Design and Analysis of Algorithms: Homework #5

Due on ICON, at 11:59pm, on April 12, 2018

Professor Kasturi Varadarajan

Heather Kemp

Problem 1

We are given two strings, called the *source* and the *target*, which both have the same number of characters *n*. The goal is to transform the target string into the source string using the *smallest* number of allowed operations. There are two types of allowed operations, a *flip* and a *substitution*.

- A substitution replaces a character in a certain position by another character. For example *eetie* is transformed to *eerie* via a substitution of the third character.
- A flip reverses a contiguous substring of the string. In particular, flip(i,j) reverses the substring starting at the i-th character and ending at the j-th character. For example, *eteie* is transformed to *eetie* via flip(2,3).

We will assume that the portions where any two flips are applied don't overlap. That is, if we perform flip(i,j) and flip(k,l), then the sets {i,i+1,...,j} and {k,k+1,...,l} are disjoint.
Consider the source string timeflieslikeanarrow and the target string tfemiliilzejeworrbna. The following sequence transforms the target into the source:

1. *tfemiliilzejeworrbna* (the target)
2. *tfemiliilzejeanbrrow* (a flip)
3. *tfemiliezlijeanbrrow* (a flip)
4. *timefliezlijeanbrrow* (a flip)
5. *timefliezlijeanarrow* (a substitution)
6. *timefliezlikeanarrow* (a substitution)
7. *timeflieslikeanarrow* (a substitution)

The number of operations used, which is the quantity we seek to minimize, was *6*. Note that we never need more operations than the length of the given strings.
Write a time-efficient program that takes as input a file containing the two input strings and prints out the optimal sequence of transformations from the target to the source. You can assume that the file is formatted like the following example. Each string will be a sequence of lower-case English characters.

```
1: timeflieslikeanarrow
2: tfemiliilzejeworrbna
```

Solution

Algorithm Description and Analysis

This algorithm approaches the problem by working backwards from the target to arrive at the source, working to make one letter 'in place' with each iteration.

For this to occur, there are two cases which we must consider. The first is that we run into a letter that is already 'in place', meaning it matches the letter from the destination string in that same position. In such a case, all we need to do is 'ignore' this letter by only referring to its transformation time as the time it took to transform everything after it. Thus, this case is $O(1)$, as we are referring to already calculated values. The case of non-matching values is

a little more interesting, as we have the two options given in the project specification.

The first way of handling 'out of place' characters is the easiest, which is substituting this value with the correct value. This is the easiest way of handling it, as we only perform one operation, and we only affect our current transformation, so none of our prior computations change, meaning we can use them for further computations. That is, similarly to finding elements that already match, replacing a character in a string is at most an $O(n)$ operation, in the case of using a language which requires a copy of the string to be created for mutating it, where n is the size of the string. In cases where the string can be mutated in place, this would be an $O(1)$ operation.

The more complicated character placement handling is the flip reverse. It's complicated because we don't know what flip is most optimal. For example, going from $abcdef$ to $acfedb$, we can flip b through f, but this will require 5 operations $[reverse(bcdef), replace(f, c), replace(e, f), replace(d, e), replace(e, d)]$ while reversing c through f will require 3 operations $[reverse(cdef), replace(b, c), replace(c, b)]$. As shown in these examples, though, a swap doesn't just consist of that swap, it also leads to replacement operations inside of them to make sure the values are now in place. We know we can only substitute these values by the specification that we cannot flip elements in the same section twice, and as a result, only consider the operations done in the initial flip and the subsequent replacement calls as the total amount of calls for that transformation. For the reverse function itself, though, which is called by the outer loop which handles the cycling of these different forms, only $O(n)$ operations are done, as swapping values requires an iteration over the n character that make up a string and rebuilding the string in reverse order, which can be done in $O(1)$ time.

We then want to record the results of this source and destination pair, and so we store the smallest of our resulting moves as the result for this iteration, and proceed with these methods until we reach the end of the string, at which point we can just compare all of the previous minimum values we encountered to see which was truly the most efficient. This can be done in $O(n)$ time, as we pick one value for each of n 'rounds' of the algorithm and simply loop through them.

The program as a whole, has two very clear for loops, the second of which is for the reverse substring calculations, which go from n to 0 in their longest state, meaning without accounting for the runtime of our three different cases, our base program is $O(n^2)$. Alternatively put, to iterate over all possible reversals from the end of the string back to the beginning, we have n^2 possible combinations, as dictated by the commonly known handshake problem's solution. As we discussed in explaining the three different parts of our program, however, running into the correct character is $O(1)$, swapping a character is $O(n)$ and reversing a substring is $O(n)$ time, meaning that as we run reverse, a $O(n)$ operation, n^2 times, our entire algorithm faces a runtime of $O(n^3)$.

While the steps are returned in reverse order, from the source to the target, it can be assume that the user can trivially reverse the order of the steps to arrive at the order of steps to go from the target to the source.

During testing on a Macbook Pro, two randomly generated strings of length 500 performed at around 49 seconds, while two randomly generated strings of length 1000 performed at around 7 minutes and 29 seconds. This recorded runtime is of course dependent on the hardware the tester is using.

Running the Program

This program was written in Java in the Eclipse IDE for Java Developers, Neon.1 Release. It can be run from this IDE or from the terminal, both of which will be described here.

To run this in the Terminal, navigate to the project base folder downloaded from ICON, and once you are in the base folder, run `java -cp src transformMain`. If the class file did not transfer, you may need to run `javac`

`transformMain` in the src folder to re-compile the program. A standard input prompt should appear on your console looking for a text file.

To run this in Eclipse, on the toolbar, select `File > Open Projects from File System > Directory` and then navigate to the AlgorithmsHW5 base folder downloaded from ICON. After having this folder, verify that AlgorithmsHW5 is checked on the import screen and click `Finish`. Once the file is imported, locate the project folder on the package explorer task bar (usually on the lefthand side of the screen) and right click the project. Then select `Run As > Java Application`. Then, in the Eclipse console, a prompt will appear asking for a text file.

If it is in the same folder as this project, simply `test.txt` will work, which is the example text file given. Otherwise, an absolute path works, as the Java `File` and `Scanner` objects are used for this code.
Examples of the execution of this program are included in `TestRuns.txt`.