This attached code was for a code challenge where was requested to found anagrams given the MD5 digests.

In this file I will briefly explain the algorithm, make some considerations and possible improvements.

For more details the code has been commented.

The purpose of the algorithm is to compute all possible sentences with the given anagram words, get their md5s and test them against the given ones. Since the complexity of this computation quickly become unfeasible, this search has been done finding the anagram sentences with ‘patterns’ of growing number of words, starting from 1 (max word length in vocabulary is 23). At 3 words patterns the easy and medium phrases were founded. At 4 came out the hard one.

The algorithm in brief will:

* Take a source anagram and various options, build a Trie with the English dictionary given in the challenge page.
* Compute all the possible patterns of phrases for the given number of letters: First find the combinations and then permute them (using the same lexical algorithm without repetitions needed later).
* For each pattern use the lexical sorted, white space free, source anagram and calculate its lexical permutations. Thanks to this ordering it is possible to make important simplifications. For each word forming the pattern left to right, check if the current permutation contains a valid word using a Trie. If not, restart the permutation from the last valid word index found in the Trie. Doing so it is not necessary to explore all the following lexical permutations that will not result in acceptable sentences of this pattern. If a match is found for each word in the pattern, compute the MD5 and check it with the target ones.

Example:

Actual permutation for pattern [7, 7,…]: correctwordwax…

‘correct’ fits, wordwax no, last match found in Trie is -> word -> restart permutation from there and obtain in lexical order next element ‘correctwordxaw…’ (assuming ‘x‘ is the next greater word after ‘w’ on the right)…and so on.

The program was set to explore all the possibilities for patterns of at most 4 words and without any further simplification, complete its total run in about 25 minutes.

These results were obtained on a computer with an AMD Ryzen 5 2600X Six-Core Processor, 3600 Mhz. To bring the CPU usage at maximum I made use of a number of 10 threads that will take each a different pattern to explore.

For all the algorithms of lexical permutations, combinations, as for the Trie I have used some results found on the internet and credited them in the source code (which in some case I slightly modified for my purposes). I used only iterative solutions and tried to have an eye for performance.

Being N the number of letters, time complexity is roughly given by the main lexical permutations for each pattern, which would be in the worst case N! multiplied for the number of patterns (compositions) which are 2^(N-1).

In practice each pattern will be reduced excluding as previously said edge cases while the lexical permutation will skip all the subsequent outcomes that don’t follow valid vocabulary words. It will then be a function of the words length in the pattern and not of solely N.

Space complexity is a minor concern since an iterative approach has been used without storing additional data per iteration. Approximatively each iteration requires constant space plus ‘static’ resources like the dictionary Trie. Threads also raise the memory requirement in a constant fashion.

In the actual code there are some rules used to reduce the number of patterns that may be applied or added. For example, limiting the number of small words in the pattern (1 or 2 letters) or not considering single words other than vocals. These assumptions will obviously limit the possibilities of potential correct sentences (ex: a a k k k in k k worlds).

I tried to run this code with 5 words for the sake of curiosity adding some filtering rules but it resulted as expected in a very long computation time.

Possible improvements:

* Code structure, performance improvement
* More tests
* Trying new algorithms
* Check if changing pattern calculator algorithm from 2 steps to 1 might benefit.
* Memoization or caching approaches may benefit at the expense of memory
* Think out of the box: another approach? Starting from a different mathematical observation.

Instructions:

* Maven is needed for dependencies (Guava)
* The dictionary has to be placed in the main project directory
* A file with the matches will be generated

Best Regards,

Gabriele Ceccarelli