Introduction to Algorithms

Chapter 32 : String Matching

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2025

Outline of Topics

Table of Contents

Definition of String Matching Problem

String-matching Problem:

1. 1. Find one occurrence of a pattern in a text ;
2. 2. Find out all the occurrences of a pattern in a text.

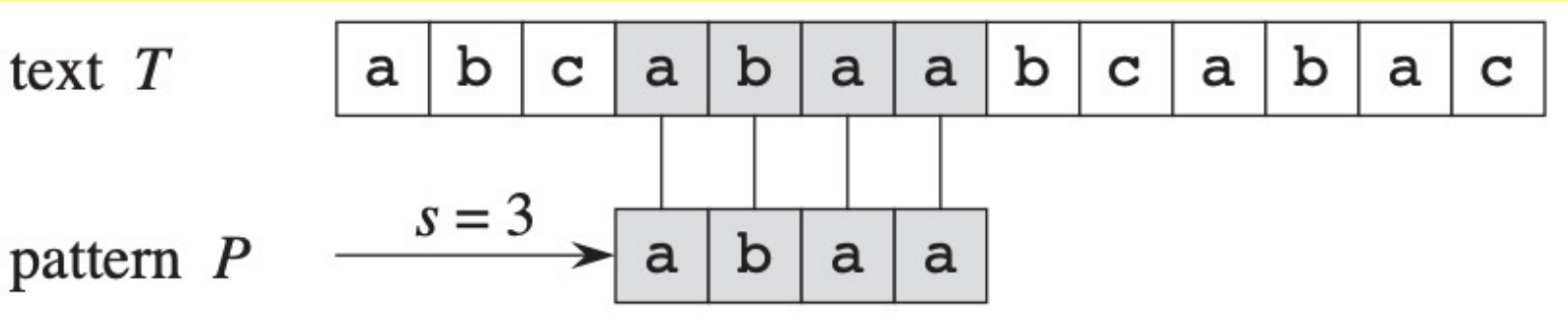
Applications require two kinds of solution depending on which string, the pattern or the text, is given first.

1. 1. Algorithms based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern and solve the first kind of problem.
2. 2. The notion of indexes realized by trees or automata is used in the second kind of solutions.

String Matching

- Word processors
- Virus scanning
- Text information retrieval systems (Lexis, Nexis)
- Digital libraries
- Natural language processing
- Specialized databases
- Computational molecular biology
- Web search engines
- Bioinformatics

An Example of String Matching



Notation and Terminology

Parameters

- T: the text is an array $T[1..n]$ of length n
- P: the pattern is an array $P[1..m]$ of length m.
- n: the length of the text.
- m: the length of the pattern.
  Typically, $n >> m$.
- $\sum$: the alphabet.
- $\sqsupset$: suffix. e.g., $cca \sqsupset bcca$
- $\sigma(x) = max\{k : P_k \sqsupset x\}$: suffix function

The Basic Idea of String Matching

sliding window mechanism

- 1. Scan the text T with a window of the length of m;
- 2. Firstly align the pattern with the left end of the text;
- 3. Compare the P with the corresponding character of the T;
- 4. Move the window to the right after each successful match or each mismatch;
- 5. Repeat steps 3 and 4 until the right end of the window is beyond the right of the text.

When comparing, the order can be from left to right, right to left, or even in a specific order.

Table of Contents

Brute Force

Brute force

Check for pattern starting at every text position, trying to match any substring of length m in the text with the pattern.

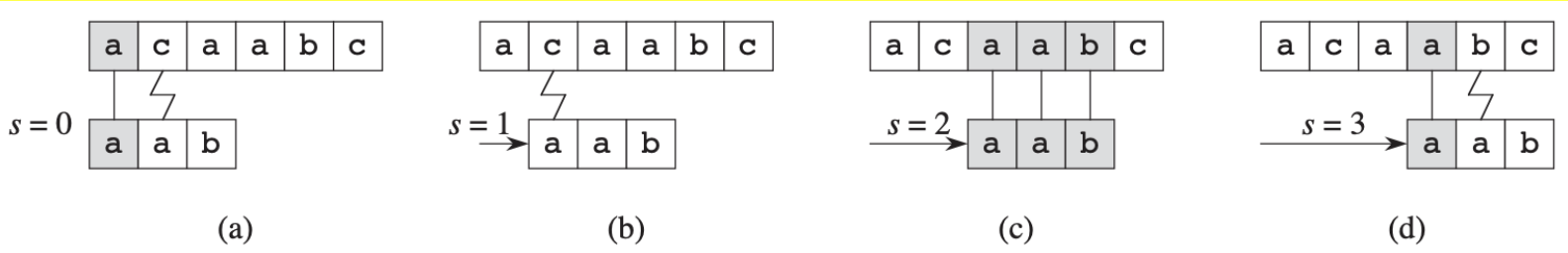
Analysis of brute force:

- running time depends on pattern and text
- can be slow when strings repeat themselves
- worst case: $O(mn)$ comparisons
- too slow when m and n are large.

Brute Force

NATIVE-STRING-MATCHING(T, P)

1: $n = T.length$
2: $m = P.length$
3: **for** $s = 0$ to $n - m$ **do**
4: **if** $P[1..m] == T[s+1..s+m]$ **then**
5: print "Pattern occurs with shift" s



Brute Force

Time Complexity: $O((n-m+1)m)$. Why is it slow?

NATIVE-STRING-MATCHING(T,P)

1: $n = T.length$
2: $m = P.length$
3: **for** $s = 0$ to $n-m$ **do**
4: **if** $P[1..m] == T[s+1..s+m]$ **then**
5: print "Pattern occurs with shift" s

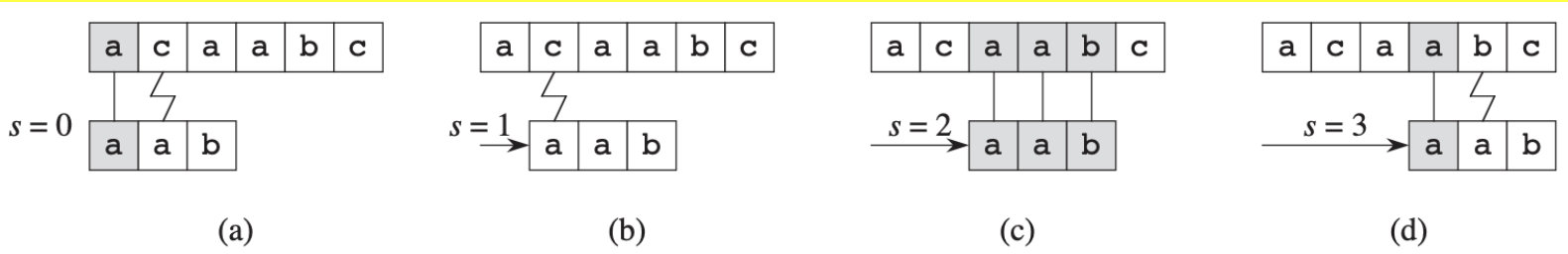


Table of Contents

The Basic Idea of Rabin-Karp Algorithm

Basic Idea of Rabin-Karp Algorithm

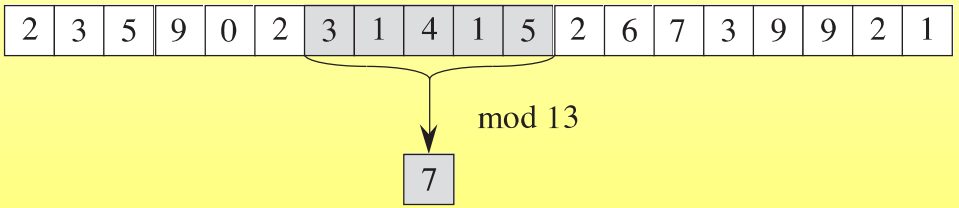
A string search algorithm which compares a string's hash values, rather than the strings themselves. For efficiency, the hash value of the next position in the text is easily computed from the hash value of the current position.

- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- If the hash values are equal, the algorithm will compare the pattern and the M-character sequence.
- In this way, there is only one comparison per text subsequence, and character matching is only needed when hash values match.

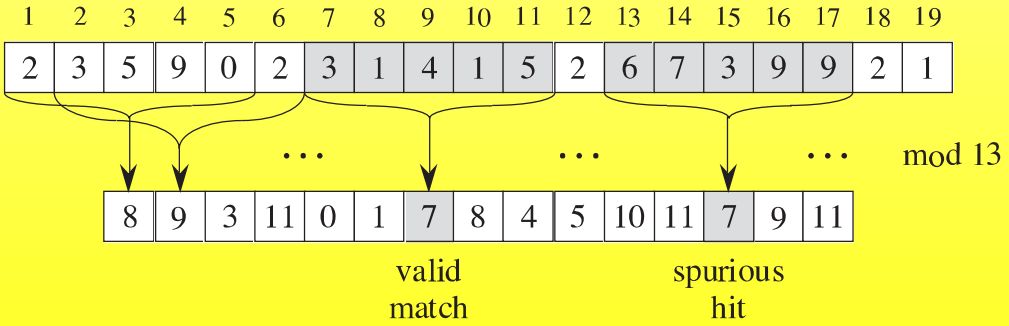
How Rabin-Karp works

- An hashing function *hash* should have the following properties:
  - Efficiently computable
  - Highly discriminating for strings
  - $t_{s+1} = hash(T[s+2,...,s+m+1])$ must be easily computable from $t_s = hash(T[s+1,...,s+m])$ and $T[s+m+1]$
  - $hash(T[s+2,...,s+m+1]) = rehash(T[s+1], T[s+m+1], hash(T[s+1,...,s+m]))$
- Choosing $hash(k) = k \bmod q$, q is a large prime.

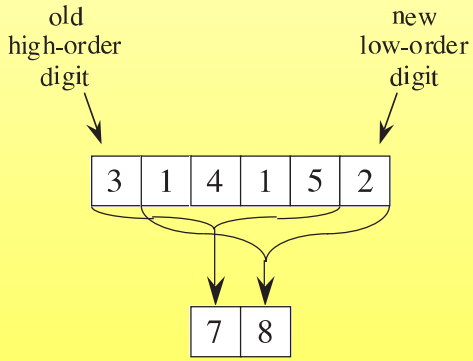
Example of R.K. Algorithm

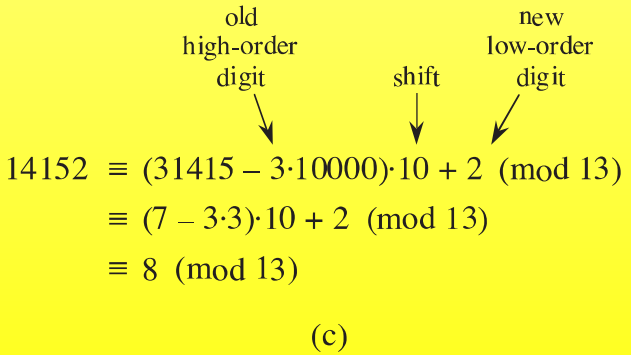


(a)



(b)





(c)

How Rabin-Karp works

- Let $d = |\sum|$, define a function *ord*: $\sum \to \{0, 1, 2, ..., d-1\}$
- For a word w of length m in the text T, let $hash(w)$ be defined as follows:
  - Let: $x[i] = ord(w[i]), 1 \leq i \leq m$
  - $hash(w[1..m]) =$ $(x[1] \cdot d^{m-1} + x[2] \cdot d^{m-2} + \cdots + x[m] \cdot d^0) \bmod q$,where q is a large number, $hash(w[1..m])$ is an integer.
  - $hash(w[2..m+1]) =$ $(x[2] \cdot d^{m-1} + x[3] \cdot d^{m-2} + \cdots + x[m+1] \cdot d^0) \bmod q =$ $((hash(w[1..m]) - x[1] \cdot d^{m-1}) \cdot d + x[m+1] \cdot d^0) \bmod q$.
  - $rehash(a, b, t_s) = ((t_s - a \cdot h) \cdot d + b) \bmod q$,where $d^{m-1} \bmod q$ can be calculated in advance and recorded as h.

Rabin-Karp Algorithm Pseudocode

RK(P, T, d, q)
1: $n = T.length, \quad m = P.length, \quad h = d^{m-1} \bmod q;$
2: $p = 0, \quad t_0 = 0;$
3: **for** $i = 1$ to m **do** //pre-processing
4: $p = ((p \cdot d) + ord(P[i])) \bmod q$ // $hash(P[1..m])$
5: $t_0 = ((t_0 \cdot d) + ord(T[i])) \bmod q$ // $hash(T[1..m])$
6: **for** $s = 0$ to $n - m$ **do** // matching, $(n - m + 1)$ times
7: **if** $p == t$ && $P[1..m] == T[s+1..s+m]$ **then** // $\Theta(m)$
8: print "Pattern occurs with shift" s
9: **if** $s < n - m$ **then** // compute t_{s+1} based on t_s
10: $t_{s+1} = (t_s - ord(T[s+1]) \cdot h) \cdot d + ord(T[s+m+1])) \bmod q$

Rabin-Karp Algorithm Analysis

- The preprocessing phase of the Rabin-Karp algorithm consists in computing $hash(P)$. It can be done in constant space and $O(m)$ time.
- During searching phase, it is enough to compare $hash(P)$ with $hash(T[j..j+m-1])$ for $1 \leq j \leq n-m+1$.
- If an equality is found, it is still necessary to check the equality $P = T[j..j+m-1]$ character by character.
- The time complexity of the Rabin-Karp algorithm is $\Theta((n-m+1)m) = \Theta(mn)$ (when searching for a^m in a^n for instance). Its expected number of text character comparisons is $O(n+m) = O(n)$, when the valid points are small, e.g., O(1).

Table of Contents

Finite Automata

Finite Automata

A finite automaton is a quintuple (Q,Σ,δ,s,F):

- Q: the finite **set of states**
- Σ: the finite **input alphabet**
- δ: the transition function from $Q \times \Sigma$ to Q // deterministic FA
- s $\in$ Q: the start state
- F $\subset$ Q: the set of final (accepting) states

The Final-State Function

- A finite automaton M induces a final-state function $\phi : \Sigma^\star \to Q$ such that $\phi(w)$ is the state M ends up in after reading the string w. Thus, M accepts a string w if and only if $\phi(w) \in F$.
- We define the function ϕ recursively, using transition function δ:

$$\phi(\varepsilon) = q_0$$
$$\phi(wa) = \delta(\phi(w), a) \text{ for } w \in \Sigma^\star, a \in \Sigma$$

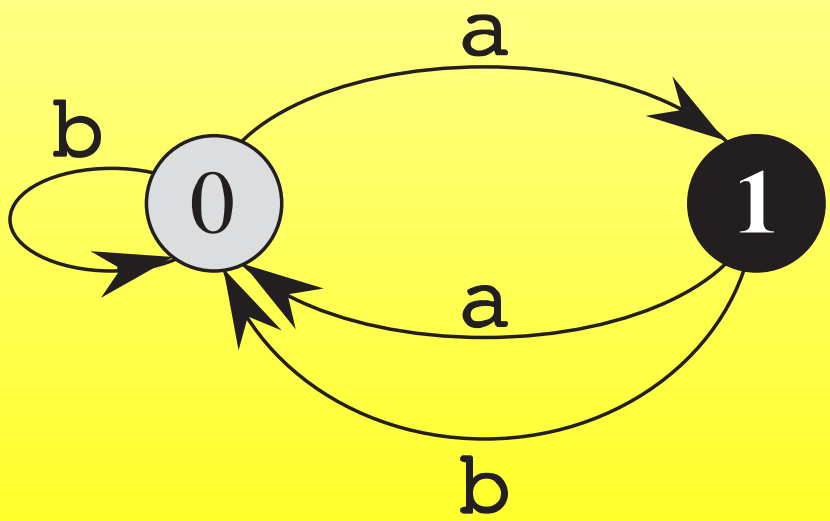
How it works

A finite automaton accepts strings in a specific language. It begins in state q_0 and reads characters one at a time from the input string. It makes transitions based on these characters. When it reaches the end of the tape, if it is in one of the accept states, that string is accepted by the FA.

e.g., transition function: $\delta(0,a) = 1$

final-state function: $\phi(ababa) = 1$

This FA accepts those strings that end in an odd number of a's.



The Suffix Function

In order to properly search for the string, the program must define a **suffix function** (σ) which checks to see how much of what it is reading matches the search string at any given moment.

Later we will see the equivalence between ϕ and σ.

$$\sigma(x) = max\{k : P_k \sqsupset x\}$$

$$P = abaabc$$

$$P_1 = a$$

$$P_2 = ab$$

$$P_3 = aba$$

$$P_4 = abaa$$

$$\sigma(abbaba) = 3 \qquad //aba$$

P_k denotes the prefix of length k of string P.

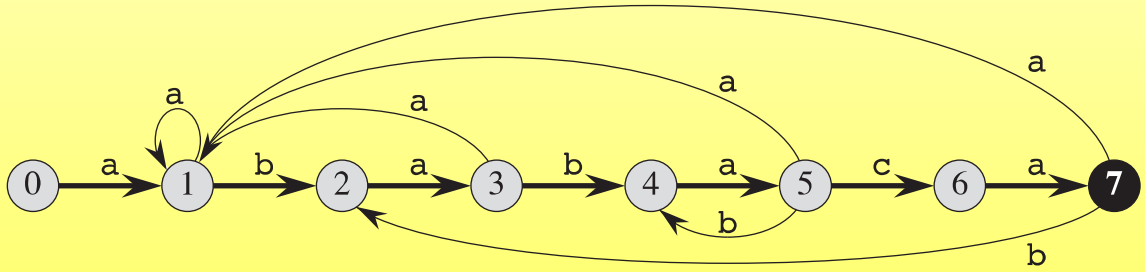
String-Matching Automata

- For any pattern P of length m, we can define its string matching automata:

$$Q = \{0, \ldots, m\} \quad \text{(state)}$$
$$q_0 = 0 \quad \text{(start state)}$$
$$F = \{m\} \quad \text{(accepting state)}$$
$$\delta(q, a) = \sigma(P_q a)$$

Example

$$\delta(q,a) = \sigma(P_q a)$$



(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	

(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)

String-Matching Automata

- The transition function chooses the next state to maintain the invariant:

$$\phi(T_i) = \sigma(T_i)$$

After scanning in i characters, the state number is the longest prefix of P that is also a suffix of T_i.

Finite-Automaton-Matcher

- The simple loop structure implies a running time for a string of length n is $O(n)$.
- However: this is only the running time for the actual string matching. It does not include the time it takes to compute the transition function.

FINITE-AUTOMATON-MATCHER(T, δ, m)
1: $n = T.length$
2: $q = 0$
3: **for** $i = 1$ to n **do**
4: $\quad q = \delta(q, T[i])$
5: $\quad$ **if** q==m **then**
6: $\quad\quad s = i - m$
7: $\quad\quad$ print "Pattern occurs at shift" s

Computing the Transition Function

COMPUTE-TRANSITION-FUNCTION(P, Σ)
1: $m = P.length$
2: **for** $q = 0$ to m **do**
3: **for** each character $a \in \Sigma$ **do**
4: $k = \min(m+1, q+2)$
5: **repeat**
6: $k = k - 1$
7: **until** $P_k \sqsupset P_q a$
8: $\delta(q,a) = k$
9: **return** δ

- This procedure computes $\delta(q,a)$ according to its definition. The loop on line 2 cycles through all the states, while the nested loop on line 3 cycles through the alphabet. Thus all state-character combinations are accounted for. Lines 4-7 set $\delta(q,a)$ to be the largest k such that $P_k \sqsupset P_q a$.

Running Time of Compute-Transition-Function

- **Running Time:** $O(m^3|\Sigma|)$
- **Outer loop:** $m|\Sigma|$
- **Inner loop:** runs at most $m+1$
- $P_k \sqsupset P_q a$**:** requires up to m comparisons

Improving Running Time

- Much faster procedures for computing the transition function exist. The time required to compute δ based on P can be improved to $O(m|\Sigma|)$
- The time it takes to find the string is linear: $O(n)$.
- This brings the total runtime to:$O(n+m|\Sigma|)$.
- Not bad if your string is fairly small relative to the text you are searching in.

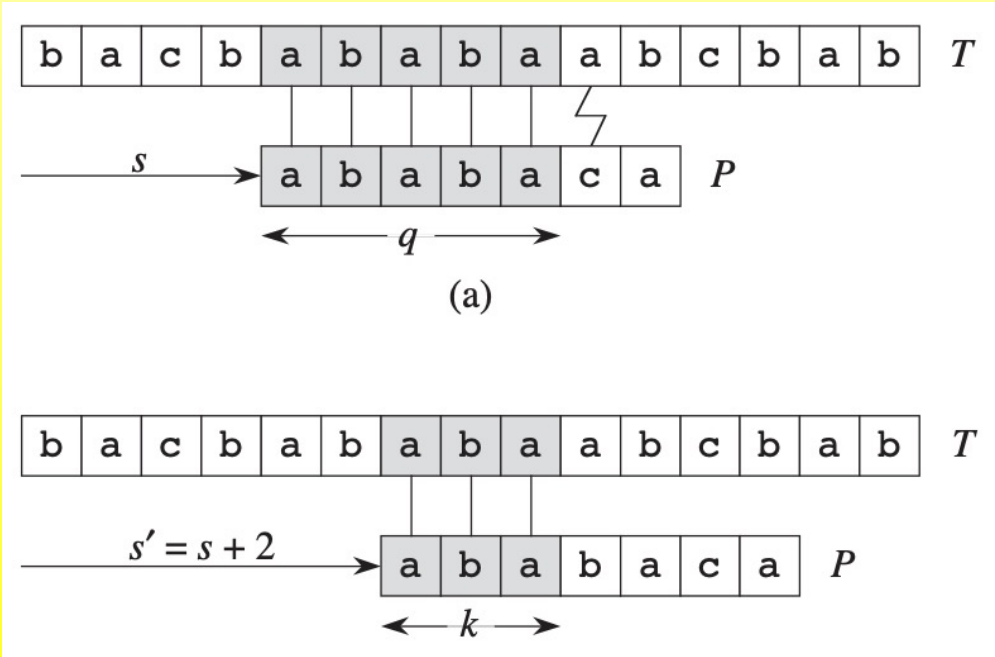
Table of Contents

The KMP Algorithm

Basic Idea of KMP

The prefix function π encapsulates knowledge about how the pattern matches against shifts of itself. We take advantage of this information to avoid testing useless shifts.

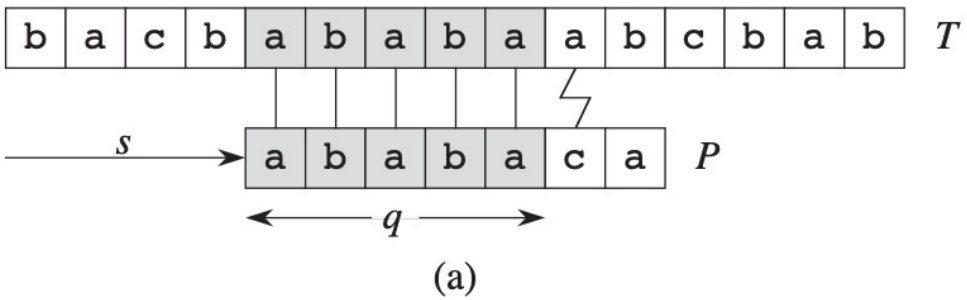
An Example of KMP Algorithm



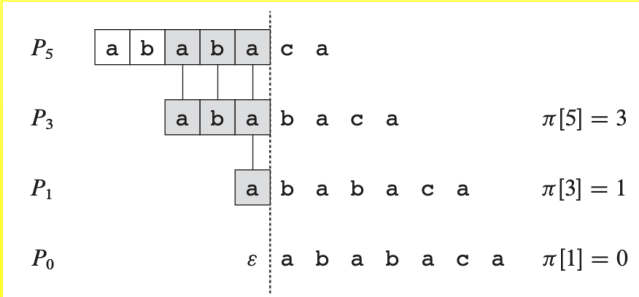
The KMP Algorithm

- The pointer only shift to the right and will not retreat to the left.
- When test the $T[s+1,...s+q+1]$, $P[1..q] = T[s+1,...s+q]$, but $P[q+1] \neq T[s+q+1]$.
- Given that pattern characters $P[1..q]$ match text characters $T[s+1..s+q]$, what is the least shift $s' > s$ such that for some $k < q, P[1..k] = T[s'+1..s'+k]$, *where* $s'+k = s+q$?
- Given a pattern $P[1..m]$, the prefix function for the pattern P is the function $\pi : \{1,2,...,m\} \to \{0,1,...m-1\}$ such that $\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$.

An Example of KMP Algorithm



i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1



The KMP Algorithm

COMPUTE-PREFIX-FUNCTION(P)

```
1:  m = P.length
2:  let π[1..m] be a new array
3:  π[1] = 0
4:  k = 0
5:  for q = 2 to m do
6:      while k > 0 and P[k+1] ≠ P[q] do
7:          k = π[k]
8:      if P[k+1] == P[q] then
9:          k = k+1
10:     π[q] = k
11: return π
```

The running time is $\Theta(m)$

The KMP Algorithm

KMP-MATCHER(T,P)

1: $n = T.length$
2: $m = P.length$
3: $\pi =$ Compute-Prefix-Function(P)
4: $q = 0$ //number of characters matched
5: **for** $i = 1$ to n **do** //scan the text from left to right
6: **while** $q > 0$ and $P[q+1] \neq T[i]$ **do**
7: $q = \pi[q]$
8: **if** $P[q+1] == T[i]$ **then**
9: $q = q+1$ //next character matches
10: **if** $q == m$ **then** //is all of P matches
11: print "Pattern occurs with shift" $i-m$
12: $q = \pi[q]$ //look for the next match

The running time is $\Theta(n)$

Summary of KMP

- Build π from pattern
- Run π on text
- $O(m+n)$ worst case string search
- Good efficiency for patterns and texts with much repetition
  - binary files
  - graphics formats
- Less useful for text strings.
- Online algorithm
  - virus scanning
  - Internet spying