Homework 1, CSCE 350, Fall 2016
Due 8 September 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.

Source data

You have been given four data files. The big one is 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.

The second is a random string of the letters A, C, G, T, which would be DNA. This is random except that I have weighted the occurrence of the letters so they are not equally likely.

The fourth is the string that is the first 20 thousand or so numeric digits of pi.

Tests

I have also provided some test values for each of the sample data files.

Your assignment

One of the issues in designing (or using) an algorithm on a real problem is to decide how complicated the algorithm needs to be. That very often depends on the characteristics of the input data and of the actual problem.

That is, you might well expect string matching on a random string of four characters (DNA) to behave different from string matching on a random-appearing string of ten numeric digits, and both of these might well be different from text searching in human languages in which words are highly nonrandom and there is enormous redundancy (consider the frequency of the trigram “the” in English).

I have given you tables of the frequency of occurrence of the individual characters in each of the three data sets. You may use these and simply read them in.

Remark–MINOR ANNOYANCE: The Dickens text is split into individual words with one word per line. The other two files are also split for convenience. Your code will have to rebuild this into one continuous string of characters, with blank spaces in between the "words" that form each line.

This would normally be a hassle because you would have to have one program that did text with spaces between words and a different program that didn't use spaces, for the DNA or the pi strings. HOWEVER, I have put in test values for DNA and for pi that include a blank space, and you do not have to worry about matching across blank spaces unless the space is in the test. That is, if the test is "ACACAC" and there happens to be a piece of the DNA that reads "ACA CAC" because it happens to break for a space halfway through, you do not have to report that that is in there.

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 also to write a separate document (which is probably no more than two pages) to do a little analysis of your results. Make sure this separate document is included 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 (additional) 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.