## Task 2: Boggle Buddy

The game of Boggle is normally played on a 4x4 grid. 16 dice, each die printed with six characters from the letters A to Z, are placed in a closed container and the container is shaken. The dice land in the 4x4 grid and the letter on the top face is the letter for that grid square. A three-minute timer is typically started and players try and make as many words from the letters on the grid as they can before time runs out. These words have to follow certain rules - for this task we will assume the rules followed are:

* Each letter (except for the first letter) must be adjacent (horizontally, vertically or diagonally) to its preceding letter.
* A grid square's letter can only be used once (if there are two copies of the same letter appearing in different grid squares - both can be used).
* The word must be one of the allowed words.

Though there are additional rules when played as a group of players, but points are assigned for the total number of letters in all valid words made.

## Part A (Code)

A regular board-game playing group have invited some tourists to play and brought Boggle. The tourists wanted to try playing but were not confident in their foreign language skills and were worried the words they'd make might not be valid for play. The group asked online if anyone would be willing to put together a system to quickly retrieve all valid words so that the tourists can quickly check the list to see that they did not make a mistake.

## Part B (Code)

Hooked on the board game, the tourists purchased a copy and brought it home as a souvenir. Their children wanted to play the game with their parents, but because they are still practicing the language, they sometimes get stuck trying to find a word. Seeing a good opportunity for their children to practice a foreign language, the parents reached out again to see if anyone could put together a tool that gives a hint for the next letter that can be played to make a word.

## Part C (Written)

The parents notice the program often takes time to think of solutions, but only on boggle games where the same letter appears lots of times. Given the sand timer for thinking of words gives such a short time, they ask if there's a way to speed it up, for example, if each letter can only be used once in each word (even if it appears more than once on the board). How might their rule modification change the worst case complexity of finding all words?

## Part D (Code)

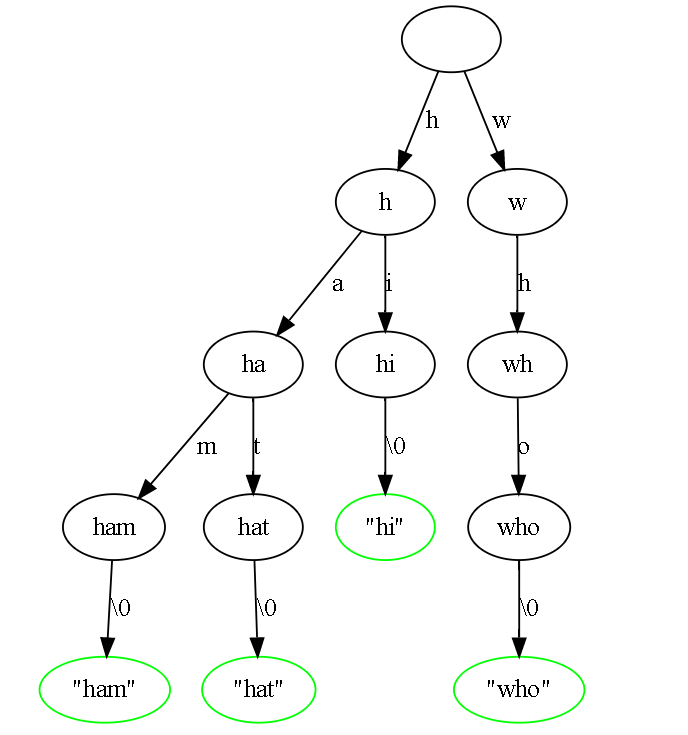
Hearing that the improvement might be promising, they ask you to put together the tool giving all valid words with the added rule.

## Task 2: Prefix Tries and Boggle Graphs

## Background - Prefix Tries

For Task 2, you will need to build a prefix trie. This is a data structure where, rather than containing a full key in each node, each edge in the trie represents a particular letter. Since each edge is associated with a character, both checking whether a given string is in the tree and finding *all* strings that match a given prefix are efficient operations.

For example, to store the strings "ham", "hat", "hi" and "who", our prefix tree might look like:



For simplicity, the tree will:

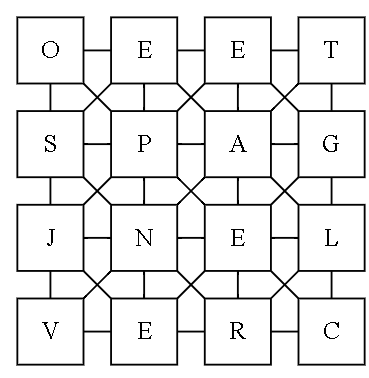
Contain a pointer for each possible following character - even if that character is unlikely to appear.

Also store the \0 character used as the delimiter when storing the string in the tree.

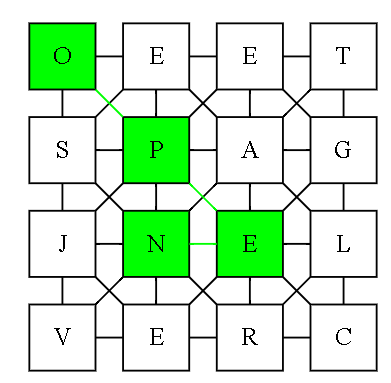
This means every node will contain 256 pointers - the majority of most of which are typically set to NULL. This also means each leaf of the tree will correspond to the completion of an inserted string - with completion of strings occurring nowhere else in the tree.

## Background - Boggle Graphs

For a given set of dice, a graph can be constructed showing the connections.



This graph can then be used to work out what are valid words to play. For example, the word "open" could be played, starting from O:



## Part A (Code)

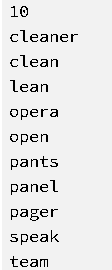
To support checking you will search simultaneously through the prefix tree and the boggle graph. Returning the list of all words starting from each position. You must use the prefix tree to limit the available locations to travel to next. You will likely find it useful to temporarily mark seen words as you travel through the prefix tree (and then unmark these at the end of the search).

Part A will take two filenames at the command line:

The first filename is the name of the dictionary of words which are allowed for that game.

The second filename is the name of the board used for boggle.

The format of the file with the first given filename will be similar to this example:



Where all files follow the format:

The first line specifies the number of words in the dictionary (10 in this example)

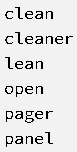
All following lines specify words in the dictionary.

The format of the file with the second given file name will be similar to this example:



Where the board this file represents matches the example board above. The format of the file will always be 16 capital letters, arranged in a 4-by-4 grid. When determining whether letters on the board can be used to make words, you must ignore capitalisation.

The output must be the list of words (alphabetically ordered (breaking ties by length)) that can be made on the board which follow the boggle rules stated earlier. For the given example this would be:



## Part B (Code)

In Part B, the file inputs are the same, but an additional input is given on the command line through stdin. This is the word constructed so far - you must output the list of letters following the given input that can follow on the boggle board as a hint.

For example, for the same input filenames as in Part A,

|  |  |
| --- | --- |
| if the input **pa** were given through stdin, the output printed to stdout must be:  output: | If the word can be terminated, a blank space ( ) character should also be printed on its own line. For example, if the input clean were given through stdin, the output printed to stdout must be:  Output: |

## Part B Notes

Where the same letter is present as the following letter (e.g. if cle were given), it must be output only once.

Any non-letter character must be printed as a space.

If the stem given is not a prefix in the dictionary, you must output nothing to stdout (e.g. if opa were given, stdout must not be written to, as no words in the dictionary match that starting stem).

If the word cannot be made on the board, its next letter must not be printed (e.g. if spe were given, a should not be output, as speak cannot be made on the boggle board).

## Part C (Written)

In Part C, you must create a pdf format document called written-tasks.pdf, which explains the impact of only allowing each letter to appear once in each word (regardless of how many times the letter appears on the board). Your answer must state an upper-bound on the time complexity reflecting the impact of this change, with each term used explained clearly.

In order to avoid trivial answers, you must assume the board could be extended arbitrarily to higher dimensions (e.g. 5x5 and beyond) and that the alphabet used could increase in size (e.g. the maximum length of a word is not 26 letters).

## Part D (Code)

In Part D, you must put together code that takes input in the same format as Part A, but which takes advantage of allowing each letter to appear only once in the word.

For the inputs given in Part A, the output for Part D would be:

