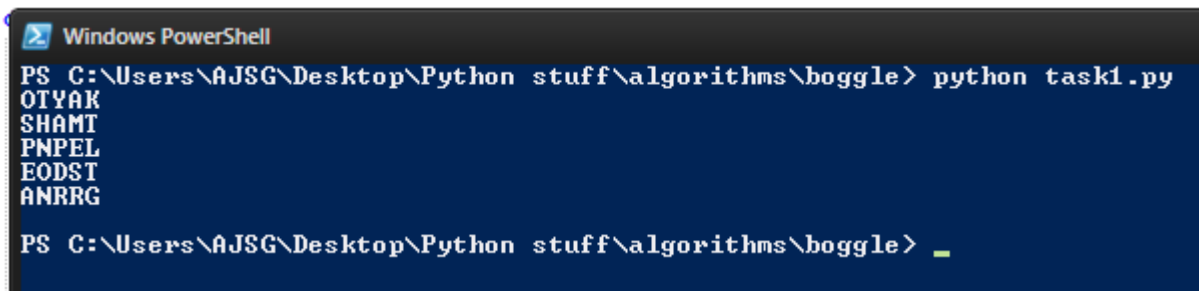


### 1.2.1 Task 1

The program 'task1.py' in the zip file randomly generates a legal 5x5 boggle board and prints it to stdout with a new line at the end of each row.

A screenshot of a Windows PowerShell terminal window. The title bar reads "Windows PowerShell". The command prompt shows the current directory as "C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle". The user has entered the command "python task1.py". The output of the script is a 5x5 boggle board, with each row on a new line: "OTYAK", "SHAMT", "PNPEL", "EODST", and "ANRRG". The prompt is now waiting for the next command.

```
PS C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle> python task1.py
OTYAK
SHAMT
PNPEL
EODST
ANRRG
PS C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle> _
```

The program imports functions defined in the BSBoggle file and needs to access dice.txt so it should be run from the same directory.

Separation of the dice data from the rest of the code was chosen so that users can modify the available dice set easily.

## 1.2.2 Task 2

### Algorithm:

My algorithm for finding a single word on a board (BSBoggle.py: 100-135) uses a helper function find() to initiate a search() for the given word starting from each of the squares in the grid.

From there, search() compares the letter at that coordinate to the letter at the head of the string it was passed.

If a match is not found, or the coordinate does not exist (ie we are falling off the board) that path is not an instance of the word we are looking for and the search falls back one step.

If a match IS found, the current path is the prefix of the word we are looking for, and search() recurses on the adjacent tiles, passing the tail of the word along.

If at any point in the recursion the passed-down string becomes empty, it indicates that we have found a path identical to the word we are looking for and the search returns successfully.

If find() completes, it indicates that no paths match our word and it is NOT on the board.

Search() also contains a clause to mark and de-mark which coordinates have been visited by the current call stack so that it avoids re-visiting coordinates which are already part of the string path.

NOTE: As per the specification the program treats instances of the letter 'Q' on the board as though they were 'QU'. This means that any word in the dictionary containing a 'Q' which is *not* followed by a 'U' *will not be detected on the board*.

### Instructions:

Required files: OneWord.py, BSBoggle.py, dictionary.txt

Optional: dice.txt

1. Run OneWord.py
2. Enter the string you wish to search for
3. Enter the configuration of the board you wish to search in (must be a square number of characters)

The program will output either “YES” or “NO” followed by a carriage return depending on whether the word is present on the board

4. A prompt will ask whether you want to run find() for all words in the dictionary
- 5a. Input of 'n' will cause the program to terminate
- 5b. Input of 'y' will iterate over the dictionary file and print all words which are present on the board.

```
Windows PowerShell
PS C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle> python OneWord.py
LASSO
EZSUEOSEFEAIAYOL
YES
Iterate through dictionary and print all words on board? y/n
y_
```

```
ZOH
ZOA EA
ZOE A
ZOE AE
ZOE AL
ZOE AS
ZOS
PS C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle> _
```

**Time/space complexity:**

Unfortunately I was not able to produce a  $\Theta(n)$  algorithm to determine whether a given string of length  $n$  was on any board. However the conditions under which the running time of the algorithm would approach worst case are extremely unlikely.

A for-loop (BSBoggle.py: 101) calls the recursive part of the algorithm for each letter on the board passing a string of length  $n$ .

This loops  $\sum_{i=0}^{d^2} 1 = d^2$  times where  $d$  is the width of the board.

The recursion (BSBoggle: 118) may call itself 8 times for the tail of the string (now size  $n-1$ ), once on each of the adjacent letters. Overhead on each recursion is constant so the complexity is:

$$T(n) = 8 * T(n - 1) + 1 = \sum_{j=0}^n 8^j \text{ assuming } T(1) = 1$$

Thus the overall formula is  $d^2 \sum_{j=0}^n 8^j$  but since the primary for loop is just a constant for a given board, the complexity is  $O(8^n)$  in the worst case.

In essence, if the word is not on the board but you are able to construct its prefixes in many ways, the complexity is exponential. Fortunately for the vast majority of cases where the degree of 'correct' letter adjacency is low, for example each letter of the string on the board has only one instance of the next letter adjacent to it, the complexity becomes linear:

$$T(n) = 1 * T(n - 1) + 1 = O(n)$$

Taking into account the fixed dice pool, board edges and 'no backtracking' rule, it is highly unlikely that the recursion will be called consistently for all of the surrounding letters. The total complexity for finding all words in the dictionary of  $m$  words iteratively is therefore on average  $O(a^n m)$  where  $a$  is the average number of correct next-letter adjacencies for each word in the dictionary.

The space complexity of this program for dictionary of  $m$  words of length  $n$  is just  $mn$  since loading the dictionary into an iterable array results in a data structure containing  $m$  elements of size  $n * \text{sizeofchar}$  (constant). The 2d array containing the board configuration can be treated as a constant and during recursion the callstack becomes at most  $n$  levels deep, with each level  $i=0$  to  $n$  storing a string of size  $n-(n-i)$ , resulting in a space complexity of  $\sum_{i=0}^n n-i = O(n^2)$ .

However since  $m$  is much greater than  $n$ ,  $mn > n^2$  and thus for any reasonable dictionary where you would expect  $m > n$  the space complexity is  $O(mn)$ .

### 1.2.3 Task 3

#### Instructions:

Required files: BSBoggle.py, dictionary.txt (program uses this as the default filename for the dictionary file however it accepts input for dictionaries with any filename)

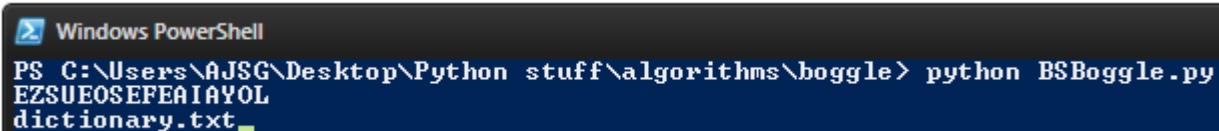
Optional: dice.txt

1. Run BSBoggle.py
- 2a. Enter the configuration of the board you wish to search in (must be a square number of characters).
- 2b. Alternatively, entering an integer  $n$  will generate a randomised board of dimensions  $n*n$  so long as there are enough dice in dice.txt. Entering a blank line defaults to a random 5x5 board. Requires dice.txt.
3. Enter the filename (including file extension) of the dictionary file. Defaults to dictionary.txt if a blank line is entered.

Program will output all words found on the board which are in the dictionary, followed by two lines displaying the total number of *unique* words and the sum of their points value.

NOTE: This implementation (and EfficientBoggle) assumes that you are not allowed to score with the same word more than once, so does not award points for constructing that word again even though it may occur on a different path on the board.

As in OneWord, all 'Q' tiles count as 'QU' for the purposes of string comparison so no words in the dictionary with lone 'Q's will be found.



```
Windows PowerShell
PS C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle> python BSBoggle.py
EZZSUEOSEFEAIAYOL
dictionary.txt_
```



```
LIASES
LO
LOESS
LOY
Words found: 144
Maximum score: 161
PS C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle> _
```

#### Worst case complexity analysis:

This algorithm is essentially the inverse of OneWord in that instead of taking a word from the dictionary and crawling the board to find a string that matches, it first crawls the board to construct a string and then tries to find it in the dictionary via binary search in  $O(\log m)$  time.

The danger here is that blind construction of strings would produce a vast number of permutations, roughly of the order  $d^2!$  where  $d$  is the dimension of the board. To limit recursive depth, I have implemented `binarySearch()` (BSBoggle: 207) to simultaneously check whether the string is a prefix of any word in the dictionary whilst it searches for a match, at an additional constant time cost within the function. This allows the recursion to fall back when it detects a string pattern which will never produce a word no matter what letters are appended to it.

As in task 2, the number of times the recursion is actually called depends on letter adjacencies (at worst 8 correct letters), and the depth of the recursion is limited at the length  $n$  of the longest word in the dictionary so in the worst case the algorithm will produce  $8^n$  strings (say a board of all 'A's where 'AAAAAAAAAAAAAAB' is the longest word in the dictionary). Each string is looked up in

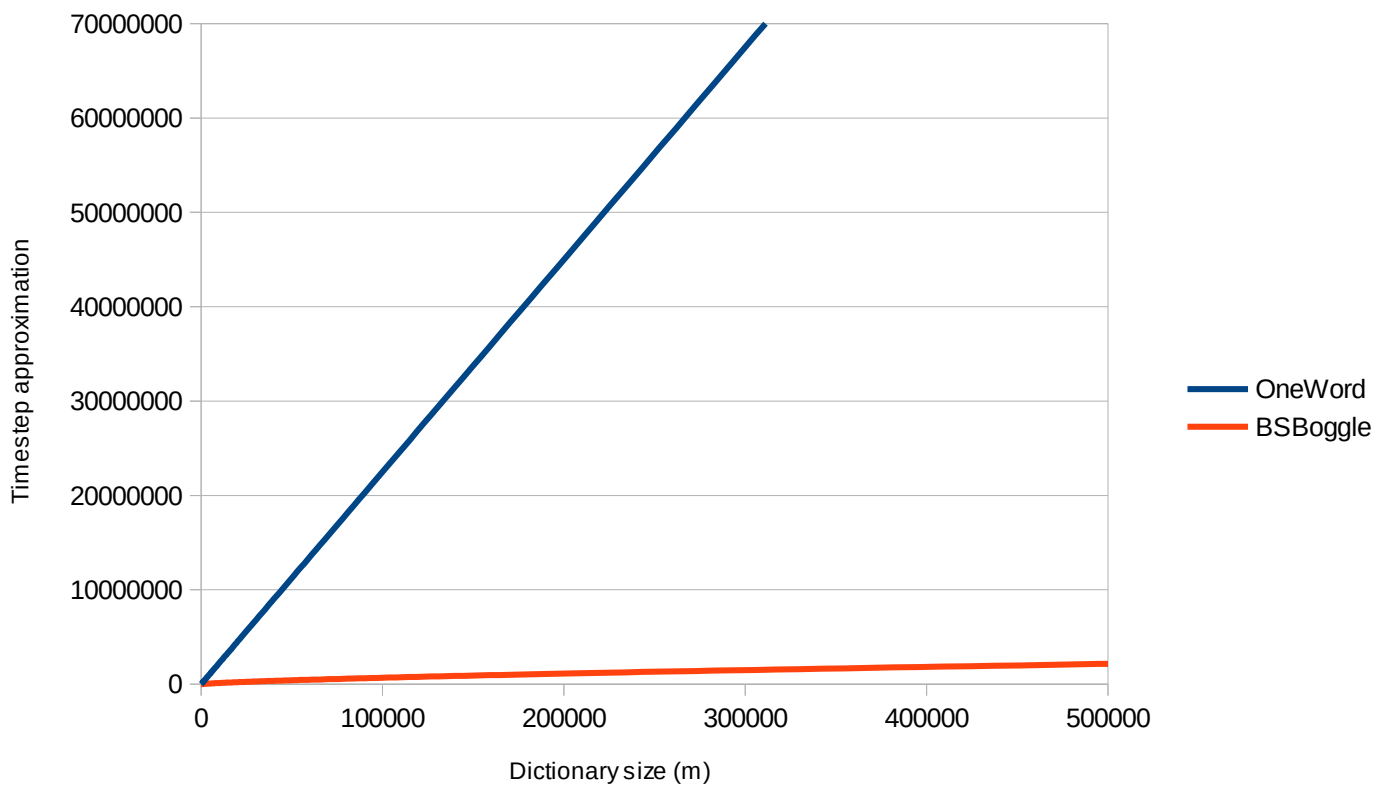
$O(\log m)$  time since we know binary search has this complexity for a list of size  $m$ , resulting in total worst case complexity of  $O(8^n \log m)$ .

NOTE: Making the same assumptions as before about adjacency and board layout, string construction can nonetheless be performed in much lower time under favourable conditions. Therefore we can consider the average case complexity as  $O(a^n \log m)$ .

### Empirical comparison to OneWord:

This comparison assumes that the length  $n$  of the average word in the dictionary is less than 10 and that average case complexity holds for the crawling algorithm on a 5x5 board.

m	1024	2048	4096	8192	16384	32768
OneWord	230400	460800	921600	1843200	3686400	7372800
BSBoggle	22941	38953	65679	110089	183584	304784



## 1.2.4 Task 4

### Data structure:

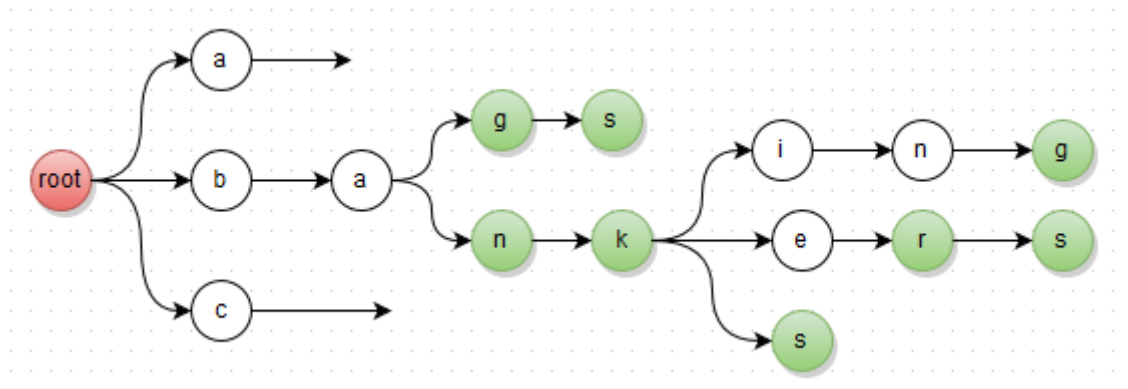
I chose to store the dictionary information in a simple prefix tree.

Paths in the tree correspond to the prefix of words, with each node representing a letter and its children the letters which could potentially follow it.

Nodes contain a boolean which denotes whether they are the last character of a word.

struct TreeNode:

```
[TreeNode pointer | Null * 26]    // array of pointers to up to 26 other nodes
bool isWord                      // true iff node marks the end of a word
```



### Algorithm:

EfficientBoggle: 156-199

Helper function `getValidStrings()` begins a traversal of the dictionary tree from the root (prefix tree constructed earlier) from each coordinate on the board, calling `buildString()` on each one.

From there, `buildString()` determines whether the letter is a child of the previous node and, if so, whether it finishes a word or not.

`BuildString` then recurses for each adjacent letter, passing down the appropriate node, unless there is no child for the letter it inspects (indicating that it is a fruitless prefix) in which case the recursion falls back.

This allows any generated string to be found in the dictionary in constant time, since the maximum depth of the tree is dictated by the longest word. Therefore, increasing the size of the dictionary will have no bearing on the complexity of finding a single word.

### Instructions:

Required files: `EfficientBoggle.py`, `dictionary.txt`

Optional: `dice.txt`

1. Run `EfficientBoggle.py`
- 2a. Enter the configuration of the board you wish to search in (must be a square number of characters).
- 2b. Alternatively, entering an integer  $n$  will generate a randomised board of dimensions  $n*n$  so long as there are enough dice in `dice.txt`. Entering a blank line defaults to a random 5x5 board.
3. Enter the filename (including file extension) of the dictionary file. Defaults to `dictionary.txt` if a blank line is entered.

The program will take several seconds to construct a prefix tree from the dictionary file before

printing all words which can be found on the board.

```
Windows PowerShell
PS C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle> python EfficientBoggle.py
EZSUEOSEFEAIAYOL
dictionary.txt
EE
ZEE
ZEES
ZO
LIAS
LIASES
LO
LOESS
LOY
Maximum score: 161
PS C:\Users\AJSG\Desktop\Python stuff\algorithms\boggle> _
```

### Analysis / Empirical comparison:

The strength of the prefix tree is that it allows lookup of a word in constant time. The tree is a maximum of  $k$  nodes deep where  $k$  is the size of the longest word in the dictionary, thus at most  $k$  nodes need to be traversed to determine whether a string is a word or not.

The board crawl works as it does in BSBoggle, however EfficientBoggle traverses the prefix tree in parallel to determine whether the letter forms a word, or the prefix of a word, in  $O(1)$  for each step.

However, assuming a uniform dictionary of size  $m$ , for any step into the recursion we eliminate 25/26 possible words, as we select only one out of the node's 26 children. This means the recursion has the form:

$T(m) = 8 * T(m/26) + \Theta(1)$  , which can be expanded to

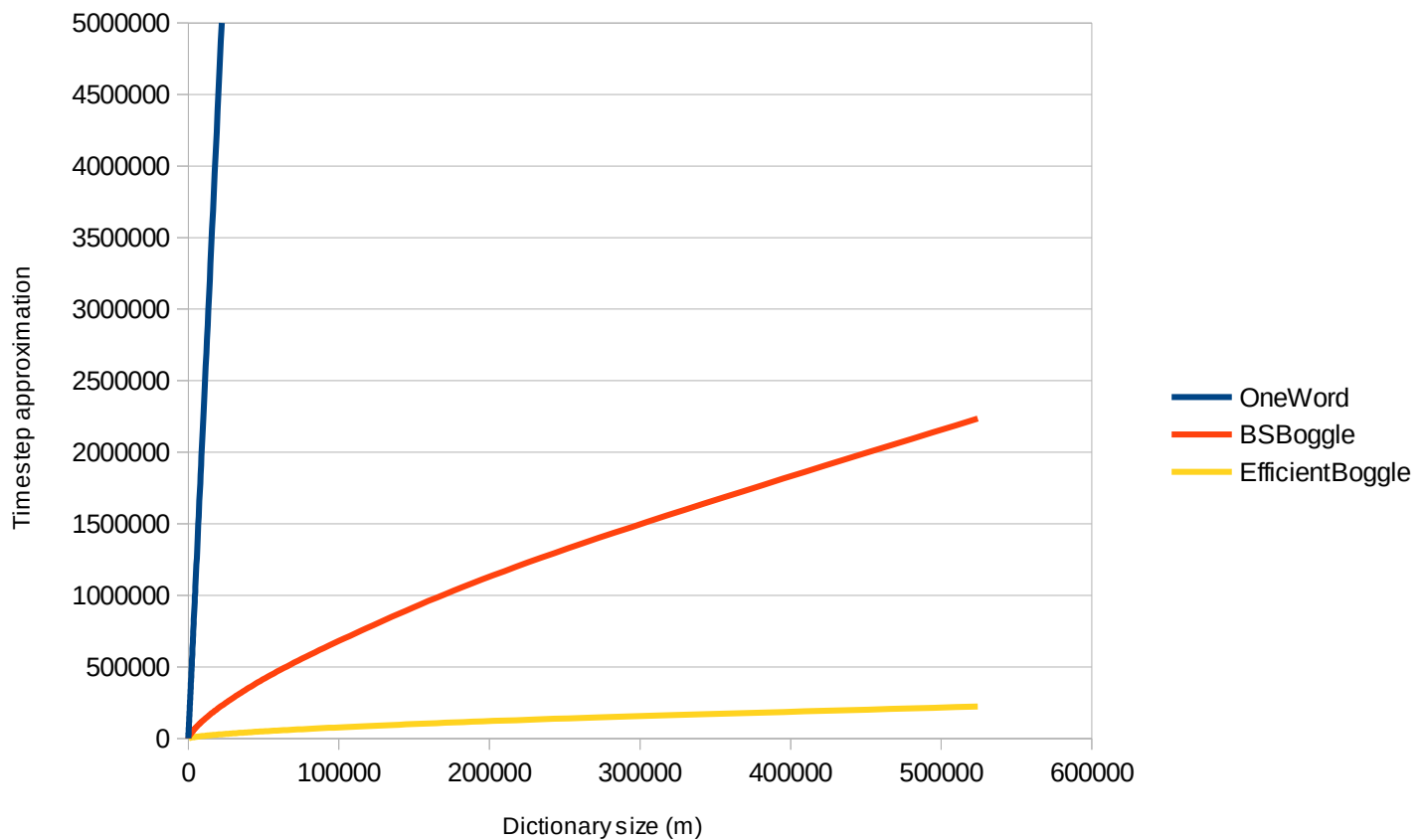
$T(m) = 1 + 8 + 64 + \dots + 8^{\log_{26} m} + \Theta(m^{\log_{26} 8})$  , which may be expressed

$$T(m) = \sum_{i=0}^{\log_{26} m} 8^i + \Theta(m^{\log_{26} 8}) + 1$$

For  $m > 512$  this function is dominated by  $8^{\log_{26} m}$  and so the overall complexity is  $O(8^{\log m})$ .

By comparison to the previous algorithms:

m	1024	2048	4096	8192	16384	32768
OneWord	230400	460800	921600	1843200	3686400	7372800
BSBoggle	22941	38953	65679	110089	183584	304784
EfficientBoggle	4171	6492	10105	15727	24478	38098

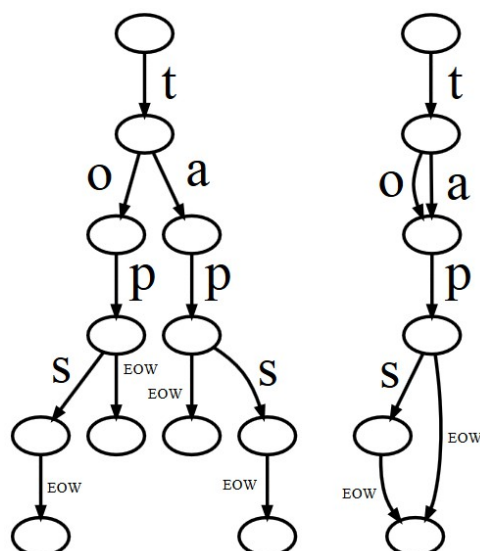


### Improvements to memory efficiency:

The space complexity of the simple prefix tree I implemented is technically in worst case  $O(nm)$  as it contains less than or equal to the sum of the characters of all the words in the dictionary (we may encounter a dictionary in which no words share a prefix).

However this is fairly misleading since the overhead for each node is significantly greater than that of a single character in memory; each node must be large enough to store pointers to 26 other nodes. Additionally, towards the bottom of the tree nodes will tend to have fewer children.

A similar but more space efficient data structure is a *directed acyclic word graph*. This data structure eliminates suffix redundancy by allowing a node to be reached from more than one path.



<http://upload.wikimedia.org/wikipedia/commons/9/9c/Trie-vs-minimal-acyclic-fa.svg>