Sequences and Pattern Matching

In this laboratory, we study the problem of pattern matching in sequences of data. This can be viewed as a combinatorial problem, in which we are searching in a deterministic sequence of characters, or it can be viewed as a probabilistic problem, in which we are searching in a sequence of characters that was randomly generated.

For consistency, in either case, we write \( X = X_1X_2X_3 \ldots \) where the \( X_j \)'s are either a randomly generated sequence of random variables or a deterministic sequence of characters, however you prefer to look at it. Most of the strategies described here will work in either case. At least, for now, we do not describe the probability model. (In the Thursday lab, we will think about probability models for randomly generated strings.) In this laboratory, we use “strings” or “words” or “sequences” almost interchangeably, to denote a collection of characters in which the order matters.

(1) Write a function that takes two inputs: (a) long sequence of data, and (b) relatively short sequence of data (compared to the longer sequence), and it gives (as a result) either “true” or “1” (or whatever you like) if the longer sequence contains the shorter sequence, or gives “false” or “0” (or something similar) if the longer sequence does not contain the shorter sequence. [If you are using a language that contains RegExps (“regular expressions”), you probably should avoid them for today, because we want to think about how to manually write the underlying algorithms that are used for string searching.]

For example, if the shorter string is:
“aababaaab”
and the longer string is:
“ababababbababaaabbabababababababababaaaaabababab”
Then the output should be “true” or “1” because we see the following:
“aababababbababaaabbabababababababababaaaaabababab”

On the other hand, if the shorter string is
“bbaa”
and the longer string is:
“abababbbabaaaa”
Then the output should be “false” or “0” because the longer string does not contain any occurrences of the shorter string.

(2) Write a function with the same kinds of inputs as in question (1) above, but the output should be different this time: The output should contain the number of (possibly-overlapping) occurrences of the shorter string that are found in the longer string.

For example, if the shorter string is: “abababa”
and the longer string is: “abababababaaabbababababababababaaaaabababaaab”
Then we have
abababababababaaabbabababababababababababaaaaaabababaaab
abababababababaaabbabababababababababababaaaaaabababaaab
abababababababaaabbabababababababababababaaaaaabababaaab
abababababababaaabbabababababababababababaaaaaabababaaab
abababababababaaabbabababababababababababaaaaaabababaaab
abababababababaaabbabababababababababababaaaaaabababaaab
So the output should be 5, because the shorter string occurs exactly 5 times in the longer string.
(3) If you used a brute-force algorithm to write the function above, sometimes things can go very badly. There are lots and lots of terrible examples, but here is one of the most simple to study, since we only have a relatively short lab.

Consider, for instance, the following example:

Let the shorter string consist of 99998 a’s in a row, followed by 2 b’s.
Let the longer string consist of 1 million a’s in a row.

Using these two words, run your algorithm. You could always start with an easier case, such as:

Let the shorter string consist of 98 a’s in a row, followed by 2 b’s.
Let the longer string consist of 1 thousand a’s in a row.

Once your algorithm is working well, try and run your algorithm on when the big string and the small string and both much larger, e.g., $10^6$ characters in the short one and $10^9$ characters in the long one. Try to make your machine get stuck.

You will need two “for loops” for such a brute-force algorithm: The outer “for loop” will scroll through the possible starting locations in the longer string; the inner “for loop” will scroll through the characters of the shorter word until a match is discovered (i.e., the shorter word gets completely scanned) or the match is found to fail.

Why does the algorithm perform so poorly? If we call the shorter word $w$, and the longer word $v$, then we see a brute-force algorithm is scanning the full length of $w$ many, many times within $v$. A brute-force algorithm (in which $w$ is 99998 a’s followed by 2 b’s) will try to match $w$ at each possible starting location in $v$, and it will take 99999 steps (i.e., until reaching the first b in the shorter word $v$) before the algorithm finds a difficulty. Then it will start over with the next possible location of a match.

A brute-force algorithm does not capture any of the autocorrelation the shorter word. The shorter word often has lots of “structure”, and good string matching algorithms take advantage of this structure! Once a search is made for the smaller word, the algorithm would run better if the 2nd, 3rd, 4th, etc., searches could somehow take advantage of what happened on the earlier search!

There are many truly excellent papers and books about all kinds of great pattern-matching algorithms. Some of them, for instance, are:

* Algorithms On Strings, Trees, and Sequences* by Gusfield (Cambridge, 1997);
* Algorithms On Strings* by Crochemore, Hancart, and Lecroq (Cambridge, 2007);
* Flexible Pattern Matching in Strings* by Navarro and Raffinot (Cambridge, 2002)
(4) The tricks that various families of algorithms use—to take advantage of the structure of the smaller string—are really amazing and quite diverse! One of the most famous, earliest examples of taking advantage of the string structure is the Knuth-Morris-Pratt algorithm. This algorithm is beautifully described by David Eppstein here:

http://www.ics.uci.edu/~eppstein/161/960227.html

Eppstein describes the difficulty that we pointed out about brute-force algorithms in questions (1) through (3). Then he gives two versions of the Knuth-Morris-Pratt algorithm. (There are dozens, maybe hundreds?, of variants of the KMP algorithm in the literature.)

Eppstein’s “KMP Version 1” fundamentally relies on the match of the inner string with itself, in the line

\[ \text{overlap}(P[0..j-1], P[0..m]) \]

In Eppstein’s “KMP Version 2,” everything is rolled together, and things fundamentally rely on the line

\[ j = \text{overlap}[j] \]

At the end of Eppstein’s page, he discusses a little more in-depth the runtime of KMP. Indeed, if the shorter string has length \( m \), and the outer string has length \( n \), it is possible to compute the overlaps cleverly in time proportional to \( m \), and then to run the entire algorithm in time proportional to \( n \), so that the entire algorithm runs in time proportional to \( m + n \). (The brute force algorithm, as you probably noticed already, can take times up to \( mn \) to finish.)

Implement the KMP algorithm. Try it on some strings for which the brute-force algorithm fails (such as the examples given above).

(5) Now that you have the KMP algorithm implemented, let’s view the longer word as randomly-generated and think about the number of times that a fixed shorter word will occur in the larger, randomly-generated word.

Let \( w \) be a fixed word. Let \( X_1 X_2 \ldots X_n \) be a randomly generated word of length \( n \), where “randomly generated” is free to your interpretation here. For instance, the letters might be independent of each other, or consecutive letters might affect each other, e.g., one letter influences which one comes next (we will study such dependence in Thursday’s laboratory this week).

Again, with \( w \) as a fixed word, let \( a_n \) denote the approximate probability that a word of length \( n \) (distributed according to whichever distribution that you selected) contains \textit{at least 3 occurrences} of \( w \). Approximate the values of \( a_n \) using random generation. For most distributions you might think about (i.e., for independent, identically distributed models), you will see that \( a_n \) should be an increasing function of \( n \), i.e., \( a_n \) should increase from 0 to 1 as \( n \) grows large. Make a visualization of this phenomenon for your example word \( w \).

(6) Now let \( w \) be fixed again, but let \( b_n \) denote the probability of \textit{exactly 3 occurrences} of \( w \) in the randomly generated word. For your probability model, where is \( b_n \) maximized? Notice that \( b_n \) will roughly increase to a maximum and then decrease back to 0, for many probability models.

(7) At this point, you should be well-prepared to make your own preliminary investigations about heuristics for pattern searching in strings. Design some problems of your own!