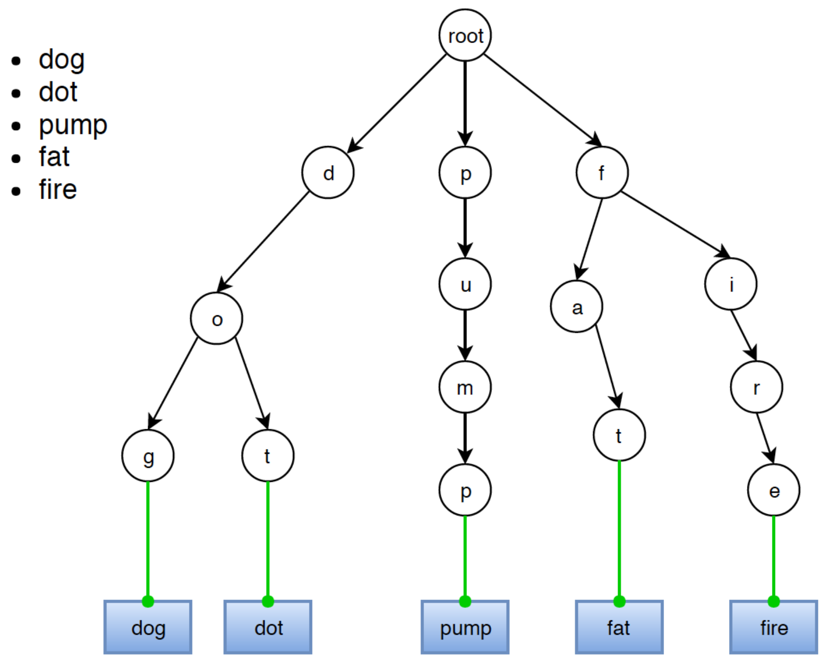
Tries: An Analysis

Created by Zach Arnold for CSE 331

What is a Trie?

A Trie (Pronounced “try” or “tree” depending on who you ask) or Prefix Tree is a data structure used to hold a library of strings. Tries store these strings in a hierarchal system similar to a binary tree, where each node represents a letter in the inserted words. Searching for words in a trie requires following pathways down the tree starting at the root node, making Tries incredibly useful in situations where speedy prefix or word lookups are necessary. Think of situations like your phone’s built in auto-complete feature, which can output predictions much faster using a trie than storing each word in a list, or even a hash map. In fact, the name “Trie” comes from being a substring of the word “retrieval”.



As shown, a trie that contains the words dog, dot, pump, fat, fire.
Source: https://jojozhuang.github.io/popular/data-structure/data-structure-trie/

There are many different ways to represent tries; some representations sacrificing speed for lower memory costs and vice-versa. The form of trie that I will be analyzing is an uncompressed, alphabetical prefix tree, meaning each node could contain up to 26 child pointers (one for each lowercase letter of the English alphabet). This is a very basic representation and will take up significant memory compared to other iterations. However, it will serve as a solid introduction to the unique capabilities and time complexities of the data structure.

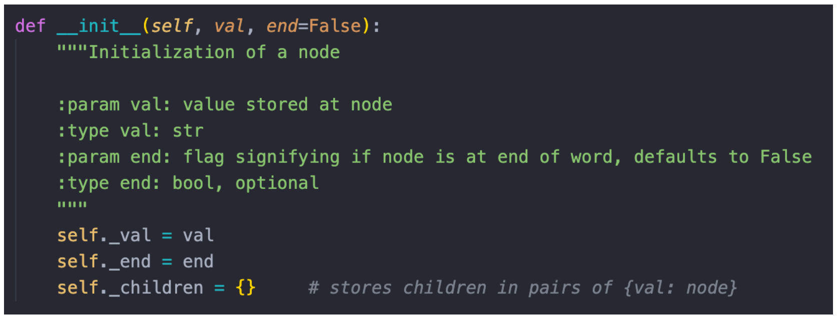
The Implementation

I built a fully functioning trie data structure in python, utilizing two classes; one for the trie's individual nodes, and one for the trie itself. Let's begin by analyzing the TrieNode class!

TrieNode

Implementing the trie's nodes was a very similar process to implementing the nodes of a binary tree. However, I added in a few optimizations and characteristics that differentiate TrieNodes from regular tree nodes. Here are the functions of TrieNode.

__init__



```python
def __init__(self, val, end=False):
    """Initialization of a node

    :param val: value stored at node
    :type val: str
    :param end: flag signifying if node is at end of word, defaults to False
    :type end: bool, optional
    """
    self._val = val
    self._end = end
    self._children = {}     # stores children in pairs of {val: node}
```

The __init__ method of the node class allocates memory for an instance. The TrieNode's attributes are self._val, which contains the value stored at that node ('a', 'b', etc.), self._end, a Boolean value that is True if the node is located at the end of a complete word, and self._children, a hash table that contains the node's children in key, value pairs. This hash table allows for guaranteed speedy lookups of children because of its hash function.

is_leaf

One-liner function that lets us know whether or not the node is a leaf node (has no children).
Time Complexity: O(1)

get_value

One-liner getter function that returns value of the node.
Time Complexity: O(1)

set_child

One-liner setter function that creates new child node and inserts it into self._children.
Time Complexity: O(1)

get_child

Getter function that returns the child node if it exists, otherwise returns None.
Time Complexity: O(1)

get_children
One-liner function that returns list of it’s children.
Time Complexity: O(C); C -> # of children node contains. (Could be O(1) considering in this version of a trie each node is guaranteed to have at most 26 children.)

delete_child
Removes node from children, raises KeyError if node doesn’t exist.
Time Complexity: O(1)

set_end
One-liner function that by default sets node’s end attribute to True, otherwise False.
Time Complexity: O(1)

get_end
One-liner function that returns node’s end attribute.
Time Complexity: O(1)

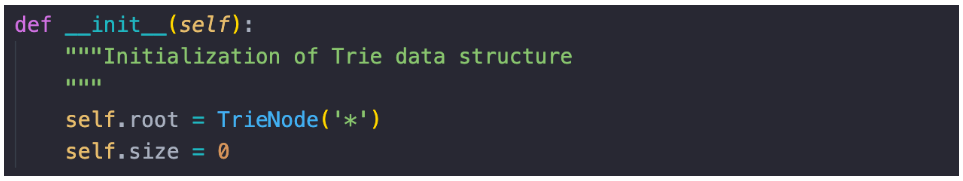
Phew! Those functions were a slog. Let’s get onto the more interesting ones within our Trie class.

Trie

The way I went about implementing a trie involved creating multiple member functions of different uses. I’ve separated these functions into two categories; mechanics and extensions. Mechanics are the necessary functionalities of the trie such as insert, search, and remove, and Extensions are the add-on functions that display the unique abilities of the Trie.

The Mechanics

__init__

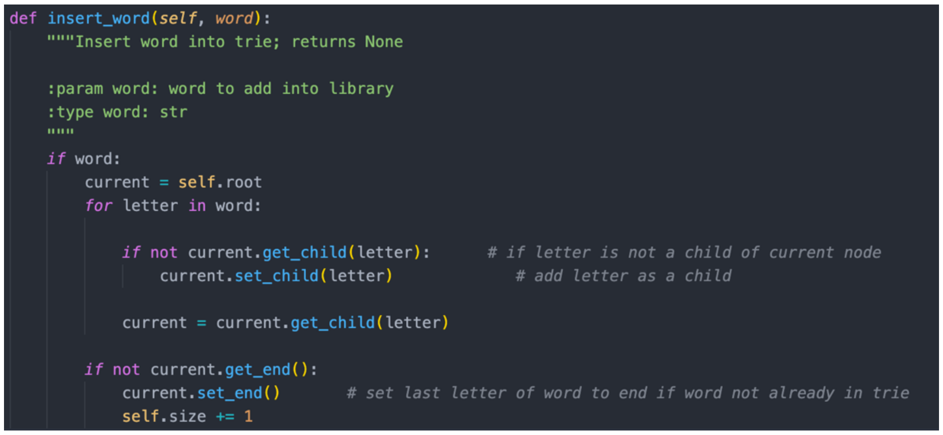


```python
def __init__(self):
    """Initialization of Trie data structure
    """
    self.root = TrieNode('*')
    self.size = 0
```

The __init__ method of the trie class allocates memory for an instance. The Trie’s attributes are self.root, a reference to an instance’s root node, and self.size, which keeps track of how many words have been entered into the trie.

is_empty
One-liner function that returns whether or not trie has 0 words.
Time Complexity: O(1)

insert_word

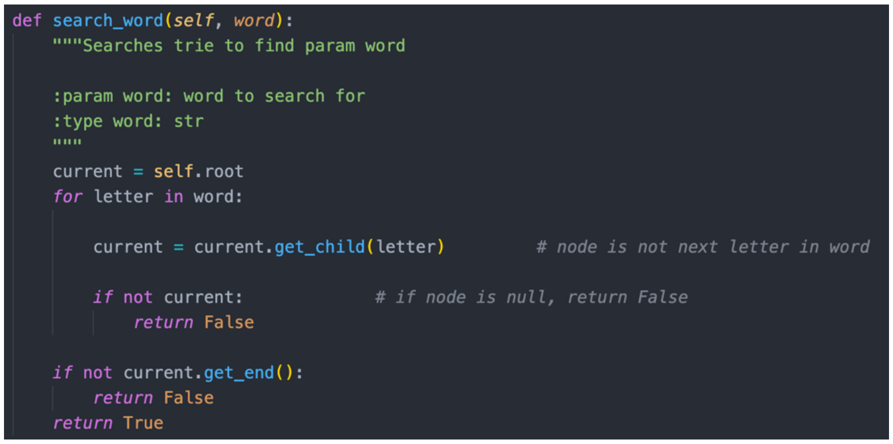


```python
def insert_word(self, word):
    """Insert word into trie; returns None

    :param word: word to add into library
    :type word: str
    """
    if word:
        current = self.root
        for letter in word:

            if not current.get_child(letter):      # if letter is not a child of current node
                current.set_child(letter)             # add letter as a child

            current = current.get_child(letter)

        if not current.get_end():
            current.set_end()       # set last letter of word to end if word not already in trie
            self.size += 1
```

Inserts word into trie. The function works by iteratively searching down the trie from the root node for the word until it can't find a particular character. If the word isn't found in full, it will begin creating new nodes down the trie for the rest of the characters until it reaches the end of the word. Once arrived at the last node, the status of that final node's end attribute tells us if we've already entered this word. If its end attribute is True, we've already entered this word. Otherwise, we'll set its end attribute to True and increment the trie's size attribute by 1.

Time Complexity: O(S); S -> # of letters in word

search_word



```python
def search_word(self, word):
    """Searches trie to find param word

    :param word: word to search for
    :type word: str
    """
    current = self.root
    for letter in word:

        current = current.get_child(letter)        # node is not next letter in word

        if not current:             # if node is null, return False
            return False

    if not current.get_end():
        return False
    return True
```

Searches for word in trie and returns whether or not the word is in the trie. The function words by iteratively searching down the trie from the root node for the word until it can't find a particular character. If a sequential letter isn't found in the trie, the function halts and returns False. Otherwise, the status of the last node's end attribute tells us whether or not the string we found is a complete word or not, similar to insert_word.

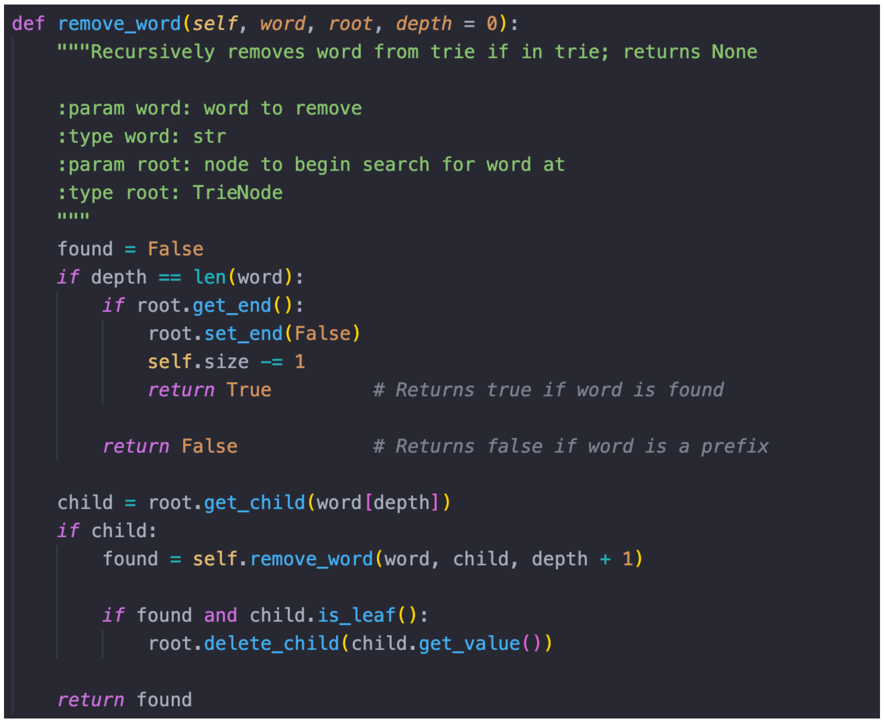
Time Complexity: O(S); S -> # of letters in word

search_prefix

Searches for prefix in trie. This function is identical to search_word aside from two characteristics that justify search_prefix's existence. For one, this function is not concerned with whether or not the prefix being searched is a full word. This unique functionality is useful in many Trie specific applications. This function also returns the last node of the prefix if found, for use in later functions. (Searching for an empty string will return root node).

Time Complexity: O(S); S -> # of letters in prefix

remove_word



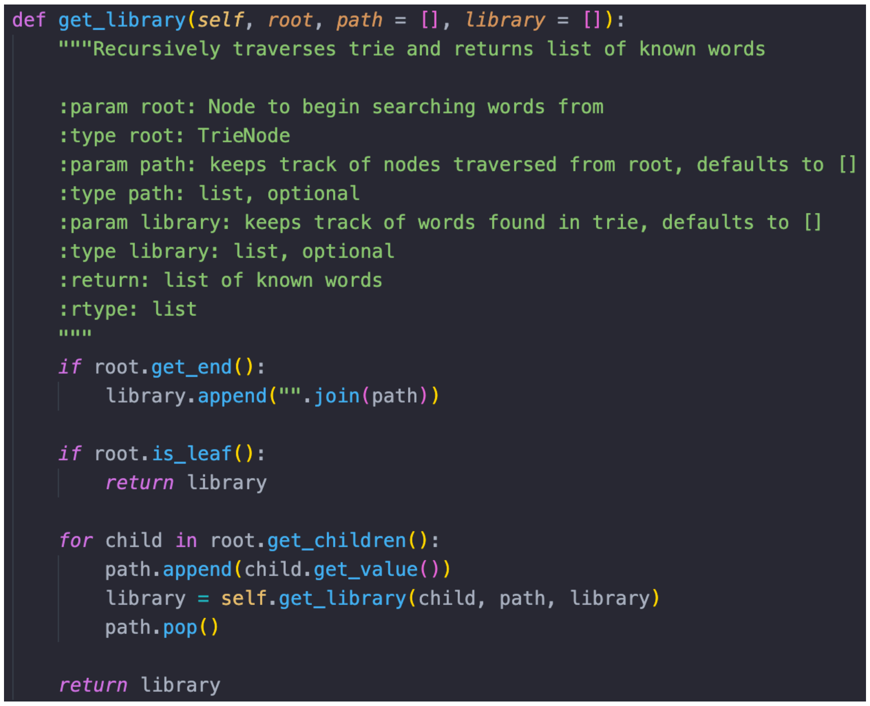
```python
def remove_word(self, word, root, depth = 0):
    """Recursively removes word from trie if in trie; returns None

    :param word: word to remove
    :type word: str
    :param root: node to begin search for word at
    :type root: TrieNode
    """
    found = False
    if depth == len(word):
        if root.get_end():
            root.set_end(False)
            self.size -= 1
            return True         # Returns true if word is found

        return False            # Returns false if word is a prefix

    child = root.get_child(word[depth])
    if child:
        found = self.remove_word(word, child, depth + 1)

        if found and child.is_leaf():
            root.delete_child(child.get_value())

    return found
```

Removes word from trie and returns whether or not removal was successful. The function first recursively makes its way down the trie from the root node searching for the word. If the word is found and is complete, it sets the final node's end attribute to false and removes each leaf node as it makes its way back up the call stack, also decrementing the trie's size attribute and returning True. Otherwise, it leaves the trie alone and returns False. The function also takes advantage of a parameter named 'depth' to avoid creating unnecessary copies of the parameter word during recursion.

Time Complexity: O(S); S -> # of letters in word

The Extensions

get_library



```python
def get_library(self, root, path = [], library = []):
    """Recursively traverses trie and returns list of known words

    :param root: Node to begin searching words from
    :type root: TrieNode
    :param path: keeps track of nodes traversed from root, defaults to []
    :type path: list, optional
    :param library: keeps track of words found in trie, defaults to []
    :type library: list, optional
    :return: list of known words
    :rtype: list
    """
    if root.get_end():
        library.append("".join(path))

    if root.is_leaf():
        return library

    for child in root.get_children():
        path.append(child.get_value())
        library = self.get_library(child, path, library)
        path.pop()

    return library
```

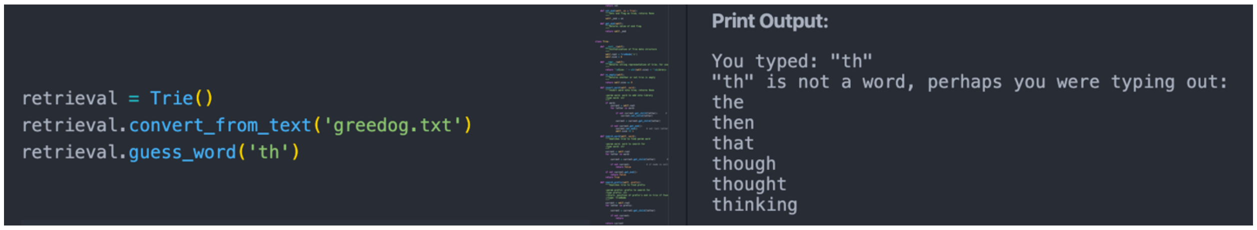
This function returns a list of words in the trie. It does this by performing a recursive depth-first search, visiting each node in the trie exactly once. Each time the function encounters a node with a True end attribute, it adds the full word into the Library list. At the end, the function returns the completed list of words. This function is particularly useful for having a readable visual of the words contained in the trie, so I built it early on and used it to debug while creating the other functions.

Time Complexity: O(L); L -> # of nodes in trie

convert_from_text

Adds contents of text file into trie. This function opens a specified text file and adds all of its words into trie. Because this instance of trie is meant to only contain alphabetic characters (aside from its root), this function strips punctuation from each word in the text file (Ex: ‘couldn’t’ -> ‘couldnt’). This function is useful for conveniently and quickly creating a large library of words. For example, I successfully created a trie containing every alphabetic word in the English dictionary (~370000 words) in roughly 4 seconds.

Time Complexity: O(N*S); N -> # of words in file, S -> avg. # of letters in word guess_word



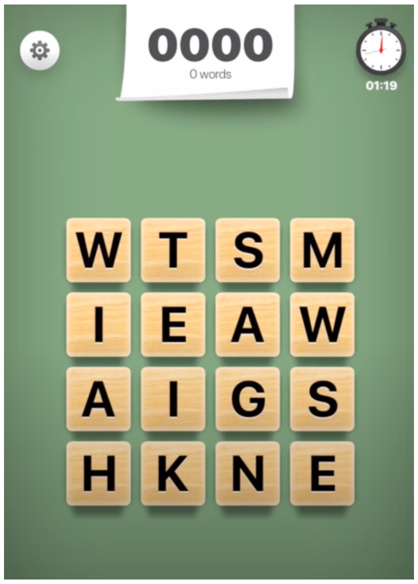
```
retrieval = Trie()
retrieval.convert_from_text('greedog.txt')
retrieval.guess_word('th')
```

Print Output:

```
You typed: "th"
"th" is not a word, perhaps you were typing out:
the
then
that
though
thought
thinking
```

This function prints output of suggestion words sharing a common prefix with the inputted parameter string. It's essentially a very basic auto-complete function. I tested this using a VSCode extension that I'm quite fond of called AREPL that updates python output in real time. This allowed me to simulate the process of an autocomplete system narrowing down guesses to determine the word it predicts you're typing. The function also returns a list of the printed suggestions, which will include the parameter word if it's in the trie.

Time Complexity: O(L); L -> # of nodes in trie

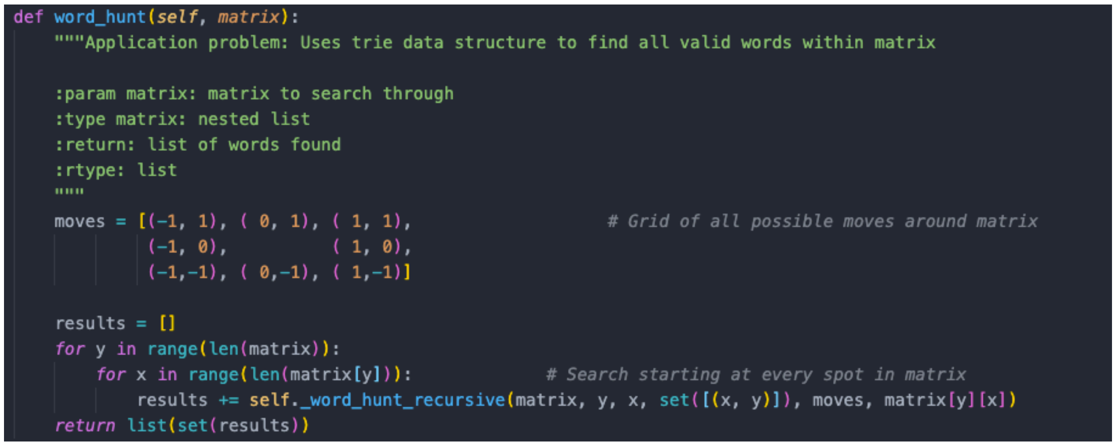


Application Problem

Have you ever played that dastardly iMessage game 'word hunt'? If not, perhaps you've played Boggle, the word game created in the early 1970s. Either way, this game has caused me a lot of headache. The game gives you a 4x4 - 6x6 grid of words, and tasks you with finding words within the grid. Words can be constructed from letters of sequentially adjacent spots in the grid. This could mean moving horizontally, vertically, or diagonally. Now if only there was a way to write a program to find the words for me…

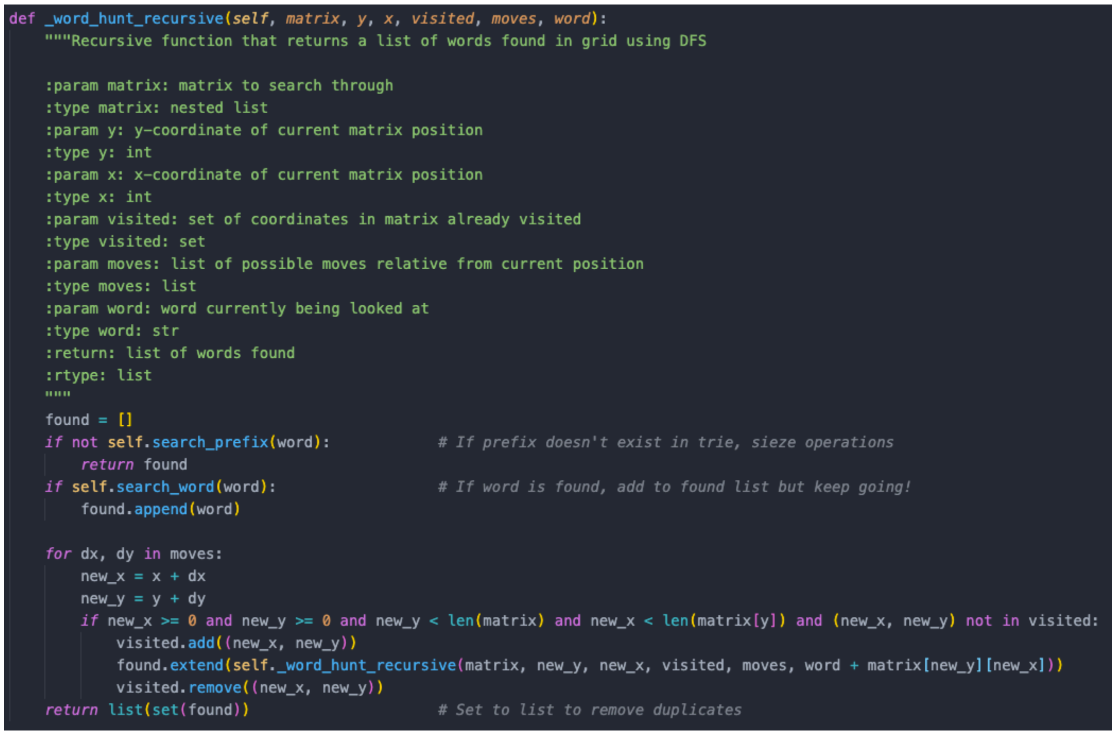
I thought up this problem while playing this game against my brother and seeing it as a great application for tries. Surprisingly (kind of) enough, this problem is actually a commonly used interview question for Software Engineers at Facebook! I approached this problem as if the grid was a graph, with the spots on the grid being vertices, and possible adjacent directions you could move from that spot being the edges. I would then perform a depth first search starting from each spot in the grid, stringing letters together and searching for words in my trie. The advantage of using a trie in this situation is that we have quick access to searching prefixes. For example, if we begin from the top left corner, and begin forming the word 'wts', it should be clear that no English word contains the prefix 'wts'. Therefore, we would not continue on from that spot, saving valuable time. Now on to the code!

word_hunt



```python
def word_hunt(self, matrix):
    """Application problem: Uses trie data structure to find all valid words within matrix

    :param matrix: matrix to search through
    :type matrix: nested list
    :return: list of words found
    :rtype: list
    """
    moves = [(-1, 1), ( 0, 1), ( 1, 1),                    # Grid of all possible moves around matrix
             (-1, 0),          ( 1, 0),
             (-1,-1), ( 0,-1), ( 1,-1)]

    results = []
    for y in range(len(matrix)):
        for x in range(len(matrix[y])):                # Search starting at every spot in matrix
            results += self._word_hunt_recursive(matrix, y, x, set([(x, y)]), moves, matrix[y][x])
    return list(set(results))
```

_word_hunt_recursive



```python
def _word_hunt_recursive(self, matrix, y, x, visited, moves, word):
    """Recursive function that returns a list of words found in grid using DFS

    :param matrix: matrix to search through
    :type matrix: nested list
    :param y: y-coordinate of current matrix position
    :type y: int
    :param x: x-coordinate of current matrix position
    :type x: int
    :param visited: set of coordinates in matrix already visited
    :type visited: set
    :param moves: list of possible moves relative from current position
    :type moves: list
    :param word: word currently being looked at
    :type word: str
    :return: list of words found
    :rtype: list
    """
    found = []
    if not self.search_prefix(word):            # If prefix doesn't exist in trie, sieze operations
        return found
    if self.search_word(word):                  # If word is found, add to found list but keep going!
        found.append(word)

    for dx, dy in moves:
        new_x = x + dx
        new_y = y + dy
        if new_x >= 0 and new_y >= 0 and new_y < len(matrix) and new_x < len(matrix[y]) and (new_x, new_y) not in visited:
            visited.add((new_x, new_y))
            found.extend(self._word_hunt_recursive(matrix, new_y, new_x, visited, moves, word + matrix[new_y][new_x]))
            visited.remove((new_x, new_y))
    return list(set(found))                     # Set to list to remove duplicates
```

The application problem contains two functions, a wrapper function and a recursive function. The wrapper function serves an abstraction of the entire problem, and returns the total found words in each of the locations on our matrix. The recursive function performs a depth first search on that spot of the grid, traversing through the matrix and linking letters together until the word being created is not a valid prefix, all while adding valid words to the list ‘found’. The recursive function also checks that each move is valid by calculating if the next position in the matrix is within its bounds and checking that we haven’t visited that spot before. The algorithm expects a matrix of at least 2x2 size.

Time Complexity: $O(N^2*8^{N*N})$; N-> Length of side of grid

I know what you're thinking. N^2*8^{N*N} is an atrocious time complexity! For a 5x5 matrix, $25*8^{25}$ is $9.45*10^{23}$! Allow me to explain. Big Oh represents an upper bound for our function. For this situation, it shows us where a worst-case-scenario might land us time wise. So, what's the worst-case scenario for a game a boggle? Well, a worst-case scenario represents a situation where our library is infinite. In other words, our library contains all combinations of alphabetic letters that are up to N^2 letters. I don't have to do the math to prove how unrealistic that would be! Because our function will immediately backtrack when an invalid prefix is found, more often than not, we end up skipping large portions of unnecessary searching. This worst case also doesn't take into consideration that not every position on the board has 8 adjacent unvisited spots. Corners and sides of the grid only have 3-5 adjacent spots, and with a smaller grid these spots represent the majority.

<u>Thanks for Reading!</u>