CSE 114 Fall 2023: Assignment 6

Due: October 29, 2023 at 11:59 PM [KST]

Read This Right Away

Directions

- At the top of every file you submit, include the following information in a comment
    - Your first and last name
    - Your Stony Brook email address
- Please read carefully and follow the directions exactly for each problem.
- Your files, Java classes, and Java methods must be named as stated in the directions (including capitalization). Work that does not meet the specifications may not be graded.
- Your source code is expected to compile and run without errors. Source code that does not compile will have a heavy deduction (i.e. at least 50% off).
- You should create your program using a text editor (e.g. emacs, vim, atom, notepad++, etc).
- You may use the command-line interface to compile and run your programs or you can now use IntelliJ IDEA.
- Your programs should be formatted in a way that is readable. In other words, indent appropriately, use informative names for variables, etc. If you are uncertain about what a readable style is, see the examples from class and textbook as a starting point for a reasonable coding style.

What Java Features to Use

For this assignment, you are *not* allowed to use more advanced features in Java than what we have studied ***at the time the assignment is posted***. If you have a question about features you may use, please ask!

What to Submit

Combine all your .java files from the problems below into a single zip file named as indicated in the **Submission Instructions** section at the end of this assignment. Upload this file to **Brightspace** as described in that section.

Multiple submissions are allowed before the due date. Only the last submission will be graded.

Please do **not** submit `.class` files or any that I did not ask for.

Naming Conventions In Java And Programming Style In General

Please use these conventions as you establish your *programming style.* Programming professionals use these, too.

- **Names:** Choose informative names,
  - e.g. `hourlyRate` rather than `hr` for a variable name.
  - `CheckingAccount` rather than `CA` for a Java class name.
- **Class names:** We start a class name with an uppercase letter as I have in my examples: Hello not hello or `HELLO`. For multi-word names you should capitalize the first letter of each word,
  - e.g. `UndergraduateStudent, GraduateStudent, CheckingAccount`
- **Variable names:** We start a variable name with a lowercase letter and capitalize each 'word' after that. This is called 'Camel case':
  - e.g. `count, hourlyRate, averageMonthlyCost`
- **Method names:** Naming convention for method names is the same as that for variable names.
- **Use of white space:** Adopt a good indentation style using white spaces and be consistent throughout to make your program more readable. Most text editors (like emacs and vim) should do the indentation automatically for you. If this is not the case, see me and I can help you configure your setup.

I will not repeat these directions in each assignment, but you should follow this style throughout the semester.

Problem 1 (25 points)

This problem will involve writing code that checks the spelling of words in a string against a dictionary. The goal, however, is not just to identify misspelled words, but also to suggest what the intended word might have been!

We will not be using any sophisticated pattern matching for this. Rather, the approach will be a little 'brute-force' as we will try a bunch of different simple transformations on each word.

I provide a skeleton class called `Corrector` in the file `Corrector.java`. This sets up the dictionary array and the input string which are given in the file `DictDoc.java`.

We will break the task down and write several helper methods to do different checks and operations.

1. I have set up the dictionary as I mentioned above. I also create the word list by passing the input string to **splitInput()**. You, however, must write splitInput() to break the input string into words. This will use the Java String utility *split()* with a regular expression that will create an array of words to check. Return the array of words. Be sure to use a regular expression similar to the one shown in class that

removes punctuation as it is splitting the sentences into words. You want to be sure all punctuation is stripped otherwise, legitimate words will show up as misspelled.

2. Now write a method called checkInDict(). It takes two parameters, a String array (String[]) which is a reference to the dictionary array. And a String to check (the next word). Here is the prototype:

   public static Boolean checkInDict(String [] dict, String word);

   This will compare *word* to each word in the dictionary array. Note, there will be issues with case so you should convert the word to lower case (all of the words in the dictionary are already in lower case).

   Verify this works by writing several calls to it with a word or two that are in the provided dictionary, and a couple that are not. Print out each boolean result. Once you are happy that checkInDict() is working, you can remove these test calls.
3. Now, you will write 3 routines that will progressively modify a word and look for it in the dictionary. Typically, simple misspellings involve one of the following:
   - Swapping 2 characters
   - Accidently, missing a character
   - Accidently, adding an extra character

   So the first will test forms of the word with adjacent characters swapped. For example, if the word *hepl* (should be *help*), is passed, this routine will try: ehpl, hpel, and help (hah, that will match!)
   Each time it swaps characters, it should pass the modified word to *checkInDict()* to test it.
   This method is called checkSwap() and its prototype is:

   public static String[] checkSwap(String rawWord);

   This method will return an array of Strings that are possible matches (there may be more than 1). So each time checkInDict() returns true, add that modified word to an array of Strings. While working through the variations of the word passed to the routine, you can allocate a 'work' array with 10 String elements (since you do not know how many different words with swapped letters will match). It is not likely there will be more than 10 matches. If there are, skip the remaining possibilities. Before you return the array of matching word variants, determine how many words were actually added to the 'work' array, declare and allocate a new array of the correct size and copy that many elements from the work array to the return array. Return that new array of Strings.
4. Next, you will write a method that will test if a letter is missing. This will be called checkMissing () and has the following prototype:

   public static String[] checkMissing(String rawWord);

   This method will check word variations where, for each position in the word (from just before the first character through just after the last character in the word), each letter of the alphabet will be inserted and the new word will be checked against the dictionary with checkInDict().

For example, if the word *miht* (should be *might*), is passed, this routine will try each of 26 letters in each possible position (*amiht*, *maiht*,mihat, mihta, *bmiht*, *mbiht*, *mibht*,… etc.) and will find at least the word ***might*** for this example.

Also, again, collect any matches in a 'work' array of strings that is limited to 10 elements. When done, create a new array with a size that matches the number of variants found in the dictionary and copy the results to it. Return that new array.

5. The last of the 3 check routines will test if an extra letter is present. It will be called checkExtra () and has the following prototype:

   ```
   public static String[] checkExtra(String rawWord);
   ```

   It will, 1 letter at a time, delete letters from the word provided as an argument. Each time, it uses checkInDict() to see if the resulting modified word is present. If so, record it in a work array to later be returned. For example, if the word *apoply* (should be *apply*), is passed, this routine will try: *poply*, *aoply*, *apply* (this will match!), *apoly*, *apopy*, and *apopl*.

   As usual, save any match in a work array of Strings (limited to 10 elements.) When done, create an array sized according to how many matches you found. Copy the resulting words into the new return array and return that to the caller.
6. Finally, write a main method that will call splitInput() to split the string in the variable *doc* into words. You will then write a loop to step through the raw words in the list, convert the word to lower case, and call the 3 check routines: checkSwap(), checkMissing(), and checkExtra(). Each of the check routines returns an array of possible correct spellings. If there are none (all three of the returned string arrays are of zero length) or the word exactly matches, then nothing is printed for that word. If there are possible correct spellings, then print out:

   ```
   <word> is misspelled. Possible:
      <word1>
      <word2>
      Etc.
   ```

If there are no possible suggestions for corrected spelling, still note that the word is misspelled and there are no suggestions as follows:

```
<word> is misspelled. No suggestions.
```

For the sentences in the variable **doc** in `DictDoc.java`:

```
    public static String doc = "Thsi is a document tihat may hve some random mispellings. Do not overuse commas, seriously. Asking quesitons is godo lke how does this wrk. The code in the proram shoudl atch a few misspellings. do you want a obot? ";
```

your application should print the following output:

```
Thsi is misspelled. Possible:
  this
tihat is misspelled. Possible:
  that
hve is misspelled. Possible:
  have
  hive
mispellings is misspelled. Possible:
  misspellings
  misspellings
commas is misspelled. No suggestions.
Asking is misspelled. No suggestions.
quesitons is misspelled. Possible:
  questions
godo is misspelled. Possible:
  good
lke is misspelled. Possible:
  like
wrk is misspelled. Possible:
  work
proram is misspelled. Possible:
  program
shoudl is misspelled. Possible:
  should
atch is misspelled. Possible:
  catch
  hatch
  latch
  match
obot is misspelled. Possible:
  boot
  robot
```

Submit the file `Corrector.java`.

Submission Instructions

Please follow this procedure for submission:

1. Place the deliverable source file (`Corrector.java`) into a folder by itself (You may include `DictDoc.java` if you wish but do not need to since you are not changing any code in that file.) The folder's name should be CSE114_HW6_<yourname>_<yourid>. So if your name is Alice Kim and your id is 12345678, the folder should be named 'CSE114_HW6_AliceKim_12345678'

2. Compress the folder and submit the zip file.
   a. On Windows, do this by clicking the right-mouse button while hovering over the folder. Select 'Send to -> Compressed (zipped) folder'. The compressed folder will have the same name with a .zip extension. You will upload that file to the Brightspace.
   b. On Mac, move the mouse over the folder then right-click (or for single button mouse, use Control-click) and select **Compress**. There should now be a file with the same name and a .zip extension. You will upload that file to the Brightspace.
3. Navigate to the course Brightspace site. Click **Assignments** in the top navbar menu. Look under the category 'Assignments'. Click **Assignment6**.
   a. Scroll down and under **Submit Assignment**, click the **Add a File** button**.**
   b. Click **My Computer** (first item in list).
   c. Find the zip file and drag it to the 'Upload' area of the presented dialog box.
   d. Click the **Add** button in the dialog.
   e. You may write comments in the comment box at the bottom.
   f. Click **Submit**. 🡐 Be sure to do this so I or the grading TA can retrieve the submission!