Programming in Python for Scientists CSCI 2202
Project 1b Due: 15 April 2021
Project: Finding Anagrams Recursively

The Problem:

An anagram is the word obtained by permuting (rearranging) the letters of a given word. *e.g.* `‘trace’` can give `‘crate’`, `’react’`, `‘cater’`. For a word of length n, there are $n!$ (n *factorial*) candidate permutations. Only a few of these are valid words. The validity of the words can be checked against a list of words, which will be provided. This list of words is read-in as the lexicon to check if the candidate anagram against. The word list must be read-in and stored as list. Initialize the list right after you import statements, outside any function.

The Idea:

- To start tackling the problem pick the 1^{st} letter.
- Do this systematically: you will need to do this for every letter, but start at the 1^{st} .
- For each choice of 1^{st} letter, try all possible orderings of the remaining letters.
- You will use recursion to handle this rearrangement.

For Example:

- If using `trace` and the 1^{st} letter chosen is `c` (from `trace`).
- The rest of the letters are the *letters to use are:* `trae`.
- When rearranging these letters, `aret, ater rate` are meaningful as they result in: `caret, cater, crate`, which are in the word-list.

Method:

To solve the problem, create a function `anagrams`. This function accepts three parameters:

(1) The list of valid words (*i.e.* the `word-list`).
(2) The **letters** to use (`trae` above).
(3) The prefix (`c` above).

For example, an initial call to `anagrams` is: `anagrams(word-list, ’trace’, ’ ’)`. There are five choices for the 1^{st} letter, each of those choices generates a recursive call, like: `anagrams(word-list, ’race’, ’t’)`. Each of the recursive calls generates and returns a solution list, which could be empty. The overall solution is the combination (concatenation) of all these individual lists. To approach this problem:

(1) Create a list for holding any solutions.
(2) Loop over the characters in `letters`, and for each character, append it to the end of the current `prefix` to create a new one.

(3) Remove the extracted character (which was added to the prefix) from `letters` to create a new string of letters.
(4) Use the `newLetters` and `newPrefix` to make a recursive call to `anagrams`. Add the result to the solutions list created earlier.
(5) When there is a *single* letter left in `letters`, append the letter to the *prefix* and then search the `word-list` for complete candidate word. This constitutes the *base case*.
(6) If the candidate word is in the `word-list`, append the candidate word to the solutions list and return the solutions.

You will need a function to read-in the `word-list` and a function to `search` for the candidate word. When a valid solutions list is returned, write it to an `output` file.

Run your program using `‘crate’`, using `small_words_5.txt` as your word list (a file containing 5-letter words) and then run `petals` against `small_words_6.txt` (a file containing 6-letter words) as your word list. Find the running time of your program, for each of the words.

To time your program, `import time` and use `start = time.time()` before the first function call and `stop = time.time()` after you have finished writing the (valid) anagrams to a file. The running time is (approximately) `stop - start`.