**Pattern matching** (also referred to as string matching) is the act of checking a given **sequence** of tokens for the presence of the constituents of some pattern. There are different algorithms. The main goal to design these types of algorithms is to reduce the time complexity. The traditional approach may take a lot of time to complete the pattern searching task for a longer text but with the help of finite automata, it can be achieved in a much shorter time.

The string-matching automaton is a very useful tool which is used in string matching algorithm. It examines every character in the text exactly once and reports all the valid shifts in O (n) time. The goal of string matching is to find the location of specific text pattern within the larger body of text (a sentence, a paragraph, a book, etc.)

**The basic idea is to build an automaton in which**

* Each character in the pattern has a state.
* Each match sends the automaton into a new state.
* If all the characters in the pattern has been matched, the automaton enters the accepting state.
* Otherwise, the automaton will return to a suitable state according to the current state and the input character such that this returned state reflects the maximum advantage we can take from the previous matching.
* the matching takes O(n) time since each character is examined once.

Suppose we want to "grep nano". Rather than just starting to write states down, let's think about what we want them to mean. At each step, we want to store in the current state the information we need about the string seen so far. Say the string seen so far is "...stuvwxy", then we need to know two things:

1. Have we already matched the string we're looking for ("nano")?
2. If not, could we possibly be in the middle of a match?

If we're in the middle of a match, we need to know how much of "nano" we've already seen. Also, depending on the characters we haven't seen yet, there may be more than one match that we could be in the middle of -- for instance if we've just seen "...nan", then we have different matches if the next characters are "o..." or if they're "ano...". But let's be optimistic, and only remember the longest partial match.

So, we want our states to be partial matches to the pattern. The possible partial matches to "nano" are "", "n", "na", "nan", or (the complete match) "nano" itself. In other words, they're just the *prefixes* of the string. In general, if the pattern has m characters, we need m+1 states; here m=4 and there are five states.

The start and accept states are obvious: they are just the 0- and m-character prefixes. So the only thing we need to decide is what the transition table should look like. If we've just seen "...nan", and see another character "x", what state should we go to? Clearly, if x is the next character in the match (here "o"), we should go to the next longer prefix (here "nano"). And clearly, once we've seen a complete match, we just stay in that state. But suppose we see a different character, such as "a"? That means that the string so far looks like "...nana". The longest partial match we could be in is just "na". So from state "nan", we should draw an arrow labeled "a" to state "na". Note that "na" is a prefix of "nano" (so it's a state) and a *suffix* of "nana" (so it's a partial match consistent with what we've just seen).

In general the transition from state+character to state is the longest string that's simultanously a prefix of the original pattern and a suffix of the state+character we've just seen. This is enough to tell us what all the transitions should be. If we're looking for pattern "nano", the transition table would be

n a o other

--- --- --- ---

empty: "n" empty empty empty

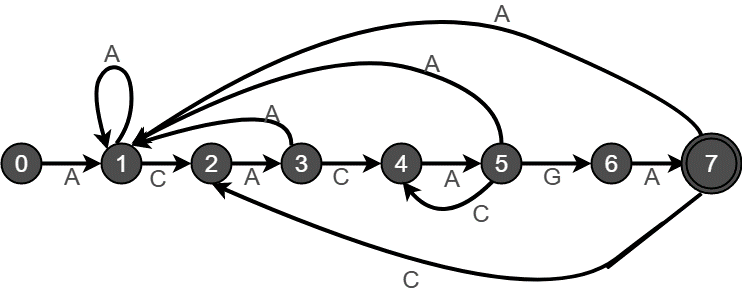
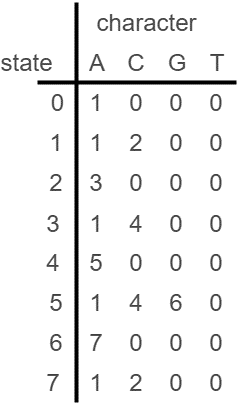
"n": "n" "na" empty empty

"na": "nan" empty empty empty

"nan": "n" "na" "nano" empty

"nano": "nano" "nano" "nano" "nano"

Simulating this on the string "banananona", we get the sequence of states empty, empty, empty, "n", "na", "nan", "na", "nan", "nano", "nano", "nano". Since we end in state "nano", this string contains "nano" in it somewhere. By paying more careful attention to when we first entered state "nano", we can tell exactly where it occurs; it is also possible to modify the machine slightly and find all occurrences of the substring rather than just the first occurrence.



**Input:**

* Text (in which pattern has to be searched)
* Pattern (that has to be searched in text)

**Output:**

* Number of occurrences of pattern
* Locations of occurrences of patterns

**Further, program will also allow some** **operations to be performed**:

* Replace the pattern with some other string in the original text.
* Delete the pattern from the original text.

These operations can be performed on all the patterns matched or just on a pattern at a specific location as inputted by the user.

**Flow of program:**

* Firstly, the user will input a pattern.
* A Finite Automata will be constructed which accepts the pattern as a substring.
* Now, User will input some text in which the pattern needs to be searched.
* The text will be passed through the Finite Automata to check for the substring (pattern) presence.
* All the occurrences of the pattern will be displayed to the user
* Now, user will be able to perform operations on the matched patterns (operations such as replace or delete)
* User will also be able to choose whether to perform operation on all the occurrences of the pattern or only on a specific pattern.
* Finally, the modified text would be displayed.