# Homework 2, CSCE 350, Spring 2016
# Due 3 March 2016

We have talked a little about string matching, and we have sketched out a naive string matching algorithm.

This assignment is to do at least a first pass at determining how bad (or how good?) the naive algorithm is on ordinary text data.

You have been given a somewhat cleaned up version of *The Tale of Two Cities*. This is from the Gutenberg project, and I have left in the beginning and ending text because Gutenberg insists on getting credit for their text. I have lower cased everything and removed nearly all the punctuation.

Remark: I have also split the text into individual words and put one word per line. Your code will have to rebuild this into one continuous string of characters, with blank spaces in between the words. I broke it into one word per line so you could more easily do subsets of the full text for testing purposes.

I have also provided 20 target patterns, each of three words, for you to search for. These are roughly the same length. Based on what I am seeing in my analysis, the results are not in fact very sensitive to the length of the target pattern.

I have also provided the frequency counts of the letters in the text.

Your assignment is to write a naive string matching algorithm and to run it on each of the target patterns, counting the number of comparisons of characters.

You are either in a separate document (which is probably no more than two pages, and more likely only one), or as part of the documentation of your actual code, to do a little analysis of your results. If you do a separate document, then just make sure to include it in the zip file that you upload to Moodle. A document in either plain ASCII text or pdf is preferred.

For example: Let's say you have 100,000 characters in the source text, and let's say that 10,000 of them are the letter "a". If you have a target pattern of 15 characters, perhaps "a target string", that starts with the letter "a", then you will have to do at least 99,985 comparisons. You will have 10,000 of them come up with a match on the first character. (Ok, maybe a small bit less than that if there are letters "a" in the last 14 characters and you don't check those.) This cost is fixed almost regardless of what algorithm you are running, so this is a cost you ignore. What you are more interested

in is the number of additional comparisons you make. That is, if the pattern appears only once in the text, and you search the entire text, what's the cost of failing to find the pattern?

You should keep a count of the number of comparisons you make IN EXCESS of the 99,985. That's the real cost of the algorithm.

Actually, it's a little more complicated than that. If there are 10,000 letters "a", then you will get an initial hit for each of those 10,000 times, and you will have to test the second letter in the pattern at least once for every hit on the first letter. That is, you can't get away from paying a cost equal to the number of occurrences in the full text of the first letter of the target pattern. If you get a hit on the letter "a" and then on the second letter (which in this case is a blank space), then you test the third letter.

So we normalize the comparison count by the frequency of occurrence of the first letter of the target. The real cost of the algorithm is

$$\text{number of comparisons after the initial hit}/freq(letter)$$

where $freq(letter)$ is the frequency of the letter $letter$ in the entire document. If in the example we are using you NEVER have the letter "a" followed by a blank space, then you would make exactly 10,000 comparisons, they would all fail, and this quotient would be 1.0. If you had 500 instances of "a ", but no instances of "a t", then you would have 10,500 comparisons and the quotient would be 1.05.

Remark: Note that the statistics and conclusions might very well be different if one ran the experiment on a different kind of string data (DNA, for example).