CSC108H Week 11 Lab

To earn your lab marks, you must actively participate in the lab. *You do not need to finish in the time allotted, you just need to arrive on time and try hard.*

As usual, pick a partner and do pair programming (take turns driving and navigating).

1 Objectives

- Begin thinking about the efficiency of algorithms
- Consider multiple algorithms to solve the same problem

2 Anagrams

Two words are anagrams if they have the same letters, ignoring the order of those letters. For example, **horse** and **shore** are anagrams because we can "reorder" the letters of one of the words to arrive at the other one. With your partner, determine which of the following pairs of words are anagramatic. Show your TA your work.

- bill, lib
- deposit, topside
- march, charm
- sport, torts
- traffic, traffic

We will explore and compare two techniques for generating all anagrams of a given word. We will focus on "real" English words; even though **horse** and **seohr** are anagrams, we are not concerned with such cases that use words not existing in an English vocabulary.

3 Permutation Approach

Our first approach will involve generating all possible permutations of a word, then testing each permutation against a vocabulary of words.

By way of example, imagine we want to generate all anagrams of the word **bore**. The word **bore** has four letters, and so has $4! = 4 * 3 * 2 * 1 = 24$ permutations of its letters. Those permutations are as follows: bore, boer, broe, breo, beor, bero, obre, ober, orbe, oreb, oebr, oerb, rboe, rbeo, robe, roeb, rebo, reob, ebor, ebro, eobr, eorb, erbo, erob. Given a suitable list of words (i.e. an English vocabulary), we could search that word list for each of these 24 permutations to determine which are legal words. In the word list we will use, we will see that only **robe** is an anagram of **bore**.

1. Download the file `generate_perms.py`. It contains a function, `generate_perms`, which, given a string, returns a list consisting of all of its permutations. Do not concern yourself with how this function works (though if I have piqued your interest, we can certainly discuss it!). Test this function on the string `'bore'` before continuing.
2. Download the file `find_anagrams.py`. We will implement the functions in this file.

3. Implement the `perm_find_anagrams` function, whose signature and docstring are given. Start by generating all permutations of `anagram`. Then, for each permutation, check whether it exists in `word_lst`; if it does, add it to your running list of anagrams.

4. The `perm_find_anagrams` function requires a list of words. While you could use a small, "hand-coded" list to test your function, it will be more convincing if we use an actual English vocabulary. Download the file `dict.txt`; it is a vocabulary of about 20000 English words.

5. Implement the function `all_words`. Test it by opening `dict.txt` in Python and passing the open file object to `all_words`. The function should return a list with each word as a separate component, with no newlines. (The list of words is not small. You probably don't want to print this entire list!)

6. The list returned by `all_words` is suitable as the `word_lst` component of `find_anagrams`. Pass the English word list and various words to `perm_find_anagrams`. Test with the following words, which increase in length: `bore`, `horse`, `arrest`, `discern`, and `drawback`. Finally, test with `microphotographic`. Uh oh. How long does this take? (Hint: don't let it finish. You'll be here all year! And, yes, it does have an anagram!)

4 Signature Approach

Our second approach will involve generating the "signature" of the word for which we want to find anagrams. We then compare this signature to the signature of each word in the dictionary. When two words have the same signatures, they will be anagramatic.

Let's go back to our **bore** example. If we arrange the letters of **bore** in alphabetical order, we get **beor**. This is the signature of **bore**. If we take any word in the vocabulary and find its signature by similarly sorting its letters, it will be an anagram of **bore** only if its signature is also **beor**. (For example, notice that the signature of **robe** is also **beor**.)

1. For practice, what is the signature of **stop**? What is the signature of **microphotographic**? (Do you hate that word yet?)

2. Implement the `signature` function, whose docstring is given in `find_anagrams.py`. Test your function using several words whose signatures you already know.

3. Implement the `sig_find_anagrams` function. You should generate the signature of `anagram` just once, prior to looping through `word_lst`. When looping through `word_lst`, compute the signature of each word to determine whether it is an anagram of `anagram`.

4. Test your `sig_find_anagrams` with our English vocabulary, and several words such as `bore` and `horse`. And, yes, test it with `microphotographic`! Fast, eh?

5 Comparing the Algorithms

We now have two different algorithms that solve the same problem. The first generates all possible permutations and then "prunes" the list based on our supplied vocabulary. The second calculates the signature of a word, then looks through the vocabulary for words with the same signature. The algorithms can exhibit drastically different performance, as we have seen. Please discuss the below questions with your partner and TA. To help, you should download `time_algs.py`, which includes some code for timing each algorithm.

1. The permutation method is fast for short words and absolutely horrendously slow for large words. Why is this the case? Experiment with different-length words to get an idea of what happens with different lengths.

2. The signature method is fast, regardless of whether we try to find anagrams of **bore** or **microphotographic**. Why does the length of the word not matter for this algorithm?

3. Are there cases where the permutation method outperforms the signature method?

4. Which algorithm is "better"? Why?