Puzzle for software engineer, levenshtein distance:

Getting some information about the data:

```
$ curl --compressed --silent http://github.com/causes/puzzles/raw/master/word_friends/word.list | tee words.csv | wc -l
264061

$ file words.csv
words.csv: ASCII text

$ grep -c "[^a-z]" words.csv
0
```

- Corpus size: 264061 words
- Words are all pure ascii without character outside a-z

Analyzing the problem:

The naive approach

Calculating the levenshtein distance of a word against the corpus is gonna be O(N), which means a brute force approach will have a quadratic complexity in the worst case !N levenshtein distance O(N^2)

The reverse approach (Query Expansion style)

Instead of calculating the levenshtein distance of a word compared to the corpus lets determine all the variation of a word and check whether they are in the corpus.

In this particular case the CHARSET is pretty limited (26 chars only)

- A word of n letters have maximum of neighbors: n (char suppression) + 25 * n (char substitution) + 26 * n+1 (char addition) = 52n+ 26 For n = 4 => 234, n = 8 => 442

As size of words is limited so the query expansion itself this is a O(1) operation, if we consider that checking whether a word belongs to the dictionary is a O(1) operation (Hash lookup) then a "Query Expansion" approach will be O(N) in the worst case

The precalculated approach

If the purpose of the exercise was to develop a service to calculate the social network of a word in a fixed dictionary, precalculating all the levenshtein distance of all the word relative to one another would be the good approach. This operation is O(N^2) but only once, after that this is a simple lookup in the graph database to find the social network.

Choosing an approach to the solution:

- The first approach is more flexible, it does not depend of the charset but tend to scale very badly
- The precalulated approach is very good if we would intend to calculate social network for a lot of different word, but just for one word it's overkill and the slowest approach as calculated the data will take much longer than just doing it for the word.
- The query expansion approach is the more promising for this particular problem, its performance depends on the size of the Charset but it scales in O(N). This approach will probably consume more memory than the others approach as a lot of string will be generated, and hash structure will have to be used to provide O(1) query against the corpus. This approach present the interesting characteristic to be easily Map/Reduced (the brute force approach too by the way).

Recommended implementation

For this kind of task a functional language with efficient mutable string and tail recursion will offer the best performance.

Given the ASCII nature of the problem pure C would be a no brainer. But any LISP, Haskel or Scheme will probably offer good results. I used ruby for the implementation because this is a ruby work application, it offers mutable string and has a decent object initialize performance.

Implementation:

From a memory and performance perspective the code have to be efficient and should not garbage collect uselessly or create unnecessary object. Ruby as a language is far from being the perfect choice for this kind of low level string manipulation and memory consuming task:

- Ruby 1.8 -> 1.9 -> 2.0 migration make low level string manipulation a maintenance issue. This version should be 1.8.7+ compliant
- The language expressivity or Object Oriented focus are not particularly useful here as anyway the algorithm and optimization will make it look like some C code while being one order of magnitude less quick.
- What is interesting in this implementation, is that it is mostly ruby limited in term of performance: Array.each 35%, Kernel.dup 25%, Array.select 11%, Hash.key? 8%. You can't do much to optimize it.
- This is a bit quicker in ruby 1.9.2 but not much, ree offer same result than MRI as there is no complex memory pattern.
- For a desktop demo this is quite ok, but as a webservice this is too slow for live use (about 50 sec on my laptop). A preprocessing approach would be more cost efficient.
- This is interesting to note that the performance hypothesis on the solution were ok as a micro benchmark over several iteration of a 300k element array is one order of magnitude slower

Some interesting caching pattern could be used with the query expansion approach. E.g caching

intermediary calculation like not reprocessing "ca*ses" or "c*uses" substring search that could occur for several different word. But empirical results shows that the lack of cache hit (less than 10%) but the big memory consumption (and thus garbage collection) it needs makes it a no gain approach. Almost all gain are made but avoid creating unnecessary instances. The version provided is probably not far from the quickest performance in pure ruby.

To try it:

```
$ ./count_sibblings words.csv causes
```