Lecture #19
String Matching (2)

Algorithm

JBNU

Jinhong Jung

In This Lecture

❑ We previously study the following

- String matching problem
  - Let's match a pattern $\boldsymbol{P}$ of length m in a document $\boldsymbol{A}$ of length n
- Naïve algorithm
  - Takes $O(mn)$ time
- Rabin-Karp algorithm
  - Takes $O(m + Fn)$ time where F is # of that fingerprints are hit
  - Average case time: $O(n)$
  - Worst case time: $O(mn)$

❑ More efficient algorithm for string matching

- Automata algorithm

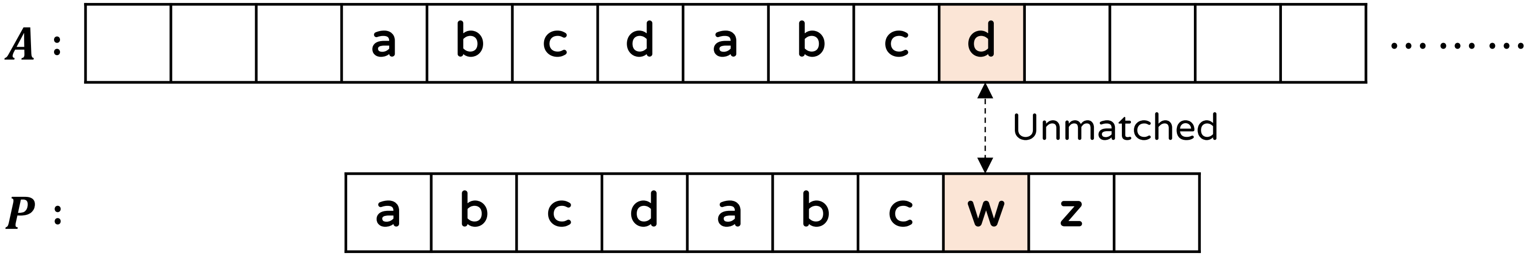
Outline

- ❑ Intuition for automata algorithm
- ❑ String matching automata
- ❑ Search phase in automata algorithm
- ❑ Automata construction phase

Intuition of Automata Algorithm

❑ How can we improve the naïve algorithm?

- Consider the following situation where $\boldsymbol{P}$ is not matched with the sub-string of $\boldsymbol{A}$



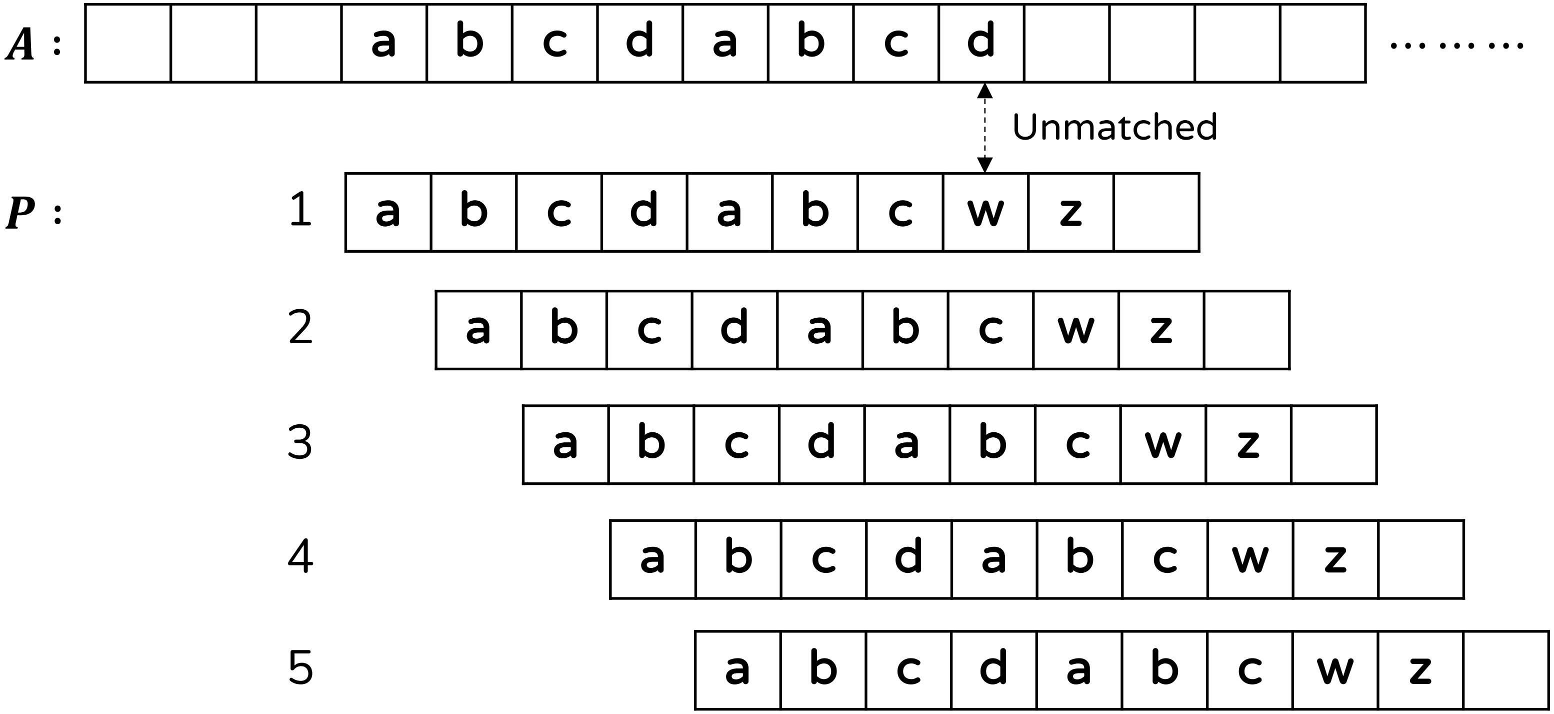
A :				a	b	c	d	a	b	c	d					………
P :				a	b	c	d	a	b	c	w	z				

Unmatched

Intuition of Automata Algorithm

❑ How can we improve the naïve algorithm?

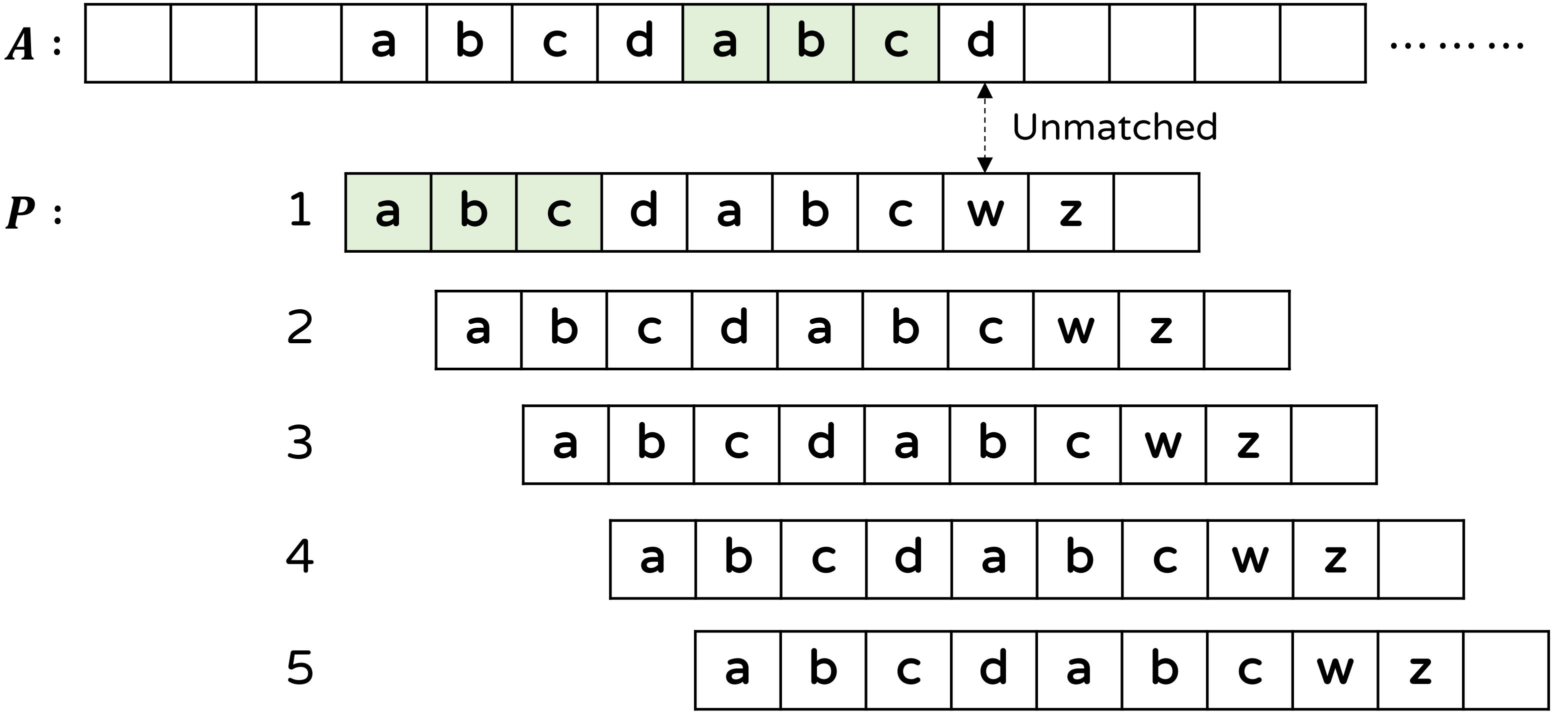
- Then, the naïve algorithm keeps searching next as the following:



Intuition of Automata Algorithm

❑ How can we improve the naïve algorithm?

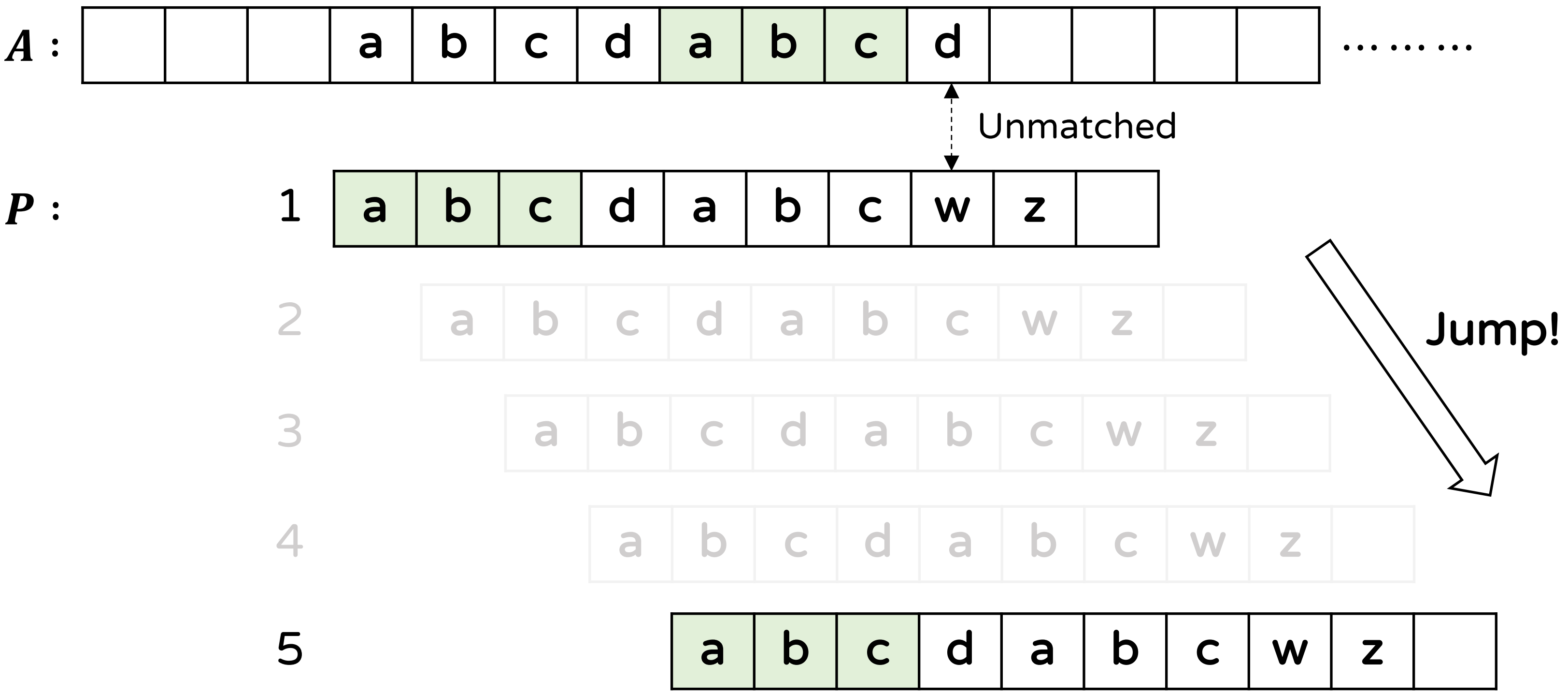
- Note that the front (prefix) of $\boldsymbol{P}$ can be partially matched with the rear (suffix) of the sub-string of $\boldsymbol{A}$.



Intuition of Automata Algorithm

❑ How can we improve the naïve algorithm?

- Using this information, we can skip Steps 2, 3, & 4 and jump to Step 5!



A : a b c d a b c d ………

Unmatched

P :

1 a b c d a b c w z

2 a b c d a b c w z

3 a b c d a b c w z

4 a b c d a b c w z

5 a b c d a b c w z

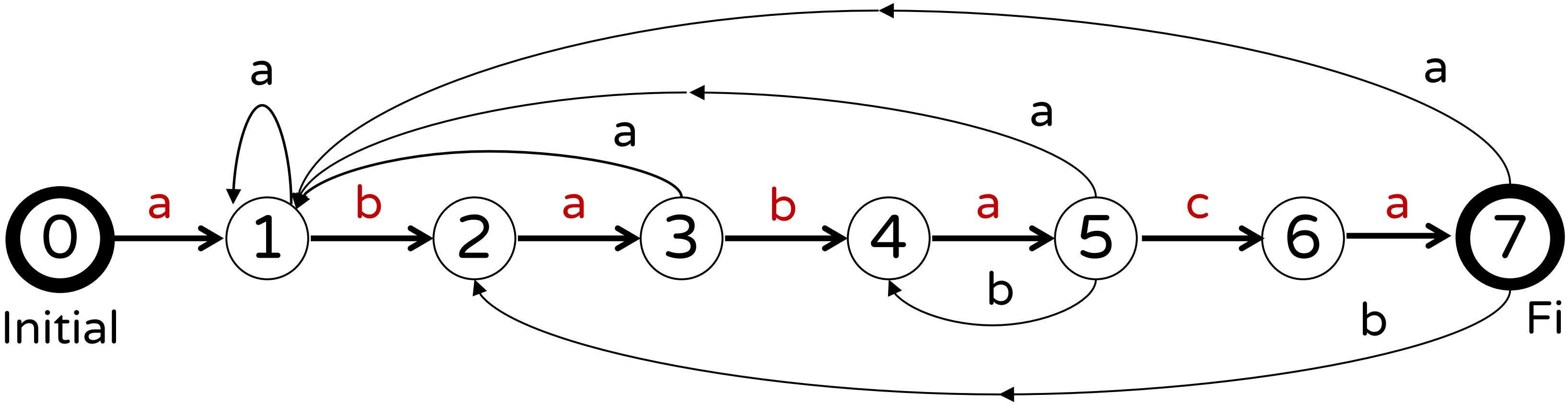
Jump!

Outline

- ❑ Intuition for automata algorithm
- ❑ String matching automata
- ❑ Search phase in automata algorithm
- ❑ Table construction phase in automata algorithm

String Matching Automata (1)

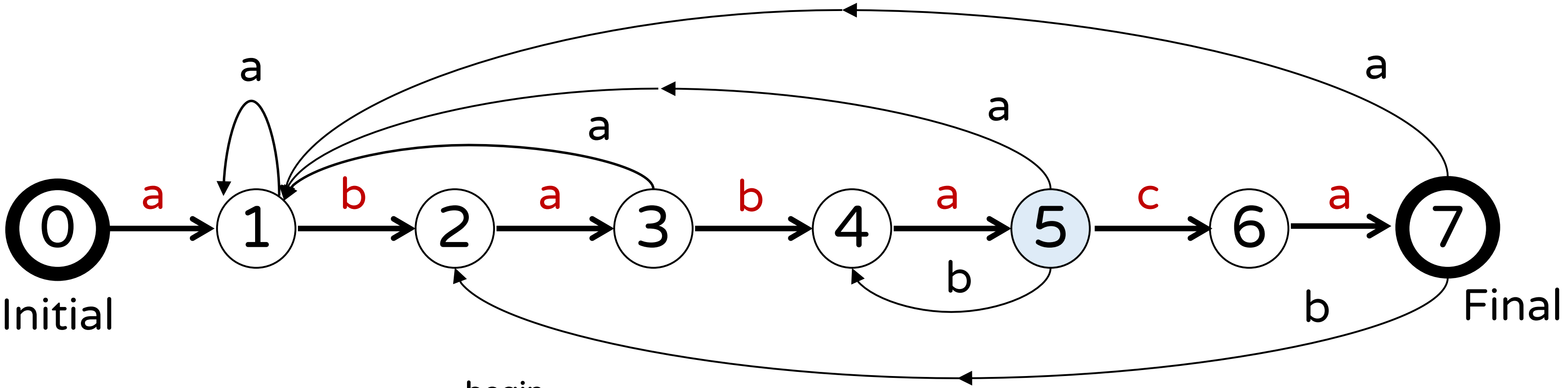
- A directed graph represents procedures of matching a pattern string
  - A node is a state while matching the pattern.
  - An edge is a transition from state to state given a label.
    - For other labels not given in the edge, go back to State 0.
  - Example of an automata of pattern “ababaca”
    - **State 0**: nothing is matched & **Final state**: “ababaca” is matched
    - **State 1**: “a” is matched & **State 3**: “aba” is matched



String Matching Automata (2)

❑ Automata knows how to handle “not-matched”

- Do not need to do match from scratch (can jump!)
- For example, suppose we are currently at State 5 for the pattern “**ababaca**”, and the next $A[i]$ is “b”



begin

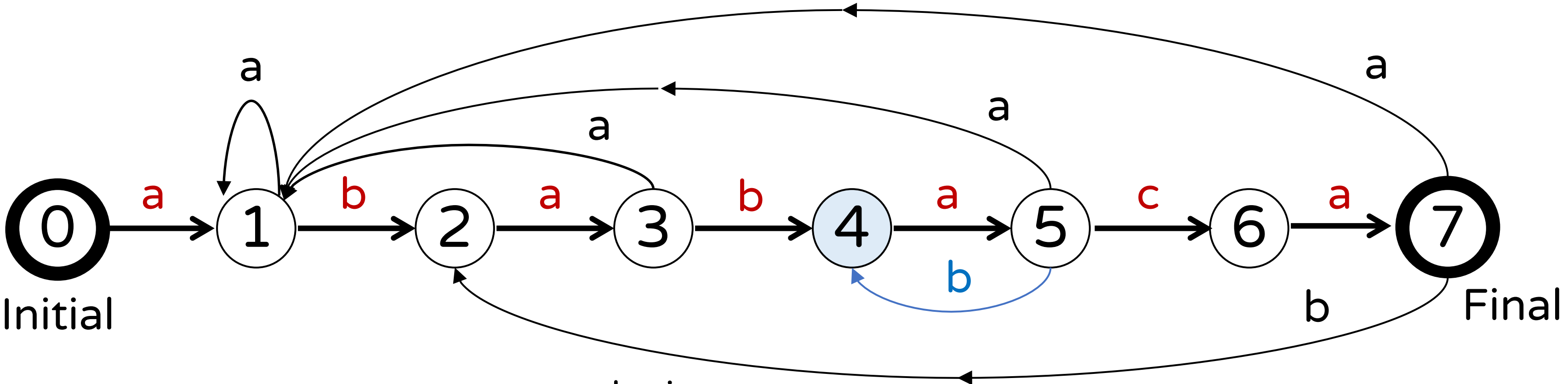
not-matched

A :				a	b	a	b	a	b						
P :				a	b	a	b	a	c	a					

String Matching Automata (3)

❑ Automata knows how to handle “not-matched”

- Do not need to do match from scratch
- By going back State 4 with “b”, we can resume matching after “abab”!



begin

Resume from here

A :															
				a	b	a	b	a	b						

P :							
	a	b	a	b	a	c	a

Automata Algorithm

❑ Phases of Automata Algorithm

- **Automata construction phase**
  - Construct the automata from the pattern string $\boldsymbol{P}$
- **Search phase**
  - Match the pattern $\boldsymbol{P}$ over the document string $\boldsymbol{A}$ with the automata

- For convenience, let’s first check the searching phase assuming a valid automata is given

- After then, let’s check how to construct the automata

Outline

❑ Intuition for automata algorithm

❑ String matching automata

❑ Search phase with automata

❑ Automata construction phase

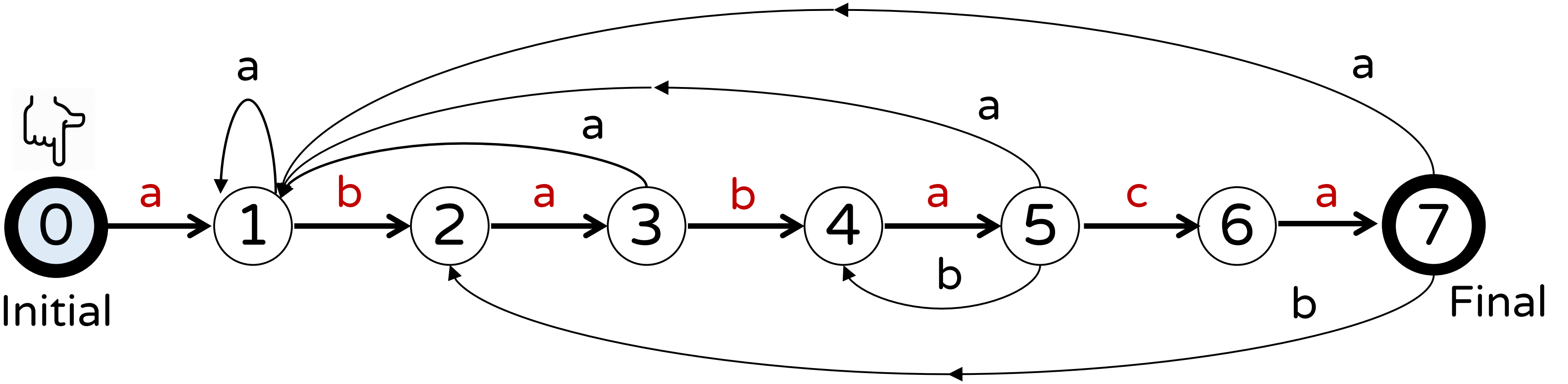
Search Phase with Automata (1)

❑ Initially, start at $A[1]$ and State 0

- No edge with label “t” at State 0 ⇒ Move to State 0

Input pattern:
“ababaca”

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t
P :	a	b	a	b	a	c	a								



Search Phase with Automata (2)

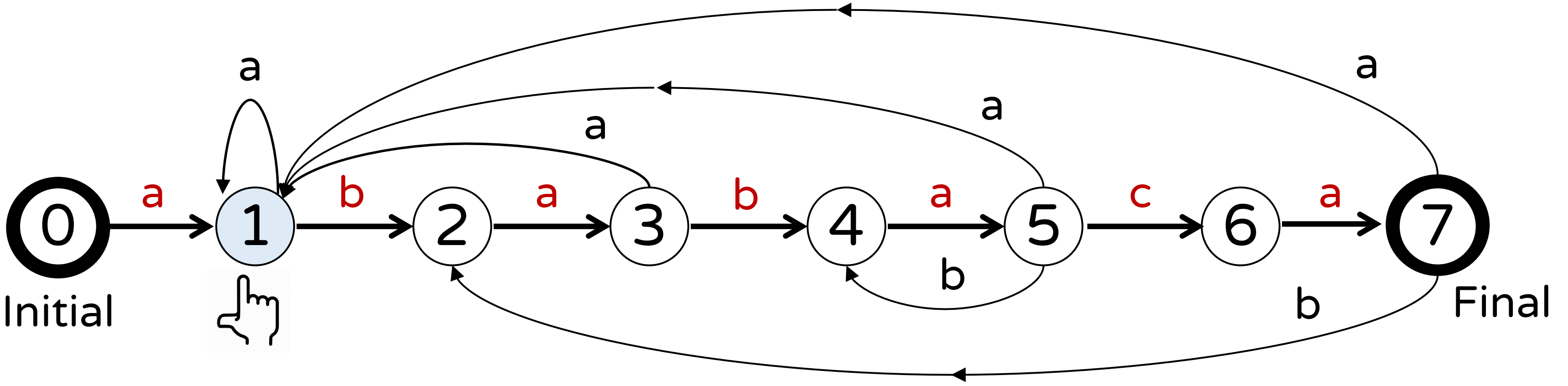
❑ Given label “a” at State 0, move to State 1

- Meaning “a” is matched

Input pattern:
“ababaca”

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

P :	a	b	a	b	a	c	a

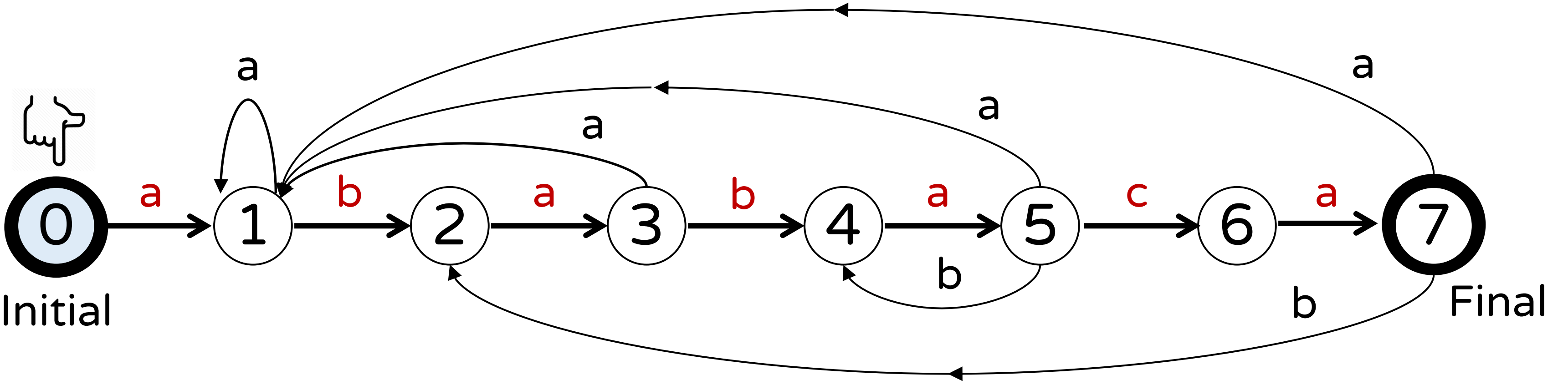


Search Phase with Automata (3)

❑ No edge with label “c” at State 1, go back to State 0

Input pattern:
“ababaca”

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t
P :		a	b	a	b	a	c	a							



Search Phase with Automata (4)

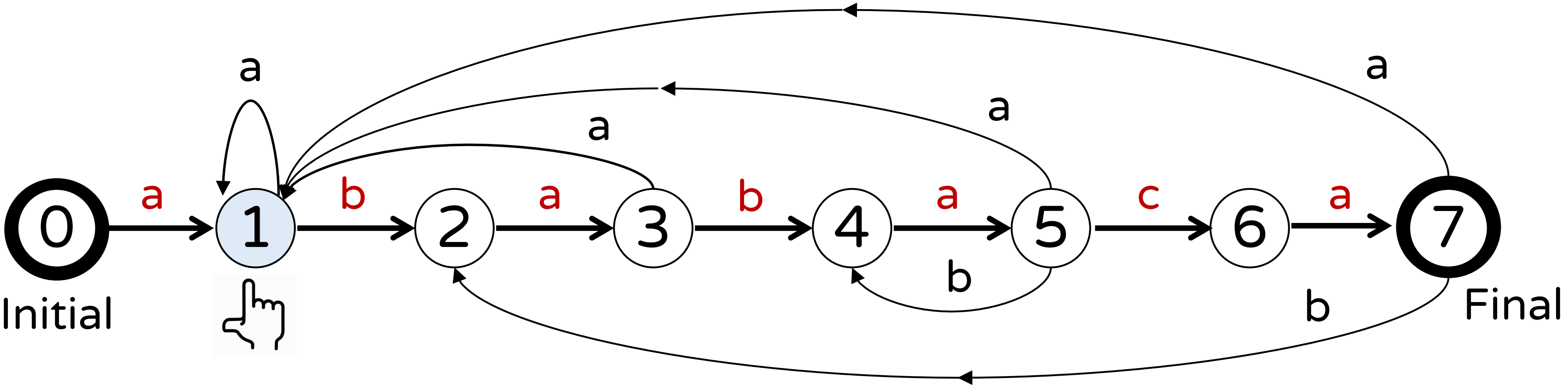
❑ Given label “a” at State 0, move to State 1

- Meaning “a” is matched

Input pattern:
“ababaca”

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

P :	a	b	a	b	a	c	a



Search Phase with Automata (5)

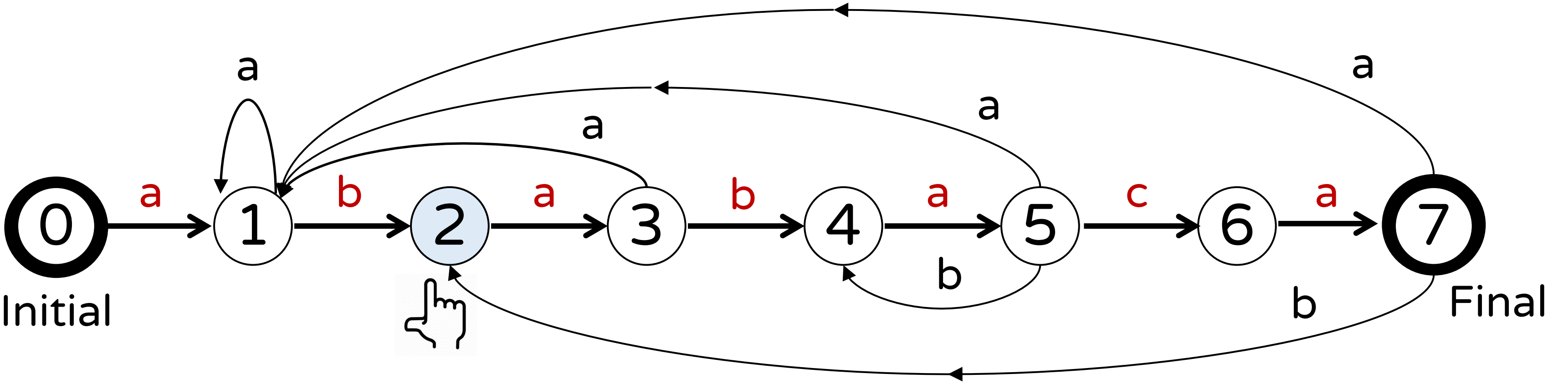
❑ Given label “b” at State 1, move to State 2

- Meaning “ab” is matched

Input pattern:
“ababaca”

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

P :	a	b	a	b	a	c	a

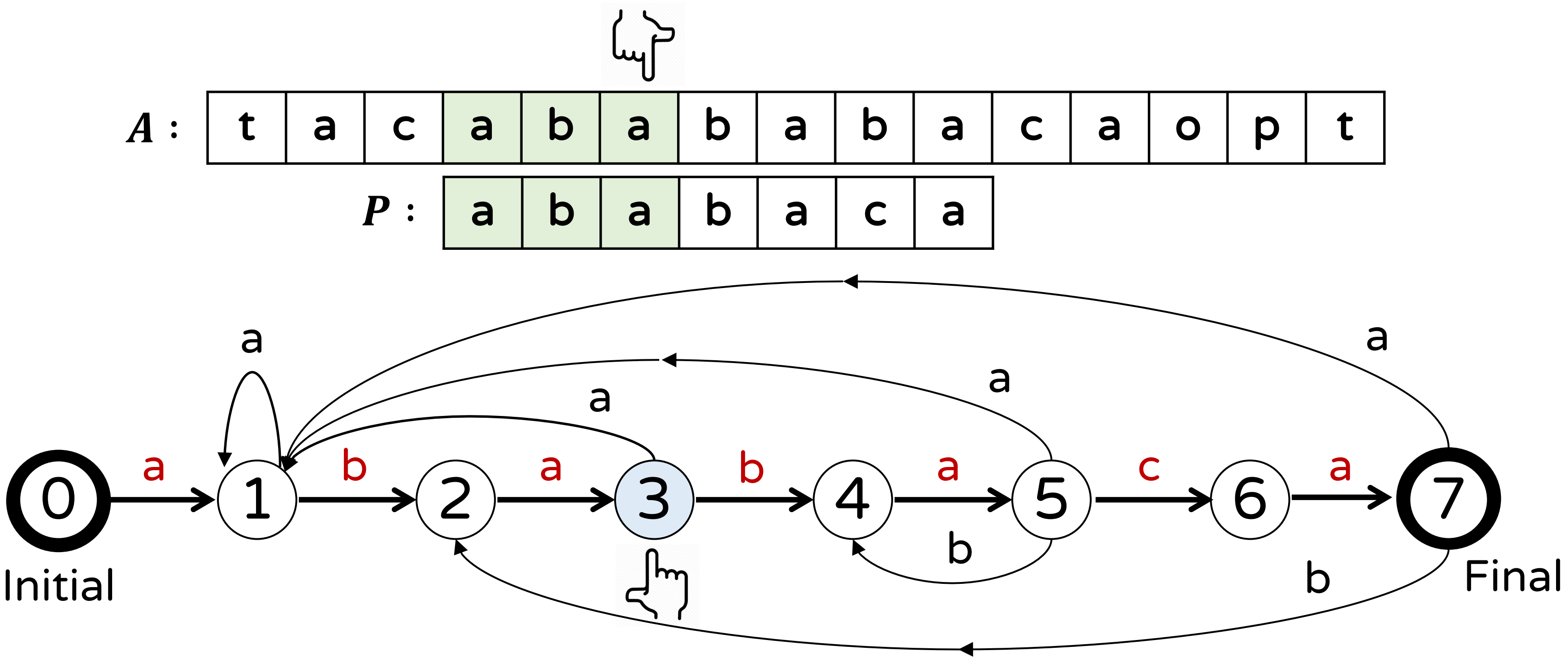


Search Phase with Automata (6)

❑ Given label “a” at State 2, move to State 3

- Meaning “aba” is matched

Input pattern:
“ababaca”



A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

P :	a	b	a	b	a	c	a

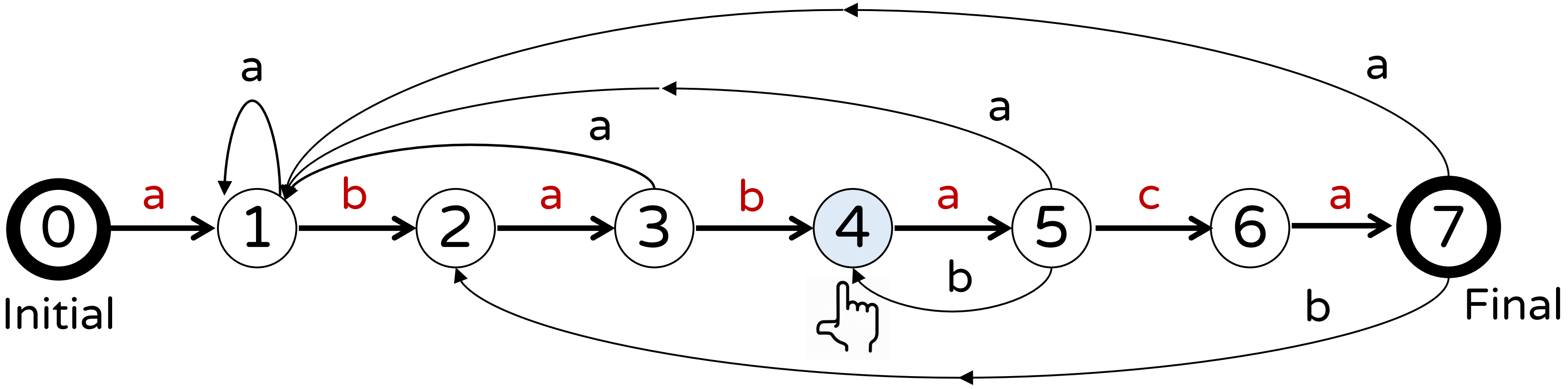
Search Phase with Automata (7)

❑ Given label “b” at State 3, move to State 4

- Meaning “abab” is matched

Input pattern:
“**abab**aca”

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t
			P :	a	b	a	b	a	c	a					



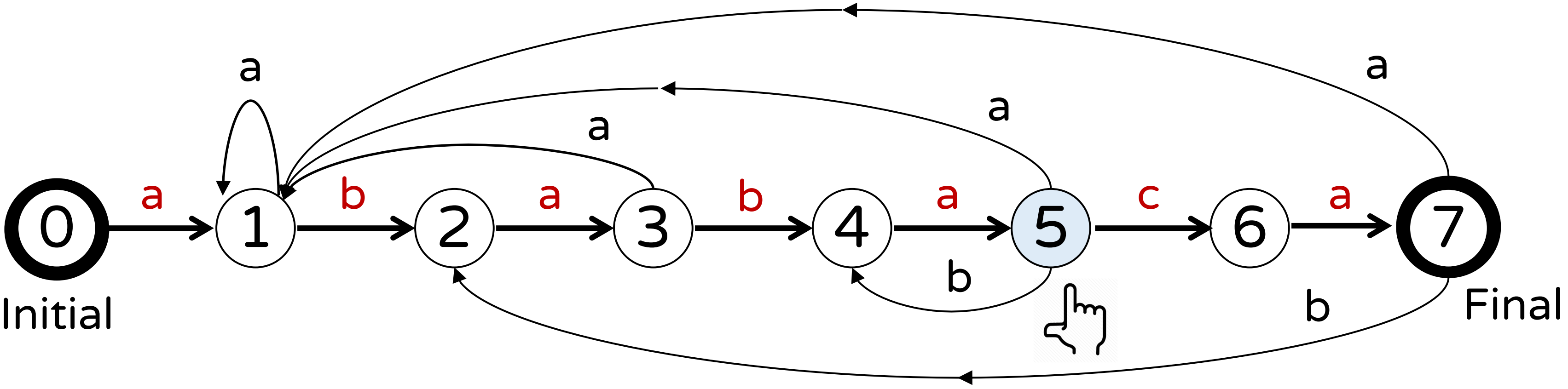
Search Phase with Automata (8)

❑ Given label “a” at State 4, move to State 5

- Meaning “ababa” is matched

Input pattern:
“ababaca”

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t
P :				a	b	a	b	a	c	a					

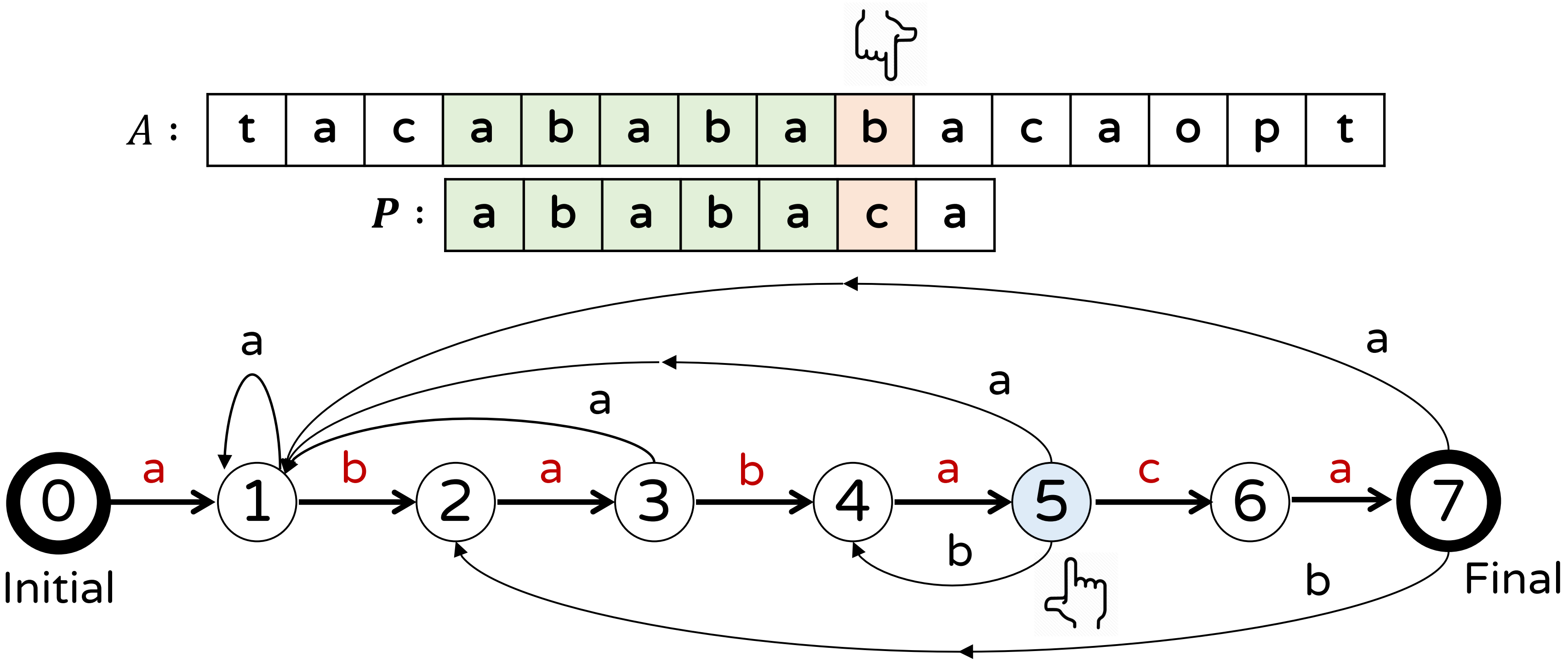


Search Phase with Automata (9)

❑ Now “b” is given at State 5

- Meaning not-matched event occurs here!

Input pattern:
“ababaca”



A : t a c a b a b a b a c a o p t

P : a b a b a c a

Search Phase with Automata (10)

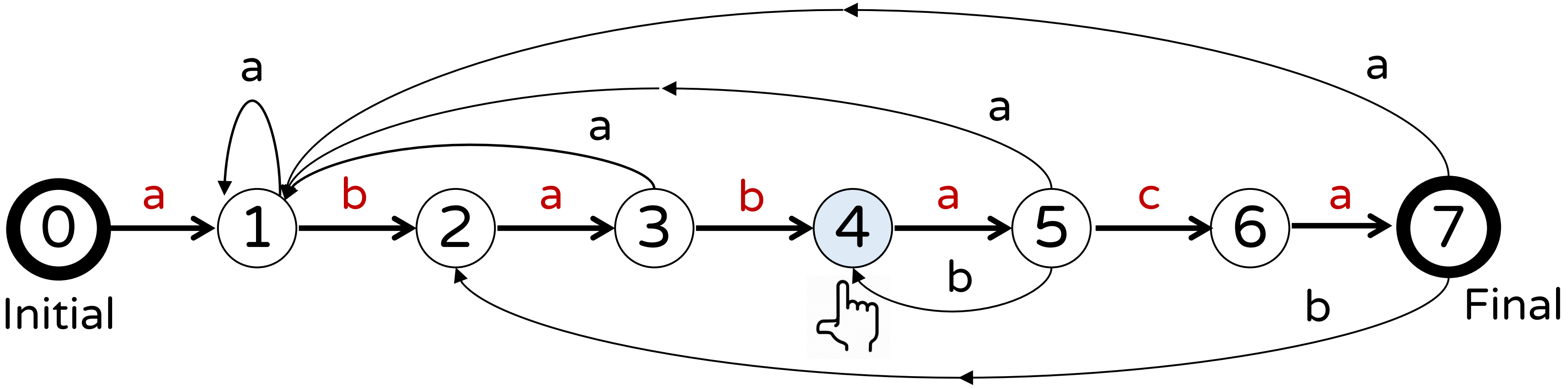
❑ Then, move to State 4 given label “b”

- Going back State 4 means we can resume from “abab”

Input pattern:
“ababaca”

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

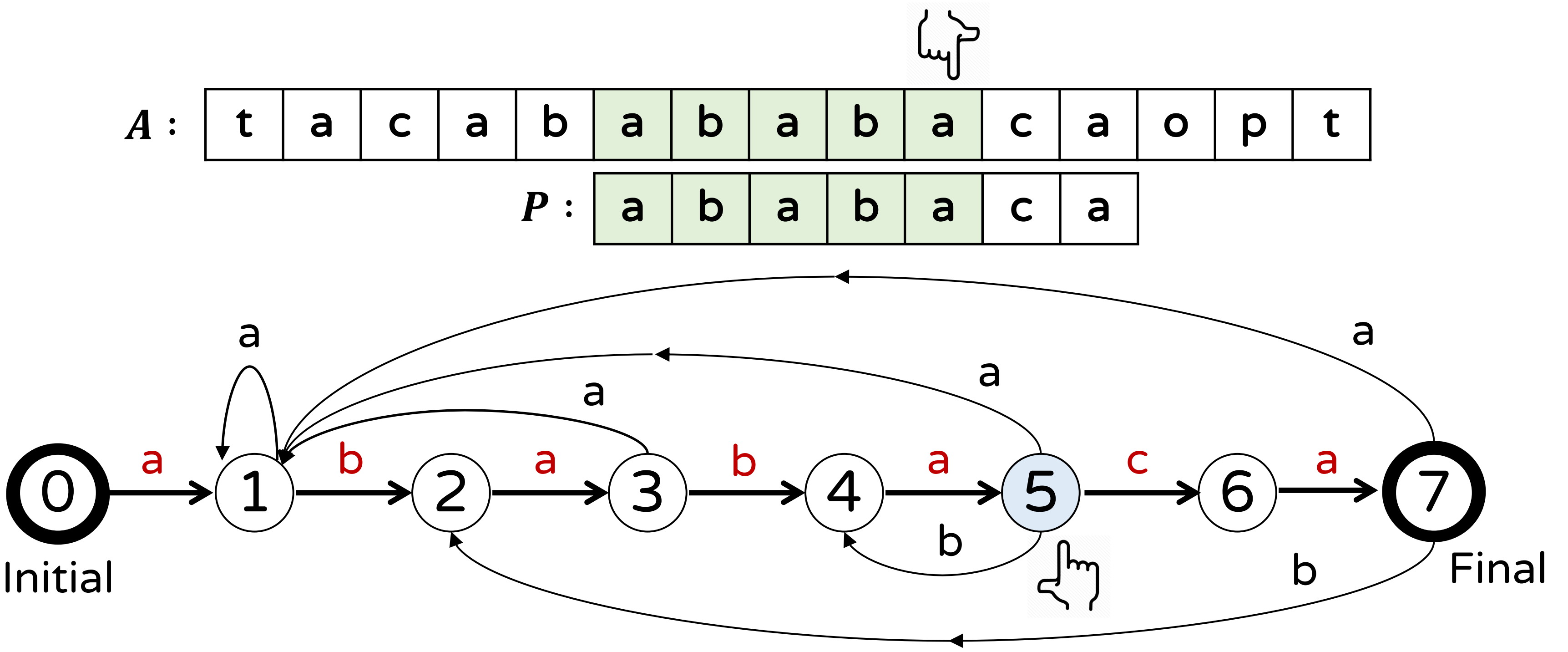
P :	a	b	a	b	a	c	a



Search Phase with Automata (11)

- Given label “a” at State 4, move to State 5

Input pattern:
“ababaca”



A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

P :	a	b	a	b	a	c	a

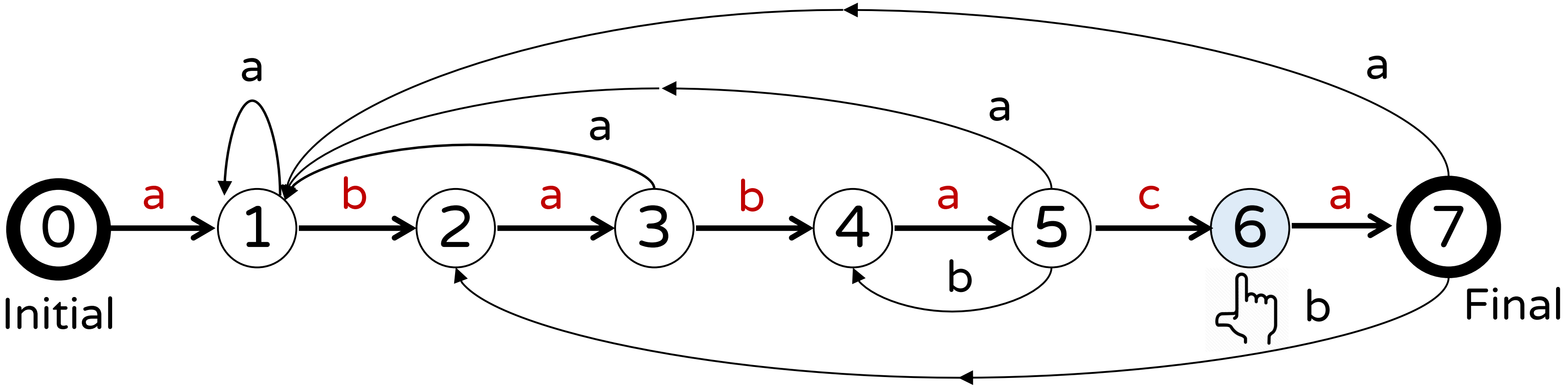
Search Phase with Automata (12)

- Given label "c" at State 5, move to State 6

Input pattern:
"ababaca"

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

P :	a	b	a	b	a	c	a

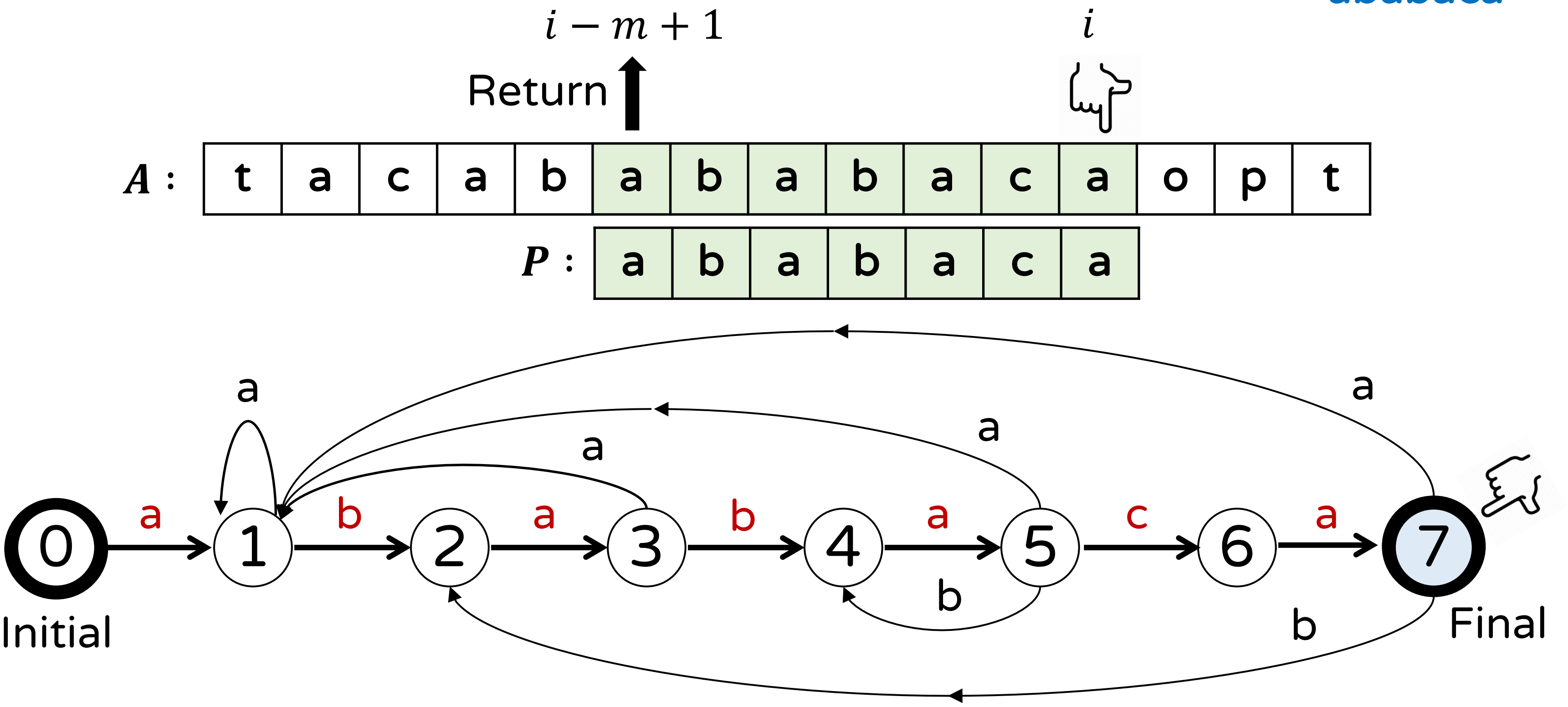


Search Phase with Automata (13)

❑ Given label “a” at State 6, move to State 7

- At the final state, the pattern is matched!

Input pattern:
“ababaca”



A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

P :	a	b	a	b	a	c	a

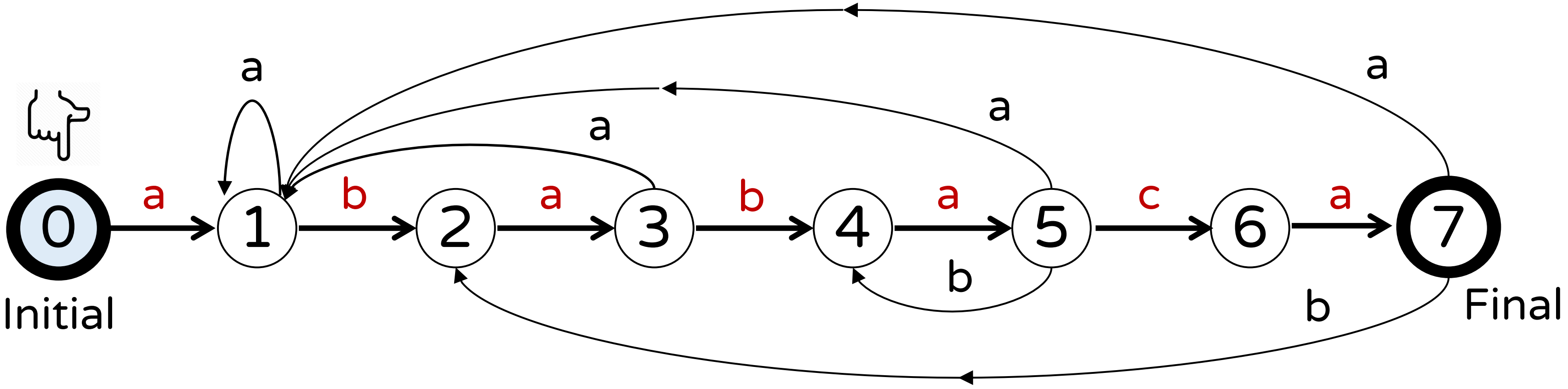
Search Phase with Automata (14)

❑ Repeat until checking all characters in A

Input pattern:
“abababaca”

A :															
	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

P :					
	a	b	a	b	…

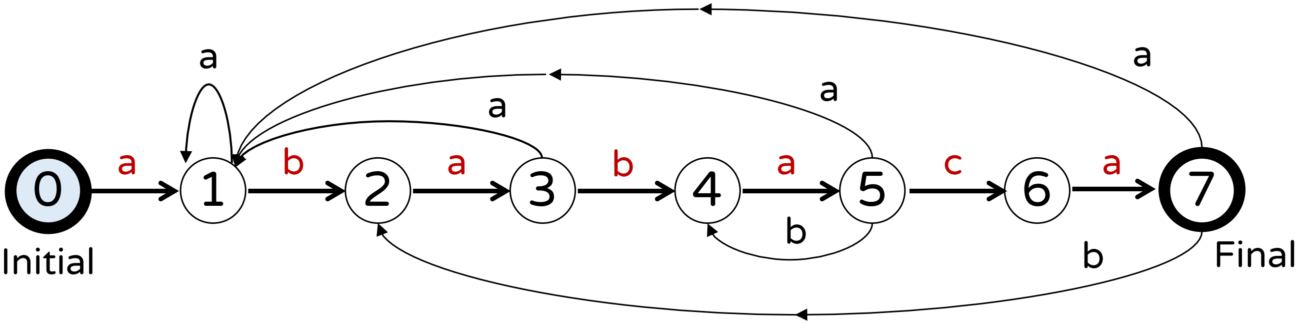


Search Phase with Automata

❑ How to represent the automata?

- The automata is represented by 2D-array called $\boldsymbol{T}$
  - Rows indicate states, and columns indicates characters
    - $\boldsymbol{\Sigma} = \{a, b, c\}$ is the set of unit characters

$\boldsymbol{P}$ = [ababaca]



$\boldsymbol{T}$	a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

Search Phase with Automata

❑ Pseudocode of search phase

```
def automata-search($\boldsymbol{A}$, $\boldsymbol{T}$):
    $s \leftarrow 0$     # state
    for $i \leftarrow 1$ to $n$:
        $c \leftarrow \boldsymbol{A}[i]$
        if $c \notin \boldsymbol{\Sigma}$: $s \leftarrow 0$
        else:     $s \leftarrow \boldsymbol{T}[s][c]$     # get the next state
        if $s$ is at the final state (=$m$):
            output “there is a matching at $\boldsymbol{A}[i - m + 1]$”
```

- $\boldsymbol{A}$ is a document string (length n)
- $\boldsymbol{T}$ is a table for the automata of pattern $\boldsymbol{P}$ (length m)

❑ Time and space complexities

- Time complexity: O(n) due to repeating the loop n times
- Space complexity: O($|\boldsymbol{\Sigma}|m + n$)
  - Input space: O($m + n$) for $\boldsymbol{A}$ and $\boldsymbol{P}$
  - Extra space: O($|\boldsymbol{\Sigma}|m$) for $\boldsymbol{T}$

Outline

- ❑ Intuition for automata algorithm
- ❑ String matching automata
- ❑ Search phase with automata
- ❑ Automata construction phase

Overview

❑ How to construct the automata from a pattern?

- Given the automata, searching is very fast ($O(n)$ time)

- We cover an easier version for constructing the automata
  - It takes $O(|\mathbf{\Sigma}|m^3)$ time, but easy-to-understand the key intuition
  - There is a more efficient version taking $O(|\mathbf{\Sigma}|m)$ time, but it is out-of-scope (due to time limit – see [link] if interested)

- To understand the algorithm, we first need to check the definition of prefix and suffix of a string

Prefix and Suffix

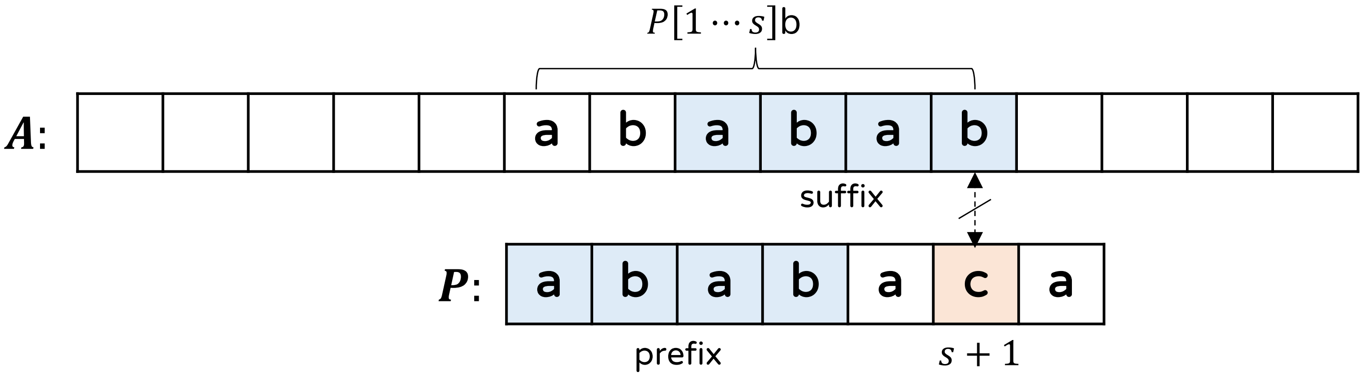
- **Prefix**: p is a prefix of a string t if there exists a string s such that $t = ps$
  - **Proper prefix** is one of prefixes excluding the original string t
  - e.g., “A”, “AB” are proper prefixes of “ABC”

- **Suffix**: s is a suffix of a string t if there exists a string p such that $t = ps$
  - Proper suffix is one of suffixes excluding the original string t
  - e.g., “C”, “BC” are proper suffixes of “ABC”

Intuition of Automata Construction

❑ Consider the following case:

- It fails matching at $s+1$, meaning $P[1 \cdots s]$ is matched



- Which part can be re-useable in the above result?
  - We can use the longest proper prefix of "$\boldsymbol{P}[1 \cdots s]$b" that is also a suffix of the sub-string of $\boldsymbol{A}$ (or "$\boldsymbol{P}[1 \cdots s]$b").
  - Let's call it "longest prefix-suffix (LPS)" of "$\boldsymbol{P}[1 \cdots s]$b"
    - Given label "b", the next state is the state of "abab" obtained from the above LPS.
  - Thus, the automata is constructed from the LPS information.

Automata Construction Phase (1)

❑ Main idea of the construction phase

- Try all possible prefixes starting from the longest possible that can be also a suffix of “$\boldsymbol{P}[1 \cdots s]x$” for each $x \in \boldsymbol{\Sigma}$

```
def construct-automata($\boldsymbol{P}$, $\boldsymbol{\Sigma}$):
    initialize $\boldsymbol{T}$
    for $s \leftarrow 0$ to $m$:
        for $x \in \boldsymbol{\Sigma}$:
            $\boldsymbol{T}[s][x]$ ← get_next_state($\boldsymbol{P}$, $s$, $x$)
    return $\boldsymbol{T}$
```

```
def get_next_state($\boldsymbol{P}$, $s$, $x$):
    if $s < m$ and $\boldsymbol{P}[s+1]$ is $x$:
        return $s + 1$
    else:
    # check if proper prefix of “$\boldsymbol{X} = \boldsymbol{P}[1 \cdots s]x$”
      is a suffix from largest to smallest
        $\boldsymbol{X} \leftarrow \boldsymbol{P}[1 \cdots s]$ + $x$
        for len ← $s$ downto 1:
            $p' \leftarrow$ get_prefix($\boldsymbol{X}$, len)
            $s' \leftarrow$ get_suffix($\boldsymbol{X}$, len)
            if $p'$ is $s'$:
                return len
    return 0    # when nothing is found
```

- Time complexity
  - $O(|\boldsymbol{\Sigma}|m^3)$

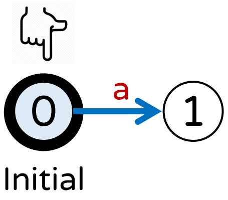
Automata Construction Phase (1)

❑ Example of $\boldsymbol{P}$ = "ababaca"

- When $s = 0$, find the LPS of "$P[1 \cdots 0]\boldsymbol{x}$" = "$\boldsymbol{x}$" for $\boldsymbol{x} \in \{a, b, c\}$

```
def get_next_state($P$, $s$, $x$):
    if $s < m$ and $P[s+1]$ is $x$:                ⇐"a"
        return $s+1$
    else:
        $X \leftarrow P[1 \cdots s]$ + $x$
        for len $\leftarrow$ $s$ downto 1:
            $p' \leftarrow$ get_prefix($X$, len)
            $s' \leftarrow$ get_suffix($X$, len)
            if $p'$ is $s'$:
                return len
    return 0    # when nothing is found            ⇐"b" and "c"
```

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



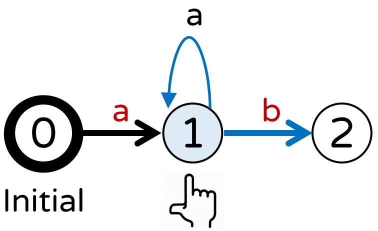
Automata Construction Phase (2)

❑ Example of P = "ababaca"

- When $s = 1$, find the LPS of "$P[1 \cdots 1]x$" for $x \in \{a, b, c\}$

```
def get_next_state(P, s, x):
    if $s < m$ and $P[s+1]$ is $x$:          ⇐"ab"
        return $s+1$
    else:
         $X \leftarrow P[1 \cdots s]$ + $x$
         for len ← $s$ downto 1:             ⇐"aa"
             $p' \leftarrow$ get_prefix(X, len)
             $s' \leftarrow$ get_suffix(X, len)
             if $p'$ is $s'$:
                 return len
    return 0    # when nothing is found      ⇐"ac"
```

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



Automata Construction Phase (3)

❑ Example of $\boldsymbol{P}$ = “ababaca”

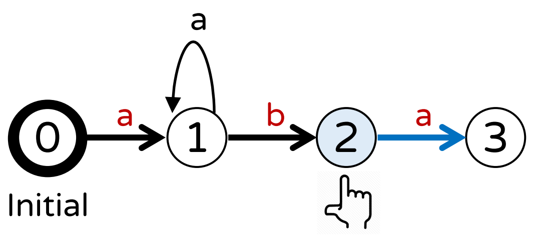
- When $s = 2$, find the LPS of “$P[1 \cdots 2]\boldsymbol{x}$” for $\boldsymbol{x} \in \{a, b, c\}$

```
def get_next_state($\boldsymbol{P}$, $s$, $x$):
    if $s < m$ and $\boldsymbol{P}[s+1]$ is $x$:
        return $s+1$
    else:
        $\boldsymbol{X} \leftarrow \boldsymbol{P}[1 \cdots s]$ + $x$
        for len ← $s$ downto 1:
            $p' \leftarrow$ get_prefix($\boldsymbol{X}$, len)
            $s' \leftarrow$ get_suffix($\boldsymbol{X}$, len)
            if $p'$ is $s'$:
                return len
    return 0    # when nothing is found
```

⇐“aba”

⇐“abb” and “abc”

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



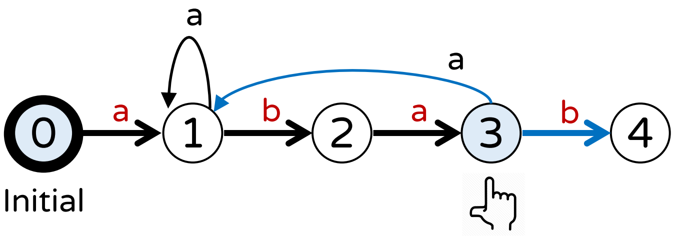
Automata Construction Phase (4)

❑ Example of P = "ababaca"

- When $s = 3$, find the LPS of "$P[1 \cdots 3]x$" for $x \in \{a, b, c\}$

```
def get_next_state($P$, $s$, $x$):
    if $s < m$ and $P[s+1]$ is $x$:                ⇐"abab"
        return $s + 1$
    else:
        $X \leftarrow P[1 \cdots s]$ + $x$
        for len ← $s$ downto 1:
            $p' \leftarrow$ get_prefix($X$, len)       ⇐"abaa"
            $s' \leftarrow$ get_suffix($X$, len)
            if $p'$ is $s'$:
                return len
    return 0    # when nothing is found            ⇐"abac"
```

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



Automata Construction Phase (5)

❑ Example of $\boldsymbol{P}$ = "ababaca"

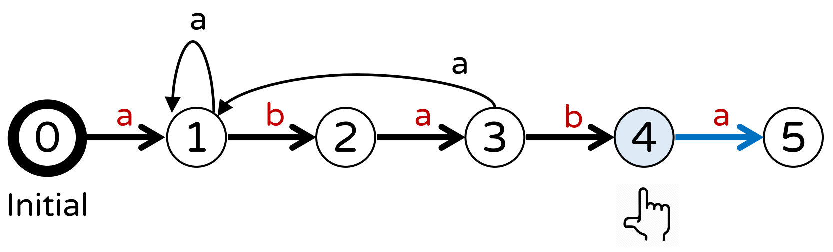
- When $s = 4$, find the LPS of "$P[1 \cdots 4]\boldsymbol{x}$" for $\boldsymbol{x} \in \{a, b, c\}$

```
def get_next_state($\boldsymbol{P}$, $s$, $x$):
    if $s < m$ and $\boldsymbol{P}[s+1]$ is $x$:
        return $s+1$
    else:
        $\boldsymbol{X} \leftarrow \boldsymbol{P}[1 \cdots s]$ + $x$
        for len ← $s$ downto 1:
            $p' \leftarrow$ get_prefix($\boldsymbol{X}$, len)
            $s' \leftarrow$ get_suffix($\boldsymbol{X}$, len)
            if $p'$ is $s'$:
                return len
    return 0    # when nothing is found
```

⇐"ababa"

⇐"ababb" and "ababc"

$\boldsymbol{T}$	a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5			
6			
7			



Automata Construction Phase (6)

❑ Example of P = "ababaca"

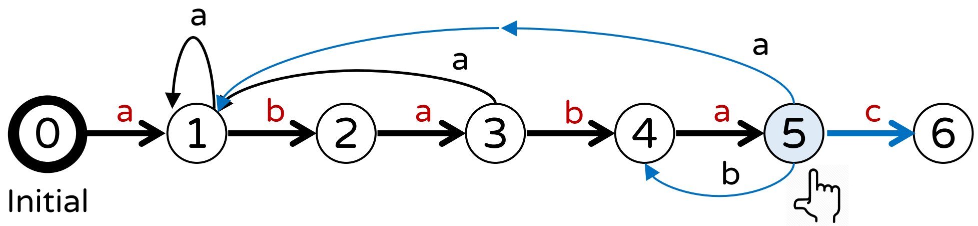
- When $s = 5$, find the LPS of "$P[1\cdots 5]x$" for $x \in \{a, b, c\}$

```
def get_next_state($P$, $s$, $x$):
    if $s < m$ and $P[s+1]$ is $x$:
        return $s+1$
    else:
        $X \leftarrow P[1\cdots s]$ + $x$
        for len ← $s$ downto 1:
            $p' \leftarrow$ get_prefix($X$, len)
            $s' \leftarrow$ get_suffix($X$, len)
            if $p'$ is $s'$:
                return len
    return 0    # when nothing is found
```

⇐"ababac"

⇐"ababaa" & "ababab"

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



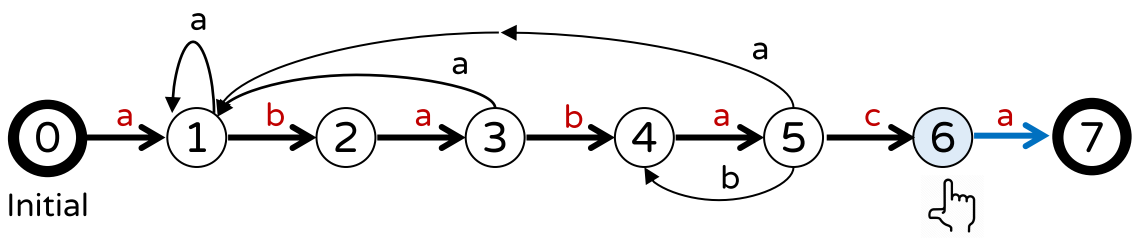
Automata Construction Phase (7)

❑ Example of $\boldsymbol{P}$ = “ababaca”

- When $s = 6$, find the LPS of “$P[1 \cdots 6]\boldsymbol{x}$” for $\boldsymbol{x} \in \{a, b, c\}$

```
def get_next_state($\boldsymbol{P}$, $s$, $x$):
    if $s < m$ and $\boldsymbol{P}[s+1]$ is $x$:        ⇐“ababaca”
        return $s + 1$
    else:
        $\boldsymbol{X} \leftarrow \boldsymbol{P}[1 \cdots s]$ + $x$
        for len ← $s$ downto 1:
            $p' \leftarrow$ get_prefix($\boldsymbol{X}$, len)
            $s' \leftarrow$ get_suffix($\boldsymbol{X}$, len)
            if $p'$ is $s'$:
                return len
    return 0    # when nothing is found        ⇐“ababacb” & “ababacc”
```

$\boldsymbol{T}$	a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7			



Automata Construction Phase (8)

❑ Example of $\boldsymbol{P}$ = "ababaca"

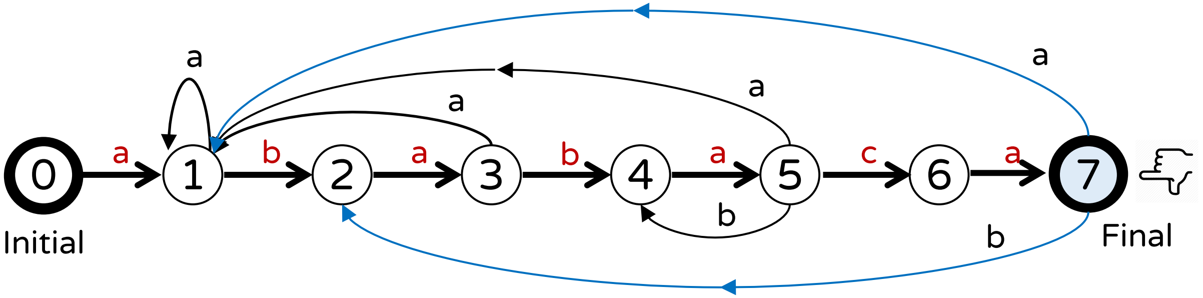
- When $s = 7$, find the LPS of "$P[1 \cdots 7]\boldsymbol{x}$" for $\boldsymbol{x} \in \{a, b, c\}$

```
def get_next_state($\boldsymbol{P}$, $s$, $x$):
    if $s < m$ and $\boldsymbol{P}[s+1]$ is $x$:
        return $s+1$
    else:
        $\boldsymbol{X} \leftarrow \boldsymbol{P}[1 \cdots s]$ + $x$
        for len ← $s$ downto 1:
            $p' \leftarrow$ get_prefix($\boldsymbol{X}$, len)
            $s' \leftarrow$ get_suffix($\boldsymbol{X}$, len)
            if $p'$ is $s'$:
                return len
    return 0    # when nothing is found
```

⇐"ababacaa" & "ababacab"

⇐"ababacac"

$\boldsymbol{T}$	a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0



What You Need To Know

❑ String automata algorithm

- Do not need to do match from scratch when not matched
- Automata knows where we jump when not matched, which is constructed by longest prefix-suffix information
  - Searching takes $O(n)$ time, and constructing takes $O(|\mathbf{\Sigma}|m^3)$ time

Algorithm	Time			Space	
	Preprocessing	Searching	Total	Input	Extra
Naïve	$O(1)$	$O(mn)$	$O(mn)$	$O(m+n)$	$O(1)$
Rabin-Karp	$O(m)$	$O(n+Fm)$	$O(n+Fm)$		$O(1)$
Automata	$O(\|\mathbf{\Sigma}\|m^3)$	$O(n)$	$O(\|\mathbf{\Sigma}\|m^3+n)$		$O(\|\mathbf{\Sigma}\|m)$

∗ Rabin-karp's search phase shows $O(n)$ average-case time and $O(mn)$ worst-case time
∗ Automata can be constructed in $O(|\mathbf{\Sigma}|m)$ time using the optimized version

In Next Lecture

- Can we do string matching faster than automata?
  - Yes! KMP algorithm does!

Thank You