Data Structures and Algorithm Design
CSCI 340
Spring 2019
Programming Project 6
Plagiarism Checker

Due Date: Thursday, May 9
Points: 50

Objective:
The student will write a Java program to check source code for possible plagiarism.

The textbook discusses dynamic programming and the longest common subsequence problem on pages 598-604.

Source Code Plagiarism
Consider the problem of detecting source code plagiarism in a class of students. If two students turn in completely identical programs, is it obvious that they colluded. CS students are smarter than that, however. Modern IDEs and editors allow students to do global replaces on variable names in order to make the code less similar.

The goal of this assignment is two develop a Java program that will read in two source files, compare them for similarity and publish a score on how similar they are.

Longest Common Subsequences
Consider the following code sequences:

```
while(i<10){
```
and
```
while (i<=9) {
```

We cannot help but notice that they are very similar.
The only differences are:

- a space before the `(` in the second ,
- `10` becomes `=9`,
- an additional space before the { in the second string.

A ***subsequence*** is any sequence of characters from a string. The word sequence implies that we take the characters in order, but that we can skip characters, too. For example, the first string above contains the subsequence `whl(i10{`

A ***common subsequence*** is a subsequence that appears in both strings. The above example gives a good intuitive explanation of what we mean by a common subsequence. Both strings contain `while`, so it is a common subsequence. Both strings contain `(` and `)` so `()` is a common subsequence.

A ***longest common subsequence (lcs)*** is a common subsequence with maximal length. In the above strings it is `while(i<){`. It has length 10.
At the heart of this assignment is writing a method that finds the length of a longest common subsequence of two strings.

An Efficient Algorithm to Find LCS Length

This section of the discussion borrows heavily from Cormen, *An Introduction to Algorithms, 2nd ed*, MIT Press, 2001.

A brute force algorithm to find lcs length is fairly easy to write, just take all common subsequences and returns the maximum length. The problem is that there are an exponential number of common subsequences.

A greedy approach fails, because it is impossible to tell if we should be skipping characters in one string or the other. Maybe we want to skip some characters in both.

This brings us to dynamic programming. Finding an LCS has an optimal substructure property.
Let X = <x1, x_2, … x_m> and Y = <y_1, y_2, … y_n>
Let Z = any LCS of X and Y = <z_1, z_2, … z_k >

1. If $x_m = y_n$ then $z_k = x_m = y_n$ and
   Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. If $x_m <> y_n$ and $z_k <> x_m$ then
   Z is an LCS of X_{m-1} and Y
3. If $x_m <> y_n$ and $z_k <> y_n$ then
   Z is an LCS of X and Y_{n-1}

A recursive formulation for lcs length follows from this structure:

$$c[i,j]=\begin{cases}0 & if\ i=0\ or\ j=0\\ c[i-1,j-1]+1 & if\ i,j>0\ and\ x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & if\ i,j>0\ and\ x_i\neq y_j\end{cases}$$

where c[i,j] is the lcs length. This gives us a recursive formulation that can be implemented in just a few lines of code. Unfortunately, the recursive implementation of this formulation is still exponential in complexity.

The above formulation uses array like notation. That is, we use square braces to surround our i's and j's. We can fill the array from the top down, left to right, and we will always have the values we need to calculate the current value. The final answer will be in the lower right corner of the array. This reduces the complexity to O(m*n), where m and n are the lengths of the two strings.

i/j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Here is an example matching two strings. The arrows in the cells tell us which of the neighboring cells were used to calculate this particular value. When an arrow is to the upper left, we have a character match and increase the value by 1. Note that the found longest common subsequence is BCBA, which has length 4, and is shown in red above. It is not unique. BCAB and BDAB are also longest common subsequences.

Space Considerations

The c array can get large in a hurry. It is also O(m*n). Consider checking two 1,000 character programs. The c array will be 1,000 x 1,000, or contain a total of 1,000,000 entries. An Integer in Java is 4 bytes, so the array will contain 4 Megabytes. A 1,000 character program is well less than a page in length, but still requires a huge data structure.

Here is the trick to save space. We really don't need the lcs. We only need the `lcsLength`. That is, we want the entry in the lower right of the table, 4 in the above example, not the BCBA. Because of this, we only need to maintain enough of the array to calculate the `lcsLength` entries. This turns out to be the current row and the previous row. Create your array with only two rows. When filling what would be an even numbered row, assume row 1 is the previous row. When filling what would be an odd numbered row, assume row 0 is the previous row. This reduces the space needed to O(2n).

Plagiarism Score

`lcsLength` is not a good metric for potential plagiarism because it depends on the length of the two input strings. The longer the strings, the higher the expected value of `lcsLength`. We will need to normalize our plagiarism score as:

```
plagiarismScore = 200*lcsLength /(m + n)
```

Using this metric, a score of 100 means that the files were completely identical, while a score of 0 means that the files have absolutely no characters in common.

Your Assignment

Write a Java class named PlagiarismChecker that contains the following methods:

`public int lcsLength(String prog1, String prog2)`

This method should calculate the lcs_length of the two programs.

`public double plagiarismScore(String filename1, String filename2)`

This method should use `lcsLength()` to calculate the plagiarism score for the two files. Note: when reading in the files, you should keep the comments, carriage returns and all other characters. You want to create two long strings that are exactly the characters in each file. Do not use a Scanner, as it will abandon the carriage returns. See the sample code from earlier in the semester that used `RandomAccessFiles`.

`public void plagiarismChecker (String[] filenames, double threshold)`

This method should do a pairwise comparison of all the files in the array and print a neatly formatted report listing any suspicious pairs. The report should include their plagiarism score. For purposes of this assignment, any pair of programs is suspicious if the plagiarism score is greater than or equal to the specified threshold.

On the lab machines in the course directory `/home/student/Classes/Cs340`, there is a subdirectory named `Plagiarism`, that contains anonymized student code files that you can run your checker on.

Grading Considerations

Only electronic submission is required.
Good software engineering, e.g. appropriate documentation and lots of comments, is expected.
Your instructor will use your programs to check all submissions for plagiarism. (In other words, you are writing a program that will grade itself!)