MATH 495R, HOMEWORK 24: BOGGLE

In this lab, we will program the computer to play Boggle, a popular word search game.

Rules of Boggle: Boggle is played with 16 dice, each of which has letters on its faces. The dice are shaken, and then arranged into a four-by-four grid. Players then have three minutes to search the grid for words and write them down. A word in the grid is created by using letters that touch each other in sequence. The letters may be adjacent in any direction (horizontally, vertically or diagonally, either up or down, right or left). No letter cube may be used more than once in a single word (although it may be used multiple times to form different words)

When time is up, players compare their word lists, and delete any words that are on more than one list. The words remaining on the list are then scored according to the following table :

3-4 letter words	1 point
5 letter words	2 points
6 letter words	3 points
7 letter words	5 points
8+ letter words	11 points

Note that one and two letter words receive no points.

As an example, a Boggle grid, together with some words from the grid are listed below.

E	E	A	L
M	A	R	O
S	A	I	S
I	K	T	G

Some words found in this grid are: ARE (1 point), AREA (1 point), ARIA (1 point), ARIAS (2 points), EAR (1 point), EARS (1 point), EARL (1 point), EMERITA (5 points), EMERITAS (11 points), LARIAT (3 points), LARIATS (5 points), RATIO (2 points), RATIOS (3 points), SOIREE (3 points), TAR (1 point), TIARA (2 points), TIARAS (3 points).

Note that there are many other words that can be found in the grid above.

One letter cube has the letters "QU" on a single face; although these letters are on a single cube, when used in a word, they count as two letters towards the length of the word. They must be used in order ("Q" first, immediately followed by "U").

We will program a computer to generate a complete list of all words found in a Boggle grid. Obviously, playing the game against such a computer would not be much fun (since we would always score 0 points), so we will not bother to program a way for us to play the game.

Before beginning our program, we need to download a dictionary. For this project, download the Scrabble word list found at

`https://www.wordgamedictionary.com/twl06/download/twl06.txt`

and store it in the directory where you will be writting your program.

In addition, I have created a template for today's project. You can download it at

`https://math.byu.edu/~doud/Math495R/Lab24template.py`.

In the template, I have prewritten two functions for you: `create_boggle_grid()`, which scrambles the letter cubes, and creates the random grid of letters, and `print_grid(grid)`, which prints out a boggle grid. In these functions, a boggle grid is stored as an two-dimensional array of characters. In addition, the letter cube with the letters "QU" is represented in these functions as just "Q". As an example, the Boggle grid on the previous page would be represented as

```
[['E','E','A','L'],['M','A','R','O'],['S','A','I','S'],['I','K','T','G']]
```

In order to complete today's project, you will need to code the following functions.

1. We begin by writing a function `load_words()` that reads the dictionary file into an array, and returns the array of valid words in the dictionary. To do this, we need to use some basic file reading operations.

   To open the file for reading, we begin with the statement

   ```
   with open('twl06.txt') as word_file:
   ```

   This statement creates a variable `word_file` that points to the dictionary file, and allows us to read it. We will read it one line at a time. The command

   ```
   line=word_file.readline()
   ```

   reads a single line from the file `word_file`, and places it in the variable `line` as a string. Note that when reading a line from a file, the resulting string typically ends with a character indicating the end of the line. You should remove this character, and put the resulting word into an array named `valid_words`, which should be returned by the function.

   The words in the dictionary file are in alphabetical order—you should make sure that the words remain in order as you put them into the array.

2. The next function that we will write is `check(word,i,valid_words)`. The input here is a string `word`, and integer `i` and a list of all valid words from our dictionary.

   The variable `i` indicates a point in the dictionary so that every entry `valid_words[j]` with `j<i` comes before `word` alphabetically. A perfectly good value for `i` would be 0, but by inputting this value, we can often speed up our program significantly.

   The program should find the first value of `i` for which `word<=valid_words[i]` (i.e. either `word=valid_words[i]` or `word` comes before `valid_words[i]` alphabetically. For instance, if `word` is the string "accus" (which is not a valid word), the function should find `i` so that `valid_words[i]='accusable'`. Note that the relevant portion of the dictionary reads

   ```
   ...,'accurst','accusable','accusably','accusal',accusals','accusant',...
   ```

   so all words that precede `'accusable'` in the dictionary also come before `'accus'`.

   We note a few things about the value of `i` that we have found.

   (a) If `word` is a word in the dictionary, it should equal the value of `valid_words[i]` at this point.

(b) If the letters in `word` are the beginning of any valid words in the dictionary, `valid_words[i]` should start with the letters in `word`.
(c) If the letters in `word` are not the beginning of `valid_words[i]`, then there is no word in the dictionary that starts with `word`.

The function should check if `word` is the beginning of `valid_words[i]`. If it is, the function should return `i`, otherwise it should return `-1`.

One point of this function is that it is not only finding whether `word` is in the dictionary, but also whether `word` is the beginning of any words in the dictionary.

3. The function `seek(grid,m,n,used,word,location,valid_words,wordlist)` is the core of the program, and has the following inputs.
   - `grid` is a Boggle grid, as a two-dimensional array.
   - `m,n` give a location in the Boggle grid, representing the next letter that we wish to look at.
   - `word` consists of the letters that we have already traversed in the Boggle grid, put into a string.
   - `used` gives the coordinates of the letters in the Boggle grid that are already contained in `word`; we cannot use these letters again.
   - `location` give the location in the dictionary of the first word equal to or following (in alphabetical order) `word`.
   - `valid_words` is the dictionary of allowable words.
   - `wordlist` is the list of all words that we have already found in the Boggle grid.

   We will want to use a global variable `score` in this function. To do this, use the command `global score` after the function declaration. This is a variable that is defined outside of the function, but that we nevertheless want to access inside the function.

   The function should perform the following actions:
   - Create a new word, consisting of `word` with the letter in `grid[m][n]` tacked onto the end. Note that this new word will come after `word` alphabetically. Since `word` is a string, changing it here would change it outside the function as well–we do not want to do that, so we need a new variable name; I suggest `new_word`.
   - If the letter we added to `word` is a “Q”, then we should also add a “U” to it (since the two letters come together on a letter cube).
   - Use the function `location=check(new_word,location,valid_words)` to find whether `new_word` is a valid beginning for a word in the dictionary. If not, we do not need to deal with it anymore, and we will not do the following steps.
   - Check if `new_word` is actually a valid word in the dictionary (i.e. if it is equal to `valid_words[location]`). If it is, and it is not already in `wordlist`, add it to `wordlist`, and add the appropriate number of points (based on its length) to `score`.
   - Whether or not `new_word` is a valid word in the dictionary, loop through all of the adjacent squares to `grid[m][n]`. If the square has not already been used, recursively run `seek(grid,new_m,new_n,used+[[m,n]],new_word,location,valid_words,wordlist)`

   Note that `seek` DOES NOT NEED TO RETURN ANY VALUES! It updates the relevant variables directly, so that there is no need to return a value. The last action on the list above is the trickiest–it may take a few lines of code to accomplish.

   At this point, if we set `wordlist=[]` and run `seek(grid,0,0,[],"",0,valid_words,wordlist)` the function should find all words in the grid starting with the letter in the [0,0] location.

4. The final function that we need is `solve_grid(grid,valid_words)`. It should initialize `wordlist` to `[]`, and then run the function

   `seek(grid,i,j,[],"",0,valid_words,wordlist)`

   for all $0 \leq i, j < 4$. When it finishes, it should sort `wordlist` into alphabetical order, and print it, and then print "The score is `score`."

Tips

- To convert a string to uppercase, use `string.upper()`. To convert a string to lowercase, use `string.lower()`.
- While debugging, it might be convenient to use the same Boggle grid over and over. To do this, pass an integer to the function `create_boggle_grid()`. This integer will be used as a seed for the random number generator, so you will get the same grid each time you run the program.
- If your dictionary is not in alphabetical order, there will be some difficulty. In particular, if the dictionary contains a line at the beginning that is not a word in the dictionary, you may need to deal with this.
-
- In debugging your program, it might help to use the same Boggle grid over and over. to do this, change

  ```
  grid=create_boggle_grid()
  ```

  to pass an integer to the function `create_boggle_grid()`. For instance, you could change it to

  ```
  grid=create_boggle_grid(49359778)
  ```

  to use a board that includes an 11 letter word and has a score of 306 points. Another good number to use is 98674512, which gives a board with a score of 799.

  The integer passed to `create_boggle_grid()` acts as a seed for the random number generator, guaranteeing that all random choices from that point on are made in the same way each time.