You will write some “counter distance” code and unit-tests for it while other group members independently implement some I/O code and a counter data structure. The sample solution is 40-45 lines, not including testing code or the header file. (Though the shortest of the 3 assignments, the testing is probably the most difficult because you do not have the counter data structure.)

Requirements:

- Put your code in two files, 5c.c and 5c_test.c. Both should include 5c.h, which you should write. 5c.h needs just these prototypes plus typical header-file stuff:

  ```c
  struct WordCounter;
  typedef struct WordCounter * word_counter_t;

  int how_many(word_counter_t counter, char* word);
  int longest_word(word_counter_t counter);
  int does_longer_exist(word_counter_t counter, char * word);
  ```

- In 5c.c, you will implement the function `average_distance` (described below), using helper functions you write and helper functions declared in 5c.h. You should not implement the counter data structure or the helper functions declared in 5c.h. Your testing code (5c_test.c) will have to provide “stubs” (fake implementations) for the declarations in 5c.h.

- A counter is a data-structure that for any word (any sequence of lower-case English letters) reports a non-negative number – to get the number, call `how_many` with the counter and the word (plus a trailing ’\0’ so `how_many` knows the word’s length). A counter can report the length of its longest word with a non-zero number (`longest_word`). Finally, it can take a word (with a trailing ’\0’) and report true (1) if it has any words that start with the given word, are strictly longer, and have a non-zero number (`does_longer_exist`).

- Given two “counters” `c1` and `c2` we calculate the distance between them as follows. Let `sum` be a variable (of type `double` since it might get big) initialized to 0.0.
  
  - For every word `w` in `c1` with a non-zero number `n`, get the number `m` for `w` in `c2` and add the square of the difference between `m` and `n` to `sum`.
  - For every word `w` in `c2` with a non-zero number `n`, if the number for `w` in `c1` is 0, then add the square of `n` to `sum`.

  The distance is then the square root of `sum`.

  Note this is the “Euclidean distance” where we have one dimension for every word (i.e., a very high-dimensional space). Note also the definition is symmetric (the distance from `c1` to `c2` equals the distance from `c2` to `c1`).

- `average_distance` should match this prototype:

  ```c
  double average_distance(word_counter_t c, int len, word_counter_t * arr);
  ```

  The third argument points to an array holding `len` counters. Return the average distance of `c` to these counters. See the next page for how to break the problem down into helper functions. See especially how to avoid generating every possible word.

- In 5c_test.c put unit tests for your code and a `main` that runs them.
Advice:

- Understand the algorithm before you start coding.
- To compute average_distance, use a helper function distance that takes two counters and computes their distance. Sum the results and divide by the number of counters in the array.
- Computing the two components of sum is so similar that it’s easiest to write a helper function that takes a flag (a boolean argument) indicating whether to add the sum for all words or only for words whose number in the second counter is zero. For example:

  double sum_one_direction(word_counter_t from, word_counter_t to, int only_to_zero); // the flag

- You can use longest_word to determine the size of an array large enough to hold any word you will pass to how_many. Reuse the array rather than allocating a new one for every word.
- For the core of the algorithm, you need to consider every possible sequence of lower-case English letters up to the longest possible length in one of the counters. However, this is too inefficient (if there’s a 10-letter word, this would be $26^{10}$ which is over 100 trillion). Therefore, you must use does_longer_exist to avoid trying most letter sequences. Read on...
- For the core of the algorithm, you will want to use recursion. (If you fight this advice, you will regret it!) Use a function like this:

  double sum_prefix(word_counter_t from, word_counter_t to, int only_to_zero, char * buf, int i);

  The caller ensures buf[0], ..., buf[i-1] are already set to some prefix and the rest of buf (which is large enough for any word in from) holds '\0'. The callee takes care of every longer word that starts with buf[0], ..., buf[i-1] returning the sum of their sums. To do so, it uses a loop to:

  - Set buf[i] to each lower-case letter and compute the sum for the resulting word.
  - If from has longer words starting with buf[0], ... buf[i], then recur with i+1 for i and add in all the results. Remember after the recursive call to set buf[i+1] back to '\0'.

  Note the initial call to sum_prefix uses 0 for i, which means compute the sum (in one direction) for all words with length greater than 0.
- For your loop, you may assume the lower-case English letters have numeric values that are consecutive and in order (so you start with 'a' and increment until you get through 'z').
- To use the sqrt function in the math library, include math.h and compile with -lm.

Assessment and turn-in:

Your solutions should be:

- Correct C code that compiles without warnings using gcc -Wall and does not have space leaks
- In good style, including indentation and line breaks
- Of reasonable size

Your test code should provide good coverage.

Use turnin for course cse303 and project hw5. If you use late-days, use project hw5late1 (for 1 late day) or hw5late2 (for 2) instead of hw4.