String Matching

- String matching problem
  - prefix
  - suffix
  - automata
  - String-matching automata
  - suffix function
  - prefix function
  - Knuth-Morris-Pratt algorithm

Chapter 32: String Matching

String-matching problem

1. **Text: an array $T[1 .. n]$ containing n characters drawn from a finite alphabet Σ**

   (for instance, Σ = {0, 1} or Σ = {a, b, …, z}.)

   Pattern: an array $P[1 .. m]$ ($m \leq n$)
2. **Finding all occurrences of a pattern in a text is a problem that arises frequently in text-editing programs.**

Definition

We say that pattern P occurs with shift s in text T (or, equivalently, that pattern P occurs beginning at position $s + 1$ in text T)

if $0 \le s \le n - m$ and

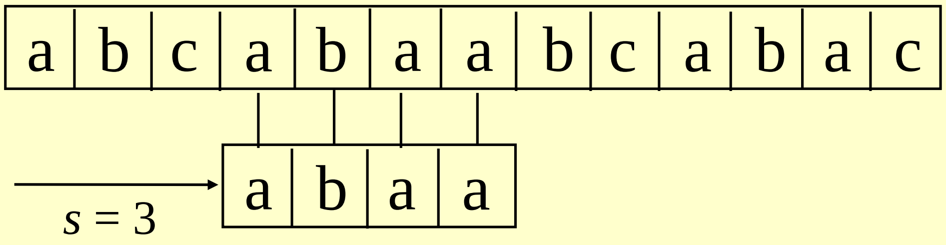
$$T[s + 1 .. s + m] = P[1 .. m]$$

(i.e., if $T[s + j] = P[j]$ for $1 \le j \le m$).

Valid shift s – if P occurs with shift s in T. Otherwise, s is an invalid shift.

text T:

pattern P:



We will find all the valid shifts.

■ Naïve algorithm

Naïve-String-Matcher(T, P)

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

Obviously, the time complexity of this algorithm is bounded by O(nm).

In the following, we will discuss Knuth-Morris-Pratt algorithm, which needs only O($n + m$) time.

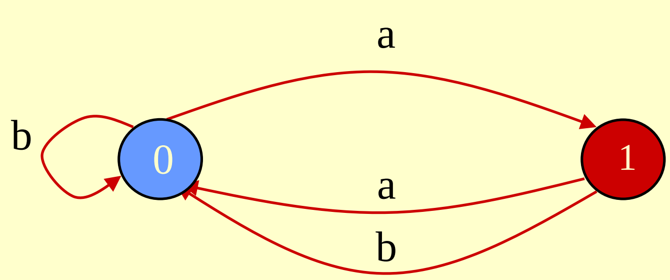
Finite automata

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

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

Example: $Q = \{0, 1\}$, $q_0 = 0$, $A = \{1\}$, $\Sigma = \{a, b\}$
$\delta(0, a) = 1$, $\delta(0, b) = 0$, $\delta(1, a) = 0$, $\delta(1, b) = 0$.

state	input a	input b
0	1	0
1	0	0



$(b^n a^l b^m)^+$
$n \geq 0$.
l is an odd integer.
$m \geq 0$.

■ String-matching automata for patterns

Σ^* - the set of all finite-length strings formed using characters from the alphabet Σ

ε - zero-length *empty string*

$|x|$ - the length of string x

xy - the concatenation of two strings x and y, which has length $|x| + |y|$ and consists of the characters from x followed by the characters from y

prefix – a string w is a prefix of a string x, denoted w ⦿ x, if $x = wy$ for some $y \in \Sigma^*$.

suffix – a string w is a suffix of a string x, denoted w ▪ x, if $x = yw$ for some $y \in \Sigma^*$.

Example: ab ⦿ abcca. cca ▪ abcca.

■ String-matching automata for patterns

- P_k - $P[1 .. k]$ ($k \le m$), a prefix of $P[1 .. m]$

  suffix function σ - a mapping from Σ^* to $\{0, 1, \ldots, m\}$ 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 ■ x}.

  Note that $P_0 = \varepsilon$ is a suffix of every string.

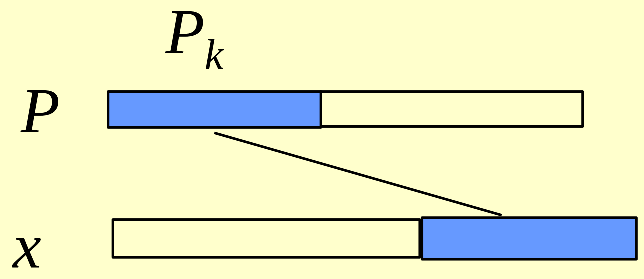
- Example

  P = ab

  We have $\sigma(\varepsilon) = 0$

  σ(ccaca) = 1 P = ab

  σ(ccab) = 2 P = ab

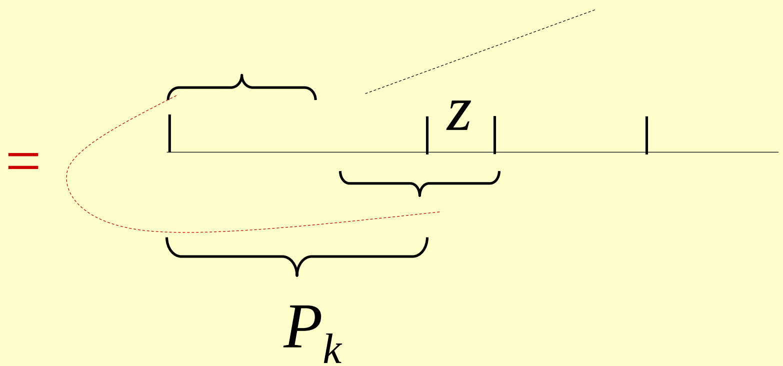


■ String-matching automata for a pattern

For a pattern $P[1 .. m]$, its string-matching automaton can be constructed as follows.

1. The state set Q is $\{0, 1, \ldots, m\}$. The start state q_0 is state 0, and state m is the only accepting state.
2. The transition function δ is defined by the following equation, for any state k and character z:

$$\delta(k, z) = \sigma(P_k z)$$



$P = \underline{ab}cad \ldots \ldots$

$\delta(4, b) = \sigma(P_4 b) = \sigma(abc\underline{ab}) = 2$

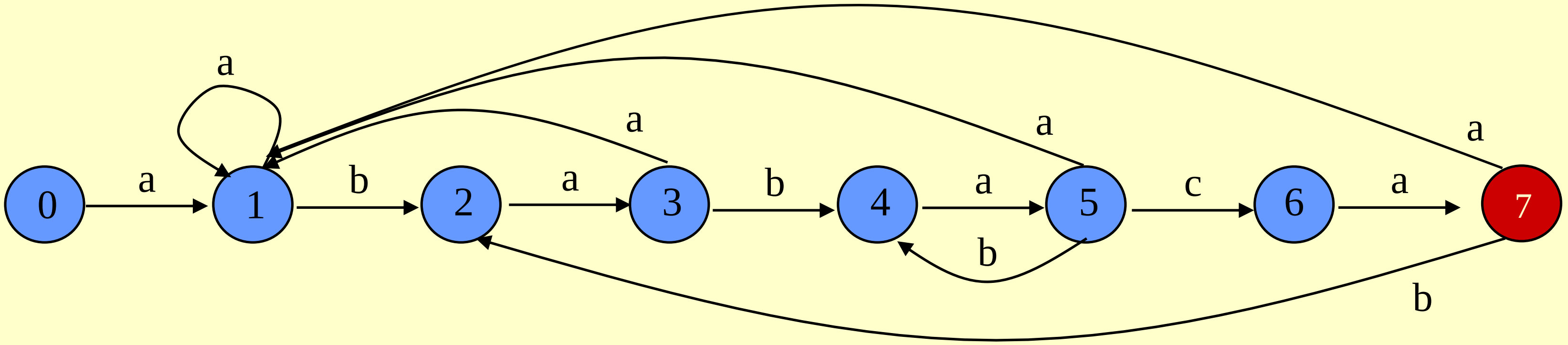
$\delta(4, d) = \sigma(P_4 d) = \sigma(\underline{abcad}) = 5$

■ String-matching automata for patterns

- Example

P = ababaca

$\delta(k, z) = \sigma(P_k z)$



P	a	b	a	b	a	c	a	
State	0	1	2	3	4	5	6	7

transition function

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

Assume that $P_0 = \varepsilon$.

■ String-matching automata for patterns

- Example

P = ababaca

$\delta(k, z) = \sigma(P_k z)$

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

transition function

Assume that $P_0 = \varepsilon$.

$\delta(0, \text{a}) = \sigma(P_0\text{a}) = \sigma(\text{a}) = 1$ $\delta(1, \text{a}) = \sigma(P_1\text{a}) = \sigma(\text{aa}) = 1$

$\delta(0, \text{b}) = \sigma(P_0\text{b}) = \sigma(\text{b}) = 0$ $\delta(1, \text{b}) = \sigma(P_1\text{b}) = \sigma(\text{ab}) = 2$ $\cdots\ \cdots$

$\delta(0, \text{c}) = \sigma(P_0\text{c}) = \sigma(\text{c}) = 0$ $\delta(1, \text{c}) = \sigma(P_1\text{c}) = \sigma(\text{ac}) = 0$

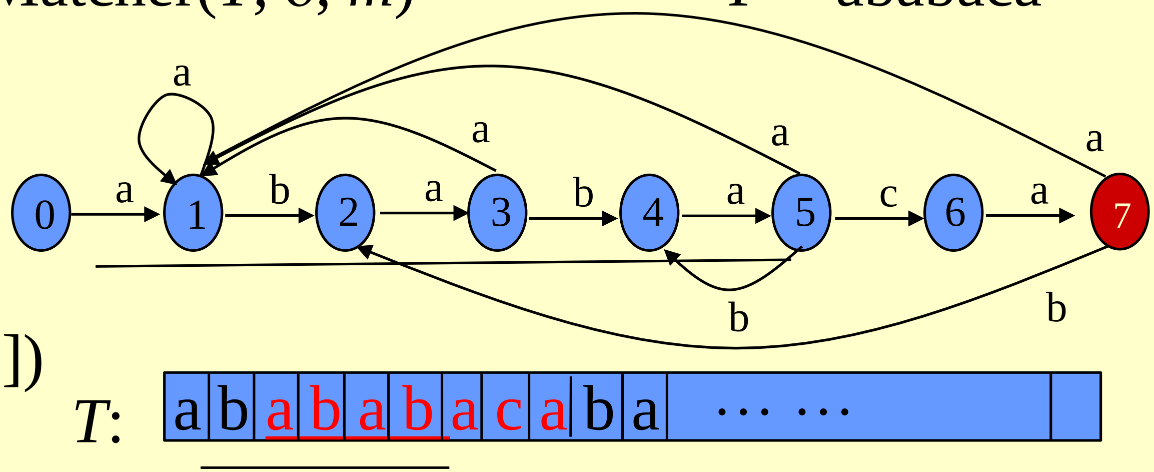
■Finite-Automaton-Matcher

- String matching by using the finite automaton

Finite-Automaton-Matcher(T, δ, m)

P = ababaca

1. $n \leftarrow length[T]$
2. $q \leftarrow 0$
3. **for** $i \leftarrow 1$ **to** n
4. **do** $q \leftarrow \delta(q, T[i])$
5. **if** $q = m$
6. **then** print "pattern occurs with shift" $i - m$

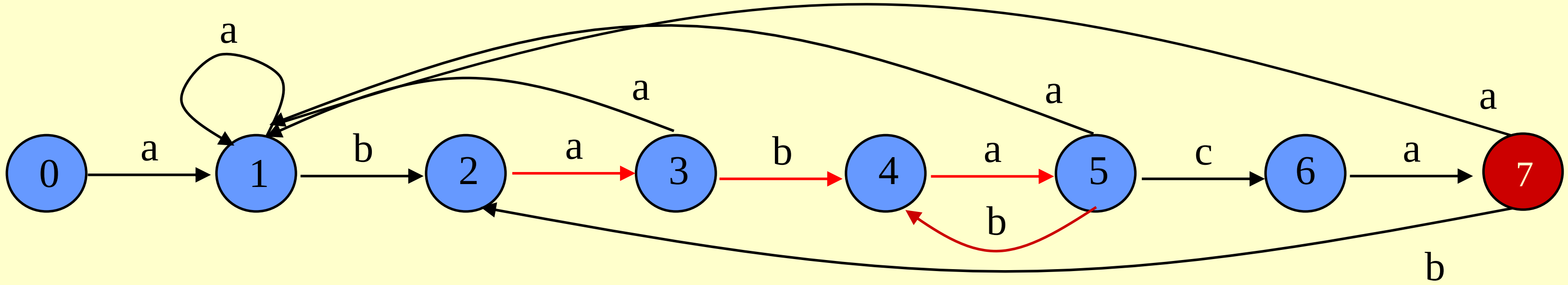


If the finite automaton is available, the algorithm needs only $O(n + m)$ time.

■ Finite-Automaton-Matcher

- Example

P = ababaca, T = abababacaba $\qquad \delta(k, z) = \sigma(P_k z)$



step 1: $q = 0$, $T[1]$ = a. Go into the state $q = 1$.
step 2: $q = 1$, $T[2]$ = b. Go into the state $q = 2$.
step 3: $q = 2$, $T[3]$ = a. Go into the state $q = 3$.
step 4: $q = 3$, $T[4]$ = b. Go into the state $q = 4$.
step 5: $q = 4$, $T[5]$ = a. Go into the state $q = 5$.
step 6: $q = 5$, $T[6]$ = b. Go into the state $q = 4$.
step 7: $q = 4$, $T[7]$ = a. Go into the state $q = 5$.
step 8: $q = 5$, $T[8]$ = c. Go into the state $q = 6$.
step 9: $q = 6$, $T[9]$ = a. Go into the state $q = 7$.

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

■ Knuth-Morris-Pratt algorithm

- Dynamic computation of the transition function δ

We needn't compute δ altogether, but using an auxiliary function π, called a *prefix function*, to calculate δ–values "on the fly".

prefix function π - a mapping from $\{1, \ldots, m\}$ to $\{0, 1, \ldots, m\}$ such that

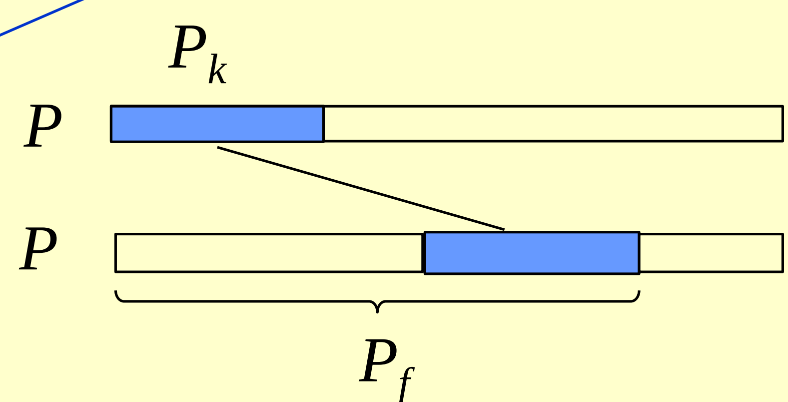
$$\pi(f) = \max\{k: k < q, P_k \blacksquare P_f\}.$$

comparison with suffix function:

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

$\sigma(P_k z) = \delta(k, z)$

$z \in \Sigma$



■ Knuth-Morris-Pratt algorithm

- Example

P = ababaca

$O(|\Sigma|m)$

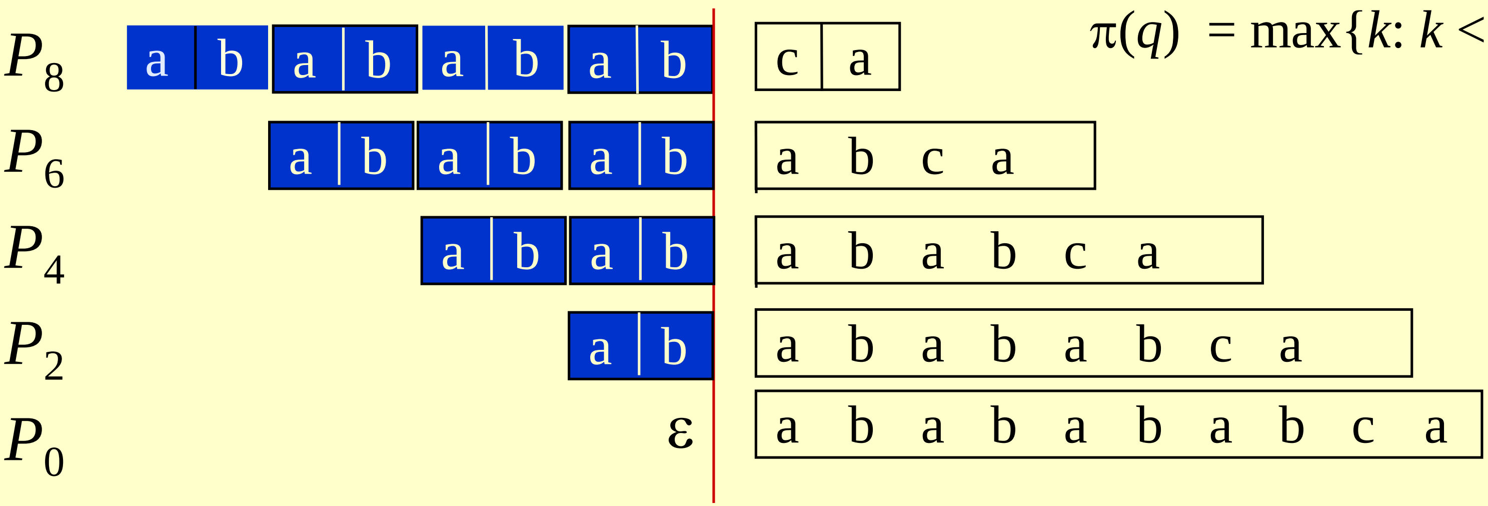
1	1	3	1	5	1	7	1
0	2	0	4	0	4	0	2
0	0	0	0	0	6	0	0

P = ababababca

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

$O(m)$

$\pi(q) = \max\{k: k < q, P_k$ ■ $P_q\}$



$\pi[8] = 6$

$\pi[6] = 4$

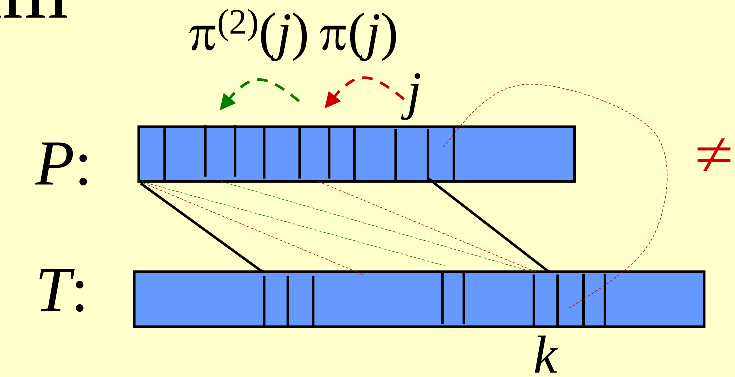
$\pi[4] = 2$

$\pi[2] = 0$

By using the values of prefix function values,
we will dynamically compute suffix function values.
In this way, a suffix function value is computed only
when it is needed. Thus, a lot of time can be saved.

How?

Knuth-Morris-Pratt algorithm



- function $\pi^{(u)}(j)$

  i) $\pi^{(1)}(j) = \pi(j)$, and

  ii) $\pi^{(u)}(j) = \pi(\pi^{(u-1)}(j))$, for $u > 1$.

  That is, $\pi^{(u)}(j)$ is just π applied u times to j.

  Example: $\pi^{(2)}(6) = \pi(\pi(6)) = \pi(4) = 2$ for P = ababaca.

- How to use $\pi^{(u)}(j)$?

  Suppose that the automaton is in state j, having read $T[1 .. k]$, and that $T[k+1] \neq P[j+1]$. Then, apply π repeatedly until it find the smallest value of u for which either

  1. $\pi^{(u)}(j) = l$ and $T[k + 1] = P[l + 1]$, or
  2. $\pi^{(u)}(j) = 0$ and $T[k + 1] \neq P[1]$.

- Knuth-Morris-Pratt algorithm

  - How to use $\pi^{(u)}(j)$?

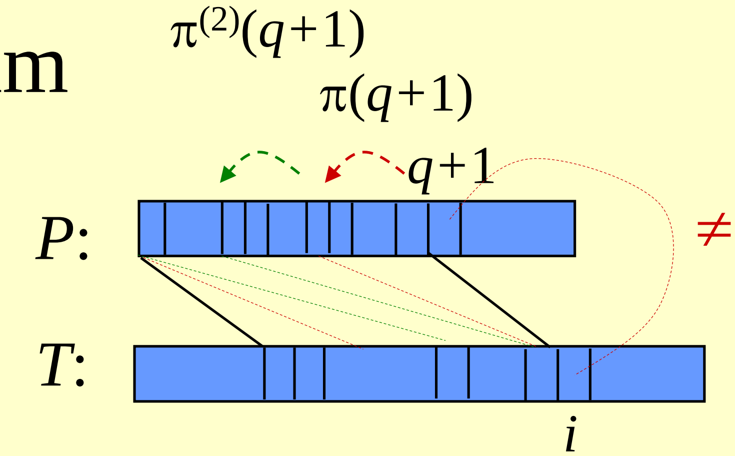
    1. $\pi^{(u)}(j) = l$ and $T[k+1] = P[l+1]$, or
    2. $\pi^{(u)}(j) = 0$ and $T[k+1] \neq P[1]$.

  That is, the automaton backs up through $\pi^{(1)}(j)$, $\pi^{(2)}(j)$, … until either Case 1 or 2 holds for $\pi^{(u)}(j)$ but not for $\pi^{(u-1)}(j)$.

- If Case 1 holds, the automaton enters state l.
- If Case 2 holds, it enters state 0.

  In either case, input pointer is advanced to position $T[k + 2]$. In Case 1, $P[1 .. l]$ is the longest prefix of P that is a suffix of $T[1 .. k]$, then $P[1 .. \pi^{(u)}(j) + 1] = P[1 .. l + 1]$ is the longest prefix of P that is a suffix of $T[1 .. k + 1]$. In Case 2, no prefix of P is a suffix of $T[1 .. k + 1]$ and we will search P from scratch.

■ Knuth-Morris-Pratt algorithm

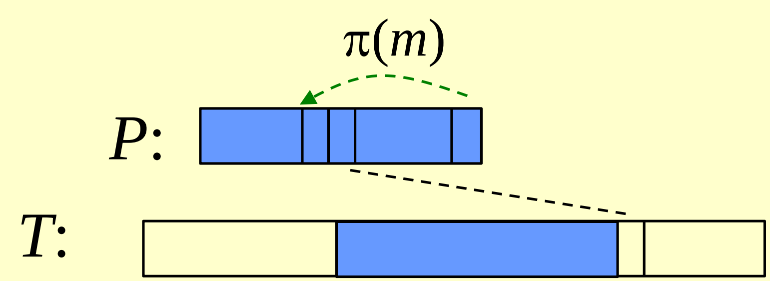


- Algorithm

KMP-Matcher(T, P)

```
n ← length[T]
m ← length[P]
π ← Compute-Prefix-Function(P)
q ← 0
for i ← 1 to n
    do while q > 0 and P[q + 1] ≠ T[i]
            do q ← π[q]
        if P[q + 1] = T[i]
        then q ← q + 1
        if q = m
            then print "pattern occurs with shift" i – m
                q ← π[q]
```

Compute $\pi^{(u)}(q+1)$



■ Knuth-Morris-Pratt algorithm

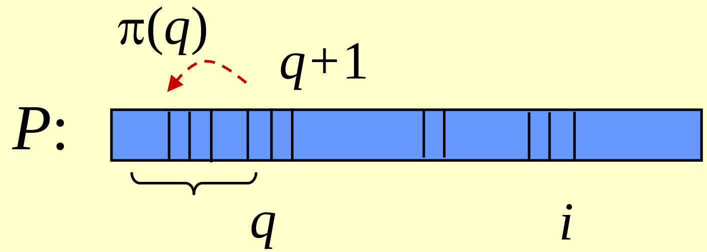
- Algorithm

Compute-Prefix-Function(P)

1. $m \leftarrow length[T]$
2. $\pi[1] \leftarrow 0$
3. $q \leftarrow 0$
4. **for** $i \leftarrow 2$ **to** m
5. **do while** $q > 0$ and $P[q + 1] \neq P[i]$
6. **do** $q \leftarrow \pi[q]$
7. **if** $P[q + 1] = P[i]$
8. **then** $q \leftarrow q + 1$
9. $\pi[i] \leftarrow q$
10. **return** π

4. $q \leftarrow 0$
5. **for** $i \leftarrow 1$ **to** n
6. **do while** $q > 0$ and $P[q + 1] \neq T[i]$
7. **do** $q \leftarrow \pi[q]$
8. **if** $P[q + 1] = T[i]$
9. **then** $q \leftarrow q + 1$
10. **if** $q = m$
11. **then** print …

/*if $q = 0$ or $P[q + 1] = P[i]$, going out of the while-loop.*/



P = abababababca

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

■ Knuth-Morris-Pratt algorithm – sample trace

- Example

  P = abababababca,

  T = ababaababababca

```
π[1] = 0
q ← 0
for i ← 2 to n
do while q > 0 and P[q + 1] ≠ P[i]
      do q ← π[q]
      if P[q + 1] = P[i]
      then q ← q + 1
      π[i] ← q
```

***Compute prefix function*:**

$\pi[1] = 0$

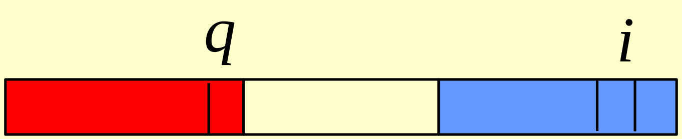
$q = 0$

$i = 2$, $P[q + 1] = P[1] = \text{a}$, $P[i] = P[2] = \text{b}$, $P[q + 1] \neq P[i]$

$\pi[i] \leftarrow q$ ($\pi[2] \leftarrow 0$)

$i = 3$, $P[q + 1] = P[1] = \text{a}$, $P[i] = P[3] = \text{a}$, $P[q + 1] = P[i]$

$q \leftarrow q + 1$, $\pi[i] \leftarrow q$ ($\pi[3] \leftarrow 1$)



$q = 1$

$i = 4$, $P[q + 1] = P[2] = \text{b}$, $P[i] = P[4] = \text{b}$, $P[q + 1] = P[i]$

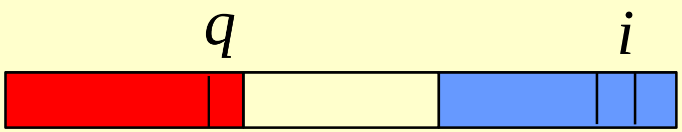
$q \leftarrow q + 1$, $\pi[i] \leftarrow q$ ($\pi[4] \leftarrow 2$)

P = ababababca

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

■ Knuth-Morris-Pratt algorithm – sample trace

- Example



$q = 2$

$i = 5$, $P[q + 1] = P[3] =$ a, $P[i] = P[5] =$ a, $P[q + 1] = P[i]$

$q \leftarrow q + 1$, $\pi[i] \leftarrow q$ ($\pi[5] \leftarrow 3$)

$q = 3$

$i = 6$, $P[q + 1] = P[4] =$ b, $P[i] = P[6] =$ b, $P[q + 1] = P[i]$

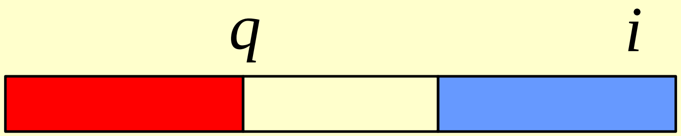
$q \leftarrow q + 1$, $\pi[i] \leftarrow q$ ($\pi[6] \leftarrow 4$)

```
q ← 0
for i ← 2 to n
do while q > 0 and P[q + 1] ≠ P[i]
      do q ← π[q]
      if P[q + 1] = P[i]
      then q ← q + 1
      π[i] ← q
```

$P =$ abababbabca

■ Knuth-Morris-Pratt algorithm – sample trace

- Example



$q = 4$

$i = 7$, $P[q + 1] = P[5] = \text{a}$, $P[i] = P[7] = \text{a}$, $P[q + 1] = P[i]$

$q \leftarrow q + 1$, $\pi[i] \leftarrow q$ ($\pi[7] \leftarrow 5$)

$q = 5$

$i = 8$, $P[q + 1] = P[6] = \text{b}$, $P[i] = P[8] = \text{b}$, $P[q + 1] = P[i]$

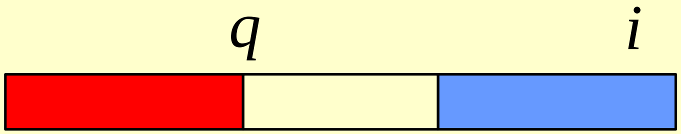
$q \leftarrow q + 1$, $\pi[i] \leftarrow q$ ($\pi[8] \leftarrow 6$)

```
q ← 0
for i ← 2 to n
do while q > 0 and P[q + 1] ≠ P[i]
      do q ← π[q]
      if P[q + 1] = P[i]
      then q ← q + 1
      π[i] ← q
```

P = abababbabca

■ Knuth-Morris-Pratt algorithm – sample trace

- Example



$q = 6$

$i = 9$, $P[q + 1] = P[7] = \text{a}$, $P[i] = P[9] = \text{c}$, $P[q + 1] \neq P[i]$

$q \leftarrow \pi[q]$ ($q \leftarrow \pi[6] = 4$)

$P[q + 1] = P[5] = \text{a}$, $P[i] = P[9] = \text{c}$, $P[q + 1] \neq P[i]$

$q \leftarrow \pi[q]$ ($q \leftarrow \pi[4] = 2$)

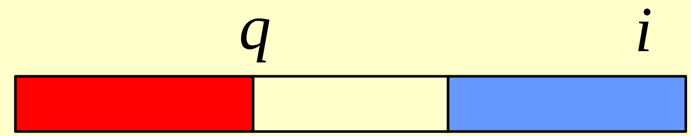
$P[q + 1] = P[3] = \text{a}$, $P[i] = P[9] = \text{c}$, $P[q + 1] \neq P[i]$

$q \leftarrow \pi[q]$ ($q \leftarrow \pi[2] = 0$), $\pi[i] \leftarrow q$ ($\pi[9] \leftarrow 0$)

$P = \text{abababbabca}$

```
q ← 0
for i ← 2 to n
do while q > 0 and P[q + 1] ≠ P[i]
      do q ← π[q]
      if P[q + 1] = P[i]
      then q ← q + 1
      π[i] ← q
```

■ Knuth-Morris-Pratt algorithm – sample trace



- Example

  $q = 0$

  $i = 10$, $P[q + 1] = P[1] = \text{a}$, $P[i] = P[10] = \text{a}$, $P[q + 1] = P[i]$

  $q \leftarrow q + 1$, $\pi[i] \leftarrow q$ ($\pi[10] \leftarrow 1$)

$P = \text{ababababca}$

```
q ← 0
for i ← 2 to n
do while q > 0 and P[q + 1] ≠ P[i]
      do q ← π[q]
      if P[q + 1] = P[i]
      then q ← q + 1
      π[i] ← q
```

■ Knuth-Morris-Pratt algorithm

Theorem Algorithm Compute-Prefix-Function(P) computes π in O($|P|$) steps.

Proof. The cost of the **while** statement is proportional to the number of times q is decremented by the statement $q \leftarrow \pi[q]$ following **do** in line 6. The only way k is increased is by assigning $q \leftarrow q + 1$ in line 8. Since $q = 0$ initially, and line 8 is executed at most ($|P| – 1$) times, we conclude that the **while** statement on lines 5 and 6 cannot be executed more than $|P|$ times. Thus, the total cost of executing lines 5 and 6 is O($|P|$). The remainder of the algorithm is clearly O($|P|$), and thus the whole algorithm takes O($|P|$) time.