159.271 Computational Thinking for Problem Solving

# Tutorial 1: Mixing It Up

To get full marks for this tutorial you will need correct code with comments specifying preconditions, loop invariants, and post-conditions.

*Anagrams* are rearrangements of the letters of a word that create another word (or sometimes, set of words, but we'll ignore that and just look for single words). They are common in crosswords and similar word-based puzzles. For example, 'post' is an anagram of 'tops' and stop'.

The problem that you should solve is very simple to describe: create a program that, given a dictionary of words, prints all the anagram classes, that is all the sets of words that are anagrams of each other, in decreasing order of set size.

Here is an example of what the output could look like (it won't look exactly like this, especially if some of these words aren't in the dictionary that you use as the input.)

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['resmelts', 'smelters', 'termless']
['retainers', 'ternaries']
['generating', 'greatening']
```

Thinking about this problem it might seem at first that you can just start with the first word in a dictionary ('aardvark', maybe) and then create all of its anagrams, then look through the dictionary to see if they exist (which is relatively easy since dictionaries are usually in alphabetical order) and so on. *What is the computational complexity of this method?*

Another solution is to compare each pair of words to see if they match. *What is the complexity of this method?*

To improve on these algorithms there is one key insight that you need, and that is that since we are rearranging the letters anyway, we can do it first. Think of hashing, which you saw in 159172: if we compute some sort of signature of a string then, if it based on the letters, but not their order, it will give us the anagram classes. One such signature is just to sort the string into alphabetical order. You can then build a dictionary to store the words in their classes, using these signatures as keys. So there are three parts to the algorithm: (1) read in the words and compute the signature of each one by sorting the letters, (2) build a dictionary to store the words as you go, (3) manipulate the dictionary to print out the anagram classes in decreasing order of size. Design an algorithm to do this, and then implement it in Python.

For testing, UNIX computers have a dictionary built in, in **`/usr/dict/words`**. I have included a version of this file on Stream. It contains 234,936 words, so your algorithm has to be efficient. Use the **`time`** function from the **`time`** module to find the running time of your program on this dictionary. Fastest (correct) demonstrated program will win an exciting assortment of lollies from the Ashhurst dairy, which I drive past daily!