Module 06: String Algorithms

Module Outline:

- Naive String Matching
- Finite Automata Matcher
- Rabin Karp matching algorithm
- Knuth Morris Pratt
- Tries
- Suffix Tree
- Suffix Array

Introduction

Given a text array $T[1 \ldots n]$ and a pattern array $P[1 \ldots m]$ such that the elements of T and P are characters taken from alphabet Σ . e.g., $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$.

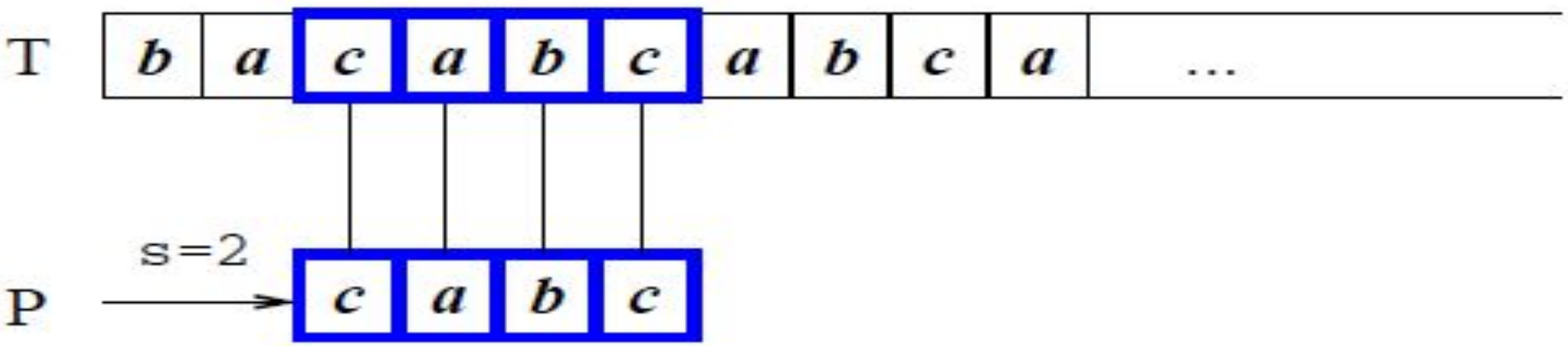
The **String Matching Problem** is to find *all* the occurrence of P in T .

 Some real world applications:

- Searching particular patterns in DNA sequences.
- In internet search engines to find Web pages relevant to input queries.
- Electronic surveillance

Example:

A pattern P occurs with **shift** s in T , if $P[1 \ldots m]$ = $T[s+1 \ldots s+m]$. The String Matching Problem is to find all values of s. Obviously, we must have $0 \leq s \leq n-m$.



Terminology:

A string w is a **prefix** of x if $x = w$ y, for some string y.

Similarly, a string w is a **suffix** of x if $x = y w$, for some string y.

Naïve String Matching

Problem: Given a String **Text[0..n-1]** and a String **Pattern[0..m-1]**, find all occurrences of Pattern[] in Text[]. You may assume that n > m.

Examples:

Input1: Text[] = “TODAY IS A GOOD DAY”, Pattern[] = “GOOD”

Output1: Pattern found at index 11

Input2: Text[] = “A FRIEND IN NEED IS A FRIEND INDEED”, Pattern[] = “FRIEND”

Output2: Pattern found at index 2, Pattern found at index 22

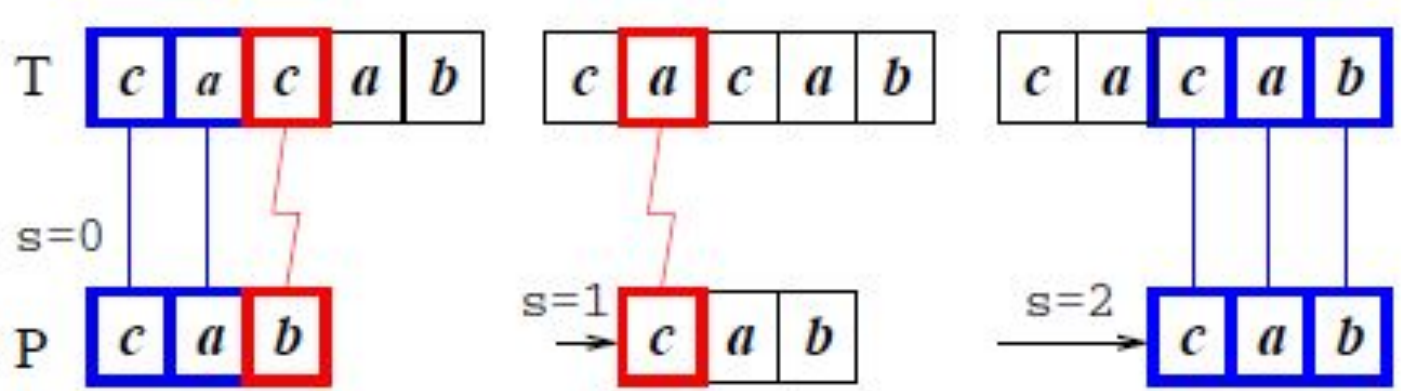
Naïve String Matching Algorithm:

```
Naïve_String_Matcher(pattern, text) {
     int M = strlen(pattern); int N = strlen(text);
     for (int s = 0; s <= N - M; s++) {
       for (int j = 0; j < M; j++)
           if (text [s + j] != pattern[j])
               break;
           if (j == M)
           printf("Pattern found at index %d \n", s);
   }
}
```

Time Complexity: O(N*M)

Example:

Initially, P is aligned with T at the first index position. P is then compared with T from **left-to-right**. If a mismatch occurs, "slide" P to *right* by 1 position, and start the comparison again.



Finite Automata Matcher

Finite automata

A *finite automaton* M, illustrated below, 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.

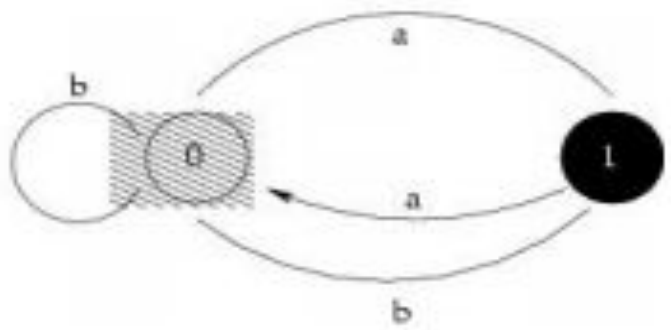
- The finite automaton begins in state q_0 and read the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state $\delta(q,a)$.
- As long as M is in a state belonging to A, M is said to have accepted the string read so far, an input that is not accepted is said to be rejected.

Example:

A two-state automaton

- $Q = \{0,1\}$.
- $q_0 \in Q = 0$.
- $A \in Q = 1$.
- $\Sigma = \{a,b\}$
- δ the table in the left-hand side of the figure.

state	input a	input b
0	1	0
1	0	0



- Figure 1: An automaton. It accepts any string ending with an odd number of a's

- The automaton can also be represented as a state-transition diagram as in the right-hand side of the figure.
- This automaton accepts those strings that end in an odd number of a's. $x=yz$, where $y=\varepsilon$ or y ends with b and $z=a^k$ and k is odd.
- abbaa rejected, abaaa accepted, bbbaaaabaaa accepted.

Construction of string-matching automaton.

A suffix function w.r.t. pattern $P[1..m]$, σ , is a mapping from Σ^* to $\{0,1,...,m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x: $\sigma(x)=\max\{k: P_k \sqsupset x\}$. For example,

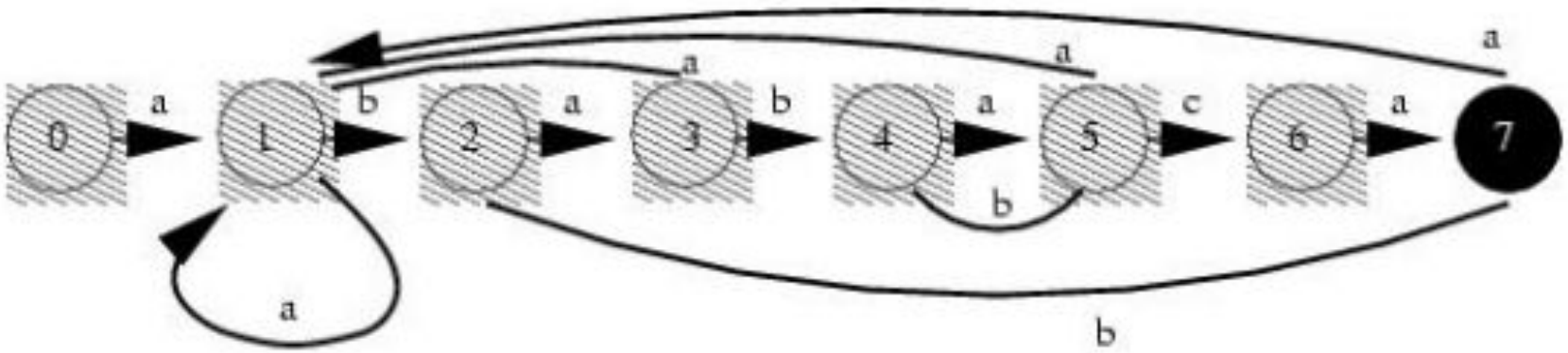
$P=ab$, $P_0 = \varepsilon$, $\sigma(\varepsilon)=0$, $\sigma(ccaca)=1$, $\sigma(ccab)=2$.

For $P[1..m]$, $\sigma(x)=m$ if and only if $P \sqsupset x$ (* a valid shift *). The whole pattern is the suffix of x.

Basic Idea:

- 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.

Example:



(a)

state	input 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	

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 $O(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

Algorithm for FA string matching:

```
FINITE-AUTOMATON-MATCHER(T,δ,m)
1.  n ← length[T]
2.  q ← 0
3.  for i ← 1 to n
4.       do q ← δ(q, T[i])
5.       if q=m then
6.            print `Pattern occurs with shift' i-m
```

Computation of transition function:

COMPUTE-TRANSITION-FUNCTION(P,Σ)

1. $m \leftarrow length[P]$
2. for $q \leftarrow 0$ to m (for each state)
3. do for each character $a \in \Sigma$ ($|\sum|$)
4. do $k \leftarrow min(m+1, q+2)$
5. repeat $k \leftarrow k-1$ $(1 \leq k \leq m+1)$
6. until $P_k \sqsupset P_q a$ $(\sum k)$
7. $\delta(q,a) \leftarrow k$
8. return δ

Running Time of Compute-Transition-Function

Running Time: $O(m^3 |\Sigma|)$

Outer loop: $m |\Sigma|$

Inner loop: runs at most $m+1$

$P_k \supset P_q a$: requires up to m comparisons

Improving Running Time

Much faster procedures for computing the transition function exist. The time required to compute P can be improved **to O(m|Σ|)**.

The time it takes to find the string is linear: **O(n)**.

This brings the total runtime to:

O(n + m|Σ|)

Not bad if your string is fairly small relative to the text you are searching in.

Exercise:

Given pattern P=abba, Σ={a,b}, construct its automaton.

Show how the automaton works for text T[1..12]=baabbabbaaba.