CS 113
Homework Assignment 3

Given: March 31, 2014 **Due:** April 22, 2014

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.

---

[150] 1. Birthday Paradox. Given below is a sample client program that uses classes `RandomBirthday` and `BirthdayParadox`. `RandomBirthday` generates birthdays uniformly at random for non-leap years. The `BirthdayParadox` class is designed to illustrate the paradox. You must design these classes based on the following client program.

```
from birthdayGenerator import RandomBirthday
from birthdayParadox import BirthdayParadox

def main():
    bday = RandomBirthday()
    for i,birthday in enumerate(bday):
        if i <= 5:
           print (birthday)
        else:
           break

    paradox = BirthdayParadox()
    paradox.distribution(28).display()
    print ('*********************************')

    print ('In {0} out of 10000 runs some pair out of 15 people
          (randomly chosen in each run) shared the same birthday'.format(
          paradox.countShareBday(15)))
    print ('*********************************')

    paradox.tabulate((10,15))
    print ('*********************************')
    print()
    paradox.tabulate((33,36),2000)
```

The output of the above client program is as follows.

```
[('oct', 25)]
[('apr', 28)]
[('feb', 10)]
[('mar', 17)]
[('jan', 2)]
[('mar', 1)]

Jan 2(1) 13(1) 17(1) 18(1)
Feb 6(1) 26(1)
Mar 6(1) 13(1)
Apr 19(1) 20(1)
May 14(1) 31(2)
Jun
Jul 9(1)
Aug 4(1) 8(1) 10(1) 16(1) 17(1)
Sep 7(1)
Oct 3(1) 5(1) 15(1) 24(1)
Nov 13(1) 14(1) 20(1)
Dec 12(1)

******************************
In 2519 out of 10000 runs some pair out of 15 people (randomly chosen in each
run) shared the same birthday.
******************************
Number of runs:  10000
-----------------------
#people       Pr[some two share the same birthday]
  10              1139/10000 =0.1139
  11              1369/10000 =0.1369
  12              1706/10000 =0.1706
  13              1929/10000 =0.1929
  14              2206/10000 =0.2206
  15              2461/10000 =0.2461
******************************

Number of runs:  2000
-----------------------
#people       Pr[some two share the same birthday]
  33              1571/2000 =0.7855
  34              1626/2000 =0.8130
  35              1612/2000 =0.8060
  36              1680/2000 =0.8400
```

Note that your `BirthdayParadox` class must implement at least the following four methods in addition to the constructor.

1. `countShareBday`: performs number of runs as specified by the input. In each run it "chooses people" (as many as specified in the input) uniformly at random. It returns the number of runs in which some pair of people share the same birthday.

2. `distribution`: notes the different birthdays of randomly chosen people, as many as specified by the input to the method.

3. `display`: outputs the birth dates of the chosen people with the number of people born on each date shown in parenthesis.

4. `tabulate`: outputs a table with each row containing the number of people, say n, that lies in the interval given by the input, and fraction of runs (number of which is specified by the input) in which some pair out of the n people (randomly chosen in each run) shares the same birthday.

The class `RandomBirthday` generates birthdays uniformly at random for non-leap years. The class `BirthdayParadox` uses `RandomBirthday`. Appropriate exception-handling must be implemented.

[200] 2. Spell Checker. This project builds on the Spell Checker project that you implemented in Homework 2. There key differences between this project and the one in Homework 2 are as follows.

1. You will now build a larger dictionary and will therefore use more sophisticated 2-level data structure to provide efficient insertion and search. Specifically, the `buildDict` program will identify the 500 most frequently occuring words in the text documents it reads. It will write these words first into `words.dat`, followed by the rest, less frequently occuring words. When `spellCheck` builds the dictionary from `words.dat`, it puts the 500 most frequently occuring words in one "high priority" data structure (we will call this the *primary dictionary*) and the rest of the words in a second, "lower priority" data structure (we will call this the *secondary dictionary*). The primary dictionary needs to be extremely fast because the words in it are likely to be accessed very often. The secondary dictionary can afford to be a little slower.

2. You will provide a more sophisticated interface to the user, by suggesting possible replacement words for errors discovered by `spellCheck`. In order to do this, we need to model the kinds of errors that people make and then use this model to search for possible correct words the user might have actually meant to type. More details on this are described below.

3. The spelling checker built in Homework 2 makes no attempt to correctly deal with words that contain apostrophes such as "I'll" or "could've" etc. This project asks you to fix this problem. In this project you will also make an attempt to avoid including proper nouns in the dictionary.

Overview: As in the previous project, you will have to write two separate programs: `buildDict` and `spellCheck`. As in Homework 2, the program `buildDict` builds a dictionary of words, while the program `spellCheck` uses this dictionary of words to determine which words in a given document are misspelled. The program `buildDict` builds a dictionary by extracting words from a document known to be free of spelling errors. In addition to the text of the three novels that you used in Homework 2, you should use the following three new novels also to build a dictionary.

- *Treasure Island* by Robert Louis Stevenson.
- *Gulliver's Travels into Several Remote Nations of the World* by Jonathan Swift.
- *War and Peace* by Leo Tolstoy.

In this project, `buildDict` also keeps track of the *frequency* of occurrence of each word that it extracts from the document. When `buildDict` completes extracting all the words from the document, it writes the extracted words onto a text file called `words.dat`. In particular, `buildDict` starts by writing the 500 most frequent words (in any order) into `words.dat` and then writes the remaining words (in any order) into `words.dat`. The `spellCheck` program reads `words.dat` and creates a 2-level dictionary: a *primary dictionary* containing the 500 most frequent words and a *secondary dictionary* containing the remaining words.

Building a dictionary: The rules that define what a word is are as described in Homework 2 handout, with the following changes. Now you should discard capitalized words that do not start a sentence. That is, such strings should no longer be considered valid words and should not be inserted into the dictionary. A word is said to be capitalized, if its first letter is in upper case and the rest of the letters are in lower case. A word is said to begin a sentence, if the non-whitespace character immediately preceding the first letter of the word is a period. The reason for throwing out capitalized words that do not begin a sentence is that such words are likely to be proper nouns. It is possible that throwing out words as described above may not remove all proper nouns and may in fact remove some words that are not proper nouns.

As you may have already noticed, the spelling checker in Homework 2 makes no attempt to correctly deal with words that contain apostrophes such as "I'll" or "don't" or "could've." For example, if a text document used by `buildDict` contains the sentence:

```
I could've taken it yesterday, but I guess now I'll have to take it
next Sunday.
```

Then `ve` and `ll` were flagged as valid words, which they are clearly not. In this project you should implement a clean way of dealing with apostrophes. Specifically, if words such as "I'll" and "could've" occur in the text document that `buildDict` reads, then these words should be considered to be correctly spelled words. However, "ve" and "ll" should not be considered being correct. You should think about modifying our definition of a word from Homework 2, in order to cleanly deal with apostrophes. You are responsible for figuring out a simple and clean scheme that can reasonably deal with apostrophes.

Two-level dictionary: For this project you will use *hashing with chaining* to implement the dictionary. The idea of a hash table is simple. Let U be the set of all possible strings of lower case letters. Let $h : U \to [0, \ldots, M-1]$ be a function that maps each string in U to an integer in the range 0 through $M-1$. Then any subset S of strings can be stored in an array, let us call this `table`, of size M by storing a string $s \in S$ in slot `table`$[h(s)]$. To determine whether a given string $s' \in U$ is also in S, we simply compute $h(s')$ and determine if `table`$[h(s')]$ contains s'. The only problem with this method is that there can be *collisions* between strings. For example suppose that for two distinct strings s and s' in S, $h(s)$ and $h(s')$ are both equal to some integer k. This means that both s and s' have to be stored in slot `table`$[k]$. This approach to resolving collisions is called *hashing with chaining*. The efficiency of this approach depends on making sure that none of the lists are too long.

The only aspect of a hash table left to discuss is the hash function h itself. Any string of lower case letters can be thought of as a number in base-26 by thinking of each letter as the number obtained by subtracting 97 (the ASCII value of 'a') from the ASCII value of the letter. For example, the string `next` can be thought of as the following base-26 number: 13 4 23 19. Thus a string $s_{n-1}s_{n-2}\ldots s_1 s_0$ has an equivalent decimal value defined as:

$$D(s) = 26^{n-1} \cdot x_{n-1} + 26^{n-2} \cdot x_{n-2} + \cdots + 26^1 \cdot x_1 + 26^0 \cdot x_0$$

where x_i is the ASCII value of s_i minus 97, for each i, $0 \le i \le n-1$. Now for any string s define the hash function $h(s)$ as $h(s) = D(s) \bmod M$. Thus the function h maps any string onto the range $0, \ldots, M-1$. Typically, in most languages you will have to watch out for possible integer overflow while computing $D(s)$. In other words, if s is very long, then $D(s)$ may be a very large integer and because of integer overflow $D(s)$ may have the wrong value. Here is an illustration of how to get around this problem. Suppose that the string $s = abcd$. Then $D(s)$ can be written as

$$D(s) = 26\,(26\,(26\,(a') + b') + c') + d'$$

where a', b', c', and d' are the ASCII values of a, b, c, and d, respectively. From this observe that $D(s)$ can be computed in a loop, such that in each iteration the current value of $D(s)$ is multiplied by 26 and the ASCII value of the next letter is added. You can get around the integer overflow problem by taking "mod M" at the end of each iteration rather than after $D(s)$ has been completely computed. It is not hard to show that $D(s)$ computed in this manner will be identical to $D(s)$ computed by taking "mod S" once at the end, after $D(s)$ has been completely computed.

It is critical to the efficiency of a hash table based data structure that the hash value of the given string be computed extremely efficiently. Tricks such as bit shifting can be used to speed up the computation of the hash function described above.

The only thing left to decide is the value of M. It is also common practice to choose the size of the hash table to be a prime number. The program `buildDict` separates the words it extracts into two categories of high frequency and low frequency words so as to help `spellCheck` build a more efficient, two-level dictionary data structure. The `spellCheck` program stores the 500 high frequency words in primary dictionary. Since the number of words being stored in the primary dictionary is small, we can afford to use a hash table,

that has a small load factor - say 1/10. This means that the size of the hash table will be at least 5000. So for your table size you might want to pick the smallest prime greater than 5000. The rest of the words (i.e., the low frequency words) are stored in the secondary dictionary. We may have to store a large number of words in the secondary dictionary and therefore due to memory constraints, we cannot use a hash table with too small a load factor. So for the secondary dictionary, use a hash table with a load factor of about 2 or 3. In other words, use a table whose size is about half or even a third of the number of words that you intend to store in the table.

The motivation for this two-level organization of the dictionary is that many empirical studies have found that 80% of the words in an average document come from a list of about 500 words! For these words, we can afford to build a hash table with small load factor and get, almost instantaneous response to insert and search operations. For the rest of the words, we can afford to provide slightly slower access, because they are accessed less frequently. This two-level organization provides a good balance between time and space efficiency.

While checking for spelling errors, the `spellCheck` program takes each word and first looks for it in the primary dictionary. If the word does not appear in the primary dictionary, it then looks in the secondary dictionary. This order of looking first in the primary dictionary and then in the secondary dictionary is quite important. If the word is not found in either dictionary, `spellCheck` will flag the word as being mispelled.

Suggested Replacements: Like in Homework 2, `spellCheck` responds to a misspelled word by providing the following prompt:

```
replace (R), replace all (P), ignore (I), ignore all (N), exit (E):
```

However, what the program does next is different. If the user responds with an `R` or a `P`, then the program responds by producing a list of at most 10 suggested replacement words that are labeled 0 through 9, followed by an option that allows the user to input their own replacement word. So for example, if `spellCheck` encounters the word `ont` it might respond as follows:

```
(0) ant
(1) one
(2) on
(3) opt
(4) out
(5) Use my replacement
```

Note that the list of suggested replacements may not always contain 10 words, 10 is just the maximum number of replacement words produced by the program. We use the limit 10 because we do not want to inundate the user with too many options. Also note that sometimes, the program may find no replacement words and in this case the user will just see the prompt

```
(0) Use my replacement
```

The user responds by typing a number corresponding to the option they want. If, in the above example the user types number between 0 and 4 (inclusive) then the program just uses the corresponding replacement word from the list. Otherwise, if the user types 5, then the program needs to ask the user for a replacement word and it uses whatever the user types in response. The question now is: how do we generate reasonable replacement words? We do this by assuming that the most likely errors a user makes while typing are

- Missing a letter (typing `tet` instead of `text`),
- Including an extra letter (typing `taext` instead of `text`),
- Substituting one letter for another, (typing `rext` instead of `text`) and
- Swapping a pair of consecutive letters (typing `txet` instead of `text`).

Now consider strings s and t. We say that t is at an *edit distance* 1 away from s if

- t is obtained by deleting a letter in s or
- t is obtained by inserting a letter into s or
- t is obtained by substituting another letter for some letter in s or
- t is obtained by swapping a pair of letters in s.

Suppose we have encountered a misspelled word w. To find meaningful replacements for w, we find all words in the dictionary (i.e., correctly spelled words) that are at most edit distance 2 away from w. The choice of 2 is somewhat arbitrary and depending on the results from this project we may fine tune this value. Also, recall that we are interested in reporting at most 10 possible replacements, so if the number of words in the dictionary within edit distance 2 of w is more than 10 we have to prune the list to contain exactly 10 words. Correctly spelled words that are edit distance 1 away should be considered better replacements than those that are edit distance 2 away. You should keep this in mind when pruning the list of candidate replacements.

The actual procedure for finding all correctly spelled words that are at most edit distance 2 away from w is fairly simple and you are responsible for figuring this out.

Also, if the mispelled word begins with an upper-case letter then the all of the suggested replacements must begin with an upper-case letter and the word that it is replaced by in the output must also begin with an upper-case letter.

Your spell-checker must have a method called `statistics` that outputs information about the tables that are used to store the primary and the secondary dictionaries. Since the performance of hashing with chaining is determined by the length of the lists that are formed due to collision the `statistics()` method should output a table that shows the distribution of the lengths of the lists.

Sample Run: Let the content of `mytext`, the file that is to be spell-checked, be as follows.

```
Too mony erors.  Mony of these erors will be corrected by Rajv.
```

Consider the following `main` program.

```
from spellCheck import SpellChecker

def main():
  sc = SpellChecker()
  sc.statistics()
  query = 'Name of the document to be spell-checked:  '
  sc.spellcheck(raw_input(query))
```

When the above program is executed using some dictionary that is constructed using some text the output is as follows.

```
Stats on Primary Dictionary
-------------------------------
Length          Number of Lists
    0               4529
    1                448
    2                 26

Stats on Secondary Dictionary
-------------------------------
Length          Number of Lists
    0               1365
    1               2332
    2               2284
    3               1548
    4                764
    5                352
    6                154
    7                 45
    8                 10
    9                  2
    10                 2

Name of the document to be spell-checked:  mytext

           ``mony'' is unknown.

replace(R), replace all(P), ignore(I), ignore all(N), exit(E): r
( 0 ) money
( 1 ) bony
( 2 ) many
( 3 ) monk
( 4 ) mon
```

```
( 5 ) agony
( 6 ) among
( 7 ) irony
( 8 ) stony
( 9 ) smoky
( 10 ) Use my replacement
Your choice:  2

          ‘‘erors’’ is unknown.

replace(R), replace all(P), ignore(I), ignore all(N), exit(E): P
( 0 ) errors
( 1 ) ferons
( 2 ) heroes
( 3 ) jurors
( 4 ) terrors
( 5 ) error
( 6 ) brows
( 7 ) crops
( 8 ) cross
( 9 ) crows
( 10 ) Use my replacement
Your choice:  0

          ‘‘Mony’’ is unknown.

replace(R), replace all(P), ignore(I), ignore all(N), exit(E): R
( 0 ) Money
( 1 ) Bony
( 2 ) Many
( 3 ) Monk
( 4 ) Mon
( 5 ) Agony
( 6 ) Among
( 7 ) Irony
( 8 ) Stony
( 9 ) Smoky
( 10 ) Use my replacement
Your choice:  2

          ‘‘Rajv’’ is unknown.

replace(R), replace all(P), ignore(I), ignore all(N), exit(E): R
( 0 ) Naiv
( 1 ) Race
```

```
( 2 ) Rack
( 3 ) Raft
( 4 ) Rage
( 5 ) Rags
( 6 ) Rag
( 7 ) Rah
( 8 ) Rail
( 9 ) Rain
( 10 ) Use my replacement
Your choice:  10
Replacement word:  rajiv
```

The contents of `mytext.out` will be as follows.

```
Too many errors.  Many of these errors will be corrected by Rajiv.
```

Final Remarks: You must not use `re` module in your program. Appropriate exception-handling must be implemented. Points will be deducted for programs designed poorly.

Submission Guidelines. You must create a directory called P3_<last name> under which you must create two more directories for the two programs. The directories must be named "birthdayParadox" and "spellChecker". The "birthdayParadox" directory must contain the files `birthdayGenerator.py` and `birthdayParadox.py`. The "spellChecker" directory must contain the files `buildDictionary.py`, `dictionary.py` that contains the `Dictionary` class definition, and `spellCheck.py` that contains the `SpellChecker` class definition. Both these directories should not contain files that are not directly relevant to the problem.

Before you submit your program, you must create a *compressed*, *tar* file of the directory P3_<last name>. This can be done in two steps as follows.

```
% tar cvf P3_<last name>.tar P3_<last name>
% gzip P3_<last name>.tar
```

The above commands should create a file P3_<last name>.tar.gz . You must e-mail the above file as an attachment to `rajivg@crab.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
```