Scrabble-Indexer time(1) results:

The scrabble-indexer part of our program runs in an average of 2.77 real seconds, 4.30 user seconds, and .415 system seconds. That is, from a stopwatch perspective, it took an average 2.77 seconds. The fact that it took 4.3 user seconds implies that at least part of the program is multithreaded. Since I didn’t write any multithreaded code, odds are that it was either the Sort that I used from Java Collections (which is more than likely), and/or some sort of compiler optimization voodoo that parallelized my existing code\*. All of the threads individually took less than the 2.77 real seconds to run, but cumulatively took up an average of 4.3 seconds of cpu time.

The .415 system seconds are time that the program spent in the kernel doing system calls.

These timed tests are of my final version of the indexer program. The first version deduplicated the words being written to the index files in O(n) time, instead of the current program’s O(1) time (I was able to achieve this constant time deduplication, because the word lists were already sorted by score and alphabetical order, so any duplicate words would be next to each other). That version, instead of taking an average of 2.77 seconds to run, took approximately 2.5 minutes to run. I’m happy with my 2.77 seconds.

\*I did some adhoc timing inside of my program, getting a “ real start time” benchmark by assigning the value of System.currentTimeMillis() to a member variable during the constructor. I found that the parsing of the initial file and scoring of the words contained in that file by score was done in ~0.4 seconds after the constructor was initially called, sorting those words was done by ~ 0.6 seconds, and that the whole process was done ~1.25 seconds after the constructor was called. Given an overall (real) run time of ~2.77 seconds, this means that the program spent ~ 1.52 seconds prior to running any of the actual program logic I wrote.

Of the ~1.25 seconds spent after entering into the meat of the program:

* ~0.4 seconds were spent reading in words from the file and computing their scrabble scores;
* ~0.2 seconds were spent sorting the ~100k words gathered from the file,
* ~0.5 seconds were spent populating the contains-letter hash table, and
* ~0.15 seconds were spent deduplicating the contains-letter lists in the hash table and writing them to their respective index files.

I postulate that most of the multi-threaded behavior happened in the sorting stage, and was facilitated by the library calls. I say this, because the writing of the words in the list to various files happened in near-linear time, and the sort would be at best, O(nlogn). Given that, when n = ~100k, nlogn is ~11.5 times larger than n, we’d expect the sorting to take at least 11 times more than the write at the end. The sort actually takes around ~0.2 real seconds, which is considerably less than 11.5 times the ~0.15 the write-to-disk step took. I’d say that, since 11.5 times 0.15 is around 1.7, it’s reasonable to say that the sorting should have taken ~1.7 seconds in user time.

Add that 1.7 seconds of user time to:

* the ~1.52 seconds of pre-constructor real runtime
* the ~0.4 seconds of reading in and scoring words
* the ~0.5 seconds of populating the contains-letter hash tables
* the ~0.15 seconds of deduplicating and writing words to their respective contains-letter index files.

And you get around ~4.27 seconds of user run time, which is within .03 seconds of the actual average user run time.

Suggester Analysis:

I put together a series of test cases for the scrabble-suggester:

(0) QUERY = "" K = 5, testing the program's BASE run time using invalid arguments (this test case only prints the USAGE message).

(1) QUERY = "et" and K = 10, the provided example test case in the spec

(2) QUERY = "embezzlements" K = 10, testing the program's reaction to being asked for the top ten occurrences of a query that will only be found once

(3) QUERY = "jen" K = 15, testing the program's reaction to being asked for the top 15 occurrences of a query that occurs fewer than 15 times in the dictionary

(5) QUERY= "an" K = 300, testing a common string and asking for a large (K) # of occurrences.

(6) QUERY = "zygl" K= 5, testing for the top 5 occurrences of a query that has no occurrences

(7) QUERY = "e" K=76170, testing for the worst: query consists of the most common letter, and K is equal to two more than the total number of words that contain that letter in the dictionary.

Test (0) is used to establish a base runtime--that is, how fast the program runs if no actual program logic is executed, and it only spits out a usage message before exiting.

Test (1) is used as a "normal" case, to establish how long a non-edge-case would take to run.

Tests (2), (3),and to an extent (7), are used to test how the program runs when asked to find more occurrences of a string than actually exist.

Test (5) is used to test how the program behaves when given a moderate load, where K is indeed less than the total number of occurrences of QUERY.

Test (6) is used to test how the program behaves when given a QUERY that doesn't exist in the source word file at all.

Test (7) is used to test for the worst possible case available given the provided dictionary and the program structure. Since e is the letter most commonly found in the dictionary's words, the contains-letter index file for E is the largest file, with 76168 lines/words. By picking 'e' as our query, we guarantee that all lines in e's contains-letter index file will match the query. By picking a K that is two larger than the number of words in that file, we test for index-out-of-bounds type errors, and we make sure we go through the largest possible index file, line-by-line, until we hit the end.

Results:

TEST O: