CS 113
Homework Assignment 4

Given: November 24, 2009 **Due:** December 09, 2009

The assignment is due by 11:59 p.m. of the due date. The point value of each problem is shown in [ ]. Each solution must be the student's own work. Assistance should only be sought or accepted from the course instructor. Any violation of this rule will be dealt with harshly. Follow the documentation guidelines while writing and documenting your programs. Programs that are not well-documented will be penalized heavily. Your program must work on *all* possible inputs.

---

[100] 1. Spell Checker. This project builds on the Spell Checker part of Homework 3. In this, you will generate suggested replacements in a much more efficient way.

Suggesting Replacements
The procedure for suggesting replacements, used in Homework 3, was inefficient and somewhat simple-minded. Here is a more sohisticated and efficient algorithm for generating replacements. We will call (s, t) as a *replacement pair* if the words s and t are at edit distance 1 from each other. In Homework 3 we defined the following 4 types of replacement pairs.

- *Deletion pair*: t is obtained by deleting a letter in s,
- *Insertion pair*: t is obtained by inserting a letter into s,
- *Substitution pair*: t is obtained by substituting another letter for some letter in s, and
- *Swapping pair*: t is obtained by swapping a pair of letters in s.

We will now think of some replacement pairs as being *important* and others as being *unimportant*. The idea is that a pair (s, t) is important if there are reasons to believe that a user is likely to use the word s instead of t (or vice versa). We will use the following principle to determine which replacement pairs are important and which are not: *If two words sound similar, it is more likely that a user will use one instead of another.* This principle has specific implications to the 4 kinds of replacement pairs we consider. Here are examples.

- *Deletion pairs*: Dropping the letter t from the word `letter` does not change the sound of the word much, because of the two consecutive `t`'s in the word. Similarly, dropping the letter `i` or the second `e` in the word `receive`, does not change the sound of the word too much. So pairs such as `(letter, leter)` and `(receive, receve)` are considered important whereas pairs such as `(letter, lette)` and `(receive, eceive)` are not. This is because in the latter cases, the sound of the word changes significantly when the letter is dropped.

- *Insertion pairs*: If an (s, t) is an insertion replacement pair, then it is also a deletion replacement pair. Therefore, the above discussion holds for insertion replacement pairs as well. For example, I have often written `neccessary` for the word `necessary`. Since the mispelled word, obtained from the correctly spelled word via the insertion of an extra `c`, sounds similar to the original word, the pair `(necessary, neccessary)` is considered important.

- *Substition pairs*: If a pair of letters sound the same then a user is more likely to substitute one for another. For example, the letters `c` and `s` sound similar in some contexts and a user might type `nesessary` instead of `necessary`. This means that replacement pairs such as `(necessary, nesessary)` are considered important, whereas the pair `(necessary, netessary)` is not. In the latter case, replacing the letter `c` by a `t` drastically changes the sound of the word.

- *Swapping pairs*: If a pair of letters sound the same then swapping them will create a new word that sounds similar to the original word. For example, swapping the letters `e` and `i` in `receive` yields a similar sounding word and so the replacement pair `(receive, recieve)` is considered important whereas the pair `(receive, erceive)` is not.

To apply these ideas systematically, here is a table of pairs of letters that we will think of as being similar in sound.

a	e
c	s
c	k
e	i
g	j
i	y
o	u
v	w
s	z

Also, every letter sounds similar to itself. Given this table, we can tag each replacement pair as either being important or unimportant as follows, depending on the type of the pair.

- *Deletion pairs*: If two similar sounding letters occur consecutively in a word s and one of them is deleted, to give us the word t, then the pair (s, t) is important. All other deletion pairs are unimportant.

- *Insertion pairs*: Since every deletion pair is also an insertion pair as well, the important/unimportant tags for these kinds of pairs are defined above.

- *Substitution pairs*: If the word t is obtained by substituting a letter in the word s by a similar sounding letter, then the pair (s, t) is important. All other substition pairs are unimportant.

- *Swapping pairs*: If a two similar sounding letters occur in a word s and these are swapped, to give us the word t, then the pair (s, t) is important. All other swapping pairs are unimportant.

Here is how the idea of important and unimportant pairs comes into play while searching for replacements. Given a mispelled word w, use the following steps to find replacements.

(1) In this step we consider only the important replacement pairs. We start with w and generate replacement words as in Homework 3 but just using important replacement pairs. New strings will be generated as the search proceeds; check whether a string that is generated belongs to the dictionary (as in Homework 3) or not. If a generated string does belong to the dictionary, enqueue it into a queue of replacement strings. Terminate the search when either 10 replacement strings have been found or when 2000 strings have been generated. The number 2000 is somewhat arbitrary and should be fine-tuned for better performance. Note that the search may proceed down several levels starting at w, before it is terminated. Recall that in Homework 3, the search went down just 2 levels. This was because in each level there were a lot of words (especially in the second level) and we could not afford to go down more levels. Here, since we are only using important pairs, our search is narrower and can therefore afford to be deeper. Also, note that the queue of replacements contains words in the order in which they were generated and this order does correspond to a decreasing order of relevance of the replacement words.

(2) Suppose that in Step (1), x replacement words have been found, where x is smaller than 10. In Step (2), the goal is to find the remaining $10 - x$ replacement words. In Step (2) we do the following. Starting with the mispelled word w, we use all pairs (not just important pairs) to generate level one. For subsequent levels we use just the important pairs. As in Step (1), we terminate the search either when all the replacement words we seek have been found or we have generated 2000 words.

While I have specified quite a few details, I have left enough unspecified so as to leave you with a few design decisions to make. These decisions will affect the efficiency, robustness, and readability of your program. So make your choices with care.

Submission Guidelines. You must create a directory called P4_<last name> under which you must create directory for the programs. The directory must be named "spellChecker". The The "spellChecker" directory must contain the files `buildDictionary.py`, `dictionary.py` that contains the `Dictionary` class definition, and `spellCheck.py` that contains the spell checker program. Furthermore, this directories should not contain files that are not directly relevant to the problem. So the files that were used to build the dictionary should not be included.

Before you submit your program, you must create a *compressed*, *tar* file of the directory P4_<last name>. This can be done in two steps as follows.

```
% tar cvf P4_<last name>.tar P4_<last name>
% gzip P4_<last name>.tar
```

The above commands should create a file P4_<last name>.tar.gz . You must e-mail the above file as an attachment to `rajivg@clam.rutgers.edu`. The programs are due by 11:59p.m. of the due date. Hard copies of your programs must be turned in by the end of the first class after the deadline.

Some other commands that may be useful are the following.
Uncompressing a file can be done using the `gunzip` command.

```
% gunzip <file name>.gz
```

Extracting files from an archive can be done as follows.

```
% tar xvf <file name>.tar
```

To list all the files in the archive do the following.

```
% tar tf <file name>.tar
```