Thomas Back

CS 470

Professor Doerry

2/2/2018

Project 1: Fred Flinstone Boggle-Solver

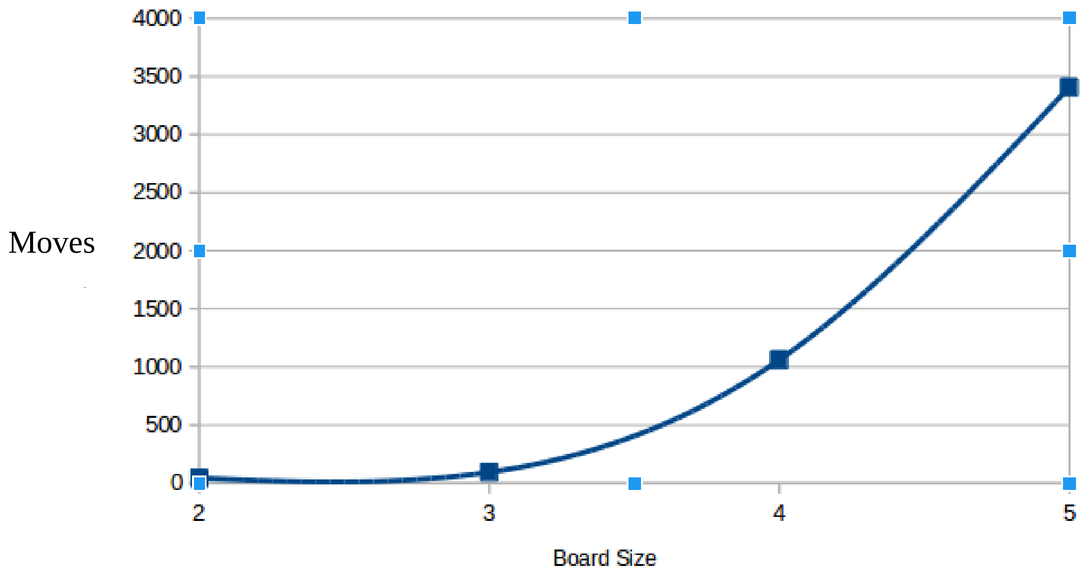
Analysis:

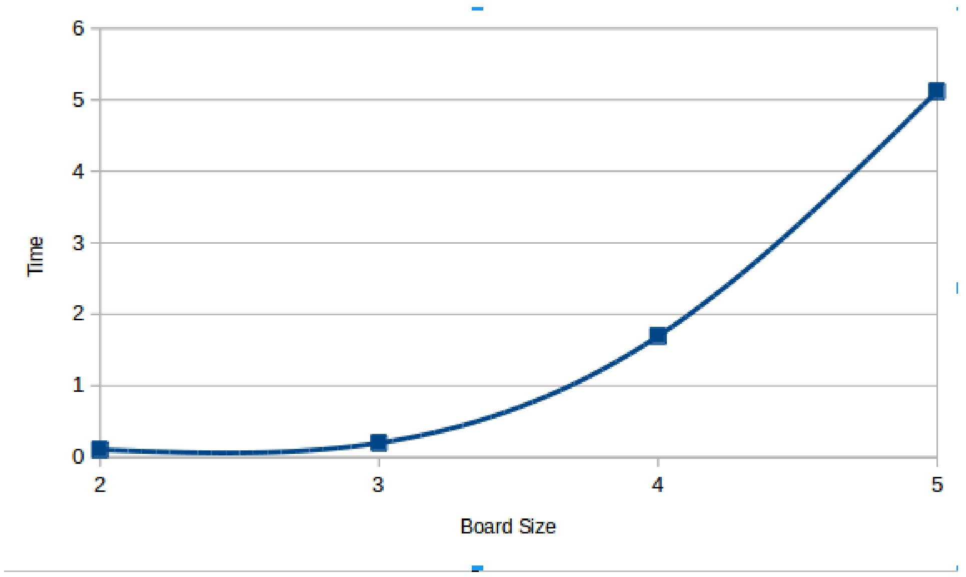
1. A clear description of your algorithm, i.e., the approach/strategy that your code takes in solving the problem. Being able to clearly outline an algorithmic approach is a valuable communication skill in our business, and demonstrates the extent to which you truly understand what you're doing. Do NOT walk through your functions/code in low-level detail! You need to describe how your program solves the problem abstractly, as in the key features of the approach and the steps the program goes through in executing it.

> A boggle board can be thought of a grid with an axis for each letter. Depending on the value of N, the board will always be an NxN value, giving the agent the knowledge to begin the search at each position of the board. The rules of boggle allow a path to be followed to any surrounding panel, including across and diagonal panels, the only restriction being that a previously visited panel can never be used as a possible option in the same search branch. With this in mind the agent computes possible paths, and sorts out the possible paths with valid path options by cross referencing the previous path. Once a collection of all next valid moves is created, the same set of steps can be followed recursively, branching out into searches based on the next panel that is selected. At each panel, that panel's letter is added to a string to be passed through each recursion to be checked if it is a valid string within the dictionary, and if it is a complete word, before running each next iteration. If it is not a word, the search branch will be cut by returning out of that iteration. One global structure has a found word added to it from each search branch as the list of found words.

2.

a. Observe your own results: Run your solver on several 4x4, 3x3, and 2x2 boards. How many different words did your solver explore on each? How much time was taken on each. Now analyze: Come up with a curve showing your results. Then use this to predict the time/moves it would take to explore a 5x5 board.





On the 2x2 board the solver was able to find 3 unique words; the 3x3 board found 0 words; the 4x4 board returned 71 words. In terms of time taken to find all solutions, the 2x2 took 0.107713 seconds, the 3x3 took 0.201461 seconds, the 4x4 took 1.6938 seconds. There seems to be an exponential difference between the values of times and moves as N rises, giving a parabolic curve. I attempted to find what exponent as the power of N would give the time in seconds.

For 2x2: $2^x = 0.107713$, $\log_2(0.107713)$ gave me -2.5.

for 3x3 : $3^x = 0.201461$ $\log_3(0.201461)$ gave me -1.45

for 4x4: $4^x = 1.6938$ log4(1.6938) gave me .38013

After this analysis, it did not seem like there was a correlation between this value with only the exponent as a value and that there must be more to the complexity. I believe the exponent must still be around 2 since the NxN value will add some measure of N^2 complexity. If we then consider that would be required for a full search to take us across every panel of an NxN grid, then we would have $N*2(N^2)$ for the overall complexity, but still does not compute the exact time it would take for 5x5.

Inputting the first four points into a graph and having it average the 5th has returned around 3500 moves for a 5x5 and an estimate of about 5 seconds to complete.

b. Analyze the problem generally: How many possible combinations of letters (i.e. actual words or not) can be constructed from an NxN board? Walk through your reasoning carefully, showing how your value comes together. Let's keep it simple just to get a decent upper bound without needing a PhD in combinatorics: Ignore detailed paths possible on the board and just assume that every letter on the board could be chained with every other letter on the board...how many words could be made that way? How does your analysis match up with your empirical findings in the last question?

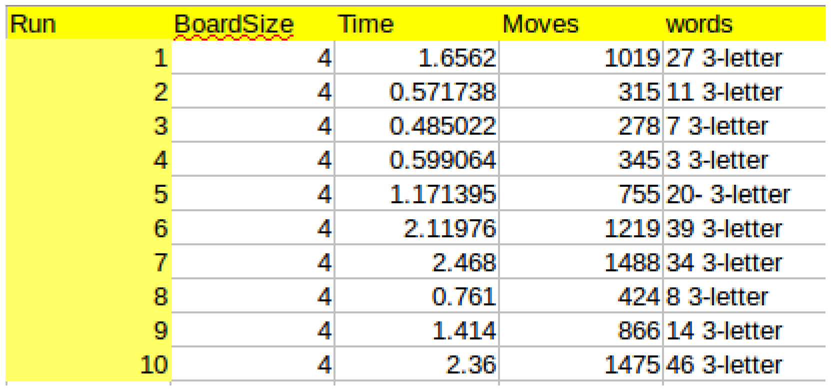
In order to solve how many possible combinations of letters can be matched on a board, we can look at an example board and its state of letters:

D U I T
N Q K Y
U A P G
N C H Y

If we take a look at the above 4x4 board we can see that if we snake across the board from D or (0,0) from right to left proceeding down a column at the end of each row, then we could have a 16 letter long word, if it turned out to be a word. With there being a possible of 26 letters in the alphabet and assuming that any letter could be chained with the next despite it being a word or not, we could get $N^2(26!)$ The factorial is applied to the 26 as each square could provide that amount of possible combinations chained with the following.

c. Use your solver to solve at least 10-20 different boards, then ponder the solution stats you got. Based clearly on your observations, consider the following: Suppose there is a Boggle competition where human players are given a sequence of boards to solve, and the time they have to do so decreases with each board. Now examine the outcomes from the boards that you've run your solver on. What strategy for finding words would a "smart" (or as we'll call it in this course, "rational") player employ to maximize points in a time-limited time? Don't just speculate, support your answer clearly with your empirical results!

After running the solver on a board of 4x4 size 10 times, I received these results.



Run	BoardSize	Time	Moves	words
1	4	1.6562	1019	27 3-letter
2	4	0.571738	315	11 3-letter
3	4	0.485022	278	7 3-letter
4	4	0.599064	345	3 3-letter
5	4	1.171395	755	20- 3-letter
6	4	2.11976	1219	39 3-letter
7	4	2.468	1488	34 3-letter
8	4	0.761	424	8 3-letter
9	4	1.414	866	14 3-letter
10	4	2.36	1475	46 3-letter

The first thing I noticed is that the time varied widely in between a few of them, with the moves of course having a close correlation to rising along with the time. The second thing that caught my eye is that on runs with the longest times and most moves, the amount of 3-letter(or words less than 4 letters long in general) were much higher. If my agent was running in a competition then it would undoubtedly perform irrationally under the concept of a time limit, because it would hang up spending its time gathering an abundance of smaller length words. These smaller length words that are not a part of a bigger board could be remembered with each play, and the boggle solver could drop the paths that follow down towards these words with no resulting bigger word. The smaller length words accumulate less points than the bigger words so in order to solve points and maximize time, the solver's smart aspect could be a memory bank of smaller words that could not result in forming a bigger word and a feature to drop the search if these words are beginning to form.

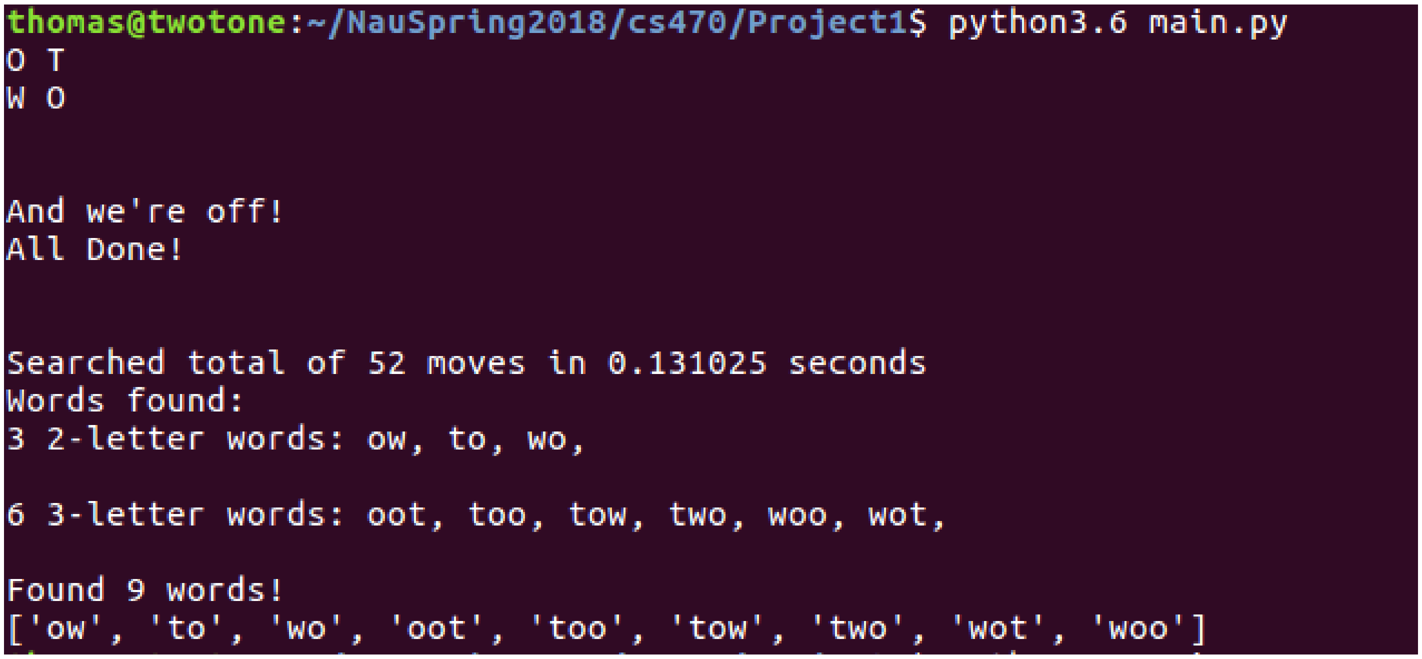
d. In the sample output provided above, there are two runs on the same board shown, with/without "cleverness" turned on...with drastic differences in time/resources used to find identical results. What the heck could that devious Dr. D be doing here to achieve this? Magic? Hint: put in some print statements to watch your program work...and then reflect on the implications of where effort is wasted.

The difference between the two outputs in my source code is exactly one line of code...plus another easily-created resource.

One indicator of the underlying cleverness mechanic is the difference in total moves between the outputs. Without cleverness for example on a 4x4 board, there is a total of 120,294,640 moves, which is about 200,000 more moves than the output for the clever run through. After removing a line from my own code that checks to see if the current string is even a valid option within the entire dictionary, I realized that this could be the piece of code that helps guide search branches from veering off into totally fruitless paths. Dr. Doerry most likely has a data structure of the dictionary for fast access of checking the prefixes of the built words, and will cancel that search branch if the string becomes invalid, rather than check to see if the string is a valid word at each go and then continuing, even if it is or is not leading to be built anywhere. Earlier we noted the total combination of words possible, which would be an absurd amount to traverse, keeping in mind that it will assume every prefix is going to be a word until it reaches the end of that long combination of search. The cleverness is most definitely that it retracts these search branches that are invalid, and only focuses on the potential prefixes for words.

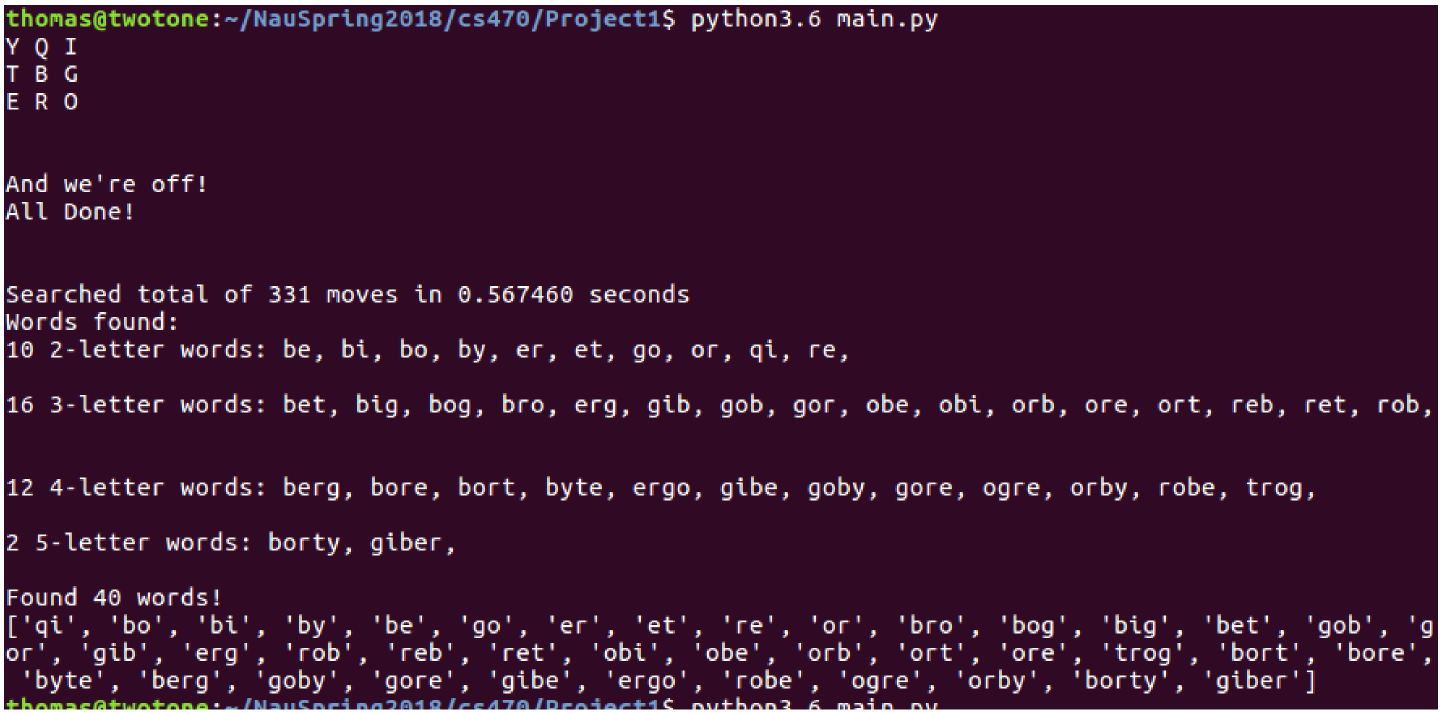
Board Outputs

2x2 board Testing Board Output



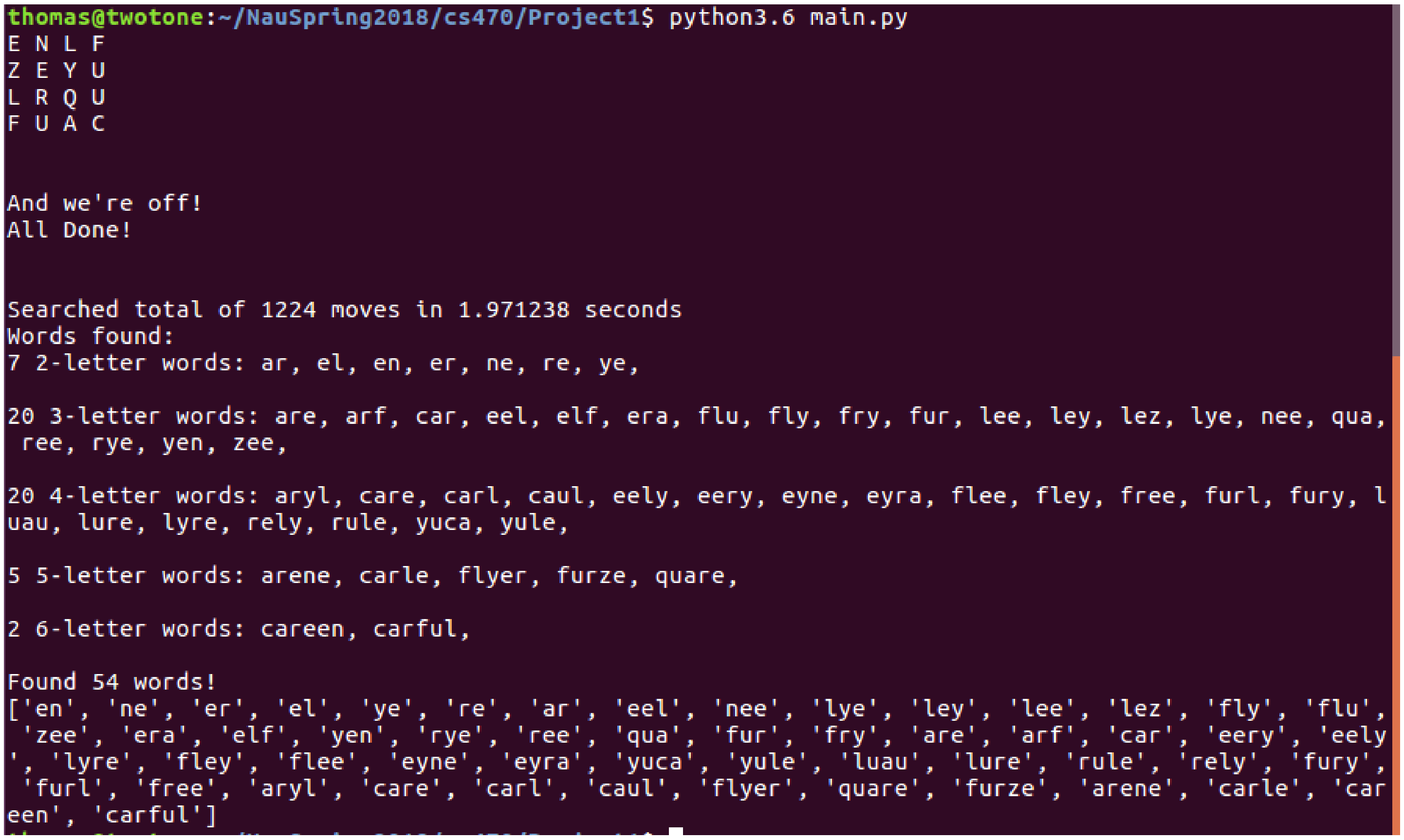
```
thomas@twotone:~/NauSpring2018/cs470/Project1$ python3.6 main.py
O T
W O


And we're off!
All Done!


Searched total of 52 moves in 0.131025 seconds
Words found:
3 2-letter words: ow, to, wo,

6 3-letter words: oot, too, tow, two, woo, wot,

Found 9 words!
['ow', 'to', 'wo', 'oot', 'too', 'tow', 'two', 'wot', 'woo']
```

3x3 Board Testing Board Output



```
thomas@twotone:~/NauSpring2018/cs470/Project1$ python3.6 main.py
Y Q I
T B G
E R O


And we're off!
All Done!


Searched total of 331 moves in 0.567460 seconds
Words found:
10 2-letter words: be, bi, bo, by, er, et, go, or, qi, re,

16 3-letter words: bet, big, bog, bro, erg, gib, gob, gor, obe, obi, orb, ore, ort, reb, ret, rob,


12 4-letter words: berg, bore, bort, byte, ergo, gibe, goby, gore, ogre, orby, robe, trog,

2 5-letter words: borty, giber,

Found 40 words!
['qi', 'bo', 'bi', 'by', 'be', 'go', 'er', 'et', 're', 'or', 'bro', 'bog', 'big', 'bet', 'gob', 'g
or', 'gib', 'erg', 'rob', 'reb', 'ret', 'obi', 'obe', 'orb', 'ort', 'ore', 'trog', 'bort', 'bore',
 'byte', 'berg', 'goby', 'gore', 'gibe', 'ergo', 'robe', 'ogre', 'orby', 'borty', 'giber']
thomas@twotone:~/NauSpring2018/cs470/Project1$ python3.6 main.py
```

4x4 board Testing Board Output



```
thomas@twotone:~/NauSpring2018/cs470/Project1$ python3.6 main.py
E N L F
Z E Y U
L R Q U
F U A C


And we're off!
All Done!


Searched total of 1224 moves in 1.971238 seconds
Words found:
7 2-letter words: ar, el, en, er, ne, re, ye,

20 3-letter words: are, arf, car, eel, elf, era, flu, fly, fry, fur, lee, ley, lez, lye, nee, qua,
 ree, rye, yen, zee,

20 4-letter words: aryl, care, carl, caul, eely, eery, eyne, eyra, flee, fley, free, furl, fury, l
uau, lure, lyre, rely, rule, yuca, yule,

5 5-letter words: arene, carle, flyer, furze, quare,

2 6-letter words: careen, carful,

Found 54 words!
['en', 'ne', 'er', 'el', 'ye', 're', 'ar', 'eel', 'nee', 'lye', 'ley', 'lee', 'lez', 'fly', 'flu',
 'zee', 'era', 'elf', 'yen', 'rye', 'ree', 'qua', 'fur', 'fry', 'are', 'arf', 'car', 'eery', 'eely
', 'lyre', 'fley', 'flee', 'eyne', 'eyra', 'yuca', 'yule', 'luau', 'lure', 'rule', 'rely', 'fury',
 'furl', 'free', 'aryl', 'care', 'carl', 'caul', 'flyer', 'quare', 'furze', 'arene', 'carle', 'car
een', 'carful']
```

Fully Commented Code

```
#### Boggle Solver Main ####

#### By Thomas Back     ####

#### 1/24/2018 CS470    ####


import time


class boggleSolver:


    def __init__(self, dic):

        self.board = []

        self.n = 0

        self.completeWords = []

        self.dic = dic

        self.moves = 0

        self.cleverness = False


    #loads NxN board into matrix

    def loadBoard(self, boardFile):

        #temp array

        temp = []

        count = 0


        board = open(boardFile)

        board = board.read()

        board.strip(' ')


        #for loop to loop through letters, disregards white space

        for letter in range(len(board)):


            #if reached new line, save that dimension within array
            if board[letter] == '\n':

                self.board.append(temp)

                temp = []

                if self.n == 0:

                    self.n = count -1

                count = 0


            #if blank space, skip

            elif board[letter] == " ":

                continue


            #append the letter to the temp array to be inserted as row into the board array

            else :

                temp.append(board[letter])

                count += 1




        return self.board


    def printBoard(self,board):


        #placeholder for a row of letters to be printed at once

        printLine = ""


        #with a 2d array we will have nested for loops to print from row and col

        for col in range(len(board)):


            for row in range(len(board[col])):

                printLine = printLine + board[col][row] + " "
            print(printLine)
            printLine = ""




    def withinBoundsCheck(self, Position):
        #Helper function that returns false if the positions are less than 0 or greater than
or equal to the max, N

        if Position[0] < 0 or Position[0] > self.n:
            return False

        if Position[1] < 0 or Position[1] > self.n:
            return False

        return True

    def possibleMoves(self, currPos):
        #generates all possible next positions, (x-y pairs in a list, set or whatver you
decide)
        #we could load currPos as a list of two elements, [0] always x, [1] always y
        #first check if the currpos is within the bounds of the board
        possMovesArr = []

        if not self.withinBoundsCheck(currPos):
            print("Error, current position is not within bounds\n")
            return -1

        #if within the bounds then move on
        currPos[0] += 1
        if self.withinBoundsCheck(currPos):
```

```
    possMovesArr.append(currPos[:])

currPos[1] += 1
if self.withinBoundsCheck(currPos):
    possMovesArr.append(currPos[:])

currPos[0] -= 1
if self.withinBoundsCheck(currPos):
    possMovesArr.append(currPos[:])

currPos[0] -= 1
if self.withinBoundsCheck(currPos):
    possMovesArr.append(currPos[:])

currPos[1] -= 1
if self.withinBoundsCheck(currPos):
    possMovesArr.append(currPos[:])

currPos[1] -= 1
if self.withinBoundsCheck(currPos):
    possMovesArr.append(currPos[:])

currPos[0] += 1
if self.withinBoundsCheck(currPos):
    possMovesArr.append(currPos[:])

currPos[0] += 1
if self.withinBoundsCheck(currPos):
    possMovesArr.append(currPos[:])

return possMovesArr
```

```
def legalMoves(self, possibleMoves, visited, word):

    valid = []

    for i in possibleMoves:

        if i not in visited:

            valid.append(i)


    return valid


def examineState(self, currPosition, path, word):


    #hold the number of moves

    self.moves += 1

    path.append(currPosition[:])


    #build the word with the new letter in the current position

    word += self.board[currPosition[0]][currPosition[1]]



    #compute new paths to begin search with first gaining possible moves

    possible = self.possibleMoves(currPosition[:])



    #now compute the legal moves in those direction

    legal = self.legalMoves(possible, path, word)


    #now compute the word that should be formed

    word = word.lower()
```

```
        if "\n" + word in self.dic:
            #check to see if the word is complete and then return after appending
            if ("\n" + word + "\n") in self.dic:
                if word not in self.completeWords:
                    self.completeWords.append(word[:])

            #recursive element to begin search in new path
            for next in legal:
                self.examineState(next[:], path[:], word)
        else:
            return



def main():
    begin = time.time()

    #open dictionary for reading
    dic = open("twl06.txt")
    dic = dic.read()

    #load board and begin the solution
    solve = boggleSolver(dic)
    myboard = solve.loadBoard('boardex')
    solve.printBoard(myboard)
    print("\n")

    solve.cleverness = True
    #begin with empty list
    print("And we're off!")
    for k in range(solve.n + 1):
        for j in range(solve.n + 1):
```

```
        solve.examineState([k,j], [], "")


end = time.time()

print("All Done!")

print("\n")


#save the total time

timeResult = end - begin

print("Searched total of %d moves in %f seconds" % (solve.moves, timeResult))

solve.completeWords.sort(key=len)

print("Words found: ")


#print array to loop through and hold words of same length

printArr = []

for words in range(len(solve.completeWords)):

    printArr.append(solve.completeWords[words])

    try:


        # if the length differs between indices, print and switch to next round of words

        if len(printArr[-1]) != len(solve.completeWords[words+1]):

            printArr.sort()

            print("%d %d-letter words: " % (len(printArr), len(printArr[-1])), end = "")

            for i in printArr:

                print("%s, " % (i), end =""),

            print("\n")

            printArr = []


    except IndexError:

        printArr.sort()

        print("%d %d-letter words: " % (len(printArr), len(printArr[-1])), end = "")

        for i in printArr:
                print("%s, " % (i), end =""),
                printArr = []

    print("\n")

    print("Found %d words!" % (len(solve.completeWords)))
    print(solve.completeWords)


if __name__ == "__main__":
    main()
```