

Natalie Parde  
UIC CS 421

# Text Preprocessing and Edit Distance



# What is text preprocessing?

- The process of automatically organizing, normalizing, and manipulating text so it can be handled more easily by downstream language processing tasks.

“Have some wine,” the March Hare said in an encouraging tone.

Alice looked all round the table, but there was nothing on it but tea. “I don't see any wine,” she remarked.

“There isn't any,” said the March Hare.

- Lewis Carroll, *Alice's Adventures in Wonderland*



have some wine [PERSON 1] said in an encouraging tone

[PERSON 2] looked all round the table but there was nothing on it but tea

i don't see any wine she remarked

there isn't any said [PERSON 1]

- Lewis Carroll, *Alice's Adventures in Wonderland*

# Text preprocessing steps can (and should!) vary depending on your needs.

capitalization

written numbers vs. digits

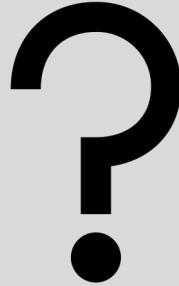
proper nouns

punctuation

British vs. American spellings (for English text)

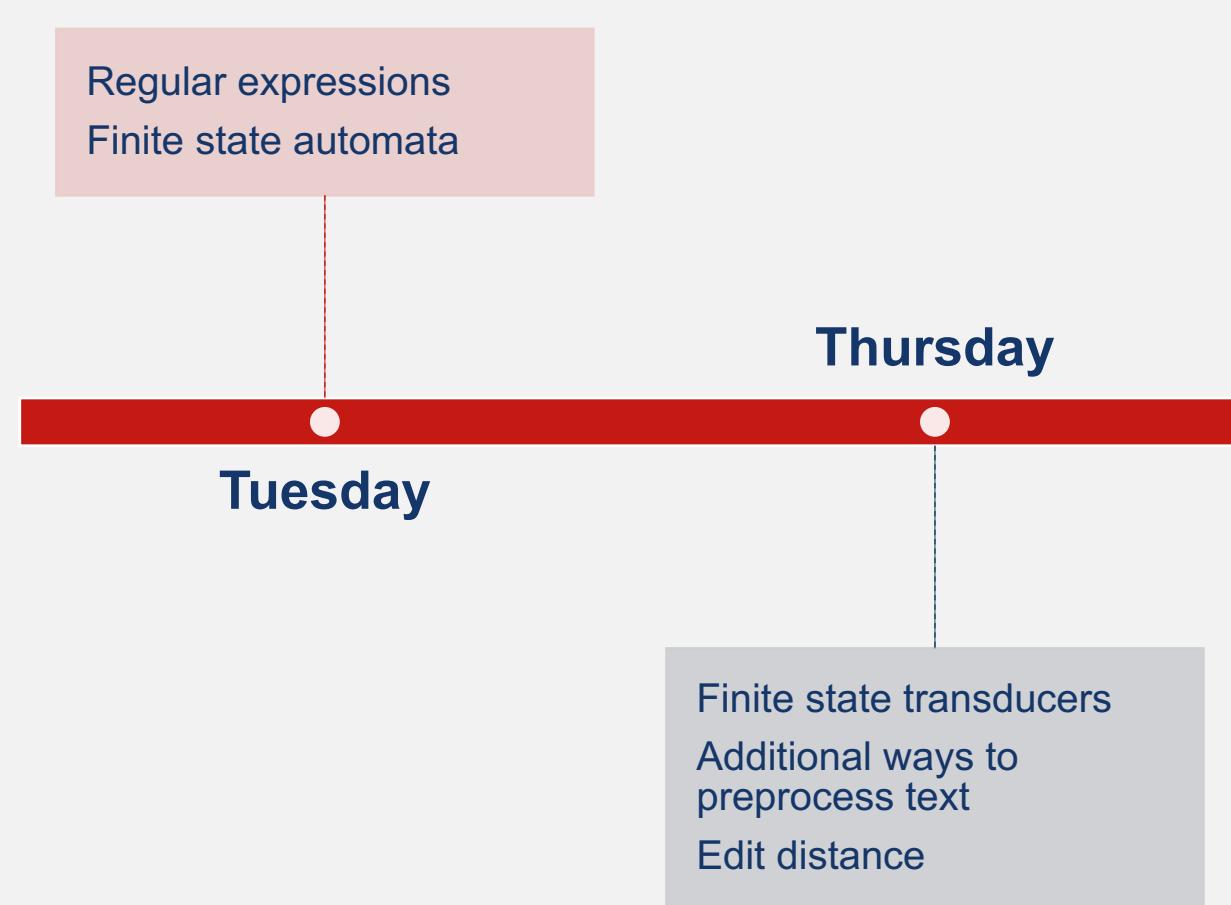


Important



Not Important

# This Week's Topics



# This Week's Topics



Regular expressions  
Finite state automata

Tuesday

Thursday

Finite state transducers  
Additional ways to preprocess text  
Edit distance



## One way to preprocess text is by using regular expressions.

- Regular expressions: A formal language for specifying text strings.
- How can we search for any of these?
  - Donut
  - donut
  - Doughnut
  - doughnut
  - Donuts
  - doughnuts

# Regular Expression Terminology

---

**Regex:** Common abbreviation for regular expression

---

**Disjunction:** Logical OR

---

**Range:** All characters in a sequence from  $c_1-c_2$

---

**Negation:** Logical NOT

---

**Scope:** Indicates to which characters the regex applies

---

**Anchor:** Matches the beginning or end of a string

## Disjunctions and Ranges

- Disjunction: Letters inside square brackets [az]
- Range: Hyphen between the first and last characters in the range [a-z]

Pattern	Matches	Example
[dD]onut	donut, Donut	This morning would be better with a <b>donut</b> .
[0123456789]	Any digit	This morning would be better with <b>5</b> donuts.
[A-Z]	An uppercase letter	<b>Donuts</b> are an excellent way to start the day.
[0-9]	Any digit	I just ate <b>5</b> donuts.

Pattern	Matches	Example
[^dD]onut	Any letter except “d” or “D” before the sequence “onut”	This morning would be better with a <b>coconut</b> .
[^A-Z]	Not an uppercase letter	<b>D</b> onuts are an excellent way to start the day.
[^^]	Not a caret	<b>W</b> hat is your favorite kind of donut?
D^o	The pattern “D^o”	Is <b>D^o</b> nut a good name for my donut shop?

## Negation in Disjunction

- Negation: A caret (^) at the beginning of a disjunction [^az]
  - The caret must be at the beginning of the disjunction to negate it

# More about disjunctions!

- The pipe | indicates the union (logical OR) of two regular expressions
- $a|b|c$  is equivalent to  $[abc]$

Pattern	Matches	Example
d D	“d” or “D”	This morning would be better with a <b>donut</b> .

# Special Characters in Regular Expressions

- **\***: Means that there must be 0 or more occurrences of the preceding expression
- **.**: A wildcard that can mean any character
- **+**: Means that there must be 1 or more occurrences of the preceding expression
- **?**: Means that there must be 0 or 1 occurrences of the preceding expression
- **{m}**: Means that there must be  $m$  instances of the preceding expression
- **{m,n}**: Means that there must be between  $m$  and  $n$  instances of the preceding expression
- **(abc)**: Means that the operation should be applied to the specified sequence
- **\**: Used to escape special characters so that the regular expression searches for their literal form

# Regular Expressions: Special Characters

Pattern	Matches	Example
donuts*	“donut” or “donuts” or “donutss” or “donutsss”....	This morning I had many <b>donuts</b> .
.onut	Any character followed by “onut”	Can I have a <b>coconut donut</b> ?
donuts+	“donuts” or “donutss” or “donutsss”....	Do you want one donut or two <b>donuts</b> ?
donuts?	“donut” or “donuts”	Do you want one <b>donut</b> or two <b>donuts</b> ?
donuts{1}	“donuts”	Do you want one donut or two <b>donuts</b> ?
donuts{0,1}	“donut” or “donuts”	Do you want one <b>donut</b> or two <b>donuts</b> ?
.o(nut)?	Any character followed by “o” or “onut”	Can I have a <b>disco donut</b> ?
donut\.	“donut.”	Please give me one <b>donut</b> .

# Regular Expressions: Anchors

- Indicate that a pattern should be matched only at the beginning or end of a word

Pattern	Matches	Example
<code>^Donuts</code>	“Donuts” only when it is at the beginning of a string	<b>Donuts</b> are an excellent way to start the day.
<code>donuts\$</code>	“donuts” only when it is at the end of the string	I just ate 5 <b>donuts</b>



# Shorthand Character Classes

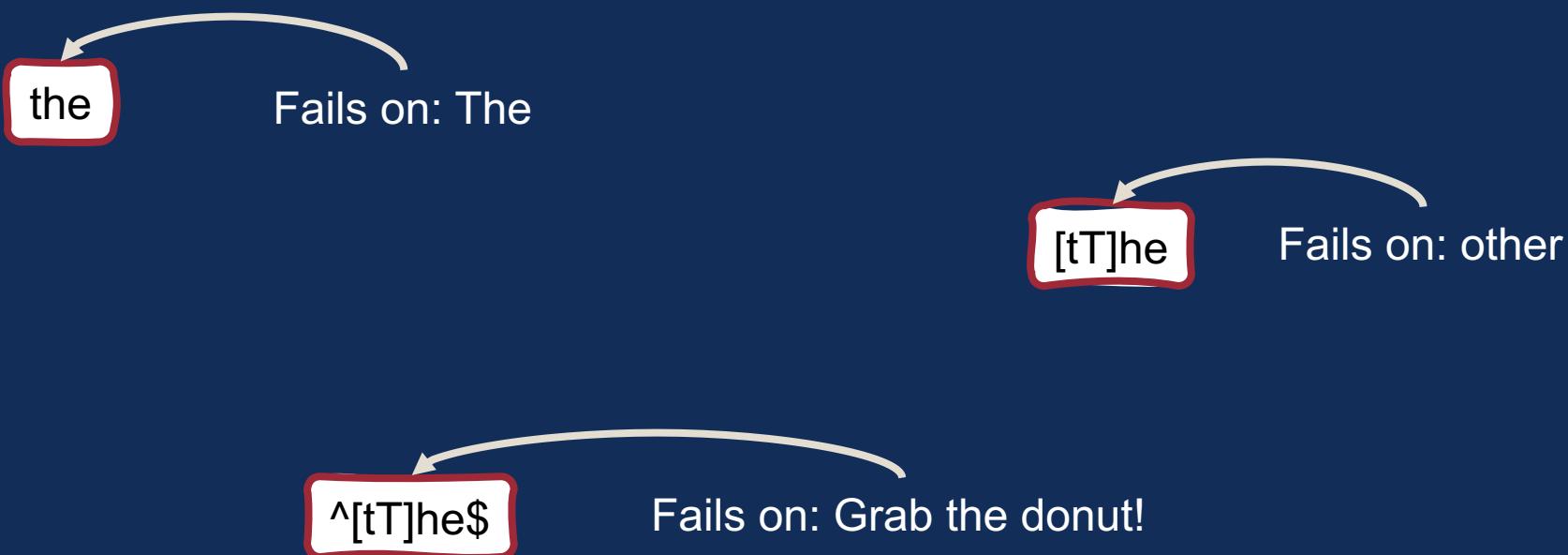
- Many programming languages support several predefined, shorthand character classes for regular expressions:
  - **\d**: Any digit
  - **\D**: Any non-digit
  - **\s**: Any whitespace
  - **\S**: Any non-whitespace
  - **\w**: Any alphanumeric character
  - **\W**: Any non-alphanumeric character
  - **\b**: Empty string or certain characters at word boundaries
- Can be very helpful, but make sure to understand what patterns they're matching before using them
  - May be defined differently from what you need!

# Shorthand Character Classes

- Can be used inside or outside disjunctions
  - Can be negated
  - Note that different programming languages:
    - May have different sets of shorthand character classes
    - May implement the same shorthands differently



# Regular expressions are often trickier than they initially seem....



# Errors

- When we match strings that we didn't want to match, we have **false positive** (Type I) errors
- When we don't match strings that we wanted to match, we have **false negative** (Type II) errors

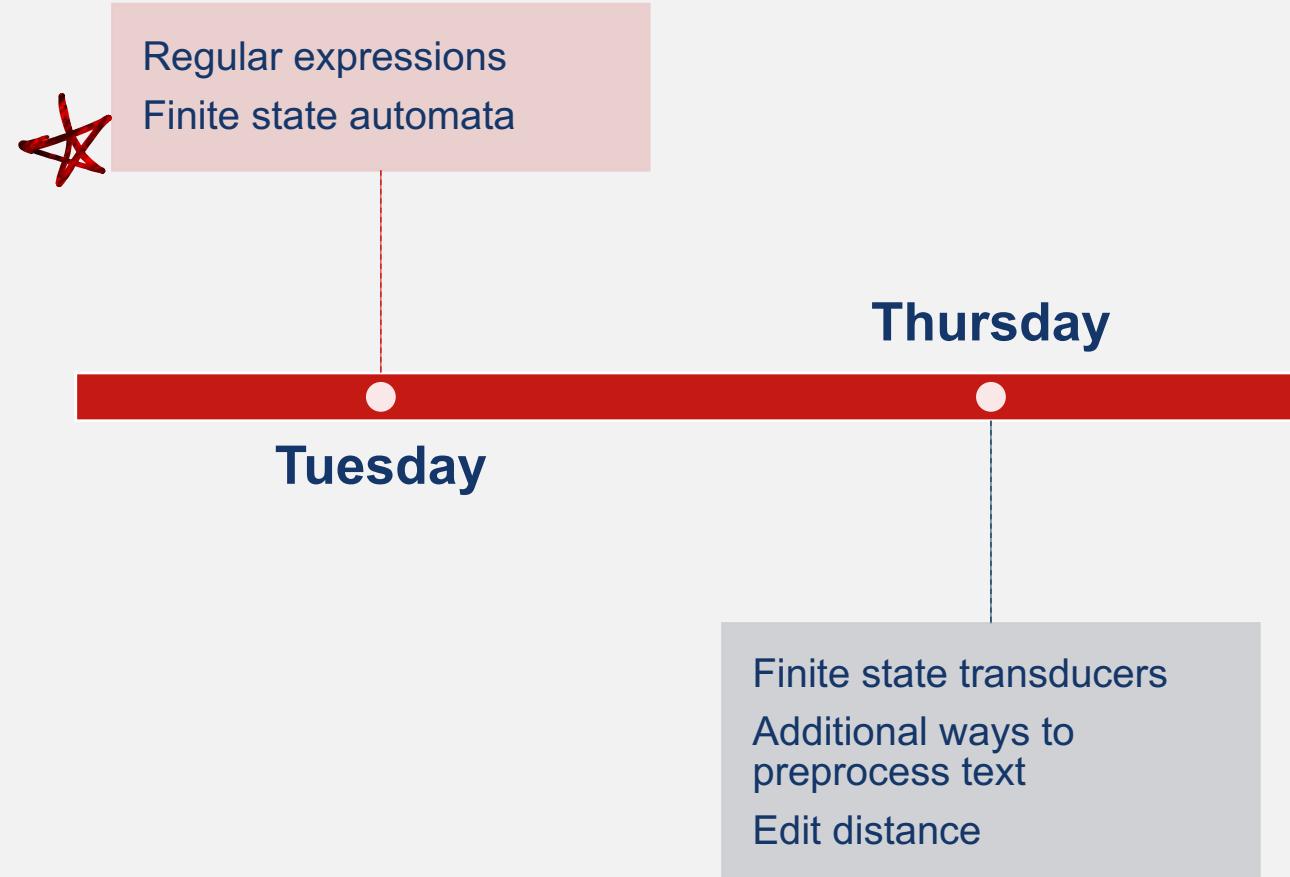
# Errors

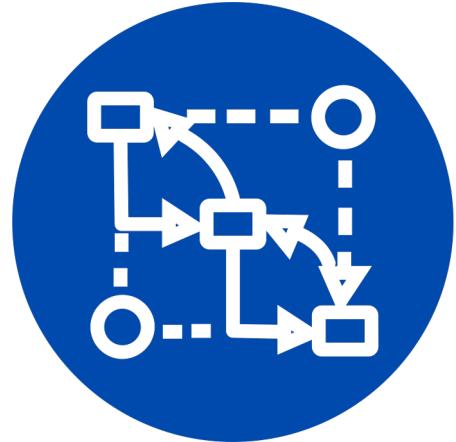
- This is a recurring theme in NLP!
- Regardless of what application we're developing, we often try to do two things to improve performance:
  - Increase **accuracy** or **precision** (minimizing false positives)
  - Increase **coverage** or **recall** (minimizing false negatives)

# Regular Expressions: Takeaway Points

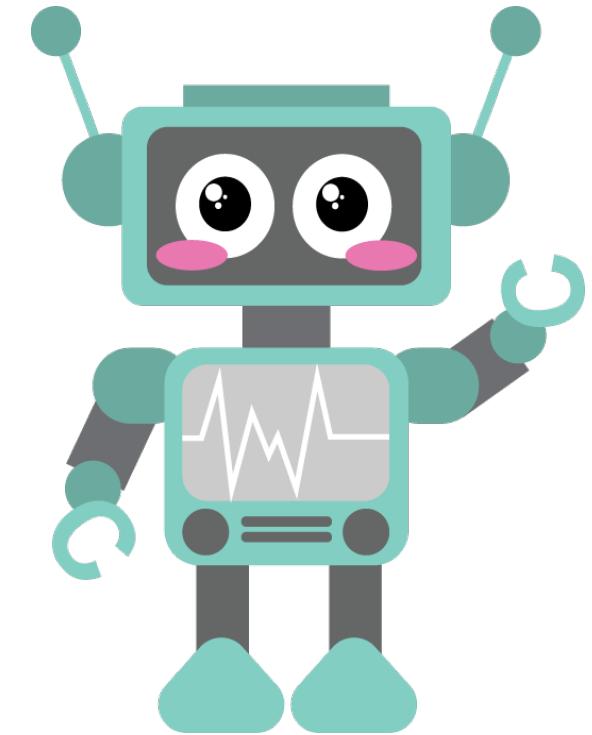
- Regular expressions are powerful and have many different uses
  - Text tokenization
  - Text normalization
  - Feature extraction
- They allow us to search for specific strings of text using disjunctions, negations, and a variety of special characters

# This Week's Topics





**Regular expressions  
can be matched using  
finite state automata.**



# What are finite state automata?

- Computational models of regular languages
- Used to model regular expressions, but also used in other applications that function by transitioning between finite states
  - Dialogue systems
  - Morphological parsing
- Terminology:
  - Singular: Finite State Automaton (FSA)
  - Plural: Finite State Automata (FSAs)

# Key Components

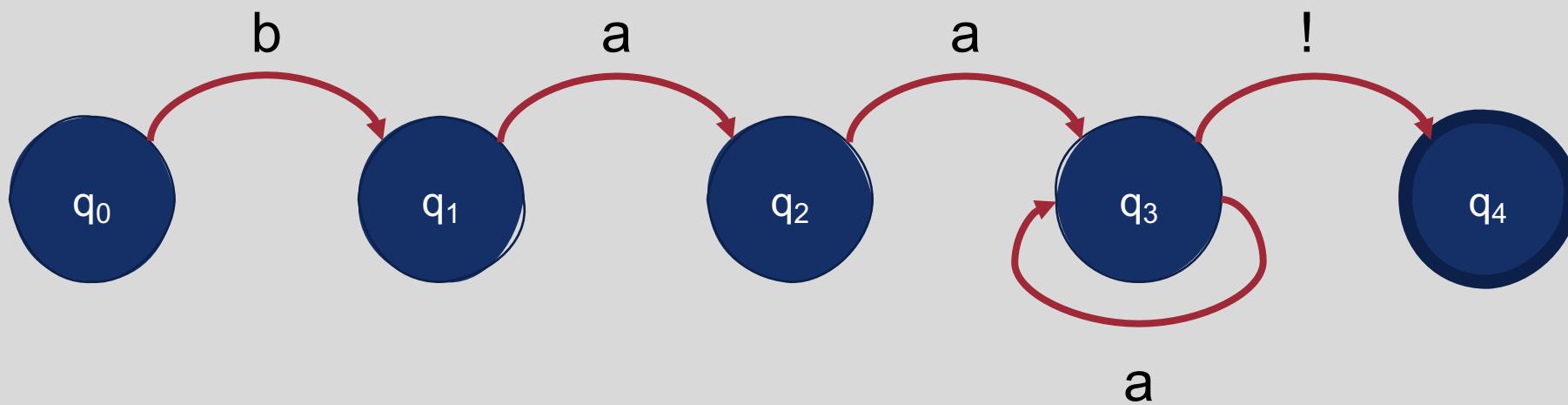
- Finite set of states
  - Start state
  - Final state
- Set of transitions from one state to another

# How do FSAs work?

- For a given sequence of items (characters, words, etc.) to match, **begin in the start state**
- **If the next item** in the sequence **matches a state that can be transitioned to** from the current state, **go to that state**
- **Repeat**
  - If no transitions are possible, **stop**
  - If the state you stopped in is a final state, **accept the sequence**

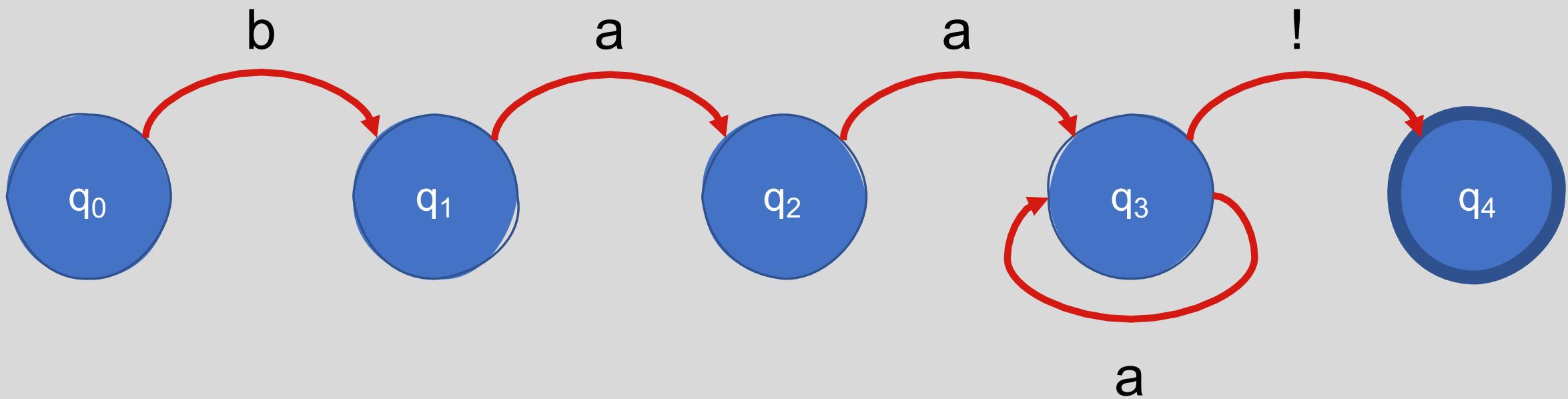
# FSAs are often represented graphically.

- Nodes = states
- Arcs = transitions

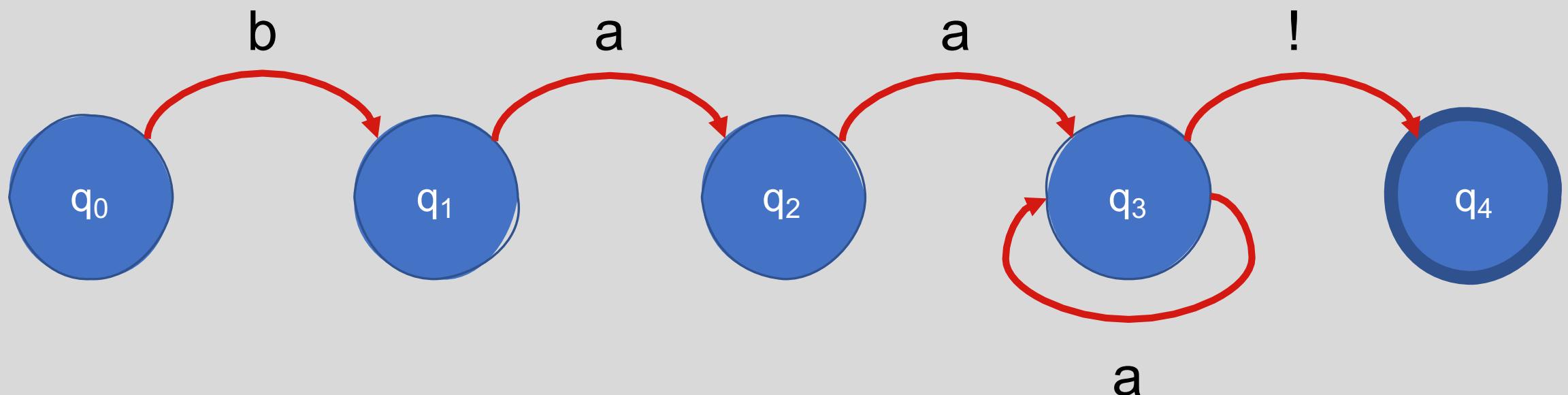


# What do we know about this FSA?

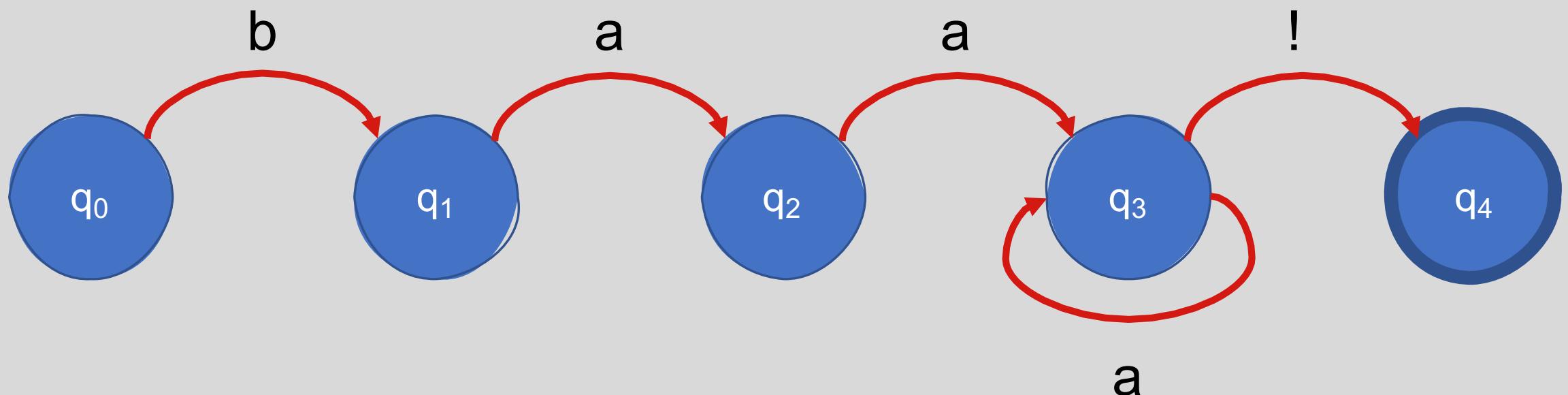
- Five states
  - $q_0$  is the start state
  - $q_4$  is the final (accept) state
- Five transitions
- Alphabet = {a, b, !}



# Regex that this FSA matches: $baa^+$ !

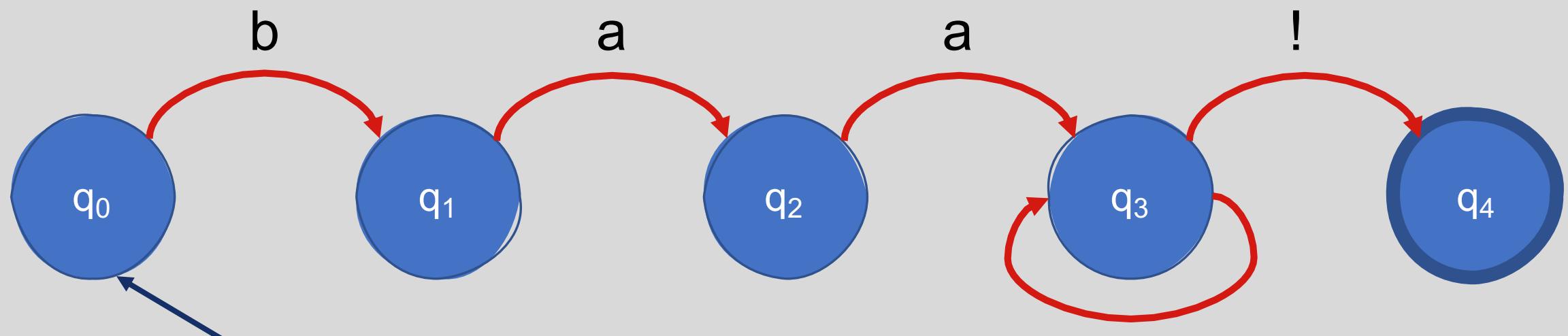


# Regex that this FSA matches: **baa<sup>+</sup>!**



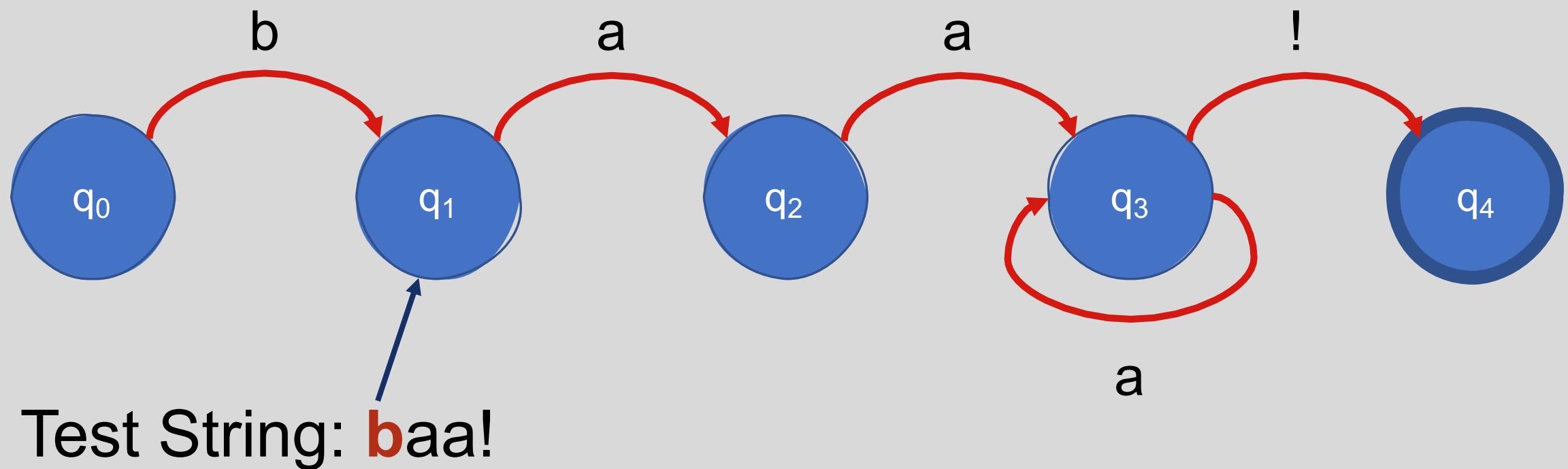
Test String: **baa!**

# Regex that this FSA matches: **baa<sup>+</sup>!**

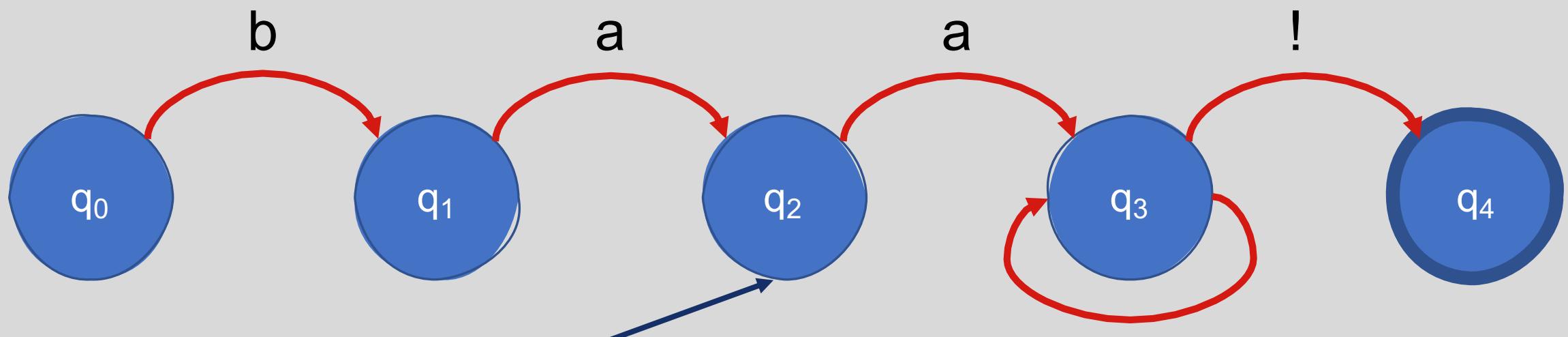


Test String: **baa!**

# Regex that this FSA matches: **baa<sup>+</sup>!**

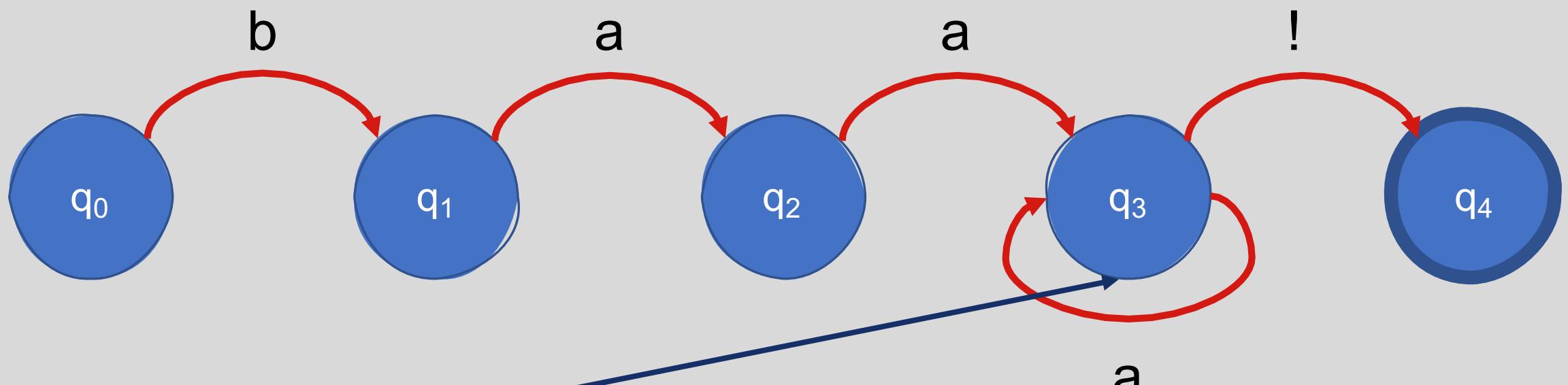


# Regex that this FSA matches: **baa<sup>+</sup>!**



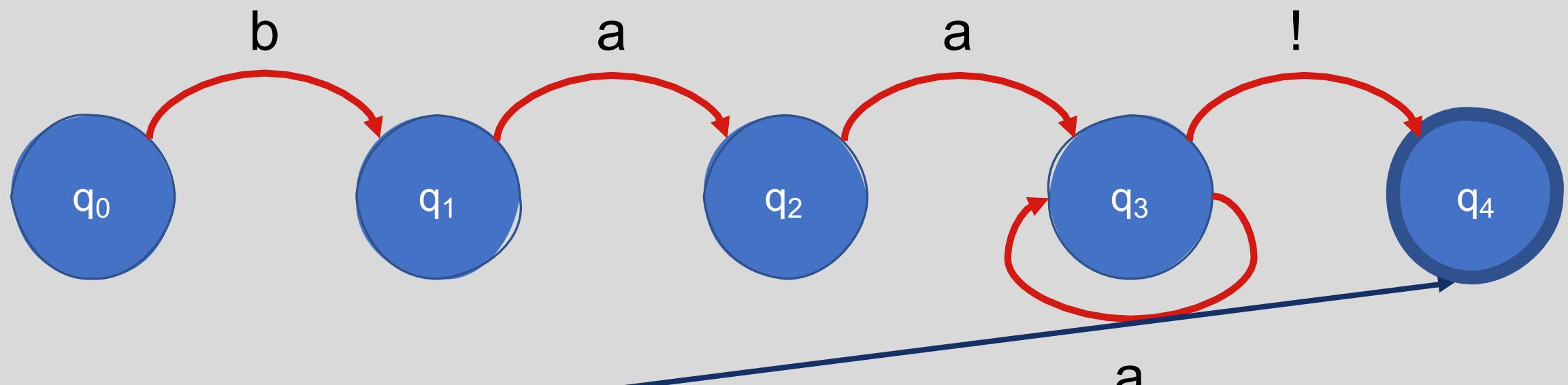
Test String: **baa!**

# Regex that this FSA matches: **baa<sup>+</sup>!**



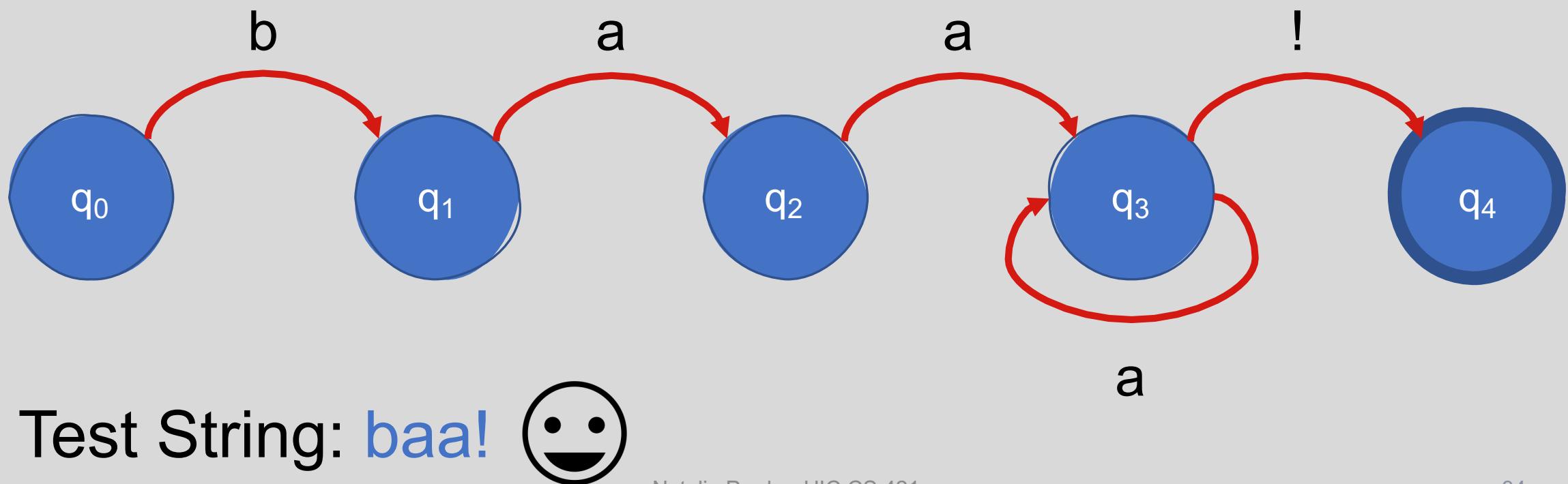
Test String: **baa!**

# Regex that this FSA matches: **baa<sup>+</sup>!**

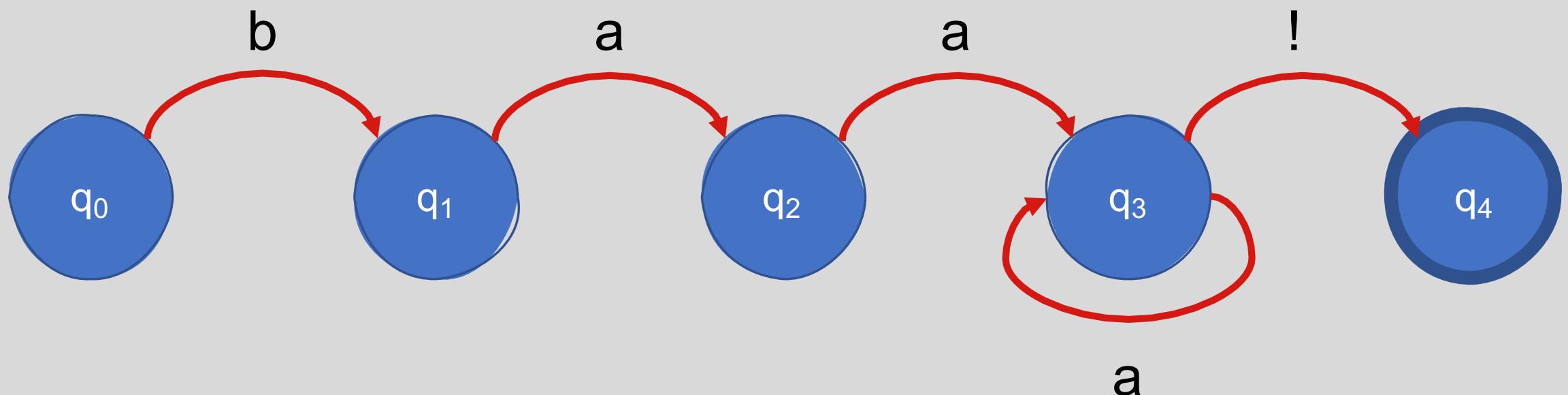


Test String: **baa!**

# Regex that this FSA matches: $baa^+$ !

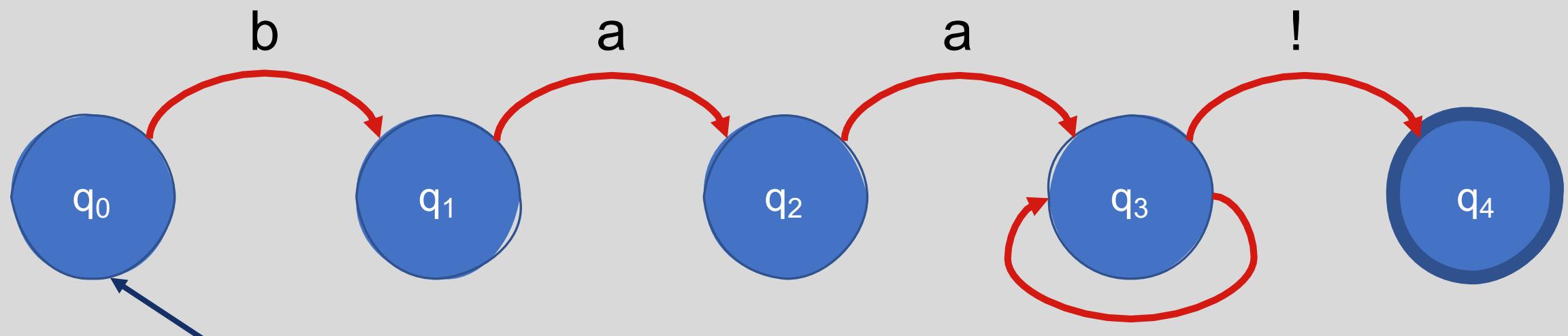


# Regex that this FSA matches: $baa^+$ !



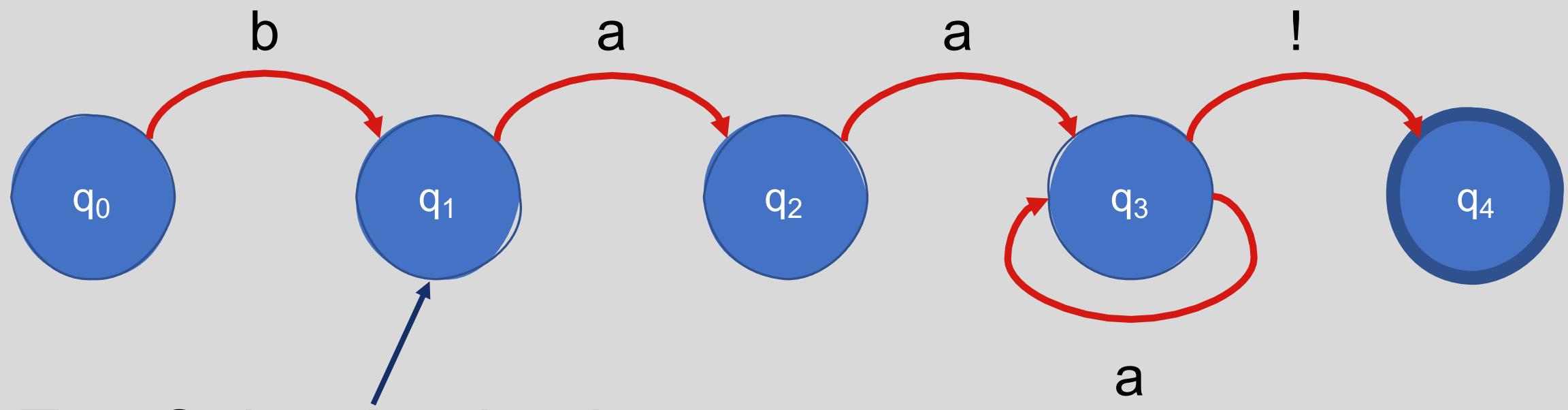
Test String: baabaa!

# Regex that this FSA matches: $baa^+$ !



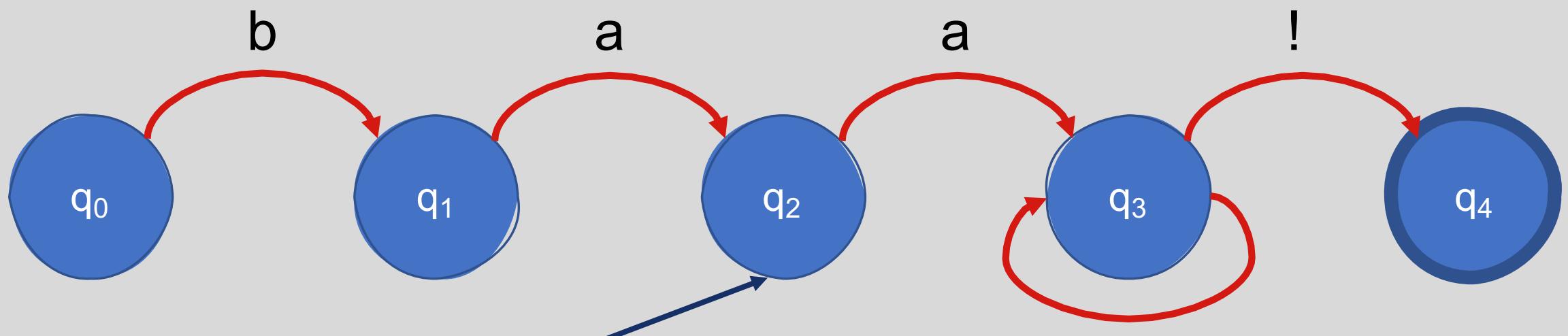
Test String: baabaa!

# Regex that this FSA matches: **baa<sup>+</sup>!**



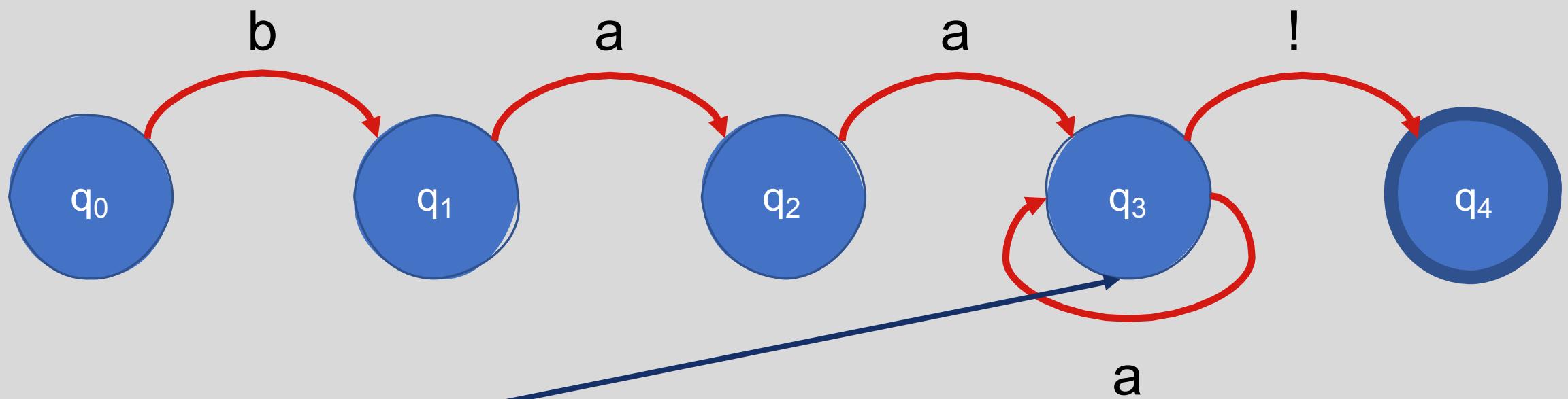
Test String: **baabaa!**

# Regex that this FSA matches: **baa<sup>+</sup>!**



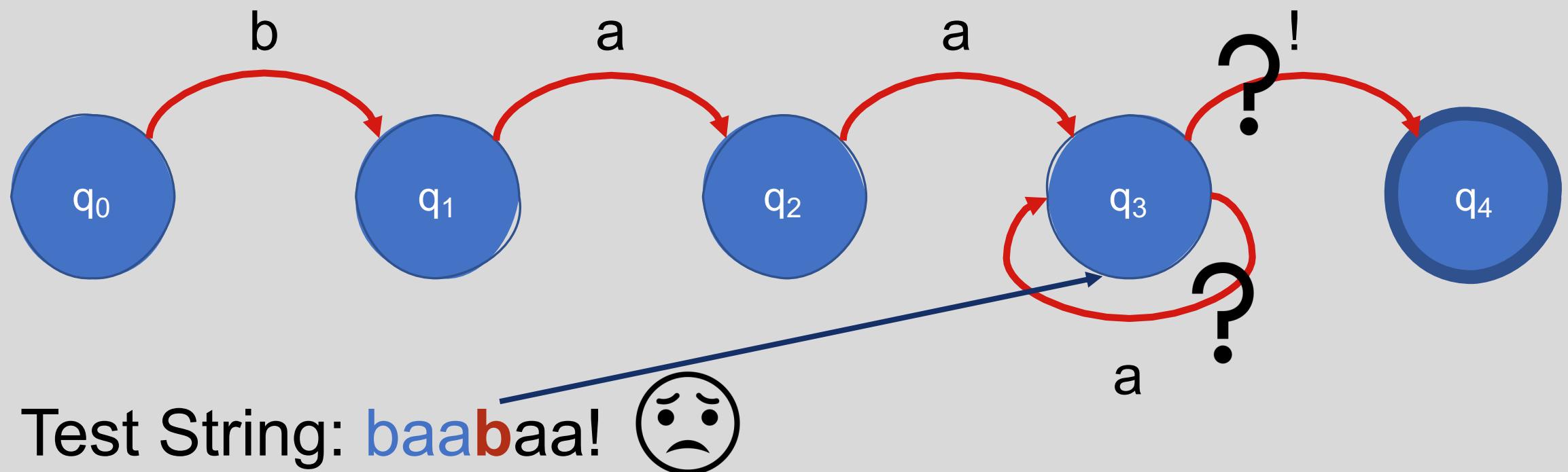
Test String: **baabaa!**

# Regex that this FSA matches: **baa<sup>+</sup>!**

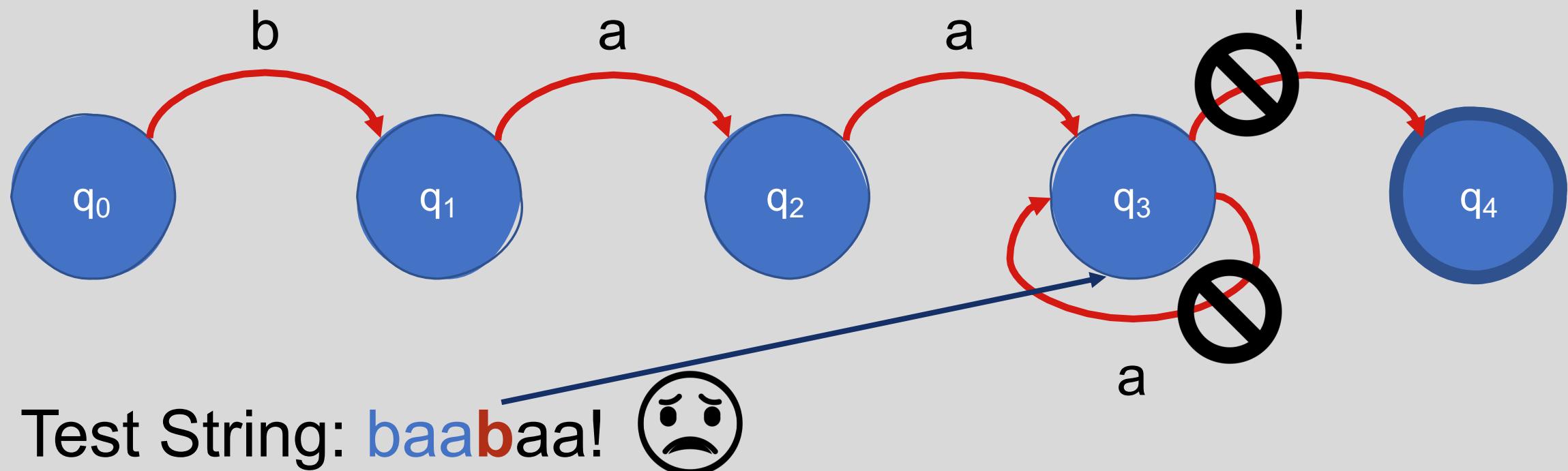


Test String: ba**a**baa!

# Regex that this FSA matches: $baa^+$ !



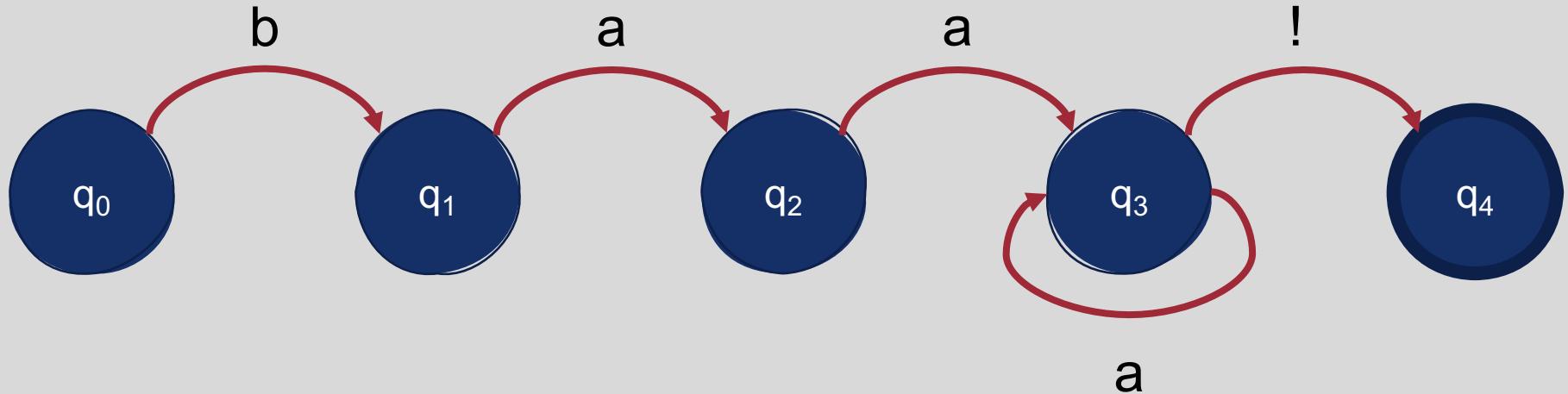
# Regex that this FSA matches: $baa^+$ !



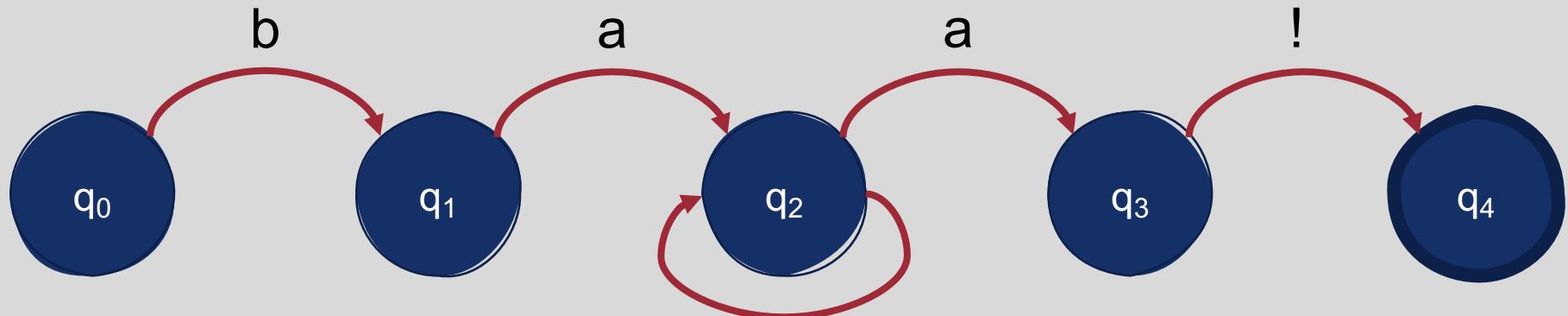
Test String: **baabaa!**

# Note: More than one FSA can correspond to the same test string in a regular language!

Test String:  
baaa!



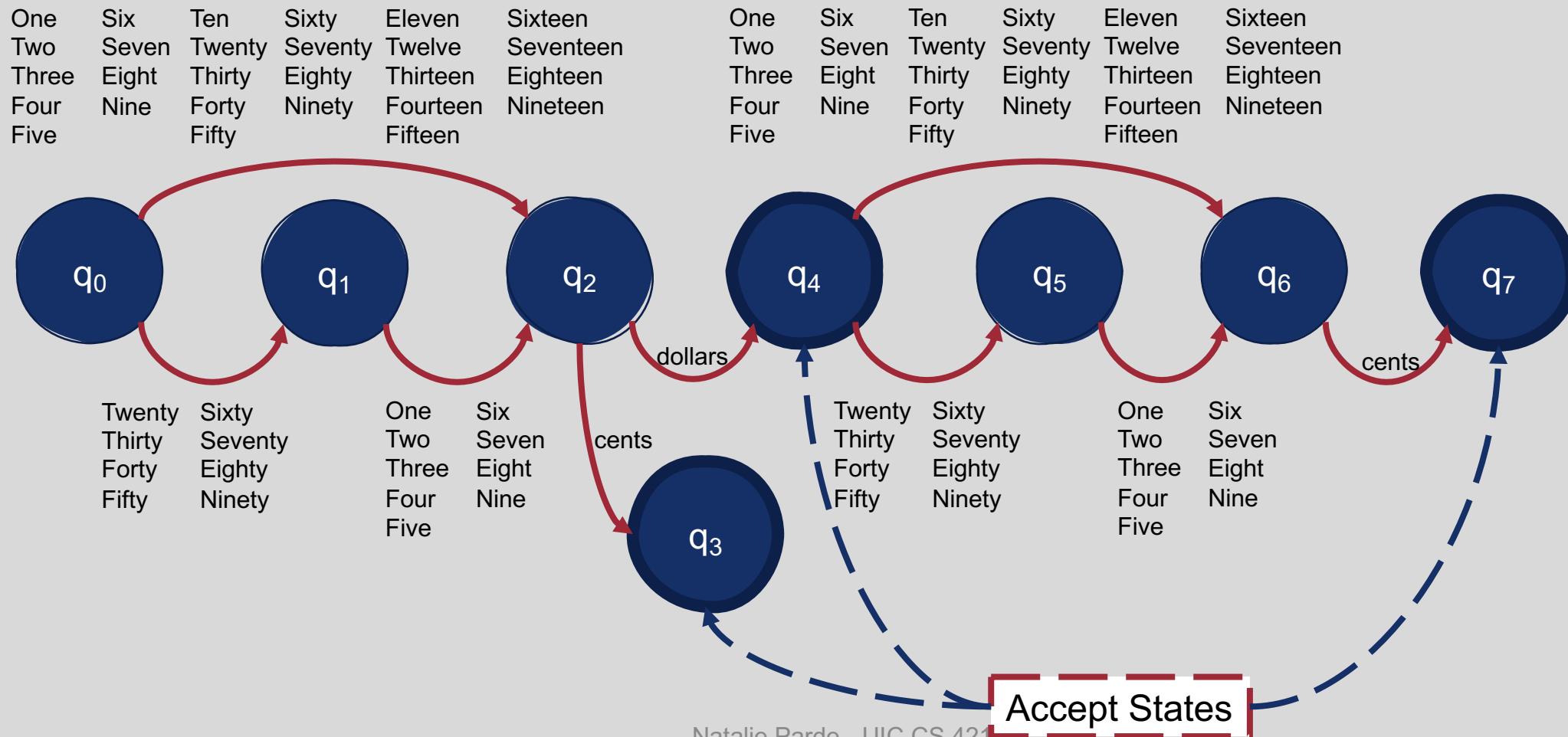
Test String:  
baaa!



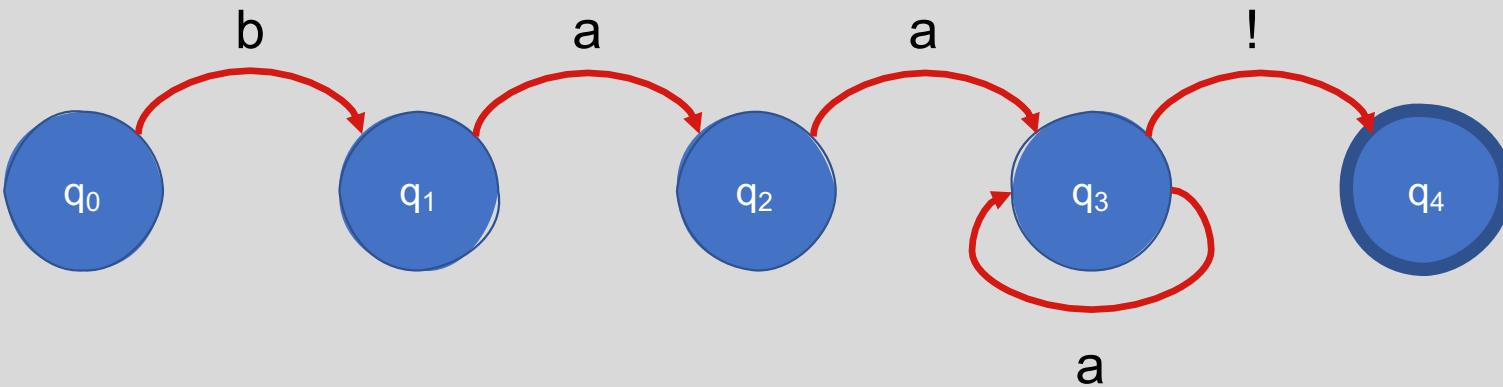
# Formal Definition

- A finite state automaton can be specified by enumerating the following properties:
  - The set of states,  $Q$
  - A finite alphabet,  $\Sigma$
  - A start state,  $q_0$
  - A set of accept/final states,  $F \subseteq Q$
  - A transition function or transition matrix between states,  $\delta(q,i)$
- $\delta(q,i)$ : Given a state  $q \in Q$  and input  $i \in \Sigma$ ,  **$\delta(q,i)$  returns a new state  $q' \in Q$ .**

# Example: FSA for Dollar Amounts

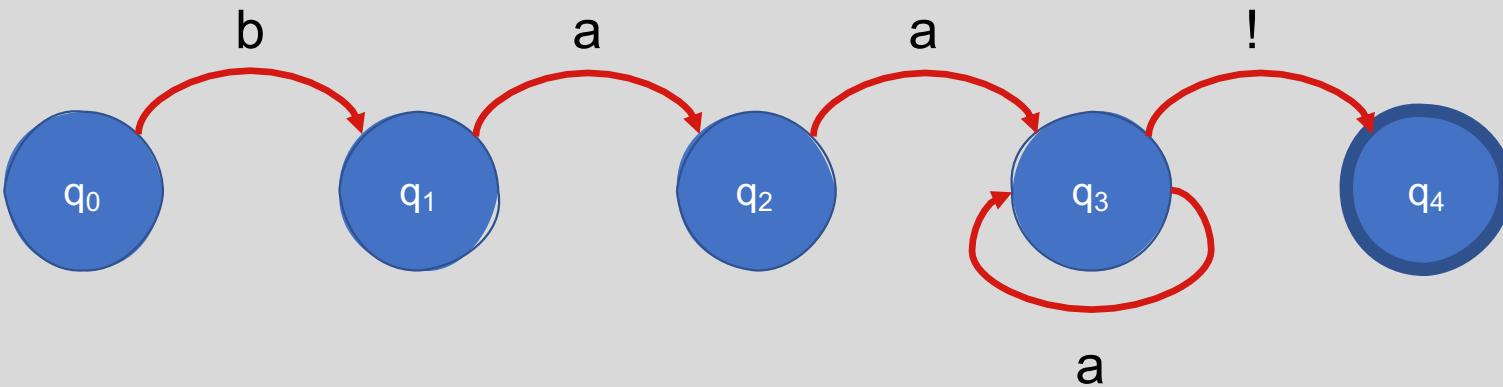


# State transitions in FSAs can be represented using tables.



Currently in State	Next Item in Sequence			
	b	a	!	<end>
$q_0$				
$q_1$				
$q_2$				
$q_3$				
$q_4$				

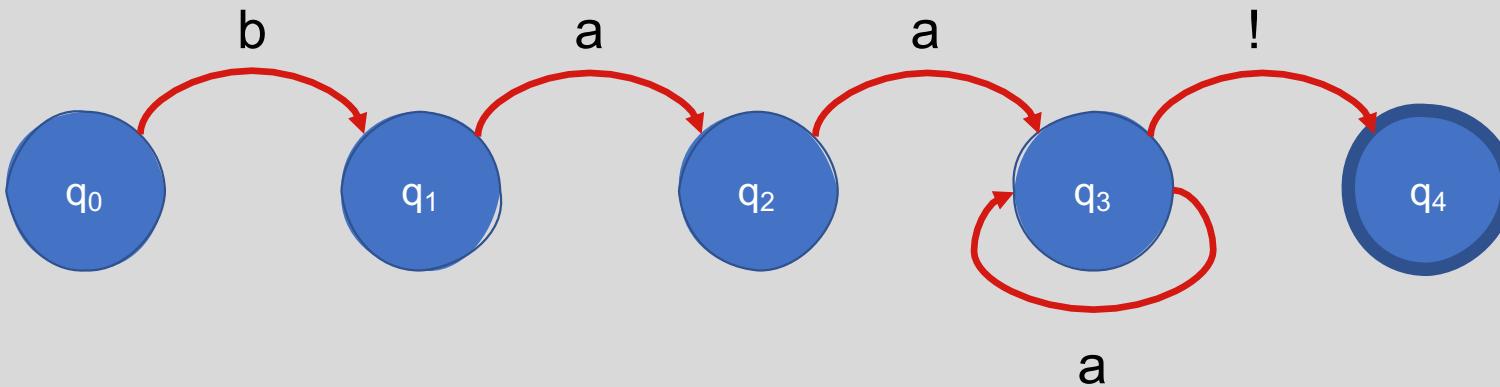
# State transitions in FSAs can be represented using tables.



	Next Item in Sequence			
Currently in State	b	a	!	<end>
q0	q1			
q1				
q2				
q3				
q4				

Go to State

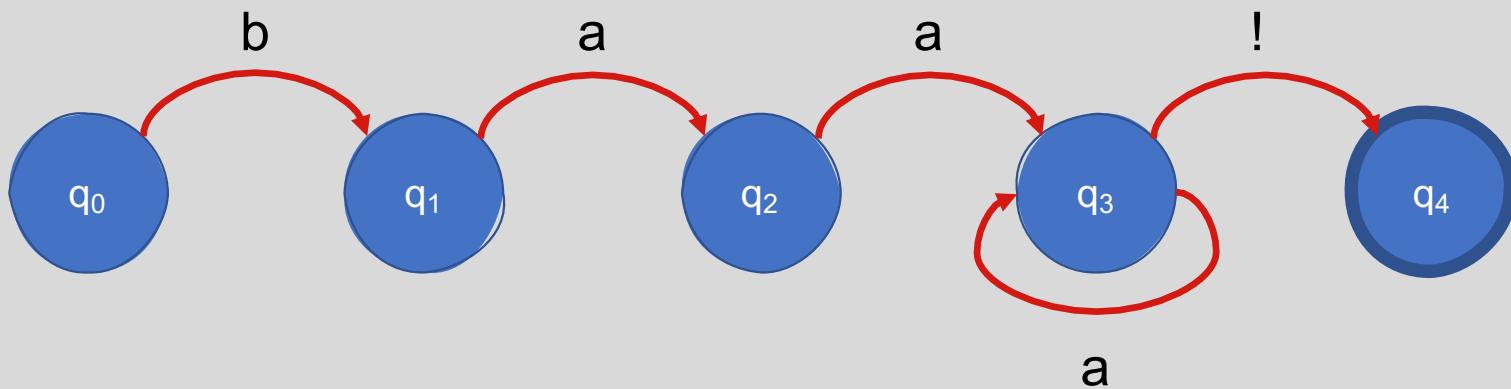
# State transitions in FSAs can be represented using tables.



		Next Item in Sequence			
		b	a	!	<end>
Currently in State	q0	q1	:(	:(	:(
	q1				
	q2				
	q3				
	q4				

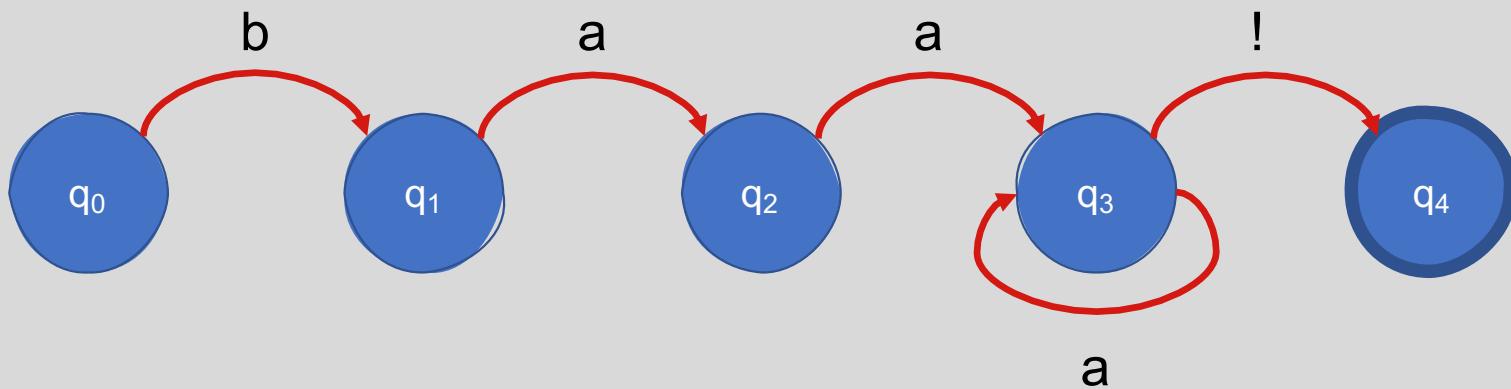
Go to State

**State transitions in FSAs can be represented using tables.**



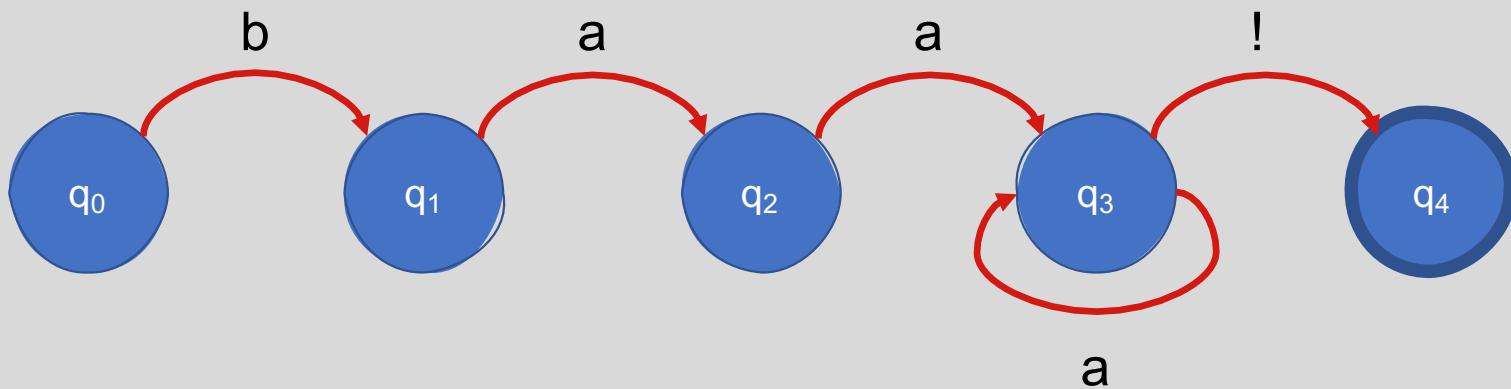
	b	a	!	<end>
q <sub>0</sub>	q <sub>1</sub>	:(	:(	:(
q <sub>1</sub>	:)	q <sub>2</sub>		
q <sub>2</sub>				
q <sub>3</sub>				
q <sub>4</sub>				

**State transitions in FSAs can be represented using tables.**



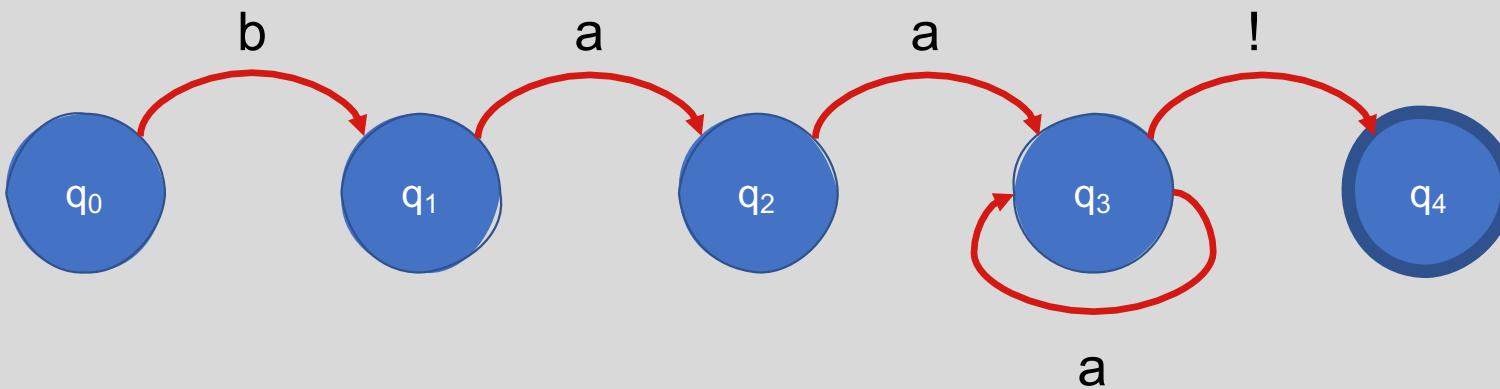
		Next Item in Sequence			
		b	a	!	<end>
Currently in State	q <sub>0</sub>	q <sub>1</sub>	:(	:(	:(
	q <sub>1</sub>	:(	q <sub>2</sub>	:(	:(
	q <sub>2</sub>	:(	q <sub>3</sub>		
	q <sub>3</sub>				
	q <sub>4</sub>				

**State transitions in FSAs can be represented using tables.**



	b	a	!	<end>
Currently in State	$q_0$	$q_1$	:(	:(
	:(	$q_2$	:(	:(
	:(	$q_3$	:(	:(
	:(	$q_3$	:(	

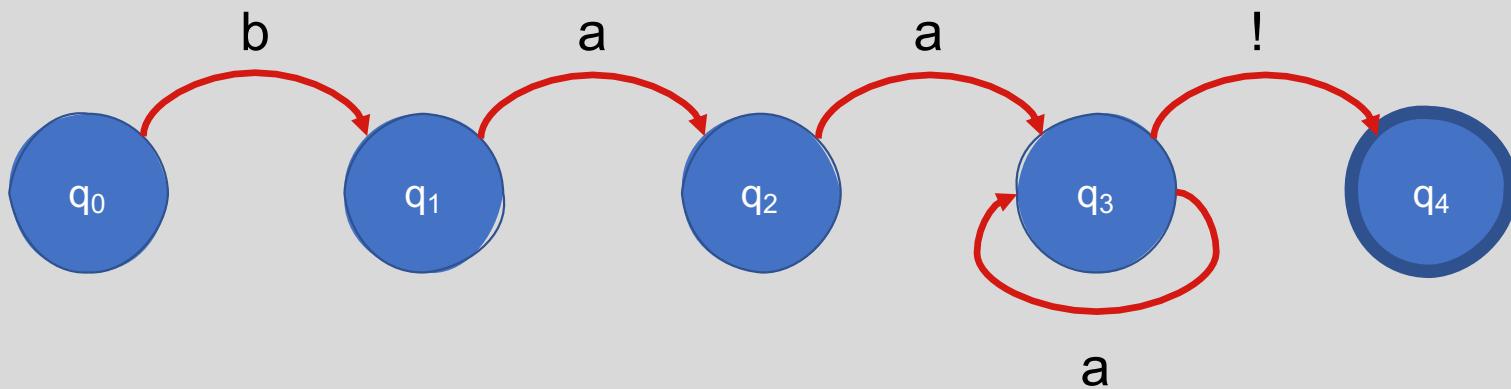
# State transitions in FSAs can be represented using tables.



		Next Item in Sequence			
		b	a	!	<end>
Currently in State	q0	q1	∅	∅	∅
	q1	∅	q2	∅	∅
	q2	∅	q3	∅	∅
	q3	∅	q3	q4	
	q4				

Go to State

**State transitions in FSAs can be represented using tables.**



	b	a	!	<end>
Currently in State	$q_0$	$q_1$	:(	:(
	$q_0$	:(	$q_2$	:(
	$q_2$	:(	$q_3$	:(
	$q_3$	:(	$q_4$	:(
	$q_4$	:(	:(	:)

**State  
transition  
tables  
simplify the  
process of  
determining  
whether  
your input  
will be  
accepted by  
the FSA.**

- For a given sequence of items to match, **begin in the start state** with the first item in the sequence
- **Consult the table** ...is a transition to any other state permissible with the current item?
- If so, **move to the state indicated by the table**
- If you make it to the end of your sequence and to a final state, **accept**

# Algorithmically, this looks like:

```
index ← beginning of sequence
current_state ← initial state of FSA
loop:
    if end of sequence has been reached:
        if current_state is an accept state:
            return accept
        else:
            return reject
    else if transition_table[current_state, sequence[index]] is empty:
        return reject
    else:
        current_state ← transition_table[current_state, sequence[index]]
        index ← index + 1
end
```

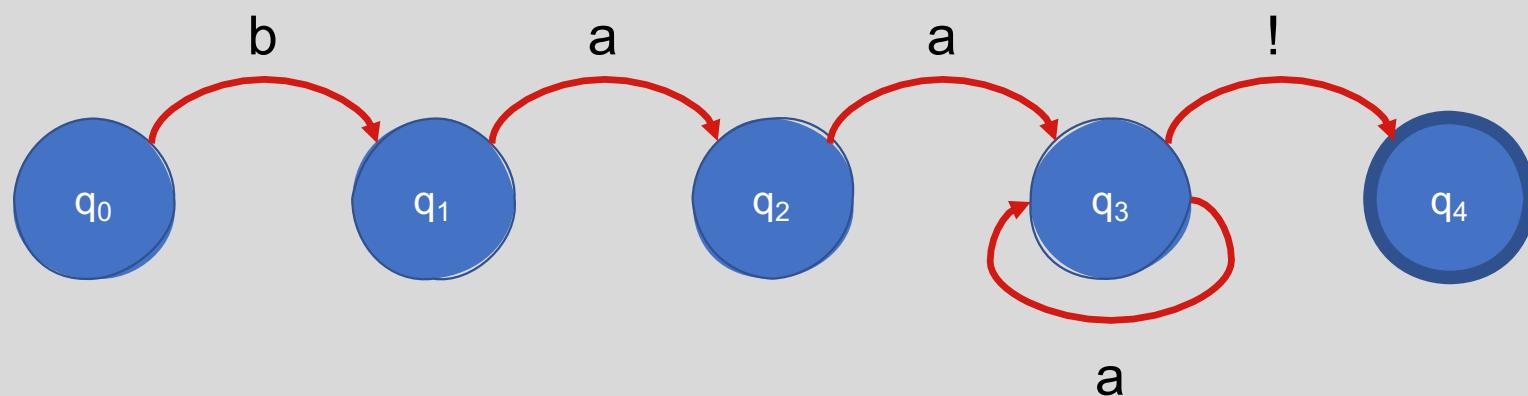
# Deterministic vs. Non-Deterministic FSAs

**Deterministic FSA:** At each point in processing a sequence, there is one unique thing to do (no choices!)

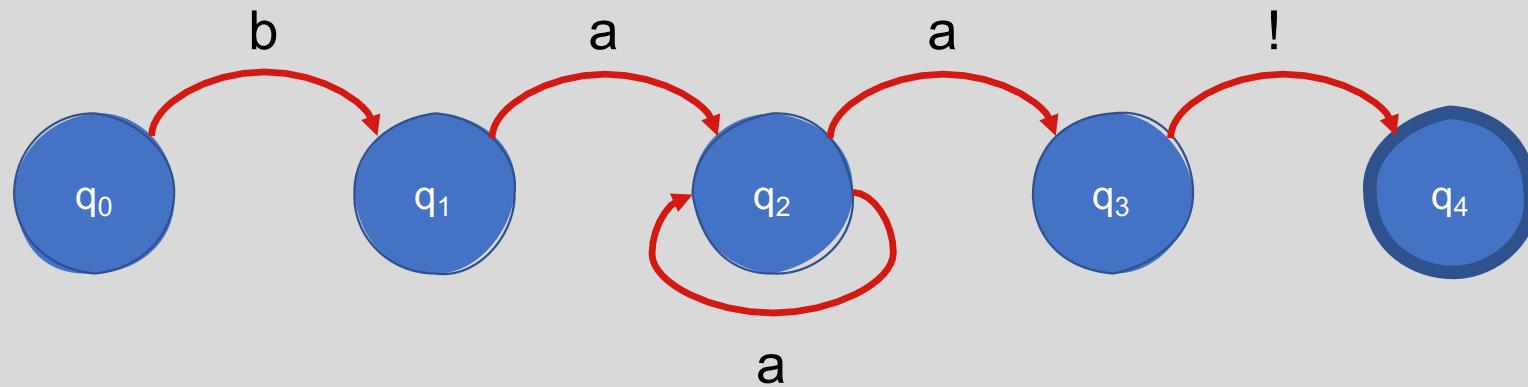
**Non-Deterministic FSA:** At one or more points in processing a sequence, there are multiple permissible next steps (choices!)

# Deterministic or Non-Deterministic?

?

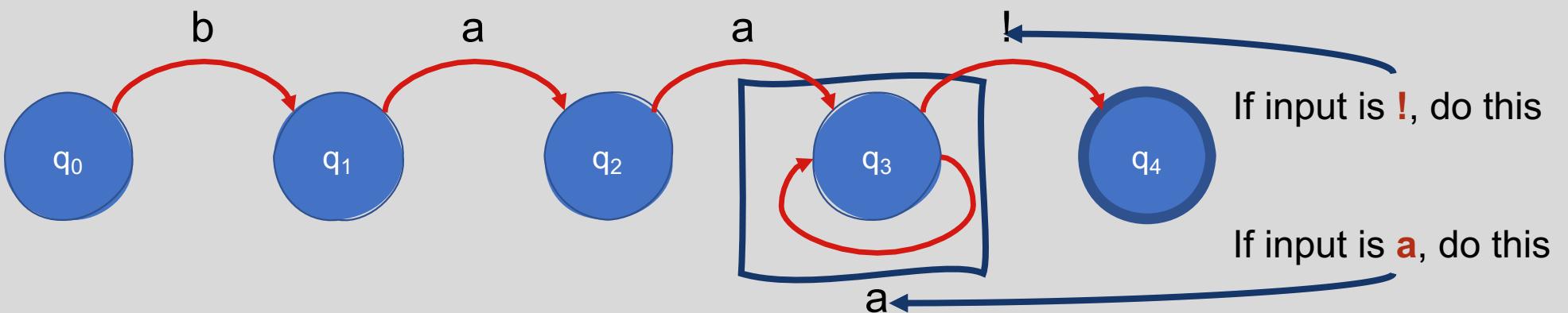


?

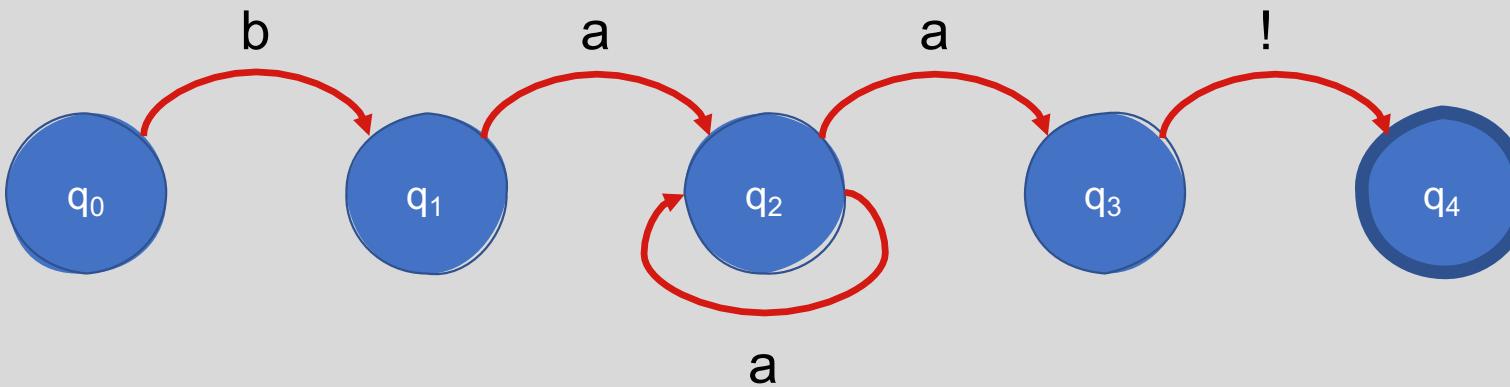


# Deterministic or Non-Deterministic?

Deterministic!

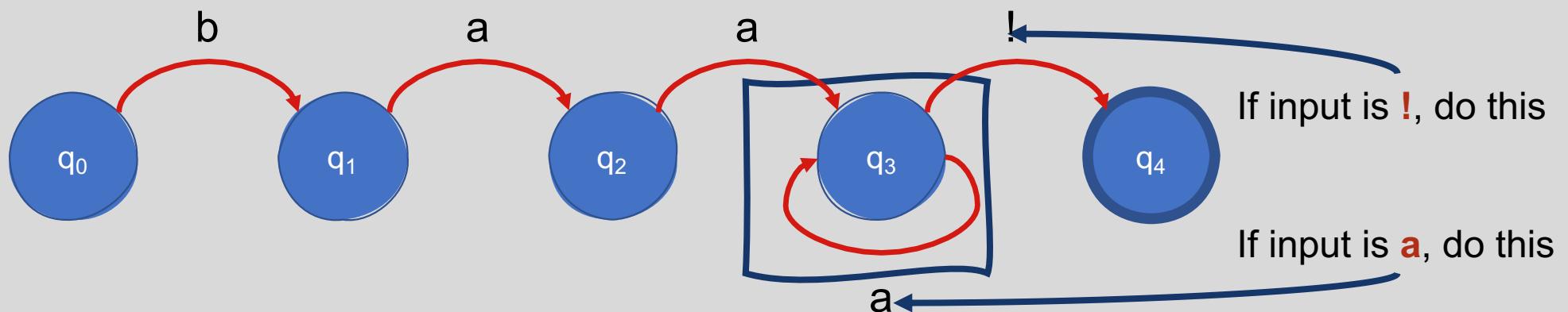


?

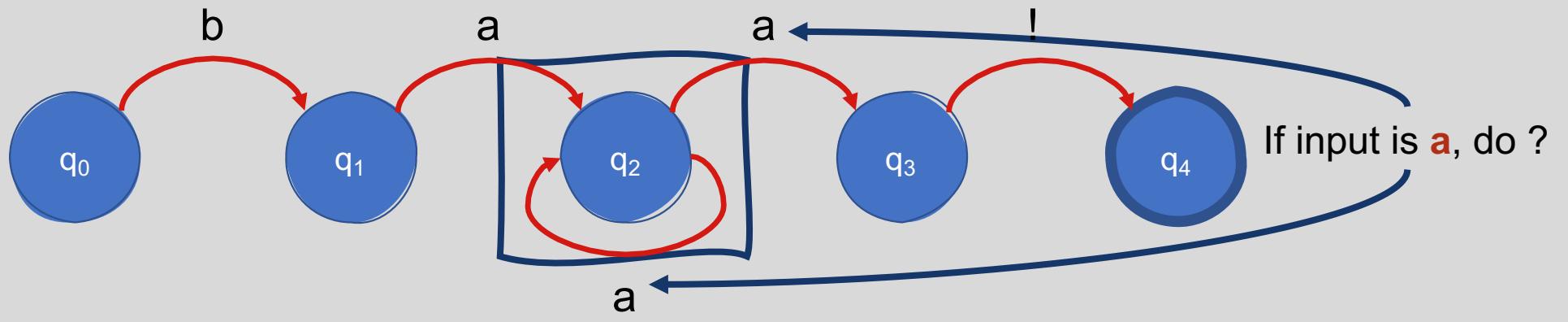


# Deterministic or Non-Deterministic?

Deterministic!



Non-Deterministic!



**Every non-deterministic FSA can be converted to a deterministic FSA.**

- Both are equally powerful
- Deterministic FSAs can accept as many languages as non-deterministic ones



# Non-Deterministic FSAs: How to check for input acceptance?

- Either:
  1. Convert the non-deterministic FSA to a deterministic FSA and then check that version, or
  2. Manage the process as a state-space search

# **Non-Deterministic FSA Search Assumptions**

---

There is at least one path through the FSA for an input that is part of the language defined by the FSA

---

Not all paths through the FSA for an “accept” input lead to an accept state

---

No paths through the FSA lead to an accept state for an input that is not a valid member of the language



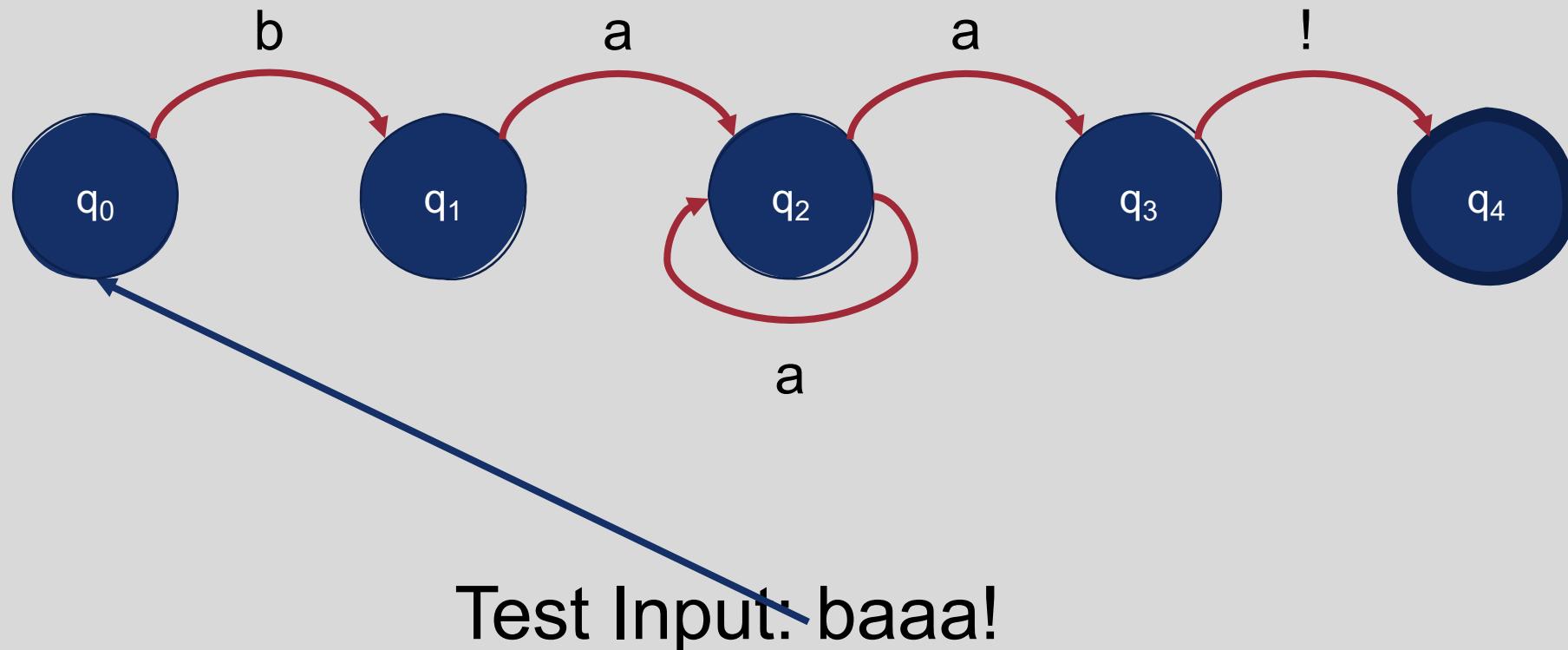
# Non-Deterministic FSA Search

- States in the search space are pairings of sequence indices and states in the FSA
- By keeping track of which states have and have not been explored, we can systematically explore all the paths through an FSA given an input

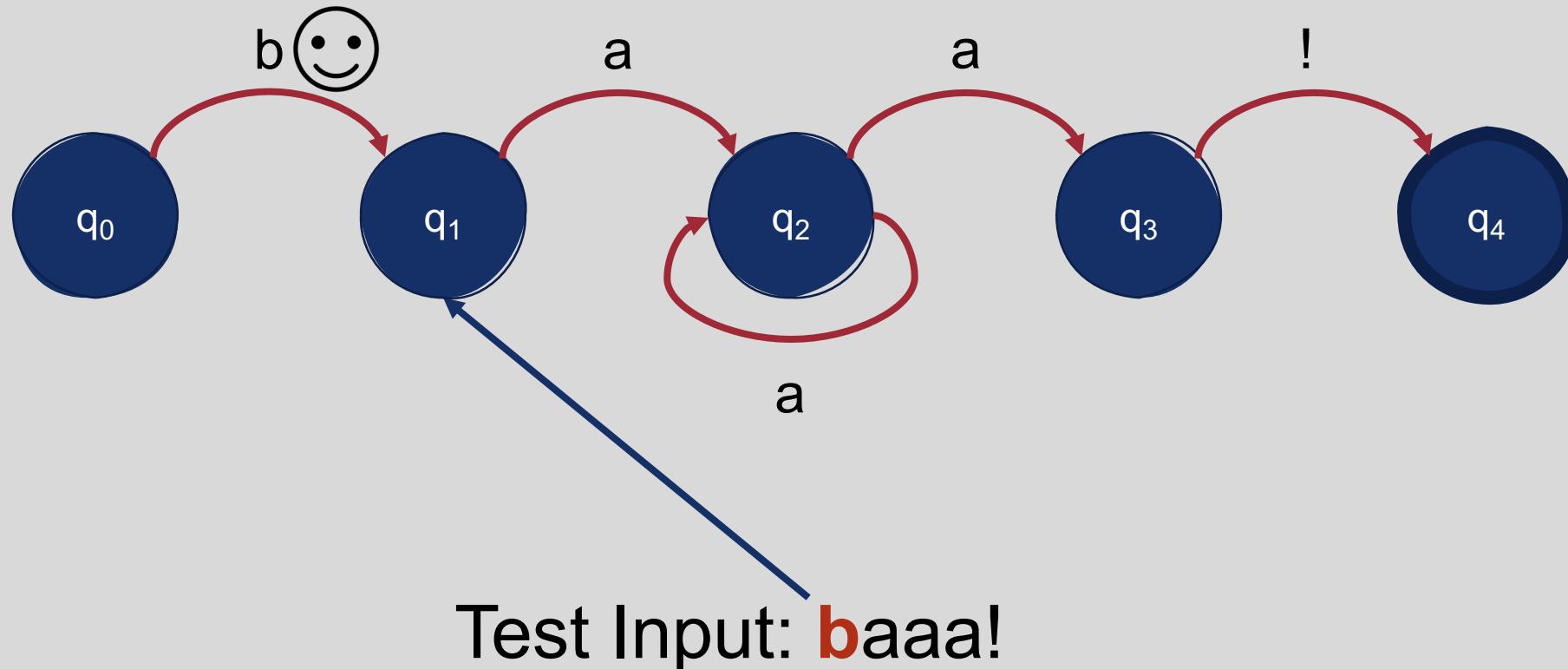
# Non-Deterministic FSA Search Assumptions

- **Success:** Path is found for a given input that ends in an accept
- **Failure:** All possible paths for a given input lead to failure

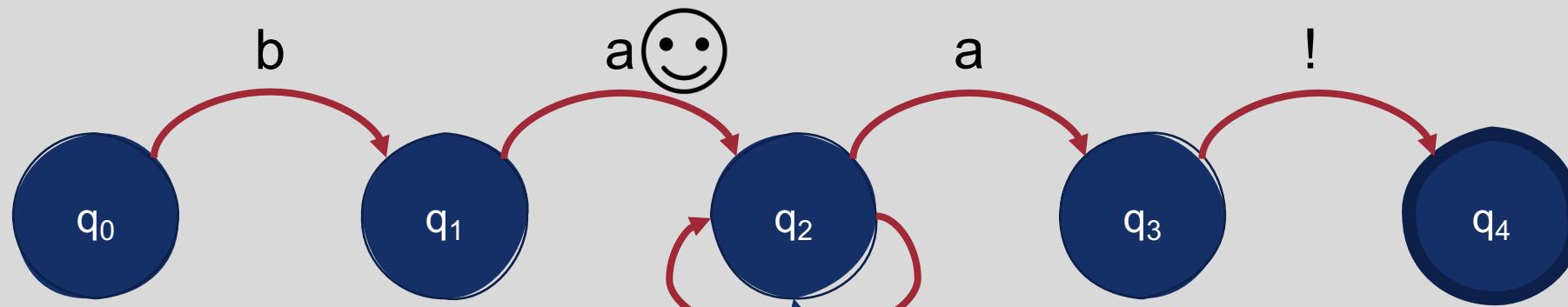
# Example: Non-Deterministic FSA Search



# Example: Non-Deterministic FSA Search

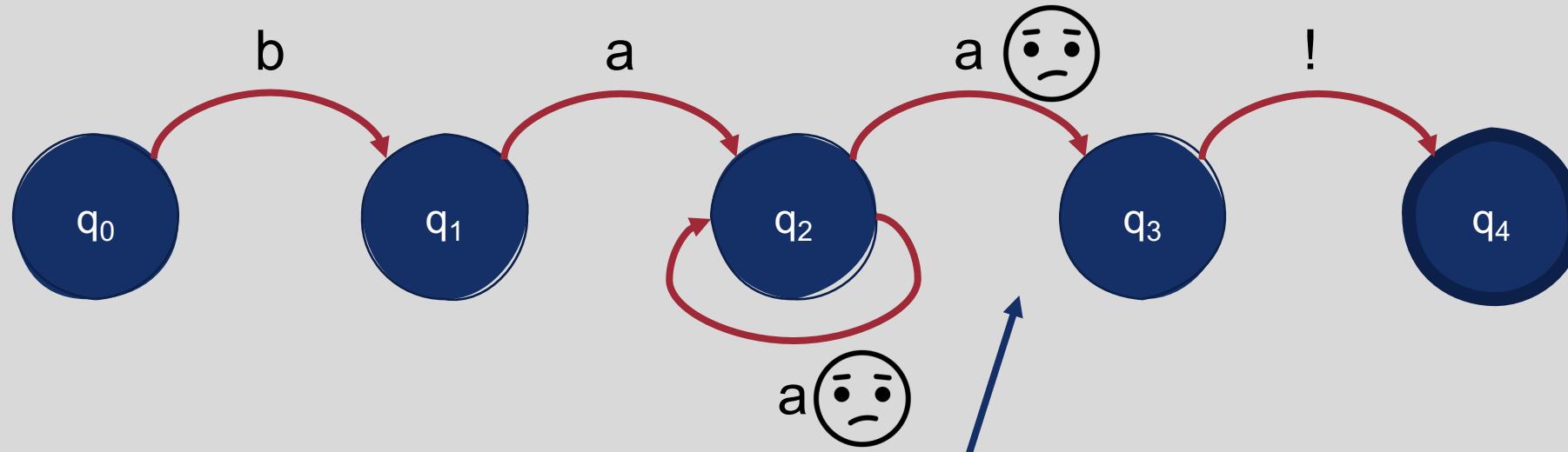


# Example: Non-Deterministic FSA Search



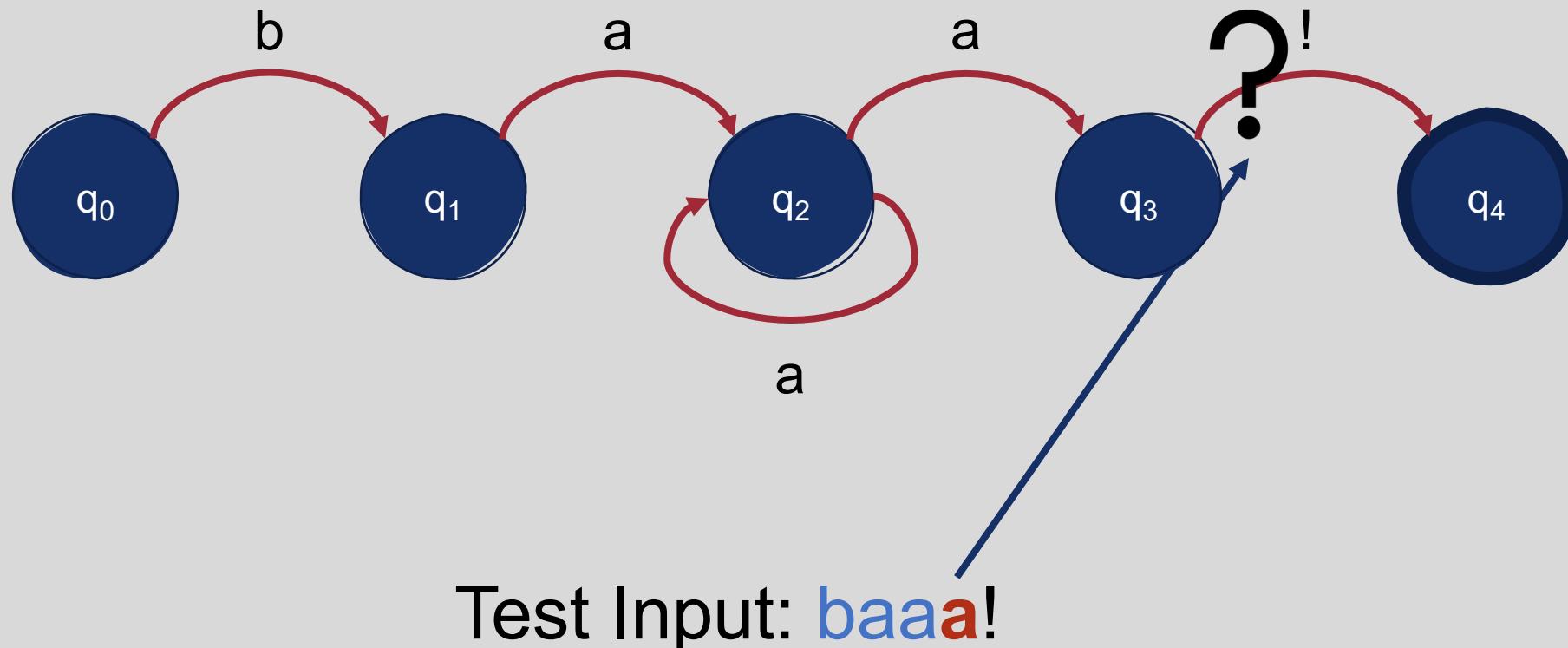
Test Input: **b**a<sup>aa</sup>!

# Example: Non-Deterministic FSA Search

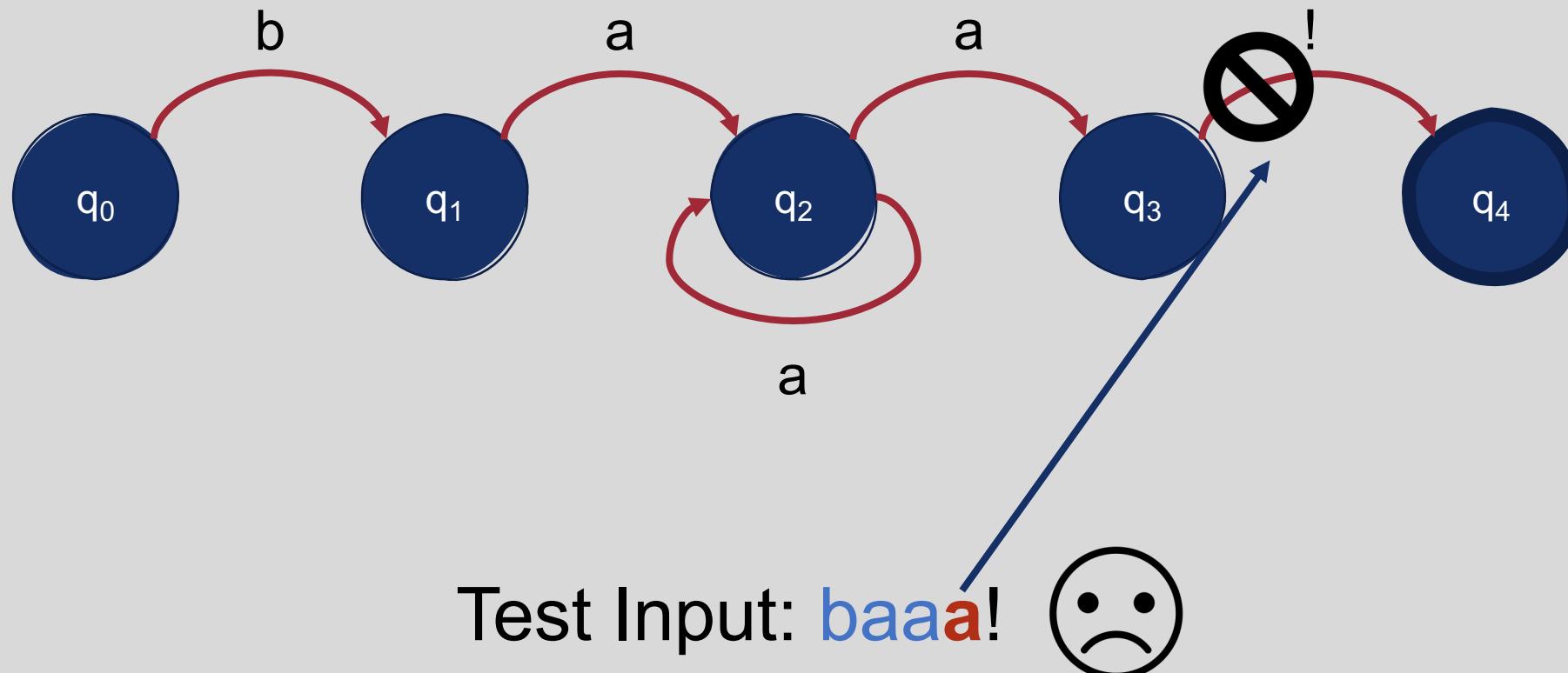


Test Input: **ba**  
**a**!

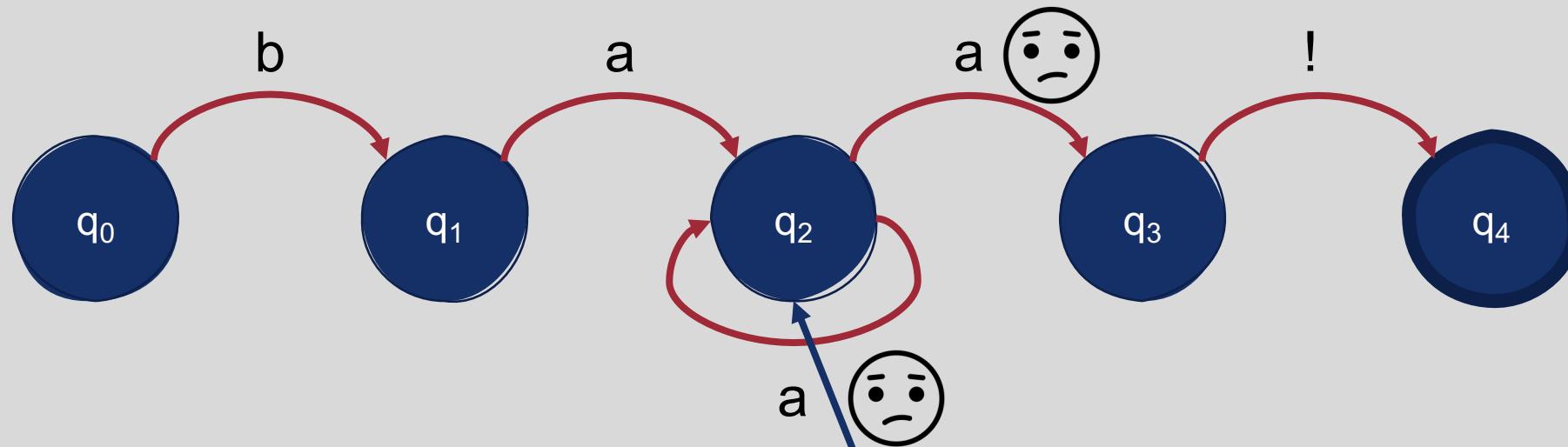
# Example: Non-Deterministic FSA Search



# Example: Non-Deterministic FSA Search

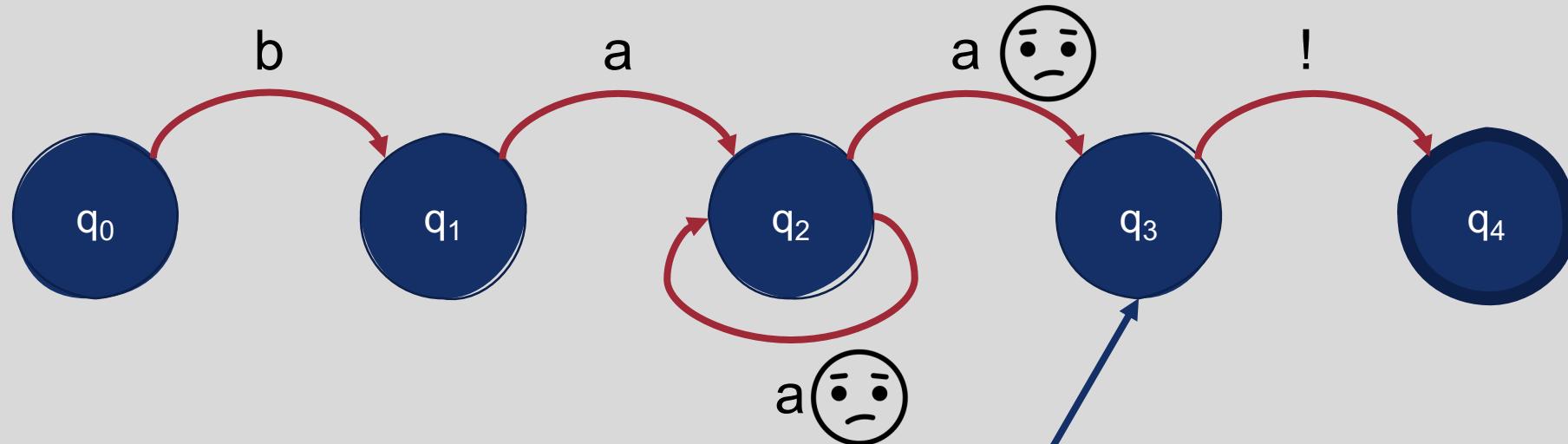


# Example: Non-Deterministic FSA Search



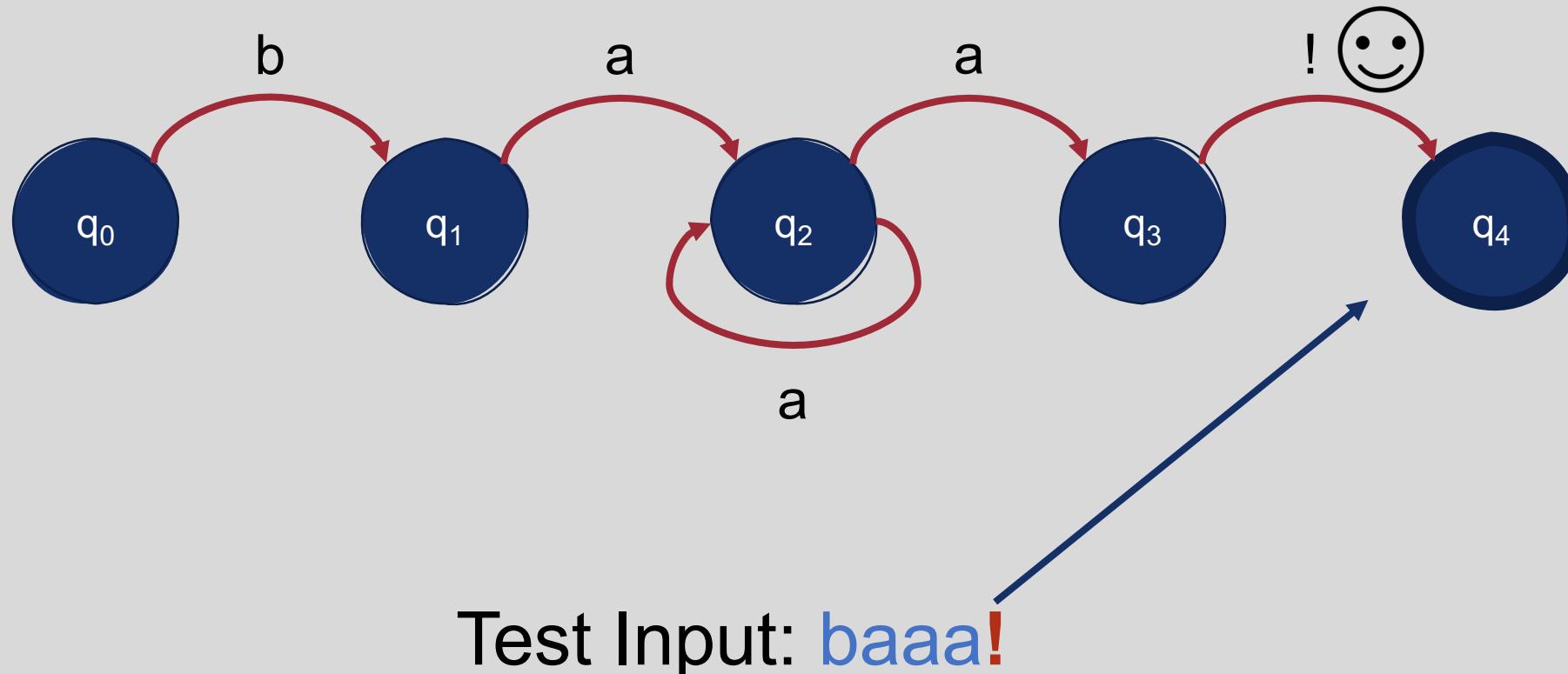
Test Input: ba<sub>red</sub>a!

# Example: Non-Deterministic FSA Search

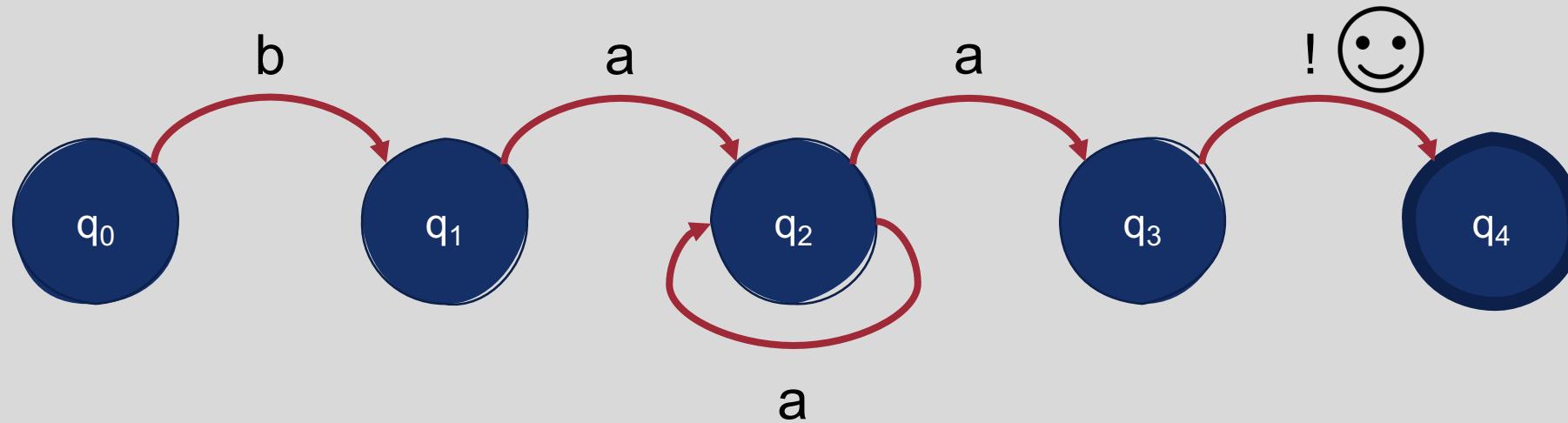


Test Input: **baaa!**

# Example: Non-Deterministic FSA Search



# Example: Non-Deterministic FSA Search



Test Input: **baaa!**

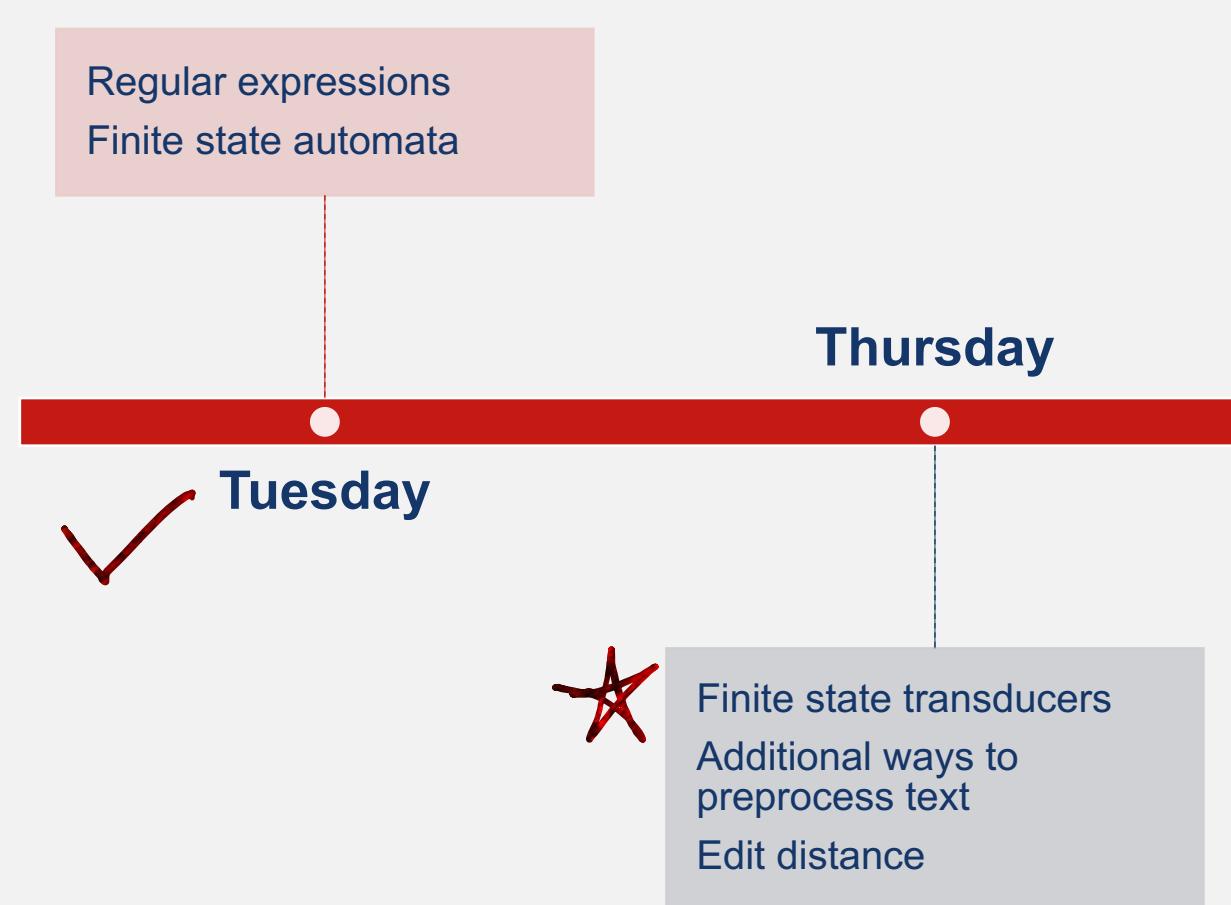
# Additional Characteristics of FSAs

- You can apply set operations to any FSA
  - Union
  - Concatenation
  - Negation
    - For non-deterministic FSAs, first convert to a deterministic FSA
  - Intersection
- To do so, you may need to utilize an  $\epsilon$  transition
  - $\epsilon$  transition: Move from one state to another without consuming an item from the input sequence

# Summary: Finite State Automata

- FSAs are computational models that describe regular languages
- To determine whether an input item is a member of an FSA's language, you can process it sequentially from the start to (hopefully) the final state
- State transitions in FSAs can be represented using tables
- FSAs can be either deterministic or non-deterministic

# This Week's Topics



**FSAs allow us to model regular languages, but they don't allow us to define mappings between sets of items.**

- How can we extend this model to accommodate those mappings?
  - Finite state transducers

# Finite State Transducers

---

**Finite State Transducer (FST):** A special type of FSA that describes mappings between two sets of items

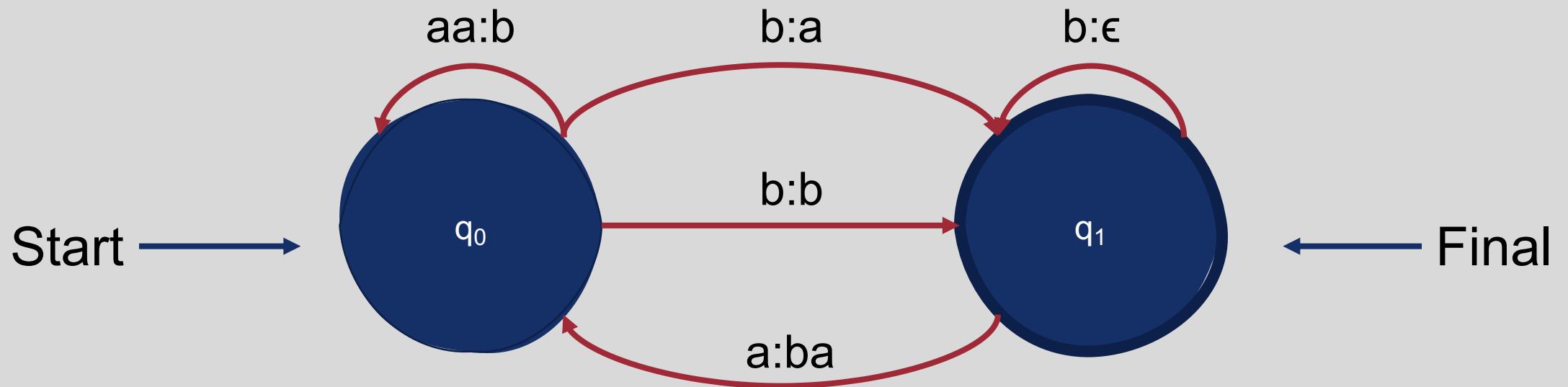
---

FSTs recognize or generate pairs of items

---

FSAs can be converted to FSTs by labeling each arc with input and output items (e.g., **a:b** for an input of **a** and an output of **b**)

# Example: Simple FST



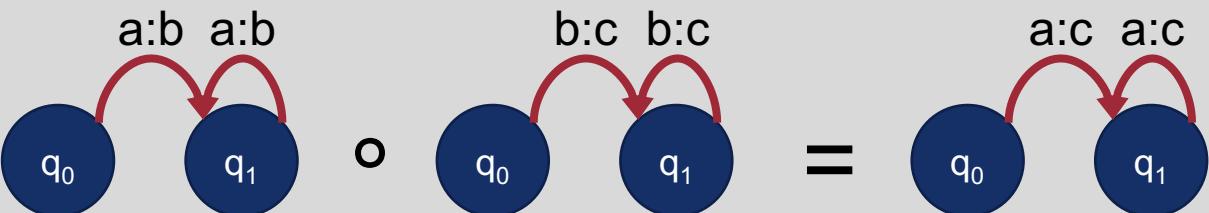


# Formal Definition

- A finite state transducer can be specified by enumerating the following properties:
  - The set of states,  $Q$
  - A finite **input** alphabet,  $\Sigma$
  - **A finite output alphabet,  $\Delta$**
  - A start state,  $q_0$
  - A set of accept/final states,  $F \subseteq Q$
  - A transition function or transition matrix between states,  $\delta(q,i)$
  - **An output function giving the set of possible outputs for each state and input,  $\sigma(q,i)$**
- $\delta(q,i)$ : Given a state  $q \in Q$  and input  $i \in \Sigma$ ,  **$\delta(q,i)$  returns a new state  $q' \in Q$ .**

# Formal Properties

- FSTs are compositional:
  - Letting  $T_1$  be an FST from  $I_1$  to  $O_1$ , and
  - Letting  $T_2$  be an FST from  $I_2$  to  $O_2$ ,
  - The two FSTs can be composed such that the resulting FST maps directly from  $I_1$  to  $O_2$
- FSTs can be inverted:
  - Letting  $T$  be an FST that maps from  $I$  to  $O$ , its inversion ( $T^{-1}$ ) will map from  $O$  to  $I$ .



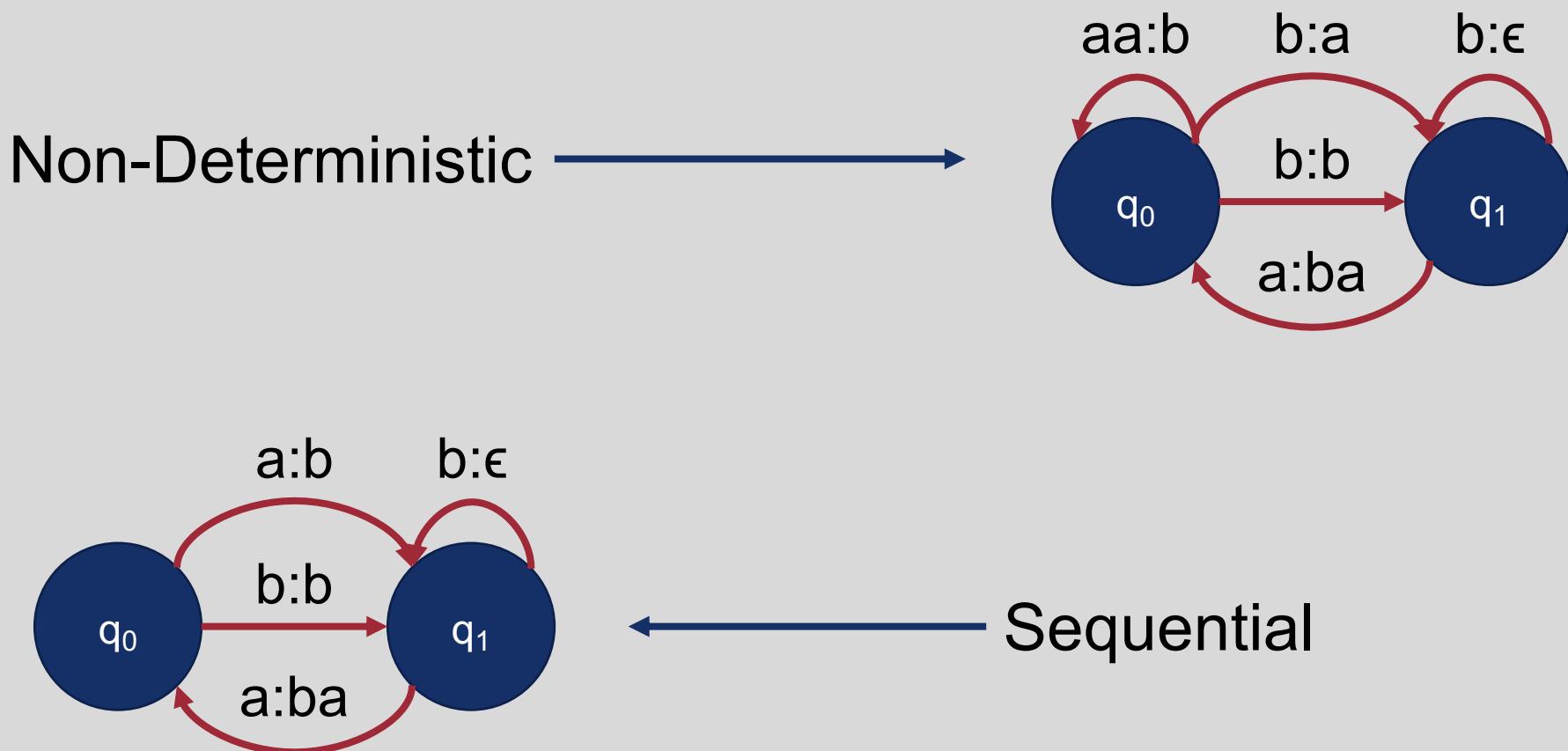
# Deterministic vs. Non-Deterministic FSTs

Just like FSAs, **FSTs can be non-deterministic**  
...one input can be translated to many possible outputs!

Unlike FSAs, **not all non-deterministic FSTs can be converted to deterministic FSTs**

FSTs with underlying deterministic FSAs (at any state, a given input maps to at most one transition out of the state) are called **sequential transducers**

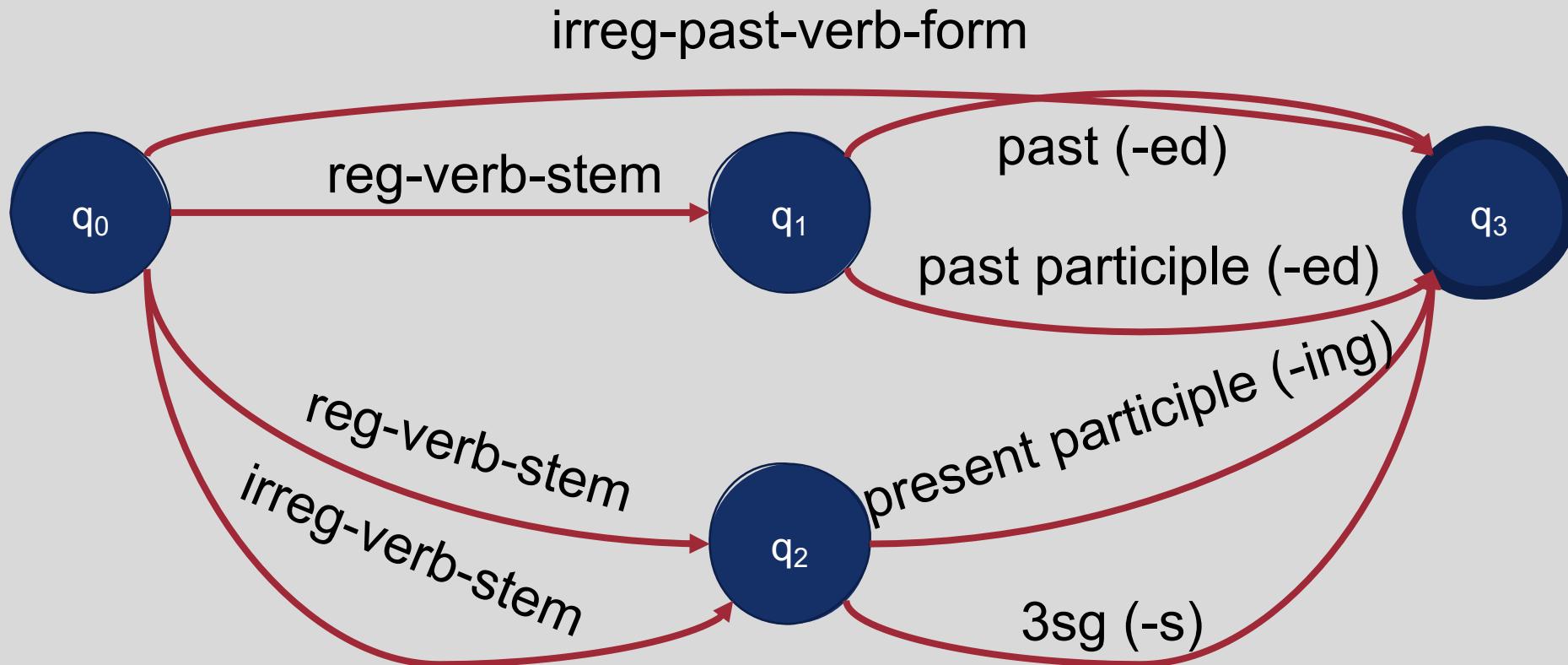
# Examples: Non-Deterministic and Sequential Transducers



# When can we use FSTs in NLP?

- Case example: **Morphological parsing**
  - The task of recognizing the component morphemes of words (e.g., foxes → fox + es) and building structured representations of those components
- **Morphemes:**
  - Small meaningful units that make up words
  - **Stems:** The core meaning-bearing units
  - **Affixes:** Bits and pieces that adhere to stems and add additional information
    - -ed
    - -ing
    - -s

# Example Morphological Lexicon



# Why is morphological parsing necessary?

- Useful for breaking language into more easily interpretable parts
- Morphemes can be **productive**
  - Example: **-ing** attaches to almost every verb, including brand new words
    - “Why are you Instagram**ing** that?”
- Some languages are very **morphologically complex**
  - Uygarlastiramadiklarimizdanmissinizcasina
    - Uygar “civilized” + **las** “become”
    - + **tir** “cause” + **ama** “not able”
    - + **dik** “past” + **lar** “plural”
    - + **imiz** “p1pl” + **dan** “abl”
    - + **mis** “past” + **siniz** “2pl” + **casina** “as if”



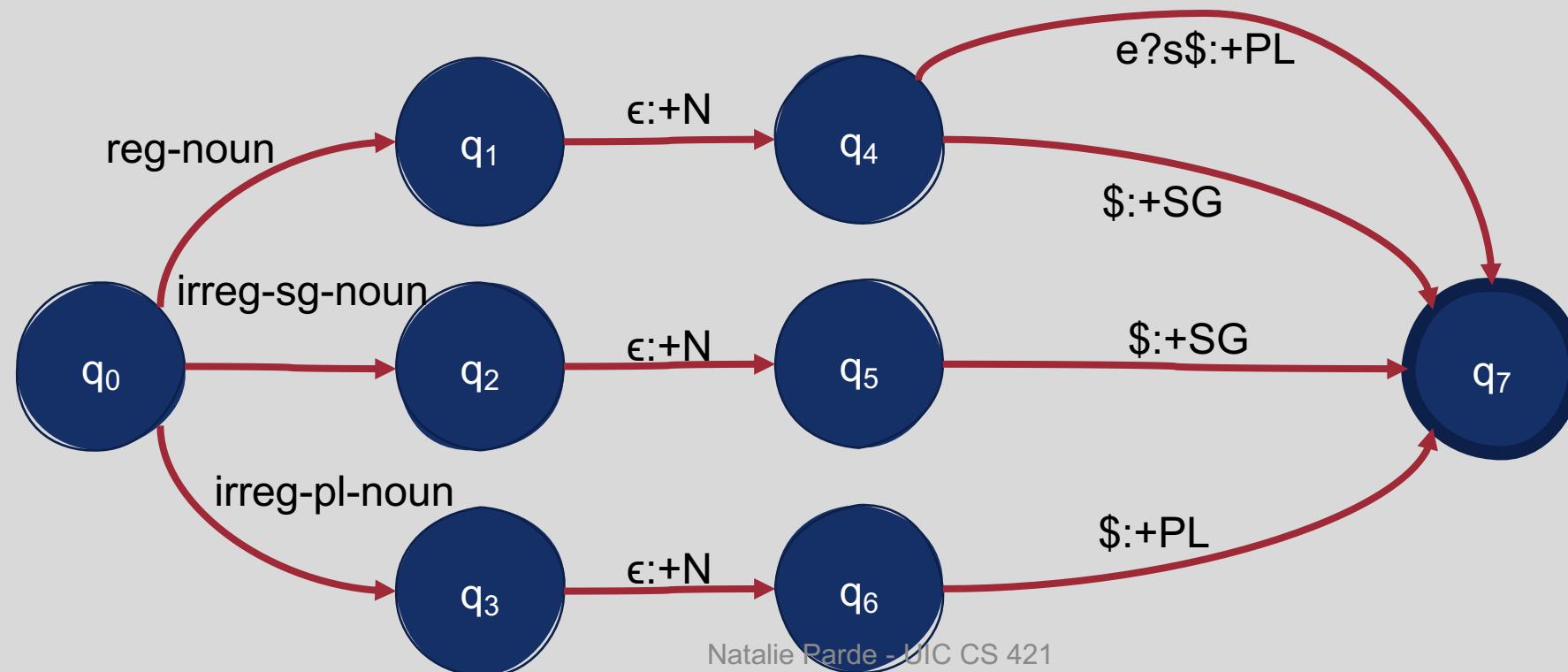
Surface Text	Morphological Parse
cats	cat +N +PL
cat	cat +N +SG
cities	city +N +PL
geese	goose +N +PL
goose	goose +N +SG
merging	merge +V +PresPart
caught	catch +V +Past

# Finite State Morphological Parsing

Goal: Take input surface realizations and produce morphological parses as output

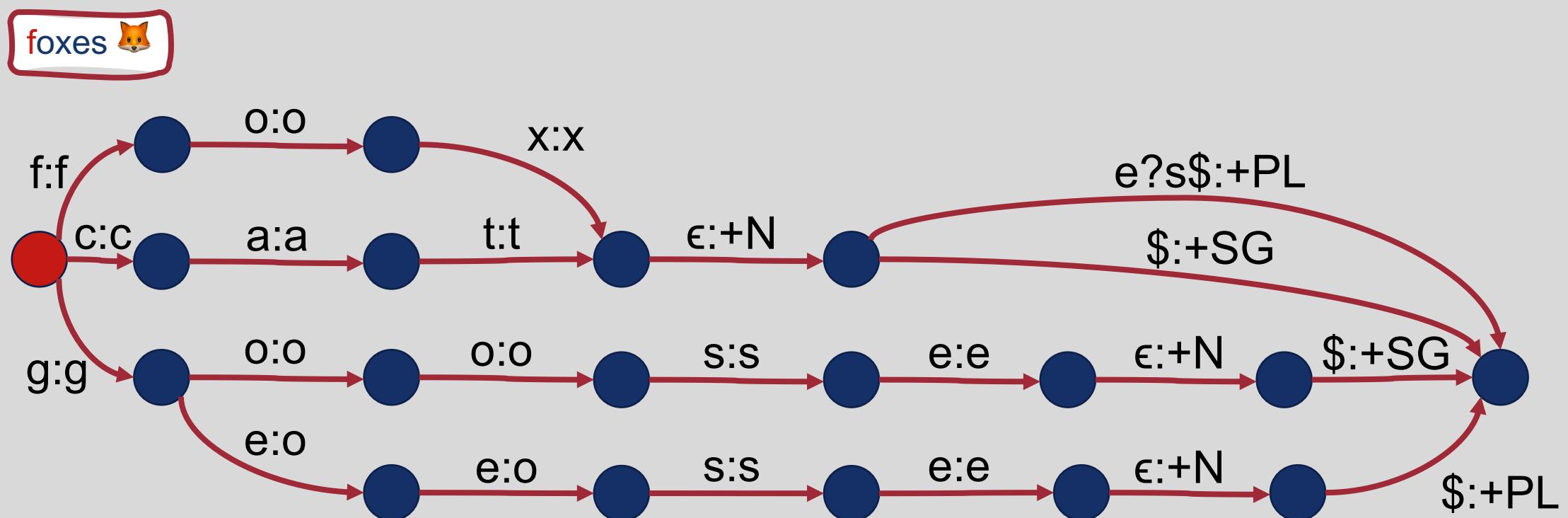
# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



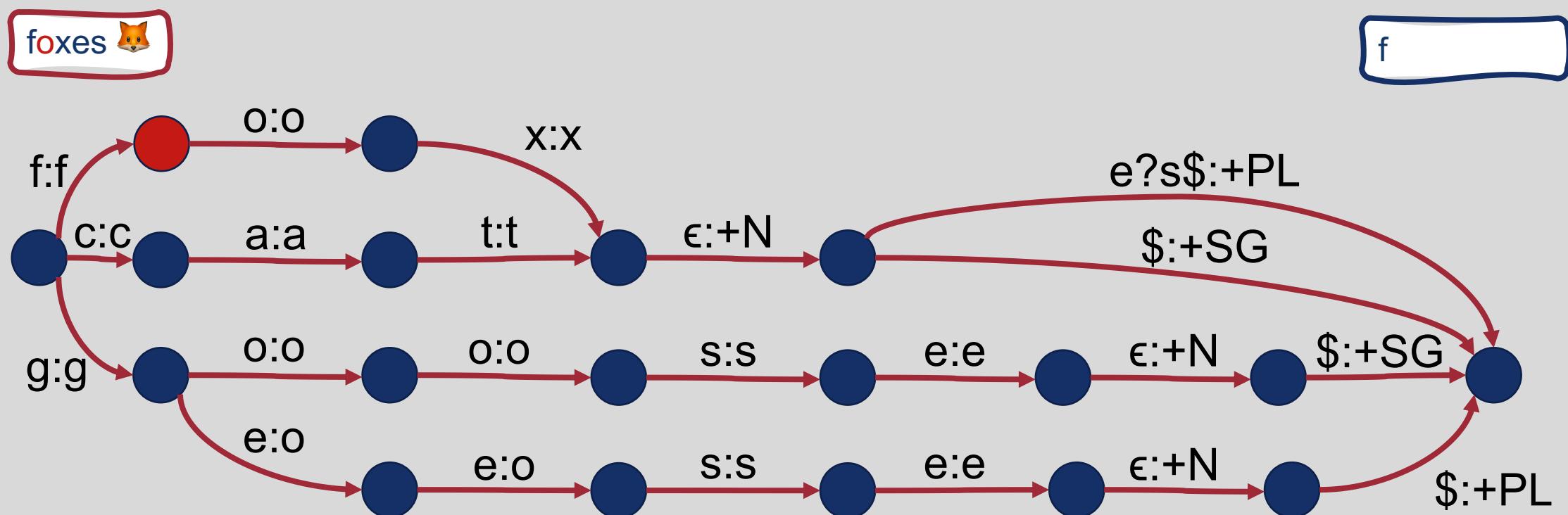
# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



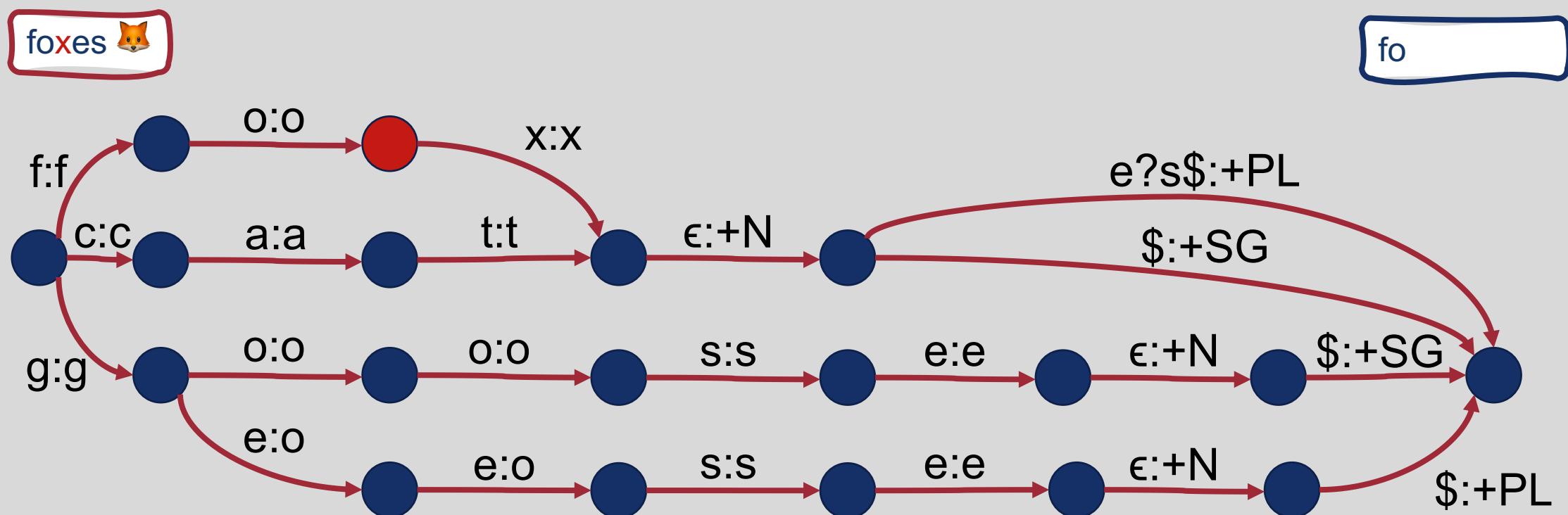
# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



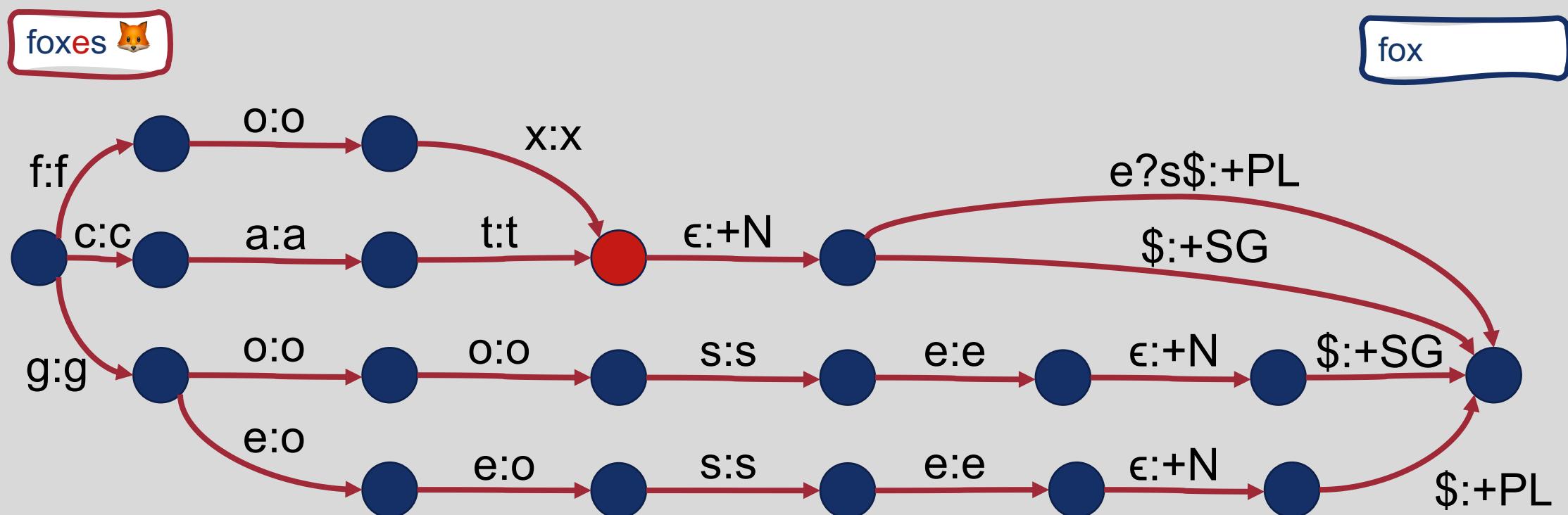
# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



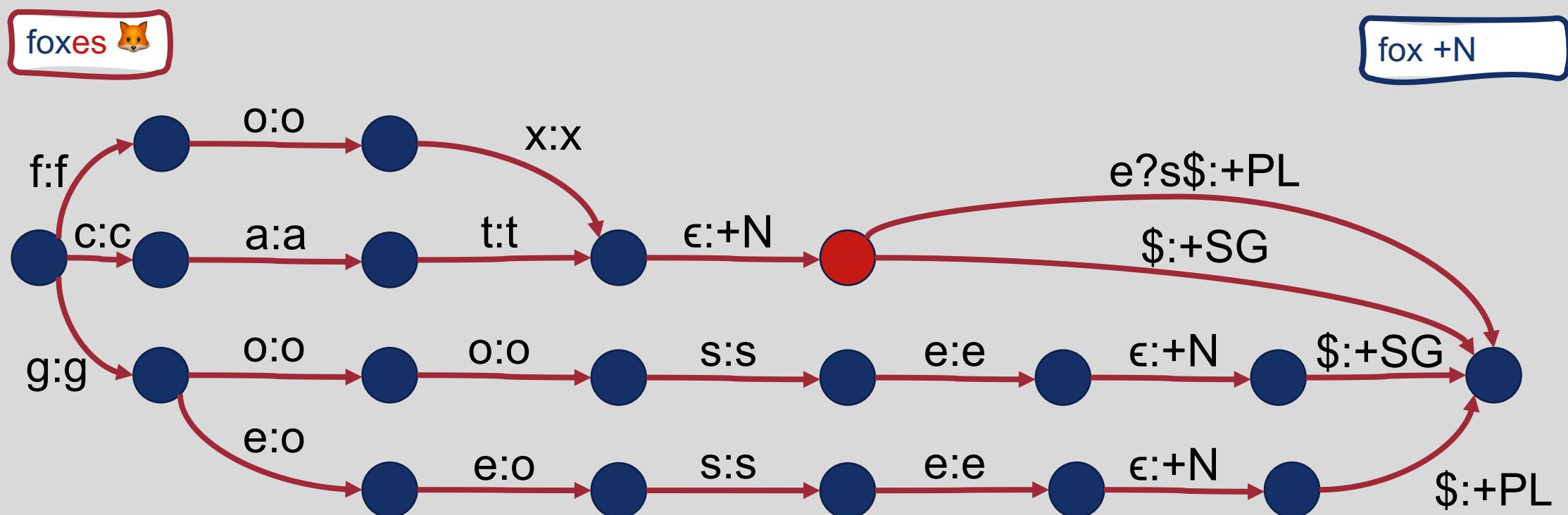
# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



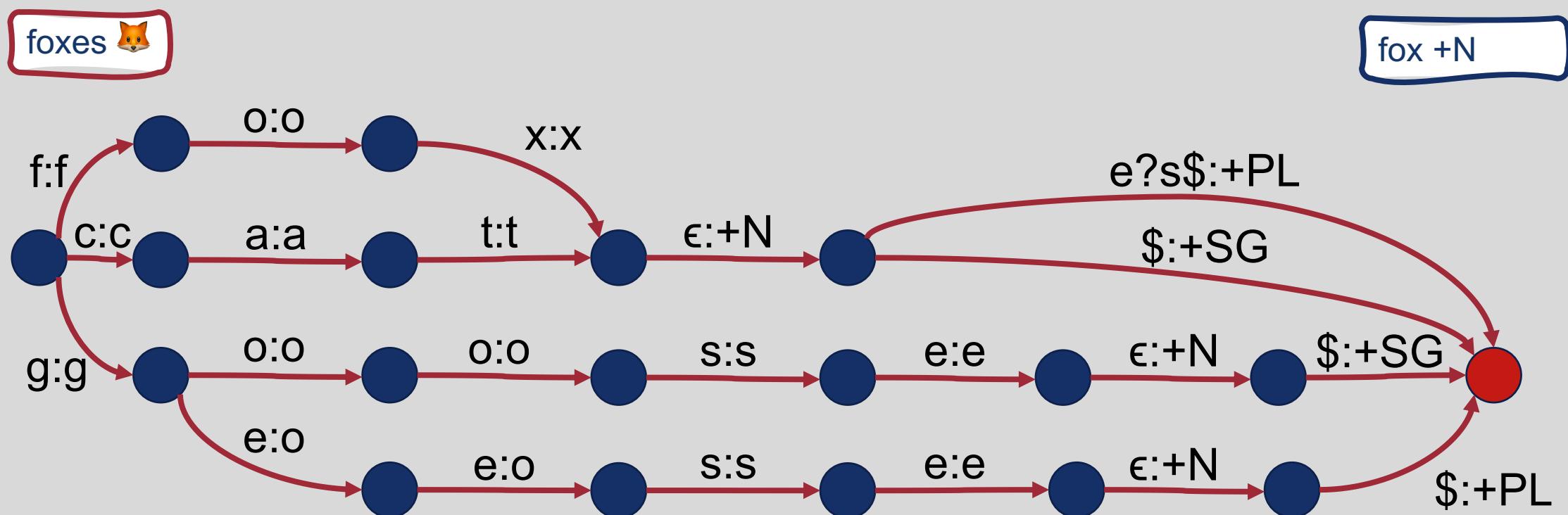
# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



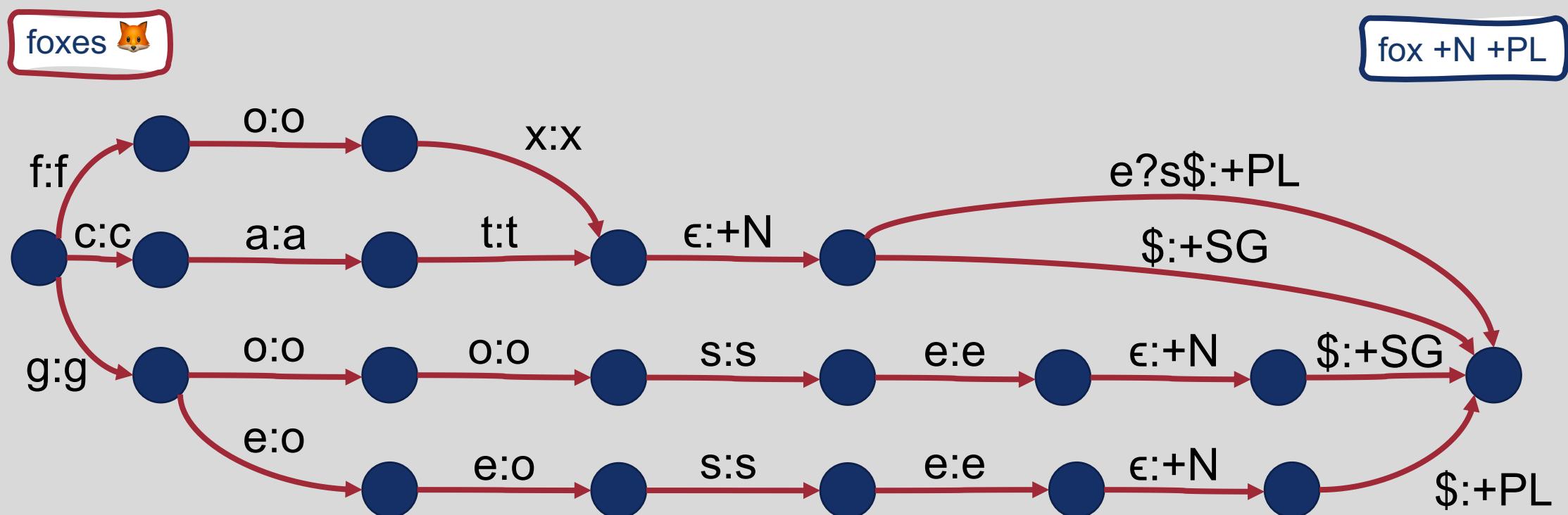
# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



# Finite State Morphological Parsing

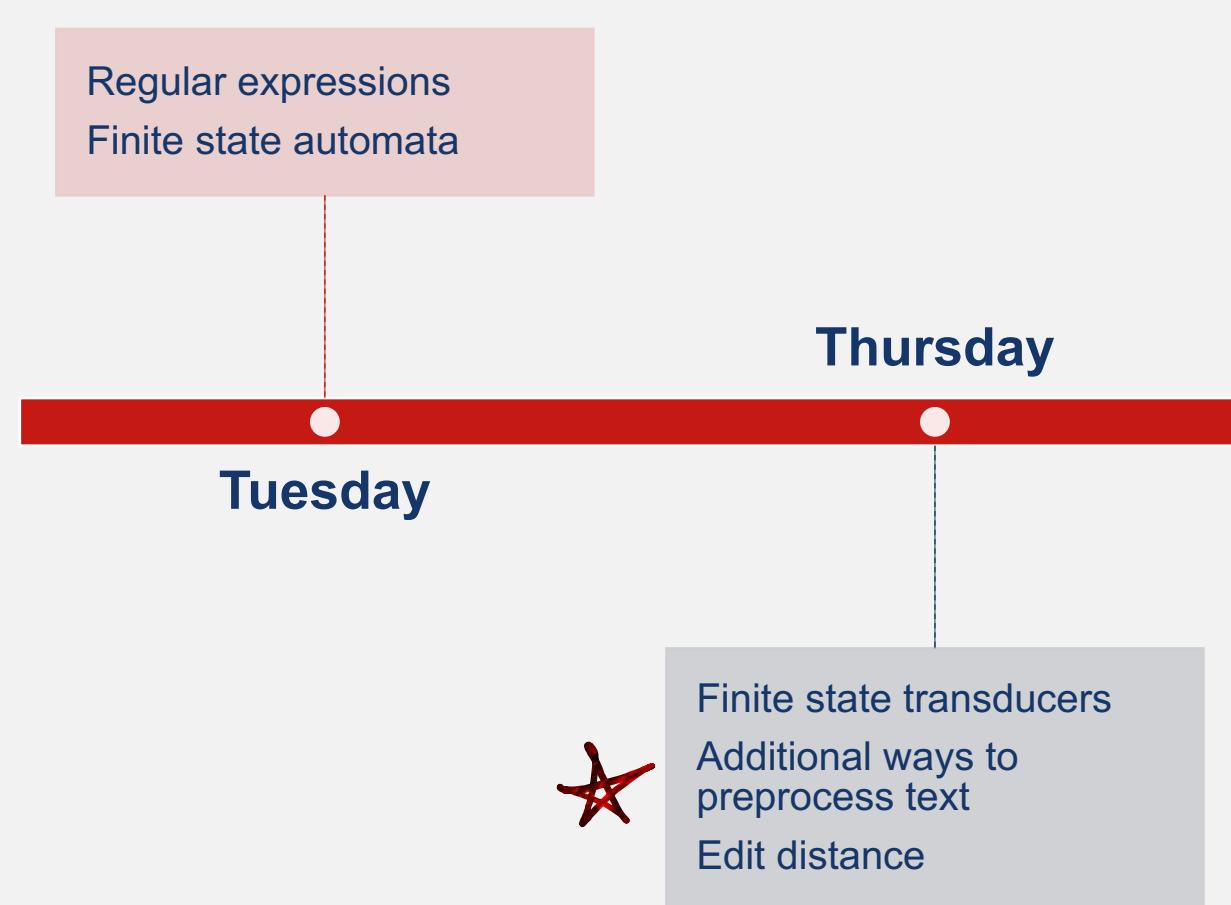
reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



## **Summary: Finite State Transducers**

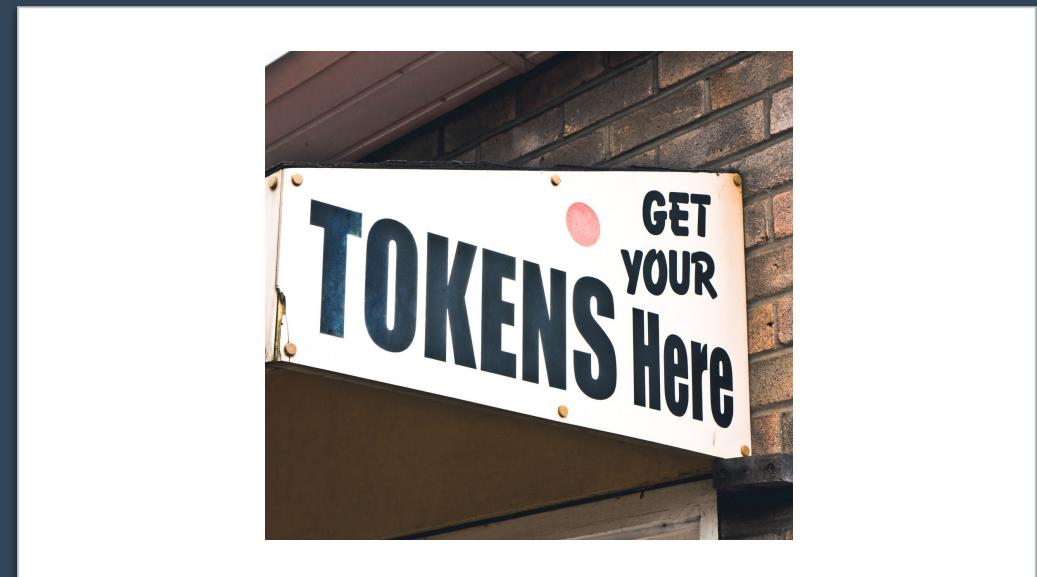
- FSTs are FSAs that describe mappings between two sets
- All non-deterministic FSAs can be converted to deterministic versions, but all non-deterministic FSTs cannot
- FSTs with underlying deterministic FSAs are called sequential transducers
- FSTs are particularly useful for morphological parsing

# This Week's Topics



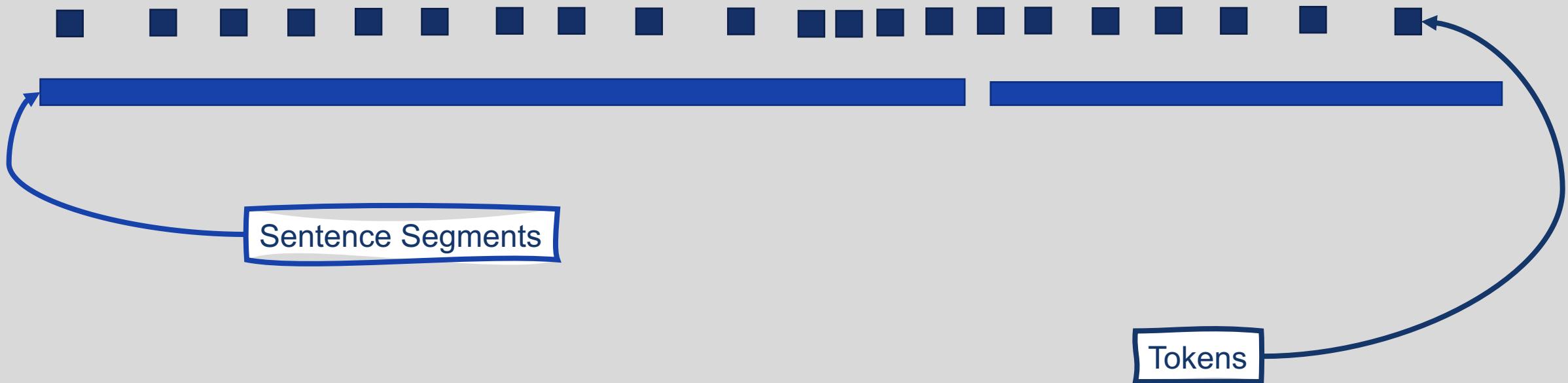
# What are some other ways that we can preprocess text?

- Text tokenization is an important first step for most NLP tasks
  - Often implemented using regular expressions
- Typical NLP pipeline:
  - Segmenting sentences (if applicable)
  - Tokenizing words
  - Normalizing word formats (e.g., favourite → favorite)



# Sentence Segmentation and Text Tokenization

Alice looked all round the table, but there was nothing on it but tea. "I don't see any wine," she remarked.





---

## How many words in a string of text?

- I do uh main- mainly business data processing
  - Fragments, filled pauses
- Seuss's **cat** in the hat is different from other **cats!**
  - **Lemma:** Words with the same stem, coarse-grained part of speech, and general word sense
    - cat and cats = same lemma
  - **Wordform:** The full inflected surface form of a word
    - cat and cats = different wordforms



# Types vs. Tokens

Alice looked all round the table, but there was nothing on it **but tea**.

- **Type**: an element of the vocabulary.
- **Token**: an instance of that type in running text.
- How many?
  - 14 tokens (or 16 if punctuation is tokenized separately)
  - 13 types (or 15 if punctuation is tokenized separately)

---

## Type and Token Counts in Popular Datasets

Dataset	Tokens = N	Types = $ V $
Switchboard phone conversations	2.4 million	20K
Shakespeare	884K	31K
Google N-grams	1 trillion	13 million

**N** = number of tokens

**V** = vocabulary = set of types

$|V|$  is the size of the vocabulary

# Tokenization requires many individual decisions!

- Finland's capital → *Finland* 's or *Finland's* ?
- isn't → *is not* or *isn't* or *is n't* ?
- Hewlett-Packard → *Hewlett Packard* or *Hewlett-Packard* or  
*Hewlett- Packard* or *Hewlett -Packard* ?
- San Francisco → one token or two?
- a.m., Ph.D. → ??

# Some of these decisions are language-specific:

## Contractions

- *L'ensemble* → one token or two?
  - *L* ? *L'* ? *Le* ?
  - Want *L'ensemble* to match with *un ensemble*

## Tokens Not Delineated by Whitespace

- *Lebensversicherungsgesellschaftsangestellter*
  - life insurance company employee
- 莎拉波娃现在居住在美国东南部的佛罗里达。
  - Sharapova now lives in Florida in the southeastern United States.

# Text Normalization

---

- **Normalization:** Manipulating text such that all **forms** of the same word match (e.g., U.S.A. = USA, flavour = flavor, etc.)
- To normalize text, you must define **equivalence classes**
  - Example: “.” characters in a term → not important
- Words with the same characters but different capitalization are often considered equivalent to one another (**case folding**)
  - Example: Hello = hello
  - Not a perfect strategy!
    - US != us
- Useful equivalence classes vary depending on task
  - Capitalization can be very important in sentiment analysis

# Lemmatization

- Reduce inflections or variant forms to base form
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
- *the boy's cars are different colors* → *the boy car be differ color*
- Tricky because you need to find the correct dictionary headword form

# Stemming

- Automatically reduces words to their stems using simple rules
  - language dependent
  - Example: {automate(s), automatic, automation} → automat
- Pros: Very quick, simple to implement
- Cons: Groups together some words that don't really mean the same thing, and doesn't group together some words that do mean the same thing
  - {meanness, meaning} → mean
  - {goose} → goos, {geese} → gees



# Porter Stemming

- Step 1a
  - sses → ss    caresses → caress
  - ies → i        ponies → poni
  - ss → ss        caress → caress
  - s → ø          cats → cat
- Step 1b
  - (\*v\*)ing → ø    walking → walk
  - sing → sing
  - (\*v\*)ed → ø     plastered → plaster
  - ...
- Step 2 (for long stems)
  - ational → ate    relational → relate
  - izer → ize        digitizer → digitize
  - ator → ate        operator → operate
  - ...
- Step 3 (for longer stems)
  - al → ø            revival → reviv
  - able → ø        adjustable → adjust
  - ate → ø            activate → activ
  - ...

Much like tokenization, stemming methods generally need to be customized for specific languages!

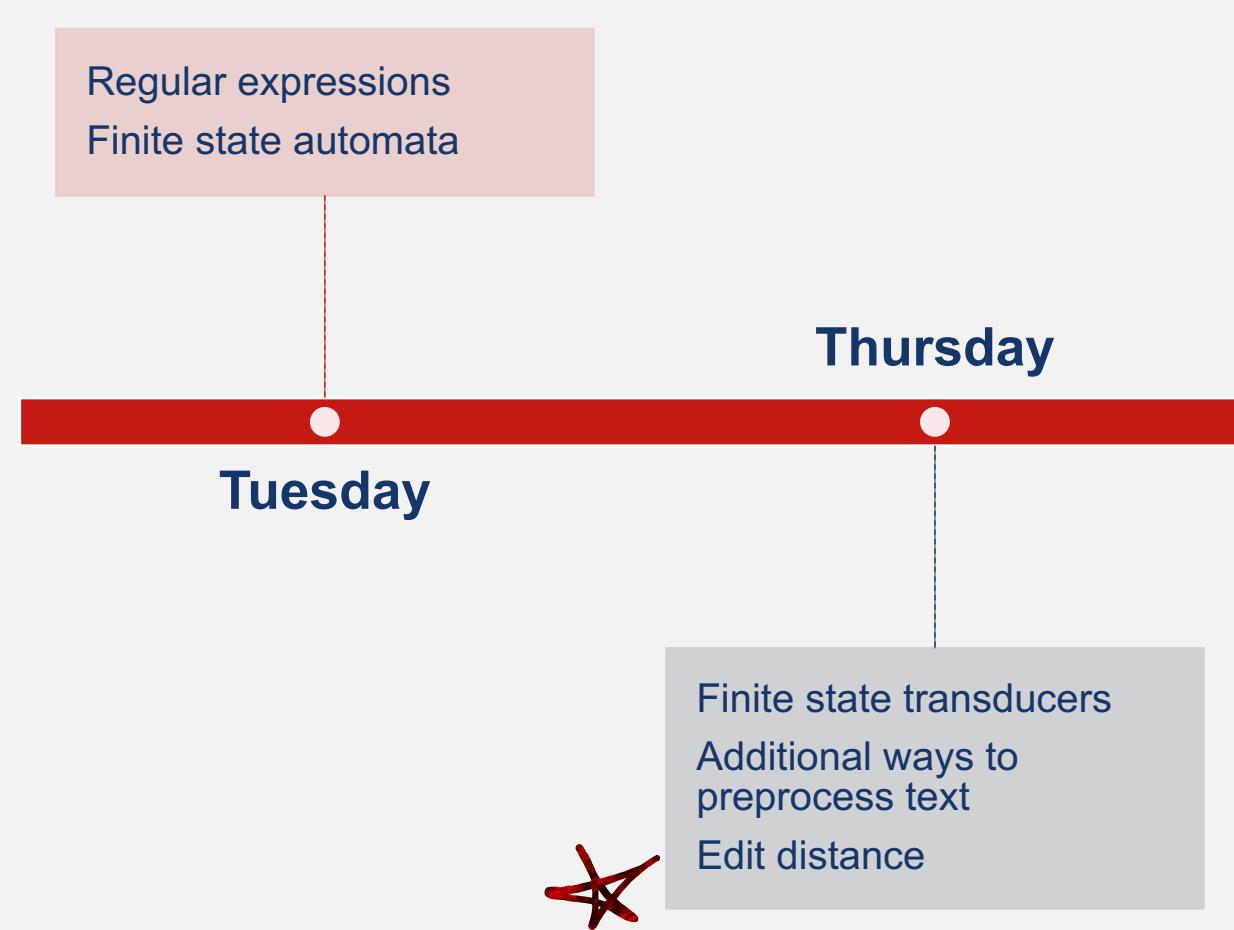
# Sentence Segmentation

---



- !, ? are relatively unambiguous
- . is more ambiguous
  - Sentence boundary
  - Abbreviations like Inc. or Dr.
  - Numbers like .02% or 4.3

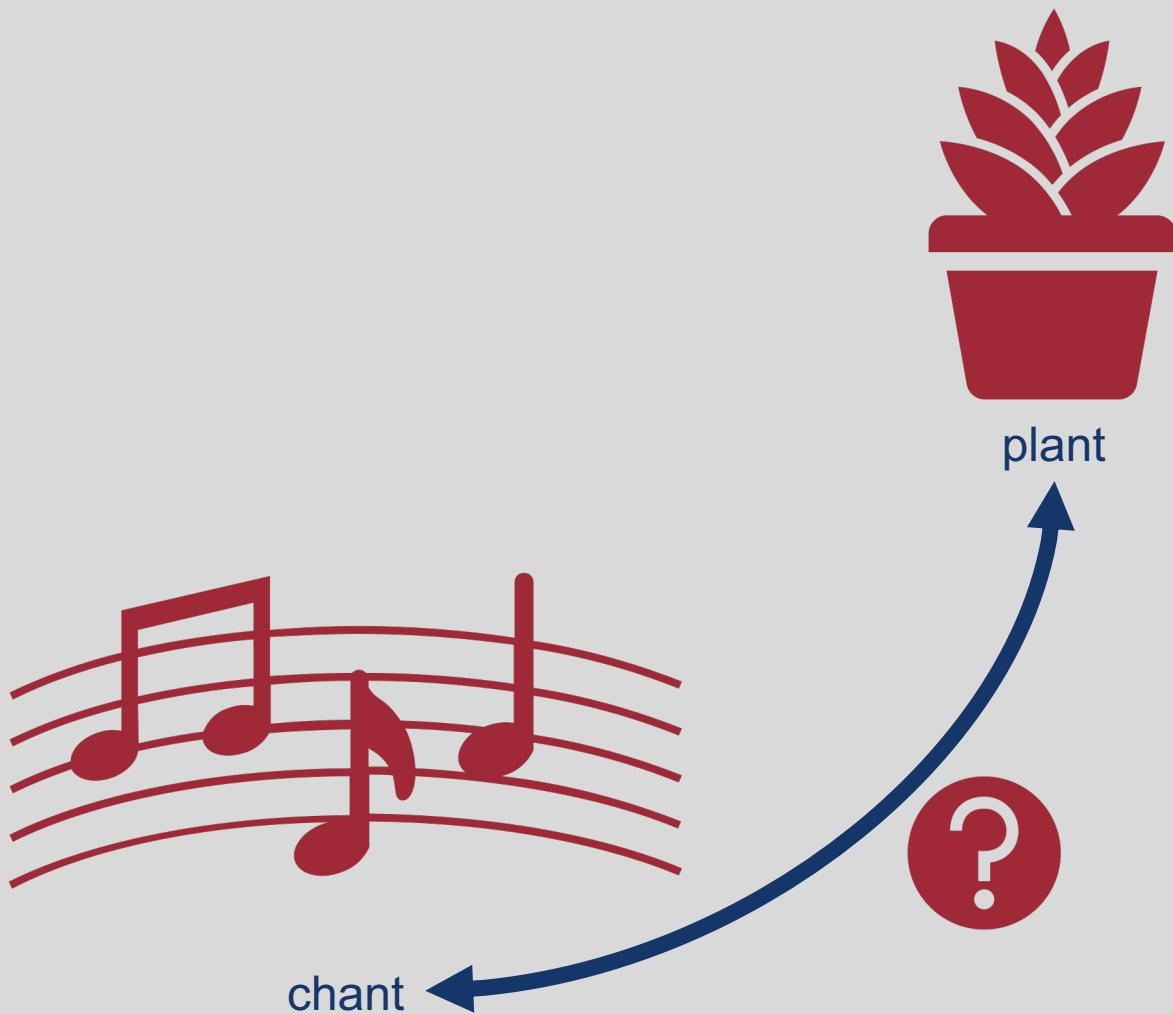
# This Week's Topics



# **We know how to preprocess strings now ...but how can we find the distance between them?**

Popular string (or other sequence) comparison technique:

- Minimum edit distance



# Edit Distance

- Simple way to answer the question: How similar are two strings?
  - Note: This distance is not necessarily associated with **semantic** distance

# Other Uses of Edit Distance in NLP

- Evaluating machine translation and speech recognition using **word error rate**

Spokesman confirms senior government adviser was shot

Spokesman said the senior adviser was shot dead

S

I

D

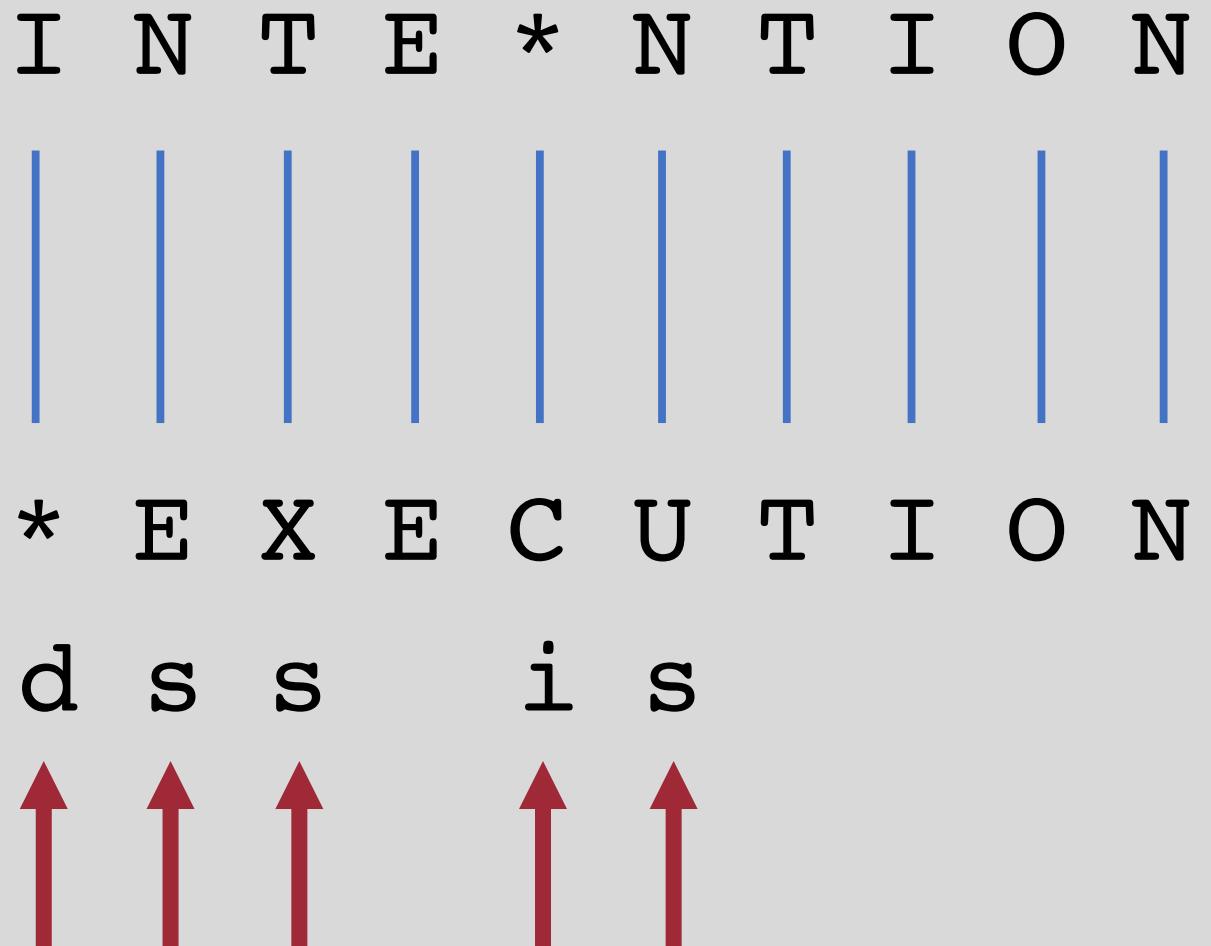
I

# Minimum Edit Distance

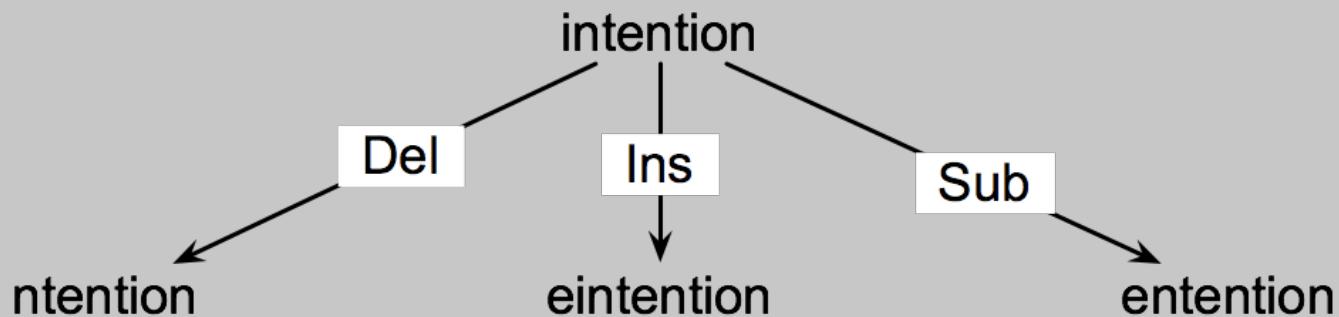
- Minimum number of editing operations needed to transform one string into another
- Possible editing operations:
  - Insertion
  - Deletion
  - Substitution

# Minimum Edit Distance

- Assuming we have the two aligned strings on the right:
  - If each operation has a cost of 1 (Levenshtein distance)
    - Distance between these is 5
  - If substitutions cost 2 (alternative also proposed by Levenshtein)
    - Distance between them is 8



# How to find the minimum edit distance?



- Search for a path (sequence of edits) from the start string to the final string:
  - **Initial state:** the word we're transforming
  - **Operators:** insert, delete, substitute
  - **Goal state:** the word we're trying to get to
  - **Path cost:** what we want to minimize (the number of edits)

- +
- 
- 

# However, the search space of all edit sequences is huge!

- We can't afford to navigate naïvely
- Lots of distinct paths wind up at the same state
  - We don't have to keep track of all of them (just the shortest paths)

# Formal Definition: Minimum Edit Distance

- For two strings
  - $X$  of length  $n$
  - $Y$  of length  $m$
- We define  $D(i,j)$  as the edit distance between  $X[1..i]$  and  $Y[1..j]$ 
  - $X[1..i]$  = the first  $i$  characters of  $X$
- The edit distance between  $X$  and  $Y$  is thus  $D(n,m)$

# Intuition: Dynamic Programming

- Minimum edit distance can be solved using **dynamic programming**
  - Stores intermediate outputs in a table
  - Intuition: If some string B is in the optimal path from string A to string C, then that path must also include the optimal path from A to B
- $D(n,m)$  is computed tabularly, combining solutions to subproblems
- Bottom-up
  - We compute  $D(i,j)$  for small  $i,j$
  - And compute larger  $D(i,j)$  based on previously computed smaller values
    - i.e., compute  $D(i,j)$  for all  $i$  ( $0 < i < n$ ) and  $j$  ( $0 < j < m$ )

# Algorithmically, this looks like:

- Initialization

$$D(i, 0) = i$$

$$D(0, j) = j$$

- Algorithm:

For each  $i = 1 \dots n$

    For each  $j = 1 \dots m$

$$D(i, j) = \min \left\{ \begin{array}{l} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + 2; \text{ if } X(i) \neq Y(j) \\ 0; \text{ if } X(i) = Y(j) \end{array} \right.$$

Cost for these actions



- Termination:

$D(N, M)$  is distance

# The Edit Distance Table

N											
O											
I											
T											
N											
E											
T											
N											
I											
#											
	#	E	X	E	C	U	T	I	O	N	

# The Edit Distance Table

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	<b>0</b>	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

# The Edit Distance Table

N	9										
O	8										
I	7										
T	6										
N	5										
E	4										
T	3										
N	2										
I	1										
#	0	1	2	3	4	5	6	7	8	9	
	#	E	X	E	C	U	T	I	O	N	

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$



# The Edit Distance Table

N	9										
O	8										
I	7										
T	6										
N	5										
E	4										
T	3										
N	2										
I	1	2									
#	0	1	2	3	4	5	6	7	8	9	
	#	E	X	E	C	U	T	I	O	N	

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$



# The Edit Distance Table

N	9										
O	8										
I	7										
T	6										
N	5										
E	4										
T	3										
N	2										
I	1	2	3	4	5	6	7				
#	<b>0</b>	1	2	3	4	5	6	7	8	9	
	#	E	X	E	C	U	T	I	O	N	

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

# The Edit Distance Table

N	9										
O	8										
I	7										
T	6										
N	5										
E	4										
T	3										
N	2										
I	1	2	3	4	5	6	7	6			
#	<b>0</b>	1	2	3	4	5	6	7	8	9	
	#	E	X	E	C	U	T	I	O	N	

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

# The Edit Distance Table

N	9	8	9	10	11	12	11	10	9	8
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

# Backtrace for Computing Alignments

- We know the minimum edit distance now ...but what is the alignment between the two strings?
- We can figure this out by maintaining a **backtrace**
  - For each new cell, remember where we came from!
    - $D(i-1,j)$  ?
    - $D(i,j-1)$  ?
    - $D(i-1,j-1)$  ?
  - Once we reach the end of the table (upper right corner), we can trace backward using these pointers to figure out the alignment

# The Edit Distance Table

N	9	8	9	10	11	12	11	10	9	8
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

# The Edit Distance Table

N	9	8	9	10	11	12	11	10	9	8
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

# Algorithmically, this looks like:

- Base conditions:

$$D(i, 0) = i$$

$$D(0, j) = j$$

Termination:

$D(N, M)$  is distance

- Algorithm:

For each  $i = 1 \dots n$

For each  $j = 1 \dots m$

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + 2; \begin{cases} \text{if } X(i) \neq Y(j) \\ 0; \begin{cases} \text{if } X(i) = Y(j) \end{cases} \end{cases} \end{cases}$$

$$\text{ptr}(i, j) = \begin{cases} \text{LEFT} \\ \text{DOWN} \\ \text{DIAG} \end{cases}$$

# Summary

- +
- 
- 

- **Text Preprocessing:** Preparing text for downstream language processing tasks
  - Tokenization
  - Normalization
  - Lemmatization
  - Stemming
- Regular expressions are a powerful tool for text preprocessing!
- **Edit Distance:** Determining the similarity between two strings based on the number of insertions, deletions, and substitutions needed to transform one to another
- Minimum edit distance, computed using dynamic programming, allows you to find the smallest number of edits needed to do so.