

Lecture 6

String Matching

Bo Tang @ SUSTech, Fall 2024

Our Roadmap

- ◆ String Concepts
- ◆ String Searching Problem
 - ◆ Brute Force Solution
 - ◆ Rabin-Karp
 - ◆ Finite State Automata
 - ◆ Knuth-Morris-Pratt

String Definition

◆ String:

- ◆ Sequence of characters over some alphabet
- ◆ Binary $\{0,1\}$: $S1 = "10000101010101001010101"$
- ◆ DNA $\{ACGT\}$: $S2 = "ACGTACGTACGTTCGA"$
- ◆ English Characters $\{a...z, A..Z\}$: $S3 = "Hello World"$

◆ Applications

- ◆ Word processors
- ◆ Virus scanning
- ◆ Text retrieval
- ◆ Natural language processing
- ◆ Web search engine

String Operators

- ◆ append: append to string
- ◆ assign: assign content to string
- ◆ insert: insert to string
- ◆ erase: erase characters from string
- ◆ replace: replace portion of string
- ◆ swap: swap string values
- ◆ find: find the specific char in the string
- ◆ Give string $s = \text{"SUSTechCS203"}$, how many sub string it has?

Handwritten diagram showing string manipulation. Top row: D S B B. A bracket connects the 'S' and the first 'B'. Bottom row: D B S B. An arrow points from the 'S' in the bottom row to the 'S' in the top row.

Handwritten notes for the last question:

Array \rightarrow Sub Array

String $s = \text{"SUSTechCS203"}$

$i \quad j \quad \dots \quad n$

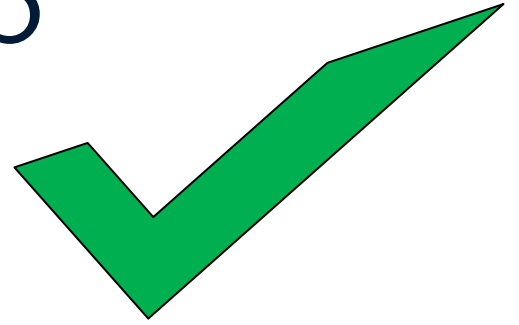
$\hookrightarrow \text{len}(s) = n.$

Handwritten formula for counting substrings:

$$\sum_{i=1}^n \sum_{j=i}^n 1, \quad 1 \leq i \leq j \leq n.$$

Below the formula, there are some scribbles and numbers like 12, 10, 11, 12, 1.

Our Roadmap



- ◆ String Concepts
- ◆ String Searching Problem
 - ◆ Brute Force Solution
 - ◆ Rabin-Karp
 - ◆ Finite State Automata
 - ◆ Knuth-Morris-Pratt

Why String Searching?

◆ **Applications in Computational Biology**

- ◆ DNA sequence is a long word (or text) over a 4-letter alphabet
- ◆ GTTTGAGTGGTCAGTCTTTTCGTTTCGACGGAGCCC.....
- ◆ Find a Specific pattern W

◆ **Finding patterns in documents formed using a large alphabet**

- ◆ Word processing
- ◆ Web searching
- ◆ Desktop search (Google, MSN)

◆ **Matching strings of bytes containing**

- ◆ Graphical data
- ◆ Machine code

◆ **grep in unix**

- ◆ grep searches for lines matching a pattern.

String Searching

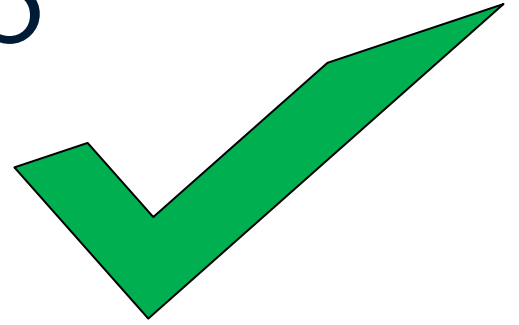
Search Text										
a	s	s	u	s	u	s	t	c	s	c

Search Pattern				
s	u	s	t	c

Successful Search										
a	s	s	u	s	u	s	t	c	s	c

- ◆ Parameter
 - ◆ n : # of characters in text
 - ◆ m : # of characters in pattern
 - ◆ Typically, $n \gg m$
 - ◆ e.g., $n = 1 \text{ Billion}$, $m = 100$

Our Roadmap



- ◆ String Concepts
- ◆ String Searching Problem
 - ◆ Brute Force Solution
 - ◆ Rabin-Karp
 - ◆ Finite State Automata
 - ◆ Knuth-Morris-Pratt

Brute Force

- ◆ Brute force

- ◆ Check for pattern starting at every text position

- ◆ **Algorithm:** BruteForce(T, P):

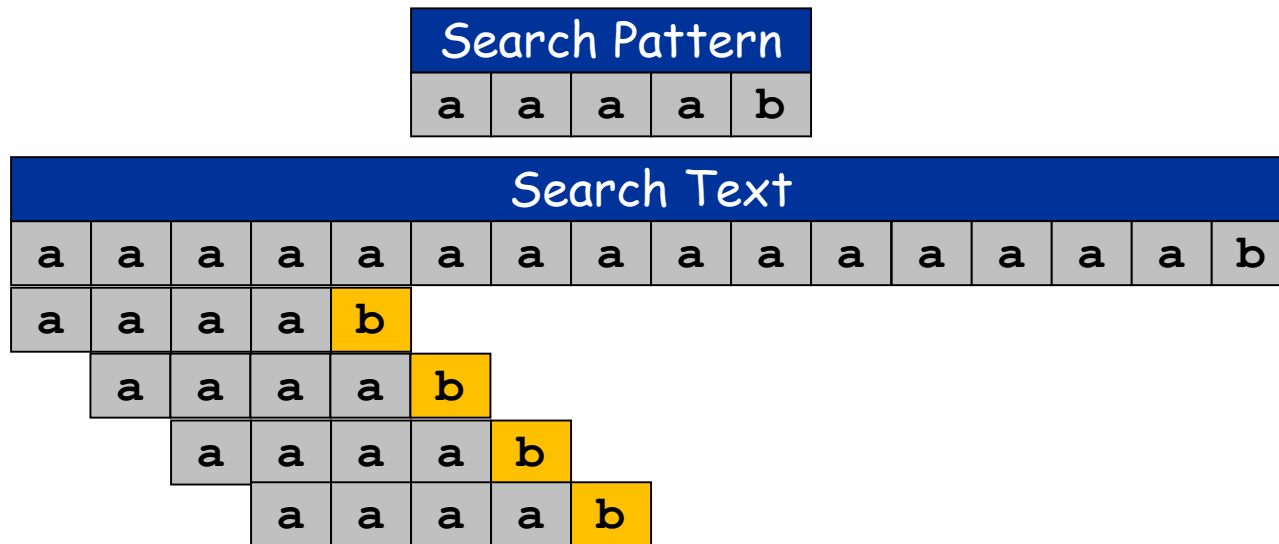
$$O(mn) \rightarrow O(n+m)$$

1. $n \leftarrow \text{len}(T)$, $m \leftarrow \text{len}(P)$
2. **for** $i \leftarrow 0$ to $n-m-1$
3. **for** $j \leftarrow 0$ to $m-1$
4. **if** $P[j] \neq T[i+j]$ **then**
5. **break**;
6. **if** $j = m-1$
7. pattern occurs with shift i

- ◆ Time complexity?

Analysis of Brute Force

- ◆ Analysis of brute force
 - ◆ Running time depends on pattern and text
 - ◆ Can be slow when strings repeat themselves
 - ◆ Worst case: mn comparisons
 - ◆ Too slow when m and n are large



■ ■ ■ ■ ■ ■

Can we do better?

- ◆ How to avoid re-computation?
 - ◆ Pre-analyze search pattern
 - ◆ Example: suppose the first 4 chars of pattern are all a's
 - ◆ If $t[0..3]$ matches $p[0..3]$ then $t[1..3]$ matches $p[0..2]$
 - ◆ No need to check $i=1, j=0,1,2$
 - ◆ Saves 3 comparisons
 - ◆ Need better ideas in general

Search Pattern				
a	a	a	a	b

Search Text																
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	b
a	a	a	a	b												
	a	a	a	a	b											

Our Roadmap

- ◆ String Concepts
- ◆ String Searching Problem
 - ◆ Brute Force Solution
 - ◆ Rabin-Karp
 - ◆ Finite State Automata
 - ◆ Knuth-Morris-Pratt



Rabin-Karp Algorithm

- Given search text T and search pattern P as follows:

Pattern			
1	3	5	9

Search Text												
2	4	6	8	0	1	2	1	3	5	9	7	2
							1	3	5	9		

size M.

radix-d for example

$$a = p[m] + d(p[m+1] + d(p[m+2] + \dots + d[p[M]]))$$

- Any idea?

radix-d 加法

$$p = [3, 9] \rightarrow \text{防止溢出，最多有1位}$$

$$a = 9 + (0 \times 10 + (0 \times 10 + (0 \times 1)))$$

2468	4680	6801	8012	121	1213	2135	1359	3597	5972
							1359		

Rabin-Karp Algorithm

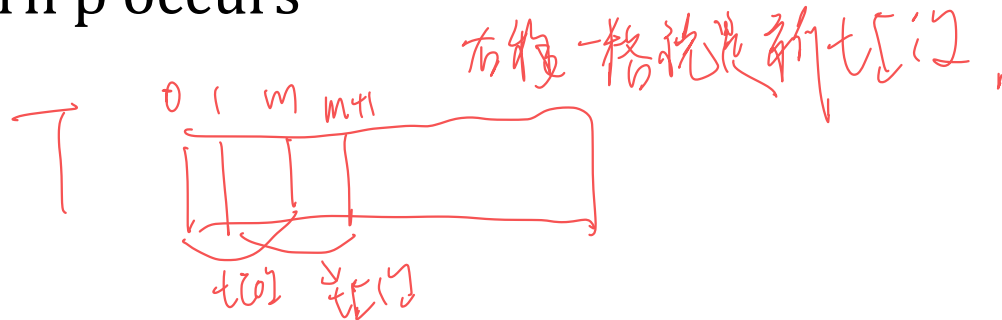
General idea

- Convert search pattern to a number p
- Convert search text to an array of numbers $t[0], \dots, t[n-m-1]$
- Compare p with $t[i]$, for each i in $[0, n-m-1]$
- if $p = t[i]$, pattern p occurs

Number theory 是一个很大的领域
 $a \equiv b \pmod{q} \rightarrow$ 每一遍取余一次 $b \equiv 0 \sim q-1$

Example

- $p = 1359$
- Array t is:



2468	4680	6801	8012	121	1213	2135	1359	3597	5972
------	------	------	------	-----	------	------	------	------	------

- $t[7] = p \rightarrow T[7,8,9,10] = P[0,1,2,3]$

Rabin-Karp Algorithm

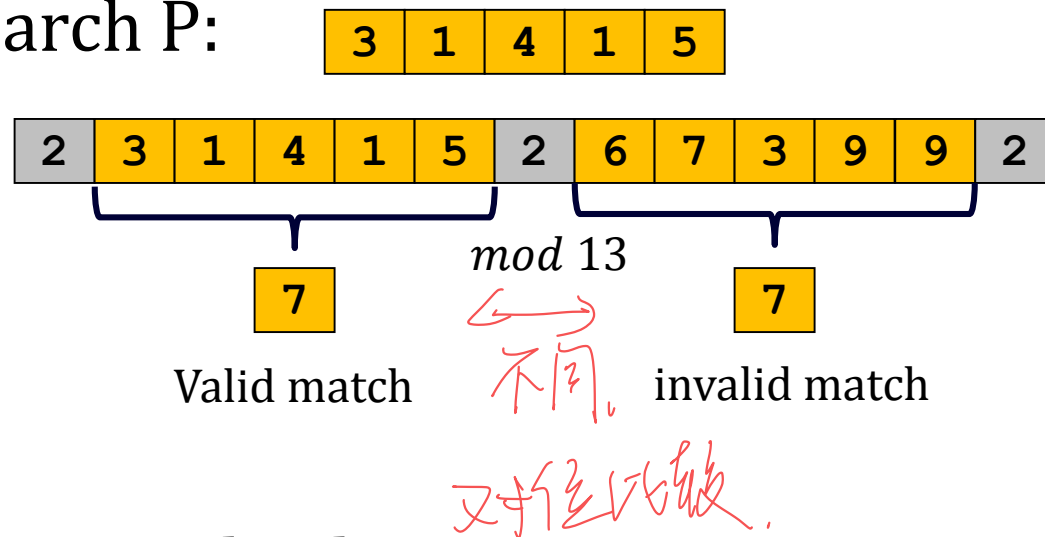
- ◆ How to convert size- m characters to a number?
 - ◆ E.g., the alphabet $\Sigma = \{a, \dots, z, A, \dots, Z\}$
 - ◆ Solution: radix- d ($d = |\Sigma|$) Horner's rule
 - ◆ $p = P[m-1] + d(P[m-2] + d(P[m-3] + \dots + d(P[1] + dP[0])))$
- ◆ When m is large, p may be too large to work
 - ◆ Modulo a proper prime number q
 - ◆ $p = P[m-1] + d(P[m-2] + d(P[m-3] + \dots + d(P[1] + dP[0]))) \bmod q$
- ◆ Compute $t[0], t[1], \dots, t[n-m-1]$ in time $O(n-m)$
 - ◆ Compute $t[i+1]$ by using $t[i]$ in $O(1)$ time
 - ◆ $t[i+1] = d(t[i] - d^{m-1}T[i]) + T[i+m]$ → 滚动计算
 - ◆ $t[i+1] = ((t[i] - hT[i]) + T[i+m]) \bmod q$, where $h \equiv d^{m-1} \pmod{q}$
 - ◆ $t[0] \rightarrow t[1] \rightarrow t[2] \rightarrow t[3] \rightarrow \dots \rightarrow t[n-m-1]$ in $O(n-m)$

Rabin-Karp Algorithm

- Correctness analysis

- $p \not\equiv t[i] \pmod{q}$ we have $p \neq t[i]$, thus, $P[0, \dots, m-1] \neq T[i, i+m-1]$
- $p \equiv t[i] \pmod{q}$, it does not imply $p = t[i]$ (**spurious hit**)

- Example: search P:



- Additional test to check

- $P[0, \dots, m-1] = T[i, i+m-1]$

Rabin-Karp Algorithm

◆ **Algorithm:** Rabin-Karp(T, P, d, q):

1. $n \leftarrow \text{len}(T)$, $m \leftarrow \text{len}(P)$
2. $h \leftarrow d^{m-1} \pmod{q}$, $p \leftarrow 0$, $t_0 \leftarrow 0$
3. **for** $j \leftarrow 0$ to $m-1$
4. $p \leftarrow (dp + P[j]) \pmod{q}$,
5. $t_0 \leftarrow (dt_0 + T[j]) \pmod{q}$,
6. **for** $i \leftarrow 0$ to $n-m$
7. **if** $p \neq t_i$ **then**
8. $t_{i+1} \leftarrow (d(t_i - T[i]h) + T[i+m]) \pmod{q}$
9. **else**
10. **If** $P[0..m-1] = T[i, i+m-1]$
11. pattern occurs with shift i
12. **Else**
13. $t_{i+1} \leftarrow (d(t_i - T[i]h) + T[i+m]) \pmod{q}$

Analysis of Rabin-Karp Alg.

◆ **Algorithm:** Rabin-Karp(T, P, d, q):

Cost of Line 1:

Cost of Line 2:

Cost of Line 3:

Cost of Line 4:

...

Cost of Line 11:

Cost of Line 12:

Cost of Line 13:

Overall Cost:

Our Roadmap

- ◆ String Concepts
- ◆ String Searching Problem
 - ◆ Brute Force Solution
 - ◆ Rabin-Karp
 - ◆ Finite State Automata → FSA
 - ◆ Knuth-Morris-Pratt → KMP

BF

↓
RK

↓
FSA

↓
KMP



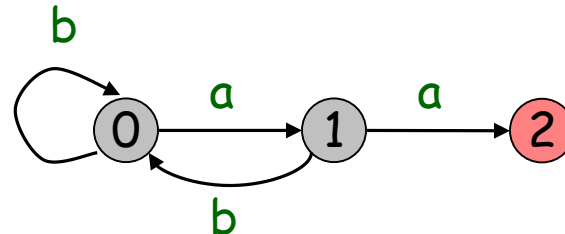
Finite State Automata

- ◆ A finite State automaton is defined by:
 - ◆ Q , a set of states
 - ◆ $q_0 \in Q$, the start state
 - ◆ $A \subseteq Q$, the accepting states
 - ◆ Σ , the input alphabet
 - ◆ δ , the transition function, from $Q \times \Sigma$ to Q

数字识别法

数字识别法 (mismatch 识别法)

	0	1
a	1	2
b	0	0

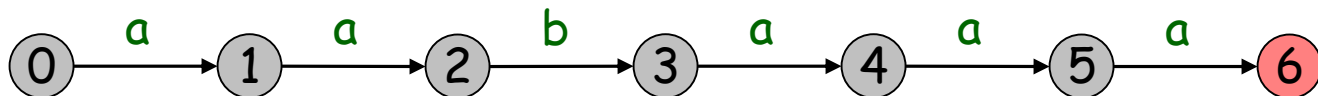


FSA idea for String Matching

- Start in state q_0
- Perform a transition from q_0 to q_1 if next character of $T = P[1]$
- State q_i means first i characters of P match.
- Transition from q_i to q_{i+1} if the next character of $T = P[i+1]$

Search Pattern					
a	a	b	a	a	a

	0	1	2	3	4	5
a	1	2	?	4	5	6
b	?	?	3	?	?	?



- How to fill these ???
 - Reset to q_0 ? Why not?

FSA construction

- ◆ FSA construction

- ◆ FSA builds itself

- ◆ Example. Build FSA for aabaaabb

- ◆ State 6. P[0..5]=aabaaa *状态0.*
 - ◆ assume you know state for p[1..5] = abaaa
 - ◆ if next char is b (match): go forward
 - ◆ if next char is a (mismatch): go to state for abaaaa
 - ◆ update X to state for p[1..6] = abaaab

有1,5,推1,6,1,6推1,7.

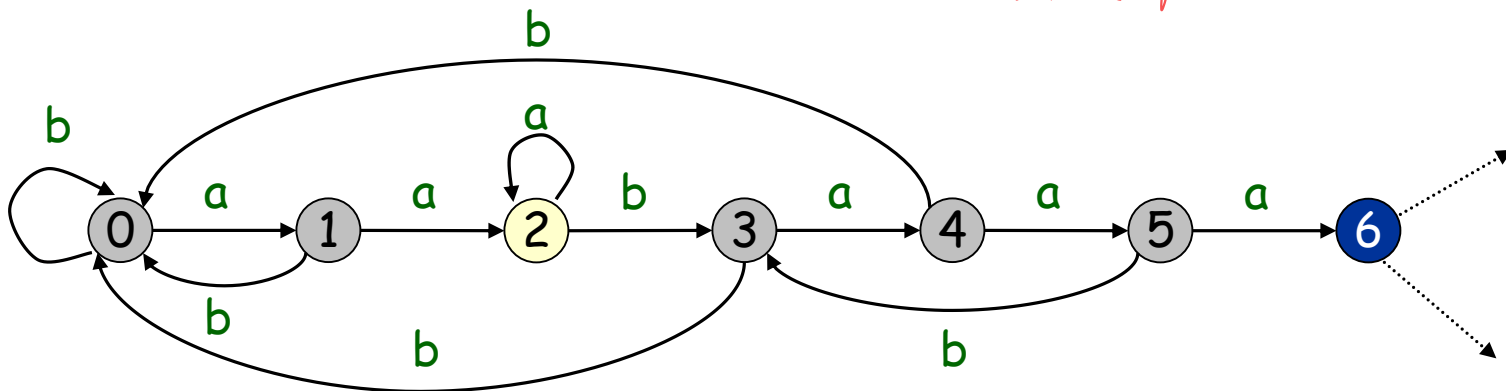
$X = 2$

$6 + 1 = 7$

$X + 'a' = 2$

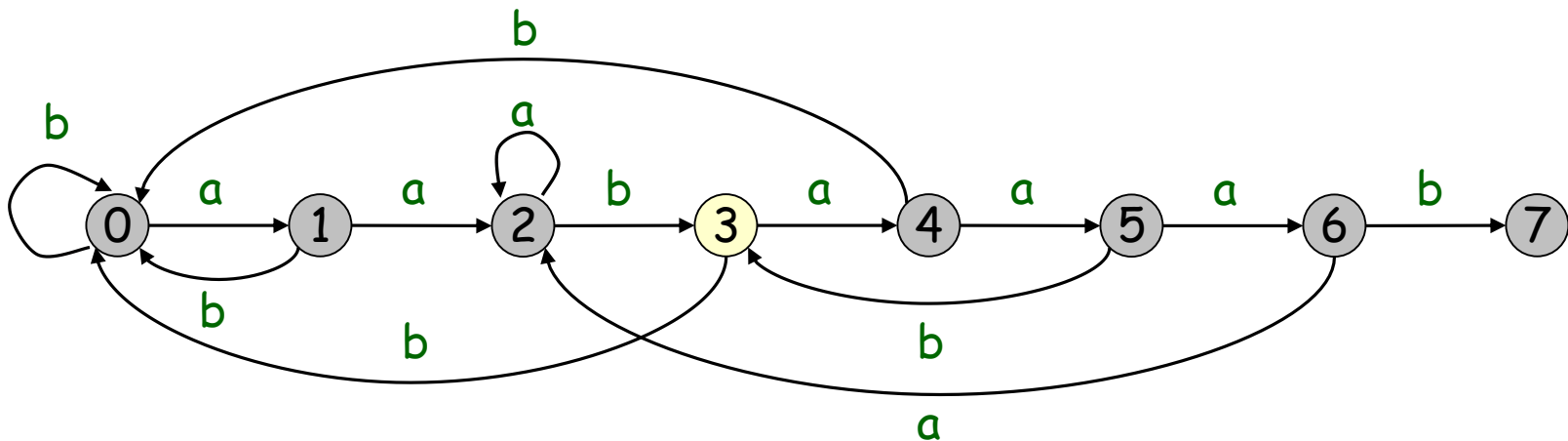
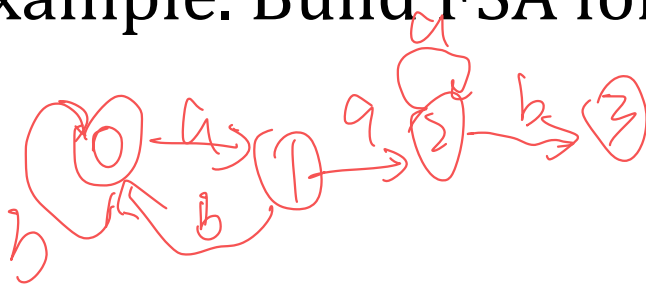
$X + 'b' = 3$

*↓
去头加b*



FSA construction

- ◆ FSA construction
 - ◆ FSA builds itself
- ◆ Example. Build FSA for aabaaabb



FSA construction

- ◆ FSA construction

- ◆ FSA builds itself

- ◆ Example. Build FSA for aabaaabb

- ◆ State 7. $p[0..6]=\text{aabaaab}$

- ◆ assume you know state for $p[1..6] = \text{abaaab}$

$X = 3$

- ◆ if next char is b (match): go forward

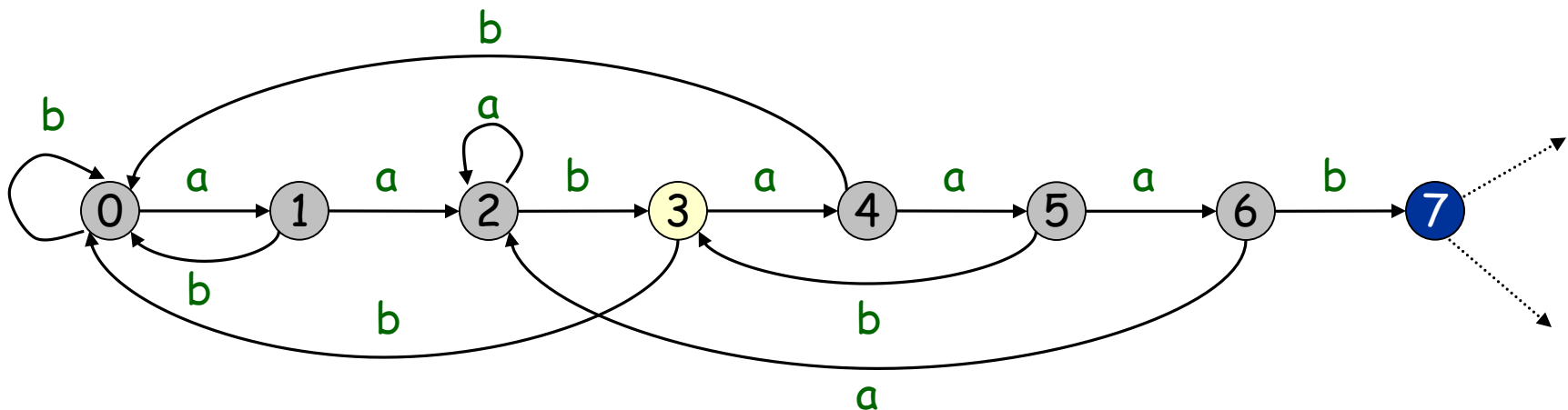
$7 + 1 = 8$

- ◆ if next char is a (mismatch): go to state for abaaaba

$X + 'a' = 4$

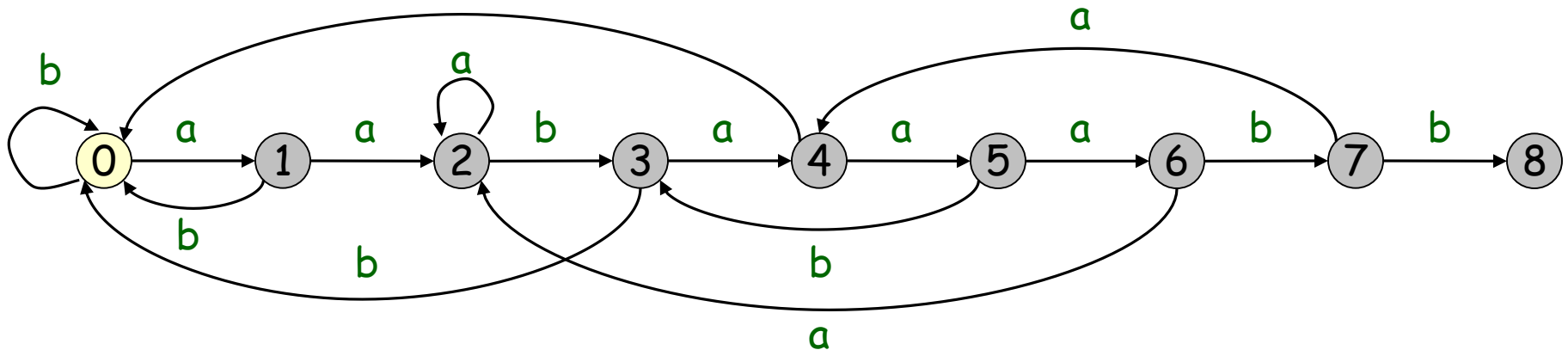
- ◆ update X to state for $p[1..7] = \text{abaaabb}$

$X + 'b' = 0$



FSA construction

- ◆ FSA construction
 - ◆ FSA builds itself
- ◆ Example. Build FSA for aabaaabb



FSA construction

◆ FSA construction

- ◆ FSA builds itself

O(mn)

◆ Crucial Insight

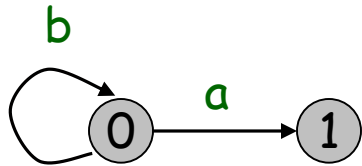
- ◆ To compute transitions for state n of FSA, suffices to have:
 - ◆ FSA for state 0 to $n-1$
 - ◆ State X that FSA ends up in with input $p[1..n-1]$
- ◆ To compute state X' that FSA ends up in with input $p[1..n]$, it suffices to have
 - ◆ FSA for states 0 to $n-1$
 - ◆ State X that FSA ends up in with input $p[1..n-1]$

FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

j	pattern[1..j]	x
---	---------------	---

a
b



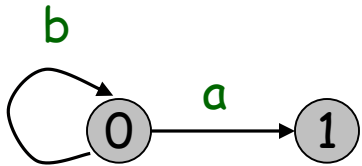
FSA construction

Search Pattern							
a	a	b	a	a	a	b	b



j	pattern[1..j]							x
0								0

	0
a	1
b	0



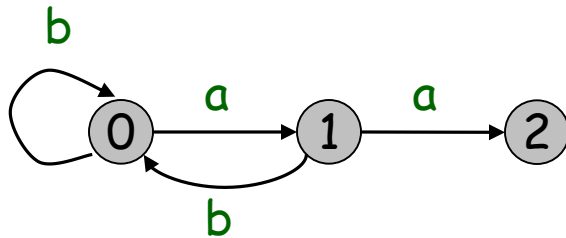
FSA construction

Search Pattern							
a	a	b	a	a	a	b	b



j	pattern[1..j]							x
0								0
1	a							1

	0	1
a	1	2
b	0	0



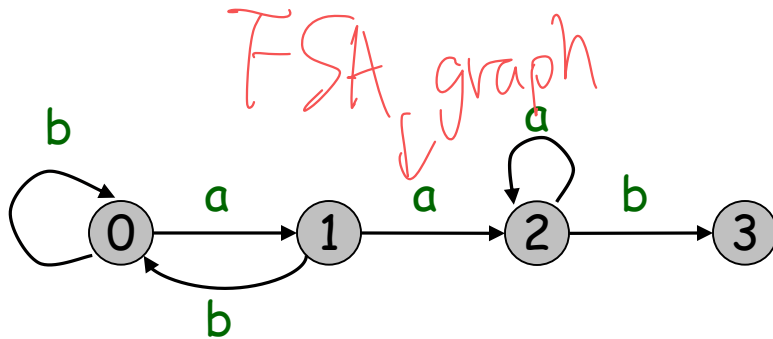
FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

	0	1	2
a	1	2	2
b	0	0	3



j	pattern[1..j]							x
0								0
1	a							1
2	a	b						0



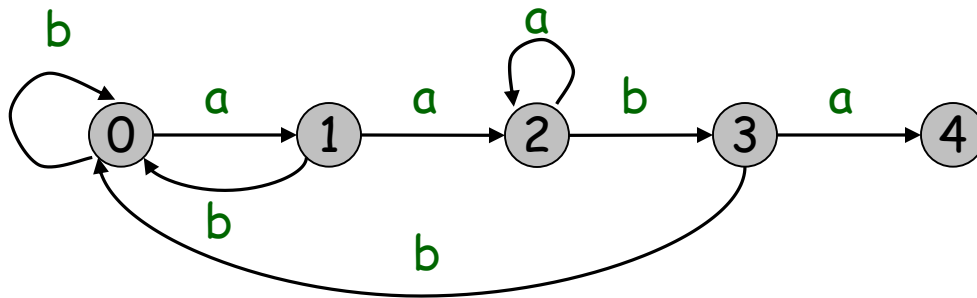
FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

	0	1	2	3
a	1	2	2	4
b	0	0	3	0



j	pattern[1..j]							x
0								0
1	a							1
2	a	b						0
3	a	b	a					1



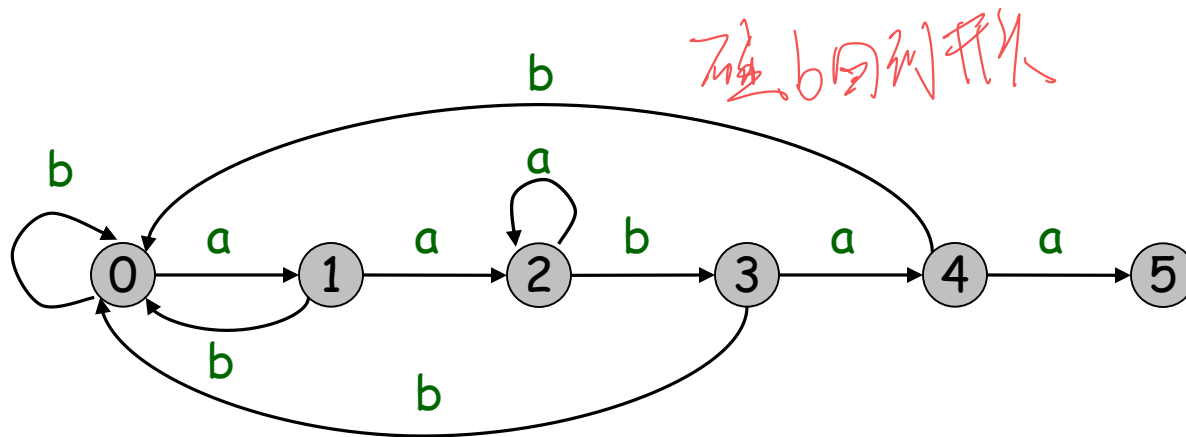
FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

	0	1	2	3	4
a	1	2	2	4	5
b	0	0	3	0	0



j	pattern[1..j]							x
0								0
1	a							1
2	a	b						0
3	a	b	a					1
4	a	b	a	a				2



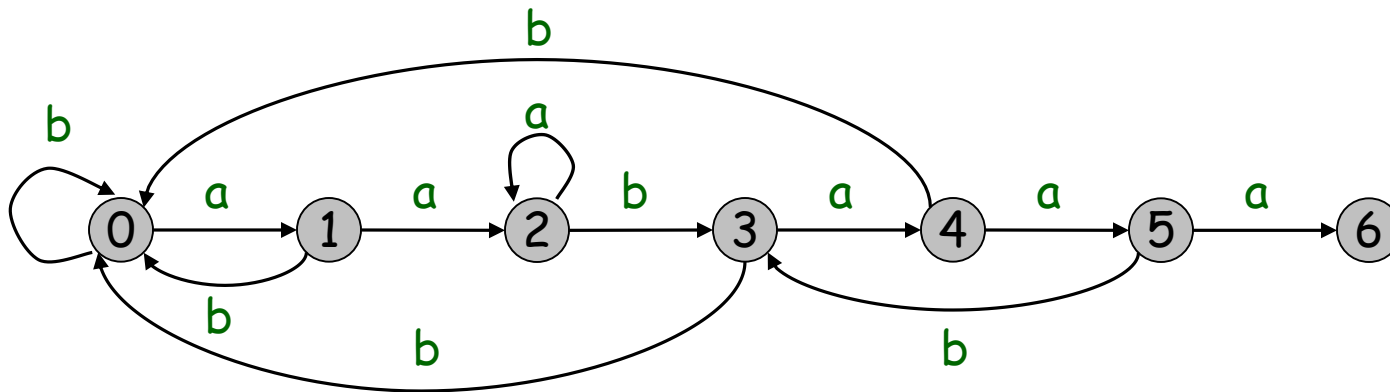
FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

	0	1	2	3	4	5
a	1	2	2	4	5	6
b	0	0	3	0	0	3



j	pattern[1..j]							x
0								0
1	a							1
2	a	b						0
3	a	b	a					1
4	a	b	a	a				2
5	a	b	a	a	a			2

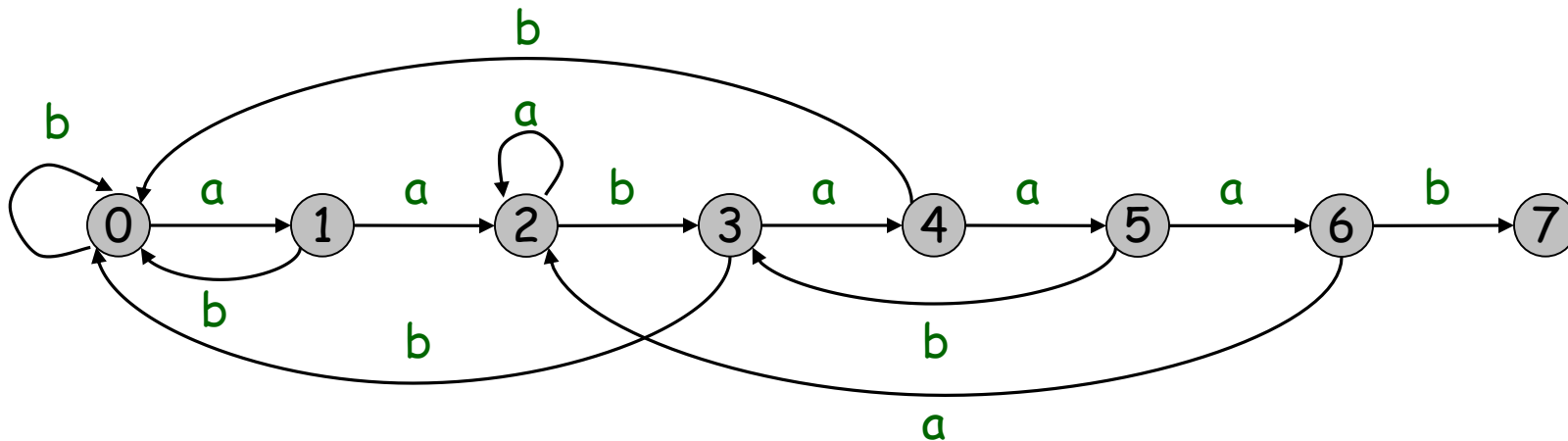


FSA construction → 状态转移图

Search Pattern							
a	a	b	a	a	a	b	b

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

j	pattern[1..j]							x
0								0
1	a							1
2	a	b						0
3	a	b	a					1
4	a	b	a	a				2
5	a	b	a	a	a			2
6	a	b	a	a	a	b		3

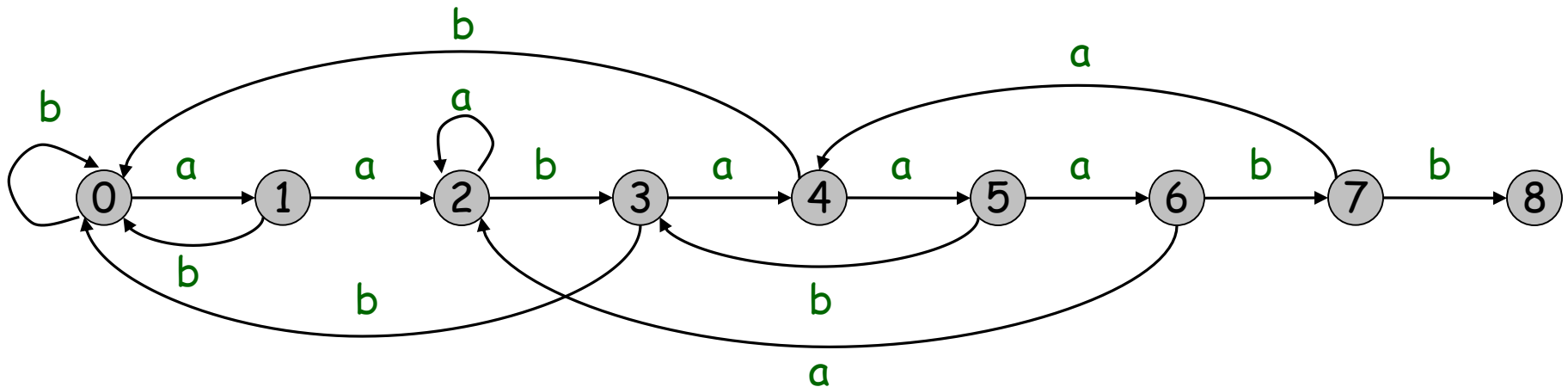


FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

	0	1	2	3	4	5	6	7
a	1	2	2	4	5	6	2	4
b	0	0	3	0	0	3	7	8

j	pattern[1..j]								x
0									0
1	a								1
2	a	b							0
3	a	b	a						1
4	a	b	a	a					2
5	a	b	a	a	a				2
6	a	b	a	a	a	b			3
7	a	b	a	a	a	b	b		0



Transition function

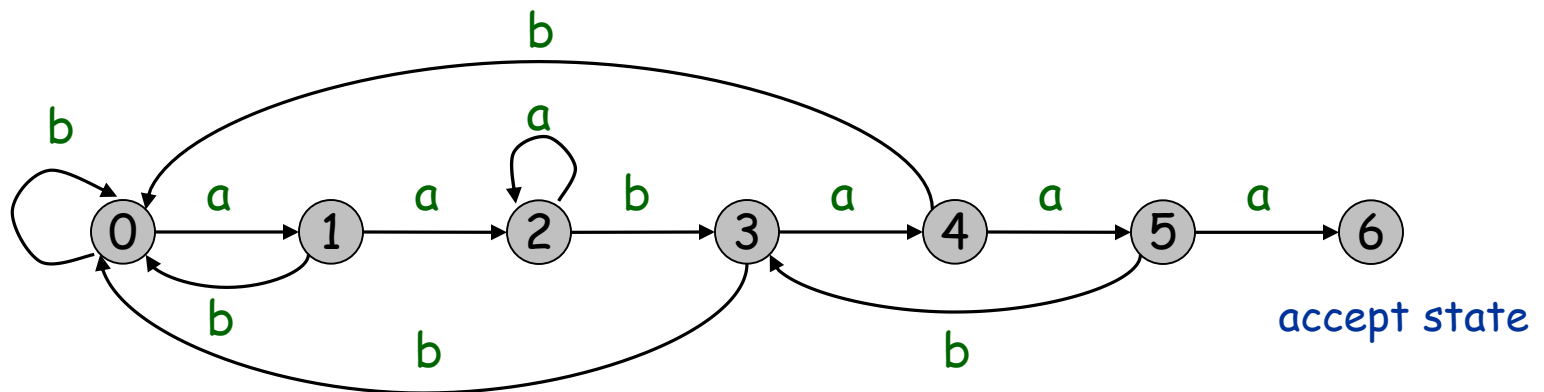
♦ Algorithm: Transition(P, Σ):

1. $m \leftarrow \text{len}(P)$
2. $X \leftarrow \emptyset$
3. Initialize $\delta(\emptyset, a)$ for each $a \in \Sigma$
4. **for** $j \leftarrow 1$ to $m-1$
5. **for** each character $a \in \Sigma$
6. **if** $P[j+1] = a$ then // char match
7. $\delta(j, a) \leftarrow j + 1$
8. **else** // char mismatch
9. $\delta(j, a) \leftarrow \delta(X, a)$
10. $X \leftarrow \delta(X, P[j+1])$
11. **return** δ

Finite State Automata (FSA)

- ◆ FSA-matching algorithm.
 - ◆ Use knowledge of how search pattern repeats itself.
- ➡ ◆ Build FSA from pattern.
- ◆ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

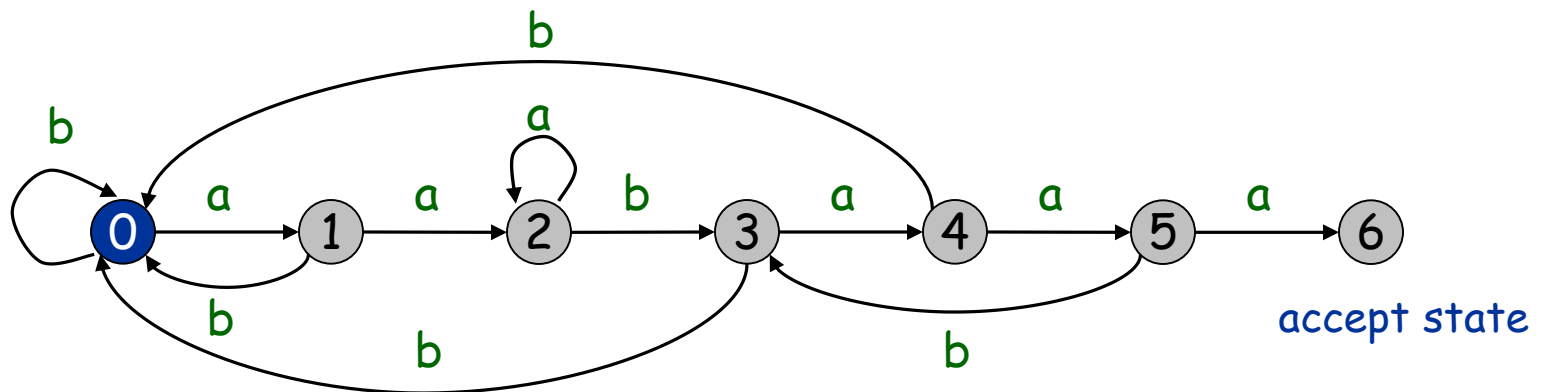


Finite State Automata (FSA)

- ◆ FSA-matching algorithm.
 - ◆ Use knowledge of how search pattern repeats itself.
 - ◆ Build FSA from pattern.
- ➔ ◆ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b



Finite State Automata (FSA)

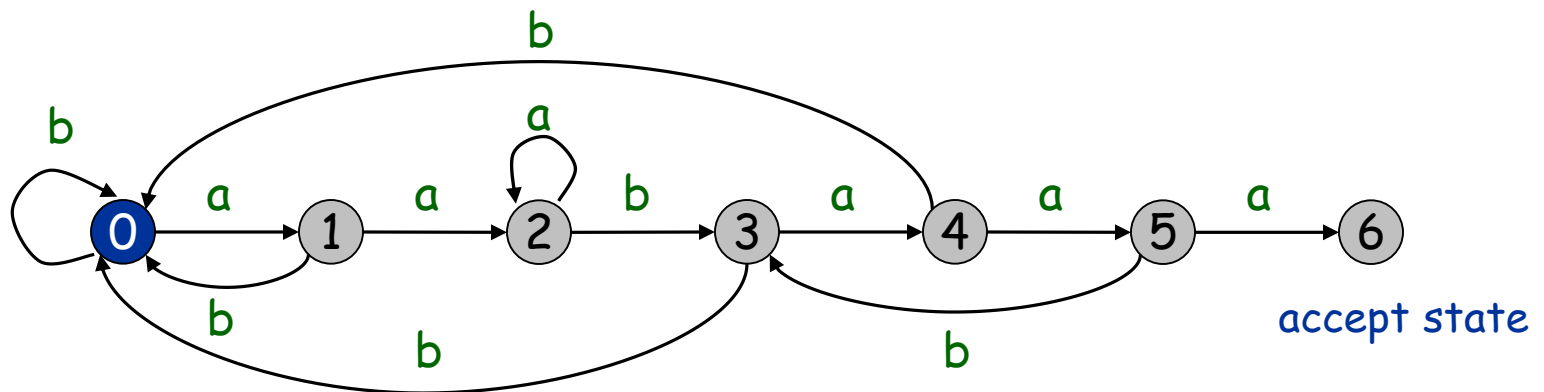
- ◆ FSA-matching algorithm

- ◆ Use knowledge of how search pattern repeats itself.
- ◆ Build FSA from pattern.

➡ ◆ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b



Finite State Automata (FSA)

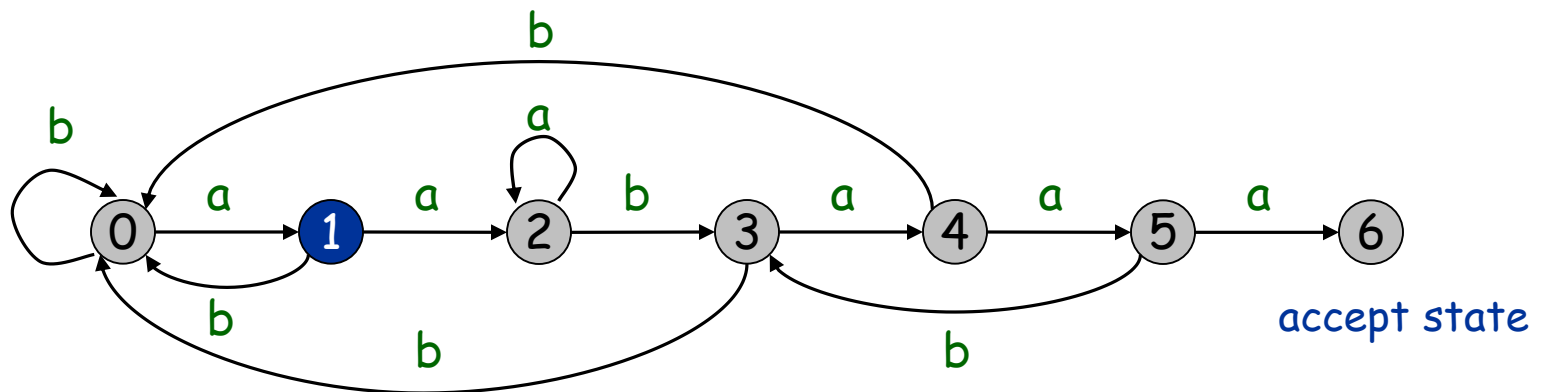
❖ FSA-matching algorithm

- ❖ Use knowledge of how search pattern repeats itself.
- ❖ Build Finite State Automata (FSA) from pattern.

➡ ❖ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

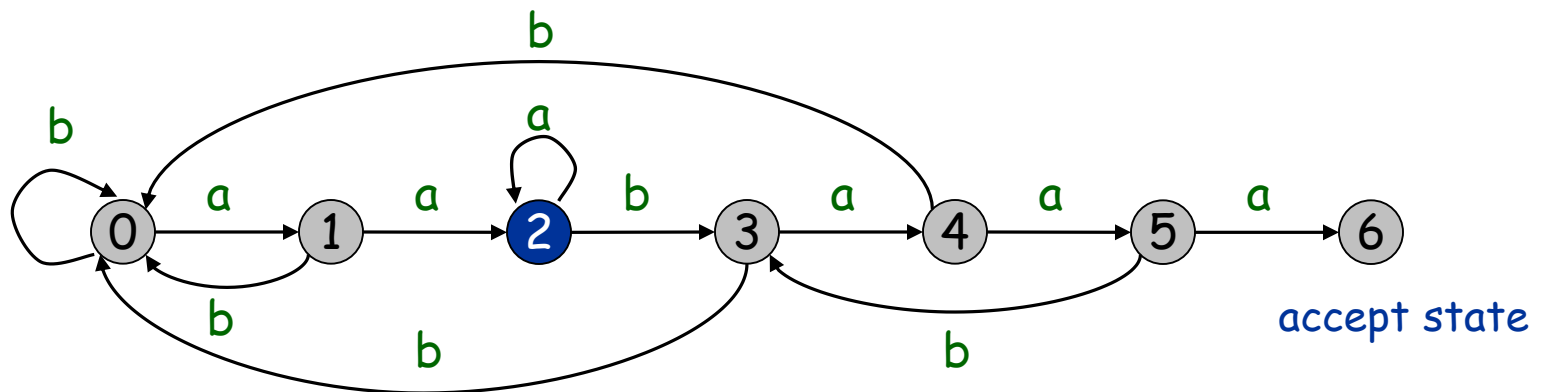


Finite State Automata (FSA)

- ❖ FSA-matching algorithm.
 - ❖ Use knowledge of how search pattern repeats itself.
 - ❖ Build FSA from pattern.
- ➡ ❖ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

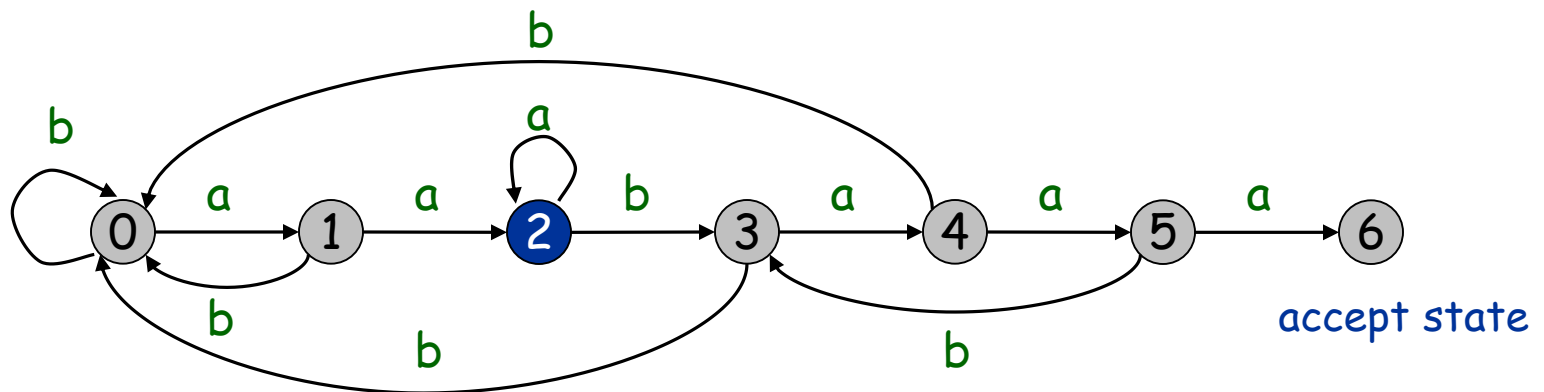


Finite State Automata (FSA)

- ❖ FSA-matching algorithm.
 - ❖ Use knowledge of how search pattern repeats itself.
 - ❖ Build FSA from pattern.
- ➡ ❖ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

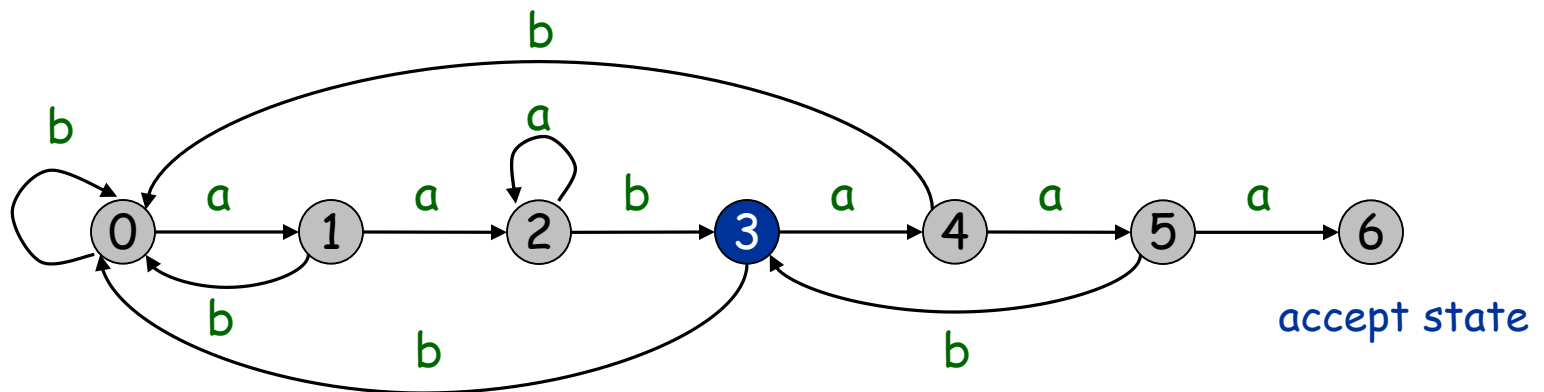


Finite State Automata (FSA)

- ◆ FSA-matching algorithm.
 - ◆ Use knowledge of how search pattern repeats itself.
 - ◆ Build FSA from pattern.
- ➡ ◆ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

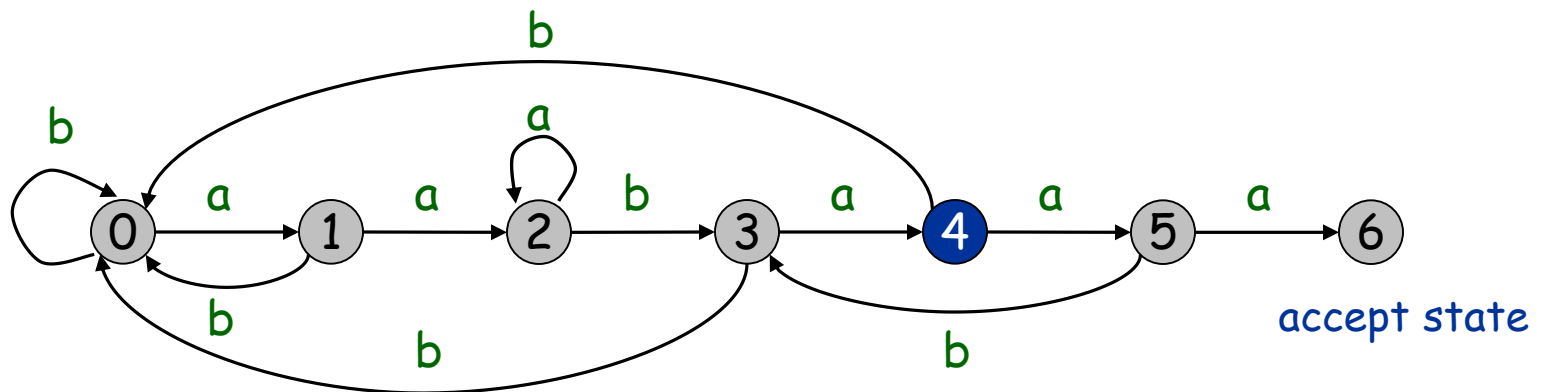


Finite State Automata (FSA)

- ◆ FSA-matching algorithm.
 - ◆ Use knowledge of how search pattern repeats itself.
 - ◆ Build FSA from pattern.
- ➡ ◆ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

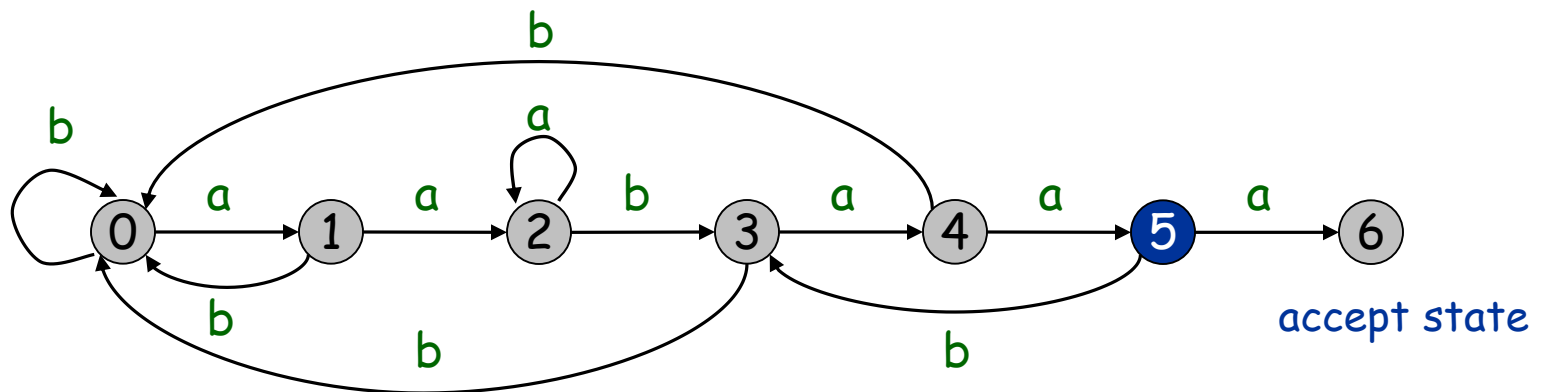


Finite State Automata (FSA)

- ◆ FSA-matching algorithm.
 - ◆ Use knowledge of how search pattern repeats itself.
 - ◆ Build FSA from pattern.
- ➡ ◆ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

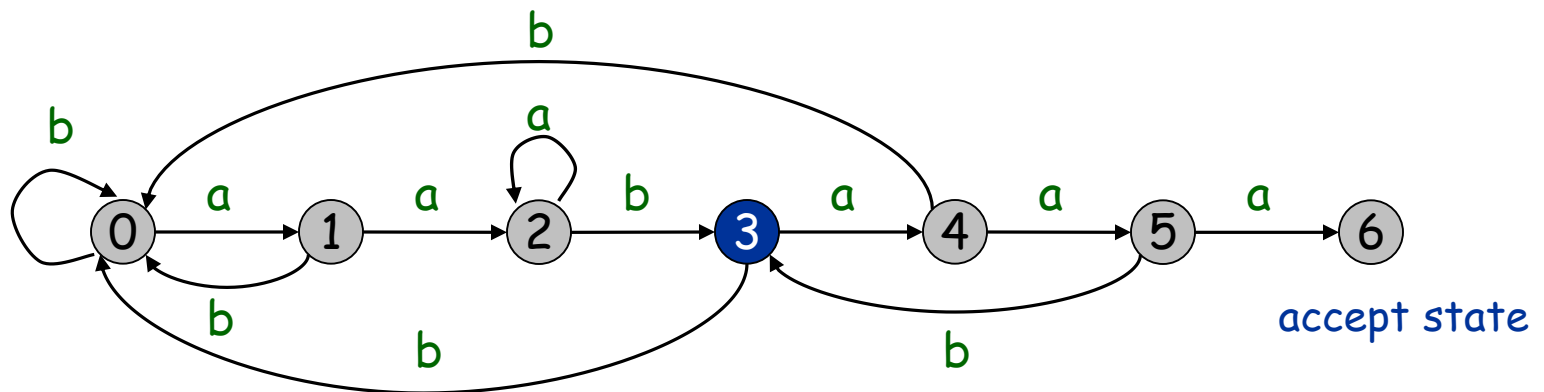


Finite State Automata (FSA)

- ❖ FSA-matching algorithm.
 - ❖ Use knowledge of how search pattern repeats itself.
 - ❖ Build FSA from pattern.
- ➡ ❖ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

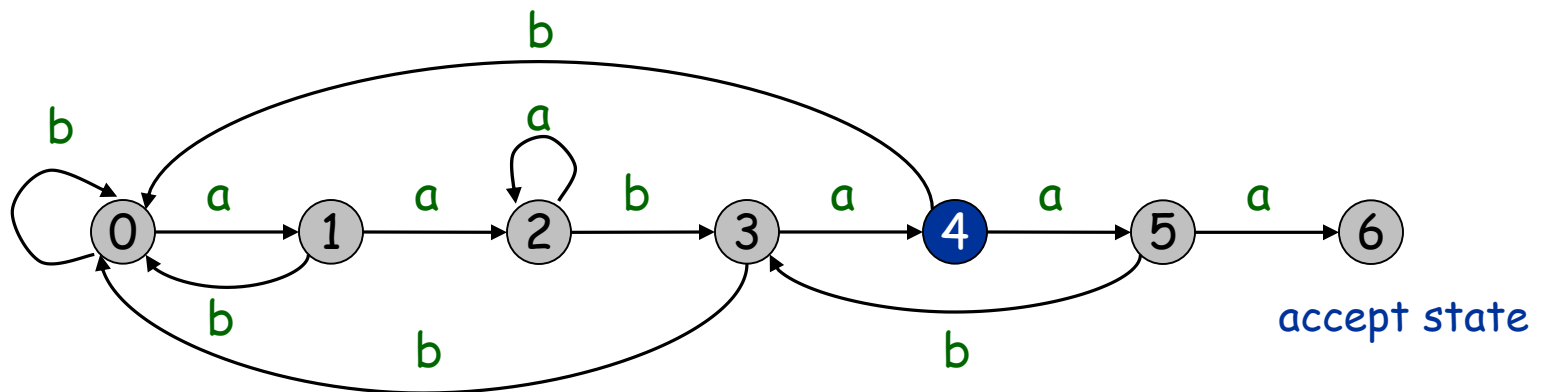


Finite State Automata (FSA)

- ❖ FSA-matching algorithm.
 - ❖ Use knowledge of how search pattern repeats itself.
 - ❖ Build FSA from pattern.
- ➡ ❖ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

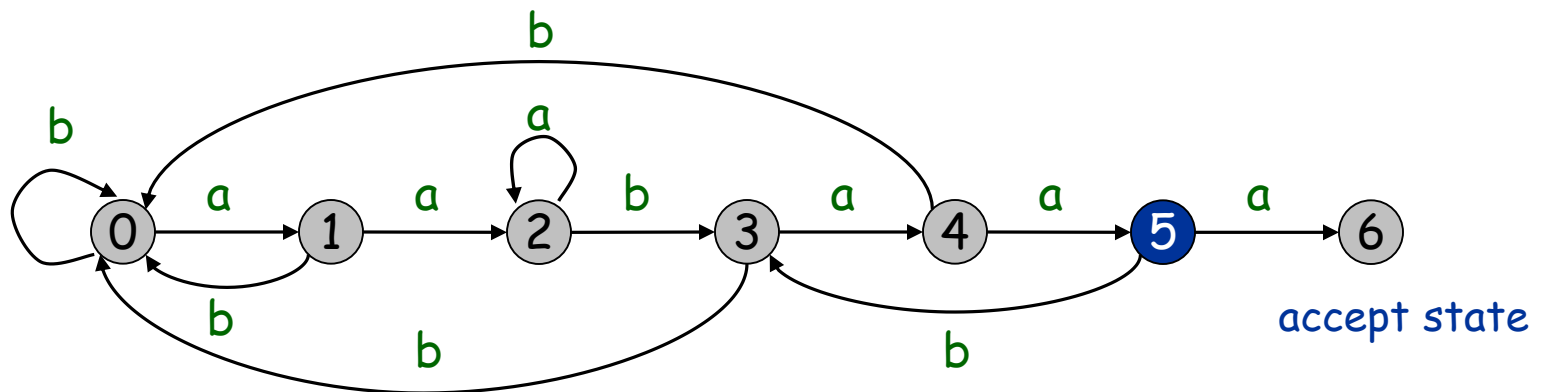


Finite State Automata (FSA)

- ◆ FSA-matching algorithm.
 - ◆ Use knowledge of how search pattern repeats itself.
 - ◆ Build FSA from pattern.
- ➔ ◆ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b

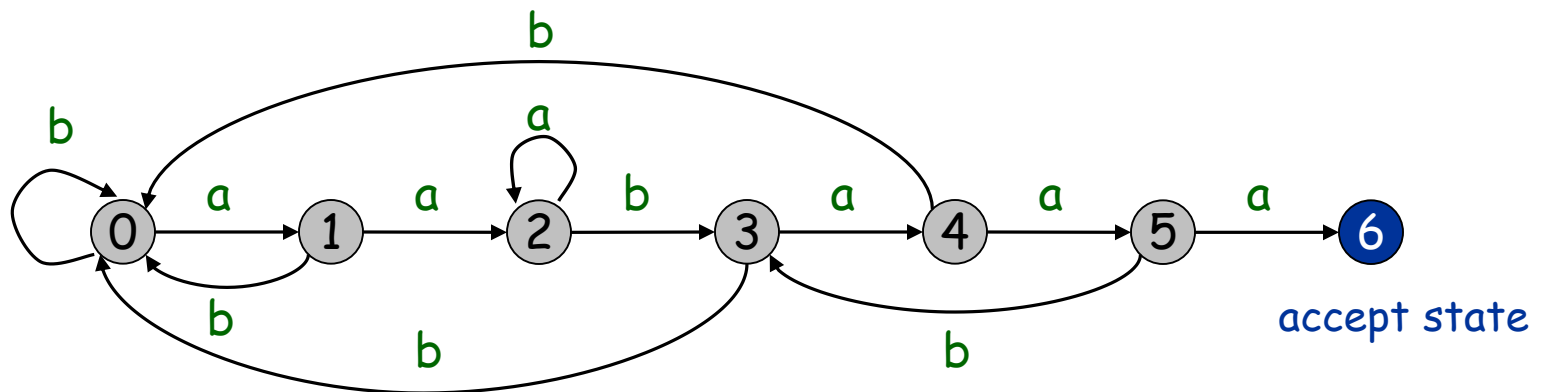


Finite State Automata (FSA)

- ❖ FSA-matching algorithm.
 - ❖ Use knowledge of how search pattern repeats itself.
 - ❖ Build FSA from pattern.
- ➡ ❖ Run FSA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	b	a	a	a	b	a	a	a	b

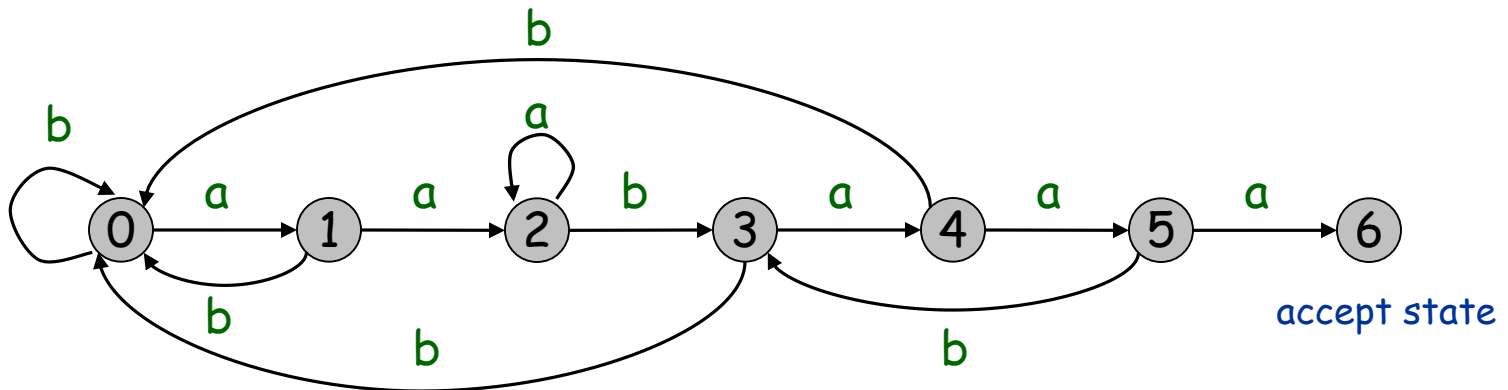


Finite State Automata (FSA)

- ◆ FSA used in KMP has special property
 - ◆ If match, go to next state
 - ◆ Only need to keep track of where to go upon character mismatch.
 - ◆ go to state $\text{next}[j]$ if character mismatches in state j

Search Pattern					
a	a	b	a	a	a

	0	1	2	3	4	5
a	1	2	2	4	5	6
b	0	0	3	0	0	3
next	0	0	2	0	0	3



FSA algorithm

♦ Algorithm: FSA(T, P):

1. $n \leftarrow \text{len}(T)$, $m \leftarrow \text{len}(P)$
2. $\delta \leftarrow \text{Transition}(P, \Sigma)$
3. $q \leftarrow \emptyset$ // q is the state of the FSA.
4. **for** $i \leftarrow 1$ to n
5. $q \leftarrow \delta(q, T[i])$
6. **if** $q = m$
7. pattern occurs with shift $i - m$

Analysis of FSA

♦ **Algorithm:** $FSA(T, P)$:

Cost of Line 1:

Cost of Line 2:

Cost of Line 3:

Cost of Line 4:

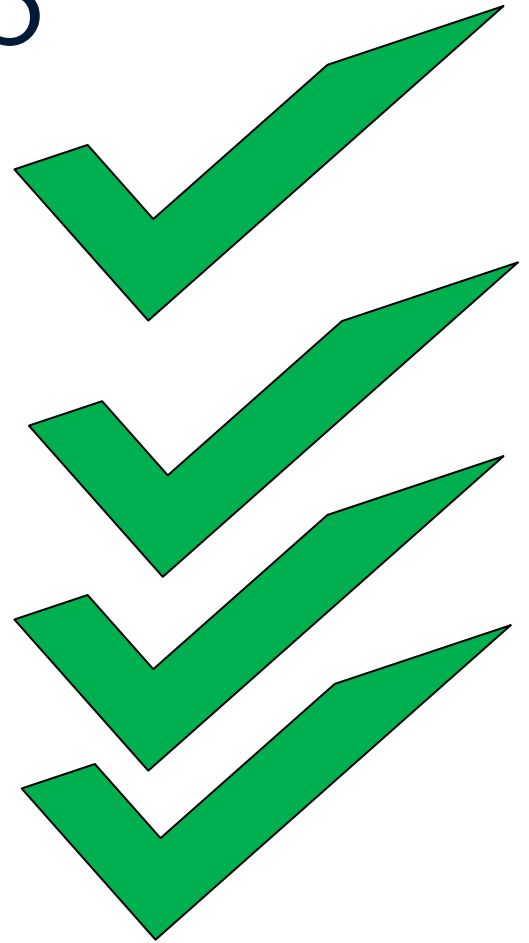
...

Cost of Line 7:

Overall Cost:

Our Roadmap

- ◆ String Concepts
- ◆ String Searching Problem
 - ◆ Brute Force Solution
 - ◆ Rabin-Karp
 - ◆ Finite State Automata
 - ◆ Knuth-Morris-Pratt



History of KMP

- ◆ Inspired by the theorem of Cook that says $O(m+n)$ algorithm should be possible
- ◆ Discovered in 1976 independently by two groups
- ◆ Knuth-Pratt
- ◆ Morris was hacker trying to build an editor
- ◆ Resolved theoretical and practical problem
 - ◆ Surprise when it was discovered
 - ◆ In hindsight, seems like right algorithm

String

- ◆ **String:** “HelloCS203”
- ◆ **Substring:** a substring of a string S is a string S' that occurs in S , e.g., $P[2, \dots, 4] = \text{“ell”}$
- ◆ **Prefix ($P[1, \dots]$):** ^{前缀串} a prefix of a string S is a substring of S that occurs at the beginning of S , e.g., $P[1, \dots, 1] = \text{“H”}$ (note that $P[1] = \text{“H”}$), $P[1, \dots, 2] = \text{“He”}$, $P[1, \dots, 5] = \text{“Hello”}$, we denote prefix as: **$P[1, \dots]$**
- ◆ **Suffix:** ^{后缀串} a suffix of a string S is a substring of S that occurs at the end of S , e.g., $P[10, \dots, 10] = \text{“3”}$, $P[8, \dots, 10] = \text{“203”}$, $P[6, \dots, 10] = \text{“CS203”}$, we denote suffix as: **$P[\dots, m]$**

Finite State Automata

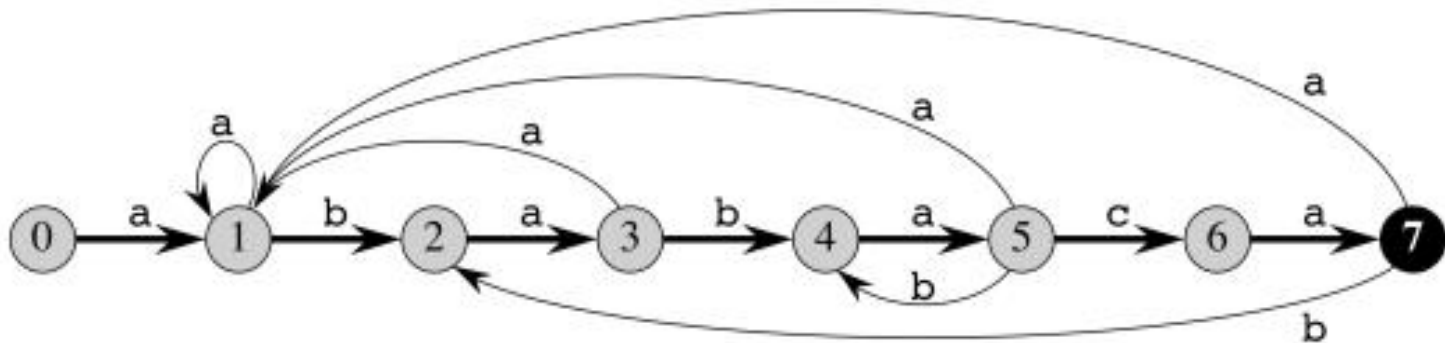
◆ P = "ababaca"

◆ Transition function table

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
P	a	b	a	b	a	c	a	

a 0 11 11
 ab 0 11 11
 aba 1 ab
 abab 2

◆ State transition graph



Finite State Automata

- ◆ P = “ababaca” and T = “abababacaba”

i	1	2	3	4	5	6	7	8	9	10	11
T	a	b	a	b	a	b	a	c	a	b	a
1	a	b	a	b	a	c	a				
2			a	b	a	b	a	c	a		
3									a	b	

- ◆ After **failure**: at $i=6$, ‘c’ was expected, but not found in $T[6]$, FSA transition to state $\delta(5,b)=4$, it means pattern prefix $P[1..4]$ = “abab” has matched the text suffix $T[2..6]$ = “abab”

	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

Finite State Automata

- ◆ In general, the FSA is constructed so that the state number tells us how much of a prefix of P has been matched.
- ◆ FSA transition function:
 - ◆ 1) Find the longest prefix of P is also a suffix of $T[...i]$, denote as k , i.e., $P[1,...,k]=T[i-k+1,...,i]$
 - ◆ 2) Read the next character at “ $k+1$ ” (i.e., $T[i+1]$), there are two kinds of transitions:
 - ◆ $P[k+1] = T[i+1]$, it is matched, continues.
 - ◆ Otherwise, it is mismatched, go to $\delta(k, T[i+1])$

Prefix Function

- ◆ Consider the first step of FSA transition function:
 - ◆ Find the longest prefix of P is also a suffix of $T[\dots i]$, denote as k , i.e., $P[1, \dots, k] = T[i-k+1, \dots, i]$
- ◆ Suppose it is mismatched at “ $P[k+1]$ ”, it means:
 - ◆ $P[k+1] \neq T[i+1]$ then,
 - ◆ we should find the longest prefix of $P[1, \dots, k]$ is also a suffix of $T[i-k+1, \dots, i]$.
- ◆ **Prefix function (next array in general),**
given $P[1..m]$, the prefix function π for P is $\pi : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m-1\}$ such that:
$$\pi[i] = \max\{k, k < i \text{ and } P[1, \dots, k] = P[i-k+1, \dots, i]\}$$

Prefix Function

- ◆ **Prefix function**, given P , the prefix function π for P is $\pi : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m-1\}$ such that:

$$\pi[q] = \max\{k, k < q \text{ and } P[1, \dots, k] = P[q-k+1, \dots, q]\}$$

- ◆ Example: $P = \text{"ababaca"}$

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

Compute next array

♦ Algorithm: NextArray(P):

1. $m \leftarrow \text{len}(P)$
2. Let $\pi[1, \dots, m]$ be a new array
3. $\pi[1] = 0, k \leftarrow 0$
4. **for** $q = 2$ to m
5. **while** $k > 0$ and $P[k+1] \neq P[q]$
6. $k \leftarrow \pi[k]$
7. **if** $P[k+1] = P[q]$
8. $k \leftarrow k + 1$
9. $\pi[q] \leftarrow k$
10. **return** π

KMP algorithm

◆ Algorithm: KMP(T, P):

```
1.  $n \leftarrow \text{len}(T)$ ,  $m \leftarrow \text{len}(P)$ 
2.  $\pi \leftarrow \text{NextArray}(P)$ 
3.  $q \leftarrow 0$ 
4. for  $i = 1$  to  $n$ 
5.     while  $q > 0$  and  $P[q+1] \neq T[i]$ 
6.          $q \leftarrow \pi[q]$ 
7.     if ( $P[q+1] = T[i]$ )
8.          $q \leftarrow q + 1$ 
9.     if  $q == m$ 
10.        print "Pattern occurs with shift"  $i-m$ 
11.         $q \leftarrow \pi[q]$ 
```

Our Roadmap

- ◆ String Concepts
- ◆ String Searching Problem
 - ◆ Brute Force Solution
 - ◆ Rabin-Karp
 - ◆ Finite State Automata
 - ◆ Knuth-Morris-Pratt



Thank You!