SPelling at Millisoft (SPAM)

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? 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 the two strings 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 an `ED` function (edit distance function):

```
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:

```
>>> ED("spam", "xsam")

2

>>> ED("foo", "")

3
```

```
>>> ED("foo", "bar")

3

>>> ED("hello", "below")

3

>>> ED("yes", "yelp")

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!

```
>>> 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 implement it using memoization. Write a new function called `fastED(S1, S2)` that computes the edit distance using a Python dictionary to memoize previously computed results. The dictionary can be a global variable in your file.

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

```
>>> fastED("antidisestablishment", "antiquities")

13

>>> fastED("xylophone", "yellow")

7

>>> fastED("follow", "yellow")

2

>>> fastEd("lower", "hover")

2
```

SPAM!

Finally, your last task is to write a function `getSuggestions()` which takes a user's input word and finds some words in the dictionary that are close in edit distance. This function is used 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`.
- 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 in order of smallest edit distance to larger edit distance should be reported.

Finding those similar words is the job of `getSuggestions()`. The code for `spam()` is in the provided template file.

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.

```
>>> 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
```

The master list of words is in the file 3esl.txt which you should download into the same directory (folder) where your program resides. It is simply a file with 21877 words in alphabetical order. Save this file on your machine.

The code in the template uses some Python features you may not be familiar with. Here are some explanations:

- The program uses a while-loop for the repeating process of prompting the user and responding to their input. We will study loops soon but you won't be writing your own loops in this assignment.
- The following three lines will open the file `3esl.txt` read it, and split it into a list called `words`.

```
f = open("3esl.txt")
contents = f.read()
words = contents.split("\n")
```

We'll talk about the reason for the dot notations in a few weeks. You will also see this notation below, for the `.time()` and `.sort()` functions.

- To prompt the user for input, we use the Python function `input(S)` which 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. We also use `.strip()` which removes any spaces at the start or end of the string.

- In order to compute the amount of time that transpires between two points in your program, we use the following.
  - The line `import time` imports the `time` package;
  - The call `time.time()` captures the number of seconds (a floating point number) that have transpired since your program started. 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.

- Once the suggestion list is obtained, the program sorts it. We use a fast sorting algorithm built into Python named `sort`. This function is used as follows: `suggestions.sort()` will modify `suggestions` by sorting it in increasing order. This will sort `suggestions` but will not return anything. For a list of tuples, the function sorts by the first element; so we get our suggestion list sorted by distance.
- We need to report the top 10 words. However, you might later wish to change 10 to 12 or even 42. So, rather than having the number 10 inside your code (that is called a "magic number" and it's a bad thing to have in your code), we define a global variable called `HITS` and use that in the `spam()`. If you later decide that you want to show a different number of similar words, you can simply change the value of that global variable.