Lecture 6
String Matching

Bo Tang @ SUSTech, Fall 2020

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 = “100001010101010010101”
  - 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="SUSTechCS203", how many sub string it has?

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 >> m
    - e.g., n = 1 Billion, m = 100

Our Roadmap

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

Brute Force

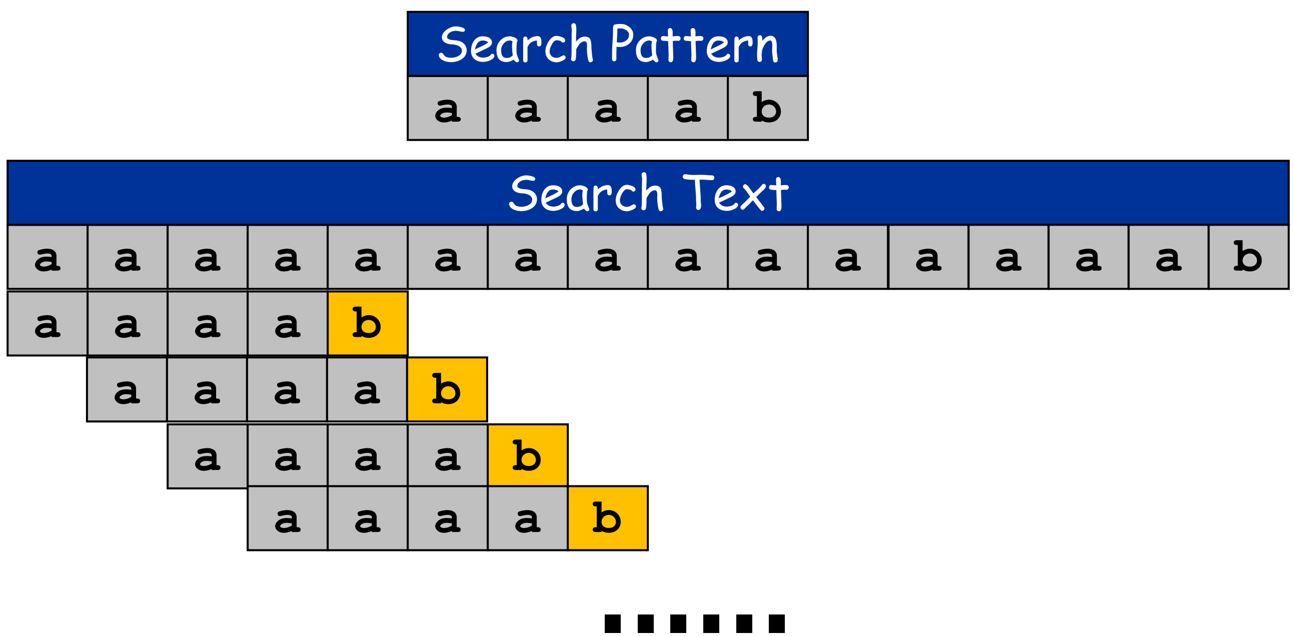
- Brute force
  - Check for pattern starting at every text position
- **Algorithm:** BruteForce(T, P):

```
n ← len(T), m ← len(P)
for i ← 0 to n-m-1
        for j ← 0 to m-1
                if P[j] != T[i+j] then
                        break;
        if j = m-1
            pattern occurs with shift i
```

- Time complexity?

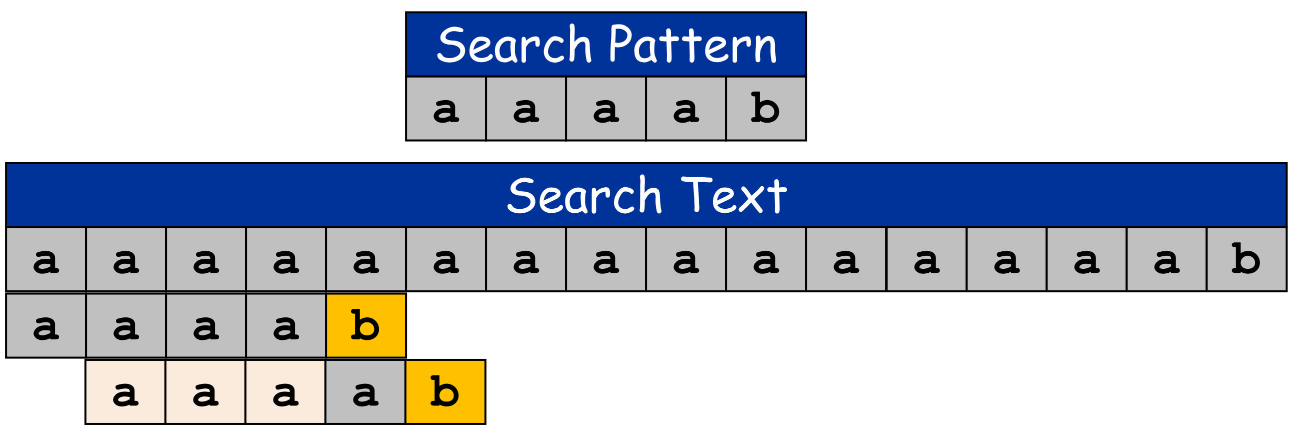
Analysis of Brute Force

- Analysis of brute force
  - Running time depends on pattern and text
  - Can be slow when strings repeat themselves
  - Worst case: mn comparisions
  - 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



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		

- Any idea?

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],…,t[n-m-1]
  - Compare p with t[i], for each i in [0,n-m-1]
  - if p=t[i], pattern p occurs
- Example
  - p = 1359
  - Array t is:

2468	4680	6801	8012	121	1213	2135	1359	3597	5972

  - t[7] = p → T[7,8,9,10]=P[0,1,2,3]

Rabin-Karp Algorithm

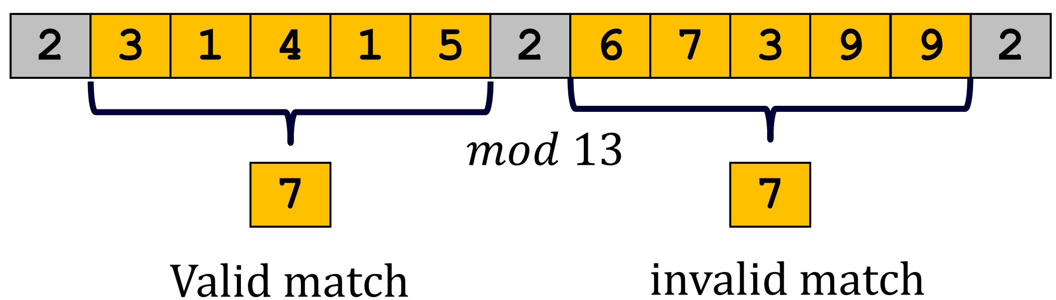
- How to convert size-m characters to a number?
  - E.g., the alphabet $\Sigma = \{a,\ldots,z,A,\ldots,Z\}$
  - Solution: radix-d ($d=|\Sigma|$) Horner's rule
  - $p = P[m-1]+d(P[m-2]+d(P[m-3]+\ldots+d(P[1]+dP[0])))$
- When m is large, p may be too large to work
  - Modulo a proper prime number q
  -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

Rabin-Karp Algorithm

- Correctness analysis
  - $p \not\equiv t[i]\ (mod\ q)$ we have $p \neq t[i]$, thus, P[0,..m-1] != T[i,i+m-1]
  - $p \equiv t[i]\ (mod\ q)$, it does not imply $p = t[i]$ (**spurious hit**)

- Example: search P:

3	1	4	1	5



- Additional test to check
  - P[0,...,m-1] = T[i, i+m-1]

Rabin-Karp Algorithm

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

```
n ← len(T), m ← len(P)
h ← d^(m-1) (mod q), p ← 0, t0 ← 0
for j ← 0 to m-1
      p ← (dp + P[j]) mod q,
      t_0 ← (dt_0 + T[j]) mod q,
for i ← 0 to n-m
      if p != t_i then
            t_{i+1} ← (d(t_i-T[i]h)+T[i+m]) mod q
      else
            If P[0,..m-1]=T[i,i+m-1]
                  pattern occurs with shift I
            Else
                  t_{i+1} ← (d(t_i-T[i]h)+T[i+m]) mod 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
  - Knuth-Morris-Pratt

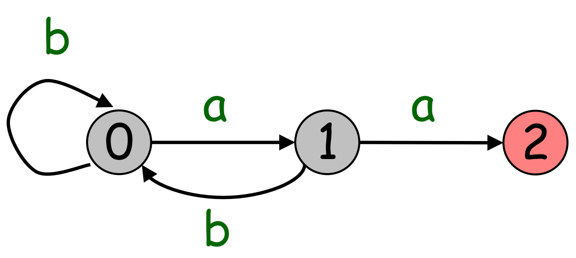
Midterm Exam

- **Time: 14 Nov. 10:00-12:00**
- **Venue: Teaching Building 1 401 -405**
- **Scope: Lecture 1 to 6**

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

	0	1
a	1	2
b	0	0

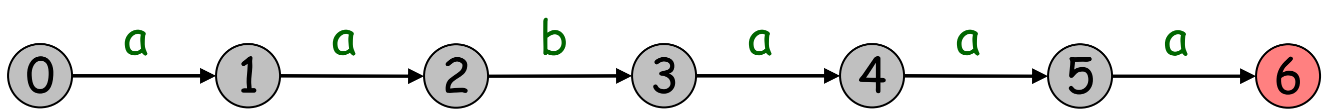


FSA idea for String Matching

- 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

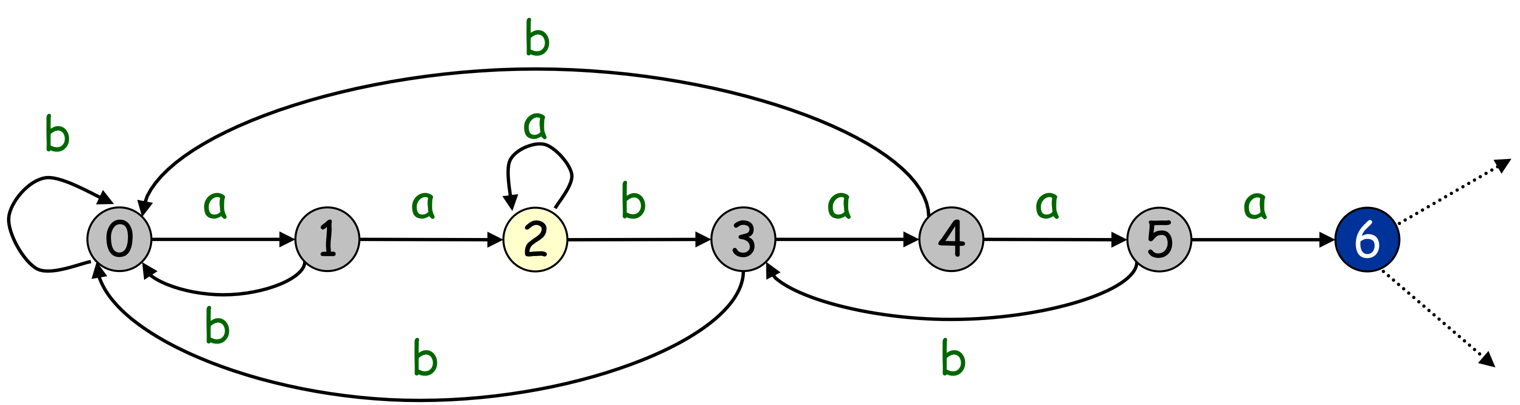
	0	1	2	3	4	5
a	1	2	?	4	5	6
b	?	?	3	?	?	?



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

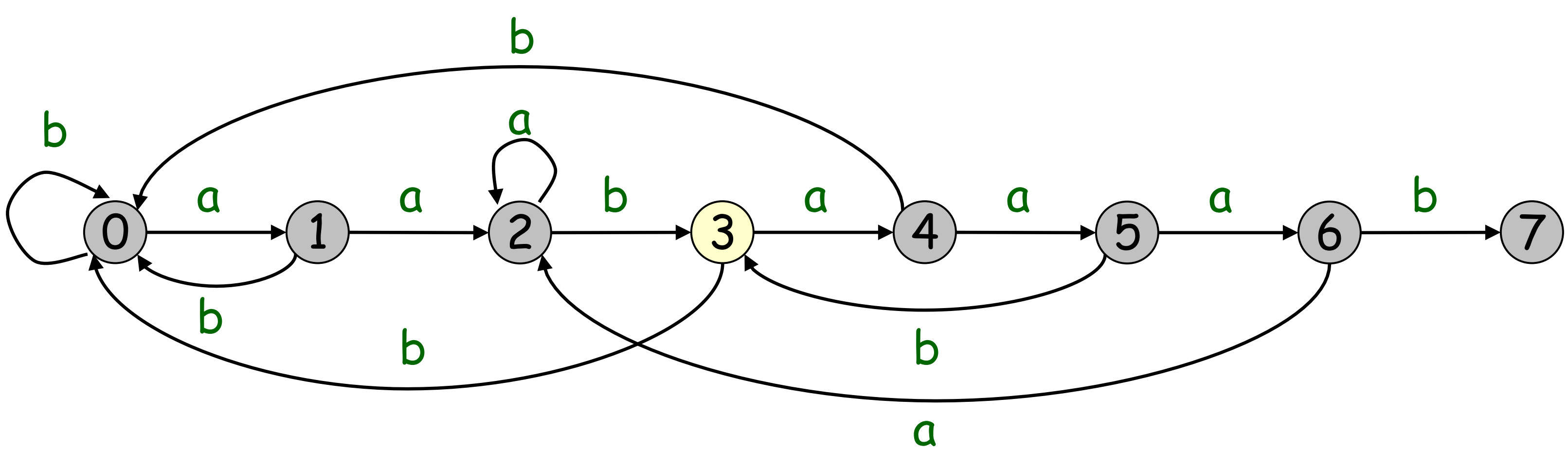
FSA construction

- FSA construction
  - FSA builds itself
- Example. Build FSA for `aabaaabb`
  - State 6. P[0..5]=aabaaa
  - assume you know state for p[1..5] = abaaa X = 2
  - if next char is b (match): go forward 6 + 1 = 7
  - if next char is a (mismatch): go to state for abaaaa X + 'a' = 2
  - update X to state for p[1..6] = abaaab X + 'b' = 3



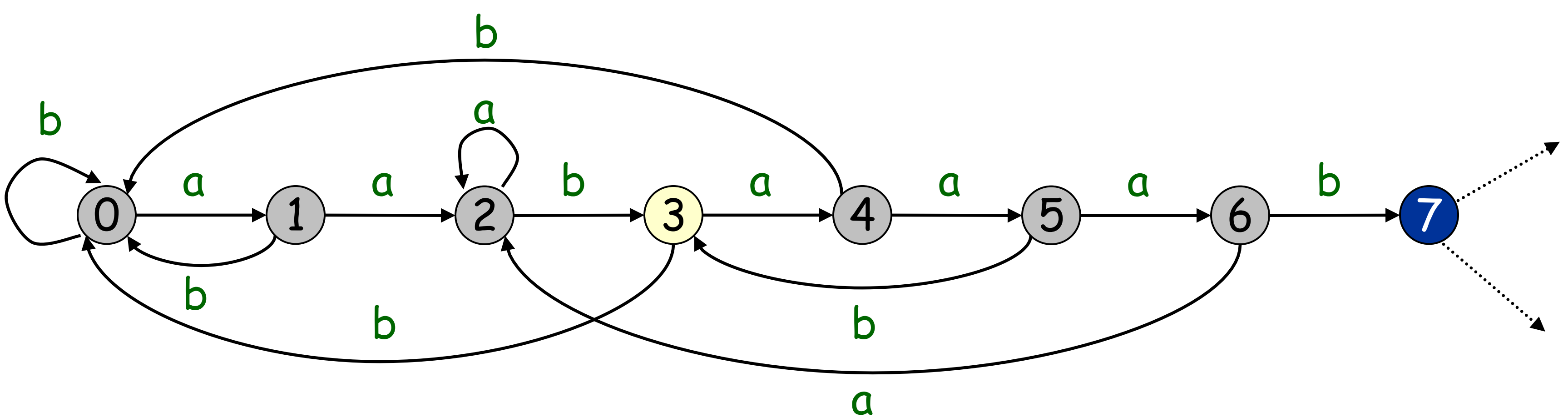
FSA construction

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



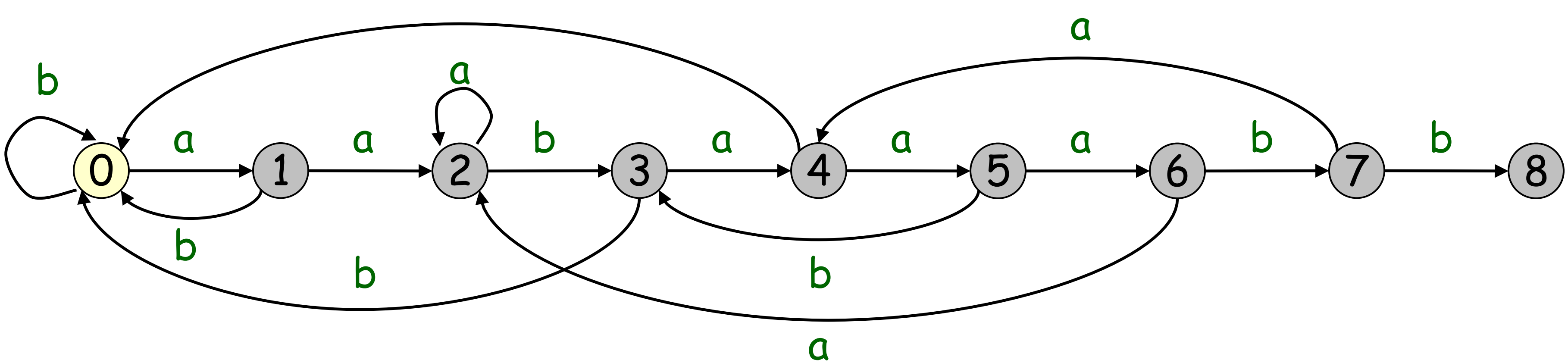
FSA construction

- FSA construction
  - FSA builds itself
- Example. Build FSA for `aabaaabb`
  - State 7. p[0..6]=aabaaab
  - assume you know state for p[1..6] = 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] = abaaabb X + 'b' = 0



FSA construction

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



FSA construction

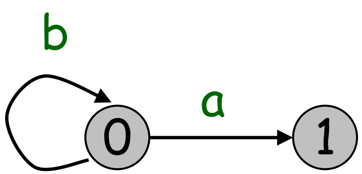
- FSA construction
  - FSA builds itself
- Crucial Insight
  - To compute transitions for state n of FSA, suffices to have:
    - FSA for state 0 to n-1
    - State X that FSA ends up in with input p[1..n-1]
  - To compute state X' that FSA ends up in with input p[1..n], it suffices to have
    - FSA 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

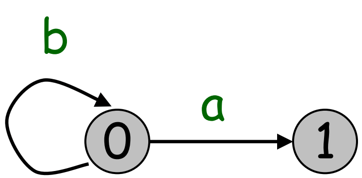


FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

j	pattern[1..j]							X
0								0

	0
a	1
b	0

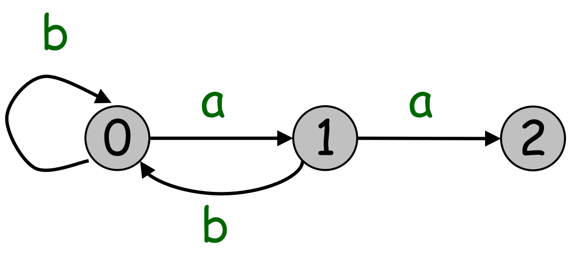


FSA construction

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

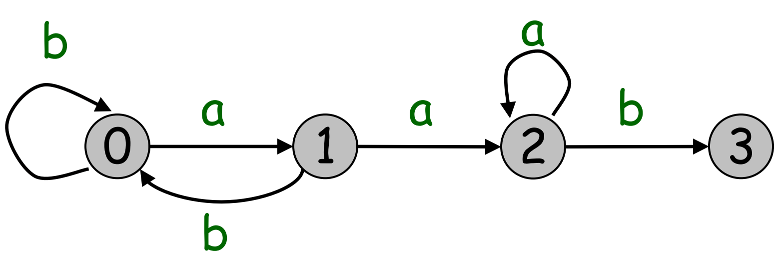


FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

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

	0	1	2
a	1	2	2
b	0	0	3

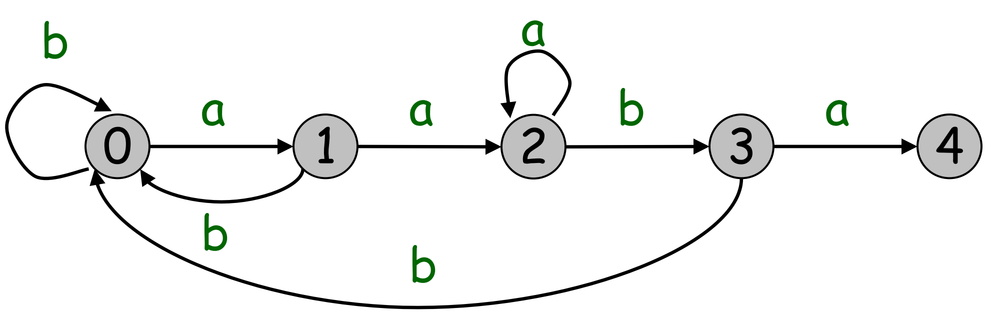


FSA construction

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

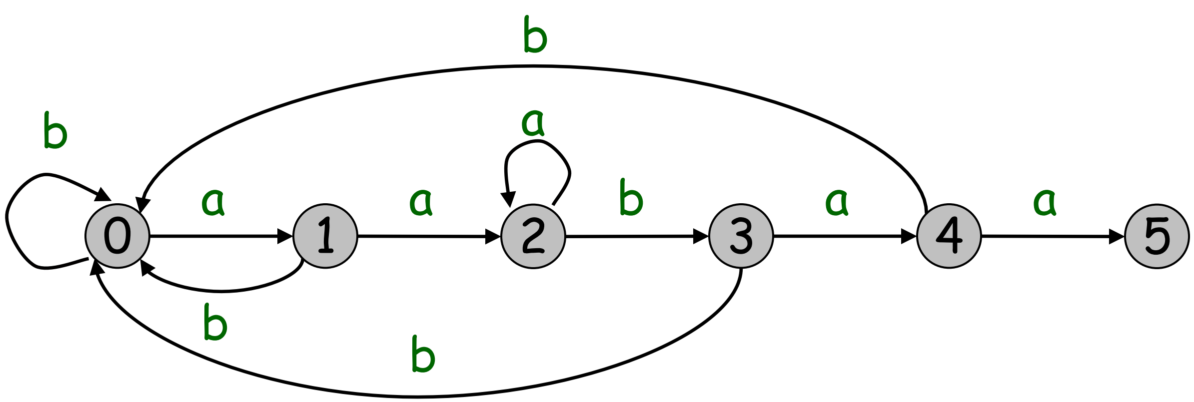


FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

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

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

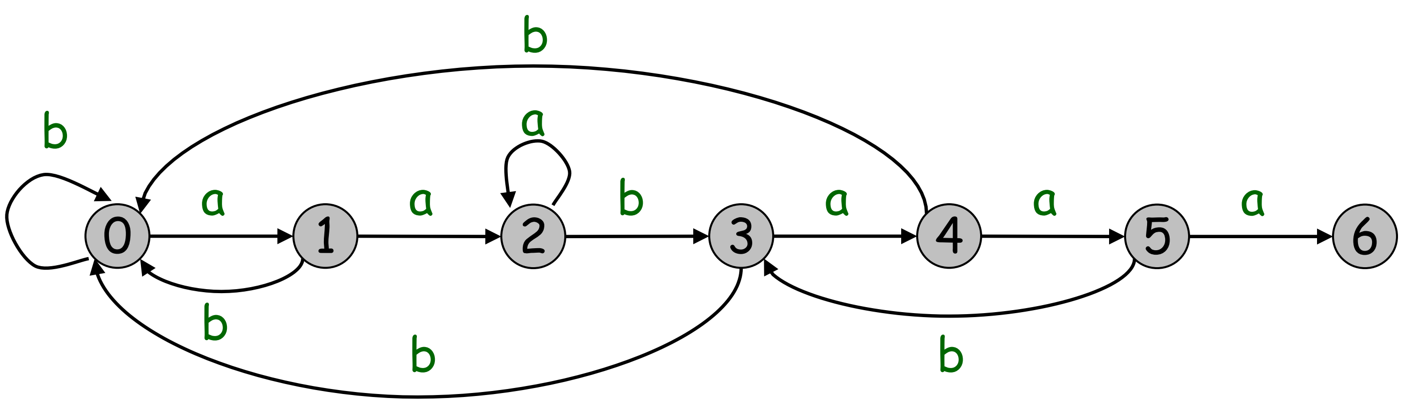


FSA construction

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

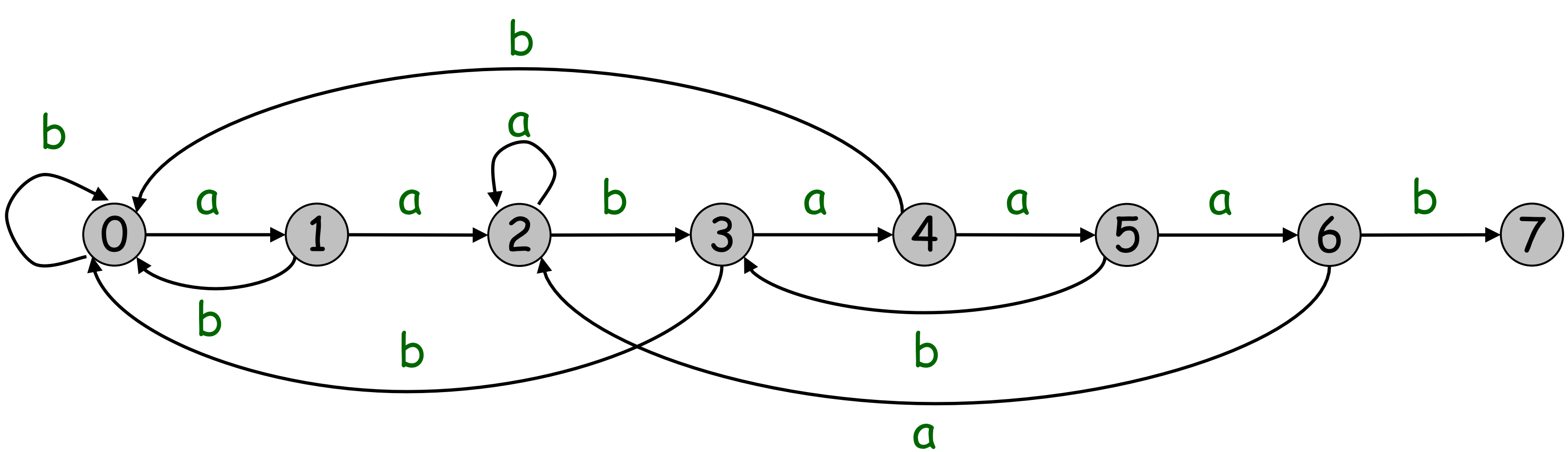


FSA construction

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

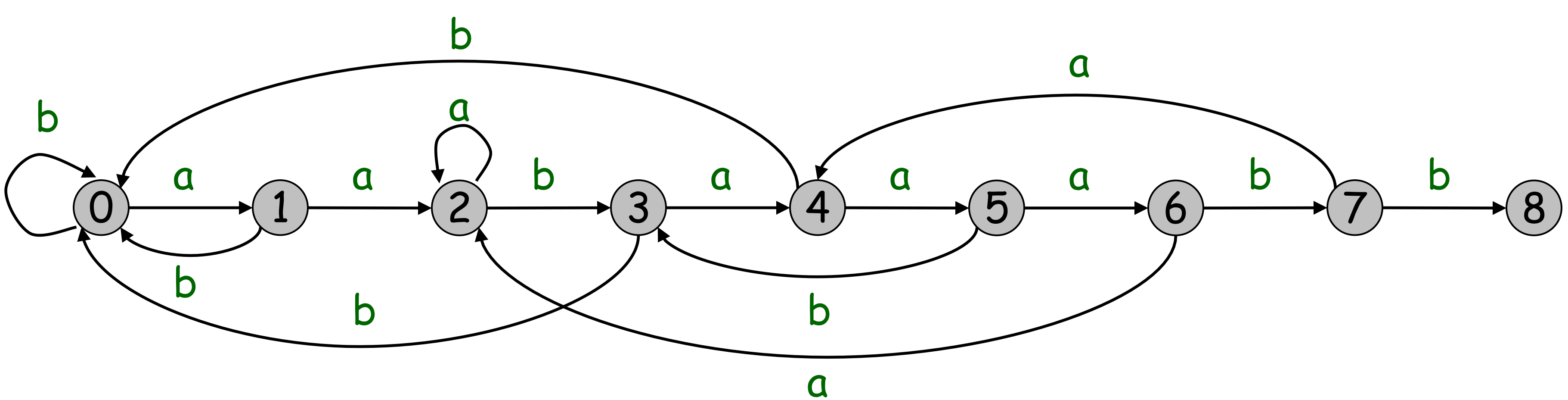


FSA construction

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

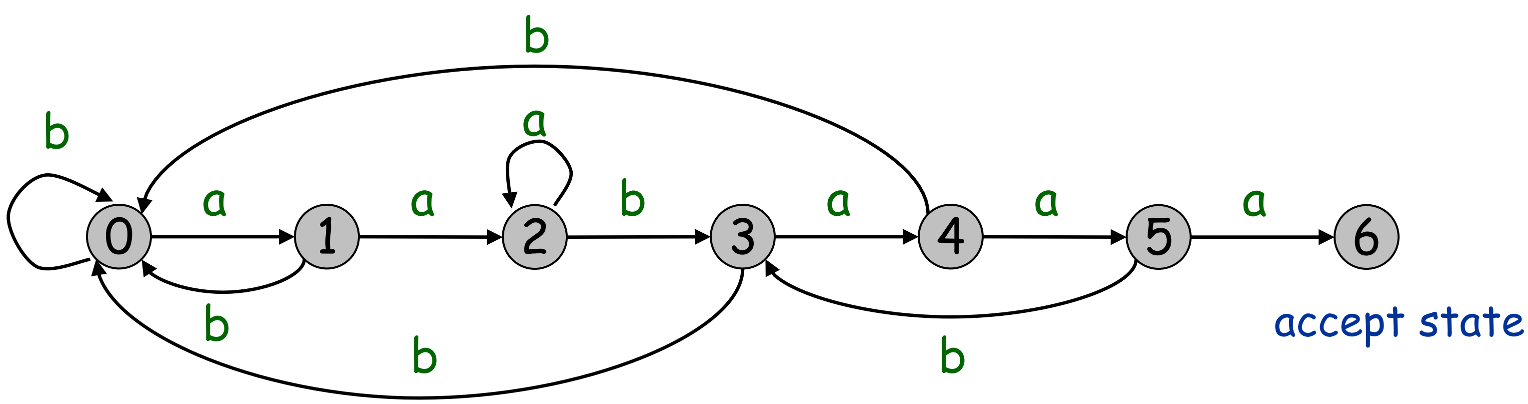
- **Algorithm:** Transition(P, Σ):

```
m ← len(P)
X ← 0
Initialize δ(0,a) for each a ∈ Σ
for j ← 1 to m-1
        for each character a ∈ Σ
                if P[j+1] = a then  // char match
                        δ(j,a) ← j + 1
                else                // char mismatch
                        δ(j,a) ← δ(X,a)
                X ← δ(X,P[j+1])
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

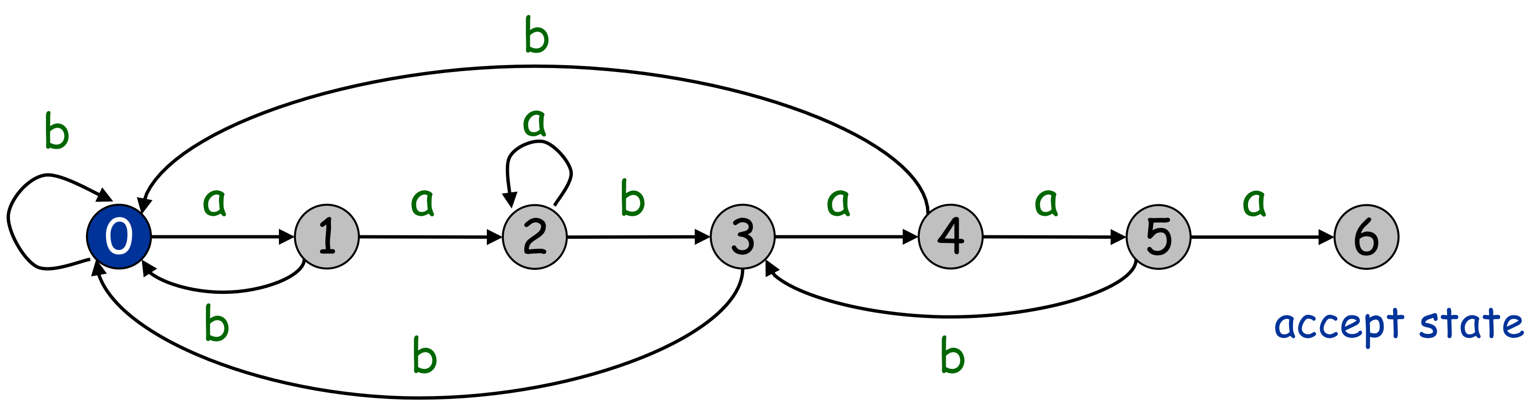


Finite State Automata (FSA)

- 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

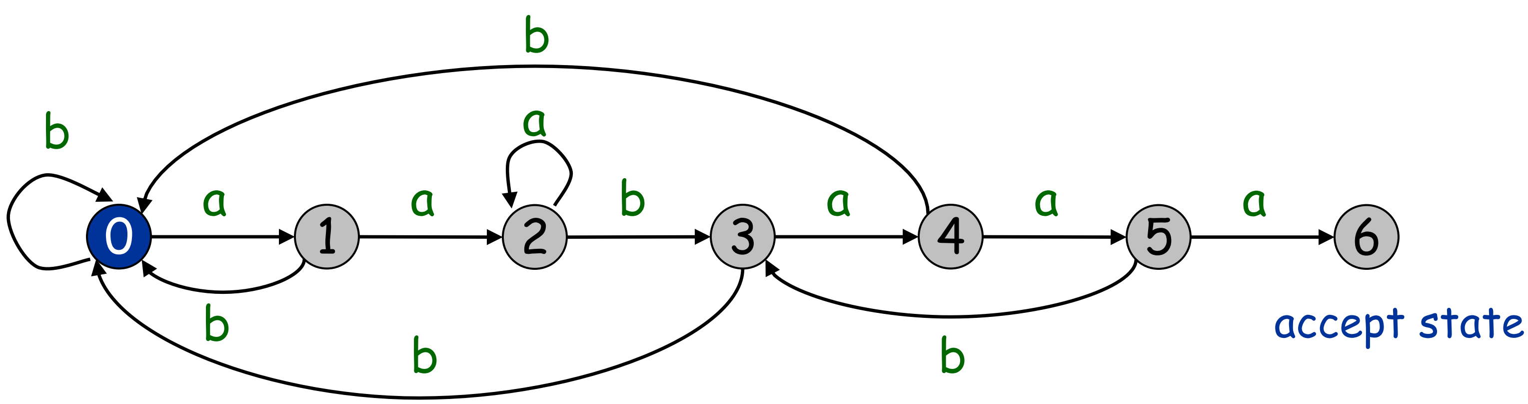


Finite State Automata (FSA)

- 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

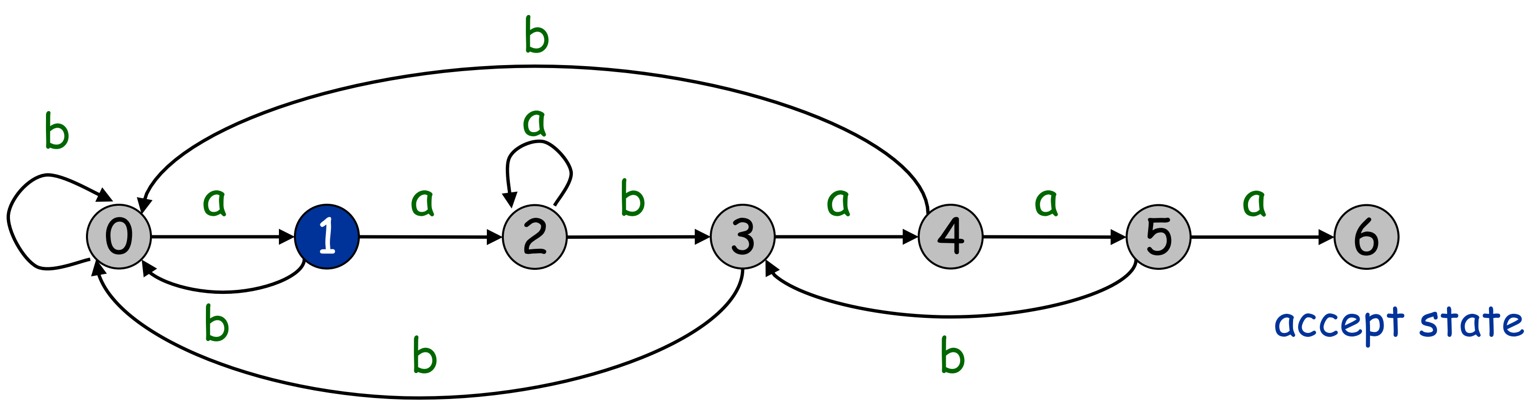


Finite State Automata (FSA)

- 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

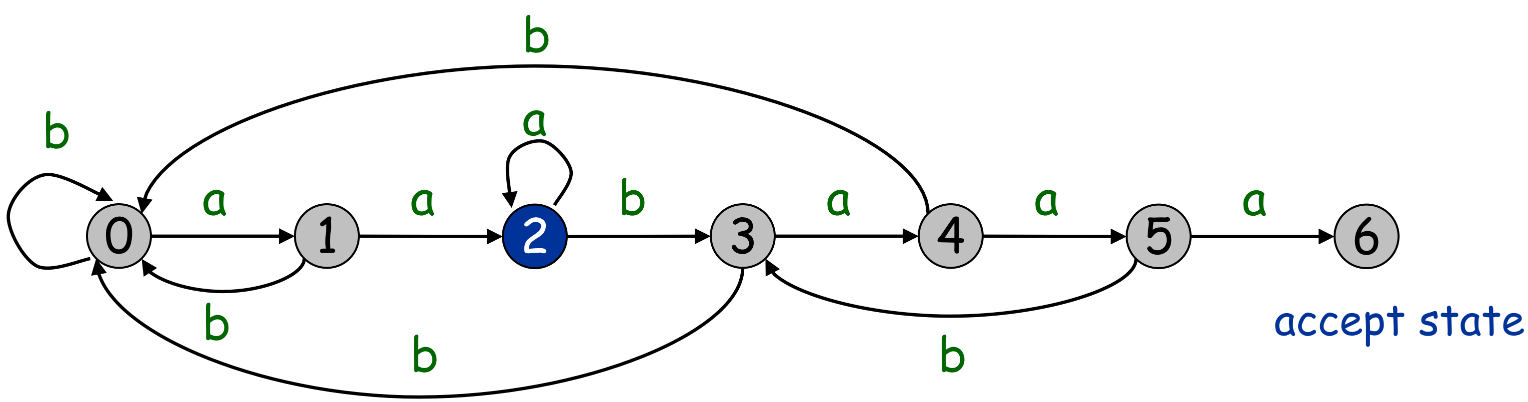


Finite State Automata (FSA)

- 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

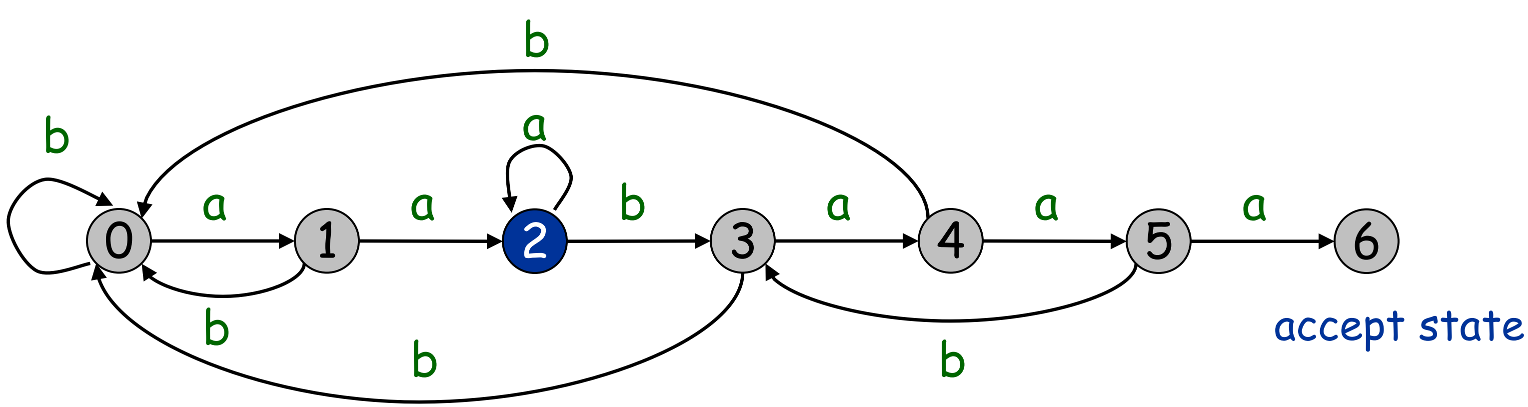


Finite State Automata (FSA)

- 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

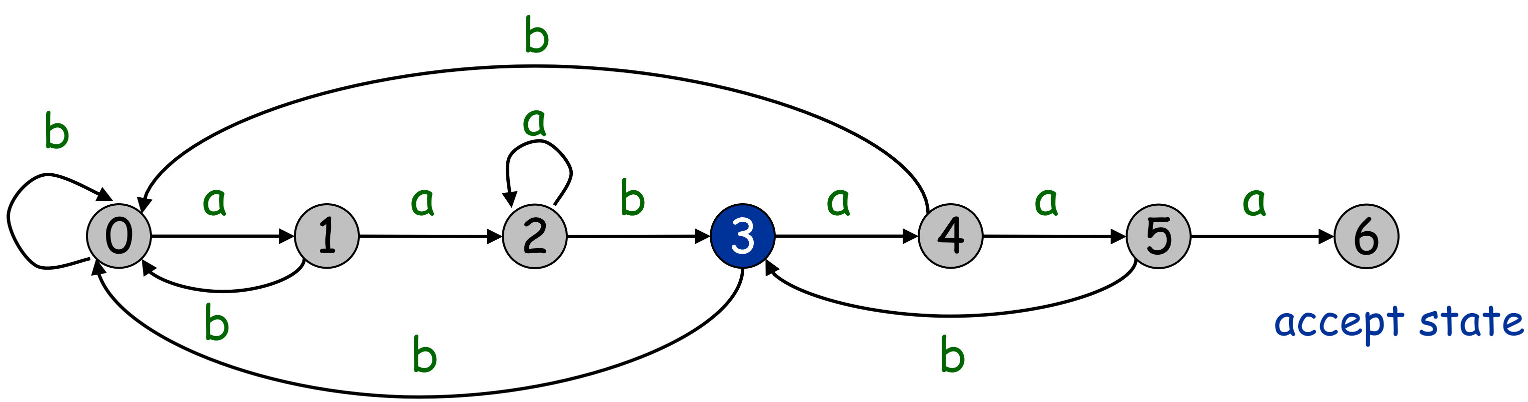


Finite State Automata (FSA)

- 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

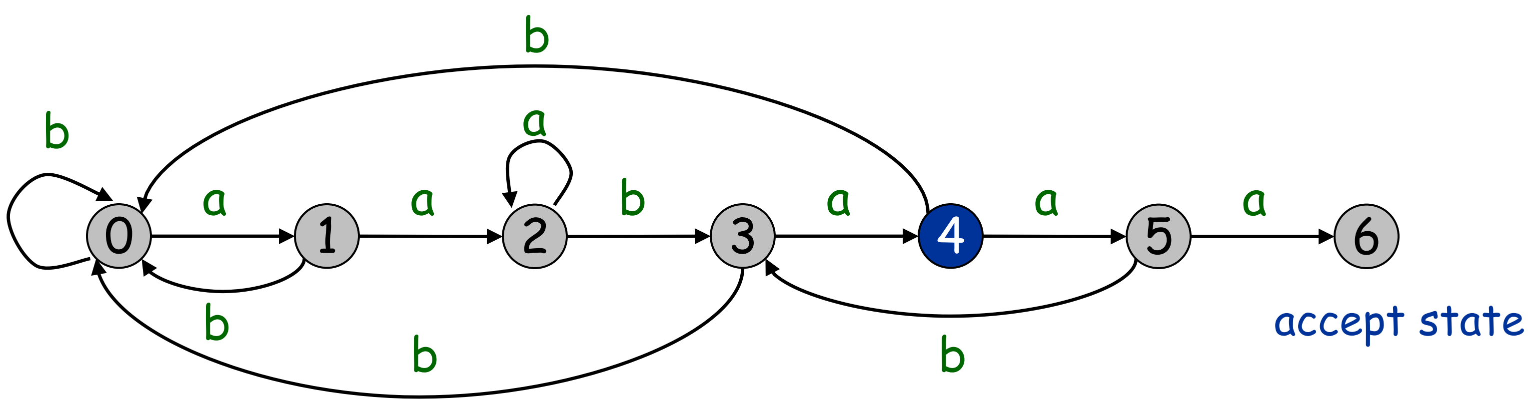


Finite State Automata (FSA)

- 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

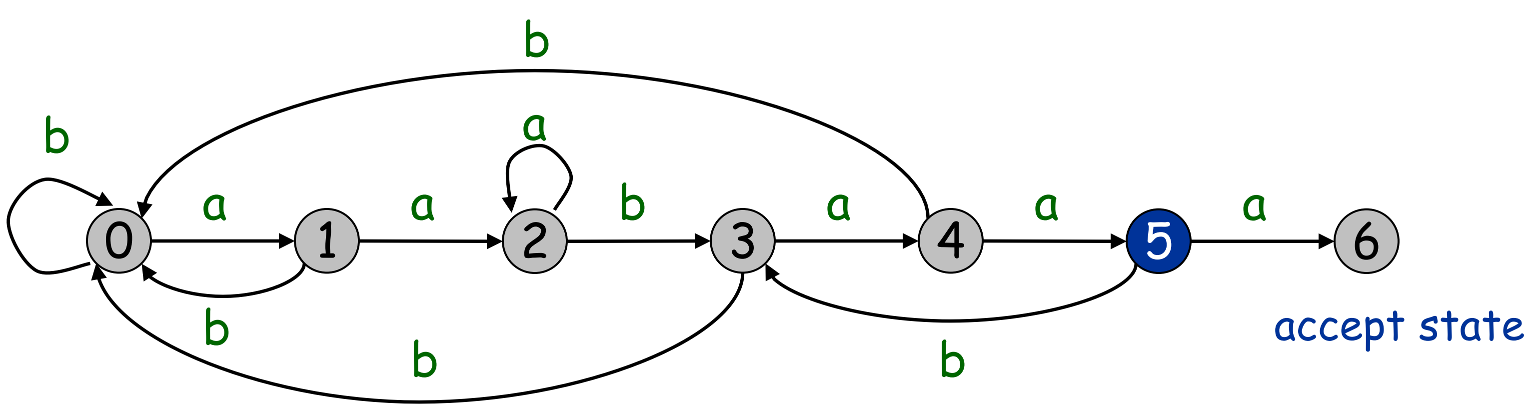


Finite State Automata (FSA)

- 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

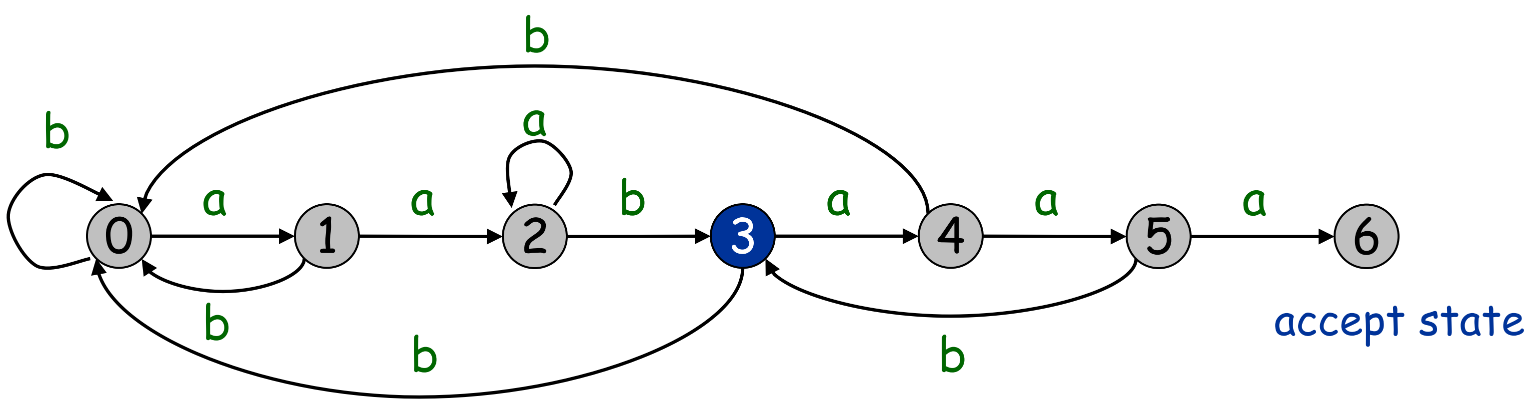


Finite State Automata (FSA)

- 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

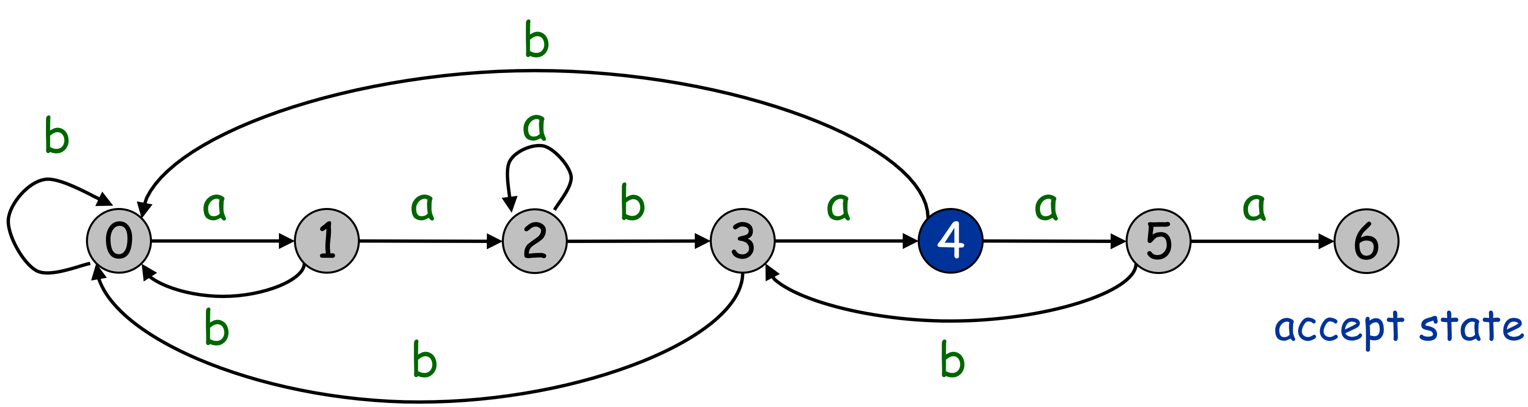


Finite State Automata (FSA)

- 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

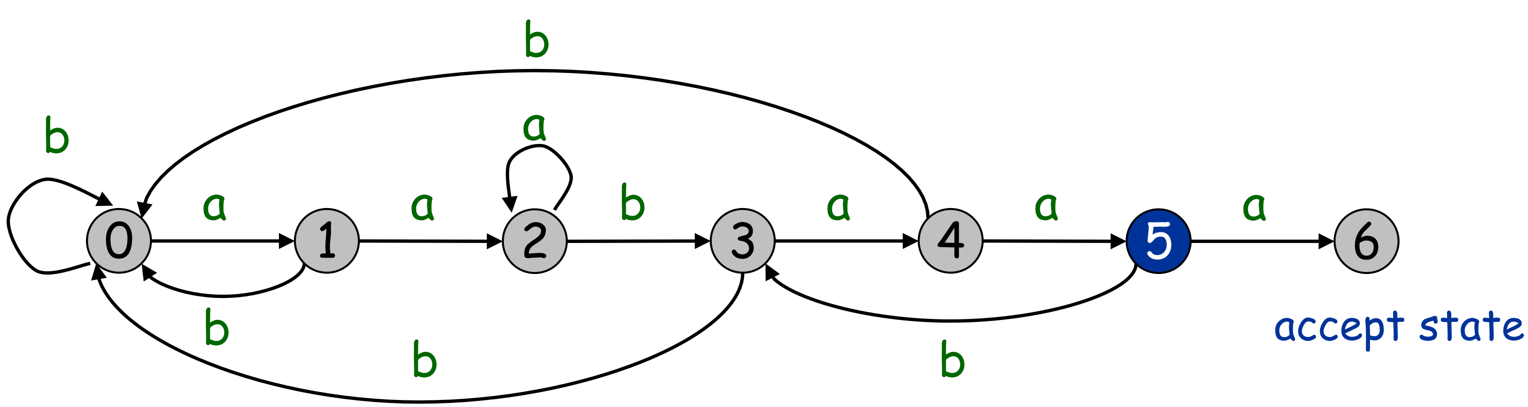


Finite State Automata (FSA)

- 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

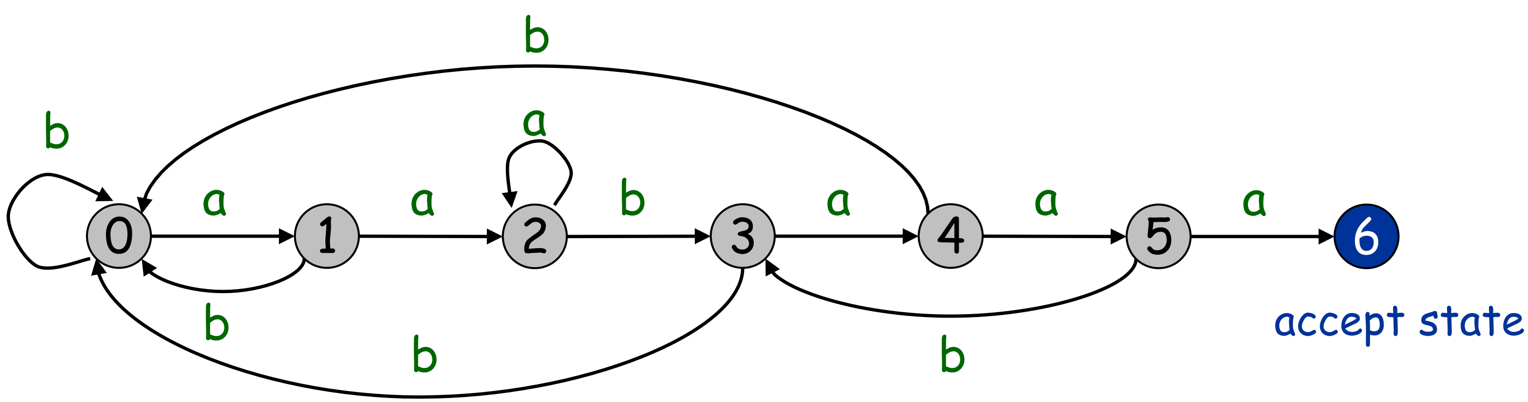


Finite State Automata (FSA)

- 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

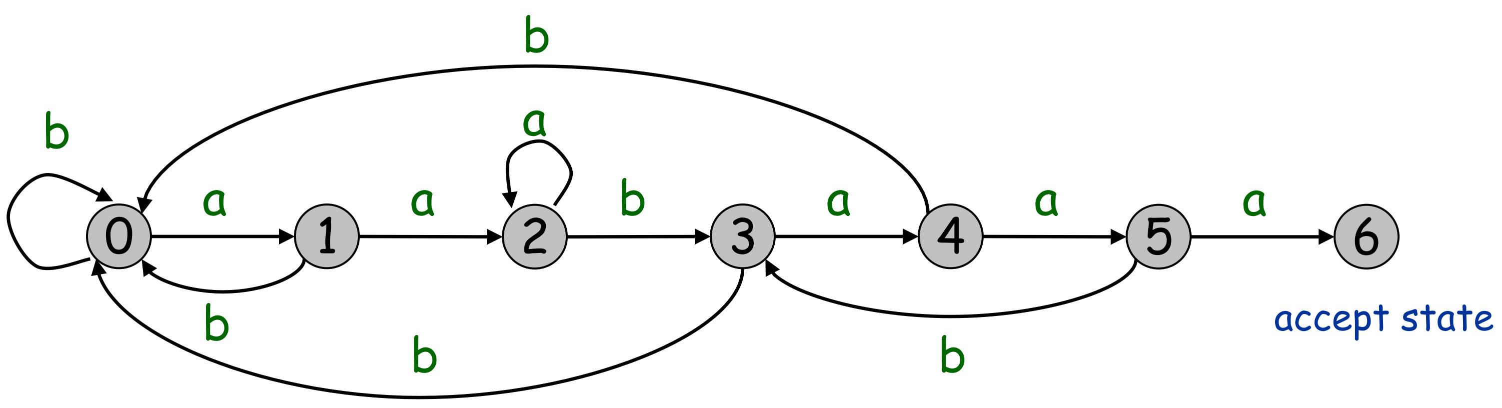


Finite State Automata (FSA)

- 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 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 ← len(T), m ← len(P)
2. δ ← Transition(P, Σ)
3. q ← 0 // q is the state of the FSA.
4. for i ← 1 to n
5.        q ← δ(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

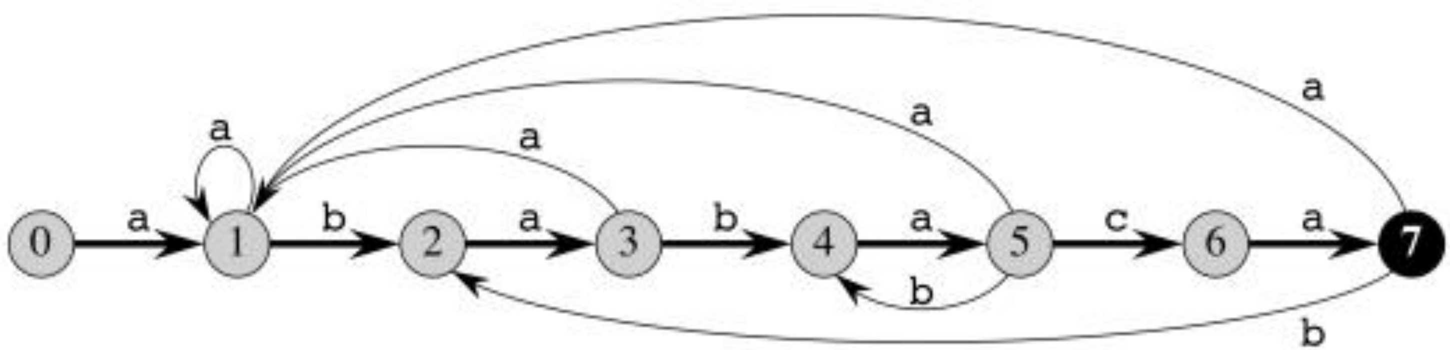
- **String**: “HelloCS203”
- **Substring**: a substring of s string S is a string S’ that occurs in S, e.g., P[1,...,3] = “ell”
- **Prefix (P[0,...]):** a prefix of a string S is a substring of S that occurs at the beginning of S, e.g., P[0,...,0] = “H” (note that P[0]=‘H’), P[0,...,1] = “He”, P[0,...,4] = “Hello”, we denote prefix as: **P[0,...]**
- **Suffix**: a suffix of a string S is a substring of S that occurs at the end of S, e.g., P[9,...,9]=“3”, P[7,...,9]=“203”, P[5,...,9] = “CS203”, we denote suffix as: **P[...,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	

- State transition graph



Finite State Automata

- P ="ababaca" and T ="abababacaba"

i	0	1	2	3	4	5	6	7	8	9	10
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=5, 'c' was expected, but not found in T[5], FSA transition to state $\delta(5,b)=4$, it means pattern prefix P[0..3] ="abab" has matched the text suffix T[2..5] = "abab"
- After **success**, at i=9, a "b" is seen, $\delta(7,b)=2$,
- thus, P[0..1] = T[8..9]

	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., $P[1,\ldots,k]=T[i-k+1,\ldots,i]$
  - 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 δ(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[...i], denote as k, i.e., $P[1,...,k]=T[i-k+1,...,i]$
- Suppose it is mismatched at "$P[k+1]$", it means:
  - $P[k+1] \,!= T[i+1]$ then,
  - we should find the longest prefix of $P[1,...,k]$ is also a suffix of $T[i-k+1, ..., i]$.
- **Prefix function (next array in general),** given $P[0..m]$, the prefix function π for P is $\pi : \{1, 2 ..., m\} \rightarrow \{0, 1, ..., m-1\}$ such that:

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

Prefix Function

- **Prefix function,** given P, the prefix function π for P is π : {1, 2 ..., m} -> {0, 1, ..., m-1} such that:

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

- Example: P =“abababaca”

i	0	1	2	3	4	5	6
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3	0	1

Prefix Function (next)

- P ="ababaca" and T ="abababacaba"

i	0	1	2	3	4	5	6	7	8	9	10
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		

- After **failure**: at i=5, 'c' was expected, but not found in T[5], then we lookup π[4] = 3

i	0	1	2	3	4	5	6
P[i]	a	b	a	b	a	c	a
π[i]	-1	0	1	2	3	0	1

Compute next array

- **Algorithm: NextArray(P):**

```
m ← len(P)
Let next[0,…,m-1] be a new array
next[0] = -1, k ← 0, j ← 1
while j < m
        if P[j] = P[k]              // char match
                k ← k + 1
                next[j] = k
                j ← j + 1
        else if k = 0             // the first mismatch
                next[j] ← 0, j ← j + 1
        else                        // char mismatch
                k ← next[k-1]
return next
```

KMP algorithm

- **Algorithm:** KMP(T, P):

```
n ← len(T), m ← len(P)
next ← NextArray(P)
i ← 0, j ← 0
While (i < n && j < m)
        if (P[j] = T[i] && j = m-1)
                pattern occurs with shift i – m
        else if (P[j] = T[i])
                i ← i + 1, j ← j + 1
        else
                j ← next[j-1]
                if j = 0
                        i ← i + 1
```

Analysis of KMP

- **Algorithm:** KMP(T, P):

```
Cost of Line 1:
Cost of Line 2:
Cost of Line 3:
Cost of Line 4:
…
Cost of Line 10:

Overall Cost:
```

Our Roadmap

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

Thank You!