

Thomas Butterfield - 6009

Queen Elizabeth Grammar School - 38255

H446 Component 3

Analysis	2
Overview	2
Solvability	5
Stakeholders	5
Current System	6
Key Features	8
Issues	11
Software and Hardware Requirements	11
Limitations	12
Requirements	13
Development	15
Design	15
Testing	23
Design	38
Testing	52
Design	57
Testing	66
Design	70
Testing	75
Design	78
Testing	80
Evaluation (Testing)	85
Evaluation	96

Project

Analysis

Overview

For my project I am going to create a 'boggle' game. Boggle is a word game, which was originally designed by Allan Turoff and distributed by Parker Brothers. The game is usually played using a 4 x 4 plastic grid of lettered dice, and the players attempt to find words in sequences of adjacent letters in this grid.

In my school, the modern foreign languages department is always looking for more ways to introduce fun vocab testing into lessons, and I believe that a word game would be a popular thing for students to play. After talking to the head of the languages department, Miss Duckitt, she also thinks that this would be a popular idea, so that is why I have chosen this particular project.

The game begins by shaking the tray of 16 dice, each with a different letter printed on each of its sides. The players then have three minutes to find as many words as they can, following the rules of the game:

1. The letters must be adjoined in a 'chain'. Letters in the chain may be adjacent horizontally, vertically or diagonally.
2. Words must contain at least three letters.
3. No letter cube may be used more than once within a single word.

An example is shown below:



This is an example of the word “super” being found using this randomly generated grid. This word has been found according to all of the rules of the game, in that all of the letters in the chain are adjacent to each other, the word is longer than three letters, and no letter cube is used more than once.

Other words which could be found legally using this grid include: “surely”, “truly”, “key”, “turn”, as well as many others.

Words such as “hi” could not be made legally, as they are less than three letters long, and therefore are not allowed according to the rules.

Another example of a word which cannot be made legally using this board is “help”, because, while the grid does contain all of those letters, they cannot be connected together in a chain, as the letters “H” and “E” are not adjacent to each other. The word “dig” cannot be made legally for the same reason.

When the timer is over, the players work out how many points they got, the longer the word, the more points it is worth:

Letters:	Points:
3 or 4 letters	1 point
5 letters	2 points
6 letters	3 points
7 letters	5 points
8+ letters	11 points



Figure 1 From WikiHow. Creative Commons Licence



Figure 2 From Wikimedia. Creative Commons Licence



Figure 3 Official rules sheet

Solvability

My project will be solvable using computational methods, as the current version of this game is played using a physical game set, which makes it more difficult, and time consuming to play, because there are parts that can break, and the players have to look up the words themselves, whereas with a computerised version, there are no physical parts that can break, and the computer will check all of the words, rather than the players having to do that themselves, saving everybody time, and making the game progress much faster, and more smoothly. Another big advantage to playing the game on a computer is that the school would then not have to purchase multiple physical versions of the board game, and the entire class will be able to play the game at the same time, saving time and money.

Some of the reason why a computer is particularly suited to this task, is that a computer can process data very quickly, and find all of the words in a grid much, much faster than a human possible could, and the computer will also not make mistakes such as spelling or counting errors.

One of the problems of the physical version of the game is that, when the game is being played, the players have no way of knowing how many words are left, or if they have found all of the words. My system will solve these problems, making it a vast improvement on the current system.

Stakeholders

I am designing this game to be used by the modern languages department of my school, as a vocabulary tester for the students, as such, the game will be primarily designed for students to play, but should also allow the teachers to control the level of complexity of words that the students will be tested on. All of the students will have a computer to themselves, meaning that the game should be designed for one player per monitor. The students should be able to type in their name before they start to play the game, so that, at the end, the teacher can see what score each pupil achieved, and students can compete between themselves to see who is the best at the game.

Current System




Currently, the languages department mainly use regular pen and paper vocabulary tests to test the knowledge of their students, but this can often be tedious, and not very exciting. However, the department does offer their students some online resources to test their vocabulary, such as a website called 'kerboodle', but I know from personal experience that the tasks on these websites can also be tedious to work through, and therefore don't encourage the students to want to learn new vocabulary. Whereas, with a computerised game, the students will be far more keen, and willing to put the effort into learning the words, so that they can do better at the game.

An example of another online system that is currently being used by modern foreign language students at my school is "Pearson Active Learn", this website has online reading and listening exercises, as well as vocab learning and testing sections. An example of these vocab learning and testing sections is shown below.

Ma vie d'internaute

[> Review answers](#)

Module 2 – Le temps des loisirs, Unit 2, Vocab Learn




Total score	Learning aids used	Time taken	Feedback
<div>100%</div>	0	3m 19s	<div> <div>Student comment</div> <div>    </div> <div>Teacher comment</div> </div>

Show details

Ma vie d'internaute

[> Review answers](#)

Module 2 – Le temps des loisirs, Unit 2, Vocab Test

Total score	Learning aids used	Time taken	Feedback
<div>71%</div>	0	7m 37s	<div> <div>Student comment</div> <div>    </div> <div>Teacher comment</div> </div>

Show details

When a student is set a vocab learning exercise by their teacher, they complete a number of questions where they are shown the French version along with the English translation, they are then supposed to hide the French version and then type it out in a text box below. The biggest problem with this is that the student does not actually have to hide the French version whilst they are typing out their answer, making it very easy to cheat.

An example of a particular question is shown below.

Vocabulary: sport
?
x

J'oublie mes soucis.

I forget my worries.

Hide

Check

Previous

Progress 20/20

Exit

As you can see, the user can type their answer in just by simply copying out the French version.

One of the more traditional ways in which students are tested, the old fashioned pen and paper method, is shown below here.

On peut	one/we can	plus de	more than
on ne peut pas	one/we can't	espèces	species
Je peux	I can	invertébrés	invertebrates
Je ne peux pas	I can't	les repas	the meals
Nous pouvons	we can	tous les jours	all day
nous ne pouvons pas	we can't	ouvert	open
Ben	(Pause)	minuit	midnight
et puis	and then	germe	closed
Beaucoup de choses	lots of things	J'ai fait	I did
ça	It/this/that	on a marche	we walked
Belle	beautiful	pendant	during/for
en plus	what's more/summers	dur	difficult/hard
passionnant	exciting	on a vu	we saw
déjà	already/seen	plus tard	later
une brochure	leaflets	Il a plu	it rained
Bien sûr	of course	il a fait froid	it was cold
Avez-vous	Have you	on a fait	we did
voilà	here it is	griller	grill
voici	gar	des saucisses	sauces/sausages
un plan	a map	l'escalade	rock climbing
à pied	by foot		

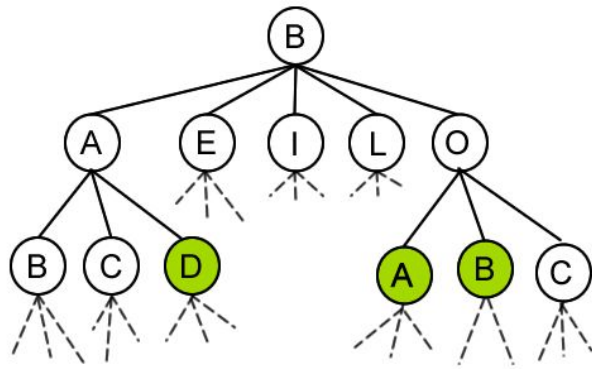
These methods of learning and testing are often criticised as being boring and tedious, and actually not very productive since there are other more enjoyable and worthwhile ways in which students can learn. Another issue which is highlighted by the image shown above is that some students may have handwriting which is difficult to read, even for themselves, which can obviously cause problems for them when they come to try to learn this vocabulary, or if the teacher wants to look at their work. This problem would be solved using a digital solution since all of the text would be using a very clear and easy to read font.

In order to find out how the current systems are working for the pupils, I spoke to some students at my school who are learning French. I asked them what they think about the current methods that the languages department has in place to test their vocabulary knowledge. The majority of them told me that they do not enjoy the tests that they are made to do on pen and paper, and find that they are therefore not particularly engaged by them, so they are not feeling encouraged to put time and effort in to actually learning the vocabulary. I then asked them about the other methods of testing that the school uses, such as the 'kerboodle' website. They all said that this was definitely better than the pen and paper testing, but most of them also said that they still feel as though it is just more work to do, and that they don't actually enjoy doing the exercises, meaning that it can get very boring, and they eventually become disinterested in completing the tasks.

Key Features

The game will be designed using a graphical user interface, because this will be the best way of displaying the features of the game on the screen, as the game relies on a 4x4 grid layout, which would be best presented in a graphical interface format, as opposed to a text-only format. If I were to design the game in a text-only basis, it would not look like the board version, which would make it more difficult to play, as users would not be as familiar with how the game worked and might not be able to recognise the game. A GUI, on the other hand, allows the user to easily interact with and play the game, just as they would do with the board version.

I will use data structures to organise the words that are being used, to allow the game to quickly check if the sequence of letters that a user is trying to use is actually a word in that language. The best data structure to use for this is a trie. A trie is a kind of search tree, it will be used here to hold the dictionary of different languages. The first letter of each word is represented by a root node, leading off from the roots, are child nodes, representing every possible second letter. An advantage of using this structure is that it is very easy to check if a word exists, for example, if I wanted to check if 'BBAL' is a word, using the trie pictured below, I would go to the root node which represents the first B, and look for a child node that represents the second B. I would fail to find it, so I would know immediately that the word doesn't exist, and I only had to make two checks, and I don't have to look any further. This is evidently far quicker, and more efficient than a regular search, such as a linear search, or even a binary search.



Earlier in this 'key features' section, I mentioned that if I were to design the game in a text-based format, it would not look like the actual board version, which would make the game harder for the user to play, as it would appear less familiar to them than a graphical, and visual format would. To demonstrate this, I have included a screengrab of what the game would look like if it were to be played in a text-based format on the next page.

```

-----
Welcome to Boggle
-----

----- Rules -----
1. The letters must be adjoining in a 'chain'. (Letter cubes in the chain may be adjacent horizontally, vertically, or diagonally.)
2. Words must contain at least three letters.
3. No letter cube may be used more than once within a single word.
4. You have 3 minutes to find as many words as you can.
5. The longer the word, the more points you score:

Letters:      Points:
3 or 4 letters 1 point
5 letters     2 points
6 letters     3 points
7 letters     5 points
8+ letters    11 points

----- Menu -----

Choose a language:
1. English
2. French

Your Choice [1-2]: 2

----- Game -----

['I', 'E', 'I', 'E']
['N', 'H', 'S', 'O']
['P', 'I', 'H', 'I']
['E', 'T', 'E', 'A']
There are 153 possible words.
Enter a word. pit
    Correct. Well done!
Enter a word. pit
    You have already used this word.
Enter a word. a
    Words must be at least 3 letters long.
Enter a word. tea

```

The most immediate and obvious issue with this is that it looks nothing like the real version does, which is mainly due to the fact that the grid of letters appear in the same font size and format as everything else on the screen does, which implicitly gives everything the same level of importance, which should not be the case in this game, as clearly, the most important thing should be the grid itself, which is what the entire game revolves around.

Another issue with this format is to do with the output that would be shown after every input by the user. Due to the fact that once text has been printed, there is no way of deleting it, using a text-based format, meaning that after the user has entered several attempts, the text at the top of the screen would soon disappear, as the page automatically scrolls down. I could see that, if you were to play this as a proper game, it would not be long before you have entered so many attempts, that the grid itself is no longer visible in the same page as the words that you are entering. This would be a major issue as it means that the user would have to repeatedly scroll back and forth between the grid, which the game is centered on, and the words which they are entering as guesses. This problem would be exacerbated by the information that the user should be provided with after each attempt that they make, in the example shown above, the game is only giving a single line of output for each input, if their word was correct. However, in my actual (graphical based) game I would like for there to be more output than just this, such as the time that they have remaining, which will be constantly updating and could not practically be outputted in a text based format, as well as the user's current score, which, in this format, would take up an additional line in each attempt, only making the aforementioned problem worse.

In general, a text based version, which looked similar to the mock up shown above, would not be as appealing to my target consumer, school children, as a version which uses a graphical user interface, as I have just explained, and is also fairly clear to see.

Issues

A possible issue that could arise with my project is that users are unable to understand how the game works and what they are supposed to do to play the game, this problem could be solved by adding help features throughout the game, so that, if they are struggling, they will be able to quickly and easily get some help, so that they can carry on playing the game.

Software and Hardware Requirements

In terms of the software requirements, my game will require an IDE on which the program can be executed, due to the fact that it is a python program, which requires specialised software to operate.

Python is cross-platform, meaning that my game will be able to run on virtually any operating system, meaning that users will not encounter any issues or setbacks due to their platform, whether it is Windows, Mac, Linux, or any other operating system.

In terms of hardware requirements, my game will be built to operate on the hardware which my Virtual Machine has, which is 512MB of RAM, as such, the standard requirements to operate the game normally are only 512MB of memory.

This is a very reasonable standard requirement, because I know for certain that it can be met by my main target client, the languages department at my school, because they use the exact same specification PCs as what I am using to design and operate my game, meaning that the performance that I experience when running the game will be exactly the same as what the users themselves will experience.

Of course additional memory and/or processors will make the game run faster, but the effects of any additional hardware will not be noticeable whilst the game is being played, the difference will only become apparent during the creation of the trie at the beginning of the game, which is the most time consuming process in the entire game.

Limitations

One of the potential limitations which faces my proposed solution is that, the game will only be able to be played using a version of python, meaning that users must have some kind of IDE installed on their computer in order to properly play the game. Similar to this, there is also the issue of some students not being able to operate the IDE in order to start the game and operate it properly, even if they do have it installed on their computers. A solution to these problems would be for the school to install a form of an IDE on every computer in the modern languages department, and to then instruct the students on how to operate the IDE in order to play the game as it was intended to be played.

A limitation of the advancement of my game is the systems that my school has in place for security purposes. As I will mention in the next section, I would like to be able to make the game playable in a multiplayer version, however, I will encounter some issues with designing this, because two computers on the school network are not allowed to communicate directly with each other.

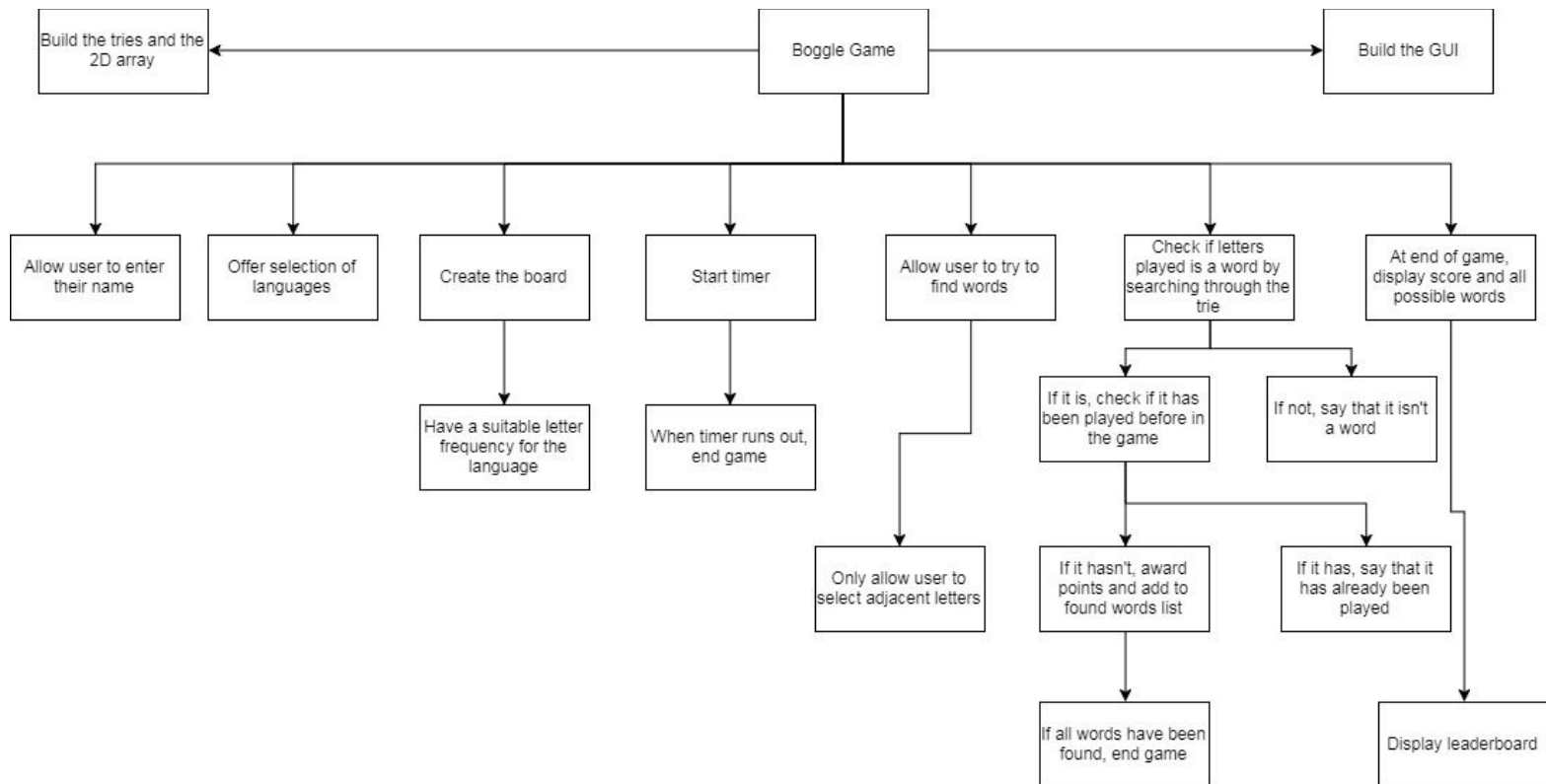
Requirements

For my game to be successful, it must do the following:

1. Allow the user to play a game of boggle, in their chosen language.
2. Help the user to learn the vocabulary of their chosen language in an enjoyable way. I will test this by asking users to test the game at a suitable stage in the development process.
3. Build a trie for the word set at the start of the game, so that the word list can be quickly accessed and used during the game.
4. The game should be able to load in less than 15 seconds, and whilst it is loading, it should display a loading screen. This is to ensure that users do not get bored or think that it has crashed.
5. The game must display a 4x4 grid of letters.
6. These letters should appear proportional to their commonality in the given language to enable players to make more words in a given language.
7. Must have an attractive and interesting GUI. This is so that it will appeal to the users, who will primarily be school-children, if the GUI is dull, then children will not be very interested in what the game has to offer, because it looks boring, so they will not engage fully with the game. I will judge this by talking to my end user and asking them what they think of current designs, and how they could be improved, and also by testing my design by asking students what they think of it and how they think it could be improved.
8. The game should (virtually) instantly tell the user if the sequence of letters that they have tried is actually a word. Words must be at least 3 letters long.
9. It should tell them how many points they have scored, but this can be done at the end of the round.
10. There should be a three minute (180 seconds) time limit on each round, so that they only have a limited amount of time to find as many words as they can.
11. The screen layout must display the grid of letters, the current score, the time remaining, information about their last guess (was it correct? If not, why not?), the language that they have chosen to play in, and the number of words remaining to be found. This is to make the interface useful, but not cluttered.
12. The user should be congratulated, in the rare case that they are able to find all of the possible words before the timer runs out.
13. The game needs to display the longest words that the user could have played, at the end of the game, and this needs to happen quickly, taking no more than a few seconds.
14. The teacher should be able to look at the list of scores and names, so that they can see how well their students have done.

15. There could be the possibility of a multiplayer version of the game. However, I may encounter problems with school firewalls, which don't allow two computers to directly communicate with each other. A multiplayer version could work in two ways, either with the two users playing together, as a team, or individually against each other. In the team version, the users would be able to guess together, using the same board, and all the words that they find are combined together and they receive a joint score at the end of the game for all of their words. In the individual, head-to-head version, users would be playing with the same board, and it would be a competition to see who could get the most points within the time limit.

Development



Design

There are two main difficult algorithms in this game, the Trie build/lookup functions and the function to find all of the possible words in a grid. As both of these depend on finding a word quickly, I wanted to make sure that my Trie structure was working effectively. My first iteration will aim to build the trie and test it to find words quickly.

In order to build the trie, I must first define the object class of the nodes which will make up the trie, one of these objects is called a 'TrieNode'.

To create a node:

I have to define a class, called 'TrieNode', this is because I am going to have to create lots and lots of these objects, and they will all have the same properties, the only thing that will differ between them is the value of some of these properties.

Each node must have the property of a parent pointer, which will point to the letter from which it stemmed in the tree. This will be used to indicate that a new node stems from a particular node. Each node must also have some 'children' variables, which will be used to indicate which letters stem from that node. This variable should initially be set as a list of 26 blank items. The reason for this is to make the data structure smaller, and less time consuming to create, since none of the nodes will actually have values for all 26 child pointers, as there are no letters which can have any letter come after them.

Each node will also have a variable called 'value', which will be a single letter string, and is the actual character that this particular node represents in the trie structure. This will be set when the node is created.

The final variable that a node will need to have is a boolean value called 'isWord', which will indicate whether or not this letter marks the end of a word, or if it is only the start or middle of a word. This will initially be set to False as a default, and will only be set to True during the creation of the trie.

The pseudocode and the actual python code for everything I have just talked about is shown below.

In pseudocode:

Class TrieNode

```
public parent  pointer
public children list of 26 pointers
public value   character
public isWord  boolean

public method  init (character, parent)
    children = None * 26
    value = character
    isWord = False
    parent = parent
```


In python:

```
class TrieNode:
    def __init__(self, parent, value):
        self.parent = parent
        self.children = [None] * 26
        #Creates a list containing 26 items, all of which are None,
        meaning that they are blank.
        self.isWord = False
        #By default, a node is not marked as the end of a word when it
        is created.
        self.value = value
        #The value of the node is passed to the node when it is created.
        if parent is not None:
            parent.children[ord(value)-97] = self
        #This creates the node as a child of its parent by making it one
        of the elements in the parent's 'children' list, thereby linking
        the two nodes together.
```

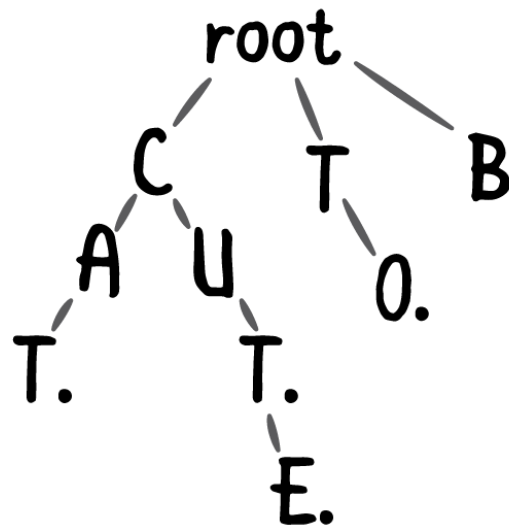
To create the trie:

1. Open the dictionary. (Taken from here: <http://www.gwicks.net/dictionaries.htm>)
2. Create a blank root node with no parent and a blank value.
3. Run a 'for' loop that goes through each word in the dictionary.
4. Start at the root node.
5. Look at the first letter of the current word.
6. Find the child node which represents this letter and make this the next node.
7. If the next node is blank, create a node in its place with its parent being the current node and its value being the current letter.
8. The next node becomes the current node.
9. Once we reach the end of a word, we mark the current node as a complete word.

An example of a small trie being created is explained below:

Let us say we were going to create a trie, using a dictionary which contained the words "Cat", "Cute", and "To". We would firstly create a blank root node, with no parent and a value of none. We then look at each word from the dictionary in turn and, starting at the root node, we look at the first letter of the first word, it is "c", so we find where the the child node which represents this letter should be and we make it our next node, we then check to see if this node is blank or if it has already been given a value. If it is blank, then we create a node in its place, with its parent being the current node, and its value being the current letter. We then make this next node our current node and move onto the next letter in the word. Once we reach the end of a word, we set the value of 'isWord' to True, to indicate that this is a full and complete word, and we move on to the next word in the dictionary.

The result of this is shown here.



The actual python code for this function is shown below:

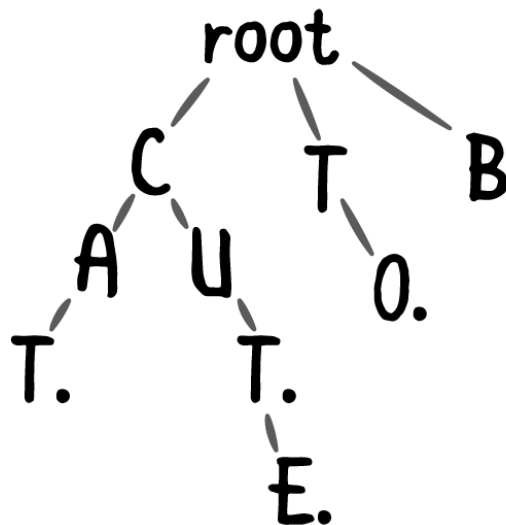
In python:

```
def makeTrie(dictionary):
#Defining a function called 'makeTrie', with one input.
    dict = open(dictionary)
    #Opens the dictionary.
    root = TrieNode(None, '')
    #Creates a blank root node.
    for word in dict:
    #Runs through every word in the dictionary.
        curNode = root
    #Starts at the root node.
        for letter in word.lower():
    #Runs through every letter in the current word.
    #Also converts the word to entirely lowercase.
            if 97 <= ord(letter) < 123:
    #Only accepts letters from a-z, whose respective
    #ASCII values are 97 and 122.
                nextNode = curNode.children[ord(letter)-97]
    #The next node to go to is the child node of the
    #current node with the corresponding letter value.
    #ord(letter)-97 gives me numerical values of any
    #letter, with 'a' being 0 and 'z' being 25.
                if nextNode is None:
    #If the next node doesn't yet exist.
                    nextNode = TrieNode(curNode, letter)
    #Create a new node, with the current node as its
    #parent, and the current letter as its value.
                curNode = nextNode
    #This new node then becomes the current node and we
    #repeat the process for the next letter in the word.
        curNode.isWord = True
    #Once we reach the end of a word, we mark the current node
    #as being the end of a complete word.
    return root
#Once we are finished, we return the root, which is now the parent of
the entire trie, and is therefore the way to access the trie.
```

To search the trie:

1. Set the current node to the root of the Trie.
2. Take the first letter in the word.
3. Find the child node representing by this letter and make that the next node.
4. Repeat this process until you either find that there is no child node there, meaning that the word does not exist, or until you reach the end of the word.
5. If you reach the end of the word, check to see if the node which you have ended up on is marked as the end of a word, if it is then the word does exist, if it is not , then the sequence of letters is the start of a word, but not a complete word.

Some examples of searching through a miniature trie shown here, are explained below:



Let us imagine we have created the trie shown here, and we want to search for the word “cat”. We would firstly set the current node to being the root of the trie, and then we would take the first letter of the word, which is ‘c’, and use it to find the child node of the current node which represented ‘c’, and we would make this the next node. We would repeat this process for ‘a’ and then for ‘t’. We would then have reached the end of the word, so we would check to see if the node that we have ended up on ‘t’, is marked at being the end of a word, and we would see that it is, meaning that this is a valid word that exists in our trie.

Now for another example, let us imagine that we are trying to search for the word ‘cu’ in the same trie (shown above). We would initially run through the same steps as we did in the previous example, finding the corresponding child nodes using the current letter that we were looking at. Firstly finding the node representing ‘c’ from the root node, and then finding the node which represented ‘u’ from the ‘c’ node. Then we would have reached the end of the word, so we would check to see whether the node on which we finished was marked as being the end of a word, and we would find that it is not, which means that, although this sequence of letters is the start of a word, it is not a full word.

Let us go through a third and final example, using the same trie (pictured above). If we were trying to find the word ‘cab’, then we would firstly find the child node of the root which represented ‘c’, and then we would find the child node of ‘c’ which represented ‘a’, and then we would try to find the child node of ‘a’ which represented ‘b’, and we would find that, in this miniature trie, it does not exist. This would mean that the word does not exist.

The actual python code for the function which performs the search is shown below:

```

def searchTrie(word, rootNode):
#Defining a function called 'searchTrie', with two inputs.
    isThere = False
    #Initially defining the 'isThere' boolean variable to be False,
    #because of the way this function works, the word is assumed to
    #not be there until proven that it is.
    curNode = rootNode
    #Make the current node the root node, which is passed to the
    #function when it is called.
    for letter in word.lower():
    #Run through every letter in the given word.
    #Also converts entire word to lowercase.
        if 97 <= ord(letter) < 123:
            #97 and 122 are the ASCII values of 'a' and 'z'
            #respectively.
            #This means that the function only looks at lower case
            #letters, if the word contains anything other than letters,
            #they will be ignored.
            nextNode = curNode.children[ord(letter)-97]
            #The next node to go to is the child node of the
            #current node with the corresponding letter value.
            if nextNode is not None:
                #If the next node does exist, because it is not none, if
                #must have a value and be an actual node.
                curNode = nextNode
                #The current node becomes this node, and we repeat
                #this process for the next letter in the word.
            else:
                #If the next node does not exist, because it is none.
                print("It's not a word")
                break
                #This means that the word does not exist in the trie,
                #so we change the 'isWord' variable to False, print
                #the appropriate output, and break from this loop.
        if curNode.isWord == True:
            #Check to see if the node we have finished on is marked as being
            #the end of a word.
            isThere = True
            print("It's there and it is a word")
        else:
            print("It's there but it's not a complete word")
    return isThere

```

Testing

Here I am going to test out the functions and class definitions which I have just written.

Firstly, the TrieNode class:

The image below is a screengrab of the code that is being run and tested here.

This code is explained in the previous design section, it should create a variable called “rootNode”, which is an object with the properties stated and explained previously, which are: “parent”, “children”, “isWord”, and “value”.

```
class TrieNode:
    def __init__(self, parent, value):
        self.parent = parent
        self.children = [None] * 26
        self.isWord = False
        self.value = value
        if parent is not None:
            parent.children[ord(value)-97] = self

rootNode = TrieNode(None, 'a')
```

If we then examine the stack data (shown here on the right), we can see what this code has actually done.

The fact that this code is working successfully is evidenced by the fact that the four aforementioned variables are shown here, and their values are correct and exactly what I expected them to be. The ‘children’ variable is a list of 26 elements, all of which are initially set to None, because this node has no children. The “isWord” boolean has been set to False, as it should be. The ‘parent’ value, which I actually passed to the class when I called it, is None, and the ‘value’ is “a”, again, as I told it to be. This shows that this __init__ function is doing everything that I want it to, it is creating a TrieNode object with four variables, correctly named and with correct values.

```
self <__main__.TrieNode 0x1db
__class__ <getset_descriptor 0x3f196f
__dict__ <getset_descriptor 0x1e404
__doc__ None
__doc__ "The most base type"
__module__ __main__
__subclass__ <classmethod_descriptor 0x
__weakref__ <getset_descriptor 0x1e405
children <list 0x1e0b0a8; len=26>
isWord False
parent None
value "a"
value "a"
```

The image below on the left is a screengrab of the code that is being executed here, and the image on the right is a screengrab of the stack data that is held after this function has been run. The stack data shows that I have a root node, which has various properties, including child nodes, which again, have the same properties, and have children as well, I have only been able to capture 5 layers of nodes in the Trie here, as it would not be practical to attempt to screengrab the data from the entire Trie structure, but it nevertheless demonstrates that the 'makeTrie' function works, and does indeed create a Trie structure from a dictionary.

[illegible]

The code that I am running here is shown below:


```
def makeTrie(dictionary):
    #dict = open(dictionary)
    root = TrieNode(None, '')
    for word in dictionary:
        curNode = root
        for letter in word.lower():
            if 97 <= ord(letter) < 123:
                nextNode = curNode.children[ord(letter)-97]
                if nextNode is None:
                    nextNode = TrieNode(curNode, letter)
                curNode = nextNode
        curNode.isWord = True
    return root

root = makeTrie(["cab"])
```

The stack data that I then looked at is shown here:

This is a screengrab of showing all of the local variables which are being held in the stack at the point when the program is paused. There is the variable 'curNode' which is one of the variables that I create in the

process of creating the trie, this will be the last node that I created in this process, the node representing the letter "b". This is shown to be true by the screengrab below, which shows all of the properties of the 'curNode' variable.

locals	<dict 0x1eccf80; len=6>
curNode	<__main__.TrieNode 0x1e9b830; len=1>
dictionary	<list 0x1e50878; len=1>
letter	"b"
nextNode	<__main__.TrieNode 0x1e9b830; len=1>
root	<__main__.TrieNode 0x1e9b550; len=1>
word	"cab"

locals	<dict 0x1eccf80; len=6>
curNode	<__main__.TrieNode 0x1e9b830; len=1>
__class__	<getset_descriptor 0x1251968; len=0>
__dict__	<getset_descriptor 0x1ed0558; len=0>
__doc__	None
__doc__ <0x12503c0>	"The most base type"
__module__	"__main__"
__subclasshook__	<classmethod_descriptor 0x1251878; len=0>
__weakref__	<getset_descriptor 0x1ed0580; len=0>
children	<list 0x1ed00a8; len=26>
isWord	True
parent	<__main__.TrieNode 0x1e9b790; len=1>
value	"b"
dictionary	<list 0x1e50878; len=1>

The main thing to highlight here is that the 'value; variable is "b", meaning that the node represents the letter "b".

The next variable which is being held in the stack is the 'dictionary' variable, which should be a list containing the word "cab", as that is what I sent the function when I called it. The screengrab below shows what exactly is held under this variable.

```
└─ curNode      <__main__.TrieNode 0x1e9b830; len=1>
└─ dictionary   <list 0x1e50878; len=1>
    0           "cab"
    letter      "b"
```

As you can see, the variable is a list with length 1, meaning it contains a single item, and that item is shown below, alongside its index in the list, it is the string "cab".

The next variable, 'letter', is used as the variable in the 'for' loop in the function, so when the function is finished, it should be equal to whatever the last letter that the function dealt with was, which in this case should be "b", because that is the last letter in the word "cab".

```
└─ curNode      <__main__.TrieNode 0x1e9b830; len=1>
└─ dictionary   <list 0x1e50878; len=1>
    letter      "b"
└─ nextNode     <__main__.TrieNode 0x1e9b830; len=1>
└─ root         <__main__.TrieNode 0x1e9b550; len=1>
    word        "cab"
```

As you can see, the stack data shows that the 'letter' variable has the value "b", which makes sense.

The next variable in the stack is 'nextNode', which should be the same as 'curNode', because the function sets 'curNode' as equal to 'nextNode' after it creates each new node, so you would expect, at the end of the function, the two to be the same.

```

> curNode          <__main__.TrieNode 0x1e9b830; len=1>
> dictionary       <list 0x1e50878; len=1>
  letter          "b"
  ▲ nextNode       <__main__.TrieNode 0x1e9b830; len=1>
    __class__      <getset_descriptor 0x1251968; len=0>
    __dict__       <getset_descriptor 0x1ed0558; len=0>
    __doc__        None
    __doc__ <0x12503c0> "The most base type"
    __module__     "__main__"
    __subclasshook__ <classmethod_descriptor 0x1251878; len=0>
    __weakref__    <getset_descriptor 0x1ed0580; len=0>
  > children       <list 0x1ed00a8; len=26>
    isWord         True
  > parent         <__main__.TrieNode 0x1e9b790; len=1>
    value          "b"
> root            <__main__.TrieNode 0x1e9b550; len=1>
  word            "cab"

```

This is exactly what the stack shows, if you look at the memory locations of 'curNode', and of 'nextNode', they are the same; "0x1e9b830", and if you expand 'nextNode' to have a look at its properties, you find that they are all exactly the same. This shows that the two variables are identical, meaning that the function is performing as expected.

If we look at the next variable in the stack data, it is the 'root' variable. This is the variable which actually contains the trie itself, so if we look into its properties, we should be able to find the trie.

▲ locals	<dict 0x1eccf80; len=6>
▷ curNode	<__main__.TrieNode 0x1e9b830; len=1>
▷ dictionary	<list 0x1e50878; len=1>
letter	"b"
▶ nextNode	<__main__.TrieNode 0x1e9b830; len=1>
▲ root	<__main__.TrieNode 0x1e9b550; len=1>
__class__	<getset_descriptor 0x1251968; len=0>
__dict__	<getset_descriptor 0x1ed0558; len=0>
__doc__	None
__doc__ <0x12503c0>	"The most base type"
__module__	"__main__"
__subclasshook__	<classmethod_descriptor 0x1251878; len=0>
__weakref__	<getset_descriptor 0x1ed0580; len=0>
▷ children	<list 0x1e50800; len=26>
isWord	False
parent	None
value	""
word	"cab"

The above screengrab shows all of the properties of 'root', including all of the variables which are held under it. The most important thing to point out in testing if the trie has been properly constructed is the 'children' variable, which, as shown in the stack data, is a list of length 26. These 26 items in the list are the 26 possible letters which could stem from the root node, so they represent all of the first letters of the words in our dictionary.

Since the only word in our “dictionary” is “cab”, we would expect the only child of the root node to be the letter “c”, as this is the only first letter of any word in our input. Therefore we would expect the third item, and only the first item, in the ‘children’ list to contain something, since “c” is the third letter in the alphabet.

The children of the root node are shown in the stack data shown below.

children	<list 0x1e50800; len=26>
0	None
1	None
2	<__main__.TrieNode 0x1e9b6d0; len=1>
3	None
4	None
5	None
6	None
7	None
8	None
9	None
10	None
11	None
12	None
13	None
14	None
15	None
16	None
17	None
18	None
19	None
20	None
21	None
22	None
23	None
24	None
25	None
isWord	False
parent	None
value	""
word	"cab"

As you can see, all of the items in the list are ‘None’, apart from the third item, which contains, as shown, an instance of a ‘TrieNode’ object.

If we then expand this item, to look at its properties, we find that it does indeed contain all of the properties of a 'TrieNode'. It is also important to point out that this node has the 'value' of "c", which is exactly as it should be, meaning that, so far, the function has successfully created a trie.

children	<list 0x1e50800; len=26>
0	None
1	None
2	<__main__.TrieNode 0x1e9b6d0; len=1>
__class__	<getset_descriptor 0x1251968; len=0>
__dict__	<getset_descriptor 0x1ed0558; len=0>
__doc__	None
__doc__	<0x1250"The most base type">
__module__	"__main__"
__subclasshook__	<classmethod_descriptor 0x1251878; len=0>
__weakref__	<getset_descriptor 0x1ed0580; len=0>
children	<list 0x1e50828; len=26>
isWord	False
parent	<__main__.TrieNode 0x1e9b550; len=1>
value	"c"
3	None

Next, we must look at the children of this node.

```

└─ children      <list 0x1e50800; len=26>
  0              None
  1              None
  └─ 2           <__main__.TrieNode 0x1e9b6d0; len=1>
    _class_      <getset_descriptor 0x1251968; len=0>
    _dict_       <getset_descriptor 0x1ed0558; len=0>
    _doc_        None
    _doc_        <0x1250"The most base type"
    _module_     "__main__"
    _subclasshook_ <classmethod_descriptor 0x1251878; len=0>
    _weakref_     <getset_descriptor 0x1ed0580; len=0>
    └─ children  <list 0x1e50828; len=26>
      0          <__main__.TrieNode 0x1e9b790; len=1>
      1          None
      2          None
      3          None
      4          None
      5          None
      6          None
      7          None
      8          None
      9          None
      10         None
      11         None
      12         None
      13         None
      14         None
      15         None
      16         None
      17         None
      18         None
      19         None
      20         None
      21         None
      22         None
      23         None

```

As you can see, it only has one child node which is not 'None', and this is the first item in the 'children' list. This makes sense because the next letter after "c" in the word "cab" is the letter "a", which is the first letter in the alphabet.

If we expand the properties of the variable contained in this list, we see the data shown below.


```

└─ children <list 0x1e50828; len=26>
  └─ 0 <__main__.TrieNode 0x1e9b790; len=1>
    _class_ <getset_descriptor 0x1251968; len=0>
    _dict_ <getset_descriptor 0x1ed0558; len=0>
    _doc_ None
    _doc_ "The most base type"
    _modul_ "__main__"
    _subcla<classmethod_descriptor 0x1251878; len=0>
    _weakre<getset_descriptor 0x1ed0580; len=0>
    ▸ children <list 0x1e50a30; len=26>
    isWord False
    ▸ parent <__main__.TrieNode 0x1e9b6d0; len=1>
    value "a"
1      None
2      None

```

This shows that the item is an instance of a 'TrieNode', and it contains all of the relevant and appropriate variables, including the 'value' variable, which is set to "a", as it should be.

Next, when we expand the list of children of this node (shown below), we find another list of 26 items, 25 of which are "None", and one of which, the second item in the list, is an instance of a 'TrieNode' object. This makes sense, as the current node we are looking at is the letter "a", and in the word "cab", the letter "b" comes directly after, which is the second letter in the alphabet.

```

└─ children <list 0x1e50a30; len=26>
  0 None
  ▸ 1 <__main__.TrieNode 0x1e9b830; len=1>
  2 None
  3 None
  4 None
  5 None
  6 None
  7 None
  8 None
  9 None
  10 None
  11 None
  12 None
  13 None
  14 None
  15 None
  16 None
  17 None
  18 None
  19 None
  20 None
  21 None
  22 None
  23 None
  24 None
  25 None
  isWord False

```


When we then expand this item from the list, we find that it does indeed have all of the properties of a TrieNode, notably its “value” is set to “b”, which is exactly what it should be, as this node is supposed to represent the letter “b”, the last letter of the word “cab”.

```

└─ children <list 0x1e50a30; len=26>
  0 None
  1 <__main__.TrieNode 0x1e9b830; len=1>
    __class__ <getset_descriptor 0x1251968; len=0>
    __dict__ <getset_descriptor 0x1ed0558; len=0>
    __doc__ None
    __doc__ "The most base type"
    __module__ "__main__"
    __subclass__ <classmethod_descriptor 0x1251878; len=0>
    __weakref__ <getset_descriptor 0x1ed0580; len=0>
    ▸ children <list 0x1ed00a8; len=26>
    isWord True
    ▸ parent <__main__.TrieNode 0x1e9b790; len=1>
    value "b"
  2 None
  3 None
  4 None

```

An important thing to point out here is that the “isWord” parameter is set to ‘True’ for this node, which makes sense, as this node represents the end of the word “cab”, and the node can only be reached by traversing the tree from the root node, to “c”, then to “a”, and finally to “b”, forming the word “cab”.

Just to make it clear that, since the trie should have been created with only one word in it, there are no other words in this trie, I have also shown a screenshot (right) of the list of the children of the final node I have talked about. This shows that all of the elements in its ‘children’ list are “None”, meaning that there are no further letters, and thereby words, which stem from here. This is the end of the trie.

```

└─ children <list 0x1ed00a8; len=26>
  0 None
  1 None
  2 None
  3 None
  4 None
  5 None
  6 None
  7 None
  8 None
  9 None
  10 None
  11 None
  12 None
  13 None
  14 None
  15 None
  16 None
  17 None
  18 None
  19 None
  20 None
  21 None
  22 None
  23 None
  24 None
  25 None
  isWord True
  ▸ parent <__main__.TrieNode 0x1e9b790; len=1>
  value "b"
  2 None
  3 None

```

I am going to test out how long this trie data structure takes to build, to see if it complies with my success criteria.

Below is a screengrab of the exact code which I ran to find this out:

```
import time

class TrieNode:
    def __init__(self, parent, value):
        self.parent = parent
        self.children = [None] * 26
        self.hasChildren = False
        self.isWord = False
        self.value = value
        if parent is not None:
            parent.children[ord(value)-97] = self

def makeTrie(dictionary):
    dict = open(dictionary)
    root = TrieNode(None, '')
    for word in dict:
        curNode = root
        for letter in word.lower():
            if 97 <= ord(letter) < 123:
                nextNode = curNode.children[ord(letter)-97]
                if nextNode is None:
                    nextNode = TrieNode(curNode, letter)
                    curNode.hasChildren = True
                curNode = nextNode
        curNode.isWord = True
    return root

start = time.time()
root = makeTrie("french.txt")
#334,871 words
end = time.time()
print(end-start)
```

As you can see, this includes the class definition of the 'TrieNode', and the function 'makeTrie' which are the two things that I need to create the trie. At the bottom here I am starting the timer, calling the function, ending the timer immediately, and then printing the time taken.

I have also commented in the number of words which are in the dictionary file, from which I am creating the trie. It is worth mentioning that this is a huge number of words, and many other dictionaries will be smaller than this, therefore this is a 'worst case scenario' test.

(This virtual machine is using one processor and 480MB of memory)

The results of these tests are shown here:

I repeated the test five times consecutively, to give a fair and accurate idea of how long this data structure actually takes to build.

14.071224927902222

14.476825952529907

14.289625883102417

14.258424997329712

14.008825063705444

It is clear to see that the time taken to build the trie is pretty consistently just over 14 seconds. This fits with my original criteria, which was for the game to load in under 15 seconds.

Therefore, this testing has successfully proved that the making of the trie, even under worst case scenario conditions, complies with my original success criteria.

Next, I am going to test the function which will search through the trie to find out whether or not a particular combination of letters is actually a word:

The image below on the left is a picture of the code from the function that is being run here. The images of the right show the tests that I have run on the function to see if it is working properly. The first one is an actual French word, so the output is 'True', because it is a full and complete word in the French dictionary.

The second test is part of a French word, but not a full word, so the output is 'False', as it should be.

The third is a French word, but with an extra letter added onto the end, so the output is 'False', because the string is not a word.

The last test is a blank input, where the word being tested is an empty string, the function correctly outputs 'False', because this is not a word in the dictionary.

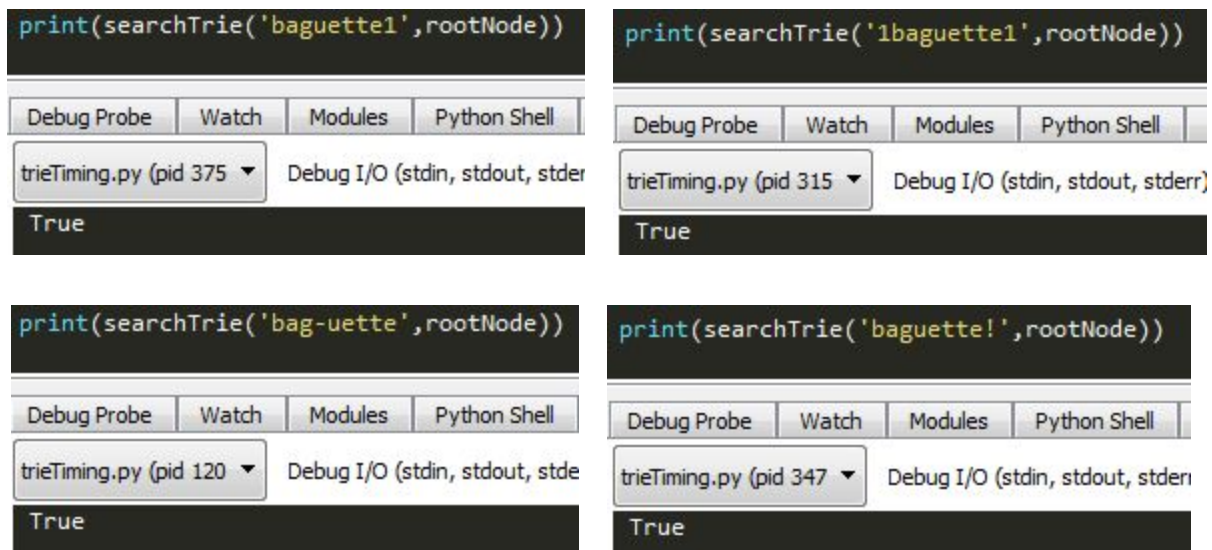
```
def searchTrie(word, rootNode):
    isThere = False
    curNode = rootNode
    for letter in word.lower():
        if 97 <= ord(letter) < 123:
            nextNode = curNode.children[ord(letter)-97]
            if nextNode is not None:
                curNode = nextNode
            else:
                return isThere
    if curNode.isWord == True:
        isThere = True
    return isThere
```



I should point out that, due to the way that this function works, in particular the 'if' statement in the fourth line of the function, it will ignore any characters which are not regular letters. This means that if I were to test a word such as "baguette1", to see if it was in the trie, the function would tell me that it was, which would clearly be wrong, as there is no such word in French.

However, this does not matter, as when the game is actually being played, this will never become an issue, since I will only be passing the function strings which exclusively contain letters. Meaning that, in effect, this error handling is done with the string before the function is used to search for it in the trie.

The following tests are examples of where this function gives the wrong output:



In fact, any and all characters which are not letters in the alphabet are simply ignored by this function. The reason why this happens is that, if I were to try to search for them, I would get an "IndexError", and receive the following message:

```
File "z:\H\My Documents\Year 13\Computer
Science\Project Coding\trieTiming.py", line 42, in
<module>
    print(searchTrie('baguette!',rootNode))
File "z:\H\My Documents\Year 13\Computer
Science\Project Coding\trieTiming.py", line 33, in
searchTrie
    curNode = curNode.children[ord(letter)-97]
builtins.IndexError: list index out of range
```

But as I have mentioned above, all of this is actually irrelevant when it comes to the actual game, as the inputs are all checked prior to being passed to the function, meaning that no input will contain anything other than regular letters.

Design

The next function that I wrote was a function which takes the frequency of all of the letters in the given language, which I obtained from Wikipedia (https://en.wikipedia.org/wiki/Letter_frequency), and generates an array which includes a certain number of each letter, which is proportional to its frequency of use in the language.

The reason why I have chosen to make the frequency of each letter on the board proportional to its frequency in the chosen language is to make it easier to find words in the grid. If the letters on the board are similar to the letters which are used in common words in the language, it will be easier to find common words. However, if the letters were entirely random, there would be the same number of a very uncommon letters appearing as common letters, making most common words very difficult to find.

The function takes as its input a dictionary data type which lists each letter of the alphabet, and its relative frequency in that language. It is worth noting that I have not ignored the special characters which are used in particular languages, such as letters with accents on them, I have taken the frequency of any such letters and added it to the frequency of its 'regular' letter counterpart. For example, I have added the frequency of the letter ' é ' to the frequency of the letter ' e ', I have done this because, when the user is trying to find words, they will be able to use the unaccented version of a letter in place of an accented letter in a word. For instance, if there is a chain of letters in a grid which spells out 'ECOLE', then the user would be able to find the word ' école ' using that chain.

The reason why I have made the game this way is that the frequency of individual special characters is so low, that if they were to appear in the grid, it would be incredibly hard to find any words which they were a part of, and therefore there would be hardly any words which involved them, making them effectively useless letters.


```
frenchFrequency = {'A':81, 'B':9, 'C':34, 'D':37, 'E':167, 'F':11, 'G':9,
'H':7, 'I':76, 'J':6, 'K':1, 'L':55, 'M':30, 'N':71, 'O':58, 'P':25, 'Q':14,
'R':67, 'S':79, 'T':72, 'U':64, 'V':18, 'W':1, 'X':4, 'Y':1, 'Z':3}
```

The 'letterFrequency' function will run through the alphabet (A to Z), and look at the corresponding dictionary value, which will be a number that represents the frequency of that letter, it will then add the current letter to the 'letters' list that many times.

The 'dictionary' input will be a data structure such as the one shown above in blue here. The reason why I have generalised the input and the function itself, so that it accepts any dictionary, is so that I can use this same function for a variety of different languages, the way this function is written, it can accept a dictionary which represents the frequency of all the letters in any language, and output an array which can then be used to generate a grid of letters.

```
def letterFrequency(dictionary):
    alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    letters = []
    for x in range(len(alphabet)):
        for y in range(dictionary[alphabet[x]]):
            letters.append(alphabet[x])
    return letters
```

This next function is a very short and simple function which creates a blank 4x4 2D array, which is going to be filled with letters. As you can see from the screengrab of the function below, there are two 'for' loops, one of which runs inside the other, so that it creates an array with 4 elements, inside each of the four elements of the 'grid'.

```
def createGrid():
    grid = []
    for row in range(4):
        grid.append([])
        for column in range(4):
            grid[row].append(" ")
    return grid
```

The output of this function should look like this:

```
grid = [ [ " ", " ", " ", " " ], [ " ", " ", " ", " " ], [ " ", " ", " ", " " ], [ " ", " ", " ", " " ],
[ " ", " ", " ", " " ], [ " ", " ", " ", " " ], [ " ", " ", " ", " " ], [ " ", " ", " ", " " ] ]
```

The next function, which must take the empty grid (shown above), and fill it up with random letters, proportional to the letter's frequency in the language.

It does this by using the array, which I talked about earlier, in which letters appear entirely proportional to their commonality in the specified language. It cycles through the 2D array, takes a random letter from the frequency array, and places it in that slot.

The code for this function is shown below:

```
def fillGrid(grid):  
    for x in range(len(grid)):  
        for y in range(len(grid[x])):  
            grid[x][y] = frenchLetters[random.randint(0,999)]  
    return grid
```

randomly selected letter into each place.

When I was designing this, I considered a number of different ways in which I could have it work. One way could have been to delete each letter from the frequency array once I picked it out, the reason why I didn't do this is that it would impact on the probability that that letter would be picked out again, which would actually skew the outcome of the function, as the probability of each letter being picked would change as the function ran, meaning that the frequency of each letter in the grid would end up being disproportionate to its commonality in the language. This method would also make the function run slower, as it would have to perform a delete after it selects each letter.

The method I selected in the end, which I believe is the best suited for this task, is to use Python's built in random number generator to select a random letter from the array, and not delete or change the array in any way whilst I am working with it, to maintain the correct probability of each letter being picked.

When I created the function that would find all of the possible words that a user could play, I had two choices in the method that I could use to search through the grid, I could use either a breadth first or a depth first search.

A depth first search would be best suited for this problem if the number of possible starting positions was very large, but since the grid is only 4x4, there are only 16 possible starting positions, meaning that a depth first search would not be ideal. Part of the reason why a depth first search would lead to a very complex function is that once I reached a dead-end in the search, where there are no more words which stem from my current position in the grid, I would have to trace back my steps, checking all of the possibilities at each step back, and exploring them if I needed to, until eventually I exhausted all of my options down that route. At the point I would have to repeat that process for all of the other neighbours of the starting point, and then for all of the different starting points.

Overall though, the common sense reason as to why I decided to use a breadth first search is that it is simply a more intuitive algorithm to use in this scenario. It is also worth pointing out that it makes more sense to implement a queue than it does to implement a stack (for this particular task), as I would have had to do if I used a depth-first algorithm.

To make the main function which actually performs the search more concise, I wrote four other 'helper' functions which are to be called throughout the main function. I am going to explain what these smaller functions do before moving on to the major function. I will go through these functions in the order in which they are first called.

Firstly, we have the translate function, this takes a list of coordinates, and translates them into a word, or at least of sequence of letters. It does this by simply going through the list of coordinates sequentially, looking up the corresponding letter from the grid, which is given by the x and y value, and appending that letter to the word variable, so that it returns a string of letters.

```
def translate(list):  
    word = ''  
    for x in range(len(list)):  
        word += fullGrid[list[x][0]][list[x][1]]  
    return word
```

An example of how this function would work is shown below:

Let us imagine that I have generated the following grid:

	0	1	2	3
0	D	G	H	I
1	K	L	P	S
2	Y	E	U	T
3	E	O	R	N

Which would be represented by the following 2D array:

```
grid = [ [{"D"}, {"K"}, {"Y"}, {"E"}] , [{"G"}, {"L"}, {"E"}, {"O"}] ,  
        [{"H"}, {"P"}, {"U"}, {"R"}] , [{"I"}, {"S"}, {"T"}, {"N"}] ]
```

If I were to then use the 'translate' function to translate a chain of grid coordinates into a string of letters, the function would look through each coordinate from the list in turn, find the corresponding letter from the 2D array and add that to the output string.

If we wanted to translate the following list of coordinates:

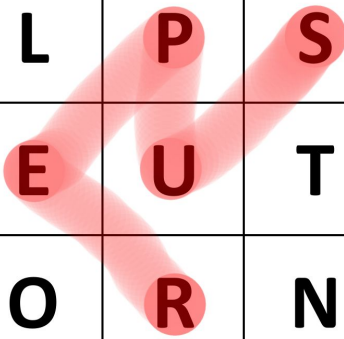
```
list = [[3,1],[2,2],[2,1],[1,2],[2,3]]
```

The function would first look at the first element in this list, and then look up element [3][1] in the grid array, which is the second item in the fourth list inside the list, which, as you can see from above, is the letter "S". The 'translate' function would then add this letter to the output string, and then repeat this process for each coordinate in the 'list'.

At the end of this process, the chain of coordinates will have been successfully translated into a string.

With the above example, the output string would be “SUPER”, which is shown by the diagram below:

	0	1	2	3
0	D	G	H	I
1	K	L	P	S
2	Y	E	U	T
3	E	O	R	N



It is clear to see how this makes sense practically, with the coordinate list that I have shown above.

Secondly, there is the 'searchStub' function, which is the same as the searchTrie function which I have already described, apart from the fact that this function does not take into account whether the input is a complete word or not, it returns True if the stub is the beginning of a word, and False if it is not. In other words, if there are no words which begin with that stub then it will output False, but if there is at least one, the output is True.

```
def searchStub(stub, rootNode):
    isThere = True
    curNode = rootNode
    for letter in stub.lower():
        if 97 <= ord(letter) < 123:
            nextNode = curNode.children[ord(letter)-97]
            if nextNode is not None:
                curNode = nextNode
            else:
                return False
    return isThere
```

Next, we have the 'validNeighbours' function, which will take two numerical value inputs; x and y. These together represent the coordinates of a position in the grid. The function will output a list of the coordinates of all of the valid neighbours of that position. This function simply works by running through these 'if' statements (shown to the right) and either fitting the criteria for them or not, depending on what the given x and y values are, and adding the relevant coordinates to the 'neighbours' list.

```
def validNeighbours(x,y):
    neighbours = []
    if x < 3:
        neighbours.append([x+1,y])
        if y < 3:
            neighbours.append([x+1,y+1])
        if y > 0:
            neighbours.append([x+1,y-1])
    if x > 0:
        neighbours.append([x-1,y])
        if y < 3:
            neighbours.append([x-1,y+1])
        if y > 0:
            neighbours.append([x-1,y-1])
    if y < 3:
        neighbours.append([x,y+1])
    if y > 0:
        neighbours.append([x,y-1])
    return(neighbours)
```

Using this function, here are two examples of an input and the correct output:

```
print(validNeighbours(2,2))
```

```
===== RESTART: /Users/thomasbutterfield/python/Project/neighboursTest
[[3, 2], [3, 3], [3, 1], [1, 2], [1, 3], [1, 1], [2, 3], [2, 1]]
>>>
```

The above example shows the input of position [2,2], and the output of all 8 of its valid neighbours.

However, if I enter [0,0] as the input.

```
print(validNeighbours(0,0))
```

The output is only three coordinates, as there are only 3 valid neighbours of this position.

```
===== RESTART: /Users/thomasbutterfield/python/Project/neighboursTest
[[1, 0], [1, 1], [0, 1]]
>>>
```

	0	1	2	3
0	D	G	H	I
1	K	L	P	S
2	Y	E	U	T
3	E	O	R	N

Below is a screengrab of the main function that I wrote which will return a list of all of the possible words that can be played using the given grid. This function calls upon the functions which I have previously mentioned and explained.

```
def possibleWords(fullGrid):
    possible = []
    queue = []
    for x in range(len(grid)):
        for y in range(len(grid[x])):
            queue.append([x,y])
    while len(queue) > 0:
        current = queue[0]
        stub = translate(current)
        if searchStub(stub, rootNode):
            neighbours = validNeighbours(current[-1][0], current[-1][1])
            for a in range(len(neighbours)):
                add = []
                if neighbours[a] not in current:
                    for b in range(len(current)):
                        add.append(current[b])
                        add.append(neighbours[a])
                        if searchStub(translate(add), rootNode):
                            queue.append(add)
            if searchTrie(stub, rootNode) and stub not in possible:
                possible.append(stub)
        del queue[0]
    return possible
```

The function begins by creating two empty lists: 'possible' and 'queue'. The former will be filled with the possible words that a user could play and will then be returned. The latter will be filled with other lists, which represent coordinate paths that are valid by the rules of the game, meaning that they do not use the same coordinate position twice.

Next, the function fills up the 'queue' list with the 16 coordinates of the grid, these coordinates are a list themselves, because they have an x and y value, but they are appended to the queue within another list, because they will become the starting point of paths later on in the function, and so to do this, they have to be a list inside a list.

The 16 coordinates represent the 16 letters in the grid, as these are all the possible first letters of a word.

Next is the major loop of the function, a while loop that runs for as long as the queue list is not empty. Firstly, we name some variables to make it easier to reference particular things later on in the function. We start by naming 'current' as being the first item in the queue, and we also name 'stub' as being the word or partial word that is represented by the coordinate path in 'current', this is done by the function 'translate'.

After this, we first of all check if the current stub is actually the start of a word in the chosen language, because if it is not, then we can simply discard it in our breadth first search, as there are no possible words that could stem from it and the function will skip to the "del queue[0]" line

and then iterate through the 'queue' list. However, if the 'searchStub' function returns True then we have the start of a valid word and the next stage of the function will happen.

We define a variable called 'neighbours' as the output of the 'validNeighbours' function, which will be an array of the coordinates of all of the adjacent grid squares to the last item in the current coordinate path list.

After this, we run a 'for' loop which cycles through all the elements in this array, taking the current coordinate list, appending each neighbour in turn, and then checking if this new list represents a valid stub of a word in the chosen language.

If it does, we add this coordinate list to the queue.

Before it adds on each neighbour, it checks if that neighbour coordinate has been previously used in the current coordinate path, and if it has, will just move on the next neighbour, this is because in the game of 'boggle', you cannot use the same tile more than once when making a word.

After this, the function then moves on and checks if the current stub is a full valid word in the language, using the previously mentioned 'searchTrie' function, which will only return True if the input is a complete word, not just the start of a word. It also checks if it has already found the word by using the 'in' function in the python library to perform a linear search for the word in the current 'possible' list. If the stub is a full word, and it has not yet been found, then the function will append the stub to the possible words list.

It then deletes the first item in the queue and runs through the next iteration of the while loop.

Below, is an actual example of how this function would work, including diagrams, to make it clearer and easier to understand.

Let us imagine that the game has generated the following grid of letters, and we want to find all of the possible words that can be made using it, using the trie which we have created from the dictionary we have been given.

	0	1	2	3
0	D	G	H	I
1	K	L	P	S
2	Y	E	U	T
3	E	O	R	N

For this example, to make it simpler, we will imagine that we are playing the game in English, and the dictionary that we are using only contains the words: “**dog**” and “**super**”.

I am going to run through the process of how the ‘possibleWords’ function would analyse this grid, using this previously stated limited dictionary.

Once the function has created the 'possible' and 'queue' lists, it fills 'queue' with the coordinates of all 16 of the grid positions, so that:

```
queue = [[0,0],[0,1],[0,2],[0,3],[1,0],[1,1],[1,2],  
[1,3],[2,0],[2,1],[2,2],[2,3],[3,0],[3,1],[3,2],  
[3,3]]
```

After this has been done, the function will start to actually look for words in the grid.

It will take the first coordinate sequence in 'queue' and use the 'translate' function to convert it to the string of letters that it represents in the grid, in this case that would simply be the letter "d", since there is currently only one coordinate, and therefore only one letter in the sequence, and the coordinate "[0,0]" represents the letter "d" in the grid shown above.

Once it has done this, the function will then use the 'searchStub' function to check, using the trie which would have been generated from the given dictionary, if there are any words which stem from this string. Since the word "dog" is in our dictionary, this function would return True, meaning that there is at least one word which stems from "d".

Next, since there are possible words which could be formed using the string "d", the function calls the 'validNeighbours' function to find all of the coordinates which are adjacent to the last coordinate in the current coordinate list that it is looking at, which in this case is [0,0]. This function would return the list:

```
neighbours = [1,0],[1,1],[0,1]
```

The function then iterates through each of these neighbours in turn, and with each one it firstly checks if the coordinate has been previously used in this coordinate path, and if it hasn't it will create a variable called 'add', which contains the current coordinate path, plus the neighbour coordinate appended onto the end, so that, in the first iteration of this part of the function, it looks like this:

```
add = [[0,0],[1,0]]
```

After this, the function will then use both the 'translate' and the 'searchStub' functions to firstly translate this new coordinate path to the string "dg", using the above board, and then to check, using the trie created from the given dictionary, to check if there are any words which stem from this string. If there were any, then this coordinate path would be appended to the end of the 'queue' list before moving on to try the next neighbour, but since there are no words which start "dg" in my dictionary, the function will move straight on to trying the next neighbour from the list. The function will find that there are no words which could stem from any words using any of the neighbours of this coordinate [0,0], and it will also find that it is not a full word, so it will delete this item from the 'queue'.

The function will then take the first item in the 'queue', which is now `[[0,1]]`, since `[[0,0]]` has just been deleted. It will run through the same process as with the last coordinate list, but this time there aren't even any words which stem from the single letter "k" in my limited dictionary, so this item will immediately be discarded and the function moves on.

	0	1	2	3
0	D	G	H	I
1	K	L	P	S
2	Y	E	U	T
3	E	O	R	N

This exact same process will happen for `[[0,2]]`, `[[0,3]]`, `[[1,0]]`, `[[1,1]]`, `[[1,2]]`, `[[1,3]]`, `[[2,0]]`, `[[2,1]]`, `[[2,2]]`, `[[2,3]]`, and `[[3,0]]`, which translate to "y", "e", "g", "l", "e", "o", "h", "p", "u", "r", and "i" respectively, as there are no words which stem from those coordinate lists (using my limited dictionary).

However, once the item `[[3,1]]` reaches the front of the 'queue', when the function looks at this coordinate path, translates it to the string "s", and searches through the trie to find if there are any words which stem from it, it will find that there is at least one. The function will then find out what all of the neighbours of the last coordinate in this current pathway is, finding that:

```
neighbours = [[2,1],[2,2],[2,0],[3,2],[3,0]]
```

The function will then cycle through each one of these neighbours in turn, trying out a new coordinate path with each of them added on to the end, translating it into a string, and checking to see if there are any words which stem from that string.

It will run through the pathway `[[3,1],[2,1]]`, which translates to "sp", from which no words stem.

Then it will try `[[3,1],[2,2]]`, which translates to "su", when it looks this up in the trie, it will find that there is at least one word which stems from this string, so the function will append this coordinate pathway onto the end of the 'queue' and move on to the next neighbour. At this point in time, our queue will look like this:

```
queue = [[[3,1]],[[3,2]],[[3,3]],[[3,1],[2,2]]]
```

The changes which have taken place to give this queue include the first 13 coordinate pathways being deleted from the front of the queue since they did not lead to any possible words in our dictionary, and the last coordinate pathway, which we have just found, being appended to the end of our queue, as it has the potential to lead to a word.

The function will then continue to look at the other neighbours: [2,0] , [3,2] , and [3,0], which, when appended to the end of [[3,1]], will result in the coordinate pathways: [[3,1] , [2,0]] , [[3,1] , [3,2]] , and [[3,1] , [3,0]] , which translate to the strings “sh”, “st” and “si”, all of which, using my limited dictionary, do not lead to any words. Therefore, these neighbours would be discarded.

Once we have run through all of the neighbours, we would then delete the first item in the ‘queue’ so that we can move on, so that:

	0	1	2	3
0	D	G	H	I
1	K	L	P	S
2	Y	E	U	T
3	E	O	R	N

```
queue = [[ [3,2] ], [ [3,3] ], [ [3,1] , [2,2] ] ]
```

We would then look at the next item in the ‘queue’, [[3,2]], which translates to “t”, find that it leads to nowhere, delete it and move on, repeating this process with [[3,3]], which translates to “h”.

At this point, the queue will look like this:

```
queue = [[ [3,1] , [2,2] ] ]
```

The function will then take this coordinate pathway, translate it to “su”, find that there is at least one word which stems from it, take the last coordinate [2,2] and find its neighbours:

```
neighbours = [ [3,2] , [3,3] , [3,1] , [1,2] , [1,3] , [1,1] , [2,3] , [2,1] ]
```

It will then cycle through each of these, testing it, as I have previously described. Once it has finished running through and testing each one, it will have found that only the coordinate pathway [[3,1] , [2,2] , [2,1]] , which translates to “sup”, leads to any possible words. Therefore, at this point the queue will appear as so:

```
queue = [[ [3,1] , [2,2] ], [ [3,1] , [2,2] , [2,1] ] ]
```

And after deleting the first item, having exhausted all of the options leading from it at this stage:

```
queue = [[ [3,1] , [2,2] , [2,1] ] ]
```

This process will be repeated for the next round of the function, so that eventually:

```
queue = [[ [3,1], [2,2], [2,1], [1,2] ]]
```

And then:

```
queue = [[ [3,1], [2,2], [2,1], [1,2], [2,3] ]]
```

This coordinate pathway from the queue represents the image shown below here:

	0	1	2	3
0	D	G	H	I
1	K	L	P	S
2	Y	E	U	T
3	E	O	R	N

	0	1	2	3
0	D	G	H	I
1	K	L	P	S
2	Y	E	U	T
3	E	O	R	N

On the next run through of the function, taking this coordinate list, the function will firstly check if there are any words which can stem from this pathway, and then it will check if the translation of this pathway is a full word itself, and if it has already been found. If it is a full word, according to the trie, and it has not previously been found, the word will be appended to the 'possible' list. Meaning that:

```
possible = ["super"]
```

The first item of the queue will then be deleted, leaving an empty queue, which means that the function is complete, and there can be no more possible words that have not yet been found.

At this point the function will return the 'possible' list, which will then be used during the game.

Testing

Firstly, I tested the 'letterFrequency' function. When this function is running normally, during the game, it should create a list of 1000 letters, which appear more often, the more common they are in the specified language.

However, it would be impractical to show a screengrab of a list of 1000 letters, so I decided to test the function using just 100 letters, with 50 "A"s and 50 "B"s.

The exact code that I ran to complete this test is all shown below:

```
def letterFrequency(dictionary):
    alphabet = 'AB'
    letters = []
    for x in range(len(alphabet)):
        for y in range(dictionary[alphabet[x]]):
            letters.append(alphabet[x])
    return letters

testFrequency = {'A':50,'B':50}

test = letterFrequency(testFrequency)

print(test[0:10])
print(test[10:20])
print(test[20:30])
print(test[30:40])
print(test[40:50])
print(test[50:60])
print(test[60:70])
print(test[70:80])
print(test[80:90])
print(test[90:100])
```

I have specified how I want the output to be printed in order to make it easier to see it, because otherwise it would just be one long line of a list. I am expecting to see a list of 100 elements, half of which are "A", and the other half "B".

The output of the above code is shown below:

This shows that there are indeed 100 elements, and it is clear to see that half of them are "A", and half "B". Therefore the function is working correctly, as intended.

```
[ 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A' ]
[ 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A' ]
[ 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A' ]
[ 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A' ]
[ 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A' ]
[ 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B' ]
[ 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B' ]
[ 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B' ]
[ 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B' ]
[ 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B' ]
```

The next functions to test are the two functions which are used to create the 2D array, and then fill it with the 'randomised' letters, proportional to the chosen language. These two functions are called 'createGrid' and 'fillGrid' respectively.

In order to test these functions, and to make it clearer if they are actually working properly or not, I am going to test them in the same way that I tested the last function, by using half "A"s and half "B"s.

I have shown below, all of the code that I am running to test these two functions together.

```
import random

def letterFrequency(dictionary):
    alphabet = 'AB'
    letters = []
    for x in range(len(alphabet)):
        for y in range(dictionary[alphabet[x]]):
            letters.append(alphabet[x])
    return letters

def createGrid():
    grid = []
    for row in range(4):
        grid.append([])
        for column in range(4):
            grid[row].append(" ")
    return grid

def fillGrid(grid):
    for x in range(len(grid)):
        for y in range(len(grid[x])):
            grid[x][y] = test[random.randint(0,999)]
    return grid

testFrequency = {'A':500,'B':500}

global test
test = letterFrequency(testFrequency)

grid = createGrid()
print(grid)
grid = fillGrid(grid)
print(grid)
```

I firstly use the 'letterFrequency' function, which I have shown to be working properly previously, to generate a list of 1000 letters, with 500 "A"s and 500 "B"s. I then call both of the functions which I am currently testing, and print their outputs.

I am expecting to firstly see a blank 4x4 2D array, as well as that same array, filled with "A"s and "B"s, with half of them being the former and half the latter.

The exact output that I receive is shown below:

```
[[ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' ']]  
[[ 'A', 'A', 'B', 'A'], [ 'B', 'B', 'B', 'A'], [ 'B', 'A', 'B', 'A'], [ 'B', 'A', 'B', 'A']]
```

As you can see, half of the letters are indeed “A”, and half “B”, in fact it is exactly an even split.

This may not happen every time, so I tested the function a few more times, and here are the outputs that I got:

```
[[ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' ']]  
[[ 'B', 'A', 'A', 'A'], [ 'A', 'B', 'A', 'B'], [ 'B', 'A', 'B', 'B'], [ 'A', 'B', 'B', 'A']]
```

```
[[ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' ']]  
[[ 'A', 'B', 'A', 'B'], [ 'A', 'B', 'B', 'B'], [ 'B', 'B', 'A', 'B'], [ 'A', 'B', 'B', 'B']]
```

```
[[ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' '], [ ' ', ' ', ' ', ' ', ' ']]  
[[ 'A', 'A', 'A', 'B'], [ 'A', 'A', 'A', 'B'], [ 'A', 'B', 'B', 'B'], [ 'B', 'A', 'A', 'A']]
```

One of the good things about using such a large list of letters to pick from (1000), is that the variation from the mean is very low, which means that it is very likely to be exactly an even split between the two letters, but if it is not, the difference from an even split is very low, in the examples shown above it only varies by one letter either side.

This testing shows that both of these functions are working together to produce a two dimensional array, which is an abstraction of the boggle board of letters, which is exactly what they were designed to do, meaning that they are working successfully.

The final function that I tested from this design stage was the 'possibleWords' function, which takes an input of the 2D array shown, generated by the functions above, which I have shown to be working, and generates all of the possible chains of letters that form a valid word in the chosen language.

The lines of code which actually call all of these functions, which are necessary for the 'possibleWords' function to run, are shown here:

```
global root
root = makeTrie("french.txt")
global test
test = letterFrequency(frenchFrequency)
global grid
grid = createGrid()
grid = fillGrid(grid)
print(grid[0])
print(grid[1])
print(grid[2])
print(grid[3])

words = possibleWords(grid)
print(len(words))
print(words)

for x in words:
    print(x)
```

I must first create the trie, then make the array of letters, use that to create the grid (all of these functions have previously been tested), and then call the 'possibleWords' function in order to test it.

It is important to point out that I have tested this function in French.

I have shown a screengrab of the output below.

The reason why I have decided to print the grid on separate lines is so that it appears just as it will in the GUI, and it makes it a lot clearer to see and to understand what is going on here.

The function has been able to find 27 words in this grid, and all of these words are printed below, as part of the testing.

All 27 of these words have been found legally, according to the rules of boggle, they are all at least three letters long, and they do not use the same letter twice.

The reason why the shortest words appear first in the output is to do with the way that the words are found, which is by using a breadth-first search, which adds the words to the final list as it finds them, and the shortest ones are found first, as described in the design section of this function.

For the function to have worked successfully, all of these words must be possible to make by chaining together adjacent letters from in the grid. You must also not be able to form any additional words that the function has not picked up on, abiding by the rules of boggle.

If you look through each one of these words, they all meet the first criteria, because they can all be made by chains from the grid. There are also no other words which can be produced from this same grid, which my function has not already found.

Therefore, this function is working correctly, exactly as it has previously been described.

Remember that this testing has been done in French.

In this iteration I have successfully managed to produce a suitable grid of letters (which satisfies success criteria number five and six) and produce a list of all valid words therein (which will be used to satisfy success criteria number eight).

This is good progress, I am on track with the development of the project, and I am still working on the plans that I laid out in my original analysis.

```
['A', 'C', 'S', 'C']
['T', 'X', 'F', 'I']
['N', 'N', 'L', 'S']
['A', 'M', 'F', 'U']
27
['SIS', 'SIL', 'SIC', 'CIL', 'TAC',
SIS
SIL
SIC
CIL
TAC
FIS
FIL
FIC
ILS
IFS
LUS
LIS
MAN
FUS
SILS
SCAT
CILS
TACS
FLIC
FILM
FILS
FICS
FISC
FLICS
FILMA
FUSIL
FILMANT
```


Design

In this section, I intend to design the screen and its layout, the screen will be the user interface in the game, so it is important that it is able to properly do its job by giving information clearly and receiving input from the user. I also intend to create the main game loop that will be running whilst the game is being played, this will be where I take input from the user and give the relevant output.

When it came to designing the graphical user interface for my boggle game, I had two options of how I could do this. I could either use tkinter or pygame functions. An advantage of using tkinter is that it comes with python, so there is no need to install any additional features, like there would be if I were to use pygame, as I would have to make sure that every machine that I ran the program on had pygame installed as well. An advantage of using pygame is that, graphically, it is easier to work with than tkinter is, which will make my designing of the GUI far easier.

I chose to use Pygame Functions to produce the graphical user interface for my game. Pygame Functions is a library of add-ons that give the programmer simple functions for creating a graphic window, such as text boxes and input boxes.

It can be found here:

https://github.com/StevePaget/Pygame_Functions

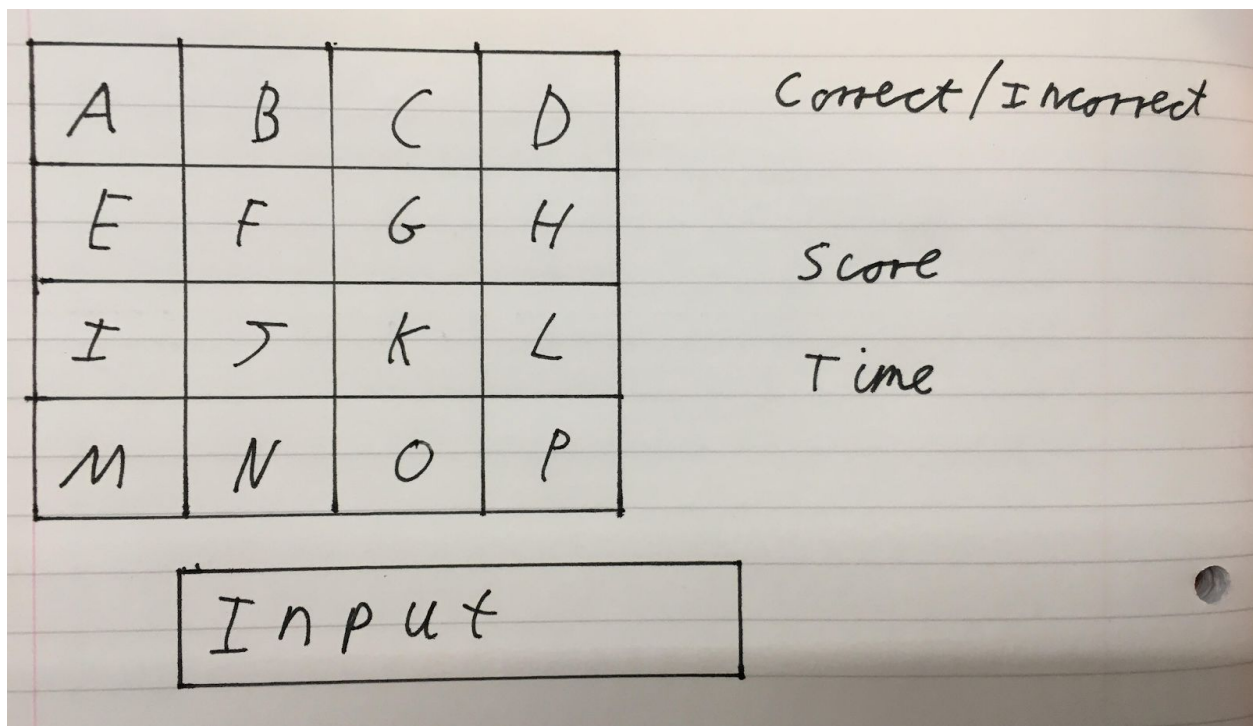
I chose to use this because the fact that it is not a built in library is not a major issue, and is resolved by simply saving the “.py” file in the same folder as my file, and then importing all of the functions at the start of the code:

```
from pygame_functions import *
```

This then makes the design stage much easier and more intuitive.

When I started to design the interface, I wanted to have the grid of letters in the top left corner of the screen, with the textbox input below that, and all other information to the right side of the screen, such as the timer and the feedback on whether a word is correct or not.

A hand-drawn sketch of what I want the screen to look like is shown below.



I have chosen to lay the screen out like this because it is the most intuitive way of doing it, as the game will be played by English children, who read from top left, it makes sense to put the most important thing (the grid of letters) in the top left corner. It is also best to have all of the information about what is happening in the game alongside this so that it is easiest to read, it would be harder to read, and to design, if this information was shown along the bottom as some of these labels will have varying length, meaning they could overlap each other and/or have big gaps between each other.

The first thing that must be done to create the GUI is to generate the Graphics Window which the game will be played in, this is done by the following code:

```
screenSize(700,700)  
setBackgroundColour("white")
```

The first line here simply uses the pygame function "screenSize" to generate a blank window that is 700 by 700, and the second line sets the background colour of the window to white, using the "setBackgroundColour" function that has been imported from pygame functions.

In order to produce the grid of letters here, I firstly created a 2D array, which held all of the letters, which have been generated using the letter frequency of the given language, as I have previously explained. I then used this to make the visual grid which is shown above. To do this, I made 16 labels, called “label1” through to “label16”, as the grid is 4x4. “makeLabel” is a pygame function, which is shown below:

```
def makeLabel(text, fontSize, xpos, ypos, fontColour='black', font='Arial', background="clear"):
    # make a text sprite
    thisText = newLabel(text, fontSize, font, fontColour, xpos, ypos, background)
    return thisText
```

This function takes an input of: the text, font size, x position, y position, and it then defaults to black text, in Arial, and with a clear background.

Below is a screengrab of the code which produces these 16 labels, and generates their position on the screen, I am using the 2D array “fullGrid” to get the letters, I am using size 14 font, and I am then positioning the labels in the correct place, with each label being 100 away from any other.

```
label1 = makeLabel(fullGrid[0][0],14,100,100)
label2 = makeLabel(fullGrid[0][1],14,100,200)
label3 = makeLabel(fullGrid[0][2],14,100,300)
label4 = makeLabel(fullGrid[0][3],14,100,400)
label5 = makeLabel(fullGrid[1][0],14,200,100)
label6 = makeLabel(fullGrid[1][1],14,200,200)
label7 = makeLabel(fullGrid[1][2],14,200,300)
label8 = makeLabel(fullGrid[1][3],14,200,400)
label9 = makeLabel(fullGrid[2][0],14,300,100)
label10 = makeLabel(fullGrid[2][1],14,300,200)
label11 = makeLabel(fullGrid[2][2],14,300,300)
label12 = makeLabel(fullGrid[2][3],14,300,400)
label13 = makeLabel(fullGrid[3][0],14,400,100)
label14 = makeLabel(fullGrid[3][1],14,400,200)
label15 = makeLabel(fullGrid[3][2],14,400,300)
label16 = makeLabel(fullGrid[3][3],14,400,400)
```

The next thing that must be done is to actually display these labels which have been generated, so that the user can see them on the screen, to do this I used the “showLabel” function from the pygame functions, which is shown below:

```
def showLabel(labelName):
    textboxGroup.add(labelName)
    updateDisplay()
```

This function only takes one input, which is the name of the label which needs to be shown.

Using this function, I wrote the following code to display all 16 labels:

```
showLabel(label1)
showLabel(label2)
showLabel(label3)
showLabel(label4)
showLabel(label5)
showLabel(label6)
showLabel(label7)
showLabel(label8)
showLabel(label9)
showLabel(label10)
showLabel(label11)
showLabel(label12)
showLabel(label13)
showLabel(label14)
showLabel(label15)
showLabel(label16)
```

The next thing that I had to produce was the background for the game, which is just a grid that I have properly resized and adjusted so that it fits all of the letters in perfectly. The code that I wrote for that is shown below:

```
background = makeSprite("Grid.png")
showSprite(background)
moveSprite(background,80,80)
transformSprite(background,0,1.2)
```

I firstly used the “makeSprite” function from pygame to create a sprite that is the grid image. I then used the “showSprite” function to actually display the sprite on the screen. Then I moved the sprite to the correct position using the “moveSprite” function. Lastly, I changed the size of the sprite using the “transformSprite” function. These functions together, produce the correctly positioned and sized background image that is displayed in the game, and that the letters fit neatly in to.

The next thing that I did was create the text-box into which the user enters their guesses. I did this with the following code:

```
textBox = makeTextBox(100,500,400,0,'',16,32)
```

This line uses the “makeTextBox” function from pygame to create a text box in the position (100,500) on the screen and is 400 wide. The maximum length of an input is also limited to 16 characters, as in this game there can be not entries which are longer than that. The font size is also set to 32, just to make the text clearer for the user to see.

After this, I created the scoring output, which is simply a label that displays the current score. It uses the “makeLabel” function from pygame functions and initially displays “Score: 0”, in size 20 font, and at position (500,200) on the screen. I then used the “showLabel” function to actually display the label on the screen. The code to do this is shown below:

```
points = 0
score = makeLabel("Score: " + str(points),20,500,200)
showLabel(score)
```

The next label that I produced was the “output” label, which is going to be used to display messages about the attempts that the user makes. such as telling them if the word is correct or not. It initially displays “Welcome” and uses size 16 font, at position (500,100) on the screen. The code used is displayed below:

```
output = makeLabel("Welcome",16,500,100)
showLabel(output)
```

Next, I created a function called “countdown”, which takes two inputs. The first input “t”, is the time that the function will start counting down from, and is therefore how long the user has to play the game. The second input “start”, is the time when the function started, I get the time by simply calling “clock()”, which gives a value of the current time. At the start of the program, I import time, and this allows me to use the function “clock()”. The time is called just before the function is run, so that it is nearly exactly the time at which the countdown function is first called, the difference is negligible. The function for the countdown is shown below:

```
t = 180
timer = makeLabel("Time: " + str(t),20,500,300)
showLabel(timer)
start = clock()
def countdown(t,start):
    now = clock()
    diff = (now-start)//1000
    left = t - diff
    left = round(left,0)
    if left < 0:
        return False
    else:
        changeLabel(timer,"Time: " + str(left))
        return True
```

Here I also created a label called “timer”, which is where the value of the time remaining will be displayed. I used size 20 font, and positioned it at (500,300) on the screen. It will initially display “Time: 180”, and the number will then subsequently decrease by 1 every second, until it reaches 0.

The next thing that I had to do, was to write the code which took the input of whatever was entered into the text box and produced the appropriate output. To do this, I enclosed everything inside a “while” loop, that was dependant on a boolean variable called “running” being set to “True”. I did this to allow myself to stop the game from running and stop taking inputs by simply changing “running” to “False”, as I thought that this would make later developments of the game much easier. The first thing that happens within this loop is the function “textBoxInput”, which is from pygame functions. The function is shown below:

```
def textBoxInput(textbox, functionToCall, args):
    # starts grabbing key inputs, putting into textbox until enter pressed
    global keydict
    textbox.text = ""
    while True:
        updateDisplay()
        returnVal = functionToCall(*args)
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RETURN:
                    textbox.clear()
                    return textbox.text, returnVal
                elif event.key == pygame.K_ESCAPE:
                    pygame.quit()
                    sys.exit()
            else:
                textbox.update(event)
        elif event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```


The most important thing to note about this function is that it allows me to also simultaneously call another function while it is waiting for the input from the text box. This allows me to call my previously described “countdown” function, so that the time remaining can be constantly updated at the same time as the user is entering an input into the text box. The line that I am using to call this function is shown here:

```
attempt, running = textBoxInput(textBox,countdown,(t,start))
```

And this is the context in which the function is being called:

```
running = True
while running:
    attempt, running = textBoxInput(textBox,countdown,(t,start))
    attempt = attempt.upper()
    if attempt == "EXITGAME":
        running = False
    length = len(attempt)
    if length < 3:
        changeLabel(output, " Words must be 3 letters or more ", background = "red")
    elif attempt in words:
        if attempt in correct:
            changeLabel(output, " You have already played this word ", background = "red")
        else:
            if length == 3 or length == 4:
                points += 1
            if length == 5:
                points += 2
            if length == 6:
                points += 3
            if length == 7:
                points += 5
            if length >= 8:
                points += 11
            changeLabel(score,"Score: " + str(points))
            remaining -= 1
            changeLabel(wordsLeft,"There are " + str(remaining) + " words left")
            changeLabel(output, " Correct ", background = "green")
            correct.append(attempt)
            if len(correct) == len(words):
                changeLabel(output, " Well done! You found all of the words ", background = "green")
                running = False
    else:
        changeLabel(output," Not a valid word ", background="red")
    end = time.time()
    if not running:
        endGame(attempt)
```

I have to take two outputs from the “textBoxInput” function. The first is “attempt”, which is the text that the user inputs into the text box, and the second is “running”, which is the value returned by my “countdown” function, this is a boolean value which is True while the time remaining is more than 0, and False if it is 0.

The next thing that happens inside of the main game loop, is that the string “attempt”, which is whatever the user has input into the text box, is changed to entirely upper case. This is because all of the letters in the grid, as well as all of the words that this will then be compared to, will also be entirely upper case. I do this with the following line:

```
attempt = attempt.upper()
```

The next stage is then to assess what the input is, and then give the correct output. Firstly, I check if the word entered is three or more letters long, if it is, then I carry on, but if it is not, then I change the “output” label to tell the user that a word must be at least three letters long. This is shown below:

```
if length < 3:  
    changeLabel(output, " Words must be at least 3 letters ", background = "red")
```

The next thing that I do, is check if the current attempt is a valid word, I do this by performing a linear search on the “words” list, which I created earlier, and have previously described. This is a list of all of the valid words that can be played using the given board, and the given language. If the user input appears in the list, then it is a valid word, but if it does not, then it is not a valid word, and so gains no points. If the word is not valid, then the “output” label changes to display a message which tells the user this. However, if the word is valid, then we move onto the next stage of dealing with the input.

After this, I check if the user has already played the word that they are attempting to play, I am able to do this, because I initially created an empty list called “correct”, which I then fill with the correct words that the user plays. I then perform a linear search on this list, to see if it contains the user’s current attempt. If it does, then the “output” label is changed to tell the user that they have already played this word, if it does not contain it, then we just carry on to the next stage.

Once we have established that the word is both valid, and has not been previously played, then we must figure out how many points the word is worth. To do this, I name a variable “length” as being the numerical value of the length of the word, this is to save some processing power, as it is more efficient to do this process one time, than to do it every time I want to compare the length of the word. I then run through a series of “if” statements:

```
if length == 3 or length == 4:  
    points += 1  
if length == 5:  
    points += 2  
if length == 6:  
    points += 3  
if length == 7:  
    points += 5  
if length >= 8:  
    points += 11
```

This will simply add the appropriate number of points to the “points” variable, which holds the current value of points that the user has scored.

After this, I then change the “score” label to display the updated score, change the “output” label to display the word “Correct”, and then append the correct word to the “correct” list, so that the user is not able to get points from using the same word again. This is done in the following three lines:

```
changeLabel(score,"Score: " + str(points))
changeLabel(output, " Correct ", background = "green")
correct.append(attempt)
```

Finally, I perform one last check, which is to test if the user has played all of the possible words. To do this, I use the “correct” list, which I have been adding to every time the user played a correct word, and the “words” list, which is the list I generated earlier that contains all of the possible words that could be played. I compare the length of these two lists, and if they are equal, then that means that the player has been able to find all of the words, so they have completed the game. If this happens, the “output” label displays a congratulations message and the “running” variable is set to “False”, so that the game stops running.

```
if len(correct) == len(words):
    changeLabel(output, " Well done! You have found all of the words ", background = "green")
    running = False
```

The entire routine which I have just described is shown on the next page, so that you can see everything in context.

```
running = True
while running:
    attempt, running = textBoxInput(textBox, countdown, (t, start))
    attempt = attempt.upper()
    if attempt == "EXITGAME":
        running = False
    length = len(attempt)
    if length < 3:
        changeLabel(output, " Words must be 3 letters or more ", background = "red")
    elif attempt in words:
        if attempt in correct:
            changeLabel(output, " You have already played this word ", background = "red")
        else:
            if length == 3 or length == 4:
                points += 1
            if length == 5:
                points += 2
            if length == 6:
                points += 3
            if length == 7:
                points += 5
            if length >= 8:
                points += 11
            changeLabel(score, "Score: " + str(points))
            remaining -= 1
            changeLabel(wordsLeft, "There are " + str(remaining) + " words left")
            changeLabel(output, " Correct ", background = "green")
            correct.append(attempt)
            if len(correct) == len(words):
                changeLabel(output, " Well done! You found all of the words ", background = "green")
                running = False
    else:
        changeLabel(output, " Not a valid word ", background="red")
    end = time.time()
    if not running:
        endGame(attempt)
```

Testing

Using all of the previously mentioned code, this is what the graphical user interface looked like when it was displayed on screen:

O	S	O	O	Welcome
A	O	T	U	Score: 0
E	U	C	I	Time: 175
L	E	T	R	

There are many different elements to this screen, as there are lots of different functions being called to make things happen and appear in different places:

1. The grid of letters are all displayed as individual 'labels', and are evenly spaced and appropriately sized so that the user can easily see them, but they are not too big.
2. The grid image has been correctly moved and scaled so that the letters all fit into the individual boxes, making the game more realistic, as this is how the game appears when it is played in its board version.
3. There is also the 'welcome' label, which will change to show any relevant information about what the user has just entered, once they do enter something.

4. The 'score' label is also appearing as intended, it is correctly displaying the current score, and will change as soon as the user enters a correct word.
5. There is also the 'time' label, which is initially set to 180, as this is the specified amount of time that a user should be allowed, according to the game's creators.
6. Lastly, there is the text box, into which the user enters their attempts, which appears as intended, underneath the board, the size of the box is appropriate, as no word entered can possibly be longer than sixteen letters, as each letter on the board may only be used once, and I have tested to see that this size of box is right for that number of letters, as they do not reach the end of the box, and the letters are still big and clear enough to see.

This means that I am satisfied with how the graphical user interface is appearing on the user's screen, because it means that the game is simple and easy to play, which is exactly what I intended it to be. Therefore this section is working correctly.

I also tested the input section here, to ensure that the program was taking the user's inputs properly. To do this, I added the print command shown here in the main game loop, which happens as soon as the input is taken and then capitalised.

```
running = True
while running:
    attempt, running = textBoxInput(textBox, countdown, (t, start))
    attempt = attempt.upper()
    print(attempt)
    if attempt == "EXITGAME":
        running = False
    length = len(attempt)
    if length < 3:
        changeLabel(output, " Words must be 3 letters or more ", background = "red")
    elif attempt in words:
        if attempt in correct:
            changeLabel(output, " You have already played this word ", background = "red")
        else:
```

The below images show how I was using this to test the input functionality, I simply entered words into the text box and checked the output in the shell to see if it matched up properly.

Here is what the interface looked like when I was doing this testing:

L	G	I	T
I	T	U	T
T	M	E	L
L	U	O	T

tut

And here is what the outputs in the shell looked like:

TULE
LIT
GIL
TIG
LEMOT
TUT

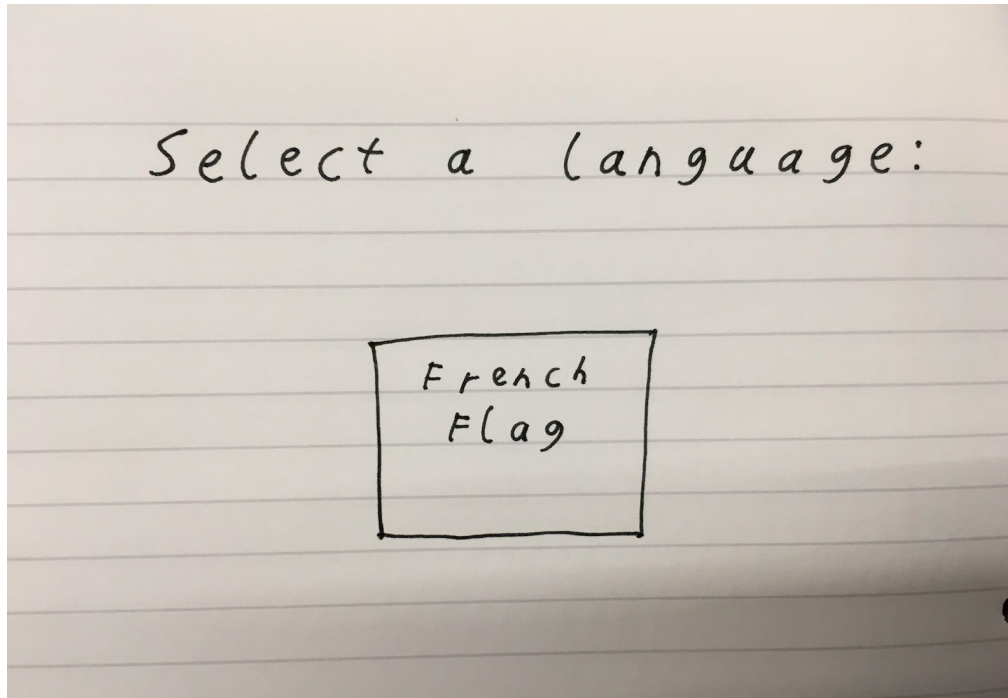
These outputs exactly match what I entered into the text box (but uppercase), meaning that the input functionality is working perfectly.

After I finished the designing and testing of this section, I asked my end user to test the current version of the game and give me some feedback. I got two different teachers from the modern languages department of my school to tell me what they thought about the current interface of the game. Both told me that they thought that all of the information that the game displayed was very useful and relevant to the student that would be playing the game, and would help them to better understand what was going on. This is exactly what I was hoping to achieve in this section, so I can conclude that this has been successful and I have achieved success criteria number eleven.

Design

The next stage of my design process was to create a 'menu' style page that appears at the start of the game, and a 'game-over' style page that appears at the end of the game.

My initial design for the menu page is shown here.



The reason why I wanted to create a menu page, is so that it would be much easier in the future to add additional languages to the game, because all I would need is to add a button which causes the game to start in that language.

The code that I wrote to display this on the screen is shown below:

```
menuText = makeLabel("Select a language: ", 60, 150, 200)
showLabel(menuText)
french = makeSprite("FrenchButton.png")
transformSprite(french, 0, 0.6)
moveSprite(french, 400, 400)
showSprite(french)
```

I chose to make the menu very minimalist, in order to allow it to be very easy and clear for the player to understand what they are supposed to do, without any explicit instructions. The primary purpose of the page is to allow the user to select a language in which to play the game. When I first designed the menu, the game could only be played in French, so there was only one button on the screen that could be clicked. In order to make this menu work, I created the image of the French flag as a 'sprite' in the game, because there is a function in 'pygame functions' which allows me to check if the user has clicked on a particular sprite, and this is exactly what I need to know when my menu page is running.

With the actual code which is running in the background while the menu page is displayed and the game is waiting for the user to select a language, is the following one:

```
menu = True
while menu:
    if spriteClicked(french) == True:
        loading()
        root = makeTrie("french.txt")
        letters = letterFrequency(frenchFrequency)
        menu = False
    pause(10)
```

The purpose of this loop is to run continuously, pausing after each iteration for 10ms, and to check if the user has clicked on the sprite, indicating that they have selected a language in which they would like to play the game.

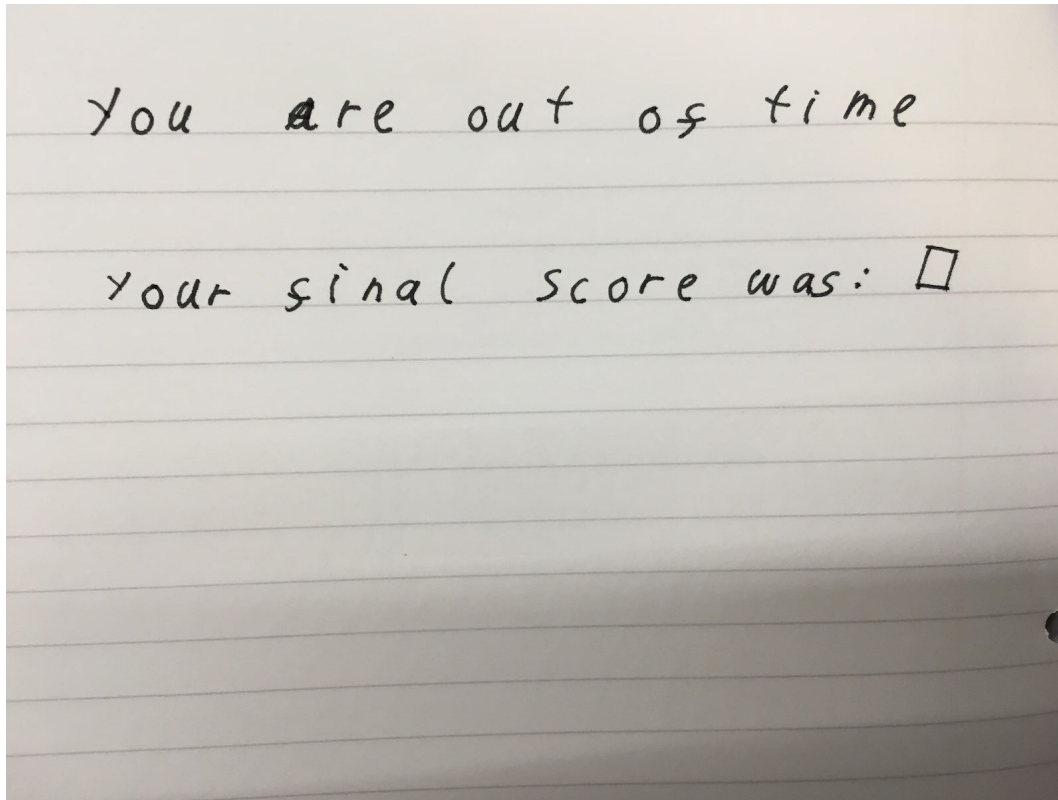
To add more languages, I would simply have to add an extra 'if' statement inside of this 'while' loop, which checks if a different sprite has been clicked, and then to call the same functions, but using the dictionary and letter frequency from that language.

I have also placed a function inside this loop which will be run when the user does make a choice called "loading()" (shown here), this is another function that I created which hides all of the menu graphics and displays a loading message on the screen while the all of the required data structures are being built. The idea behind this is that, if there were no loading screen, then, to the user, it would appear as though the game had frozen and was no longer working properly. The main reason for this is that the "makeTrie()" function takes a long time to complete (~15 seconds on the VM at school), it is the bulkiest function in the entire program and if the game simply froze on the menu screen for the time that it took to run the function, it is possible that the user would just quit the game, because they think that it has crashed, or because they are tired of waiting.

```
def loading():
    global load
    hideLabel(welcome)
    hideLabel(menuText)
    hideSprite(french)
    hideSprite(english)
    load = makeLabel("Loading...", 60, 250, 250)
    showLabel(load)
```


The next page that I created was the “game over” style page, which is displayed when the game has finished. This page can be reached by one of two ways; the user has been able to find all of the possible words, or they have run out of time.

My initial design for this end-of-game page is shown below.



This design is very basic, and includes a very limited amount of information for the user, as it is a first sketch, I will work on this page later in the development process in order to make it more useful and to fill it with more information for the player to see.

The actual code which I wrote to produce the end-of-game screen is shown here:

```
info = makeLabel("You are out of time",30,250,100)
showLabel(info)
yourScore = makeLabel("Your final score was:",30,50,300)
showLabel(yourScore)
finalScore = makeLabel(str(points),100,120,335)
showLabel(finalScore)
```


In order to make all of these labels fit together properly and work with each other, I then put them all into functions. I did this to make it very easy for me to make multiple things happen at once during the game. For example, I created a function called “startGame” because I wanted certain labels to be generated and displayed at the same time in the game and after a particular thing happens, so it makes sense to include them all in a function together, so that I then call the function after the user has selected a language in which they wish to play the game from the menu.

As part of making sure that this system of functions being used to call for the creation of labels, I had to define a lot of the variables as global variables, so that they could be accessed by different functions, without having to be passed to the function when it is called. The result of this meant that after I created all of the variables which were actually labels or sprites, I had to make them global variables, as shown here.

When I was creating the ‘menu’ function, which allowed the user to select the language in which they wished to play the game (a choice, shown here, between english and french), I had to change the ‘root’ variable to be a global variable, so that it could be accessed by other functions in the game.

```
background = makeSprite("Grid.png")
moveSprite(background,80,80)
transformSprite(background,0,1.2)
showSprite(background)
global background
output = makeLabel("Welcome",16,500,100)
showLabel(output)
global output
points = 0
global points
score = makeLabel("Score: " + str(points),20,500,200)
showLabel(score)
global score
timer = makeLabel("Time: " + str(t),20,500,300)
showLabel(timer)
global timer
textBox = makeTextBox(100,500,400,0,'',16,32)
global textBox
correct = []
global correct
```

```
menu = True
while menu:
    if spriteClicked(french) == True:
        loading()
        root = makeTrie("french.txt")
        letters = letterFrequency(frenchFrequency)
        menu = False
    if spriteClicked(english) == True:
        loading()
        root = makeTrie("english.txt")
        letters = letterFrequency(englishFrequency)
        menu = False
    pause(10)
global root
global letters
```

This had a knock-on effect on some of the functions which I created earlier and have previously talked about. This effect was that I no longer needed to pass some functions the 'root' node variable, as they could access it anyway, so there was a small change to the way in which the function 'searchTrie' was defined, meaning that when I now call it, I only have to give it one parameter, which is the word that it is searching for, and not also the node that it has to start from.

Before, the function was called like so:

```
searchTrie(word,root)
```

Whereas, now, it is called like this:

```
searchTrie(word)
```

The main benefit of this is that it is more convenient to call the function now.

Testing

I then tested out all of the functionality which I have just described above, to make sure that it was working properly, and there were no issues.

So when I first run the game, what appears on the screen is exactly what is shown below:

Select a language:



This is exactly what I intended to have appear, with the “Select a language:” text appearing in the centre of the screen, and the French flag button appearing directly below it, ready to be clicked.

When the user uses the mouse to click on the image, the game starts as normal, and the user can then play it. Which is what is supposed to happen.

Therefore, the menu page is working successfully.

Next, I moved onto the testing of the 'game-over' page, the screengrab below shows the page, as it would appear when the timer has reached zero, and the user has not yet played any valid words.

You are out of time

Your final score was: 0

The below image shows the same 'game-over' page being tested, but this time with the user having actually score some points (14 in this case), just to show that this feature also works when the user is playing the game properly.

You are out of time

Your final score was: 14

The purpose of this page is to clearly inform the user of what has happened, and why the game has ended, and also to give them all of the relevant information that they might need to understand how well they played. As such, the "You are out of time" message appears in large text at the top of the screen, to make it clear what is going on, and the user it also told what their score was, also very clearly, so that they can compare their score to their friend's score and know how well they have actually done.

This page is working properly, so therefore the testing has been successful.

At this point in the development process, I had something that was ready to give to the user and get some feedback on. So I asked my end-user to play the game in its current state and tell me what they thought of it. After playing the game, they told me that the menu page was very easy to understand and they think that it will look better once the other languages are added to it, which will happen later in the development of the project. They also found it useful to have a loading page appear whilst the game was creating the relevant data structures so that any students will not think that the game has crashed. The final thing that my end-user commented on was the end of game page, which, at this stage, shows a very limited information, even so, they said that it was critical information (why the game has ended and what their final score was). In future development I will be adding more information to this screen so that it is far more useful to the player.

In this section I have fulfilled success criteria number nine (giving the user their score at the end). This design and testing iteration has been successful and I am still on track to complete the project within the given time-frame.

Design

After completing the previous design stage, I decided that the next stage of development of the game would be to show some additional information in the end-of-game screen, which is shown once the time has run out, or the user has found all of the possible words. I thought that some useful information to display would be the longest word(s) that the player could have played, which would have scored them the highest number of points, and to then compare that with the longest word(s) that they actually played, so they can have a better sense of how well they did.

To do this, I wrote two functions, one called “longestWords”, and the other called “longestCorrect”, the first function is shown below:

```
def longestWords(words):  
    if len(words[-1]) != len(words[-2]):  
        out = "The longest possible word was:<br>"  
    else:  
        out = "The longest possible words were:<br>"  
        counter = -1  
        while len(words[counter]) == len(words[-1]):  
            out += "<br>" + str(words[counter])  
            counter -= 1  
        return out
```

One of the biggest issues that I had to solve whilst writing this function was in determining when to have the output be plural and when it had to be singular, as the output is a string of text, which I can then immediately display in a multi-line label from pygame functions. For example, it would not be okay for there to be two words which were both equally as long as each other, but for the text to read “The longest possible *word* was:”, this would make the design look a bit shabby. So in order to solve this problem, the first thing that I do in the function is to test whether there are going to be multiple words which are all as long as each other, I do this in the first line in the function, this works out because the list “words” is already sorted in ascending word length order, due to the fact that I used a breadth first search algorithm to find all of the possible words which could be made with the grid. Therefore, if the two last words are not equal in length, the output is only going to be one word, but if they are equal in length, the output is going to include at least two words. In the latter case, I then carry on counting back from the end of the list, adding all the words that have the same length as the last one, until I reach one that does not, in which case I stop and return the output string.

The next function is similar in a way to the previous one, in that it is achieving the same output, but it requires a lot more validation of the input list, as I have no idea what the size of the list will be, and it is also an unsorted list. The code for the function is shown below:

```
def longestCorrect(correct):
    length = len(correct)
    if length == 0:
        return "You didn't play any correct words."
    out = "The longest word you played was:<br>"
    if length == 1:
        return out + "<br>" + str(correct[0])
    correct = sorted(correct, key=len)
    if len(correct[-1]) != len(correct[-2]):
        return out + "<br>" + str(correct[-1])
    out = "The longest words you played were:<br>"
    out += "<br>" + str(correct[length-1])
    counter = 1
    while len(correct[-1]) == len(correct[length-counter]) and counter < length-1:
        out += "<br>" + str(correct[length-counter])
        counter += 1
    return out
```

The first thing that I do, to speed up the process of computing this function, is the name a variable "length" as being the numerical value of the length of the list "correct", this will save the computer from having to do this process repeatedly, whenever I want to use the numerical value in the function. After this, I have to validate the list, by first checking if there are actually any words in it, because if there are none, I can just return an output straight away and stop the function. I then do a similar check, for if there is just one word in the list, because if this is the case then I can immediately output the appropriate string. Next comes the main part of the function, where I first sort the list so that it is in ascending length order using a python library function. My next check is whether the output is going to include just one, or more than one word, I do this by checking if the last two words in the sorted list have the same length, because if they do not, then we know for certain that there is only one longest word, however, if their lengths are equal then we know that the output is going to contain at least two words. Once we have established that there are going to be multiple words then I begin a 'while' loop which repeats until it comes to a word which does not have the same length as the ones which have already been added, working backwards from the end of the list. Once the loop terminates, the 'out' string is then outputted, which is then used in a label to display the words to the user in an appealing format.

Testing

In order to test these two functions, I took copies of them and ran them without the graphical user interface.

Firstly, I tested the 'longestWords' function to check if it was doing what I wanted it to do, which is to take a list full of words which are ordered by their length, to find the longest one(s), and to then output these words in a string, which will then be shown at the end of the game, using a multi-line label. Below is a screengrab of the the important parts of the code that I am running to produce this test (the other parts have been previously seen and explained).

```
grid = createGrid()

global letters
global root

letters = letterFrequency(frenchFrequency)
root = makeTrie("french.txt")

fullGrid = fillGrid(grid)
words = possibleWords(fullGrid)

print(words)
print(longestWords(words))
```

I have used many of the same variables and functions that I use in the full version of the game in order to create the trie, create the randomly generated grid of letters, and to then find all of the possible words from this grid. I then gave the list of 'words' to the function, and printed both the list and the output of the function, using the list. The output from this code is shown below.

```
['MUT', 'NEE', 'NES', 'SUE', 'SUT', 'SUC', 'SEN', 'SEC', 'PUE', 'PUT', 'PUS',
'TUE', 'TUS', 'TUT', 'ERE', 'ECU', 'CET', 'CPT', 'CES', 'USE', 'TEE', 'REE', '
EUE', 'EUT', 'EUS', 'ETE', 'SUEE', 'SUER', 'SUCE', 'PUEE', 'PUER', 'PUTE', 'PU
CE', 'TUEE', 'TUER', 'ECUS', 'CREE', 'CENS', 'USER', 'USEE', 'TUTU', 'RECU', '
REES', 'ERES', 'SUCEE', 'SUCER', 'SUCRE', 'PUEES', 'PUCES', 'TUEES', 'ENCRE',
'CREEE', 'CREES', 'USNEE', 'RECUE', 'RECUT', 'RECUS', 'RECES', 'SCUTUM', 'SUCR
EE', 'ENCREE', 'CREUSE', 'RECUSE', 'RESUCEE']
The longest possible word was:<br><br>RESUCEE
```

I then tested ran this program again until I got a list of words which had more than one longest word, meaning that there were several words that were as long as each other and no others were longer. The output of this is shown below.


```
[ 'OTE', 'OTA', 'TOI', 'TOT', 'TER', 'TES', 'TAC', 'TOC', 'CLE', 'OIE', 'OIS', 'E
RE', 'ETC', 'EST', 'SEL', 'SIR', 'SET', 'LES', 'ROI', 'ROT', 'RIE', 'RIS', 'IRE'
, 'ACTE', 'OTER', 'OTES', 'OTAT', 'TOTO', 'TETA', 'TEST', 'TATE', 'TACT', 'TOCS'
, 'CLES', 'COTE', 'COTA', 'ORES', 'OIES', 'ERES', 'ETAT', 'ETOC', 'SIRE', 'LEST'
, 'LESE', 'ROIS', 'ROTE', 'ROTA', 'ROTS', 'REIS', 'RETS', 'RIES', 'RIRE', 'IOTA'
, 'IRES', 'ACTER', 'ACTES', 'OCTET', 'TOISE', 'TERSE', 'TETAT', 'TESTA', 'TATER'
, 'TATES', 'TACTS', 'COTER', 'COTES', 'COTAT', 'COTTE', 'ETATS', 'ETOCs', 'ESTOC
', 'ESCOT', 'SERIE', 'STERE', 'STORE', 'SEOIR', 'LESTE', 'LESTA', 'LESER', 'ROTE
R', 'ROTES', 'ROTAT', 'RESTA', 'RISER', 'RIRES', 'RESTE', 'ESTER', 'OTATES', 'OC
TETS', 'TORIES', 'TOISER', 'TESTAT', 'COTTES', 'COTOIE', 'SERIER', 'SIROTE', 'SI
ROTA', 'STATOR', 'LESTER', 'LESTAT', 'RESTAT', 'RIOTER', 'RESTER', 'COTATES', 'C
OTOIES', 'SIROTER', 'SIROTAT', 'STATERE', 'SCOOTER', 'ROTATES']
The longest possible words were:<br><br>ROTATES<br>SCOOTER<br>STATERE<br>SIROTAT
<br>SIROTER<br>COTOIES<br>COTATES<br>
```

As you can see, the first thing that has been printed is a list of words, order according to their length, and below this is the output from the 'longestWords' function.

In the first test it correctly identifies the word "RESUCEE" as being the longest word in this list, and in the second test it correctly identifies "ROTATES", "SCOOTER", "STATERE", "SIROTAT", "SIROTER", "COTOIES" and "COTATES" as being the joint longest words in the list.

In both of these tests, the output is also a string which makes sense, because it uses the singular or plural version of the relevant words (word/words and was/were).

The '
' tags are also in the correct place, so that when the multi-line label function reads the text in the string, it creates line breaks in the appropriate places, so that the presentation is neat and clear to understand. As such, this function has been shown to be working properly and correctly, as it was intended.

Next, I am going to test the 'longestCorrect' function. To do this, I could play the full version of the game, and add in two 'print' commands to the following code, which is run at the end of the game, when the user has run out of time (or found all of the words).

```
couldHave = longestWords(words)
didPlay = longestCorrect(correct)

print(correct)
print(didPlay)

possible = makeLabel(couldHave, 23, 25, 400, "black", "Arial", "white")
played = makeLabel(didPlay, 23, 425, 400, "black", "Arial", "white")
showLabel(possible)
showLabel(played)
```

However, instead of doing this, I decided to do some white box testing, which is much faster and easier to do. To do this, I wrote the following program:

```
def longestCorrect(correct):
    length = len(correct)
    if length == 0:
        return "You didn't play any correct words."
    out = "The longest word you played was:<br>"
    if length == 1:
        return out + "<br>" + str(correct[0]) + "<br>"
    correct = sorted(correct, key=len)
    if len(correct[-1]) != len(correct[-2]):
        return out + "<br>" + str(correct[-1]) + "<br>"
    out = "The longest words you played were:<br>"
    out += "<br>" + str(correct[length-1])
    counter = 2
    while len(correct[-1]) == len(correct[length-counter]) and counter < length+1:
        out += "<br>" + str(correct[length-counter])
        counter += 1
    return out + "<br>"

correct = ['TES', 'CAVE', 'BOL', 'LOB', 'BLOC', 'LAVES', 'OUT', 'SET', 'TUBE']
print(correct)
print(longestCorrect(correct))
```

The output of this is shown below. The first line is the list of all of the 'correct guesses' that the user would have made during the game, in the order that they were entered in, and the second line is the output from the 'longestCorrect' function.

```
['TES', 'CAVE', 'BOL', 'LOB', 'BLOC', 'LAVES', 'OUT', 'SET', 'TUBE']
The longest word you played was:<br><br>LAVES<br>
>>>
```

I tested the function again with another word added in, which was also five letters long, to make sure that it still worked properly with multiple longest words. The output from this is shown here.

```
['TES', 'CAVE', 'HELLO', 'BOL', 'LOB', 'BLOC', 'LAVES', 'OUT', 'SET', 'TUBE']  
The longest words you played were:<br><br>LAVES<br>HELLO<br>  
>>>
```

As you can see, the 'correct' list in both tests is completely unordered, and it is of unknown length when it is passed to the function, but the function is still able to find the longest word(s) from the list. It is also able to correctly format a suitable output, using the correct version of singular/plural nouns and the correct preposition for a list with a single element or multiple elements. The '
' tags are also placed in the correct place, so that they will look correct when this text is displayed by the multi-line label. Therefore, this function is shown to be working properly, just as I designed it to.

Now I am going to test both of these functions together, and test what their output will actually look like when it is displayed by the multi-line label. To do this, I took a screengrab of what the two labels look like when they are displayed together, next to each other, at the end of the game. This is exactly how the user will see them when they have finished playing the game, and they want to see how well they have done, and compare that to how well they could have possible done.

The six lines of code which call the two functions, create the labels, and then show the labels on the screen, are shown below.

```
couldHave = longestWords(words)  
didPlay = longestCorrect(correct)  
possible = makeLabel(couldHave, 23, 25, 400, "black", "Arial", "white")  
played = makeLabel(didPlay, 23, 425, 400, "black", "Arial", "white")  
showLabel(possible)  
showLabel(played)
```

And the output of this code is shown below. This is exactly what the user sees.

The longest possible words were:

METTAIENT
TENDAIENT

The longest words you played were:

ETAIT
ETAIN
TETIN
DENTE

As you can see, this example is with the game being played in French.

This shows that the output is very clear, explaining exactly what it is showing, and showing it, giving the user useful information, so that he/she can evaluate his/her performance, and compare it to the best possible performance.

This means that my two functions are working exactly as they are designed to, conveying clear and useful information to the user, at an appropriate time in the process of the program.

Evaluation (Testing)

In this section, I will have my end-user do some black box testing of the end product, in order to determine if they are satisfied with what my project actually does. Here I am not asking them to concern themselves with what the game is doing behind the scenes, only with what they are actually seeing, and their experience of playing the game.

During the final black box testing of my project by the end-user, they raised an issue that they experienced when they were testing the game on a different operating system. They told me that when they were playing the game, the functionality was fine, but the text appeared bigger or smaller than it should be, different to how it was when they had previously played the game on their windows desktop at school. Below is a screengrab of what the game looked like on the user's computer, highlighting the issue.



I realised that this issue was being caused by the default text size/font that different operating systems use, because the experience is consistent across all computers which are running on the same OS. This issue could be fixed by specifying a particular font and size for the text in the labels which appear in my game. However, this issue is actually irrelevant when it comes to the final implementation of my project, how it will be used by my end-user, this is because the game, as specified by my user's requirements, will only ever be played on windows desktops in my school. This is the exact same system that I initially designed the graphics on, meaning that I have optimised them to look exactly as they should using this default text font and size, so the game's graphics will always appear exactly as intended when it is being played.

In future development, and if my stakeholder's requirements change, I could specify the font and size of all of the text in my project, so that the graphical user interface appears exactly the same for every operating system, and I know how it will always look, irrespective of what the user is running it on.

There is also another slight issue that was raised by this testing, which I was able to rectify very quickly and easily. This is that, as you can see in the above image, there is a rectangle shaped 'bullet point' at the start of most of the lines. I realised that this was being caused by how I was writing the text that was being shown in the labels, I was using multiline strings, as well as the '
' tags to create another line, and this was causing those strange looking 'bullet points'.

Before, the code for the labels looked like this, and the output looked like it does above.

```
boggle = makeLabel("Welcome to Boggle",70,150,25)
showLabel(boggle)
rules = makeLabel('''You will be shown a 4x4 grid of letters.<br>
    You must make as many words as you can within the time limit.<br>
    The longer the word, the more points it scores.<br>''',35,25,180)
showLabel(rules)
rules2 = makeLabel('''Words can only be made according to the following rules:<br>
    1. The letters must be adjoined in a 'chain' (vertically, horizontally or diagonally).<br>
    2. Words must contain at least three letters.<br>
    3. No letter cube may be used more than once within a single word<br>''',25,25,300)
showLabel(rules2)
play = makeSprite("play.png")
transformSprite(play,0,0.2)
moveSprite(play,300,450)
showSprite(play)
```

Whereas, now that I have fixed these issues, the code looks like this, with the text all appearing in a single line string.

```
boggle = makeLabel("Welcome to Boggle",70,150,25)
showLabel(boggle)
rules = makeLabel('You will be shown a 4x4 grid of letters.<br>You must make as many words as you can within the time limit.
The longer the word, the more points it scores.<br>')
showLabel(rules)
rules2 = makeLabel('Words can only be made according to the following rules:<br>1. The letters must be adjoined in a 'chain'
2. Words must contain at least three letters.<br>3. No letter cube may be used more than once within a single word<br>')
showLabel(rules2)
play = makeSprite("play.png")
transformSprite(play,0,0.2)
moveSprite(play,300,450)
showSprite(play)
```

The output of this modified code is shown below:

Welcome to Boggle

You will be shown a 4x4 grid of letters.

You must make as many words as you can within the time limit.

The longer the word, the more points it scores.

Words can only be made according to the following rules:

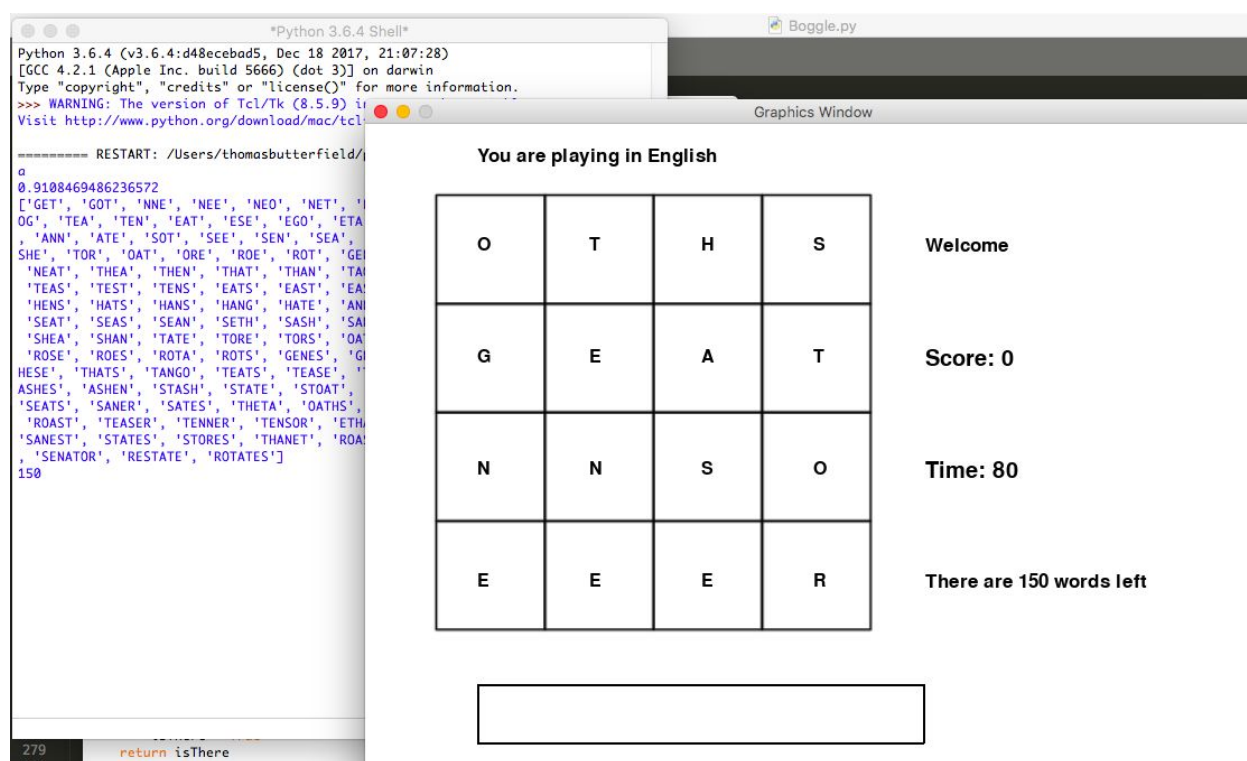
1. The letters must be adjoined in a 'chain' (vertically, horizontally or diagonally).
2. Words must contain at least three letters.
3. No letter cube may be used more than once within a single word



(The numbers are also correct now)

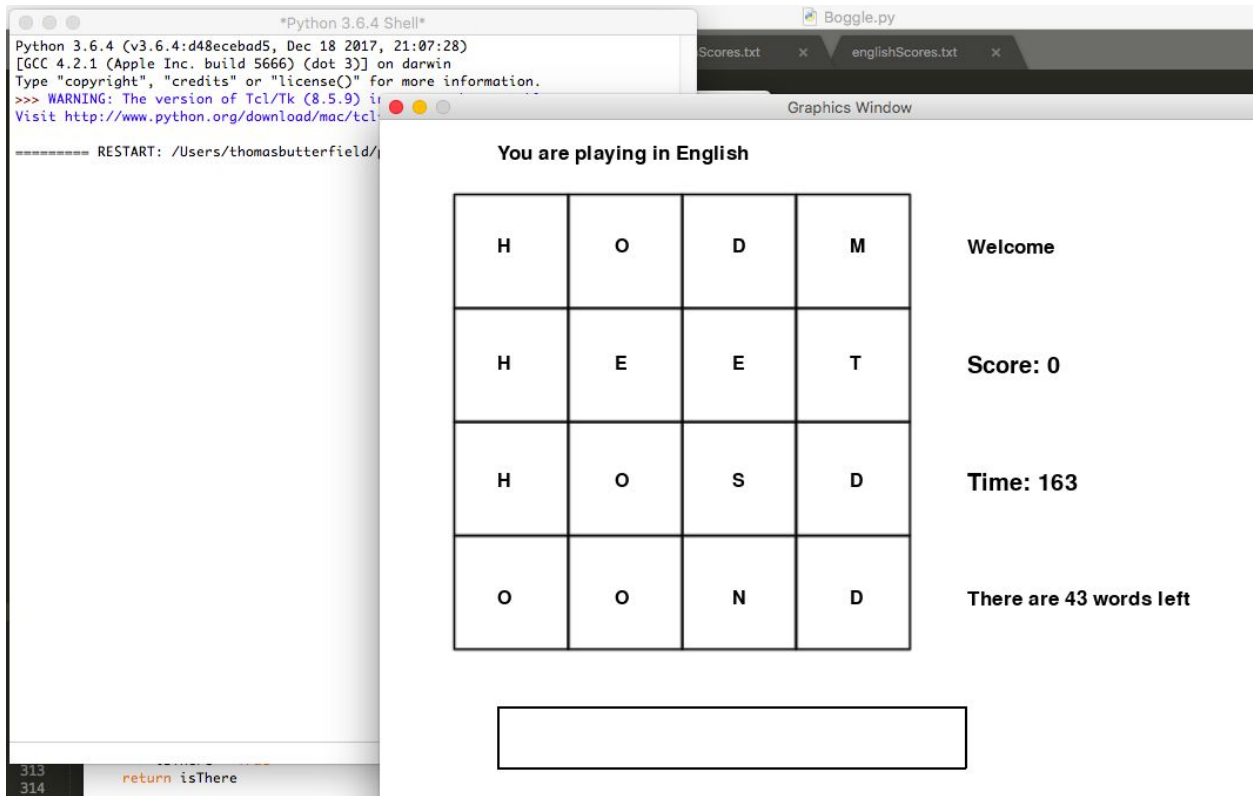
Some other feedback that my end-user gave me was that they really liked the way that the screen tells them how many words are left to find, also pointing out the fact that you don't get that with the board game.

Another thing that they highlighted to me was that when the game is run, it still prints out all of the possible words which could be made. This is a feature that I added in whilst I was developing and testing the game, but simply forgot to remove. They also said that there were some other bits of information that were being printed that they didn't understand. These, again, were things that I printed during the development process and just hadn't removed, such as the user's name, the time it took for the game to load, and the number of words available to be found. Below is an image of how it appeared at the time:



I have since modified the game so that it does not print all of this information anymore.

Now, there is nothing printed out when the user is playing the game, so the students cannot cheat or anything. What the game now looks like is shown below:



Similar to the problem which I have previously mentioned, the user experienced issues with the font sizes in the end-of-game page. This was due to reasons which I have previously mentioned, a different operating system with a different default text size. Below is what the user saw when they encountered the issue.



And here is what they should have seen, and now do see:

Well done Laura

You are out of time



Your final score was:

16

Highest Scores: 54 by Thomas

46 by Soham

42 by WINNER

The longest possible word was:

ATHENIAN

The longest word you played was:

THINE

Currently, when the game is finished, and the high scores are shown, the scores are not classified by the language in which they were played, they are all grouped together irrespective of the language in which they were achieved.

The end-user requested that this be changed so that the high scores which are shown at the end of the game are just for that language, because it would be easier to get a higher score in English than it would be in French, for example. This would mean that the user was competing on a more level playing field, as it were.

In order to do this, I created two different text files, one called "frenchScores.txt", and the other called "englishScores.txt", I then added some 'if' statements that took the language that the game had been played in, and decided which text file to open, to both read from and write to.

Before, the part of the program which dealt with the scores and the high scores looked like this, there was a single text file called "scores.txt", and it was always used.

```
scoreList = []
scoreSheet = open("scores.txt","r")
for line in scoreSheet:
    if len(line)>1:
        scoree,namee = line[:-1].split(',')
        scoree = int(scoree)
        scoreList.append((scoree,namee))
scoreSheet.close()
scoreList.append((points, name))
scoreList = sorted(scoreList)
scoreSheet = open("scores.txt","w")
for item in scoreList:
    scoreSheet.write(str(item[0]) + "," + item[1] + "\n")
scoreSheet.close()
high1 = scoreList[-1]
high2 = scoreList[-2]
high3 = scoreList[-3]
highScore = makeLabel("Highest Scores: " + str(high1[0]) +
showLabel(highScore)
```

Whereas now, this part of the program looks like this (next page):

```

scoreList = []
if language == "French":
    scoreSheet = open("frenchScores.txt","r")
if language == "English":
    scoreSheet = open("englishScores.txt","r")
for line in scoreSheet:
    if len(line)>1:
        scoree,namee = line[:-1].split(',')
        scoree = int(scoree)
        scoreList.append((scoree,namee))
scoreSheet.close()
scoreList.append((points, name))
scoreList = sorted(scoreList)
if language == "French":
    scoreSheet = open("frenchScores.txt","w")
if language == "English":
    scoreSheet = open("englishScores.txt","w")
for item in scoreList:
    scoreSheet.write(str(item[0]) + "," + item[1] + "\n")
scoreSheet.close()
high1 = scoreList[-1]
high2 = scoreList[-2]
high3 = scoreList[-3]
highScore = makeLabel("Highest Scores: " + str(high1[0]) +
showLabel(highScore)

```

As you can see, now I am using two different text files to store the scores of users who have played the game in different languages. I use the 'language' variable to decide which file I am going to be reading from and writing to, this is a global variable that is created when the user decides which language they want to play the game in.

After doing this, the high scores are displayed like so:

Firstly, if you have played the game in French:

Highest Scores: 9 by Thomas
(In French) 8 by Barnaby
7 by Steve

And then, if you have played the game in English:

Highest Scores: 19 by Hello
(In English) 8 by Barnaby
7 by Steve

The final comments that the end-user had about the project were as follows:

“Thanks for making the game. We’ve been having fun with it and it’s proving to be quite popular with the A level students. It would be even better if it could be multiplayer, but I understand if that’s too hard to do for now. Maybe another day.”

I am very appreciative of the fact that they have enjoyed the game, and that the students who they have been testing it with have enjoyed it as well. The issue that they have raised here, to do with the multiplayer aspect of the game, will be explored and explained in much greater detail in the next section (evaluation).

In relation to success criteria number two, which was to help the user to learn the vocabulary of their chosen language in an enjoyable way, I tested if I had met this requirement after I had finished my development process.

I had ten different French students play my game several times, so that they had several opportunities to score well and get a good experience of the game, after this, I asked each of them if they think that their knowledge of French vocabulary had improved, and nearly every one of them told me that they thought that it actually had. I also asked them if they preferred the old style of learning and testing of vocabulary using pen and paper, or the game, and every one of them said they much preferred to play my game than to sit regular vocabulary tests. This means that I have successfully achieved criteria number two in my original requirements section.

In my requirements section, success criteria number seven states that the Graphical User Interface of the game must be attractive for the user, so that it will appeal to school aged children. I tested the game at the end of the development process in order to find out if I had achieved this. I asked 10 students who are studying French to play the game and to give me their feedback and thoughts on how the appearance of the game on the screen could be improved, and how it currently looks. Most of them said that they liked the way the game looked because it made it easy to understand what was going on and all of the information was clear to see. However, a few students did say that they would prefer it if the screen appeared brighter and more colourful, as it would engage them even more.

In relation to success criteria number six in my original requirements, which states that the letters which are displayed in the grid:

“should appear proportional to their commonality in the given language to enable players to make more words in a given language.”

I have successfully achieved this in my project, but in order to actually investigate how much of a difference it really made to the game to have the letters specifically proportioned to their frequency in the given language, I tested it out.

To do this, I created an aside program, using the same functions as are in my original program. I compared the way that I am creating the board in the game (with the letters appearing proportional to their commonality in the language), with a board which was filled by entirely random letters, with no favouritism to any letters whatsoever, meaning that there is no consideration for the commonality of these letters in the selected language.

In this test, I am creating a full grid each time I am running the test, and then searching it to find all of the possible words that could be found using it.

Here I have a screengrab of the functions which are being called, and the main code which is being run in this test.

```
randomGrid = createGrid()
grid = createGrid()

global letters
global root

letters = letterFrequency(frenchFrequency)
root = makeTrie("french.txt")

for x in range(50):
    fullRandomGrid = fillRandomGrid(randomGrid)
    fullGrid = fillGrid(grid)
    randomWords = possibleRandomWords(fullRandomGrid)
    words = possibleWords(fullGrid)
    print(str(len(randomWords)) + ' ' + str(len(words)))
```

As you can see, I am creating two separate grids, one called 'randomGrid', and one called 'grid', I then run the test 50 times, each time filling the two grids, with one totally randomly filled, and one filled with letters proportional to their commonality in the language.

The function that I wrote specifically for this test, to fill the grid entirely randomly is show here:

```
def fillRandomGrid(randomGrid):
    alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    for x in range(len(randomGrid)):
        for y in range(len(randomGrid[x])):
            randomGrid[x][y] = alphabet[random.randint(0,25)]
    return randomGrid
```

Below is a screengrab of the output of the program, in each line, the number on the left is the number of possible words which could be played if the board was completely randomly generated, and the number on the right is the number of words that could be played, using a board generated in the same way as it is in the game.

```
Python 3.6.4 Shell
Python 3.6.4 (v3.6.4:d48eceed5, Dec 18 2017, 21:07:28)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: /Users/thomasbutterfield/python/Project/frequencyTesting.py ====
8 56
3 106
7 96
24 64
26 119
18 140
14 172
10 58
6 64
7 17
19 52
43 87
18 156
8 134
1 307
12 78
80 129
2 64
30 84
2 75
0 98
22 101
39 36
8 66
24 93
1 130
1 46
93 54
2 175
8 200
57 61
5 167
11 147
11 56
1 111
3 73
37 192
12 125
3 112
8 145
6 107
2 53
35 131
23 102
45 137
```

It is clear to see that the numbers on the right are substantially and consistently larger than the numbers on the left.

This clearly shows that it does actually make a huge difference to how easy it is to play the game and to find words by creating the grid in the way that I have done.

If the letters were all chosen completely randomly, then the game would be very boring, as there would be very few words for the user to find.

Evaluation

After completing the design and testing, I am very pleased with my final product.

To refer back to my success criteria, I have achieved criteria number one, which was to allow a user to play a game of boggle, in their chosen language, this was really the minimum criteria that I wanted to complete to classify this project as being successful, and I have been able to do that.

Criteria number two has also been achieved, helping the user to learn the vocabulary of their chosen language in an enjoyable way, and is evidenced in the previous section (Evaluation (Testing)), which is on the previous page.

Criteria number three, which was to build a trie data structure for the relevant word set at the start of the game so that the list could be quickly searched during the game, was achieved in the first design section, and then proved to be working properly in the first testing section.

The fourth success criteria was for the game to be able to load in less than 15 seconds, this was shown to be achieved in the first testing section, where I showed that even under worst case scenario conditions, the trie could be built in around 14 seconds. The second part of this success criteria was that, whilst the game was being loaded, the game should display a loading screen, so that the user does not get bored or think that it has crashed. This specific criteria was achieved in the fourth design section when I created a page that hides all of the labels and other graphics from the screen, and displays a clear loading message. The code for this function is repeated below, for reference.

```
def loading():  
    global load  
    hideLabel(welcome)  
    hideLabel(menuText)  
    hideSprite(french)  
    hideSprite(english)  
    load = makeLabel("Loading...", 60, 250, 250)  
    showLabel(load)
```

In reference to criteria number five, the game is able to successfully display a 4x4 grid of letters, which was done in the third design section, and shown to be working as intended in the third testing section.

Related to the previous criteria, criteria number six stated that the letters in this grid should appear proportional to their commonality in the given language, so that users can find more

words in the game. The functions which make this happen were explained in the second design section, and were shown to work successfully in the second testing section.

For my seventh success criteria, which was to create an attractive and interesting GUI, this is a rather vague requirement, so to quantify it, I surveyed my users at the end of the development stage of the project, as I have explained in the above section.

The issues which arose from the user testing of the GUI in the previous section are issues that could be addressed in further development of the project, because for the purposes of the success criteria, the interface is functional and technically achieves the criteria. It is worth pointing out that, as with most Graphical User Interfaces, it can always be improved and there will always be something that could be changed about it. I am happy that, as explained in the previous section, most of the users did actually find the interface to be easy to understand, and that it conveyed all of the relevant information.

In reference to my eighth success criteria, that the game should be able to (virtually) instantly tell the user if the sequence of letters that they have entered is actually a word that can be made according to the rules of the game. In the second design section of my project, I explain how I created the function which is able to do this, by creating a list of all of the possible words at the start of the game, so that I only have to perform a simple search of a list each time the user enters a word in order to determine if they are right. There are other ways in which I could have created a function which did this same job that would mean that there would be a delay between the user entering a word and the relevant feedback being provided, so I am quite pleased that I was able to think of a very efficient way of doing this.

My ninth success criteria was to tell the user how many points they scored at the end of the game. I am glad to say that I was able to actually surpass this original requirement by, as well as displaying their score at the end of the game, also displaying their score constantly throughout the time they are actually playing the game. This is a very good thing as it means that the user is constantly aware of what their current score is, so they know how far off a certain target they are and is a very useful way for them to track their progress. I included the constantly displayed score that is shown while the game is being played in the third design section, and I added the score being displayed at the end of the game in the fourth design section, I then tested this in the fourth testing section.

For requirement number ten, which was to have a 180 second time limit on the game, so that it is all about finding as many words as possible in a limited time, this was successfully achieved in the third design section, here I also created a constantly updating label which displays the time remaining, so the user is aware of what is going on.

The eleventh success criteria, to create a screen layout, was achieved in design section number three. To meet this criteria, I had to create a graphical interface which would be useful, but not too cluttered, so that it contained an appropriate amount of information that is relevant to the user and their experience, but yet still allows them to focus on the game, and isn't distracting. I can refer back to my third design and testing section in this document to evidence that I have achieved this criteria successfully. Here I asked my end user to test out the graphical user interface, and they told me that this design did perform as it was intended to do.

Success criteria number twelve was that the game had to congratulate the user in the rare case that they were able to find all of the words that could be made using the grid. This is done in the third design section. This was achieved by using an 'if' function which checks if the number of words found is equal to the number of words which could be found, if they are equal, this means that all of the words have been found. The code for this appears like so:

```
if len(correct) == len(words):  
    changeLabel(output, " Well done! You have found all of the words ", background = "green")  
    running = False
```

In the context of the main game loop, it looks like this (next page):

```

running = True
while running:
    attempt, running = textBoxInput(textBox, countdown, (t, start))
    attempt = attempt.upper()
    if attempt == "EXITGAME":
        running = False
    length = len(attempt)
    if length < 3:
        changeLabel(output, " Words must be 3 letters or more ", background = "red")
    elif attempt in words:
        if attempt in correct:
            changeLabel(output, " You have already played this word ", background = "red")
        else:
            if length == 3 or length == 4:
                points += 1
            if length == 5:
                points += 2
            if length == 6:
                points += 3
            if length == 7:
                points += 5
            if length >= 8:
                points += 11
            changeLabel(score, "Score: " + str(points))
            remaining -= 1
            changeLabel(wordsLeft, "There are " + str(remaining) + " words left")
            changeLabel(output, " Correct ", background = "green")
            correct.append(attempt)
            if len(correct) == len(words):
                changeLabel(output, " Well done! You found all of the words ", background = "green")
                running = False
    else:
        changeLabel(output, " Not a valid word ", background="red")
    end = time.time()
    if not running:
        endGame(attempt)

```

These images are both taken from the third design section in this document.

I should also point out that it is very rare that a user will be able to find all of the possible words before the time runs out, because there are often some obscure words which are there to be found that the user may not have heard of before, which is why it is such an achievement if somebody is indeed able to do this.

Requirement number thirteen was to have the game produce a list of the longest words that the user could have played, so that they can see the highest scoring words. This will help them to learn vocabulary as they will be constantly exposed to long words in their chosen language, which they will be then be able to pick up and use in their work. This criteria was achieved in design section number five, where I actually exceeded this requirement, by not only displaying the list of longest words that they could have played, but also, alongside that, a list of the longest words that they did play during the game.

This is a great way for the user to see how well they did, and compare that to how well they could have done. I have repeated the code for both of these functions, from the design section, below:

```
def longestWords(words):
    if len(words[-1]) != len(words[-2]):
        out = "The longest possible word was:<br>"
    else:
        out = "The longest possible words were:<br>"
    counter = -1
    while len(words[counter]) == len(words[-1]):
        out += "<br>" + str(words[counter])
        counter -= 1
    return out
```

```
def longestCorrect(correct):
    length = len(correct)
    if length == 0:
        return "You didn't play any correct words."
    out = "The longest word you played was:<br>"
    if length == 1:
        return out + "<br>" + str(correct[0])
    correct = sorted(correct, key=len)
    if len(correct[-1]) != len(correct[-2]):
        return out + "<br>" + str(correct[-1])
    out = "The longest words you played were:<br>"
    out += "<br>" + str(correct[length-1])
    counter = 2
    while len(correct[-1]) == len(correct[length-counter]) and counter < length+1:
        out += "<br>" + str(correct[length-counter])
        counter += 1
    return out
```

The output which is shown on screen, using these functions, is also repeated here:

The longest possible words were:

METTAIENT
TENDAIENT

The longest words you played were:

ETAIT
ETAIN
TETIN
DENTE

This is exactly what the user sees when they finish the game.

Success criteria number fourteen of this project was to allow the teacher to look at the list of scores that their pupils have achieved, alongside their names, so that they can see how well each student has done. This would enable the teacher to help out particular students, and congratulate others. With the current system, since each game is stored and played locally on a particular computer, the way that this would be done is by each student sending the teacher the text file which contains the list of all of their scores. This system is flawed because, if a student knew how to, they could change their scores before they send the text file over to the teacher. This system could be improved in further development by making the game web-based, because then I would be able to send the score of a particular student directly to the teacher immediately after the game has ended, making the student unable to change it in any way.

Success criteria number fifteen was about making the game playable in a multiplayer format, I was not able to achieve this criteria due to the reasons I mentioned in the limitations section of the analysis stage. The biggest factor in this was that my school's network security protocols do not allow two computers on the network to communicate directly with each other.

If I were to develop this project further, in order to meet this criteria and make the game able to be played in a multiplayer format, I would most likely need to make it web-based. This would allow me to host it on a website, allowing two users to play with or against one another in an online version of the game. This could also speed up the time in which the game loads, as it would be possible to have the majority of the processing done remotely, on a server, so that it didn't matter what kind of computer the user was playing the game on, they would still have the same experience.

Another advantage of taking the game online would be that a user would only need a browser to access and play the game, rather than requiring a virtual machine installed on their computer as they do with the current version.

One of the criteria which I was able to meet according to the standards of the original requirements, but would like to improve upon further, if I were to develop this project more in the future, is the time that it takes to build the trie, and to load the game. Currently, this takes around fourteen seconds under worst case scenario conditions, which is less than my original success criteria stated (15 seconds). However, I think that this could be cut down to a shorter time by cutting down the length of the dictionary of words, or by making the function which creates the trie much more efficient, so that it runs faster.

In terms of how the system will need to be maintained, there is not very much that would need to be done to keep the game fully operational in the future. The only thing which could perhaps be done would be editing the dictionary of words that is used as a reference in the game for each language, to either add new words as they are invented, or to delete words which are no longer classified as valid words in that language.

It is possible that the requirements of the stakeholders will change in the future, and so if I were to keep on improving this project, I would have to develop the game to meet those new requirements. A possible future demand of the stakeholder could be that they want the game to be easier to play and more accessible to students. If I were to do this, I would, as I have previously mentioned, make the game web-based so that students were able to play the game from home without the use of any python software.

In the future, the end-user could also ask for the game to be developed further so that instead of the user typing in their guesses, they instead click the letters on the screen that they want to try to form a chain with. This would be achievable in further development of the project, but with my current stakeholder requirements, it was not necessary to do.