String Matching

The problem of String Matching

Given a string ‘S’, the problem of string matching deals with finding whether a pattern ‘p’ occurs in ‘S’ and if ‘p’ does occur then returning position in ‘S’ where ‘p’ occurs.

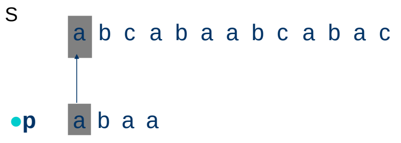
How does the O(mn) approach work

Below is an illustration of how the previously described O(mn) approach works.

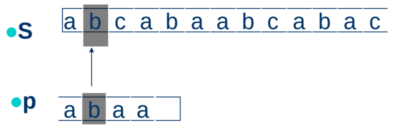
String S a b c a b a a b c a b a c

●**Pattern p** a b a a

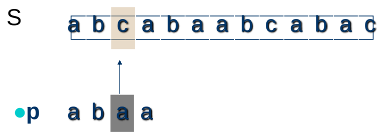
Step 1:compare p[1] with S[1]



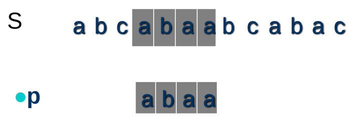
●**Step 2: compare p[2] with S[2]**



Step 3: compare p[3] with S[3]



●*Mismatch occurs here..*



●**Finally, a match would be found after shifting ‘p’ three times to the right side.**

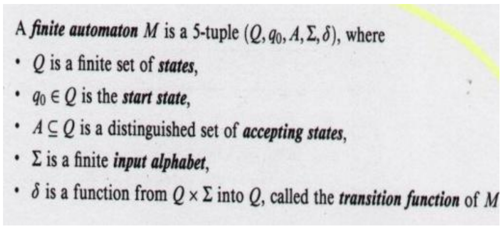
●**<u>Drawbacks of this approach:</u> if ‘m’ is the length of pattern ‘p’ and ‘n’ the length of string ‘S’, the matching time is of the order O(mn). This is a certainly a very slow running algorithm.**

●

A Finite Automaton Approach

- A directed graph that allows self-loop.
- Each vertex denotes a state and each directed edge is labeled by an input item.
- The directed edge stands for a transition from one state to another.
  - E.g., a directed edge labeled *a* connecting vertex v to vertex w, means that if we are at state v and see the input item *a*, then we should **transit to state w**.
    We write δ(v,*a*) = w.
    Where, δ is called the state transition function.
- There is a unique start state and a unique final state.

Finite Automata


A ***finite automaton*** M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M

●source: 91.503 textbook Cormen et

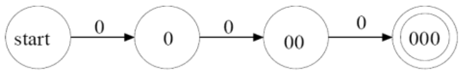
- Sometimes, an edge is labeled with **more than one input item**. This means we will make the **same transition** if we see any one of the input items indicated.
- We can think of the **finite automaton as a machine**. So the operation begins at the start state. When it reaches the final state, it stops running.

Pattern Matching Automaton

- Other than the start state, the label of each vertex is the prefix of the pattern that has been successfully matched so far.
- The final state stands for a **complete match.**
- The label of an edge is the next character in the input text.

Automaton Construction example

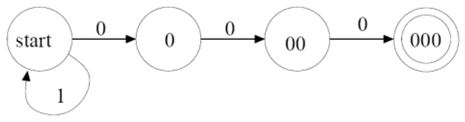
- A partially defined automaton for matching the pattern P = 000 looks like:



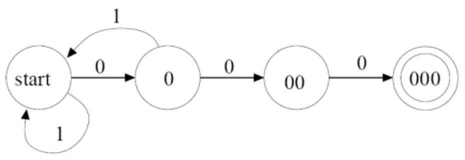
- How should we define the remaining transitions?

$\delta(\text{start}, 1)$, $\delta(0, 1)$, and $\delta(00, 1)$.

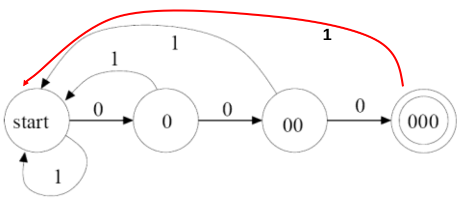
- If we are at the start state and we see 1, then we have no progress at all because we do not have any match with any character in P. So δ(start,1) = start



- If we are at state 0 and we see 1, then we have to start matching all over again at the character after 1 in the input text. So δ(0,1) =start.

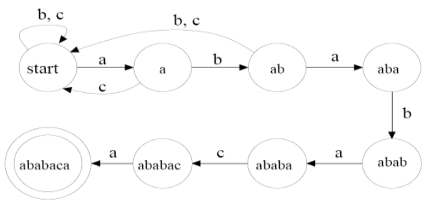


- If we are at state 00 and we see 1, then we have to start the matching all over again at the character after 1 in the input text. So δ(00,1) =start.
- The same reasoning works for the state 000 as well.



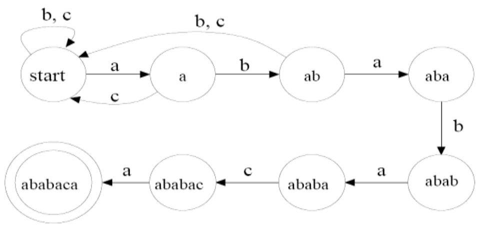
A more interesting example

- Construct an automaton for matching the pattern P = ababaca
- We start with the following partially defined automaton.



It remains to define $\delta(a,a)$, $\delta(aba,a)$, $\delta(aba,c)$, $\delta(abab,b)$, $\delta(abab,c)$, $\delta(ababa,a)$, $\delta(ababa,b)$, $\delta(ababac,b)$, and $\delta(ababac,c)$.

- When we are at state a and see a, we should go to the state a. So δ(a,a) = a.
- When we are at state aba and see c, we should go to the start state. So δ(aba,c) = start.
- But when we are at state aba and see an a, we should go to the state a
- When we are at state abab and see b or c, we should go to the state start. So δ(abab.b) = δ(abab.c) = start.

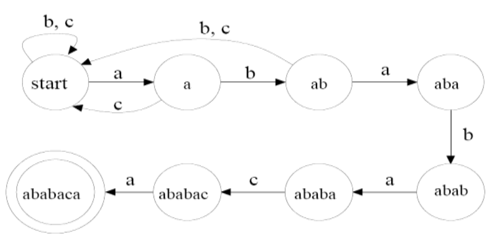


- Suppose that we use s_k to denote the state which is the prefix of P with k characters.

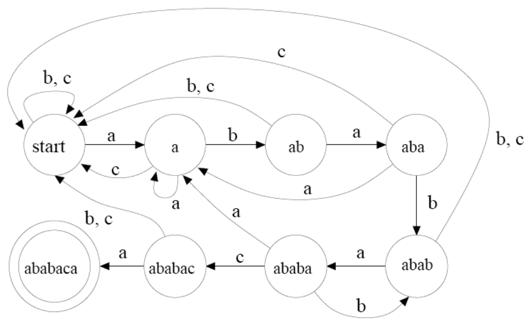
- In general, if we are at **state s_k**, we **see a character c**, which is a mismatch, then we should go back to the state s_j such that the **string s_j is a suffix of the string s_k c**.
- Moreover, j is clearly less than k but we should also choose the largest j possible.

- This corresponds to **shifting the pattern P to the right by the smallest amount** so that some **prefix of the pattern P still gives us a partial match**. This avoids missing any occurrence of the pattern P.

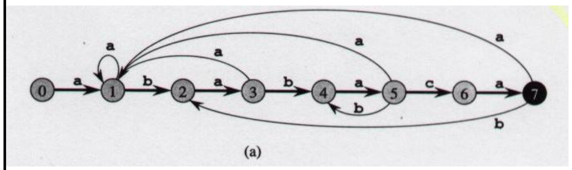
- Therefore, for this example, δ(ababa,a)= a,
- δ(ababa,b) = abab,
- δ(ababac, b) = start, and
- δ(ababac,c) = start



●**automaton for matching the pattern P = ababaca**

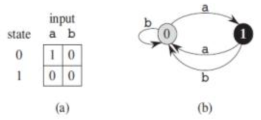


String-Matching Automaton



(a)

Finite Automata



	input	
state	a	b
0	1	0
1	0	0

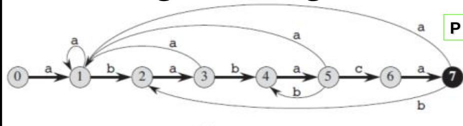
Figure 32.6 A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{a, b\}$. **(a)** A tabular representation of the transition function δ. **(b)** An equivalent state-transition diagram. State 1, shown blackend, is the only accepting state. Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates that

●<u>Strategy</u>: Build automaton for pattern, then examine each text character once.

●worst-case running time is in Q(n) + <u>*automaton creation time*</u>

String-Matching Automaton

P = ababaca



(a)

	input			
state	a	b	c	*P*
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

- What if we want to find all the occurrences of the pattern P, instead of just the first occurrence?
- Then we should define transitions out of the final state as well.
  - That is, the machine should continue to run even when it reaches the final state. But whenever the machine reaches the final state, it denotes the discovery of another occurrence of the pattern P.