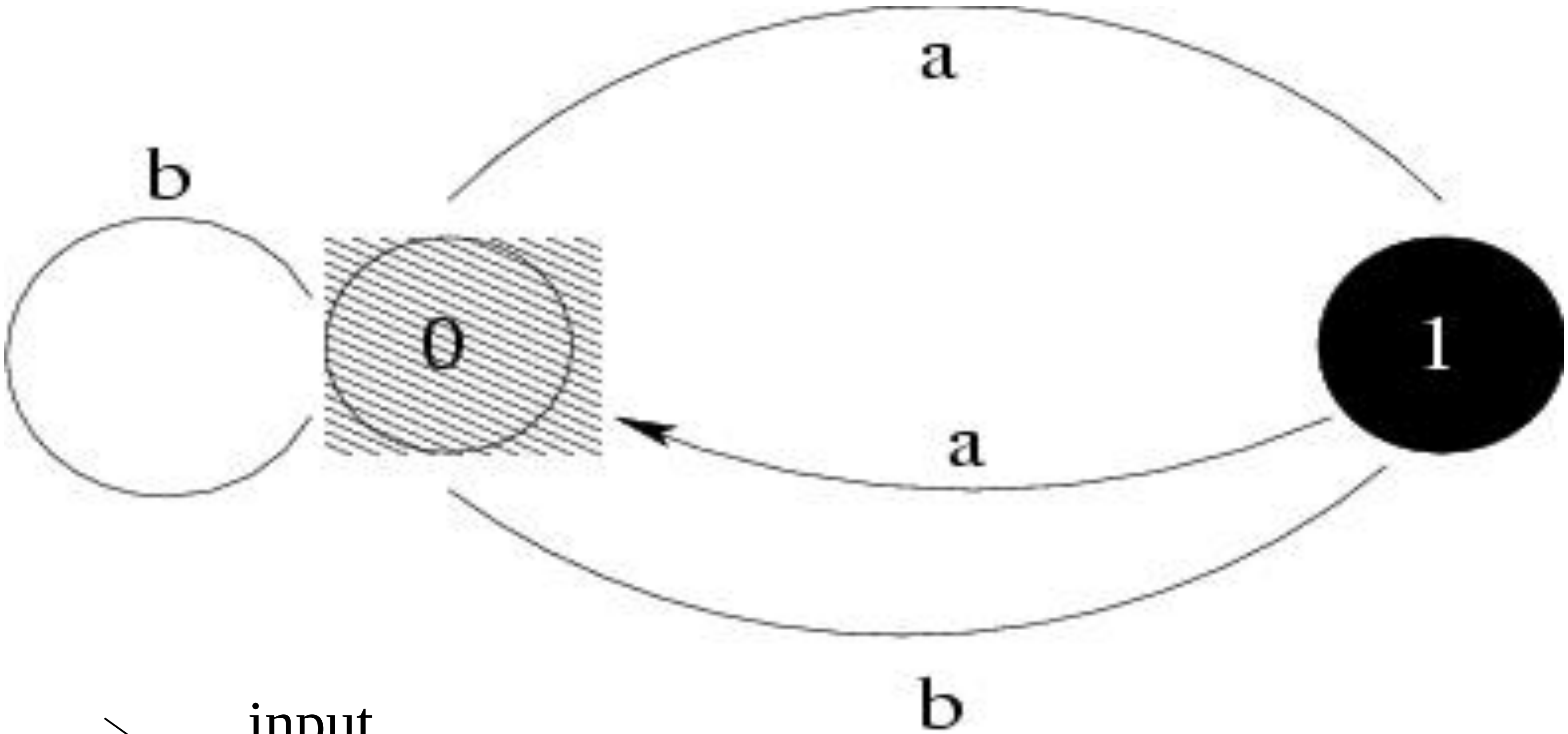
String Matching (continued)

String matching with finite automata

- The string-matching automaton is very efficient: it examines each character in the text exactly once and reports all the valid shifts in $O(n)$ time.

The basic idea is to build a 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.



	input	
State	a	b
0	1	0
1	0	0

Given pattern: a^{2k+1}
Input string = abaaa
Start state: 0
Terminate state: 1

Figure 1: An automaton.

- The construction of the string-matching automaton is based on the given pattern. The time of this construction may be $O(m^3|\Sigma|)$.

Finite automata:

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

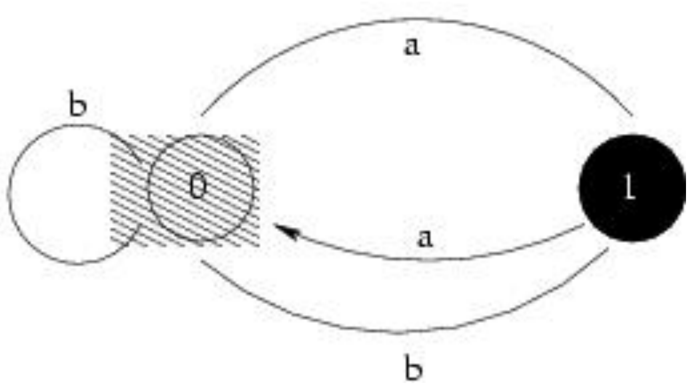
- Q is a finite set of states.
- $q_0 \in Q$ is the start state.
- $A \in Q$ is a distinguish set of accepting states.
- Σ is a finite input alphabet
- δ is a function from $Q \times \Sigma$ into Q, called the transition function of M.

- 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.

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	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.

- final-state function ψ: from Σ^* to Q such that $\psi(w)$ is the state in which M ends up after scanning the string w.

Thus, M accepts w if and only if $\psi(w) \in A$.

For example, $\psi(abbaa)=0$, and $\psi(bbabaaa)=1$.

- $\psi(\varepsilon) = q_0$, (* empty string does not change any current state *)
- $\psi(wa) = \delta(\psi(w),a)$ for $w \in \Sigma^*$, $a \in \Sigma$.

The construction of string-matching automaton.

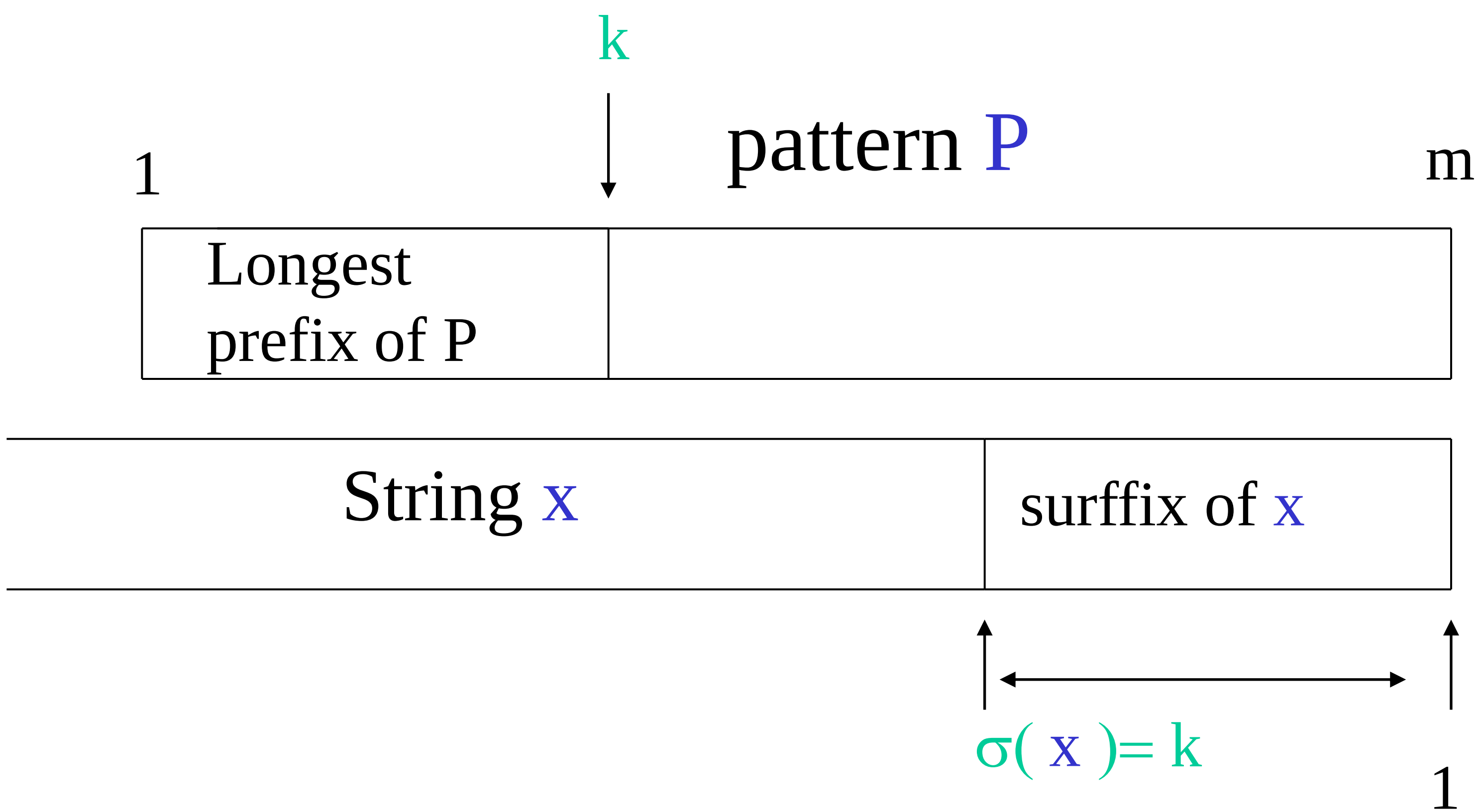
- There exists a string-matching automaton for every pattern P.

A suffix function w.r.t. pattern $P[1..m]$, σ , is 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: $\sigma(x)=\max\{k: P_k \sqsupset x\}$. 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.

The definition of σ(x)



P=abccbddca, x=cddaabccb,

$\sigma(x) = ?$

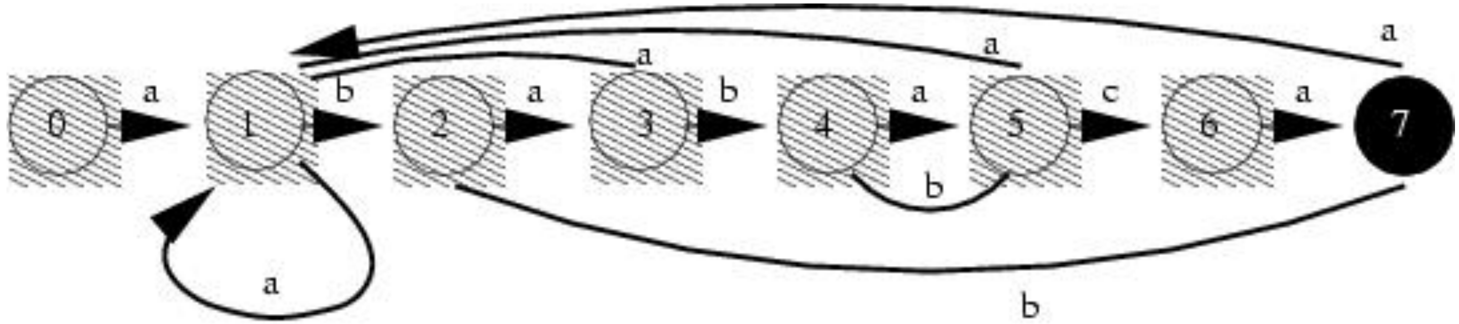
A string-matching automaton w.r.t. a given $P[1..m]$ is defined as follows.

- The state set $Q=\{0,1,\ldots,m\}$, the start state $q_0=0$, and the only accepting state $A=m$.
- The transition function δ is $\delta(q,a)=\sigma(P_q a)$.
- The machine maintains an invariant of its operation: $\psi(T_i)=\sigma(T_i)$. After scanning the first i characters of the text string T, the machine is in state $\psi(T_i)=q$, where $q=\sigma(T_i)$ is the length of the longest suffix of T_i that is also a prefix of the pattern P.

- It is proved that $\sigma(T_i a)=\sigma(P_q a)$.

That means to compute the length of the longest suffix of $T_i a$ that is prefix of P is equivalent to compute the length of the longest suffix of $P_q a$ that is the prefix of P.

- For example, P=abababca.
- δ(5, b)=4 denotes that in state 5 and reads a b. It is equivalent to P_5b=ababab and the longest prefix of P that is also the suffix of ababab is P_4=abab.
- Similarly, for δ(5,a)=1. In state 5 and reads a, which is equivalent to P_5a=ababaa and the longest prefix of P that is also the suffix of ababaa is P_1=a. How about δ(6,c)=0?



(a)

state	input a	input b	input 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	$\boxed{7}$	2	3

(c)

Figure 3: A state-transition diagram for string-matching automaton that accepts all strings ending in the string ababaca. All the left-going arrows pointing to state 0 are not shown.

FINITE-AUTOMATON-MATCHER(T,δ,m)

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 $q=m$ then
6.   print `Pattern occurs with shift' $i-m$

- Lemma (suffix-function inequality): For any string x and character a, we have $\sigma(xa) \leq \sigma(x)+1$.
- Lemma (suffix-function recursion lemma): For any string x and character a, if $q=\sigma(xa)$, then $\sigma(xa)=\sigma(P_q a)$.
- Theorem: If ψ is the final-state function of a string-matching automaton for a given pattern P and $T[1..n]$ is an input text for the automaton, then

  $\psi(T_i)=\sigma(T_i)$ for $i=0,1,\ldots,n$

The theorem shows that the automaton keeps tracking the longest prefix of the pattern which is a suffix of what has been read so far for each step.

Computing the 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 $k \leftarrow k-1$ ($1 \le k \le m+1$)
6. until $P_k \sqsupset P_q a$ ($\sum k$)
7. $\delta(q,a) \leftarrow k$
8. return δ

Example

- P= a b a b a c a
- q= 3 (implies text is … a b a …) (step 2)
- a $\leftarrow$ Σ (step 3)
- k = min(7+1, 3+2)=5, k-1=4, (steps 4,5)

  $p_4 \sqsupset p_3$ a ? No. k $\leftarrow$ k-1=3 (step 5)

  $p_3 \sqsupset p_2$ a ? Yes. $\delta(2,a) \leftarrow 3$ (steps 6,7)

  b $\leftarrow$ Σ (step 3)

  k = min(7+1, 3+2)=5, k-1=4, (steps 4,5)

  $p_4 \sqsupset p_3$ b ? Yes. $\delta(3,b) \leftarrow 4$ (steps 6,7)

  …

- This procedure builds $\delta(q,a)$ is a straight-forward way by definition. It considers all states q and all characters in Σ. For each combination, to find the the largest k such that $P_k \sqsupset P_q a$. The worst-case time complexity is

  $O(m^3 | \Sigma |)$.

- Questions to consider
- Given pattern P=abba, $\Sigma=\{a,b\}$, construct its automaton. Show how the automaton works for text T[1..12]=baabbabbaaba, using FINATE-AUTOMATON-MATCHER(T, δ, m).
- We call a pattern P non-overlappable if

  $P_k \sqsupset P_q$ implies $k=0$ or $k=q$.

  Describe the state transition diagram of the string-matching automaton for a non-overlappable pattern.

The Knuth-Morris-Pratt algorithm

- The most expensive part of the string matching automaton method is to build the transition function δ, which takes $O(m^3|\Sigma|)$ time (or at least $O(m|\Sigma|)$ time).
- The KMP algorithm avoids to directly compute δ. Instead, it computes an auxiliary function $\pi[1..m]$ pre-computed from pattern P in $O(m)$ time.
- The transition function δ can be obtained from array π in an efficient amortized constant time when the algorithm runs on a text.

The prefix function π for a pattern P:

it encapsulates the knowledge about how the pattern P matches against shifts of itself. Therefore, the knowledge can be used to avoid the useless shifts in the naive method or to avoid to pre-compute δ in the automaton method.

Notations (reminder)

Σ: alphabet, Σ^*: set of all finite-length string,

ε: empty string. w: a string. $w \sqsubset x$: w is prefix of x, $w \sqsupset x$: w is suffix of x.

Q: a finite set of states, q_0: start state, A: accepting states. δ: transition function of M. $\delta(q,a)=q'$.

ψ: final-state function. $\psi(w)$ is the state M ends up after M scanning w. $\psi(wa)=\delta(\psi(w),a)$.

σ: the suffix function corresponding to pattern P. $\sigma(x) = \max \{k : P_k \sqsupset x\}$.

- Given that pattern characters $P[1..q]$ match text characters $T[s+1..s+q]$, what is the least shift $s'>s$ such that

  $P[1..k]=T[s'+1..s'+k]$, where $s'+k=s+q$?

- The above equation is equivalent to find the largest $k<q$ such that $P_k \sqsupset P_q$. Then, $s'=s+(q-k)$ is the potential next valid shift.
- Given a pattern $P[1..m]$, the prefix function for the pattern P is the function π: $\{1,2,...,m\} \rightarrow \{0,1,...,m-1\}$ such that $\pi[q]=\max\{k:|k<q \,\&\, P_k \sqsupset P_q\}$.

- KMP-MATCHER(T,P)

```
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0 (* number of characters matched *)
for i ← 1 to n  (*scan the text from left to right *)
    do while q>0 & P[q+1] ≠ T[i]
            do q ← π[q]  (* next character does not match *)
        if P[q+1]=T[i]
            then q ← q+1  (* next character matches *)
        if q=m (* is all of P matched? *)
            then print `Pattern occurs with shift' i-m
                q ← π[q]  (* look for the next match *)
```

- COMPUTE-PREFIX-FUNCTION(P)

```
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
    do while k>0 & P[k+1] ≠ P[q]
        do k ← π[k]
      if P[k+1]=P[q]
        then k ← k+1
      π[q] ← k
return π
```

- Time Complexity:

  COMPUTE-PREFIX-FUNCTION(P) takes $\Theta(m)$ time.

(By amortized analysis.)

KMP-MATCHER(T,P) takes $\Theta(n)$ time.

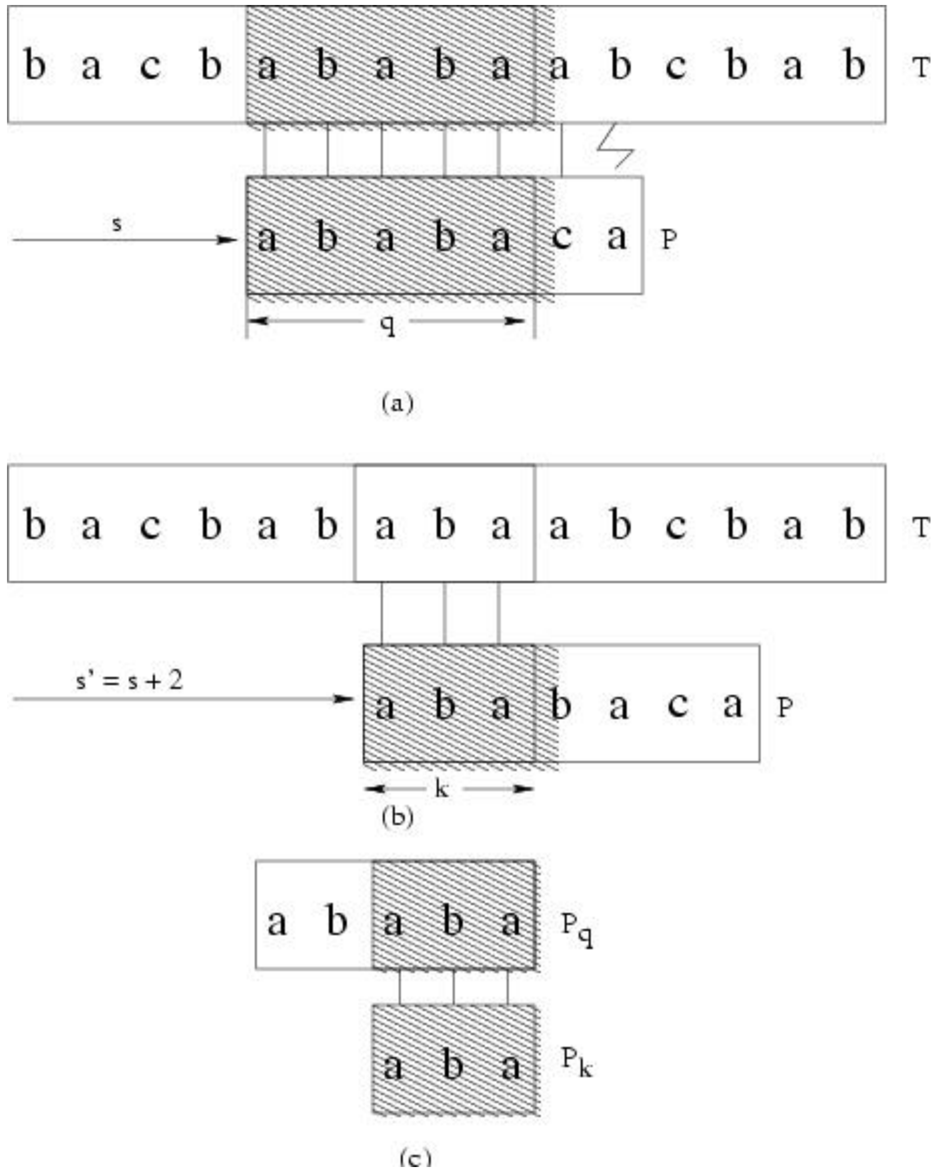


Figure 4: A demonstration for how to obtain the valid shift from the previous partial matching. It is clearly that the next potential valid shift is s'=s+(q-π[q]), where π[5]=3.

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[ι]	0	0	1	2	3	4	5	6	0	1

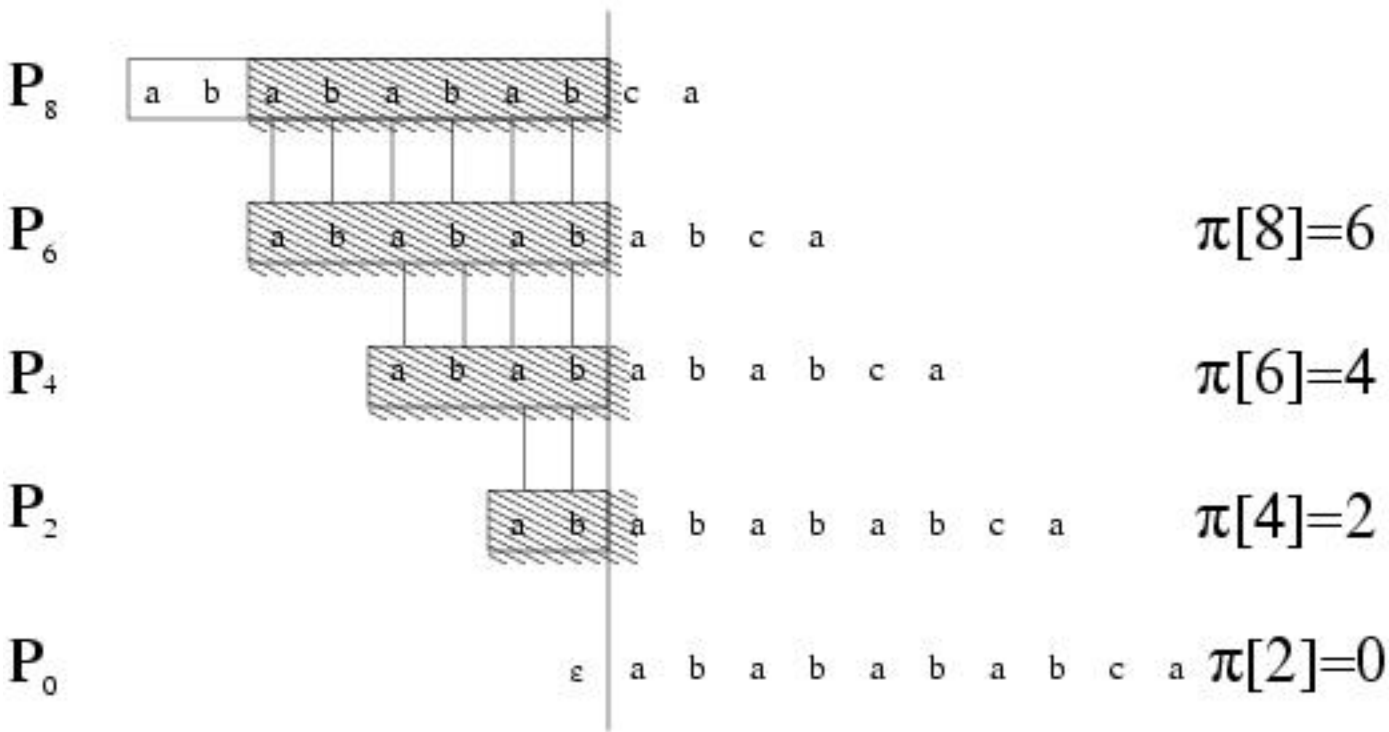


Figure 5: A demonstration for how to obtaining the π function of P