EEECE7205: Fundamentals of Computer Engineering

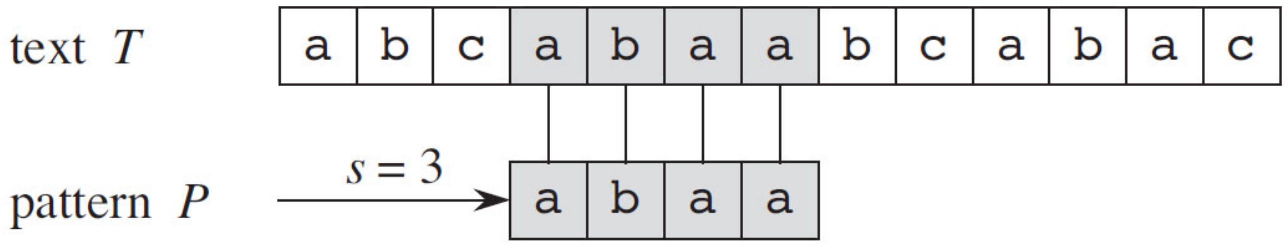
String Matching

The String-Matching Problem

- Text-editing programs frequently need to find all occurrences of a pattern in the text or to apply automatic spelling correction.
- DNA sequences are searched for particular patterns.
- Internet search engines also need algorithms to find Web pages relevant to queries.
- Efficient algorithms for this problem are called "***string matching***".
- Given a text represented as an array $T\,[1..\,n]$ of length n, the string-matching algorithm is to find in T the occurrences of a pattern $P\,[1\,..\,m]$ of length $m \leq n$.
- Assumption: the elements of P and T are characters drawn from a finite alphabet $\sum$.
- The character arrays P and T are often called ***strings*** of characters.

String Matching Definitions

- Referring to the figure below, 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*)
- This means if $0 \leq s \leq n - m$ then $T[s + 1..\, s + m] = P[1..\, m]$ (that is, $T[s + j] = P[j]$), for $1 \leq j \leq m$).
- If *P* occurs with shift *s* in *T*, then we call *s* a ***valid shift***; otherwise, we call *s* an ***invalid shift***.
- The ***string-matching problem*** is the problem of finding **all** valid shifts with which a given pattern *P* occurs in a given text *T*.



Notation and Terminology (1 of 2)

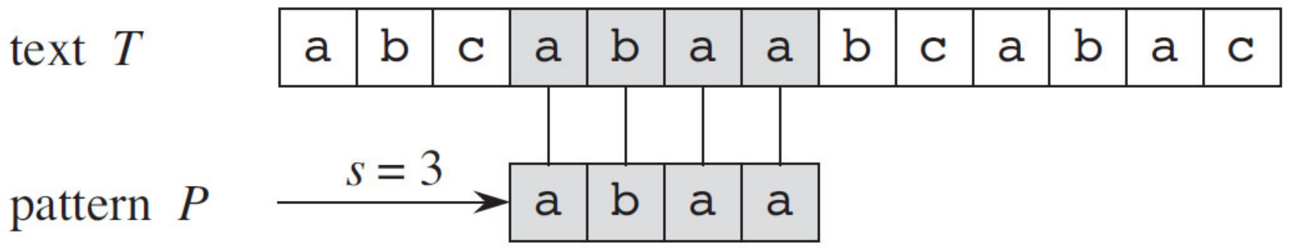
- Σ^* = set of all finite-length strings formed using characters from alphabet Σ
- Empty string: ε (also belongs to Σ^*)
- $|x|$ = length of string x
- The ***concatenation*** of two strings x and y, denoted xy, has length $|x| + |y|$ and consists of the characters from x followed by the characters from y.
- w is a ***prefix*** of x: $w \sqsubset x$ (e.g., $ab \sqsubset abcca$)
- w is a ***suffix*** of x: $w \sqsupset x$ (e.g., $cca \sqsupset abcca$)
  - In both cases $|w| \le |x|$
- The prefix and suffix relations are *transitive.*
- The empty string ε is both a suffix and a prefix of any string.

Notation and Terminology (2 of 2)

- We denote the k-character prefix $R[1..k]$ of the string $R[1..n]$ by R_k. Thus, $R_0 = \varepsilon$ and $R_n = R$.

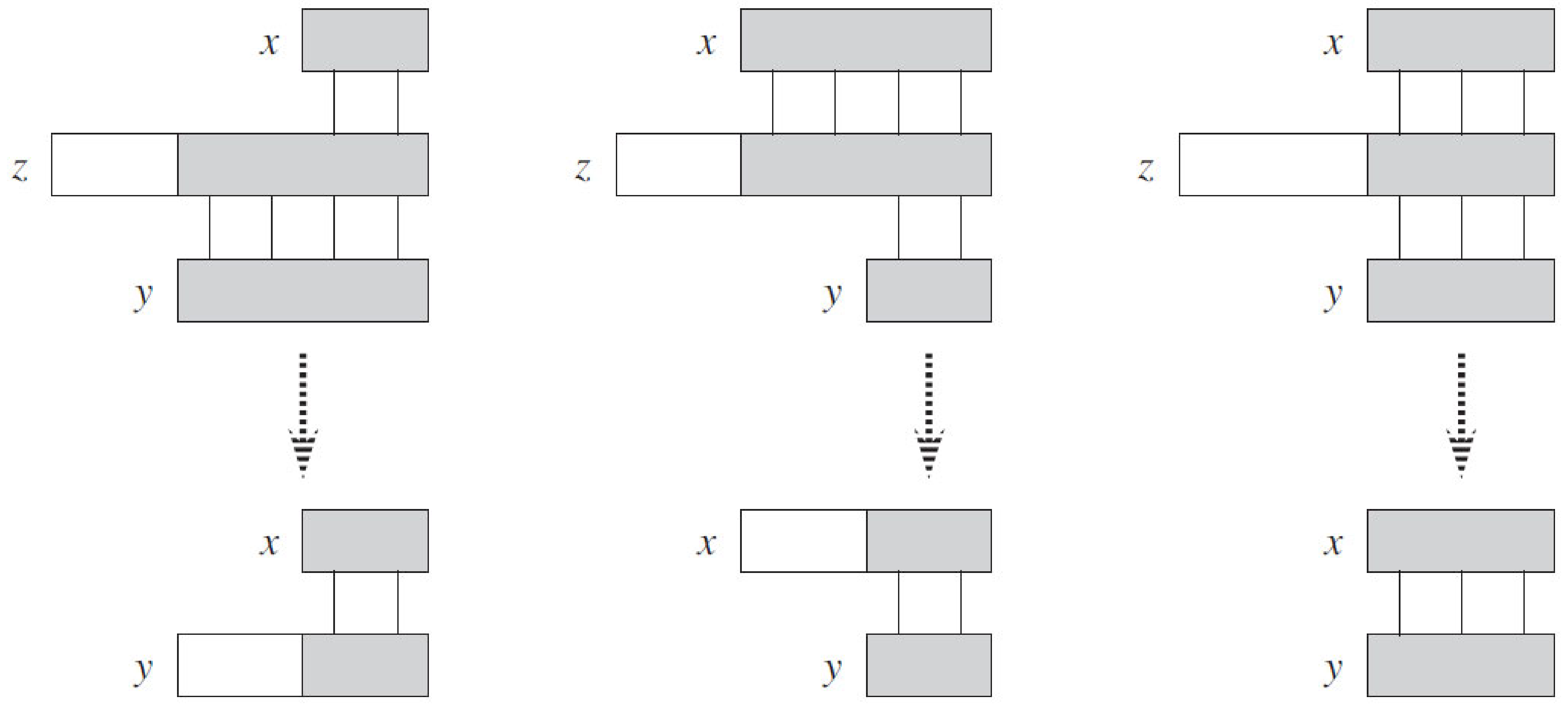
- Using this notation, we can state the string-matching problem as that of:

  finding all shifts s in the range $0 \le s \le n - m$ such that pattern $P \sqsupset T_{s+m}$. Where T is a string of length n and pattern P is of length $m \le n$.



Overlapping-Suffix Lemma

- Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \leq |y|$, then $x \sqsupset y$. If $|x| \geq |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

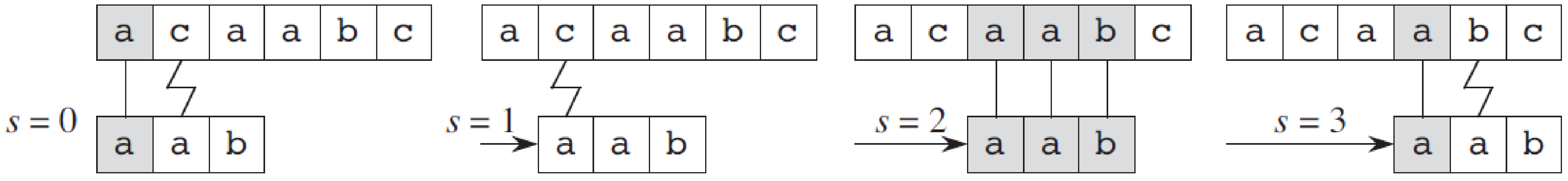


The Naive String-Matching Algorithm

- The naive algorithm finds all valid shifts using a loop that checks the condition $P[1 .. m] = T[s + 1 .. s + m]$ for each of the $n - m + 1$ possible values of s.

What is the best and worst big *O* of this algorithm?

```
NAIVE-STRING-MATCHER(T, P)
1  $n = T.length$
2  $m = P.length$
3  for $s = 0$ to $n - m$
4      if $P[1..m] == T[s+1..s+m]$
5          print "Pattern occurs with shift" $s$
```



(a) (b) (c) (d)

The Naive Algorithm Worst Case

- The following example shows the worst-case scenario to search for the pattern *BBC* (m = 3) in a text of n characters.

```
1.      BBBBBBBBBBBBBBBBBBBBBBBBBC
        BBC  3 comparisons
2.      BBBBBBBBBBBBBBBBBBBBBBBBBC
         BBC  3 comparisons
3.      BBBBBBBBBBBBBBBBBBBBBBBBBC
          BBC  3 comparisons
4.      ....
```

- Total number of comparisons = m (n-m+1)
- Time efficiency = O(nm)

The Naive Algorithm Best Case

- The following example shows the best-case scenario to search for the pattern *BBC* (m = 3) in a text of n characters.

1. CNNNNNNNNNNNNNNNNNNNNNNNNNN

   BBC 1 comparisons made
2. CNNNNNNNNNNNNNNNNNNNNNNNNNN

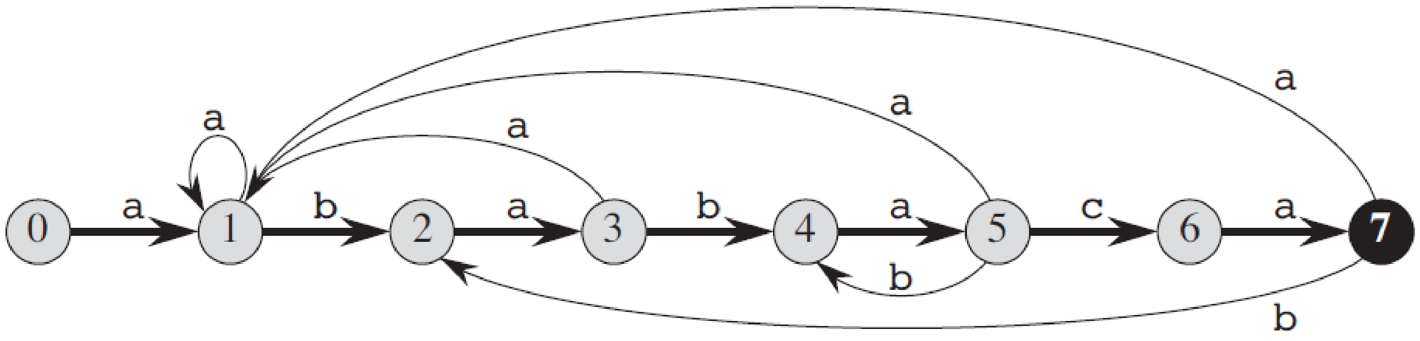
    BBC 1 comparisons made
3. CNNNNNNNNNNNNNNNNNNNNNNNNNN

     BBC 1 comparisons made
4. . . . .

- Total number of comparisons = n-m+1
- Time efficiency = O(n)

Finite Automata

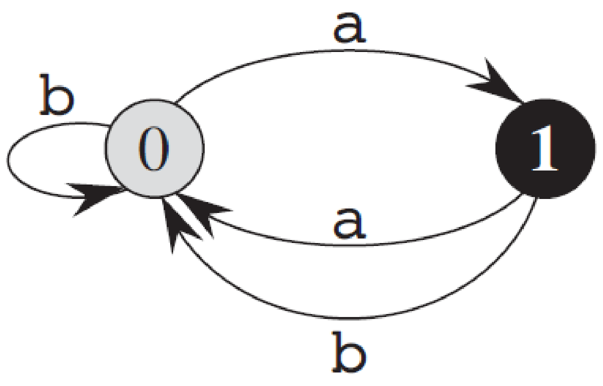
- String matching using finite automata avoids testing useless shifts as in the naive pattern-matching algorithm.
- A ***finite automaton*** M is a 5-tuple (Q, q_0, A, Σ, δ), where
  - Q is a finite set of ***states***,
  - $q_0 \in Q$ is the ***start state***,
  - $A \subseteq Q$ is a distinguished 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.



Finite Automata Example

- In the shown example:
  - $Q = \{0, 1\}$,
  - $q_0 = 0$,
  - $A = \{1\}$
  - $\sum = \{a, b\}$,
  - $\delta(0, a) = 1$
    $\delta(0, b) = 0$
    $\delta(1, a) = 0$
    $\delta(1, b) = 0$

	input	
state	a	b
0	1	0
1	0	0

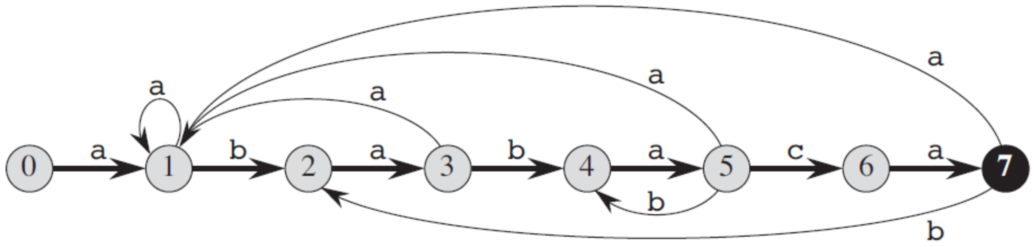


The Final-State Function

- The function Φ defined on a finite automaton *M* is called the ***final-state function***.
- For a string $w \in \Sigma^*$, $\Phi(w)$ returns the state in which *M* ends up after *M* scans the string *w*.
- *M* accepts a string *w* if and only if $\Phi(w) \in A$ (the set of accepting states).
- We define the function Φ recursively, using the transition function:

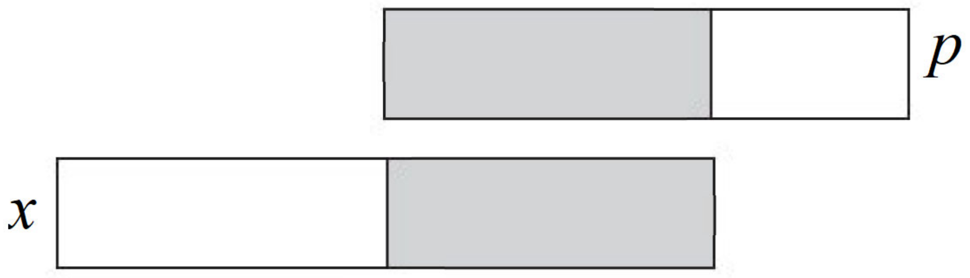
$$\phi(\varepsilon) = q_0 ,$$ Where ε is the empty string

$$\phi(wa) = \delta(\phi(w), a) \quad \text{for } w \in \Sigma^*, a \in \Sigma .$$



The Suffix Function

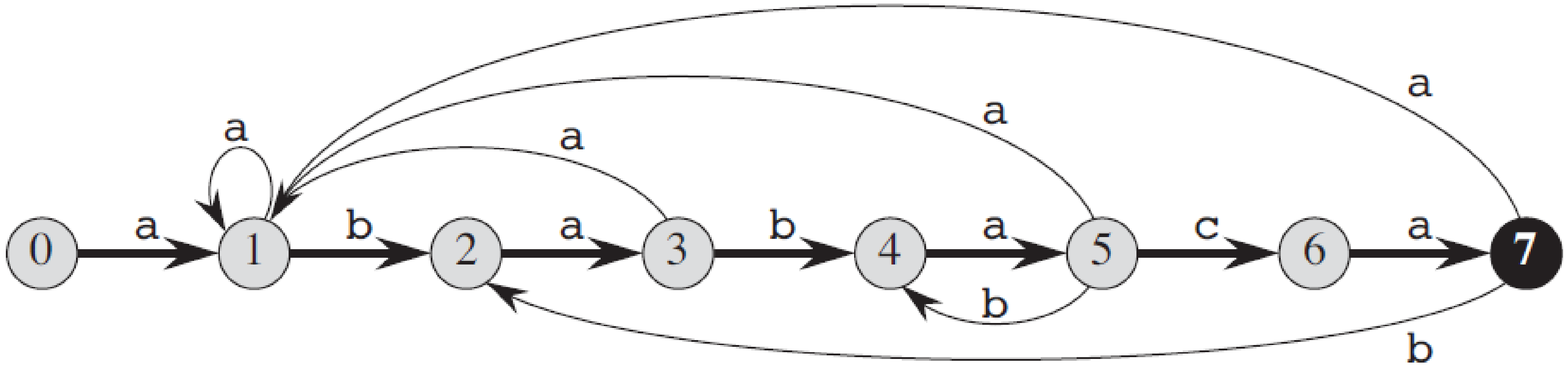
- For a given pattern $P[1 .. m]$, the ***suffix function*** σ maps Σ^* to $\{0, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x.



- Example:
  - If pattern P = ab, we have $\sigma(\varepsilon) = 0$,
  - σ(ccaca) = 1,
  - σ(ccab) = 2.
  - For a pattern P of length m, we have $\sigma(x) = m$ if and only if $P \sqsupset x$.

String-Matching Automata (1 of 2)

- For a given pattern *P*, we construct a string-matching automaton in a preprocessing step before using it to search the text string.
- The figure illustrates how we construct the automaton for the pattern *P* = ababaca
- Some edges corresponding to failing matches are omitted; by convention, if a state *i* has no outgoing edge labelled *x* then δ(*i*, *x*) = 0.



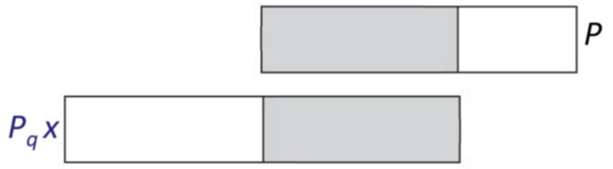
- Sample operation:

What is the big *O* of the matching process?

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

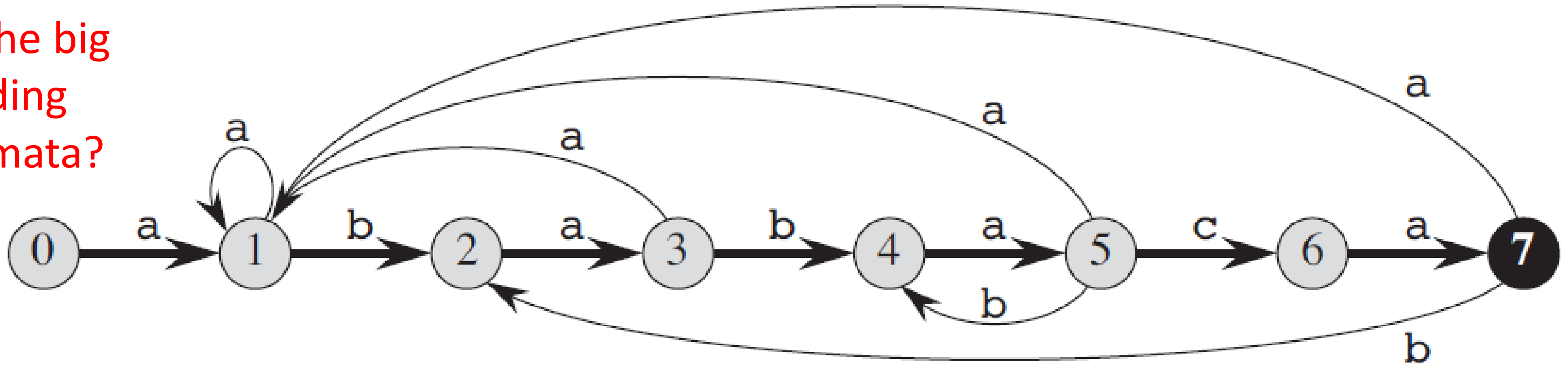
String-Matching Automata (2 of 2)

- We define the string-matching automaton that corresponds to a given pattern $P[1 \,..\, m]$ as follows:
  - 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.
  - The transition function δ is defined by equation $\delta(q, x) = \sigma(P_q x)$ for any state q and character x



- Example1: for P = ababaca, q = 5, x = b → δ(5, b) = σ(ababab) = 4
- Example2: for P = ababaca, q = 5, x = a → δ(5, a) = σ(ababaa) = 1
- Example3: for P = ababaca, q = 5, x = c → δ(5, c) = σ(ababac) = 6

What is the big O of building the automata?



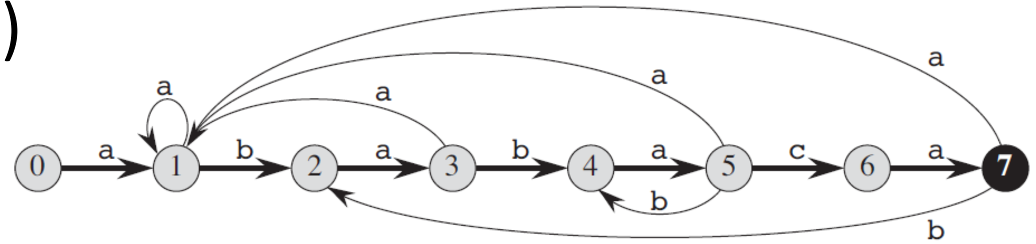
Finite Automata Matcher Algorithm

- The following algorithm uses an automaton (represented by its transition function δ) to find occurrences of a pattern *P* of length *m* in an input text *T* [1..*n*].

```
FINITE-AUTOMATON-MATCHER($T, \delta, m$)
1  $n = T.length$
2  $q = 0$
3  for $i = 1$ to $n$
4      $q = \delta(q, T[i])$
5      if $q == m$
6          print "Pattern occurs with shift" $i - m$
```

Representing the automaton

- Preprocessing time = $O(m|\Sigma|)$
- Matching time = $\Theta(n)$

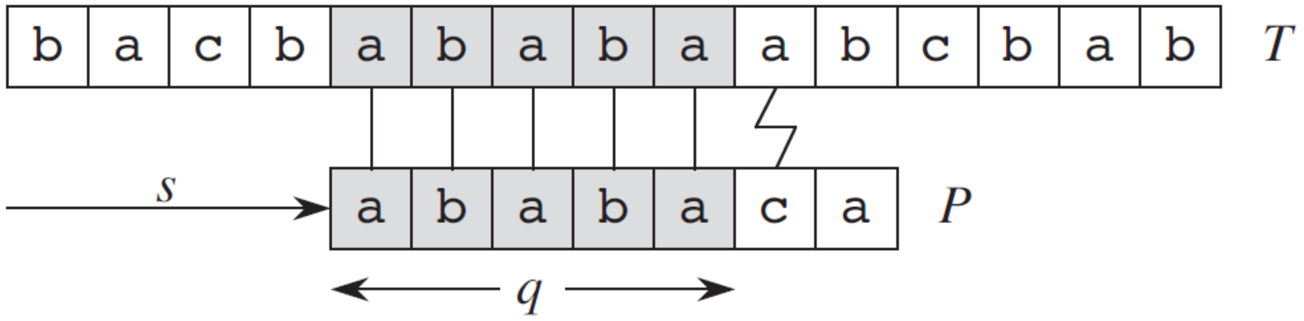


The Knuth-Morris-Pratt Algorithm

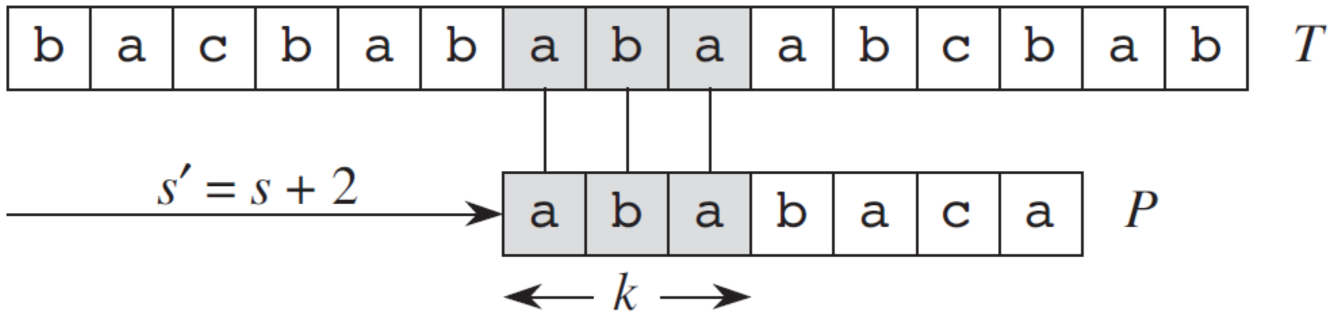
- This algorithm uses an auxiliary function π that allows us to compute the transition function δ efficiently "on the fly" as needed.
- The array π is pre-computed from the pattern in time $\Theta(m)$ and is stored in an array $\pi[1..\, m]$.
- For any state $q = 0, 1, \ldots , m$ and any character $a \in \sum$, the value $\pi[q]$ contains the information we need to compute $\delta(q , a)$.
- Since the array π has only m entries, whereas δ has $\Theta(m\ |\sum|)$ entries, we save a factor of $|\sum|$ in the pre-processing time by computing π rather than δ.
  - The idea here comes from the fact that in computing δ, there is no need to consider the characters in $\sum$ that are not in P as they will always bring us back to state 0.

Comparing *P* With Itself

- The entries in array π are precomputed by comparing the pattern *P* with itself.
- The shown pattern *P* = *ababaca* aligns with a text *T* so that the first *q* = 5 characters match.

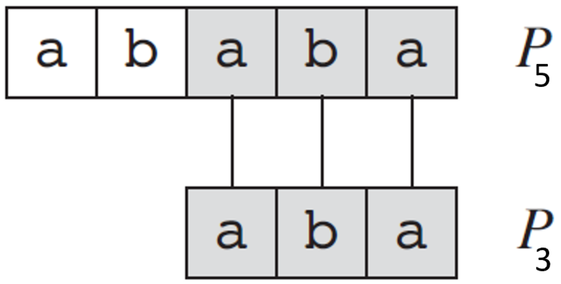


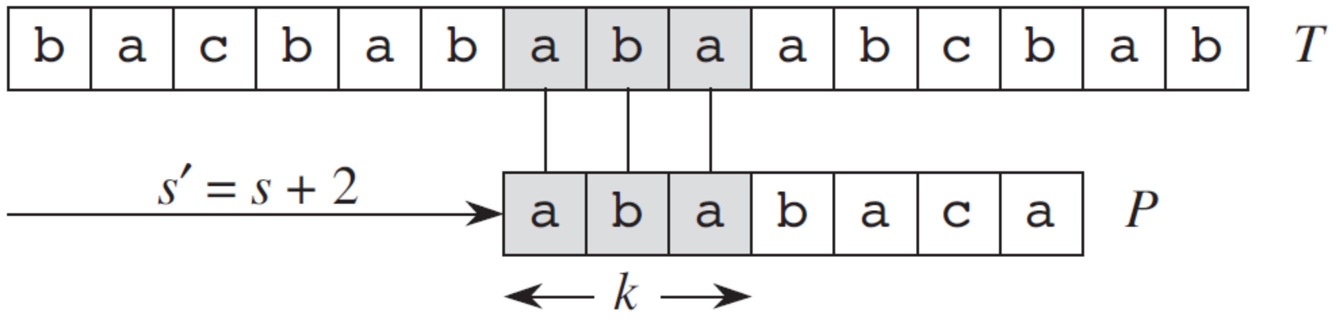
- Using only our knowledge of the 5 matched characters, we can deduce that a shift of *s* + 1 is invalid, but that a shift of *s'* = *s*+2 is potentially valid.



An Example of a π's Entry

- Here, we see that P_3 is the longest prefix of P that is also a proper suffix of P_5.
- We represent this pre-computed information in the array π, so that π[5]=3.
- Now we can continue our matching process from where we stopped at T and compare to character P[π[5]+1 = 4], which means the new shift $s' = s + (5 - \pi[5])$.
  - Comparing to the automata, this also means that at state q, the next state after a match is q+1 and the next state after a mismatch is π[q]. However here we will need to compare the mismatched character again.



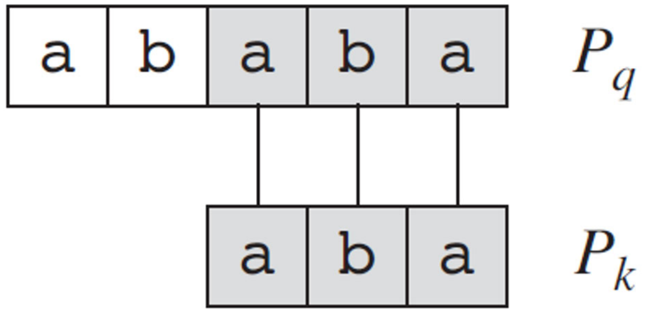


Computing π

- Given a pattern *P* of *m* characters where *q* characters have matched successfully at shift *s*, the next potentially valid shift is at $s' = s + (q - \pi[q])$
  where the ***prefix function*** for the pattern *P* is the function π: {1, 2, …, m} → {0, 1, …, m-1} such that:

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\} \ .$$

Which is the longest prefix of *P* that is also a proper suffix of P_q

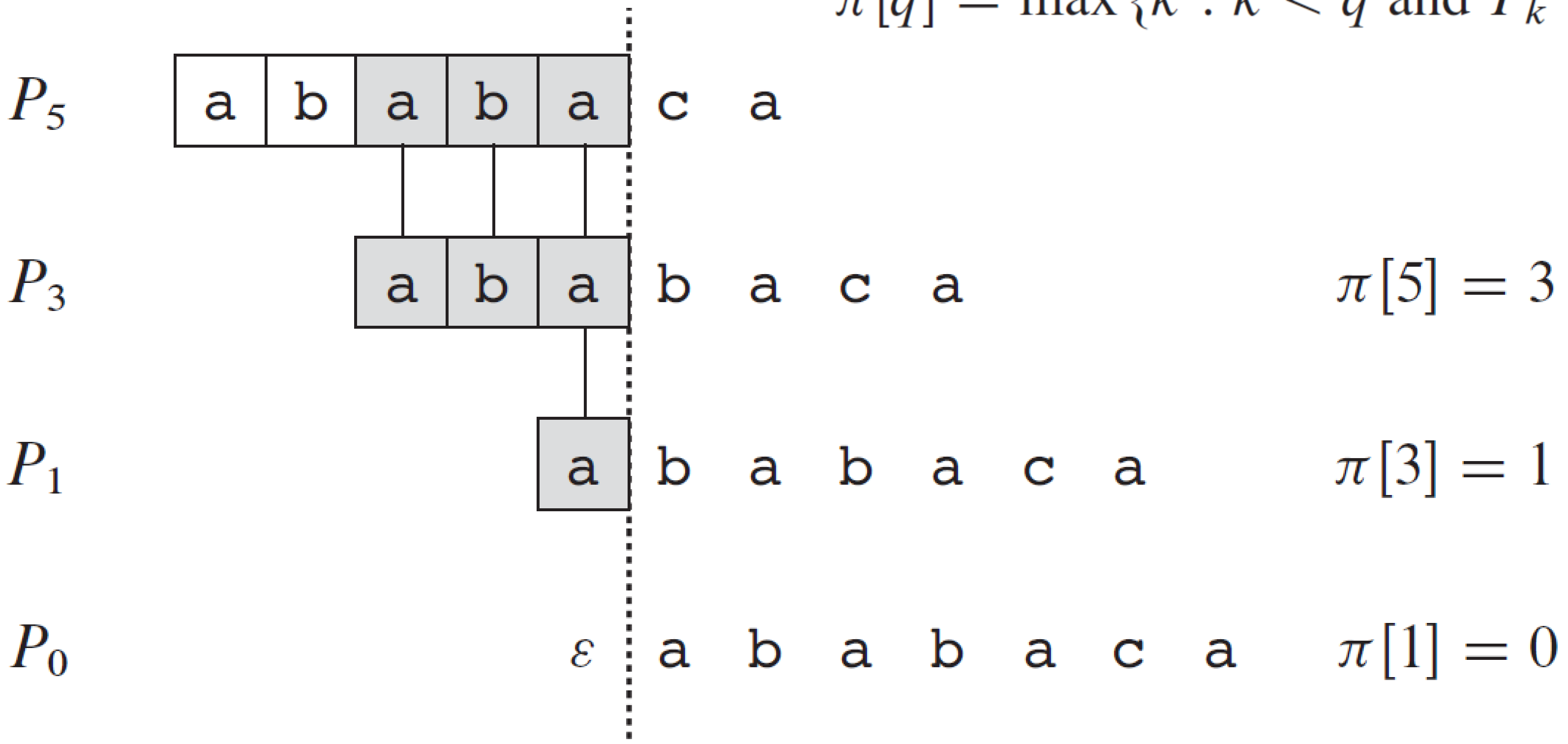


Prefix Function Example

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

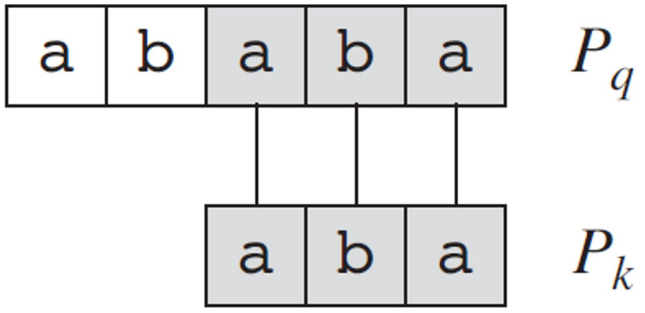
The longest prefix of *P* that is also a proper suffix of P_q

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\} .$$



KMP Preprocessing Algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
1   $m = P.length$
2   let $\pi[1..m]$ be a new array
3   $\pi[1] = 0$
4   $k = 0$
5   for $q = 2$ to $m$
6       while $k > 0$ and $P[k+1] \neq P[q]$
7           $k = \pi[k]$
8       if $P[k+1] == P[q]$
9           $k = k + 1$
10      $\pi[q] = k$
11  return $\pi$
```



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

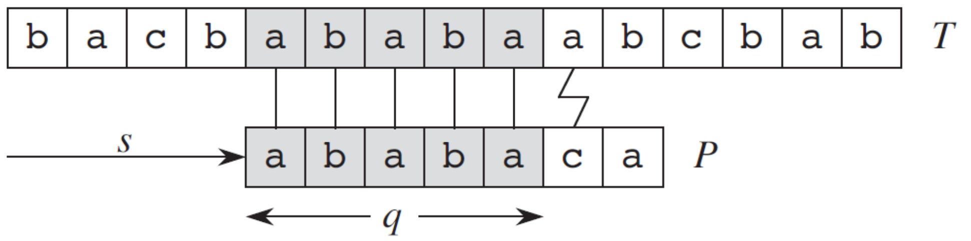
Preprocessing time is $\Theta(m)$

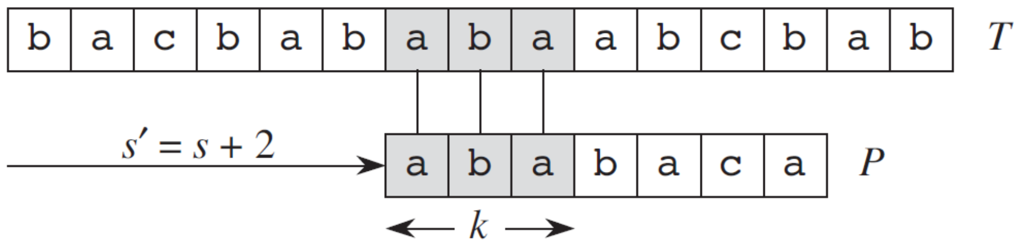
KMP Matcher Algorithm

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                  // number of characters matched
for i = 1 to n                         // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                       // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                      // next character matches
    if q == m                          // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                       // look for the next match
```

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

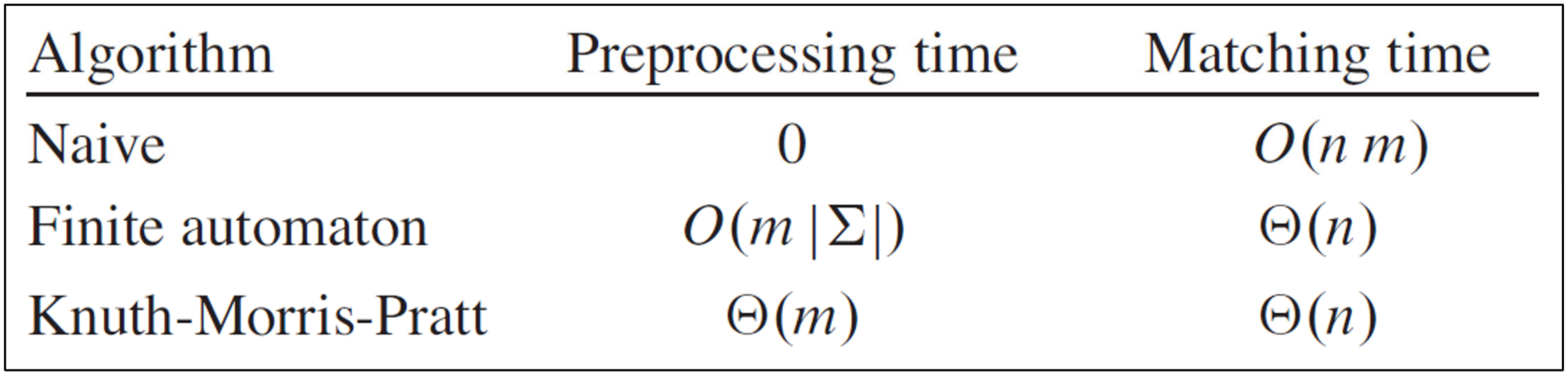
Matching time is $\Theta(n)$





Algorithms Running Time

- Except for the naive algorithm, each string-matching algorithm we studied performs some preprocessing based on the pattern and then finds all valid shifts; we call this latter phase "matching."
- The total running time of each algorithm is the sum of the preprocessing and matching times.



Algorithm	Preprocessing time	Matching time
Naive	0	$O(n\,m)$
Finite automaton	$O(m\,\lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$