Problem 2 SPAM: SPelling A la Millisoft (30 Points; Individual or Pair)

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/SPAM on 3/22/2017

Your functions for this problem should be in a file called `hw3pr2.py`.

OK, so the SPAM acronym is a stretch! Here is the gratuitous "true story": You work at Millisoft, a leading software company. One day the CEO, Gill Bates, comes into your office sipping on a Jolt Cola. "I've decided that Millisoft is going to have a new spell checking product called "spam" and it's yours to develop and implement! As an incentive, you'll get a lifetime supply (two six-packs) of Jolt when it's done."

Spell checking of this type is both useful to those of us hoo hav trubbel speling (or tpying) and also useful in biological databases where the user types in a sequence (e.g. a DNA or amino acid sequence) and the system reports back the near matches.

How do we measure the similarity of two strings? Recall from class that the "edit distance" between two strings `s1` and `s2` is the **minimum** number of "edits" that need to be made to the strings to get them to match. An "edit" is either a **replacement** of a symbol with another symbol or the **deletion** of a symbol. For example, the edit distance between "spam" and "xsam" is 2. We can first delete the "x" in "xsam" leading to "sam" and then delete the "p" in "spam" to also make "sam". That's two edits, for an edit distance of 2. That's the best possible in this case. Of course, another possible sequence of edits would have been to delete the "s" and "p" from "spam" to make "am" and delete the "x" and "s" from "xsam" to to make also "am". That's 4 edits, which is not as good as 2.

Here's the `ED` function that we wrote in class:

```
def ED(first, second):
    '''Returns the edit distance between the strings first and
second.'''

    if first == '':
        return len(second)
    elif second == '':
        return len(first)
    elif first[0] == second[0]:
        return ED(first[1:], second[1:])
    else:
        substitution = 1 + ED(first[1:], second[1:])
        deletion = 1 + ED(first[1:], second)
        insertion = 1 + ED(first, second[1:])
        return min(substitution, deletion, insertion)
```

Now, here's `ED` in action:

```
In [1]: ED("spam", "xsam")
Out[1]: 2

In [2]: ED("foo", "")
Out[2]: 3

In [3]: ED("foo", "bar")
Out[3]: 3

In [4]: ED("hello", "below")
Out[4]: 3

In [5]: ED("yes", "yelp")
Out[5]: 2
```

It's cute, but it's slow!

Try your `ED` function on a few pairs of long words. For example, here's my attempt to observe the *extraordinary* slowness of this program! Exercise your *originality* in finding other pairs of very long words to try out!

```
In [1]: ED("extraordinary", "originality")
```
Wait for a bit—you will get an answer (it's 10).

Since the recursive program is very slow, Gill has asked you to reimplement it using memoization. Write a new function called `fastED(S1, S2)` that computes the edit distance using a global Python dictionary to memoize previously computed results.

After writing `fastED(S1, S2)`, test it to make sure that it's giving the "write" answer. Here are a few more test cases:

```
In [1]: fastED("antidisestablishment", "antiquities")
Out[1]: 13

In [2]: fastED("xylophone", "yellow")
Out[2]: 7
```

```
In [3]: fastED("follow", "yellow")
Out[3]: 2

In [4]: fastEd("lower", "hover")
Out[4]: 2
```

topNmatches(word, nummatches, ListOfWords)

Next, you'll write a function whose signature is

```
    def topNmatches(  word, nummatches, ListOfWords )
```

which takes in

- `word`, a string which is the word to match (using `fastED`)
- `nummatches`, an integer which is 0 or greater
- `ListOfWords`, a list of strings against which to match `word`

and, from there, `topNmatches` should output the *alphabetically-sorted* list of a total of `nummatches` words from `ListOfWords` that have the lowest edit-distance scores with the input `word`

- you should be sure to return only `N` words. If there is a tie at the last place among those `N`, there will be an ambiguity as to which words to return
- we will not test it in those "tie" cases, however, so you're welcome to handle that ambiguity as you see fit

Here are a couple of examples:

```
In [1]: topNmatches( "spam", 3, [ "spam", "seam", "wow",
"cs5blackrocks", "span", "synecdoche" ] )
Out[1]: [ 'seam', 'spam', 'span' ]

In [2]: topNmatches( "spam", 1, [ "spam", "seam", "wow",
"cs5blackrocks", "span", "synecdoche" ] )
Out[2]: [ 'spam' ]

# we would not test the above example with nummatches == 2,
since the output would be ambiguous
```

And here are a few details that may be of help:

- You will need to find the score (edit distance) for each word in the master list of words. One way to do this is to construct another list that is just like your master list, except that each entry in that new list will be a tuple of the form `(score, word)`. For example, the tuple `(42, "spam")` would mean that the word `"spam"` has edit distance 42 from the word that the user entered. You can use `map` to build this list of tuples!
- You'll need to sort the words by score. While mergesort is very fast, the amount of recursion that it requires will likely exceed the recursion limit permitted by your computer. Therefore, use the very fast sorting algorithm built into Python, which is named `sort`. This function is used as follows: If `L` is the name of your list (it can have any name you like) then the syntax `L.sort()` will modify `L` by sorting it in increasing order. This will sort `L` but will not return anything. So, use the line `L.sort()` rather than `sortedList = L.sort()`. It's a strange syntax, but it works (and we'll talk about the reason for this syntax in a few weeks.) You may wish to sort a list of items, each of which is a list or a tuple. For example, if you have a list `L` that looks like this: `[[42, "hello"], [15,     "spam"], [7, "chocolate"]]` and you do `L.sort()`, it will sort `L` using the first element in each list as the sorting key. So, `L.sort()` in this case will change `L` to the list `[[7, "chocolate"], [15, "spam"], [42, "hello"]]`.
- You'll use `sort` again in order to get your ultimate list of words into alphabetical order!

You'll put this together into an interactive program, known as `SPAM` in the next part... .

SPAM!

Finally, your last task is to write a function called `spam()` that loads in a large master list of words and then repeatedly does the following:

- The user is shown the prompt `spell check>` and prompted to type in a word.

- If the word is in the master list, the program reports `Correct`. You can test if a string is in a list by using the `in` keyword as in `if "spam" in ["everyone", "loves", "spam"]` returns `True`.
- If the word is not in the master list, the program should compute the edit distance between the word and *every* word in the master list. Then the 10 most similar words should be reported, in order of smallest to largest edit distance. The program should also report how long it took to find this list.

Here is an example of what your program will look like when running. The actual times may vary from computer to computer, so don't worry if you see different running times on your computer. Moreover, if there are ties in the edit distance scores, you may break those ties arbitrarily when sorting.

```
In [1]: spam()
spell check> hello
Correct
spell check> spam
Suggested alternatives:
 scam
 seam
 sham
 slam
 spa
 span
 spar
 spasm
 spat
 swam
Computation time: 2.06932687759  seconds
```

Here are the ingredients that you will need:

- Download 3esl.txt into the same directory (folder) where your program resides. This is our master list of words: It is simply a file with 21877 words in alphabetical order. Save this file on your machine. (On the Macs, push `control` and mouse-click on this link and then choose to save this link in a file.) Make sure that this file is in the same directory as your program.
- The following three lines will open the file `3esl.txt`, read it, and split it into a list called `words`. ***Be sure these lines are INSIDE your spam function*** and not at the top-level of the file (they will cause all of your tests to fail!) So, the top of your function will look like this (with a better docstring, for sure):
- `def spam():`

- `""" docstring """`
- `f = open("3esl.txt")`
- `contents = f.read()`
- `words = contents.split("\n")`
- You'll need to prompt the user for input. The Python function `input(S)` displays the string `S` and then waits for the user to enter a string and hit the return key. The string that was typed in by the user is now the value returned by the `input` function. For example

  ```
  userInput = input("spell check> ")
  ```

  displays the string `spell check>`, waits for the user to type in a string, and then returns that string so that `userInput` now stores that string. (You'll find that things look nicer if your prompt string ends with a blank.)

- To compute the amount of time that transpires between two points in your program, we recommend the following:
  - First, have the line `import time` at the top of your program to import the `time` package;
  - Next, any time you like, call `time.time()` to capture the number of seconds (a floating point number) that have elapsed since some point in the past (perhaps when your program started, or perhaps some other well known time—it doesn't really matter!). By capturing `time.time()` at two different places and subtracting the first value from the second, you can determine the elapsed time in that part of the program. Here is an example:

```
        import time

        def time_example():
           yadda, yadda, yadda
           startTime = time.time()
           blah, blah, blah
           endTime = time.time()
           print("The elapsed time executing blah, blah, blah
was", endTime - startTime)
```

Submitting...

Submit your `hw3pr2.py` file. Do not submit your `3esl.txt` file; it's a big file - and we've got it!

When you submit, your file should be autograded -- be sure to check the autograding results (there may be small errors you have time to fix and resubmit!)

Also, be sure that there are no file-reading
or `input` lines ***outside*** the `spam` function (that will cause the autograding tests to fail!)