Practice Problems 2/29/2016

Part 1: Recursion

Levenshtein distance (also called **edit distance**) is the number of change/insert/delete operations that would be required to transform one string into another. This algorithm has a number of real-world uses, most notably spell checkers.

Consider the base cases:

- If the first string is empty and the second one is not, what is the distance between them?
- If the second string is empty and the first one is not, what is the distance between them?
- If both strings are empty, what is the distance between them?

Consider the recursive cases:

- What is the distance if we change a character?
- What is the distance if we insert a character?
- What is the distance if we delete a character?

The best distance overall is the smallest of these three. Consider a non-empty string. We can think of that string as **the first character**, followed by **the rest of the string**.

- "apple" = "a" + "pple"

This notion of a *(first, rest)* breakdown is very common when approaching a recursive algorithm that operates on sequences of items (in this case, characters). Notice that in some cases, the "rest" might be empty.

For the recursive case, break the string `s1` into "c1: first character of s1" + "rest of s1", and likewise the string `s2` into "c2: first character of s2" + "rest of s2". Then, the candidates for the distance between `s1` and `s2` are:

- The distance between "rest of s1" and all of `s2`, plus 1 (this would represent making `s2` by *deleting* `c1` from the beginning of `s1`)
- The distance between all of `s1` and "rest of s2", plus 1 (this would represent making `s1` by *inserting* `c2` at the beginning)
- The distance between "rest of s1" and "rest of s2", plus 1 if `c1` and `c2` are different (this would represent changing the first *character* of the string) or plus 0 if `c1` and `c2` are the same (no change needed)

The returned result is the smallest of those three recursive calls.

1. Create a new Java Project named LevenshteinDistance and create and new class called Levenshtein.
    a. Create a constructor that takes two String arguments that represent the Strings that you want to find the distance between. Store these strings as fields in your class.
    b. Write a private method *distance* that returns an integer and takes four parameters: a starting position in string 1, a length of a substring in string 1, a starting position in string 2, and a length of a substring in string 2.
    c. Implement the base cases and recursive cases described above. When computing the minimum, be aware that Java only lets you get the minimum between two values. So, the minimum of three values would be Math.min(x, Math.min(y, z))
    d. Write a public method *distance* that takes no parameters and returns an integer. Inside this method, call your recursive implementation. What values should you pass in for the starting positions and lengths to compute the distance between the two full strings that your constructor has stored in your fields?
2. Create a new test class LevenshteinTest for your Levenshtein class. Make sure that you test it thoroughly, covering all of the base cases and the recursive case. Stick to small strings, about 4-5 characters or so.
    a. clap -> cram (distance of 2)
    b. mitt -> smitten (distance of 3)
    c. start -> cart (distance of 2)