You will create a program that allows a user to spell-check and remove repeated words in a text file. The program will do the following:

- Prompt the user for a text file to check (the filename) [called input file below]
- Prompt the user for a text file of known words with correct spellings (the filename)
- Read the contents of both files
- Read the contents of a text file containing *common mistakes* (called common.txt)
- Process the input file line-by-line
    - Prompt user to correct each mistake found on each line
- Prompt the user for a new name to save the corrected document
    - Saves the updated document in the file
    - Updates commont.txt with any new common mistakes
- Prompt the user to repeat with new files or quit the program

Details of the program are illustrated in the following example run and specifications below. Suppose we have the text file (essay.txt) consisting of the following three lines:

```
It was the best of times.
It was the blurst of tims.
Where does does the cat go from here?
```

Suppose there is also a file called spell.txt which has a collection of words (one word per line, each spelled correctly). Running the program might look like the following (yellow is used to indicate user input)

```
Fancy Spell-Checker
File to check: essay.txt
File with known words [enter for default]: spell.txt
--------------------
Processing essay.txt
Line 2: the 'blurst' of
  (0) 'blurst'  (1) 'first'  (2) 'blast'  (3) 'fast'   (4) edit
Option: 2
Line 2: of 'tims'.
  (0) 'tims'    (1) 'times'  (2) 'time'   (3) 'lime'   (4) edit
Option: 0
Line 3: Where 'does does' the
  (0) 'does does'  (1) 'does'
Option: 1
--------------------
File to save updated document: new_essay.txt
Hit enter to continue with another file or quit exit: quit
```

After the program terminates, there will be a new text file (new_essay.txt) with the following three lines

```
It was the best of times.
It was the worst of tims.
Where does the cat go from here?
```

Here are some further details.

1. For each "problem" encountered, which is either a misspelled word or repeated word, the program displays which line number the problem occurred and shows a snippet of the text showing the problem. The snippet of text should include the word that comes before the problem (unless it is the first word) and the word that comes after the problem (unless it is the last word in the line). The problem should be inside quotes.

2. For each problem, some options for the user must be displayed.
   a. The option 0 is always to make no change.
   b. For misspelled words, a short list (3) of **suggested** corrections will be listed. The first suggested correction will come from the common.txt file (if word is present) and the rest (either 2 or 3 words) will be computed (see below for details). The last option will allow the user to manually enter the correction (edit).
   c. For repeated words, the second option (1) will be to remove one of the repeated words.

3. The common.txt file will have common spelling mistakes and the correct word. Each line consist of two words (with whitespace between them). The first word is the misspelled word

and the second word is the correct version of the word. For example, (correcting for Canadian vs American English), some lines in the file might be

```
color colour
neighbor neighbour
```

4. When processing a file, if the same mistake and correction occurs two (2) or more times, AND the correction does not appear in the common.txt file, then this correction should be added to the common.txt file (after the input file is processed).

   Note: we will not consider the scenario that the same word is corrected to different words (each multiple times).

5. The rest of the suggested words will be determined by finding words (from the collection of known correct words) that are "similar" to it. For this, you will compute a simplified Levenshtein distance (edit distance) between the misspelled word and every known correct word and choose from the words with smallest distance. The simplfied Levenshtein is defined as follows:

```
simplified_lev(a :str, b : str) -> int

simplified_lev(a,b) = max( len(a*), len(b*) )
```

   where a* is what remains of the string a when removing all matching leading and trailing characters of a and b, and b* is what remains of the string b when doing the same. Here are some examples:

| a | b | a* | b* | simplified_lev(a,b) |
|---|---|---|---|---|
| "abc" | "abc" | "" | "" | 0 |
| "abc" | "vwxyz" | "abc" | "vwxyz" | 5 |
| "abcxyz" | "abqq" | "cxyz" | "qq" | 4 |
| "abc23xyz" | "aWz" | "bc23xy" | "W" | 6 |
| "abcONE" | "abc" | "ONE" | "" | 3 |
| "abcONE" | "ANE" | "abcO" | "A" | 4 |
| "abXyz" | "abXCATSwz" | "y" | "CATSw" | 5 |

   You must have a helper function

```
ab_star(a : str, b:str) -> tuple
```

   where the returned value is a tuple (a*, b*) where a* and b* are as described above.

   For each word in the list of known words, you will need to compute simplified_lev() of that word and the misspelled word. From this, take any 2-3 (as needed) words with the minimal simplified Levenshtein distance. If you need 3 words and there are only 2 with the minimal distance, take those 2 and 1 other with the next smallest distance.

Note: You do not need to compute this for every known word. Notice that if len(a) = 3 and len(b) = 12, then 9 <= simplified_lev(a,b) <= 12. If a is the misspelled word, this suggests that you should stay computing the distances of all words with the same length as a first and if you find words with distance 1 then they are the best match. After checking all these words, look at words with length len(a) ±1, and then len(a) ±2, etc. You will eventually reach a point where checking longer (or shorter) words cannot possibly have a distance smaller than the minimal found so far.

6. Words will never be "broken" over a line in the input text files.

7. You are not required to find repeated words that cover two lines in the input (i.e., the first word is the last of one line and the repeated word is the first of the next line).

8. On any given line exactly one of the following will hold: (i) there will be one or more spelling mistakes, or (ii) there will be one or more repeated words, or (iii) there will be no spelling mistakes or repeated words.
    a. There will never be a spelling mistake AND a repeated word on the same line.
    b. Repeated words will never come in triples, or quadruples, or more. A repeated word will always be some word followed by itself (without punctuation) exactly once. So, you won't be tested on anything like "**Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo**".

9. Your program should not "crash" if the user enters a bad filename (for example, a file that does not exist). Instead, it should ask the user for the filename again. (Use try/except)

10. If common.txt is not present, your code should still work without it. (If new common mistakes are found, your program should create a new file common.txt with your new common mistakes in it). The order of the common mistakes in common.txt does not matter.

11. Your ab_star function can use recursion if you wish.

Write your program in a file called "**a5.py**".

Note: This question *seems* like it is asking for a LOT of things. Don't try to write the entire solution in one go. Break the problem down into different tasks and attack each task one at a time. Writing good code is an incremental process. Get something working and then move ahead to the next task. Write out what you want to do on paper before you start writing code! Have a plan before you write your code. This will help you to spend much less time writing code.

Suggestion: incremental steps will get you there faster! For example, I would start with ab_star, then work on simplified_lev, then a basic main function to get the user's input and load the files, then break down each line looking for misspelled words, then handling the misspelled words, etc.

## Recap [a5.py]

Submit a single file called **`a5.py`**. This will contain at *least* THREE functions: the program driver (**`main()`**function), and your helper **`simplified_lev()`** and **`ab_star()`** functions.

Note: Your simplified_lev and ab_star functions must be defined as

```
def simplified_lev(a:str, b:str) -> int:
   # your code
   # returns some integer

def ab_star(a:str,b:str) -> tuple:
   # your code
   # returns the tuple (a*,b*)
```