CSCI 311 - Fall 2020:
Programming Project: Autocorrect

Edward Talmage

Due: **On gradescope and in zoom demo** by Friday, Nov. 20, 2020
Based in part on problems in CLRS and other sources

Instructions

This is a group project. Work with your *LAST TWO BIT* square in the hypercube. That is, the three other students whose IDs are the same as yours except for one or both of the last two bits. You will collectively submit one version of your solutions. Be certain that the student who uploads your solutions then adds all group members to the submission on gradescope.

1 Overview

Your task is to implement a spellcheck/autocorrect system. This is, of course, very open-ended, as Google and others are continually trying to improve their algorithms. While you're welcome to create as grandiose a solution as you like, please start with a few basic features, as detailed below, and then expand if you have time and inclination.

You may use any language you want, as long as I can run it without jumping through too many hoops. Ideally something I can run on the command line would be best, as that allows me to run it over ssh while working from home if needed. Java, python, and the C family would all be fine.

- Interface: Start with a simple interface that accepts a string or text file and replaces detected spelling errors, returning a string or printing back to file. If you like, you can later implement a more interactive interface, where a user can type and have their text corrected directly.
- Algorithms: I suggest you start with the Longest-Common Subsequence problem we covered in the dynamic programming unit. Running LCS on an input word and a word in your dictionary will give you a similarity score. In general, if the word is in the dictionary, then you will get LCS length equal to the length of the word. You will need to think about how to choose words from the dictionary to check, as checking every input word against dictionary word may be VERY slow.
- Data Structures: You'll need to think about the best way to store the dictionary, and how to load it into that structure. This will depend on your choice of what dictionary words you want to compare your input against. Suggestions include a flat array, a hash table, a search tree, an encoding tree, etc.
  - You will need to use a wordlist of some sort. I've attached one simplified from Wiktionary (exact link in wordlist.txt) which should be a decent starting place.
  - If you want to look for a larger or more comprehensive wordlist, `https://www.academicvocabulary.info/` is another option. (I don't endorse them, or know much about them, but they're a site that purports to have some freely-available wordlists.)
- Heuristics: LCS will not detect everything, or give you a single answer, so you will need to make some choices about other features. I suggest adding a few rules for common errors, such as swapping adjacent letters in a word, mirroring (pressing a key on a keyboard with the correct finger of the wrong hand, such as having a "j" instead of an "f"), adjacent letters on the keyboard, etc. You'll also need to decide how to choose if LCS finds multiple candidate replacements with equal weight.

2 Deliverables

- You will need to submit all source code, along with a plain-text README describing how I can compile, run, and interact with your code. You may add features to your code after the demo, but make sure they're **clearly** documented.

- You will need to schedule a time for a short (about 10-15 minutes) demo on zoom before the end of classes. You'll briefly walk me through the algorithms you used and show how your code behaves.
- You will submit a short (try to keep it no more than 1-2 pages) description of the algorithmic structure of your project. Include asymptotic runtimes, though you don't need complete analyses, just enough detail to convince me that your analysis is accurate.

3 Workflow

I suggest you start with coding your data structure, Longest Common Subsequence detector, and interface. You may organize your team as you wish, but it may make sense to designate one member to each of these tasks and one to start the writeup. Once you have these pieces (within the first week), you can assemble them and have a working prototype. Then you can get started on interesting and distinguishing features.

4 Grading

You will be graded on the following:

- README present and clear (1pt)
- Code runs (1pt)
- Code detects some spelling errors and makes some attempt at replacement (5pts)
- Algorithmic use and structure (3pts)
- Algorithmic implementation (3pts)
- Report (2pts)
- Demo (2pts)
- Extra interface features (bonus points available)
- Extra spellcheck features and/or better performance (bonus points available)