CS375, Spring 2014
Bonus Hmwk 1 (up tp 100pts to be added to your homework score)
due: March 30, 23:59 EST.

Design and implement an algorithm for string matching. Assume the alphabet $\{G, A, T, C\}$. Use the method discussed in class (also, see the notes on the next page).

The algorithm has two key parts. First, given a search string, you need to construct the table representation of the appropriate DFA. Second, you have to implement a method that runs the string in which you search ("text") through this automaton.

To test your algorithm, generate a random 100-letter long string $w$ over the alphabet. Search for strings $w_3, w_4, \ldots, w_{10}$, where $w_i$ is the string of the last $i$ letters in $w$. for each $i = 3, \ldots, 10$, show the results, highlighting all found instances of the string.

A major part of the project is to construct a DFA that recognizes strings containing the pattern string. Show the DFA that you algorithm constructs for the string $GACGA$ and for the string $AAGCATTTAAGCA$.

Repeat 10 times: randomly generate a 100,000-letter long string $w$ over the alphabet (each letter shows at a position with the same probability). Search for the string $w_{50}$ consisting of the last 50 letters in the string. Report the time of the average time of search.

Repeat the same another 10 times, this time generating sequences of the length 200,000. Compare times. Is the behavior of your algorithm consistent with the asymptotic analysis of the coplexity of the string matching algorithm?

**Unless you obtain a permission from the grader, you must develop your project in C++. A single zip file must be submitted via the csportal. It must contain the source code, results of your experiments, discussion, and all other documentation.**

## String Matching

We are working with a very long text (a string $T$ over some alphabet $\Sigma$). It is often the case that we want to find out whether that string contains another string, say $S$ (for "search" string), where $S$ is typically much shorter than $T$.

To make our discussion concrete, let us assume that we are working with strongs over the alphabet $\Sigma = \{a, b, c\}$. A generalization to alphabets of arbitrary finite size is straightforward.

We will assume that $T = t_1 t_2 \dots t_n$, $S = s_1 s_2 \dots s_m$ and $n \geq m$.

How to determine whether $S$ is a substring of $T$? And, if $S$ is in fact a substring of $T$, how to find one occurrence (or all occurrences) of $S$ in $T$.

A brute force method works like this: for every $i = 1, 2, \dots, n$, compare strings $S$ and $t_i t_{i+1} \dots t_{i+m-1}$, and return all $i$ for which they are equal. The running time is $O(nm)$. Can we do better?

Yes. If we put DFAs to work. Let us try to consruct an DFA $M$ that accepts precisely those strings in $\Sigma^*$ that contain $S$ (this language is evidently regular). It is not really that hard. To make our discussion concrete, we will work with the alphabet $\Sigma = \{a, b, c\}$. A generalization to alphabets of arbitrary finite size is straightforward. Let us consider a search string $S = abcacabca$. Its length is 9 and so we will need 10 states in $M$, which we will name $0, 1, \dots, 9$. The intuition is this: each state remembers the largest number of last consumed letters that form a prefix of $S$. If we are in state $i$, the last $i$ letters read form a prefix of $S$ but the last $i + 1$ letters read do not.

The automaton can be visualized as a sequence of states $0, 1, \dots, 9$, with the transition from $i$ to $i + 1$, $i = 0, 1, \dots m - 1$, labeled with the letter $i + 1$ of the string $S$. Say the transition from $i$ to $i + 1$ is labeled with $a$. Where do we go if we are in the state $i$ but see $b$? This letter breaks our march towards the last state (where we know we found $S$). It moves us backwards to the closest state $i'$ that has the property that the last $i'$ letters read form a prefix of $S$.

Once we have the automaton, finding all occurrences of $S$ in $T$ takes time proportional to $n$ (assuming the size of the alphabet is fixed and independent of the size of $T$). To construct the automaton, it takes time $O(m^3)$ (again under the assumption that the size of the alphabet is fixed and independent of the size of $S$). Indeed, if we are in state $i$, the last $i$ letters read for the prefix $S_i$ of $S$. Say the next letter read is $a$ and $S_i a$ is not a prefix of $S$. We have to find the longest prefix of $S$, say $S_j$ that is a suffix of $S_i a$. Brute force allows us to do it in $O(m^2)$ time. There are up to $m$ "alignments" to try and each alignent takes $O(m)$ steps to verify.