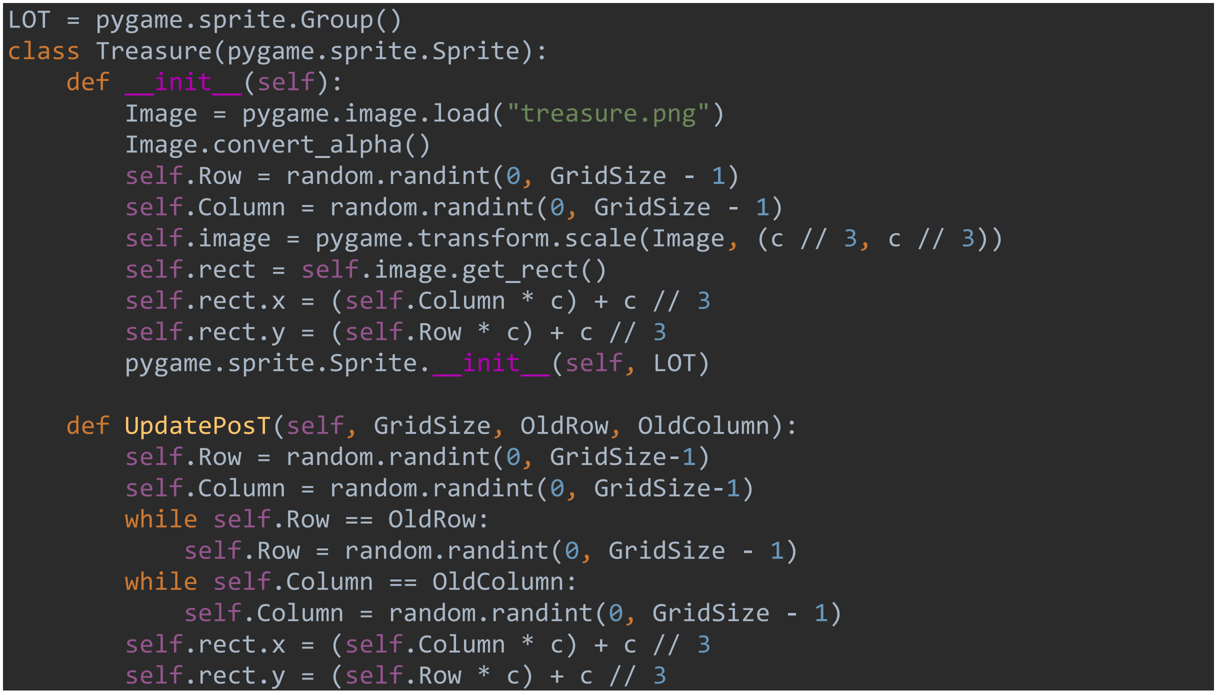
Key code 3.2 Player class–

```
LOS = pygame.sprite.Group()
class Player(pygame.sprite.Sprite):
    def __init__(self):
        self.image = pygame.Surface([c / 3, c / 3])
        self.rect = self.image.get_rect()
        self.xMoved = 0
        self.yMoved = 0
        self.Speed = c/12.5
        self.Modified = ''
        pygame.sprite.Sprite.__init__(self, LOS)

    def RecordMove(self, xChange, yChange):
        self.xMoved += xChange
        self.yMoved += yChange

    def UpdatePos(self, Walls, Treasure, Modifiers, Collision, sPath, Score,
GridSize, BraidSeverity, TreasureL):
        CollisionList = pygame.sprite.spritecollide(self, Modifiers, False)
        ModifierSum = 0
        SpeedModifier = ''
        for Modifier in CollisionList:
            if Modifier.Type == 'Fast':
                ModifierSum += 1
            elif Modifier.Type == 'Slow':
                ModifierSum -= 1
        if ModifierSum > 0:
            SpeedModifier = 'Fast'
        elif ModifierSum < 0:
            SpeedModifier = 'Slow'
        elif ModifierSum == 0:
            SpeedModifier = 'Normal'

        if SpeedModifier == 'Fast' and self.Modified == '':
            self.RecordMove(self.xMoved, self.yMoved)
            self.Modified = 'Faster'
            self.Speed = self.Speed * 2
        elif SpeedModifier == 'Slow' and self.Modified == '':
            self.RecordMove(-(self.xMoved/2),-(self.yMoved/2))
            self.Modified = 'Slower'
            self.Speed = self.Speed / 2
        if SpeedModifier == 'Normal':
            if self.Modified == 'Slower':
                self.RecordMove(self.xMoved, self.yMoved)
                self.Speed = self.Speed * 2
            elif self.Modified == 'Faster':
                self.RecordMove(-self.xMoved/2, -self.yMoved/2)
                self.Speed = self.Speed / 2
            self.Modified = ''

        self.rect.x += self.xMoved

        CollisionList = pygame.sprite.spritecollide(self, Walls, False)
        for wall in CollisionList:
            if self.xMoved > 0:
                self.rect.right = wall.rect.left
            else:
                self.rect.left = wall.rect.right

        self.rect.y += self.yMoved

        CollisionList = pygame.sprite.spritecollide(self, Walls, False)
        for wall in CollisionList:
            if self.yMoved > 0:
                self.rect.bottom = wall.rect.top
            else:
                self.rect.top = wall.rect.bottom

        CollisionList = pygame.sprite.spritecollide(self, Treasure, False)
        if len(CollisionList) > 0 or Collision == True:
            if len(CollisionList) > 0:
                Score += 1
                if GridSize <= c/2.5:
                    GridSize += 1
                if BraidSeverity < 1:
                    BraidSeverity += 0.1
            else:
                if BraidSeverity > 0.1:
                    BraidSeverity -= 0.1
                if GridSize > 2:
                    GridSize -= 1
                Score -= 1
            #Checks if player is within boundary, if not, player co-ordinates
adjusted
            if Collision == True:
                if self.rect.x > c:
                    self.rect.x -= c
                if self.rect.y > c:
                    self.rect.y -= c
            #rect.x and rect.y give the position of the top left corner of player
sprite, so a half of
            #player sprite size must be added to each co-ordinate to obtain centre
of player
            Collision = True
            TreasureL.UpdatePosT(GridSize, TreasureL.Row, TreasureL.Column)
            PlayerColumn = math.floor((self.rect.x + (c / 6)) / c)
            PlayerRow = math.floor((self.rect.y + (c / 6)) / c)

            Walls.empty()
            Modifiers.empty()
```



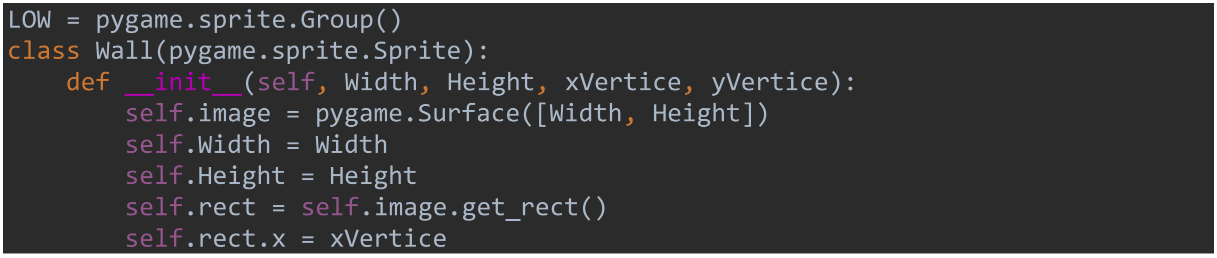
```
            LOB = [Block() for block in range(GridSize**2)]

            if Score < 5:
                LOB = BinaryTree(GridSize, LOB)

            if Score >=5 and Score < 10:
                LOB = Sidewinder(GridSize, LOB)

            elif Score >= 10:
                LOB = AldousBroder(GridSize, LOB)

            LOB = Braid(GridSize, LOB, BraidSeverity)
            LoadMaze(c, hStart, vStart, LOB, GridSize)
            try:
                sPath = Dijkstra(LOB, GridSize, PlayerRow, PlayerColumn,
TreasureL.Row, TreasureL.Column)
            except IndexError:
                return self.UpdatePos(Walls, Treasure, Modifiers, Collision,
sPath, Score, GridSize, BraidSeverity,
                          TreasureL)

        return Score, Collision, sPath, GridSize, BraidSeverity
```

Key code 3.3 Treasure class –



```
LOT = pygame.sprite.Group()
class Treasure(pygame.sprite.Sprite):
    def __init__(self):
        Image = pygame.image.load("treasure.png")
        Image.convert_alpha()
        self.Row = random.randint(0, GridSize - 1)
        self.Column = random.randint(0, GridSize - 1)
        self.image = pygame.transform.scale(Image, (c // 3, c // 3))
        self.rect = self.image.get_rect()
        self.rect.x = (self.Column * c) + c // 3
        self.rect.y = (self.Row * c) + c // 3
        pygame.sprite.Sprite.__init__(self, LOT)

    def UpdatePosT(self, GridSize, OldRow, OldColumn):
        self.Row = random.randint(0, GridSize-1)
        self.Column = random.randint(0, GridSize-1)
        while self.Row == OldRow:
            self.Row = random.randint(0, GridSize - 1)
        while self.Column == OldColumn:
            self.Column = random.randint(0, GridSize - 1)
        self.rect.x = (self.Column * c) + c // 3
        self.rect.y = (self.Row * c) + c // 3
```

Key code 3.4 Wall class –



```
LOW = pygame.sprite.Group()
class Wall(pygame.sprite.Sprite):
    def __init__(self, Width, Height, xVertice, yVertice):
        self.image = pygame.Surface([Width, Height])
        self.Width = Width
        self.Height = Height
        self.rect = self.image.get_rect()
        self.rect.x = xVertice
        self.rect.y = yVertice
        pygame.sprite.Sprite.__init__(self, LOW)
```

Key code 3.5 Modifier class –

```
LOM = pygame.sprite.Group()
class Modifier(pygame.sprite.Sprite):
    def __init__(self, Size,  xVertice, yVertice, Type):
        self.image = pygame.Surface([Size, Size]) #Square so width = height
        self.Type = Type
        if Type == 'Slow':
            self.image.fill((255,204,153))
        elif Type == 'Fast':
            self.image.fill((153,255,255))
        self.rect = self.image.get_rect()
        self.rect.x = xVertice
        self.rect.y = yVertice
        pygame.sprite.Sprite.__init__(self, LOM)
```

Complex programming techniques

Complex programming techniques 1.0 –

Below I have listed some complex programming techniques that have been used in this project

Complex programming techniques 1.1 Nested Loops –

I have used nested loops in all of the maze generation algorithms apart from Aldous-Broder, as this allowed me to loop through each column of each row, meaning that every co-ordinate on a two-dimensional grid could be visited. This was not used in Aldous-Broder because a for loop would be impossible to use for the algorithm. This is because Aldous-Broder has an indeterminant number of iterations that must be run, and therefore a while loop must be used.

Complex programming techniques 1.2 List of objects –

I have used a list of objects in the form of LOB, LOW, LOM, LOS, and LOT. This is short for List of Blocks. Using this the entire maze can be described in a single variable, this is because all the information of the maze is stored in the attributes of the blocks that make up the maze. Using this, passing the maze into functions is easy and no different from passing anything else into them.

This technique has also allowed me to easily find adjacent blocks to any block in the maze, as the index of a block in the LOB is determined by its position in the maze, and therefore the position of blocks immediately surrounding it can also be found. This saves space, as otherwise I would need to create another attribute in the block class which stores the adjacent blocks to the current block.

The use of this technique in the form of LOW, LOM, LOS, and LOT, allows for easy collision checking in pygame, and prevents me having to keep track of every modifier or wall that is currently instantiated.

Complex programming techniques 1.3 Extensive use of functions –

All the code in my solution apart from a few lines is in the form of functions, procedures or methods. I have done this because this makes developing the code much easier, as each function has its own set of local variables and changing something in one function will not create an adverse effect in another function. Another reason why this is useful is because some functions need to be called twice in the code, and if they were not functions, this would waste space and make the code look much more cluttered. This is particularly evident in UpdatePos, which calls all maze generation algorithms, and the function UpdatePos itself is called twice in the main game loop.

This also allows for the use of recursion, which is used in Menu and in Dijkstra for the purpose of validation and error-checking.

Defensive programming

Defensive programming 1.1 Menu –

```
def Menu():
    print("----------MENU----------")
    print("1. Play Game")
    print("2. View Highscores")
    print("3. Exit")
    Choice = input("Select option using number")
    return Choice
Choice = ''
while Choice != '3':
    Choice = Menu()
    if Choice == '1':
        Game()
    elif Choice == '2':
        ShowHighscores()
```

Above I have shown the Menu function and the place where it is called. Here, the player input is validated to ensure that it is one of the 3 available options.

Defensive programming 1.2 Sidewinder –

```
for Index in range(GridSize):
    LOB[Index].North = 1
    LOB[Index + (GridSize**2 - GridSize)].South = 1
```

Above I have shown an excerpt from the Sidewinder function which is placed at the end of the function. This ensures that the bottom row of blocks have a southern boundary, and that the top row of blocks have a northern boundary. This is done to make sure that even if the main loop ends up carving out of bounds, the boundary edges will remain intact.

Defensive programming 1.3 Dijkstra –

```
try:
    sPath = Dijkstra(LOB, GridSize, PlayerRow, PlayerColumn, TreasureL.Row,
TreasureL.Column)
except IndexError:
    return self.UpdatePos(Walls, Treasure, Modifiers, Collision, sPath, Score,
GridSize, BraidSeverity,
              TreasureL)
```

Above I have shown the place where Dijkstra is used in the UpdatePos method. This is put in place so that if a rare error does occur with maze generation which results in Dijkstra not being able to find a particular index (as I have experienced previously), a new maze will simply be generated recursively until Dijkstra does return a valid sPath. I have done this because very rarely Dijkstra would crash the program with an IndexError, and since it was exceedingly rare, I was not able to find a way to replicate it or debug it effectively. This precaution has fixed the issue and I have not seen the IndexError crash since implementing this, even after hundreds of playtests.

Coding style

I have tried to make my code as readable as possible. This includes adding space between functions, and between different sections of a function. This also includes spacing out calculations, so they do not appear cluttered. Additionally, all variables are capitalised appropriately, e.g. ToVisit instead of Tovisit or tovisist, for the purpose of readability. The exception to this capitalisation rule is xMoved, yMoved, hStart, and vStart, and this is because they would need to otherwise be written as XMoved, YMoved etc, which in my opinion looks less readable than simply keeping the first letter uncapitalised.

I have also separated different types of functions. For example, the maze generation functions all appear first in the code, with the game logic functions and classes appearing later. The use of pygame is completely separated from the maze generation and this is so that the two aspects of the solution may be worked on separately without interfering with each other.

I have also added comments throughout my code in particularly confusing or convoluted areas, as well as in places where the purpose of the code is not obvious or easy to be understood. For example, I have left a comment saying - #0 is north, 1 is east, 2 is south, 3 is west - in the Aldous-Broder algorithm, this is because it is not immediately obvious what these numbers represent as they are an abstract representation of compass directions.

Testing

Test plan

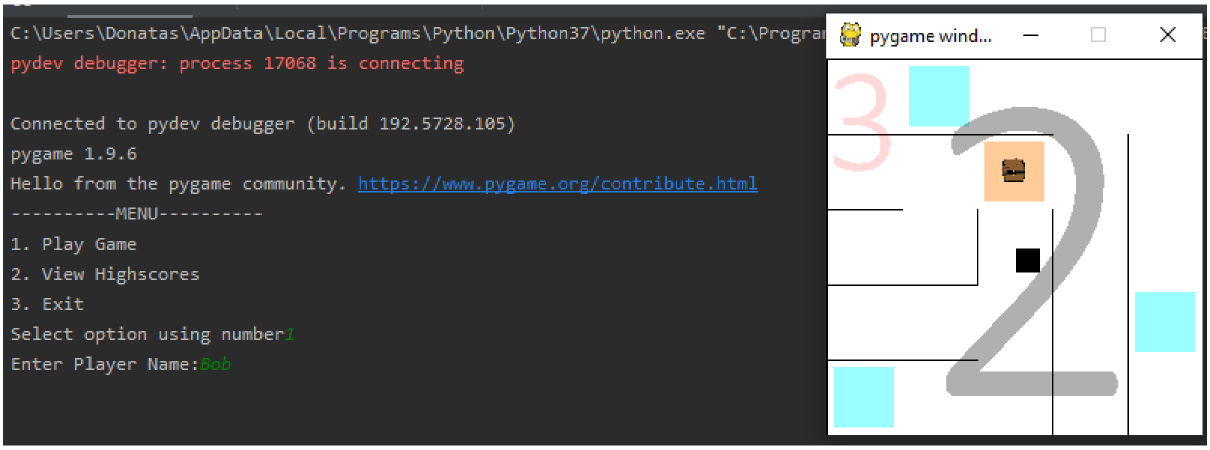
Test #	Purpose	Test data	Expected outcome	Actual outcome
1	Ensure the menu function works as intended	Input of ‘1’ in menu prompt, then player name entered	Playable window launched	Testing evidence 1
2	Ensure the menu function works as intended	Input of ‘2’ in menu prompt	10 highest scores displayed; menu displayed again	Testing evidence 2
3	Ensure the menu function works as intended	Input of ‘3’ in menu prompt	Program should exit	Testing evidence 3
4	Ensure the menu function works as intended	Input of ‘8’ in menu prompt	Menu should be redisplayed, and new input prompted for	Testing evidence 4
5	Ensure timer is displayed centred and correct size	Input of ‘1’ in menu prompt, then player name entered	Timer should be displayed correctly and start counting down	Testing evidence 5
6	Ensure lives are displayed in correct area and size	Input of ‘1’ in menu prompt, then player name entered	Lives should be displayed in top left	Testing evidence 5
7	Ensure player colliding with walls is functioning	Player moves into a wall	Player should be stopped at wall-player boundary	Testing evidence 6
8	Ensure player colliding with treasure starts new maze	Player moves into treasure	New maze should be generated	Testing evidence 7
9	Ensure player running out of time starts new maze	Player does not collide with treasure in allotted time	New maze should be generated	Testing evidence 8
10	Ensure grid size increases by 1 when player collides with treasure	Player moves into treasure	Grid size of new maze should be 1 greater	Testing evidence 7
11	Ensure grid size decreases by 1 when player runs out of time	Player does not collide with treasure in allotted time	Grid size of new maze should be 1 less	Testing evidence 8

12	Ensure maze algorithm changes to Sidewinder once Score is > 4	Player completes 5 levels	Maze should look like a Sidewinder maze	Testing evidence 9
13	Ensure maze algorithm changes to Aldous-Broder once score is > 9	Player completes 10 levels	Maze should look like an AldousBroder maze	Testing evidence 10
14	Ensure treasure location is changed when new maze is generated	Player moves into treasure	Treasure should be moved to new location in new maze	Testing evidence 7
15	Ensure lives decreases by 1 when player runs out of time	Player does not collide with treasure in allotted time	Lives should be displayed as 1 less in new maze	Testing evidence 8
16	Ensure lives increases by 1 when player has completed 10 levels	Player completes 10 levels	Lives should be displayed as 1 more in new maze	Testing evidence 10
17	Ensure modifiers are correctly displayed	Input of ‘1’ in menu prompt, then player name entered	Several modifiers should be visible in the maze	Testing evidence 1
18	Ensure game stops and menu displayed when player runs out of lives	Player loses 3 levels at the beginning, bringing life to 0	Maze of size 2 should be displayed and paused, and menu displayed	Testing evidence 11
19	Ensure player score is written to highscores.txt once player runs out of lives	Player wins some levels to gain score, then loses until lives are 0	highscores.txt file should have score, player name and date written	Testing evidence 12
20	Ensure player is not out of bounds once new maze is generated after grid size has been reduced	Player will stand on bottom right corner and run out of allotted time	Player should be in the new bottom right corner, having been moved to the left and up.	Testing evidence 13
21	Ensure new game can be begun once player loses current game	Run out of lives, then input of ‘1’ in menu prompt, then	New game should begin with reset player stats and maze generation settings	Testing evidence 14

		player name entered		
22	Ensure Dijkstra’s considers weights of connections	Input of ‘1’ in menu prompt, then player name entered	Connections with a modifier should have altered distances	Test evidence 15
23	Ensure Dijkstra’s terminates once goal block is found	Input of ‘1’ in menu prompt, then player name entered	Some blocks (which were unvisited) at the end of the algorithm should have a distance of GridSize * GridSize * 2	Test evidence 15
24	Ensure braiding is working	Start of game BraidSeverity = 0.1, advance 9 levels, BraidSeverity = 1.0	First maze should have significantly fewer dead ends than 10th maze	Test evidence 16

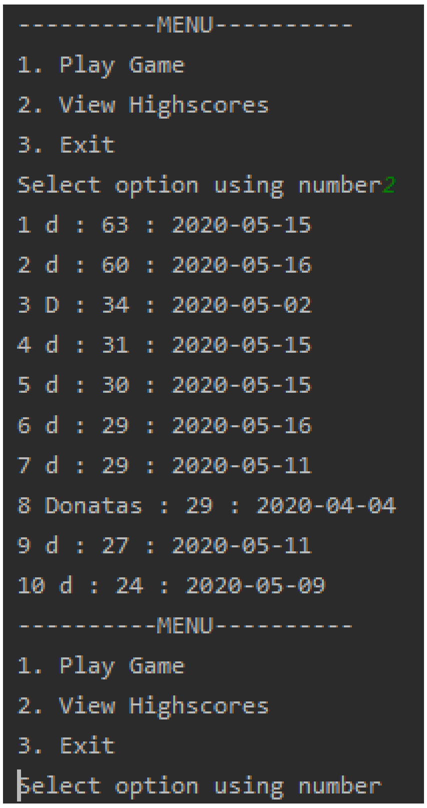
Testing evidence

Testing evidence 1 –



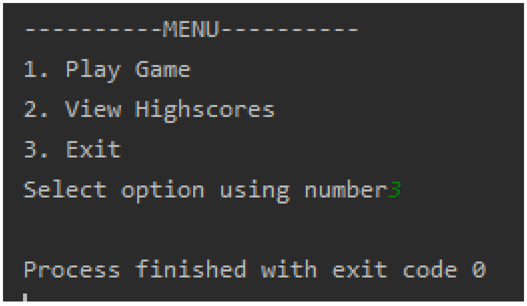
```
C:\Users\Donatas\AppData\Local\Programs\Python\Python37\python.exe "C:\Prograr
pydev debugger: process 17068 is connecting

Connected to pydev debugger (build 192.5728.105)
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number1
Enter Player Name:Bob
```

Testing evidence 2 –



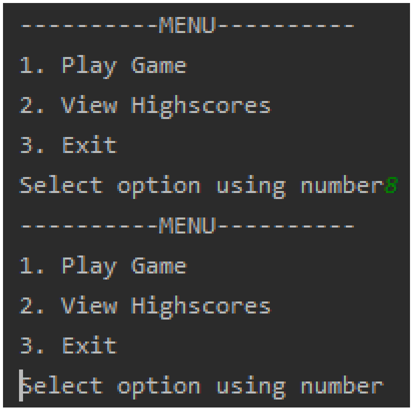
```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number2
1 d : 63 : 2020-05-15
2 d : 60 : 2020-05-16
3 D : 34 : 2020-05-02
4 d : 31 : 2020-05-15
5 d : 30 : 2020-05-15
6 d : 29 : 2020-05-16
7 d : 29 : 2020-05-11
8 Donatas : 29 : 2020-04-04
9 d : 27 : 2020-05-11
10 d : 24 : 2020-05-09
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number
```

Testing evidence 3 –



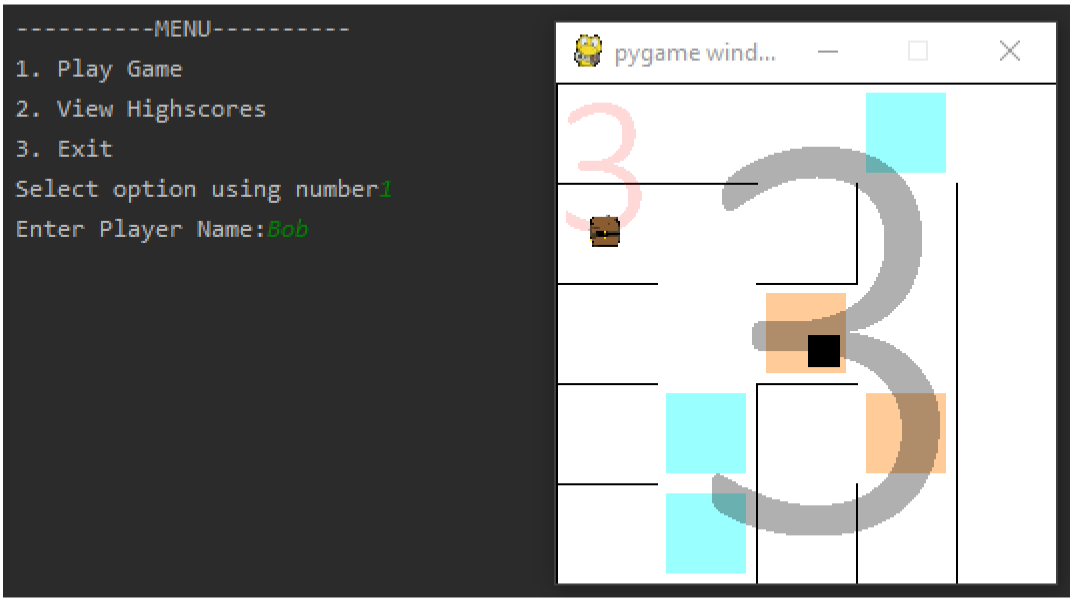
```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number3

Process finished with exit code 0
```

Testing evidence 4 –



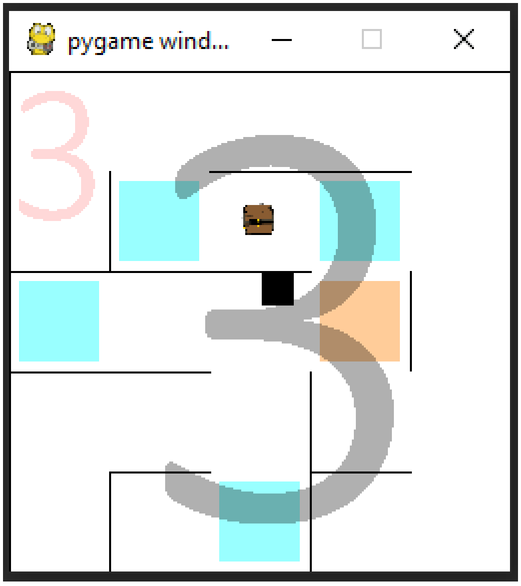
```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number8
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number
```

Testing evidence 5 –

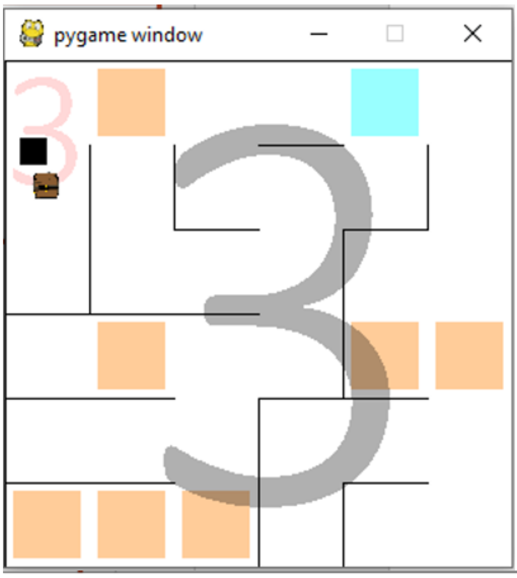


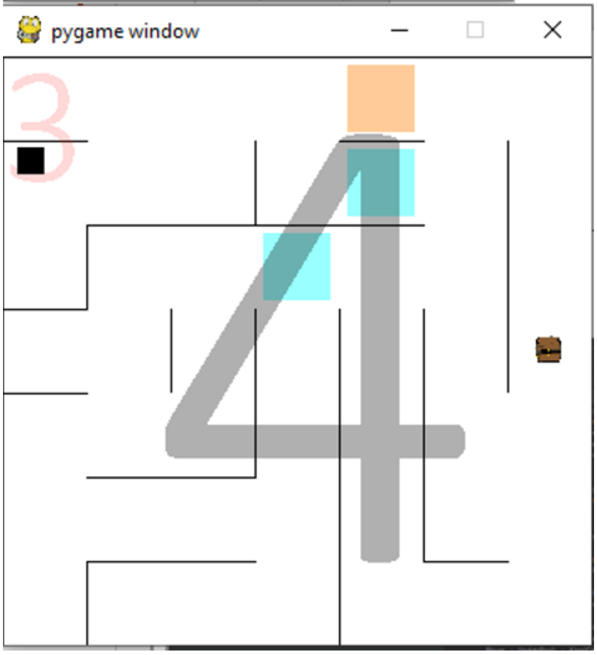
```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number1
Enter Player Name:Bob
```

Testing evidence 6 –

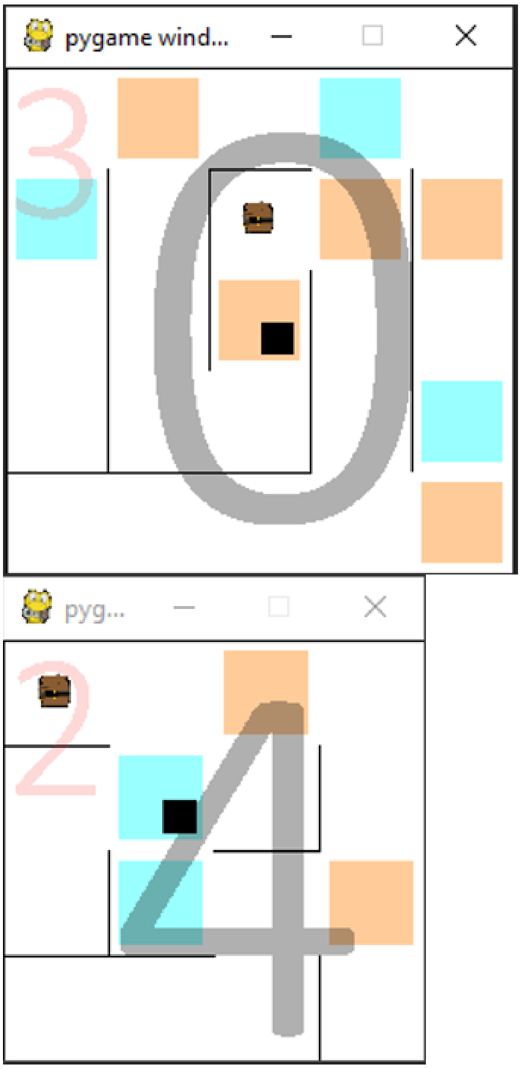


Testing evidence 7 –

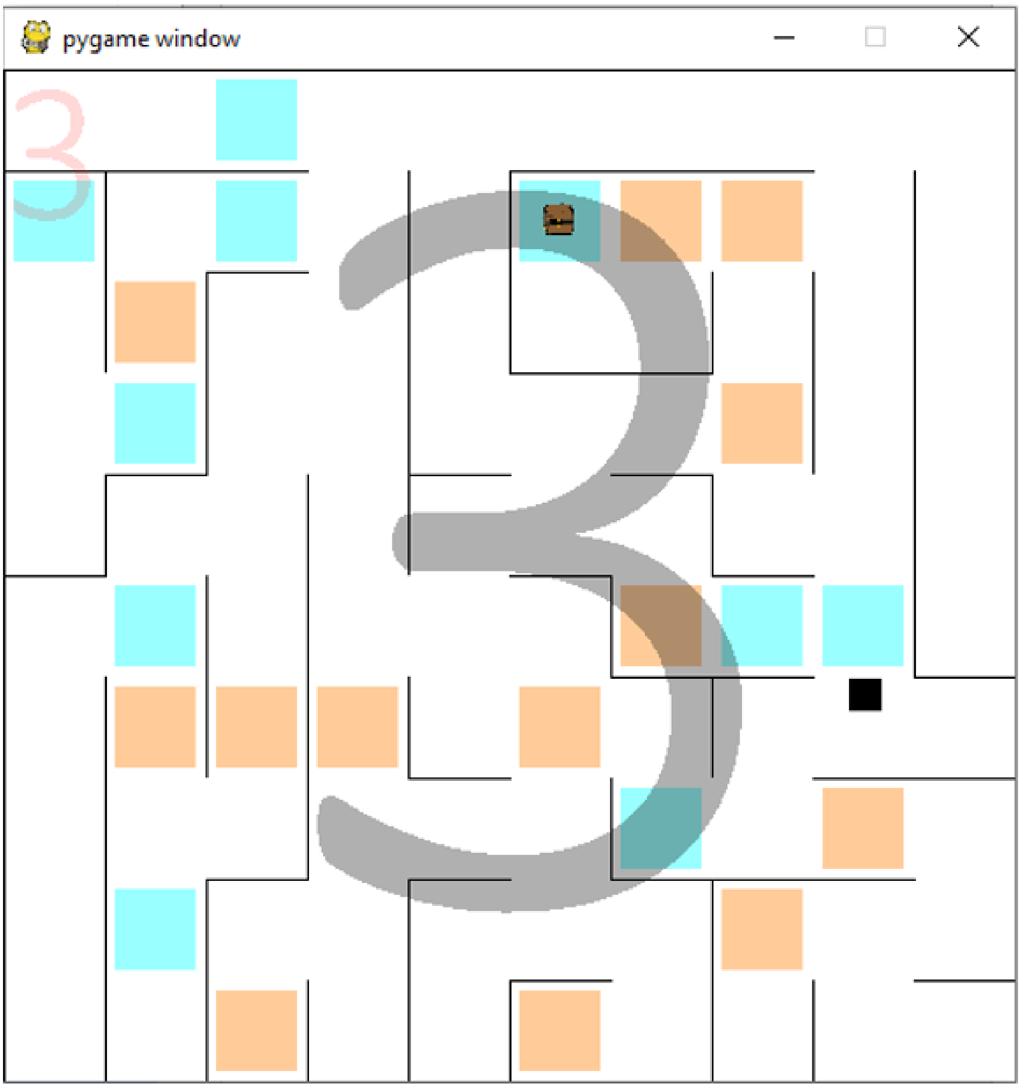




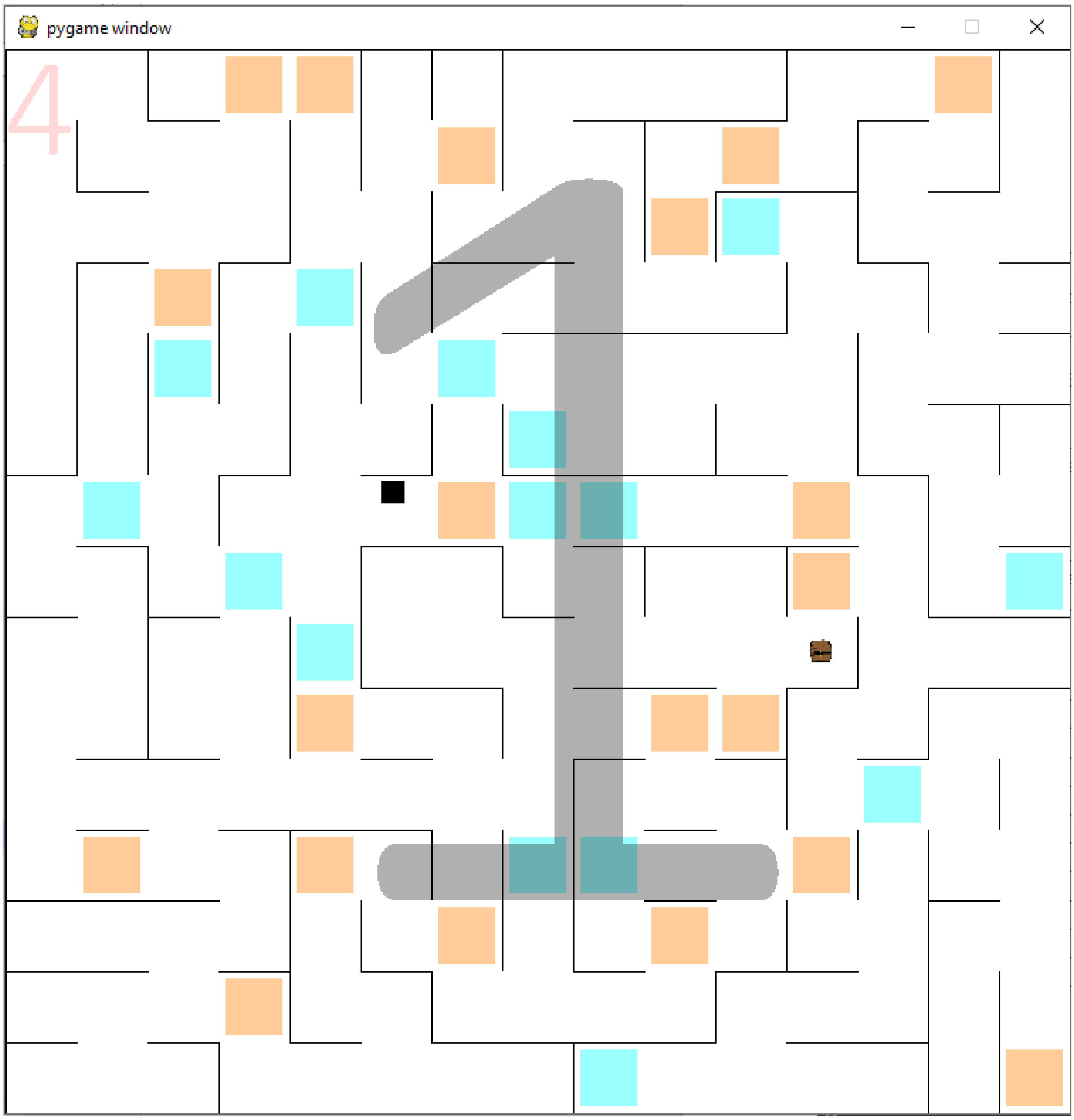
Testing evidence 8 –



Testing evidence 9 –

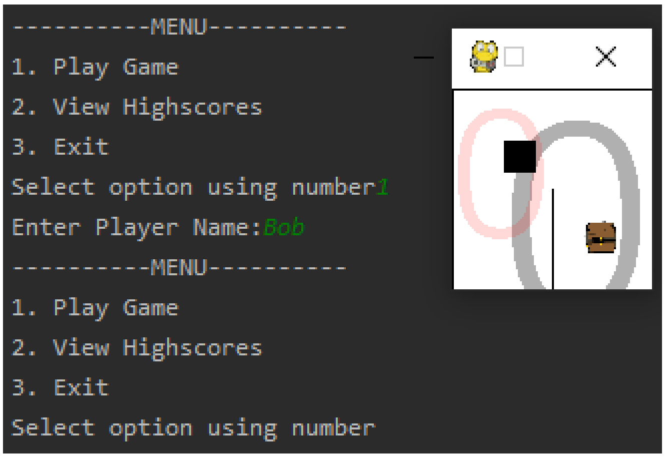


Testing evidence 10 –



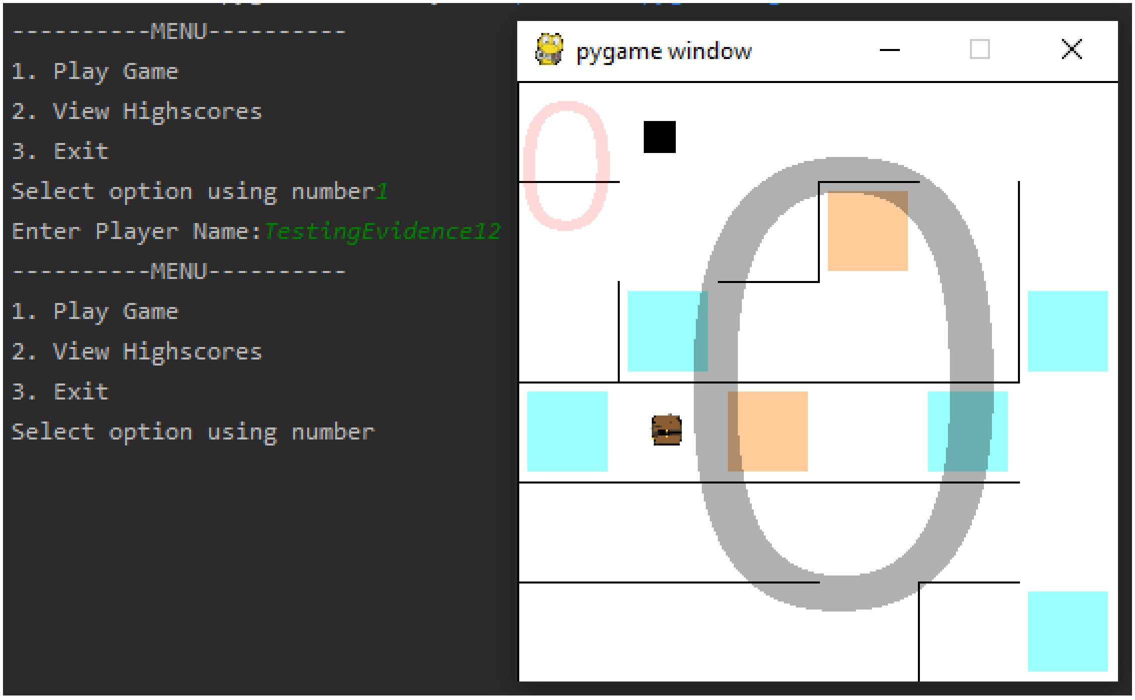
The same scale with other testing evidence could not be kept, since the grid size was too large and keeping the same scale would make this image too large horizontally for Word.

Testing evidence 11 –

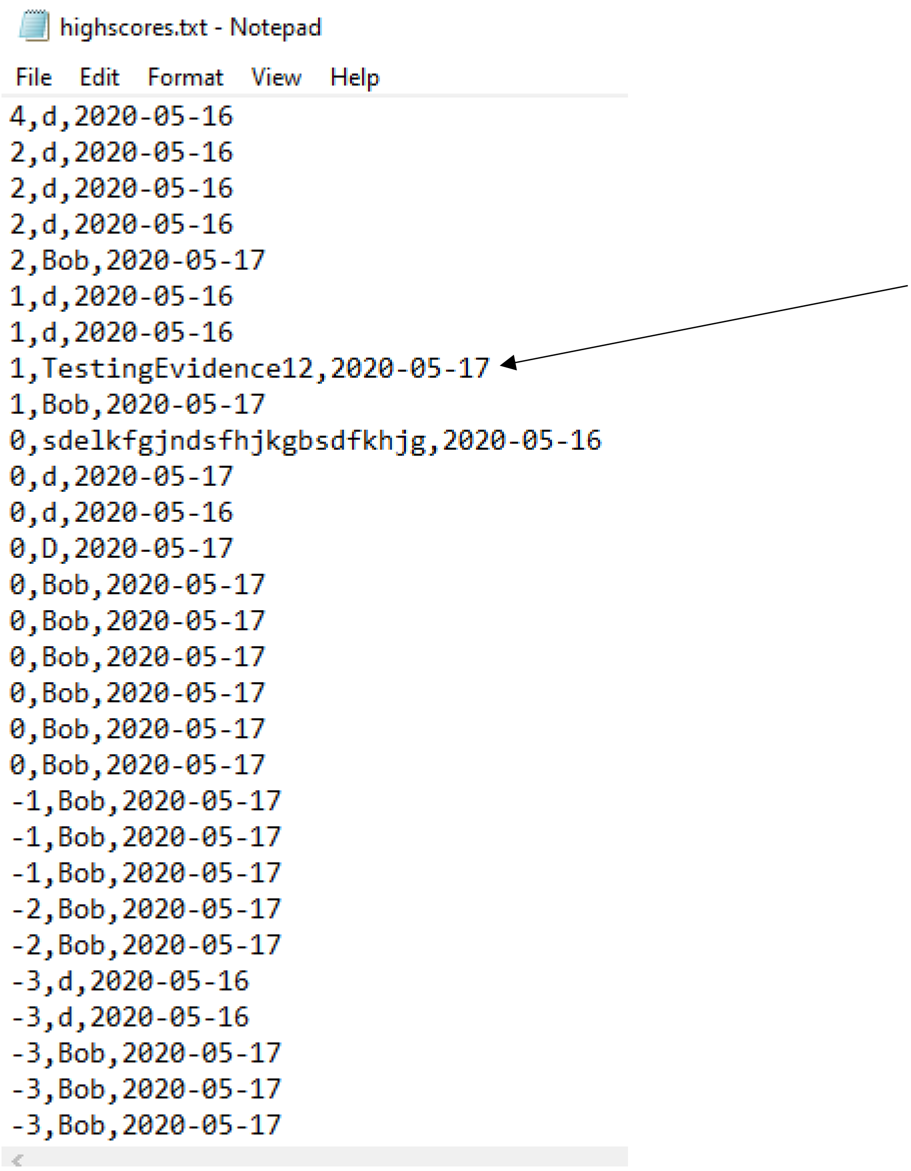


```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number1
Enter Player Name:Bob
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number
```

Testing evidence 12 –

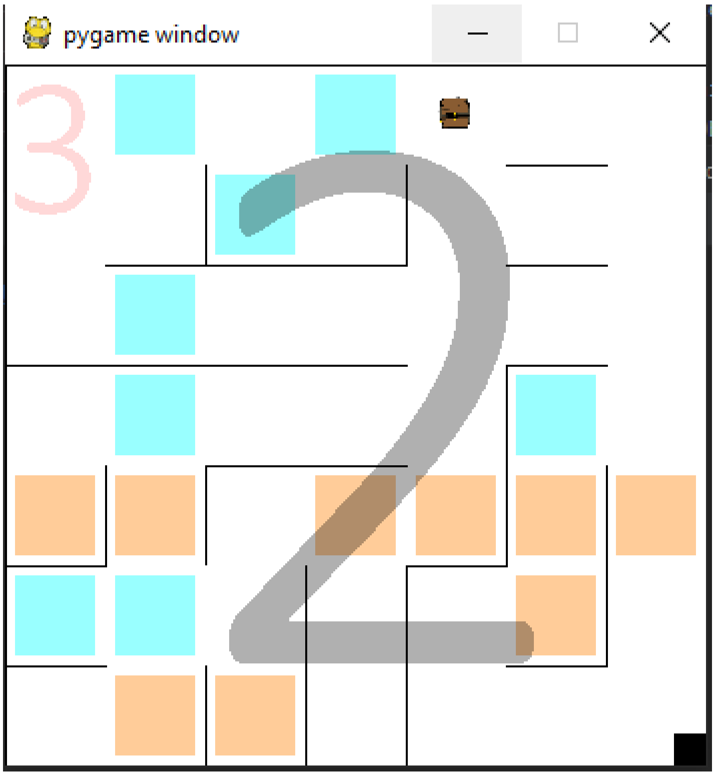


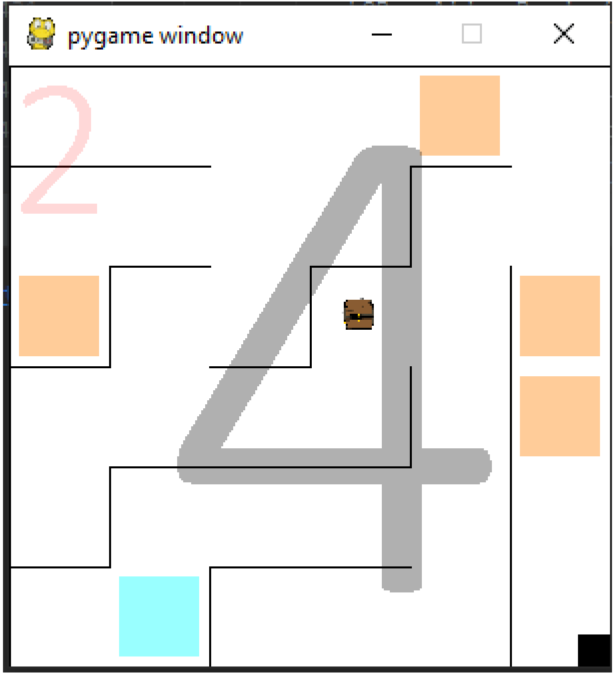
```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number1
Enter Player Name:TestingEvidence12
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number
```



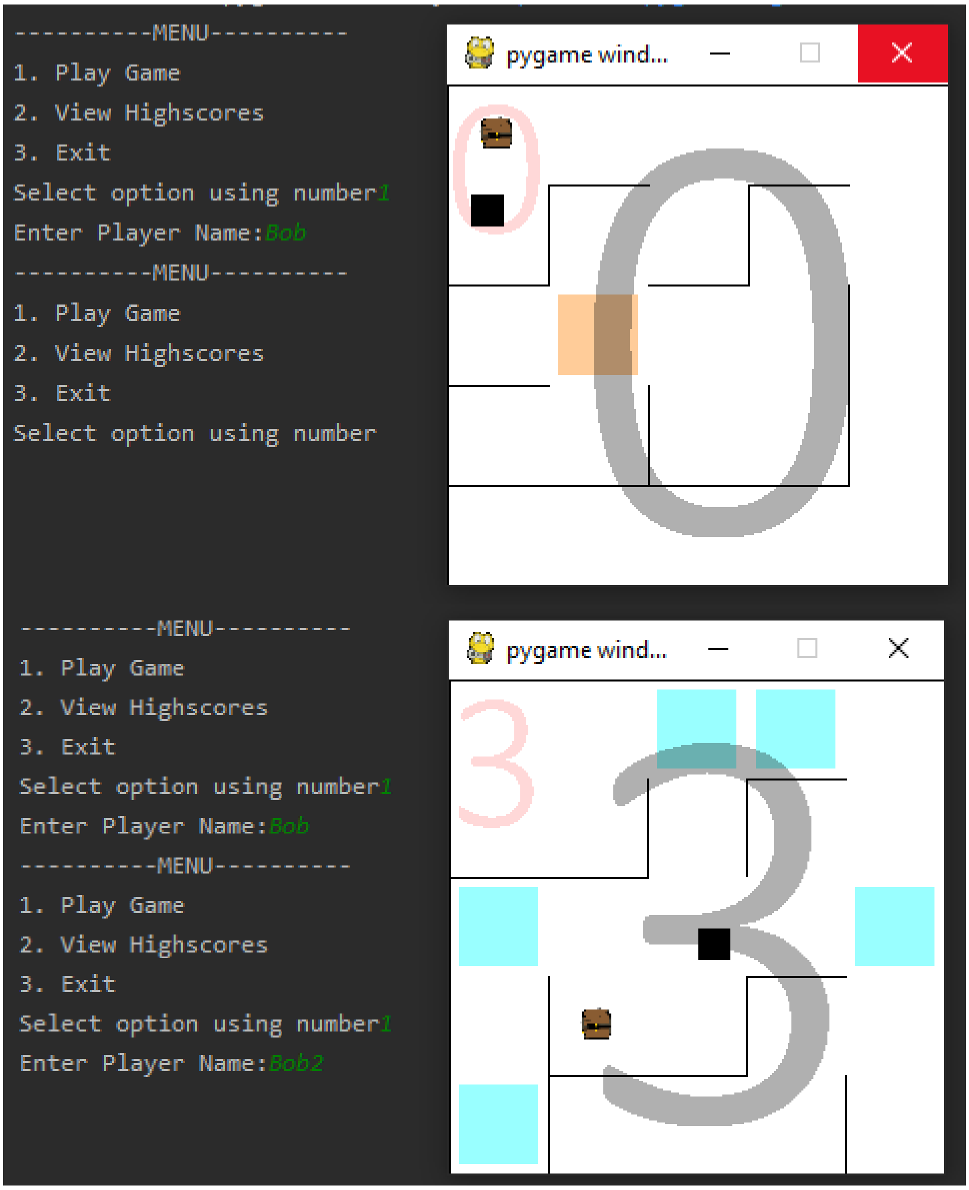
```
4,d,2020-05-16
2,d,2020-05-16
2,d,2020-05-16
2,d,2020-05-16
2,Bob,2020-05-17
1,d,2020-05-16
1,d,2020-05-16
1,TestingEvidence12,2020-05-17
1,Bob,2020-05-17
0,sdelkfgjndsfhjkgbsdfkhjg,2020-05-16
0,d,2020-05-17
0,d,2020-05-16
0,D,2020-05-17
0,Bob,2020-05-17
0,Bob,2020-05-17
0,Bob,2020-05-17
0,Bob,2020-05-17
0,Bob,2020-05-17
0,Bob,2020-05-17
-1,Bob,2020-05-17
-1,Bob,2020-05-17
-1,Bob,2020-05-17
-2,Bob,2020-05-17
-2,Bob,2020-05-17
-3,d,2020-05-16
-3,d,2020-05-16
-3,Bob,2020-05-17
-3,Bob,2020-05-17
-3,Bob,2020-05-17
```

Testing evidence 13 –





Testing evidence 14 –

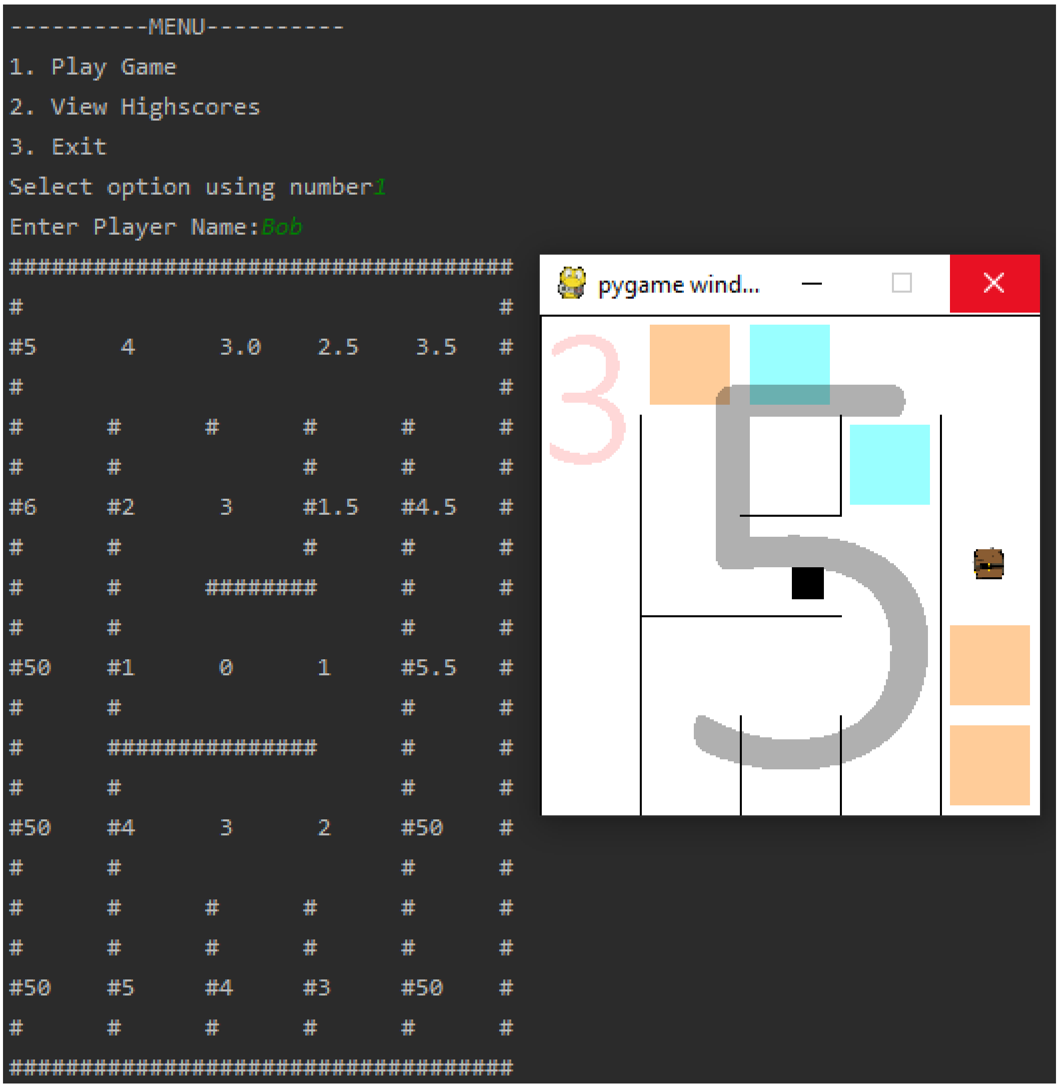


```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number1
Enter Player Name:Bob
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number
```

```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number1
Enter Player Name:Bob
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number1
Enter Player Name:Bob2
```

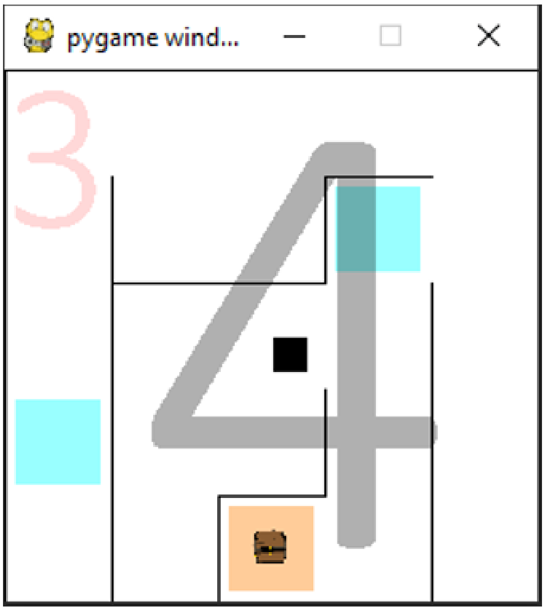
Testing evidence 15 –

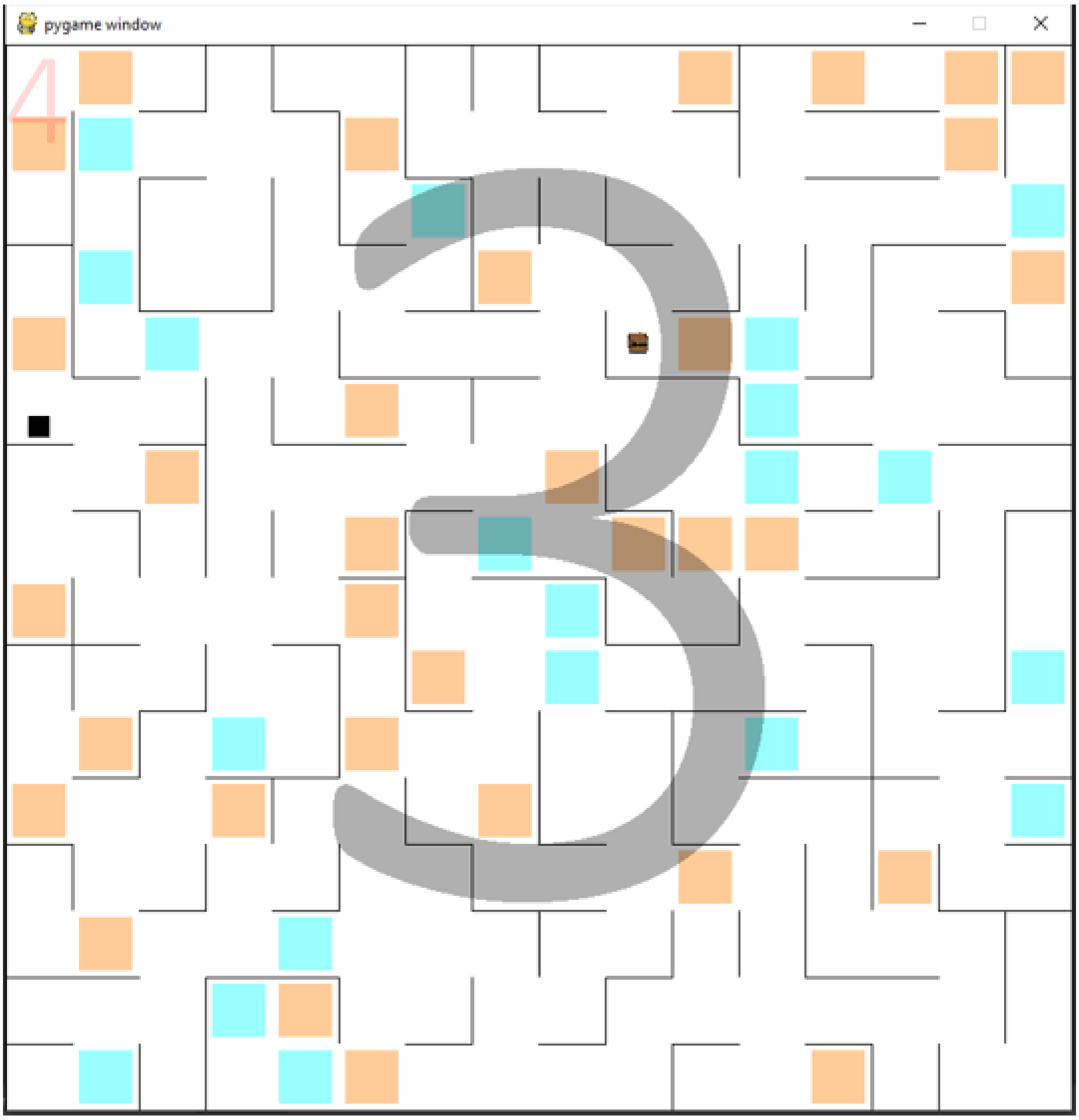
For this test evidence I have created another procedure, DisplayMazeAscii which displays the passed in maze in ascii, and shows the Distance attribute of each block



```
----------MENU----------
1. Play Game
2. View Highscores
3. Exit
Select option using number1
Enter Player Name:Bob
####################################
#                                  #
#5      4      3.0    2.5    3.5   #
#                                  #
#      #      #      #      #      #
#      #             #      #      #
#6     #2     3     #1.5   #4.5   #
#      #             #      #      #
#      #      ########      #      #
#      #                    #      #
#50    #1     0      1      #5.5   #
#      #                    #      #
#      ###############      #      #
#      #                    #      #
#50    #4     3      2      #50    #
#      #                    #      #
#      #      #      #      #      #
#      #      #      #      #      #
#50    #5     #4     #3     #50    #
#      #      #      #      #      #
####################################
```

Test evidence 16 –

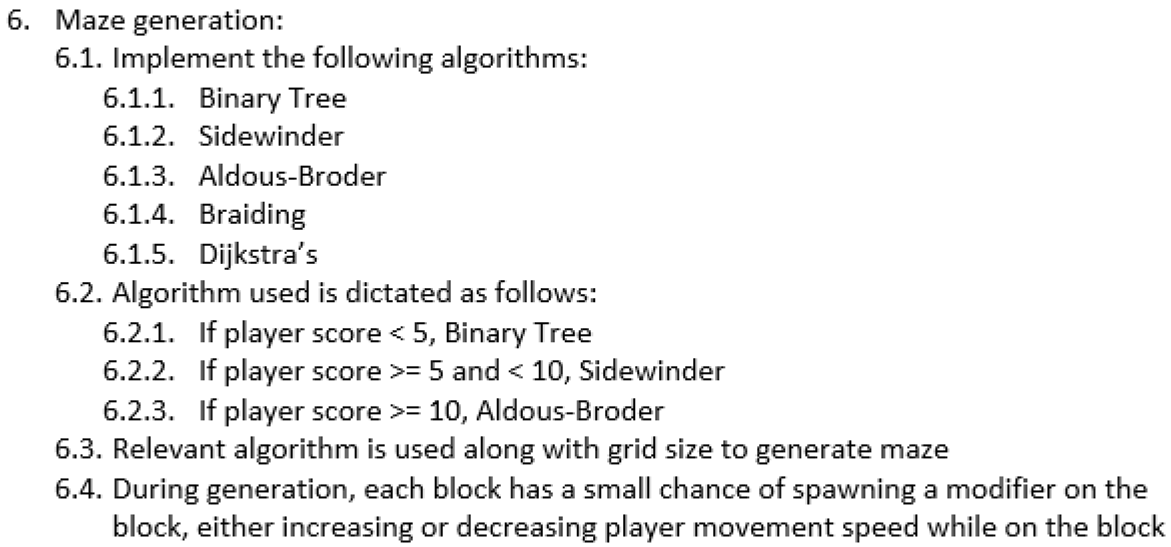




Evaluation

Objectives

1. Player prompted with menu, with the following options:
    1.1. Start a new game
    1.2. View top 10 high scores
    1.3. Exit the game
2. When player chooses to start a new game execute the following operations:
    2.1. Set starting variables for game:
        2.1.1. Grid size
        2.1.2. Lives
        2.1.3. Braid amount
        2.1.4. (Input) Player name
    2.2. Initialise Player and Treasure objects and generate random locations
    2.3. Generate a maze, apply braiding
    2.4. Calculate shortest path between player and treasure
    2.5. Multiply shortest path by length of one block, and divide by player speed for time, adding the variable 'Ease' to this time
    2.6. Launch the playable pygame window, displaying the following:
        2.6.1. Generated maze including speed modifiers on blocks
        2.6.2. Time remaining for this level
        2.6.3. Lives remaining
        2.6.4. Player and treasure sprites
    2.7. Player movement is allowed via arrow keys to traverse maze, collision with walls and modifiers on blocks are handled by pygame
3. Upon player reaching treasure:
    3.1. Score is increased by 1
    3.2. Every tenth completion 1 life point is awarded to the player (originally begins at 3)
    3.3. Grid size is increased
    3.4. Braid amount is reduced
    3.5. Ease is reduced
    3.6. New treasure location generated; player location remains same
    3.7. Objective 2.3 onwards to 2.7 is completed
4. Upon player running out of time:
    4.1. Score is decreased by 1
    4.2. 1 life point is deducted
    4.3. Grid size is decreased
    4.4. Braid amount is increased
    4.5. Ease is increased
    4.6. Objective 3.6 onwards to 3.7 completed
5. Upon life total reaching 0:
    5.1. Playable environment finishes
    5.2. High score written to csv file
    5.3. Objective 1 is completed



6. Maze generation:
    6.1. Implement the following algorithms:
        6.1.1. Binary Tree
        6.1.2. Sidewinder
        6.1.3. Aldous-Broder
        6.1.4. Braiding
        6.1.5. Dijkstra's
    6.2. Algorithm used is dictated as follows:
        6.2.1. If player score < 5, Binary Tree
        6.2.2. If player score >= 5 and < 10, Sidewinder
        6.2.3. If player score >= 10, Aldous-Broder
    6.3. Relevant algorithm is used along with grid size to generate maze
    6.4. During generation, each block has a small chance of spawning a modifier on the block, either increasing or decreasing player movement speed while on the block

Shown above are 2 images showing the initial objectives, every objective has been fulfilled

End user and target audience

Having shown this game to friends interested in playtesting the game. I have mainly had the following feedback:

1. Some mazes feel harder than others even though they have a similar grid size, braid severity and maze algorithm used
2. Once the grid size reaches a high number like 15, the pygame window needs to be moved up on the screen so that the maze can be fully seen

These two topics will be discussed in the Area for improvement section

Areas for improvement

Addressing point 1 in End user and target audience, one place for improvement could be the weights that are assigned to modifiers. Through my own testing, I have seen that the effect of the modifiers is overestimated by Dijkstra's algorithm, where fast modifiers should actually have a higher weight (e.g. 0.75), and slow modifiers should have a lower weight (e.g. 1.5). I suspect this is the case because Dijkstra's assumes the player only moves vertically and horizontally, from the centre of one block to the centre of an adjacent block. Diagonal movement is not considered, and neither is the possibility of a player not moving along the centres of blocks. This results in Dijkstra's algorithm overestimating the amount of time a player spends in a block, therefore overestimating the effect of modifiers.

Additionally, a better maze generation algorithm could have been used instead of Aldous-Broder. There are other, more efficient maze algorithms that produce mazes with no bias that could have been used instead. However, this change would barely make any difference in the game as the maximum grid size is 20 when the value of c is 50. At a grid size this small the loss in time due to using Aldous-Broder is insignificant, and unnoticeable by the player. As discussed in Research 1.3, generating a grid size of 100 using Aldous Broder and Binary

Tree produces a difference of 0.8 seconds between the two algorithms, this difference is definitely noticeable, but the largest maze in this game is 25 times smaller than a maze of size 100 ((100*100) / (20*20) = 25).

I also wanted to add more entities into the game, perhaps something similar to a life increase or time increase placed somewhere inside a maze, with a small chance of being spawned. This would ultimately add more depth to the game as the player would need to choose quickly whether to go for the extra life and risk not making it to the treasure in time or playing safe.

My implementation of Aldous-Broder could have been done better, right now there are many while loops in the function, which loop until the Decider is in the set of acceptable outcomes, e.g. looping through Decider = random.randint(0,3) until Decider is not in [1,2]. This results in time being wasted and makes an already inefficient algorithm even more so. The random library could have been used to simply select a direction to carve from a list, e.g. in the last example of Decider is not in [1,2], could instead be done to random.choice[2,3].

Finally, addressing point 2 in End user and target audience, I could have made the pygame window automatically be made full screen, or placed in the top left corner of the screen, so that the window would not need to be moved manually to continue viewing the full maze.

Conclusion

To conclude, I am content with my technical solution, although some improvements could be made. For example, the UpdatePos method is too large, to the point where it could be split into 3 different methods – one for collision, one for winning a level and one for failing a level. This would avoid having to pass in a large number of parameters into one function. Additionally, a similar issue exists for the Game function, where it could be split into more functions, perhaps one for calculating time, and one for the main game loop. I also have inefficiencies in the implementation of some algorithms, such as Aldous-Broder as discussed in Areas for improvement.

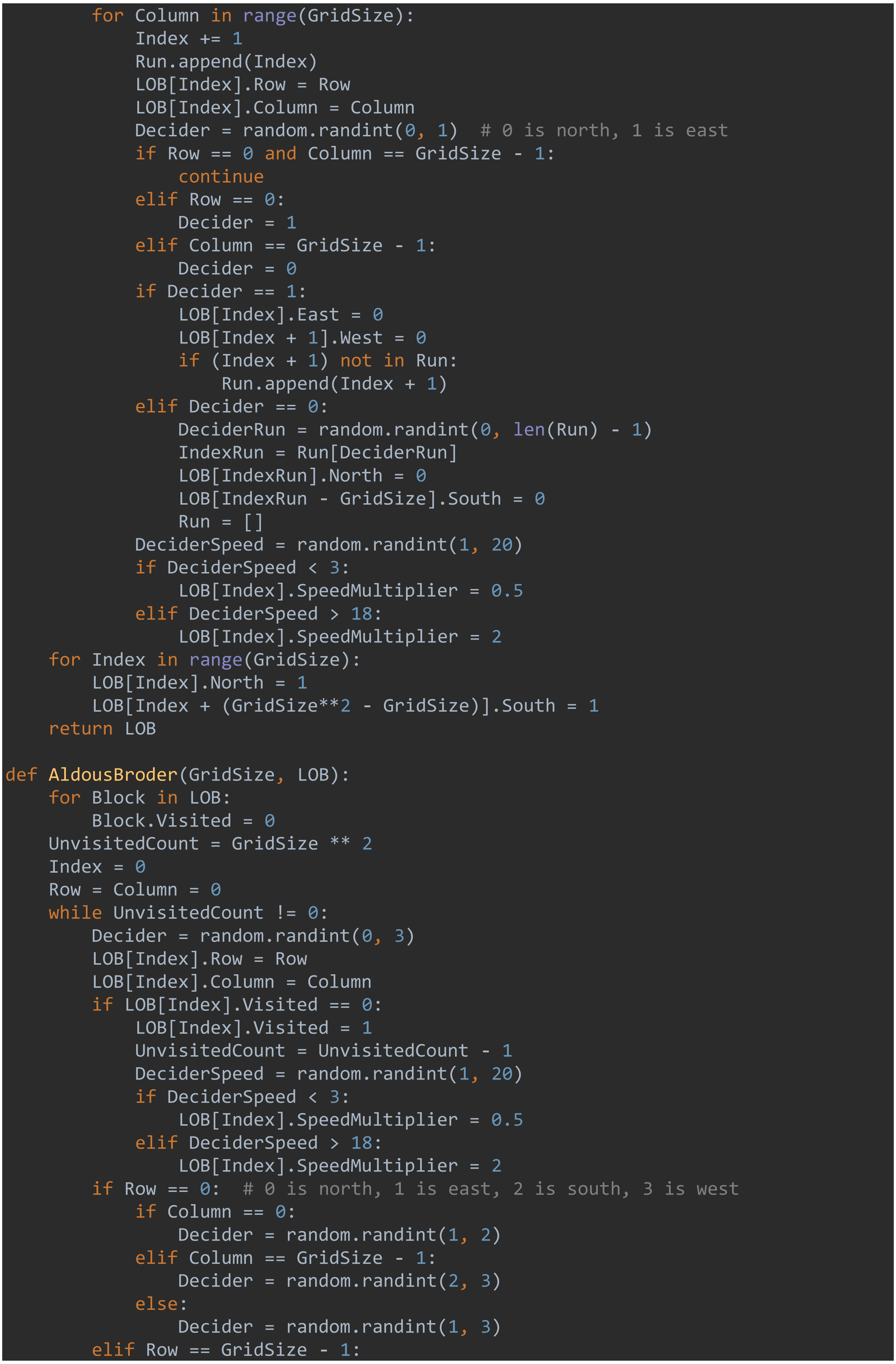
I am happy with the game design although I would have liked to add more depth to the game, in the form of other entities. At some point I wanted to add an enemy AI which hunts the player, although the timing mechanic would need to be changed to be much more lenient in this case, and the mazes would have to be significantly braided such that it would be possible to run away from the enemy AI most of the time. Some of the ideas I had would simply be incompatible with the current game, while others I simply prioritised other aspects of the game and never got around to implementing those ideas.

The aim of this project was to create a random maze game that increased and decreased in difficulty progressively as the player performed better or worse. This main aim has been achieved, even if some aspects have been left out that I at some point thought about adding.

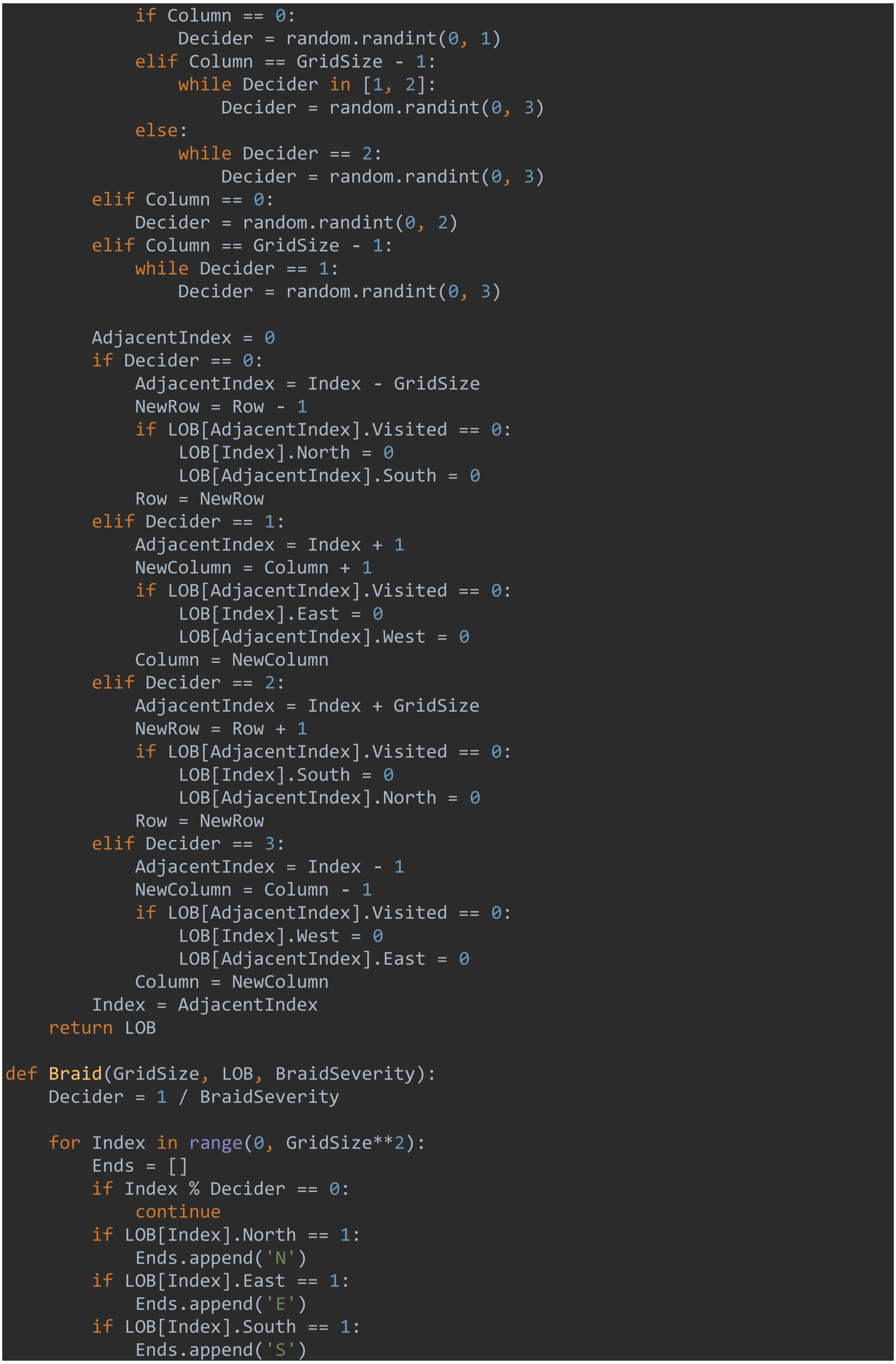
Appendices

Annotated code

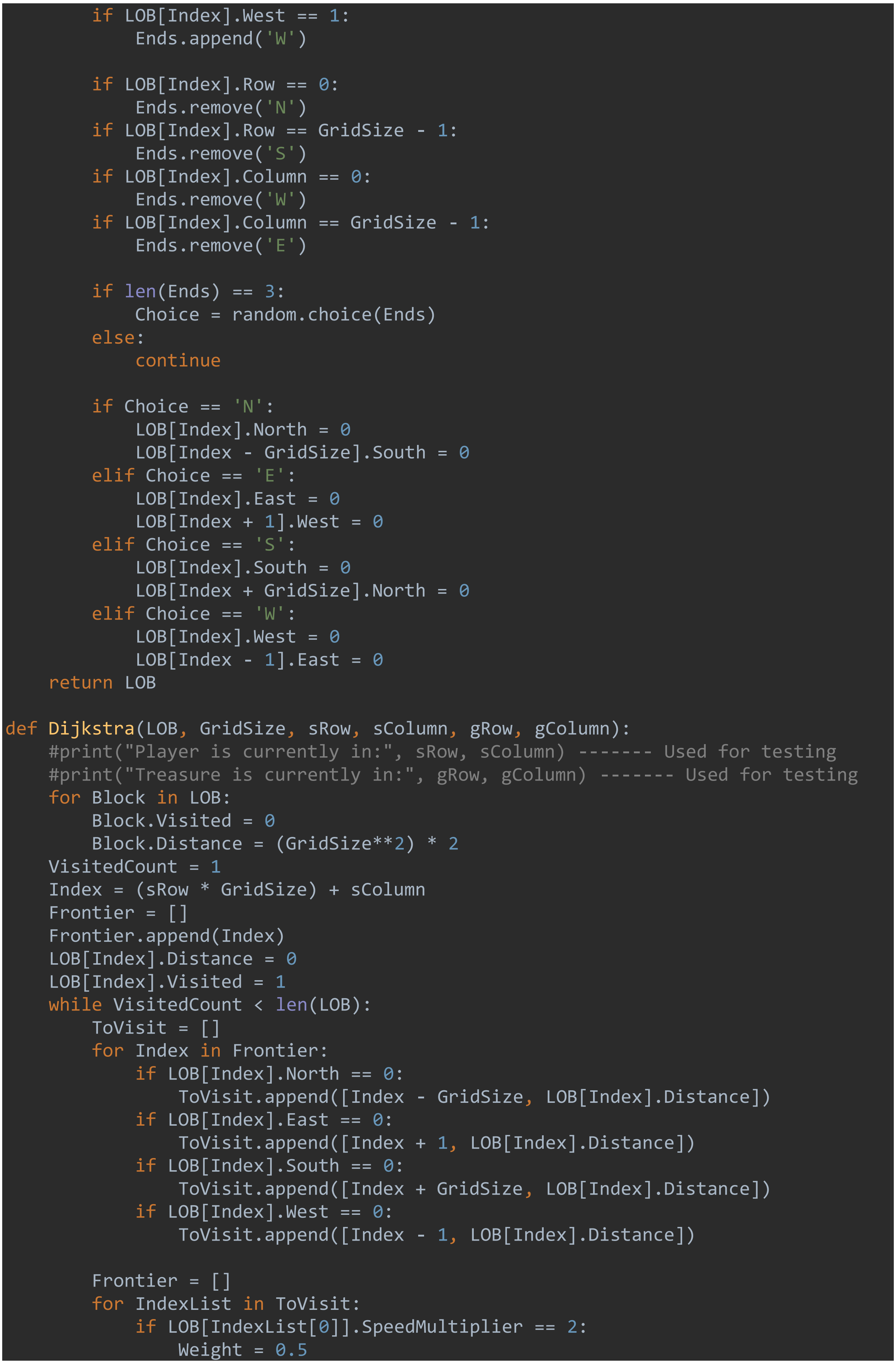
```
import random
import pygame
import csv
import math
from datetime import date

GridSize = 5
BraidSeverity = 0.1

class Block:
    def __init__(self):
        self.Row = -1
        self.Column = -1
        self.North = 1
        self.South = 1
        self.West = 1
        self.East = 1
        self.Visited = 0
        self.Distance = 0
        self.SpeedMultiplier = 1


LOB = [Block() for block in range(GridSize**2)]

def BinaryTree(GridSize, LOB):
    Index = -1
    for Row in range(GridSize):
        for Column in range(GridSize):
            Index += 1
            LOB[Index].Row = Row
            LOB[Index].Column = Column
            Decider = random.randint(0, 1)  # 0 is north, 1 is east
            if Row == 0 and Column == GridSize - 1:
                continue
            elif Row == 0:
                Decider = 1
            elif Column == GridSize - 1:
                Decider = 0
            if Decider == 0:
                LOB[Index].North = 0
                LOB[Index - GridSize].South = 0
            elif Decider == 1:
                LOB[Index].East = 0
                LOB[Index + 1].West = 0
            DeciderSpeed = random.randint(1, 20)
            if DeciderSpeed < 3:
                LOB[Index].SpeedMultiplier = 0.5
            elif DeciderSpeed > 18:
                LOB[Index].SpeedMultiplier = 2
    return LOB

def Sidewinder(GridSize, LOB):
    Index = -1
    Run = []
    for Row in range(GridSize):
```



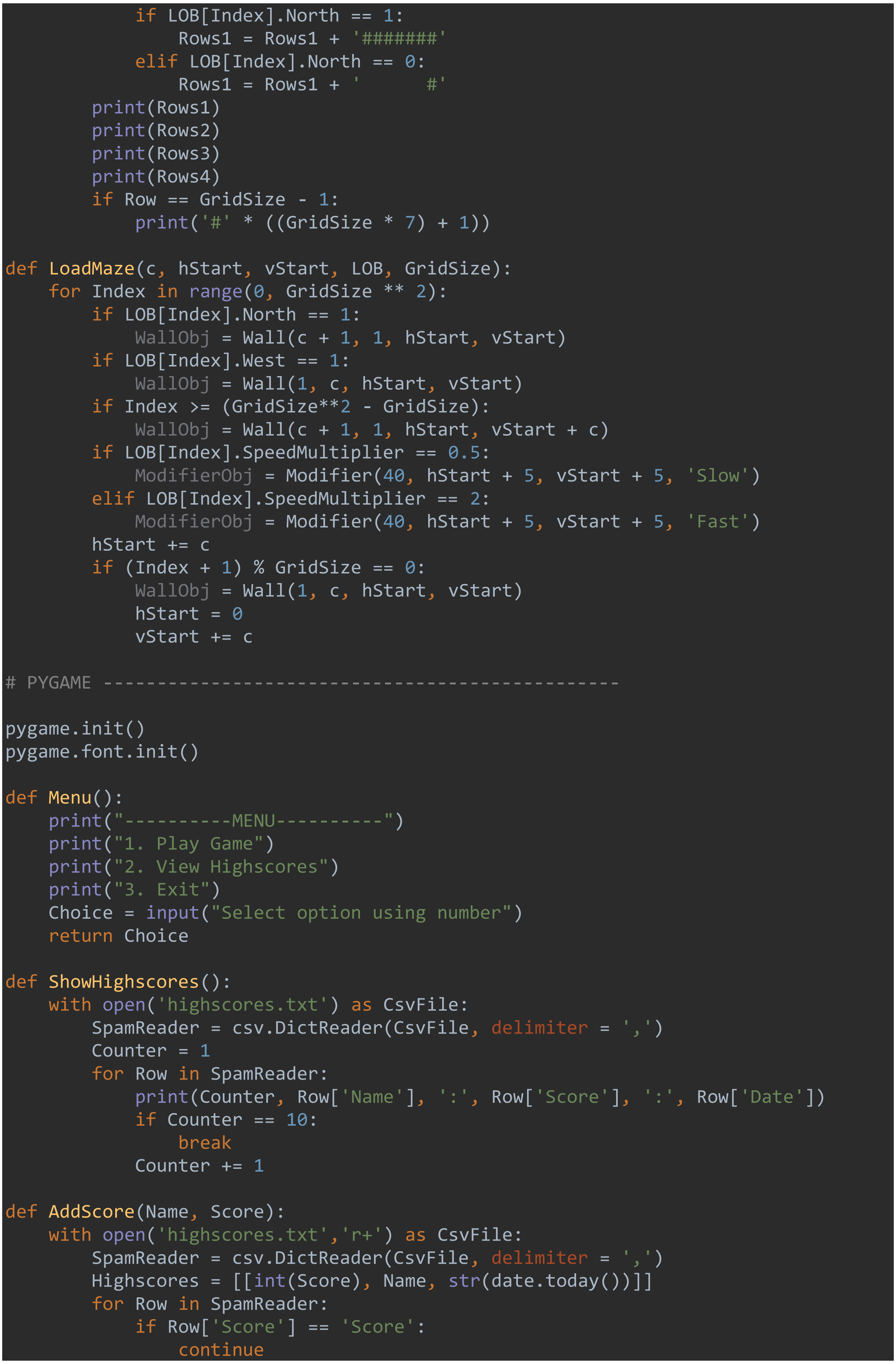
```
        for Column in range(GridSize):
            Index += 1
            Run.append(Index)
            LOB[Index].Row = Row
            LOB[Index].Column = Column
            Decider = random.randint(0, 1)  # 0 is north, 1 is east
            if Row == 0 and Column == GridSize - 1:
                continue
            elif Row == 0:
                Decider = 1
            elif Column == GridSize - 1:
                Decider = 0
            if Decider == 1:
                LOB[Index].East = 0
                LOB[Index + 1].West = 0
                if (Index + 1) not in Run:
                    Run.append(Index + 1)
            elif Decider == 0:
                DeciderRun = random.randint(0, len(Run) - 1)
                IndexRun = Run[DeciderRun]
                LOB[IndexRun].North = 0
                LOB[IndexRun - GridSize].South = 0
                Run = []
            DeciderSpeed = random.randint(1, 20)
            if DeciderSpeed < 3:
                LOB[Index].SpeedMultiplier = 0.5
            elif DeciderSpeed > 18:
                LOB[Index].SpeedMultiplier = 2
    for Index in range(GridSize):
        LOB[Index].North = 1
        LOB[Index + (GridSize**2 - GridSize)].South = 1
    return LOB

def AldousBroder(GridSize, LOB):
    for Block in LOB:
        Block.Visited = 0
    UnvisitedCount = GridSize ** 2
    Index = 0
    Row = Column = 0
    while UnvisitedCount != 0:
        Decider = random.randint(0, 3)
        LOB[Index].Row = Row
        LOB[Index].Column = Column
        if LOB[Index].Visited == 0:
            LOB[Index].Visited = 1
            UnvisitedCount = UnvisitedCount - 1
            DeciderSpeed = random.randint(1, 20)
            if DeciderSpeed < 3:
                LOB[Index].SpeedMultiplier = 0.5
            elif DeciderSpeed > 18:
                LOB[Index].SpeedMultiplier = 2
        if Row == 0:  # 0 is north, 1 is east, 2 is south, 3 is west
            if Column == 0:
                Decider = random.randint(1, 2)
            elif Column == GridSize - 1:
                Decider = random.randint(2, 3)
            else:
                Decider = random.randint(1, 3)
        elif Row == GridSize - 1:
```



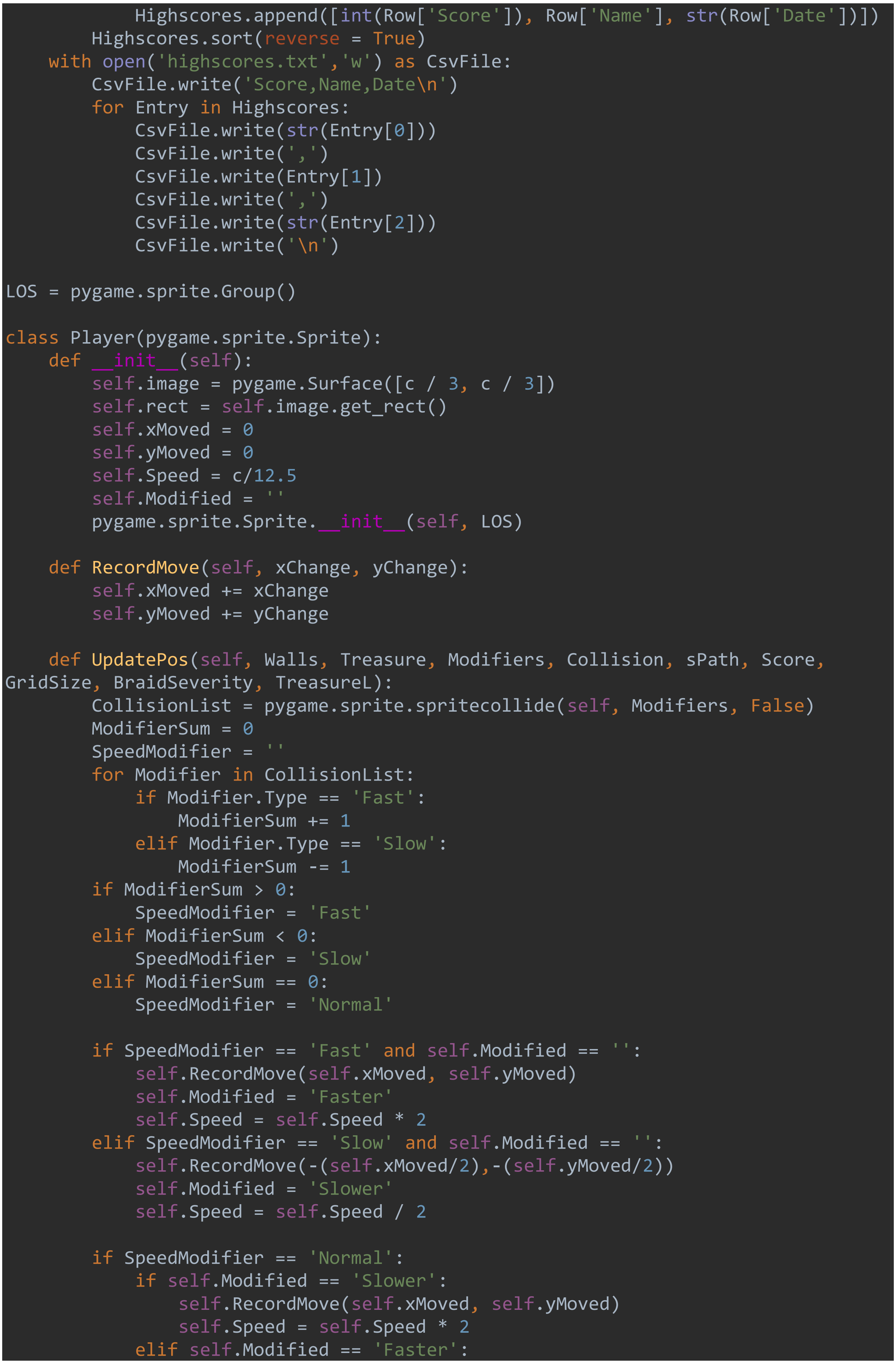
```
            if Column == 0:
                Decider = random.randint(0, 1)
            elif Column == GridSize - 1:
                while Decider in [1, 2]:
                    Decider = random.randint(0, 3)
            else:
                while Decider == 2:
                    Decider = random.randint(0, 3)
        elif Column == 0:
            Decider = random.randint(0, 2)
        elif Column == GridSize - 1:
            while Decider == 1:
                Decider = random.randint(0, 3)

        AdjacentIndex = 0
        if Decider == 0:
            AdjacentIndex = Index - GridSize
            NewRow = Row - 1
            if LOB[AdjacentIndex].Visited == 0:
                LOB[Index].North = 0
                LOB[AdjacentIndex].South = 0
            Row = NewRow
        elif Decider == 1:
            AdjacentIndex = Index + 1
            NewColumn = Column + 1
            if LOB[AdjacentIndex].Visited == 0:
                LOB[Index].East = 0
                LOB[AdjacentIndex].West = 0
            Column = NewColumn
        elif Decider == 2:
            AdjacentIndex = Index + GridSize
            NewRow = Row + 1
            if LOB[AdjacentIndex].Visited == 0:
                LOB[Index].South = 0
                LOB[AdjacentIndex].North = 0
            Row = NewRow
        elif Decider == 3:
            AdjacentIndex = Index - 1
            NewColumn = Column - 1
            if LOB[AdjacentIndex].Visited == 0:
                LOB[Index].West = 0
                LOB[AdjacentIndex].East = 0
            Column = NewColumn
        Index = AdjacentIndex
    return LOB

def Braid(GridSize, LOB, BraidSeverity):
    Decider = 1 / BraidSeverity

    for Index in range(0, GridSize**2):
        Ends = []
        if Index % Decider == 0:
            continue
        if LOB[Index].North == 1:
            Ends.append('N')
        if LOB[Index].East == 1:
            Ends.append('E')
        if LOB[Index].South == 1:
            Ends.append('S')
```



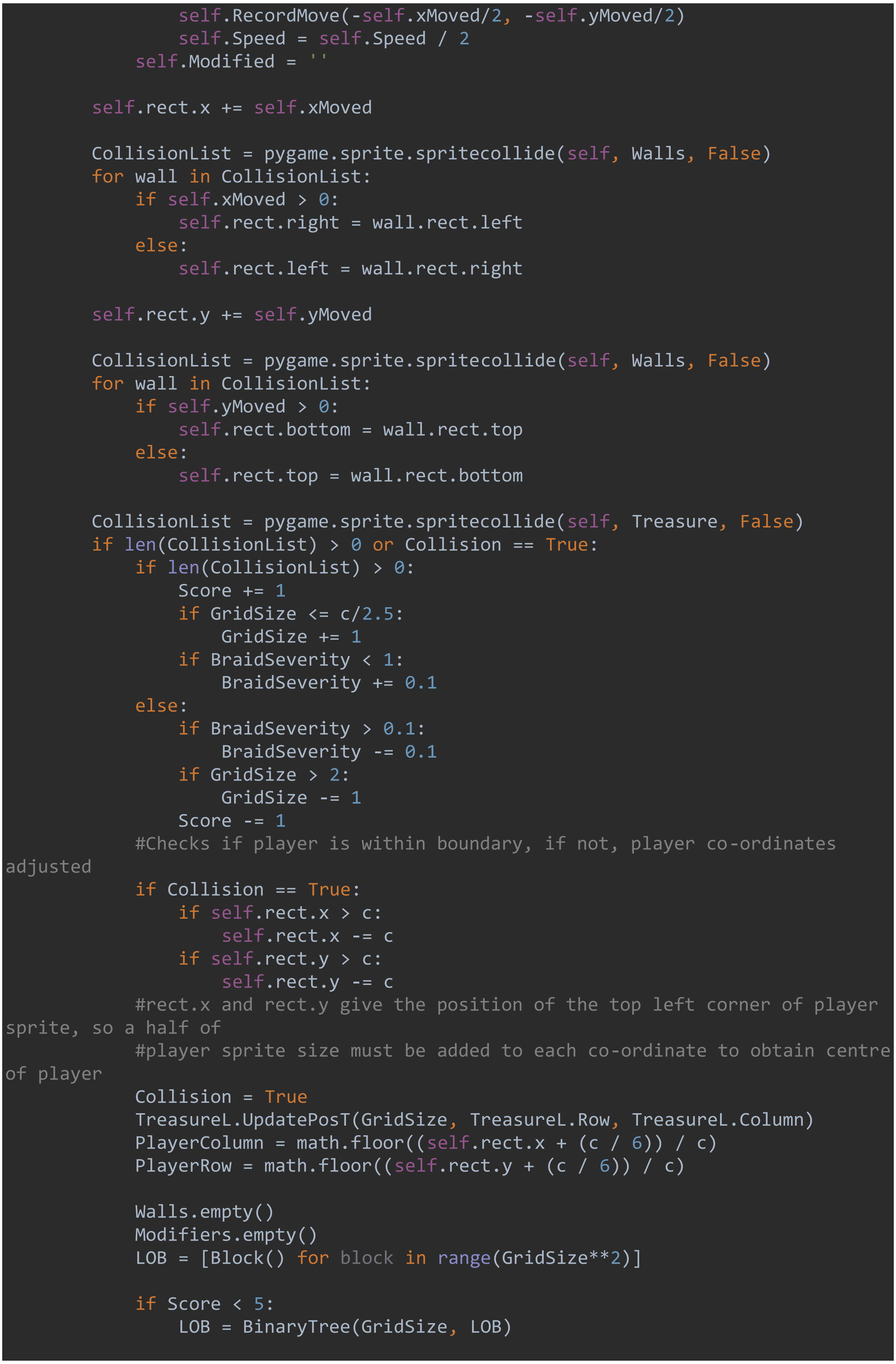
```
        if LOB[Index].West == 1:
            Ends.append('W')

        if LOB[Index].Row == 0:
            Ends.remove('N')
        if LOB[Index].Row == GridSize - 1:
            Ends.remove('S')
        if LOB[Index].Column == 0:
            Ends.remove('W')
        if LOB[Index].Column == GridSize - 1:
            Ends.remove('E')

        if len(Ends) == 3:
            Choice = random.choice(Ends)
        else:
            continue

        if Choice == 'N':
            LOB[Index].North = 0
            LOB[Index - GridSize].South = 0
        elif Choice == 'E':
            LOB[Index].East = 0
            LOB[Index + 1].West = 0
        elif Choice == 'S':
            LOB[Index].South = 0
            LOB[Index + GridSize].North = 0
        elif Choice == 'W':
            LOB[Index].West = 0
            LOB[Index - 1].East = 0
    return LOB

def Dijkstra(LOB, GridSize, sRow, sColumn, gRow, gColumn):
    #print("Player is currently in:", sRow, sColumn) ------- Used for testing
    #print("Treasure is currently in:", gRow, gColumn) ------- Used for testing
    for Block in LOB:
        Block.Visited = 0
        Block.Distance = (GridSize**2) * 2
    VisitedCount = 1
    Index = (sRow * GridSize) + sColumn
    Frontier = []
    Frontier.append(Index)
    LOB[Index].Distance = 0
    LOB[Index].Visited = 1
    while VisitedCount < len(LOB):
        ToVisit = []
        for Index in Frontier:
            if LOB[Index].North == 0:
                ToVisit.append([Index - GridSize, LOB[Index].Distance])
            if LOB[Index].East == 0:
                ToVisit.append([Index + 1, LOB[Index].Distance])
            if LOB[Index].South == 0:
                ToVisit.append([Index + GridSize, LOB[Index].Distance])
            if LOB[Index].West == 0:
                ToVisit.append([Index - 1, LOB[Index].Distance])

        Frontier = []
        for IndexList in ToVisit:
            if LOB[IndexList[0]].SpeedMultiplier == 2:
                Weight = 0.5
                if LOB[IndexList[0]].Distance > (IndexList[1] + Weight):
                    Frontier.append(IndexList[0])
                    LOB[IndexList[0]].Distance = IndexList[1] + Weight
                    if LOB[IndexList[0]].Visited == 0:
                        VisitedCount += 1
                        LOB[IndexList[0]].Visited = 1
                    if LOB[IndexList[0]].Row == gRow and LOB[IndexList[0]].Column == gColumn:
                        #DisplayMazeAscii(GridSize, LOB)# ------- Used for testing
                        return LOB[IndexList[0]].Distance
        for IndexList in ToVisit:
            if LOB[IndexList[0]].SpeedMultiplier == 1:
                Weight = 1
                if LOB[IndexList[0]].Distance > (IndexList[1] + Weight):
                    Frontier.append(IndexList[0])
                    LOB[IndexList[0]].Distance = IndexList[1] + Weight
                    if LOB[IndexList[0]].Visited == 0:
                        VisitedCount += 1
                        LOB[IndexList[0]].Visited = 1
                    if LOB[IndexList[0]].Row == gRow and LOB[IndexList[0]].Column == gColumn:
                        #DisplayMazeAscii(GridSize, LOB)# ------- Used for testing
                        return LOB[IndexList[0]].Distance
        for IndexList in ToVisit:
            if LOB[IndexList[0]].SpeedMultiplier == 0.5:
                Weight = 2
                if LOB[IndexList[0]].Distance > (IndexList[1] + Weight):
                    Frontier.append(IndexList[0])
                    LOB[IndexList[0]].Distance = IndexList[1] + Weight
                    if LOB[IndexList[0]].Visited == 0:
                        VisitedCount += 1
                        LOB[IndexList[0]].Visited = 1
                    if LOB[IndexList[0]].Row == gRow and LOB[IndexList[0]].Column == gColumn:
                        #DisplayMazeAscii(GridSize, LOB)# ------- Used for testing
                        return LOB[IndexList[0]].Distance
    DisplayMazeAscii(GridSize, LOB)
    return LOB[(gRow * GridSize) + gColumn].Distance

def DisplayMazeAscii(GridSize, LOB): #Used for testing
    Index = -1
    for Row in range(GridSize):
        Rows1, Rows2, Rows3, Rows4 = '', '', '', ''
        for Column in range(GridSize):
            Index += 1
            if Column == 0:
                Rows1 = '#'
                Rows2 = '#'
                Rows3 = '#'
                Rows4 = '#'
            Spaces = (6 - len(str(LOB[Index].Distance))) * ' '
            if LOB[Index].East == 1:  # 6 spaces before #
                Rows2 = Rows2 + '      #'
                Rows3 = Rows3 + str(LOB[Index].Distance) + Spaces + '#'
                Rows4 = Rows4 + '      #'
            elif LOB[Index].East == 0:
                Rows2 = Rows2 + '       '
                Rows3 = Rows3 + str(LOB[Index].Distance) + Spaces + ' '
                Rows4 = Rows4 + '       '
```



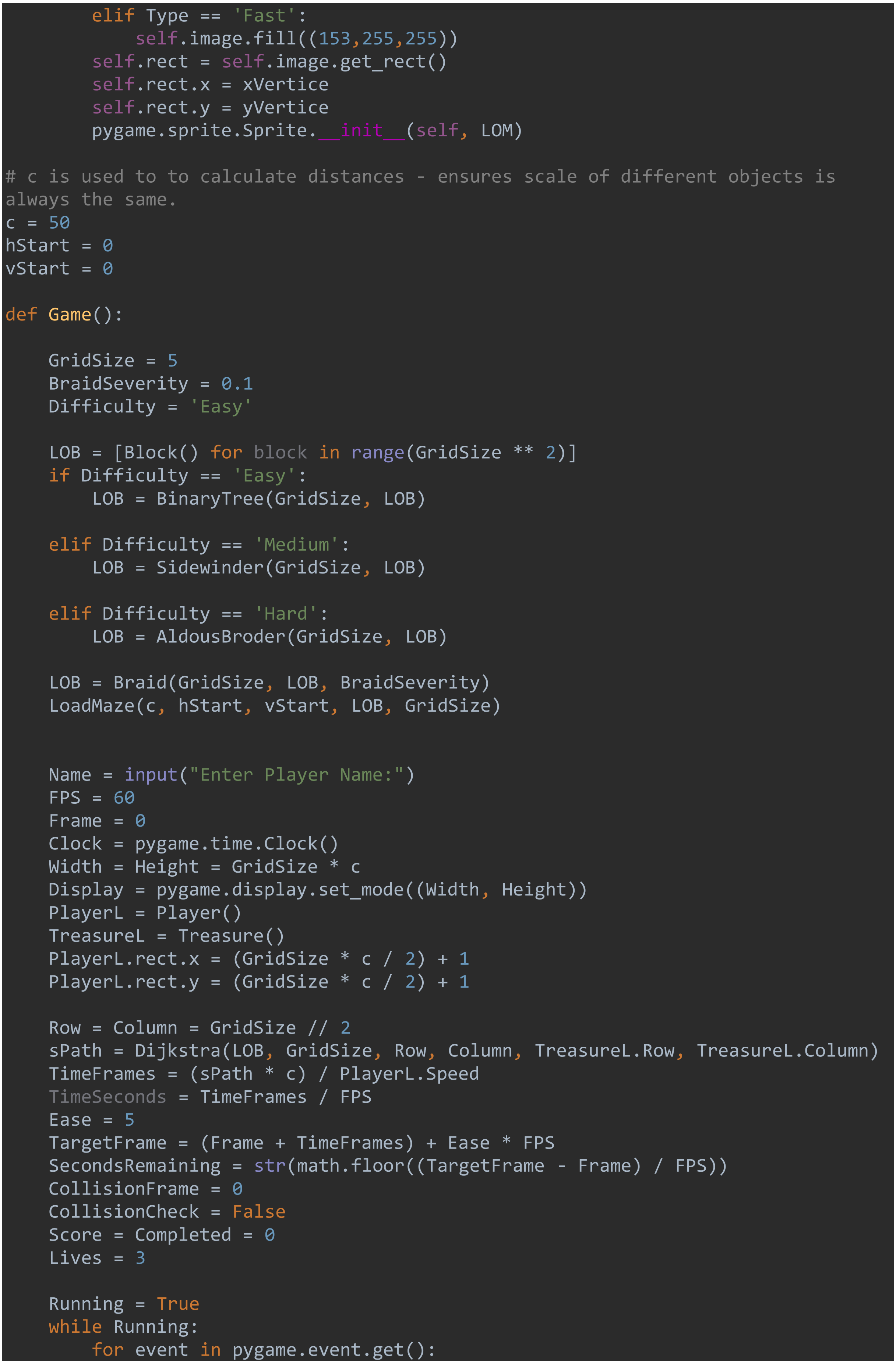
```
            if LOB[Index].North == 1:
                Rows1 = Rows1 + '#######'
            elif LOB[Index].North == 0:
                Rows1 = Rows1 + '      #'
        print(Rows1)
        print(Rows2)
        print(Rows3)
        print(Rows4)
        if Row == GridSize - 1:
            print('#' * ((GridSize * 7) + 1))

def LoadMaze(c, hStart, vStart, LOB, GridSize):
    for Index in range(0, GridSize ** 2):
        if LOB[Index].North == 1:
            WallObj = Wall(c + 1, 1, hStart, vStart)
        if LOB[Index].West == 1:
            WallObj = Wall(1, c, hStart, vStart)
        if Index >= (GridSize**2 - GridSize):
            WallObj = Wall(c + 1, 1, hStart, vStart + c)
        if LOB[Index].SpeedMultiplier == 0.5:
            ModifierObj = Modifier(40, hStart + 5, vStart + 5, 'Slow')
        elif LOB[Index].SpeedMultiplier == 2:
            ModifierObj = Modifier(40, hStart + 5, vStart + 5, 'Fast')
        hStart += c
        if (Index + 1) % GridSize == 0:
            WallObj = Wall(1, c, hStart, vStart)
            hStart = 0
            vStart += c

# PYGAME ------------------------------------------------

pygame.init()
pygame.font.init()

def Menu():
    print("----------MENU----------")
    print("1. Play Game")
    print("2. View Highscores")
    print("3. Exit")
    Choice = input("Select option using number")
    return Choice

def ShowHighscores():
    with open('highscores.txt') as CsvFile:
        SpamReader = csv.DictReader(CsvFile, delimiter = ',')
        Counter = 1
        for Row in SpamReader:
            print(Counter, Row['Name'], ':', Row['Score'], ':', Row['Date'])
            if Counter == 10:
                break
            Counter += 1

def AddScore(Name, Score):
    with open('highscores.txt','r+') as CsvFile:
        SpamReader = csv.DictReader(CsvFile, delimiter = ',')
        Highscores = [[int(Score), Name, str(date.today())]]
        for Row in SpamReader:
            if Row['Score'] == 'Score':
                continue
```



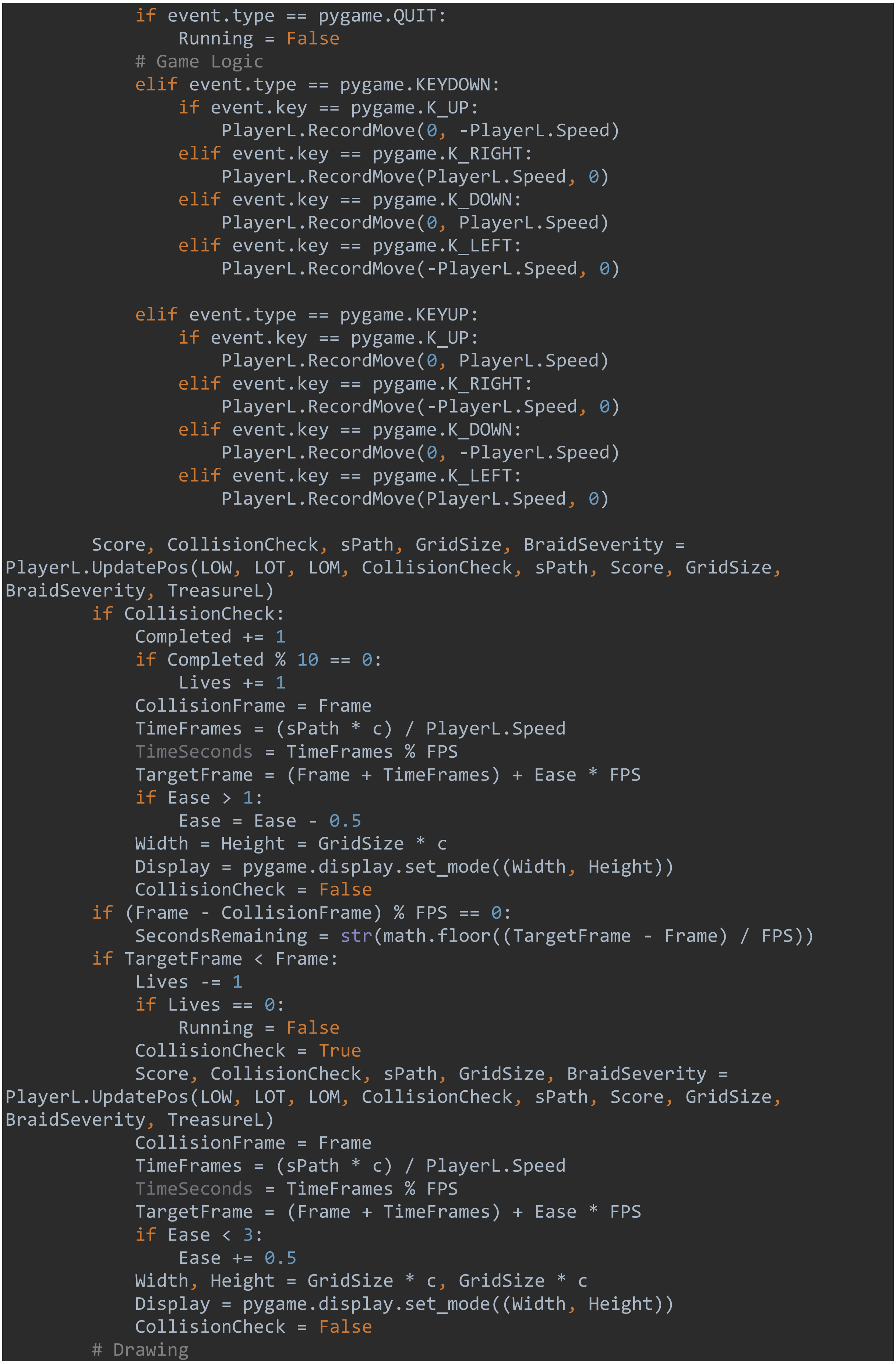
```
            Highscores.append([int(Row['Score']), Row['Name'], str(Row['Date'])])
        Highscores.sort(reverse = True)
    with open('highscores.txt','w') as CsvFile:
        CsvFile.write('Score,Name,Date\n')
        for Entry in Highscores:
            CsvFile.write(str(Entry[0]))
            CsvFile.write(',')
            CsvFile.write(Entry[1])
            CsvFile.write(',')
            CsvFile.write(str(Entry[2]))
            CsvFile.write('\n')

LOS = pygame.sprite.Group()

class Player(pygame.sprite.Sprite):
    def __init__(self):
        self.image = pygame.Surface([c / 3, c / 3])
        self.rect = self.image.get_rect()
        self.xMoved = 0
        self.yMoved = 0
        self.Speed = c/12.5
        self.Modified = ''
        pygame.sprite.Sprite.__init__(self, LOS)

    def RecordMove(self, xChange, yChange):
        self.xMoved += xChange
        self.yMoved += yChange

    def UpdatePos(self, Walls, Treasure, Modifiers, Collision, sPath, Score,
GridSize, BraidSeverity, TreasureL):
        CollisionList = pygame.sprite.spritecollide(self, Modifiers, False)
        ModifierSum = 0
        SpeedModifier = ''
        for Modifier in CollisionList:
            if Modifier.Type == 'Fast':
                ModifierSum += 1
            elif Modifier.Type == 'Slow':
                ModifierSum -= 1
        if ModifierSum > 0:
            SpeedModifier = 'Fast'
        elif ModifierSum < 0:
            SpeedModifier = 'Slow'
        elif ModifierSum == 0:
            SpeedModifier = 'Normal'

        if SpeedModifier == 'Fast' and self.Modified == '':
            self.RecordMove(self.xMoved, self.yMoved)
            self.Modified = 'Faster'
            self.Speed = self.Speed * 2
        elif SpeedModifier == 'Slow' and self.Modified == '':
            self.RecordMove(-(self.xMoved/2),-(self.yMoved/2))
            self.Modified = 'Slower'
            self.Speed = self.Speed / 2

        if SpeedModifier == 'Normal':
            if self.Modified == 'Slower':
                self.RecordMove(self.xMoved, self.yMoved)
                self.Speed = self.Speed * 2
            elif self.Modified == 'Faster':
```



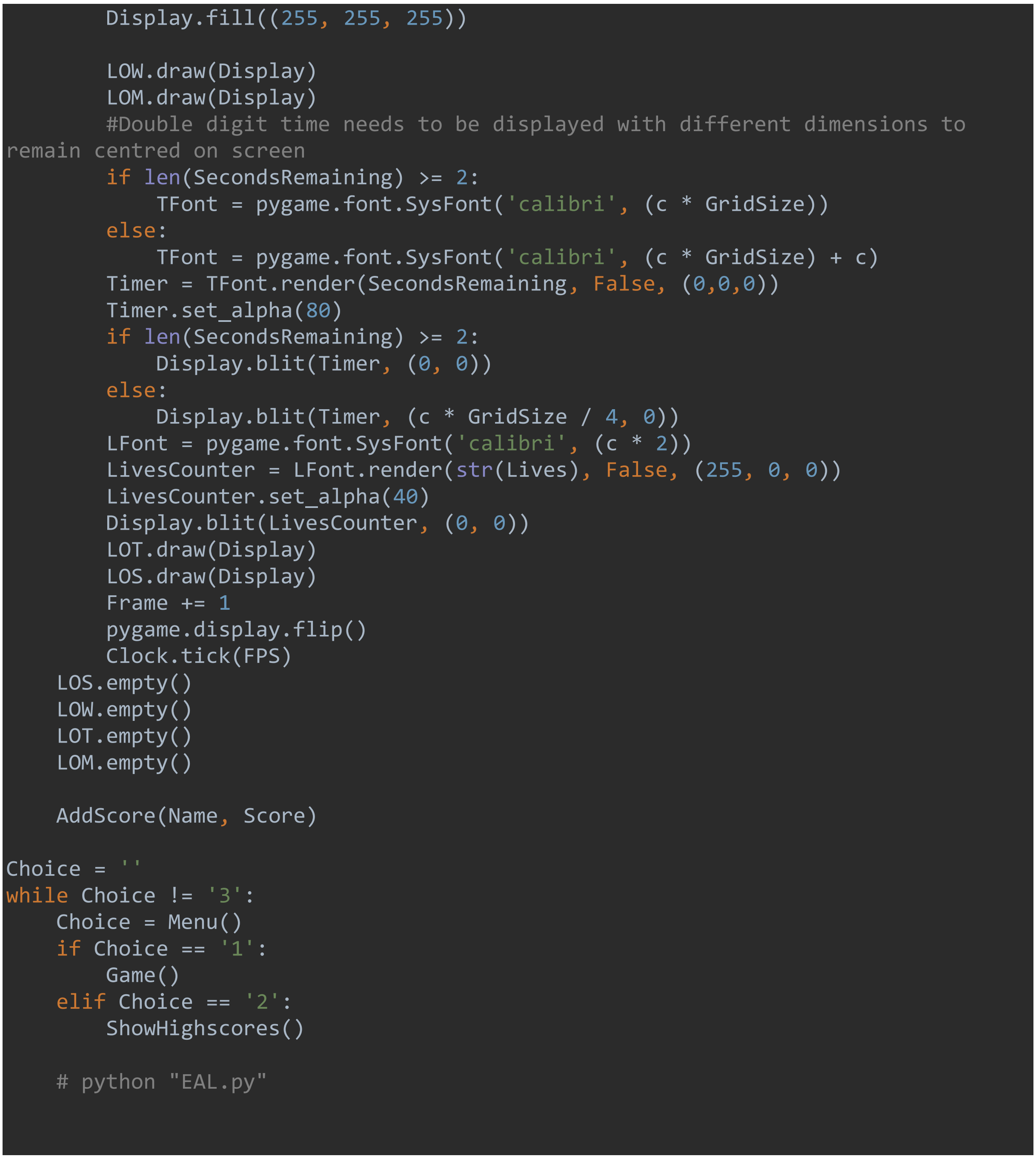
```
                self.RecordMove(-self.xMoved/2, -self.yMoved/2)
                self.Speed = self.Speed / 2
            self.Modified = ''

        self.rect.x += self.xMoved

        CollisionList = pygame.sprite.spritecollide(self, Walls, False)
        for wall in CollisionList:
            if self.xMoved > 0:
                self.rect.right = wall.rect.left
            else:
                self.rect.left = wall.rect.right

        self.rect.y += self.yMoved

        CollisionList = pygame.sprite.spritecollide(self, Walls, False)
        for wall in CollisionList:
            if self.yMoved > 0:
                self.rect.bottom = wall.rect.top
            else:
                self.rect.top = wall.rect.bottom

        CollisionList = pygame.sprite.spritecollide(self, Treasure, False)
        if len(CollisionList) > 0 or Collision == True:
            if len(CollisionList) > 0:
                Score += 1
                if GridSize <= c/2.5:
                    GridSize += 1
                if BraidSeverity < 1:
                    BraidSeverity += 0.1
            else:
                if BraidSeverity > 0.1:
                    BraidSeverity -= 0.1
                if GridSize > 2:
                    GridSize -= 1
                Score -= 1
            #Checks if player is within boundary, if not, player co-ordinates
adjusted
            if Collision == True:
                if self.rect.x > c:
                    self.rect.x -= c
                if self.rect.y > c:
                    self.rect.y -= c
            #rect.x and rect.y give the position of the top left corner of player
sprite, so a half of
            #player sprite size must be added to each co-ordinate to obtain centre
of player
            Collision = True
            TreasureL.UpdatePosT(GridSize, TreasureL.Row, TreasureL.Column)
            PlayerColumn = math.floor((self.rect.x + (c / 6)) / c)
            PlayerRow = math.floor((self.rect.y + (c / 6)) / c)

            Walls.empty()
            Modifiers.empty()
            LOB = [Block() for block in range(GridSize**2)]

            if Score < 5:
                LOB = BinaryTree(GridSize, LOB)
            if Score >=5 and Score < 10:
                LOB = Sidewinder(GridSize, LOB)

            elif Score >= 10:
                LOB = AldousBroder(GridSize, LOB)

            LOB = Braid(GridSize, LOB, BraidSeverity)
            LoadMaze(c, hStart, vStart, LOB, GridSize)
            try:
                sPath = Dijkstra(LOB, GridSize, PlayerRow, PlayerColumn,
TreasureL.Row, TreasureL.Column)
            except IndexError:
                return self.UpdatePos(Walls, Treasure, Modifiers, Collision,
sPath, Score, GridSize, BraidSeverity,
                          TreasureL)

        return Score, Collision, sPath, GridSize, BraidSeverity

LOT = pygame.sprite.Group()
class Treasure(pygame.sprite.Sprite):
    def __init__(self):
        Image = pygame.image.load("treasure.png")
        Image.convert_alpha()
        self.Row = random.randint(0, GridSize - 1)
        self.Column = random.randint(0, GridSize - 1)
        self.image = pygame.transform.scale(Image, (c // 3, c // 3))
        self.rect = self.image.get_rect()
        self.rect.x = (self.Column * c) + c // 3
        self.rect.y = (self.Row * c) + c // 3
        pygame.sprite.Sprite.__init__(self, LOT)

    def UpdatePosT(self, GridSize, OldRow, OldColumn):
        self.Row = random.randint(0, GridSize-1)
        self.Column = random.randint(0, GridSize-1)
        while self.Row == OldRow:
            self.Row = random.randint(0, GridSize - 1)
        while self.Column == OldColumn:
            self.Column = random.randint(0, GridSize - 1)
        self.rect.x = (self.Column * c) + c // 3
        self.rect.y = (self.Row * c) + c // 3

LOW = pygame.sprite.Group()
class Wall(pygame.sprite.Sprite):
    def __init__(self, Width, Height, xVertice, yVertice):
        self.image = pygame.Surface([Width, Height])
        self.Width = Width
        self.Height = Height
        self.rect = self.image.get_rect()
        self.rect.x = xVertice
        self.rect.y = yVertice
        pygame.sprite.Sprite.__init__(self, LOW)

LOM = pygame.sprite.Group()
class Modifier(pygame.sprite.Sprite):
    def __init__(self, Size,  xVertice, yVertice, Type):
        self.image = pygame.Surface([Size, Size]) #Square so width = height
        self.Type = Type
        if Type == 'Slow':
            self.image.fill((255,204,153))
```



```
        elif Type == 'Fast':
            self.image.fill((153,255,255))
        self.rect = self.image.get_rect()
        self.rect.x = xVertice
        self.rect.y = yVertice
        pygame.sprite.Sprite.__init__(self, LOM)

# c is used to to calculate distances - ensures scale of different objects is always the same.
c = 50
hStart = 0
vStart = 0

def Game():

    GridSize = 5
    BraidSeverity = 0.1
    Difficulty = 'Easy'

    LOB = [Block() for block in range(GridSize ** 2)]
    if Difficulty == 'Easy':
        LOB = BinaryTree(GridSize, LOB)

    elif Difficulty == 'Medium':
        LOB = Sidewinder(GridSize, LOB)

    elif Difficulty == 'Hard':
        LOB = AldousBroder(GridSize, LOB)

    LOB = Braid(GridSize, LOB, BraidSeverity)
    LoadMaze(c, hStart, vStart, LOB, GridSize)


    Name = input("Enter Player Name:")
    FPS = 60
    Frame = 0
    Clock = pygame.time.Clock()
    Width = Height = GridSize * c
    Display = pygame.display.set_mode((Width, Height))
    PlayerL = Player()
    TreasureL = Treasure()
    PlayerL.rect.x = (GridSize * c / 2) + 1
    PlayerL.rect.y = (GridSize * c / 2) + 1

    Row = Column = GridSize // 2
    sPath = Dijkstra(LOB, GridSize, Row, Column, TreasureL.Row, TreasureL.Column)
    TimeFrames = (sPath * c) / PlayerL.Speed
    TimeSeconds = TimeFrames / FPS
    Ease = 5
    TargetFrame = (Frame + TimeFrames) + Ease * FPS
    SecondsRemaining = str(math.floor((TargetFrame - Frame) / FPS))
    CollisionFrame = 0
    CollisionCheck = False
    Score = Completed = 0
    Lives = 3

    Running = True
    while Running:
        for event in pygame.event.get():
```



```
            if event.type == pygame.QUIT:
                Running = False
            # Game Logic
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_UP:
                    PlayerL.RecordMove(0, -PlayerL.Speed)
                elif event.key == pygame.K_RIGHT:
                    PlayerL.RecordMove(PlayerL.Speed, 0)
                elif event.key == pygame.K_DOWN:
                    PlayerL.RecordMove(0, PlayerL.Speed)
                elif event.key == pygame.K_LEFT:
                    PlayerL.RecordMove(-PlayerL.Speed, 0)

            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_UP:
                    PlayerL.RecordMove(0, PlayerL.Speed)
                elif event.key == pygame.K_RIGHT:
                    PlayerL.RecordMove(-PlayerL.Speed, 0)
                elif event.key == pygame.K_DOWN:
                    PlayerL.RecordMove(0, -PlayerL.Speed)
                elif event.key == pygame.K_LEFT:
                    PlayerL.RecordMove(PlayerL.Speed, 0)

        Score, CollisionCheck, sPath, GridSize, BraidSeverity =
PlayerL.UpdatePos(LOW, LOT, LOM, CollisionCheck, sPath, Score, GridSize,
BraidSeverity, TreasureL)
        if CollisionCheck:
            Completed += 1
            if Completed % 10 == 0:
                Lives += 1
            CollisionFrame = Frame
            TimeFrames = (sPath * c) / PlayerL.Speed
            TimeSeconds = TimeFrames % FPS
            TargetFrame = (Frame + TimeFrames) + Ease * FPS
            if Ease > 1:
                Ease = Ease - 0.5
            Width = Height = GridSize * c
            Display = pygame.display.set_mode((Width, Height))
            CollisionCheck = False
        if (Frame - CollisionFrame) % FPS == 0:
            SecondsRemaining = str(math.floor((TargetFrame - Frame) / FPS))
        if TargetFrame < Frame:
            Lives -= 1
            if Lives == 0:
                Running = False
            CollisionCheck = True
            Score, CollisionCheck, sPath, GridSize, BraidSeverity =
PlayerL.UpdatePos(LOW, LOT, LOM, CollisionCheck, sPath, Score, GridSize,
BraidSeverity, TreasureL)
            CollisionFrame = Frame
            TimeFrames = (sPath * c) / PlayerL.Speed
            TimeSeconds = TimeFrames % FPS
            TargetFrame = (Frame + TimeFrames) + Ease * FPS
            if Ease < 3:
                Ease += 0.5
            Width, Height = GridSize * c, GridSize * c
            Display = pygame.display.set_mode((Width, Height))
            CollisionCheck = False
        # Drawing
```



```
        Display.fill((255, 255, 255))

        LOW.draw(Display)
        LOM.draw(Display)
        #Double digit time needs to be displayed with different dimensions to 
remain centred on screen
        if len(SecondsRemaining) >= 2:
            TFont = pygame.font.SysFont('calibri', (c * GridSize))
        else:
            TFont = pygame.font.SysFont('calibri', (c * GridSize) + c)
        Timer = TFont.render(SecondsRemaining, False, (0,0,0))
        Timer.set_alpha(80)
        if len(SecondsRemaining) >= 2:
            Display.blit(Timer, (0, 0))
        else:
            Display.blit(Timer, (c * GridSize / 4, 0))
        LFont = pygame.font.SysFont('calibri', (c * 2))
        LivesCounter = LFont.render(str(Lives), False, (255, 0, 0))
        LivesCounter.set_alpha(40)
        Display.blit(LivesCounter, (0, 0))
        LOT.draw(Display)
        LOS.draw(Display)
        Frame += 1
        pygame.display.flip()
        Clock.tick(FPS)
    LOS.empty()
    LOW.empty()
    LOT.empty()
    LOM.empty()

    AddScore(Name, Score)

Choice = ''
while Choice != '3':
    Choice = Menu()
    if Choice == '1':
        Game()
    elif Choice == '2':
        ShowHighscores()

    # python "EAL.py"
```