hw2

Due Jan 27, 2016 by 11:59am **Points** 10 **Submitting** a file upload **File Types** zip
Available after Jan 21, 2016 at 12am

Due: Noon on Wednesday, Jan. 27

You should complete this homework individually. We will begin working in pairs for HW3.

For this homework, you will design and implement a statistical spelling correction program, correct, based on an idea of Peter Norvig's (http://norvig.com/spell-correct.html) . The objective is to deepen your knowledge of Rust and its module system, and to start thinking about performance.

The concept

The purpose of correct is to find possible corrections for misspelled words. It consists of two phases: The first phase is a training module, which consumes a corpus of correctly spelled words and counts the number of occurrences of each word. The second phase uses the results of the first to check individual words. Specifically, it checks whether each word is spelled correctly according to the training module and, if not, whether "small edits" can reach a variant that is correctly spelled.

Given a word, an edit action is one of the following:

- the deletion of one letter;
- the transposition of two neighboring letters;
- the replacement of one letter with another letter; and
- the insertion of a letter at any position.

In this context, Norvig suggests that "small edits" means the application of one edit action possibly followed by the application of a second one to the result of the first.

Once the second part has generated all possible candidate for a potentially misspelled word, it picks the most frequently used one from the training corpus. If none of the candidates is a correct word, correct reports a failure.

Implementation details

The correct program consumes a training file on the command line and then reads words—one per line—from standard input. For each word from standard in, correct prints one line. The line consists of just the word if it is spelled correctly. If the word is not correctly spelled, correct prints the word and the best improvement or "-" if there aren't any improvements found.

For example:

```
$ cat corpus.txt

hello world hello word hello world

$ cat input.txt

hello

hell

word

wordl

wor

wo

w
```

```
$ cargo run corpus.txt < input.txt

hello

hell, hello

word

wordl, world

wor, world

wo, word

w, -
```

Performance

The combinatorics of this problem, considered naïvely, may be prohibitive. In particular, consider a *k*-letter word. The number of possible letter replacements is *kd*, where *d* is the size of the alphabet. And because we are applying two edits, there are *kd* words one step further away. Ignoring redundancy, this suggests that we have to check $\sim k^2d^2$, and that's not considering the other three kinds of edits. Thus, actually generating each edited word and then checking it in a hash table will result in poor performance.

So, is there a better way? (Yes, much better.) But how? I'd suggest considering whether there's a data structure other than a hash table that would let you prune your search tree more eagerly. Feel free to discuss below.

Evaluation

Your grade will be based on:

- correctness (how closely your program adheres to its specification),
- style (not expecting the most idiomatic Rust at this point, but I'll be looking for good factoring—don't put everything in main), and
- efficiency (no need to benchmark or profile, but do choose sensible data structures and avoid needless copying).

You can get full credit without finding the fast algorithm if the other aspects of your program are excellent.

How to submit

Please submit a tar or zip archive of your Cargo project directory here on Canvas. Name your archive *NETID*-hw1.{zip,tgz} (*e.g.,* jat489-hw1.tgz).