CSE 1322L – Assignment 5 (Spring 2026)

Introduction

When performing a spellcheck, your computer checks if the word you typed can be found in a dictionary. If the word isn't in the dictionary, the computer will suggest a few words from the dictionary which it believes could be the word that you tried to type. There are several algorithms to determine this and, in this assignment, you will write a program which implements the "Restricted Damerau-Levenshtein Edit Distance". This will be done by using recursion to calculate the edit distance between words that the user types against words in a dictionary, suggesting the 5 words whose edit distance is the smallest.

Restricted Damerau-Levenshtein Edit Distance

In order to convert a string to another string, we allow ourselves 4 operations: insertion of a character, deletion of a character, substitution of a character, and transposition of two adjacent characters. For example:

- mats → meats: add an e
- mats → mat: remove the s
- mats → mate: replace the s with an e
- mats → mast: swap the s and the t

The examples above all have an edit distance of 1, as only one operation is needed to convert the string on the left with the string on the right. We can, however, perform as many operations as we want:

- loot → sloop: replace the t with a p, insert the s. **Edit distance of 2**
- flowers → bee: delete the s, delete the r, replace the w with an e, replace the o with a b, delete the l, delete the f. **Edit distance of 6**
- mats → master: swap the t and the s, add an e, add an r. **Edit distance of 3**

It is worth noting that transposition has a restriction: if two letters are swapped, you may no longer:

- delete either letter
- swap either letter with another letter
- insert any letter between the two letters

For example, let's say that we wish to transform the word "do" into "old". We can see that this can easily be done using one transposition and one insertion for a total of 2 operations:

do → od: swap d and o

od → old: insert l

However, because of our restrictions, no letters can be added between two characters which have been transposed. As such, we must instead perform 3 operations:

do → o: remove the d

o → ol: insert the l

ol → old: insert the d

While there are usually several combinations of the 4 operations which allow us to convert a string into another string, we are only interested in the ones that minimize the number of operations used. The Restricted Damerau-Levenshtein Edit Distance is usually described with the recursive definition below, given two strings, X and Y, and a function C which computes the edit distance:

$$C_{i,j} = min \begin{cases} 0, & if\ i = j = 0 \\ C_{i-1,j} + 1, & if\ i > 0 \\ C_{i,j-1} + 1, & if\ j > 0 \\ C_{i-1,j-1} + 1, & if\ i,j > 0\ and\ X_i \neq Y_j \\ C_{i-1,j-1} + 0, & if\ i,j > 0\ and\ X_i = Y_j \\ C_{i-2,j-2} + 1, & if\ i,j > 1\ and\ X_i = Y_{j-1}\ and\ X_{i-1} = Y_j \end{cases}$$

While the definition above is intimidating, you can find its explanation below.

- Base cases:
  - if X is empty, return the length of Y
    - Explanation: if X is empty, then to transform X into Y we need to add all the characters in Y. The number of operations will be whatever the length of Y is
  - if Y is empty, return the length of X
    - Same as above
- Recursive cases: If neither of the base cases above are true, the function C must instead **return the smallest** of the recursive cases below:
  - if X has at least 1 character, calculate **1 + C(X.substring(1), Y)**
  - if Y has at least 1 character, calculate **1 + C(X, Y.substring(1))**
  - if X and Y both have at least 1 character, and the first character of X matches the first character of Y, calculate **0 + C(X.substring(1), Y.substring(1))**
  - if X and Y both have at least 1 character, and the first character of X does not match the first character of Y, calculate **1 + C(X.substring(1), Y.substring(1))**
  - if X and Y both have at least 2 characters, and the first character of X matches the second character of Y, and the second character of X matches the first character of Y, calculate **1 + C(X.substring(2), Y.substring(2))**

As an example, suppose we wish to determine the edit distance between "barista" and "abort" using the algorithm above. The method call would look like this:

```
C("barista", "abort")
```

Given that neither string is empty, we must evaluate which recursive case returns the lowest number. Below, you can see which recursive cases are evaluated, as well as why they are evaluated:

- `1 + C("arista", "abort")`: because "barista" has at least one character
- `1 + C("barista", "bort")`: because "abort" has at least one character
- `1 + C("rista", "ort")`: because both strings have at least two characters and the "ba" in "barista" can be transposed into the "ab" in "abort"
- `1 + C("arista", "bort")`: because "barista" and "abort" have at least one character, but they do not start with the same character

Note that we do not evaluate (`0 + C("arista", "bort")`) because "barista" and "abort" do not start with the same character.

Given that we do not know what value the recursive calls above will return, we must first resolve them. For example, when evaluating `(1 + C("arista", "abort"))`, we need to evaluate the following recursive cases:

- `0 + C("rista", "bort")`: because "arista" and "abort" have at least one character, and they both share the same first character
- `1 + C("rista", "abort")`: because "arista" has at least one character
- `1 + C("arista", "bort")`: because "abort" has at least one character
- We do not evaluate (`1 + C("ista", "ort")`) because the first two characters of "arista" cannot be transposed into the first two characters of "abort"
- We do not evaluate (`1 + C("rista", "bort")`) because "arista" and "abort" start with the same character

This process must continue until we find the edit distance between the two original strings. Try to do this process by hand to see if you really understand how this algorithm works. If you do it correctly, you should arrive at an edit distance of 5.

Project Setup

Unlike previous assignments where you've had to create most or all files in your project and then submit them, that will not be the case this time. The zip file near these instructions contains all the files your project needs. If you inspect the files, you will find most of them to look familiar: they are all files from Assignment 2.

While you should already be familiar with these files, here's a brief description of each:

- **dictionary.txt**: a text file containing thousands of words, one per line
- **Dictionary.java**: A class with a single (public) static method. It loads the file above into an arraylist, which can then be used as a dictionary to do spellchecking.
- **Word.java**: A simple data class which holds 2 strings and the edit distance between them
- **Assignment5.java**: the driver. It loads a dictionary from file and then prompts the user for a sentence. If the user enters an empty sentence, the program terminates. If the user enters anything else, the program will spellcheck the words in the user's sentence.

The only file which will not look completely familiar is RecursiveSpellChecker.java. This file contains some static methods which you had to write for your Assignment 2 driver. These methods are:

- **spellCheckSentence**: spellchecks all words in a sentence, making suggestions using words in the dictionary or adding new words to the dictionary.
- **updateCandidates**: given an arraylist of Words and a Word, this method inserts the Word into the arraylist. Whenever the method returns, the arraylist is guaranteed to have 5 elements and to always be sorted in ascending order according to the edit distances of its elements. This method does this recursively, unlike the one you had to write.
- **splitOnSpaces**: You did not write this method for Assignment 2. This method is a recursive implementation of the String's split().

Feel free to inspect the methods but do not edit them. Your grader will assume that these methods are unchanged when grading your submission.

Note that 4 of the methods in this file are mostly empty:

- countSpaces
- getWord
- calculateEditDistance
- stripPunctuation

You must implement these methods to bring the project to full functionality.

Note that **3 of these methods must be implemented recursively**. While you definitely remember from your lectures how a recursive method is supposed to work, find below some rules which establish whether a method is being implemented recursively:

- The method must have at least one code path where it calls itself (recursive case)
- The method must have at least one code path where it terminates without calling itself (base case)
- The method is free to call any built-in Java methods
- The method **cannot** have loops in its body
- The method **cannot** access any static fields from any of the classes you have written
- The method is free to call any method you have written, provided the method being called also complies with the restrictions above
    - as you can see, even your instructions are recursive

Requirements

Download the zip file that is near these instructions and place all its contents (Dictionary.java, dictionary.txt, Word.java, RecursiveSpellCheker.java, and Assignment5.java) into your project's "src" folder.

Implement the methods below, which can be found in RecursiveSpellChecker.java:

private static int countSpaces(String): Returns the number of space characters (`' '`) in the argument. **This method must do this recursively**.

private static String getWord(String): Returns the next word in the argument. **This method must do this recursively**. A word, in this circumstance, is defined as all the characters from the start of the string up to but not including the first space found. See a few examples below:

Method call	Returned value
`getWord("Be Brave!")`	`"Be"`
`getWord("On your marks…")`	`"On"`
`getWord("Nothing")`	`"Nothing"`
`getWord(" boo!")`	`""`
`getWord("")`	`""`

private static int calculateEditDistance(String, String): Implements the Restricted Damerau–Levenshtein Edit Distance algorithm discussed above. **This method must do this recursively**.

private static String[] stripPunctuation(String): Given a string, if said string is a word followed by a punctuation mark, this method must separate the two. We are doing this to avoid false negatives when comparing strings to dictionary words. For example, the string `"Alice!"` and the string `"Alice,"` are supposed to represent the same word, but neither of them will be an exact match with `"Alice"`. This method will be used to fix that.

Create a string array of size 2. If the argument's last character is a punctuation mark, place the argument without its last character into the first index of the array and argument's last character into the second index. Note that, for the purpose of this assignment, only the following characters are considered punctuation:

- Periods ('.')
- Commas (',')
- Exclamation marks ('!')
- Question marks ('?')
- Colons (':')
- Semi-colons (';')

If the argument's last character ISN'T a punctuation mark, the String array will instead have the whole argument as its first element and an empty string as its second argument.

Regardless of if the argument has punctuation or not, return the array you created.

The table below shows a few examples:

String argument	Returned array
`"Alice!"`	`{"Alice", "!"}`
`"Alice,"`	`{"Alice", ","}`
`"Alice"`	`{"Alice", ""}`

This method is **not** recursive. The restrictions discussed under Project Setup do not apply.

Deliverables

- RecursiveSpellChecker.java

Considerations

- You will get partial credit for partial work, as long as the rubric permits it.
- You will have to make use of String methods to complete this assignment. You can find their documentation in the links below:
  - https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/String.html
- There is no need to submit Dictionary.java, dictionary.txt, Word.java, or Assignment5.java. Your grader already has these files.
- In Assignment 2, we only entered sentences which had no punctuation. With stripPunctuation(), that restriction has been partially lifted.
  - Still, our implementation is far from perfect. The word `"Alice"` and the word `""Alice""` ("Alice" surrounded by quotes) would be considered 2 different words as far as our program is concerned.
- The dictionary words were retrieved from the repository in the link below:
  - https://github.com/first20hours/google-10000-english
- The algorithm described in this assignment is discussed in a paper by Leonid Boystov, which you can find in the link below:
  - https://dl.acm.org/doi/10.1145/1963190.1963191

Note: Your program will run much slower compared to Assignment 2 due to the heavy use of recursion.

Sample Output (user input in red)

```
[Spell Checker]

Enter a sentence to spell-check, or nothing to quit: The quick brown fox leaps over the lazy dog.
You've entered 'The quick brown fox leaps over the lazy dog.'

'leaps' not in dictionary. Pick an option:
0. Replace with 'leads'
1. Replace with 'laos'
2. Replace with 'seas'
3. Replace with 'sleeps'
4. Replace with 'loops'
5. Add 'leaps' to dictionary
5
'leaps' added to dictionary

The final sentence is: 'The quick brown fox leaps over the lazy dog.'

Enter a sentence to spell-check, or nothing to quit: A few small steps for a man, several giant leaps for humanity!
You've entered 'A few small steps for a man, several giant leaps for humanity!'

The final sentence is: 'A few small steps for a man, several giant leaps for humanity!'

Enter a sentence to spell-check, or nothing to quit: Beter a birb in the hnad than two in the busch.
You've entered 'Beter a birb in the hnad than two in the busch.'

'beter' not in dictionary. Pick an option:
0. Replace with 'beer'
1. Replace with 'meter'
2. Replace with 'peter'
3. Replace with 'better'
4. Replace with 'bother'
5. Add 'beter' to dictionary
3
```

```
Replaced 'beter' with 'better'

'birb' not in dictionary. Pick an option:
0. Replace with 'bird'
1. Replace with 'kirk'
2. Replace with 'ira'
3. Replace with 'cir'
4. Replace with 'rb'
5. Add 'birb' to dictionary
0
Replaced 'birb' with 'bird'

'hnad' not in dictionary. Pick an option:
0. Replace with 'head'
1. Replace with 'hand'
2. Replace with 'had'
3. Replace with 'ind'
4. Replace with 'grad'
5. Add 'hnad' to dictionary
1
Replaced 'hnad' with 'hand'

'busch' not in dictionary. Pick an option:
0. Replace with 'bunch'
1. Replace with 'bush'
2. Replace with 'buses'
3. Replace with 'buck'
4. Replace with 'punch'
5. Add 'busch' to dictionary
1
Replaced 'busch' with 'bush'

The final sentence is: 'better a bird in the hand than two in the
bush.'

Enter a sentence to spell-check, or nothing to quit:
You've entered ''

Shutting off...
```