

# Finite Automata

## Part One

Problem Set Four checkpoint  
due in the box up front.  
Late Problem Set Three's  
also due up front.

# Last Thoughts on First-Order Logic

# An Extremely Important Table

	When is this true?	When is this false?
$\forall x. P(x)$	For any choice of $x$ , $P(x)$	$\exists x. \neg P(x)$
$\exists x. P(x)$	For some choice of $x$ , $P(x)$	$\forall x. \neg P(x)$
$\forall x. \neg P(x)$	For any choice of $x$ , $\neg P(x)$	$\exists x. P(x)$
$\exists x. \neg P(x)$	For some choice of $x$ , $\neg P(x)$	$\forall x. P(x)$

# Negating First-Order Statements

- Use the equivalences

$$\neg \forall x. \varphi \equiv \exists x. \neg \varphi$$

$$\neg \exists x. \varphi \equiv \forall x. \neg \varphi$$

to negate quantifiers.

- Mechanically:
  - Push the negation across the quantifier.
  - Change the quantifier from  $\forall$  to  $\exists$  or vice-versa.
- Use techniques from propositional logic to negate connectives.

# Analyzing Relations

“ $R$  is a binary relation over set  $A$  that is not reflexive”

$$\neg \forall a \in A. aRa$$

$$\exists a \in A. \neg aRa$$

“Some  $a \in A$  is not related to itself by  $R$ .”

# Analyzing Relations

“ $R$  is a binary relation over  $A$  that is not antisymmetric”

$$\neg \forall x \in A. \forall y \in A. (xRy \wedge yRx \rightarrow x = y)$$

$$\exists x \in A. \neg \forall y \in A. (xRy \wedge yRx \rightarrow x = y)$$

$$\exists x \in A. \exists y \in A. \neg (xRy \wedge yRx \rightarrow x = y)$$

$$\exists x \in A. \exists y \in A. (xRy \wedge yRx \wedge \neg (x = y))$$

$$\exists x \in A. \exists y \in A. (xRy \wedge yRx \wedge x \neq y)$$

“Some  $x \in A$  and  $y \in A$  are related to one another by  $R$ , but are not equal”

# A Useful Equivalence

- The following equivalences are useful when negating statements in first-order logic:

$$\neg(p \wedge q) \equiv p \rightarrow \neg q$$

$$\neg(p \rightarrow q) \equiv p \wedge \neg q$$

- These identities are useful when negating statements involving quantifiers.
  - $\wedge$  is used in existentially-quantified statements.
  - $\rightarrow$  is used in universally-quantified statements.

# Negating Quantifiers

- What is the negation of the following statement?

$$\exists x. (\textit{Puppy}(x) \wedge \textit{Cute}(x))$$

- We can obtain it as follows:

$$\neg \exists x. (\textit{Puppy}(x) \wedge \textit{Cute}(x))$$

$$\forall x. \neg (\textit{Puppy}(x) \wedge \textit{Cute}(x))$$

$$\forall x. (\textit{Puppy}(x) \rightarrow \neg \textit{Cute}(x))$$

- “All puppies are not cute.”

# Uniqueness

# Uniqueness

- Often, statements have the form “there is a *unique*  $x$  such that ...”
- Some sources use a **uniqueness quantifier** to express this:

$$\exists! n. P(n)$$

- However, it's possible to encode uniqueness using just the two quantifiers we've seen.

$$\exists! n. P(n) \equiv \exists n. (P(n) \wedge \forall m. (P(m) \rightarrow m = n))$$

There is some  $n$   
where  $P(n)$  is true

And whenever  $P$  is  
true, it must be for  $n$ .

# Uniqueness

- Often, statements have the form “there is a *unique*  $x$  such that ...”
- Some sources use a **uniqueness quantifier** to express this:

$$\exists! n. P(n)$$

- However, it's possible to encode uniqueness using just the two quantifiers we've seen.

$$\exists! n. P(n) \equiv \exists n. (P(n) \wedge \forall m. (P(m) \rightarrow m = n))$$

- In CS103, do not use the  $\exists!$  quantifier. Just use  $\exists$  and  $\forall$ .

# Summary of First-Order Logic

- Predicates allow us to reason about different properties of the same object.
- Functions allow us to transform objects into one another.
- Quantifiers allow us to reason about properties of some or all objects.
- There are many useful identities for negating first-order formulae.

# **An Important Milestone**

# Recap: **Discrete Mathematics**

- The past four weeks have focused exclusively on discrete mathematics:

Induction

Functions

Graphs

The Pigeonhole Principle

Relations

Logic

Set Theory

Cardinality

- These are the building blocks we will use throughout the rest of the quarter.
- These are the building blocks you will use throughout the rest of your CS career.


# Next Up: **Computability Theory**

- It's time to switch gears and address the limits of what can be computed.
- We'll explore
  - What is the formal definition of a computer?
  - What might computers look like with various resource constraints?
  - What problems can be solved by computers?
  - What problems ***can't*** be solved by computers?
- **Get ready to explore the boundaries of what computers could ever be made to do.**

# Computability Theory

What problems can we solve with a computer?

What kind of  
computer?




An **automaton** (plural: **automata**) is a mathematical model of a computing device.

# Why Build Models?

- The models of computation we will explore in this class correspond to different conceptions of what a computer could do.
- **Finite automata** (this week) are an abstraction of computers with finite resource constraints.
  - Provide upper bounds for the computing machines that we can actually build.
- **Turing machines** (later) are an abstraction of computers with unbounded resources.
  - Provide upper bounds for what we could ever hope to accomplish.

What problems can we solve with a computer?

What is a  
"problem?"



# Strings

- An **alphabet** is a finite set of **characters**.
  - Typically, we use the symbol  $\Sigma$  to refer to an alphabet.
- A **string** is a finite sequence of characters drawn from some alphabet.
- Example: If  $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ , some valid strings include
  - **a**
  - **aabaaabbabaaabaaaabbbb**
  - **abbababba**
- The **empty string** contains no characters and is denoted  $\epsilon$ .

# Languages

- A **formal language** is a set of strings.
- We say that  $L$  is a **language over  $\Sigma$**  if it is a set of strings formed from characters in  $\Sigma$ .
- Example: The language of palindromes over  $\Sigma = \{a, b, c\}$  is the set  
$$\{\epsilon, a, b, c, aa, bb, cc, aaa, aba, aca, bab, \dots\}$$
- The set of all strings composed from letters in  $\Sigma$  is denoted  $\Sigma^*$ .
- Formally:  $L$  is a language over  $\Sigma$  iff  $L \subseteq \Sigma^*$ .

# The Model

- **Goal:** Given an alphabet  $\Sigma$  and a language  $L$  over  $\Sigma$ , can we build an automaton that can determine which strings are in  $L$ ?
- Actually a very expressive and powerful framework; we'll see this over the next few weeks.

# To Summarize

- An **automaton** is an idealized mathematical computing machine.
- A **language** is a set of strings.
- The automata we will study will accept as input a string and (attempt to) output whether that string is contained in a particular language.

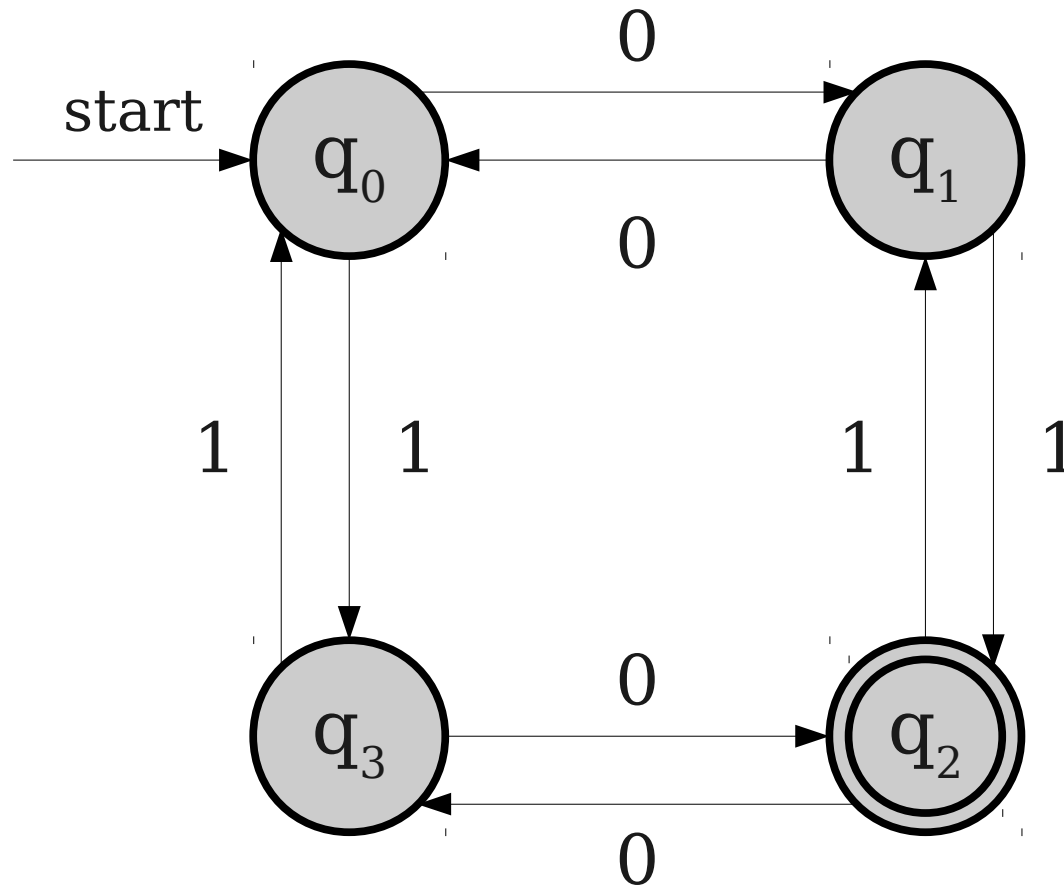
What problems can we solve with a computer?

# Finite Automata

A **finite automaton** is a mathematical machine for determining whether a string is contained within some language.

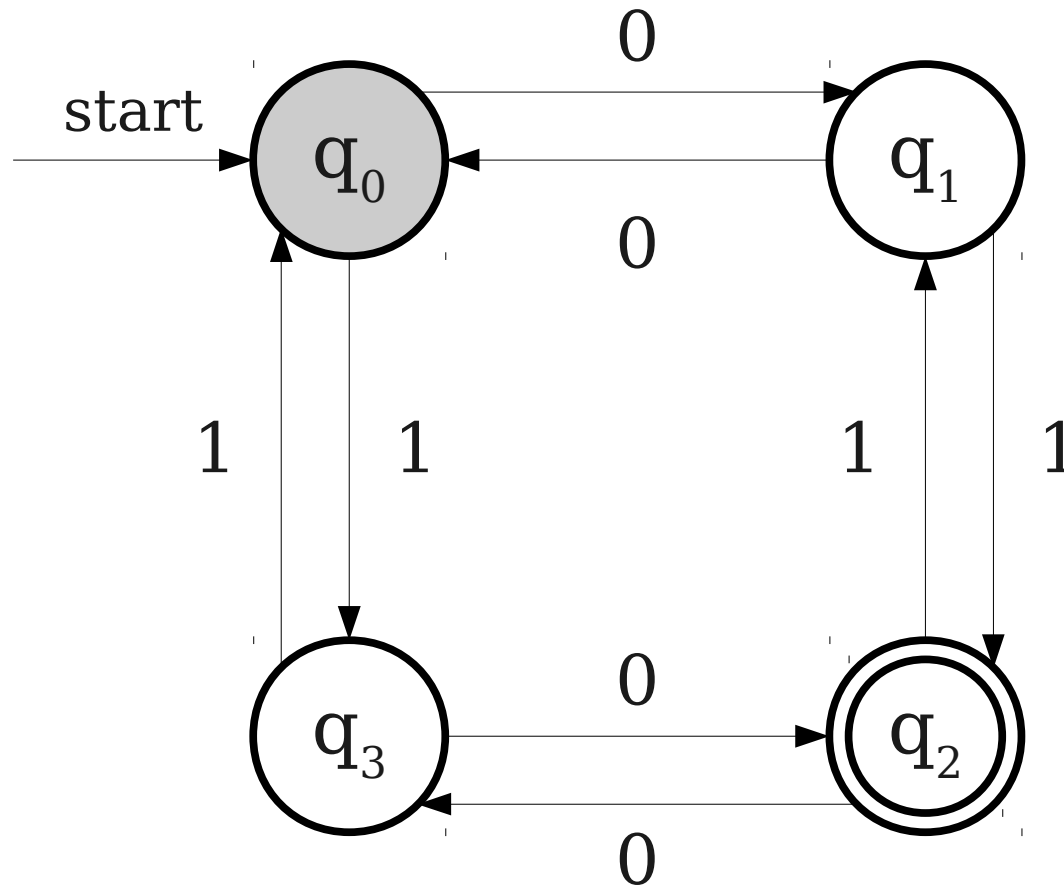
Each finite automaton consists of a set of **states** connected by **transitions**.

# A Simple Finite Automaton



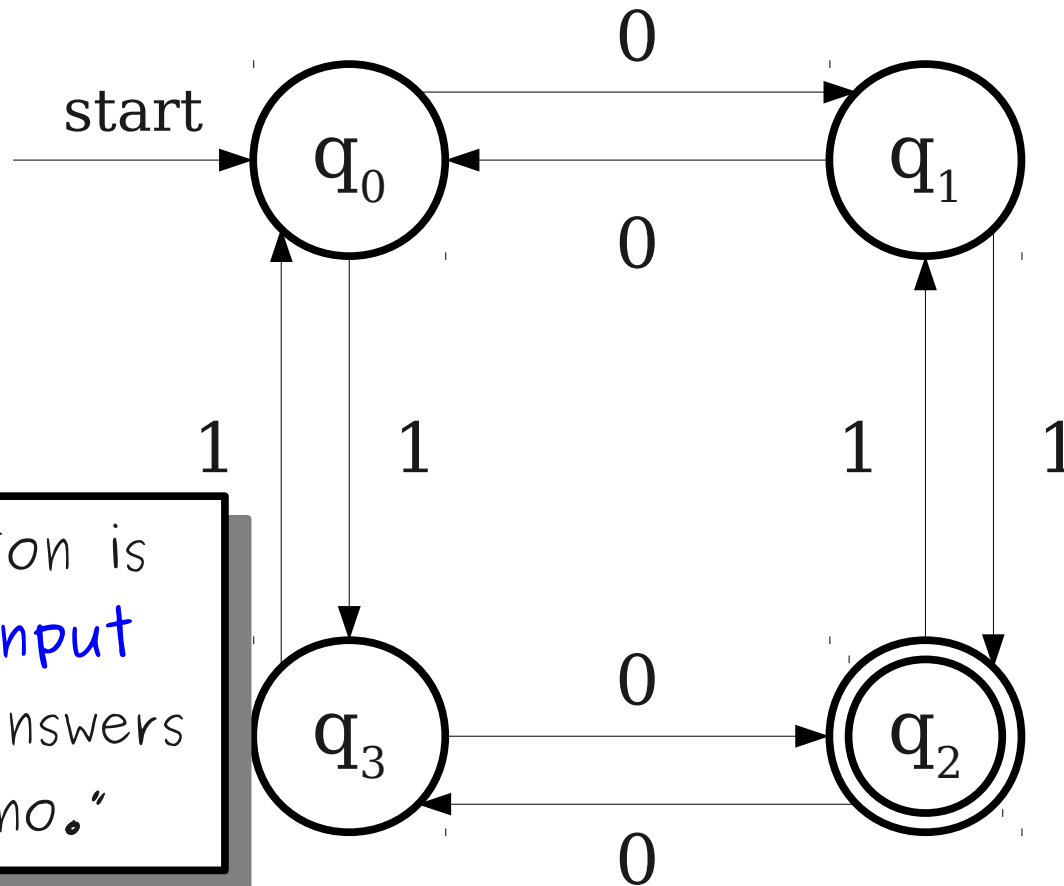
Each circle  
represents a **state**  
of the automaton.

# A Simple Finite Automaton



One special state is  
designated as the  
*start state*.

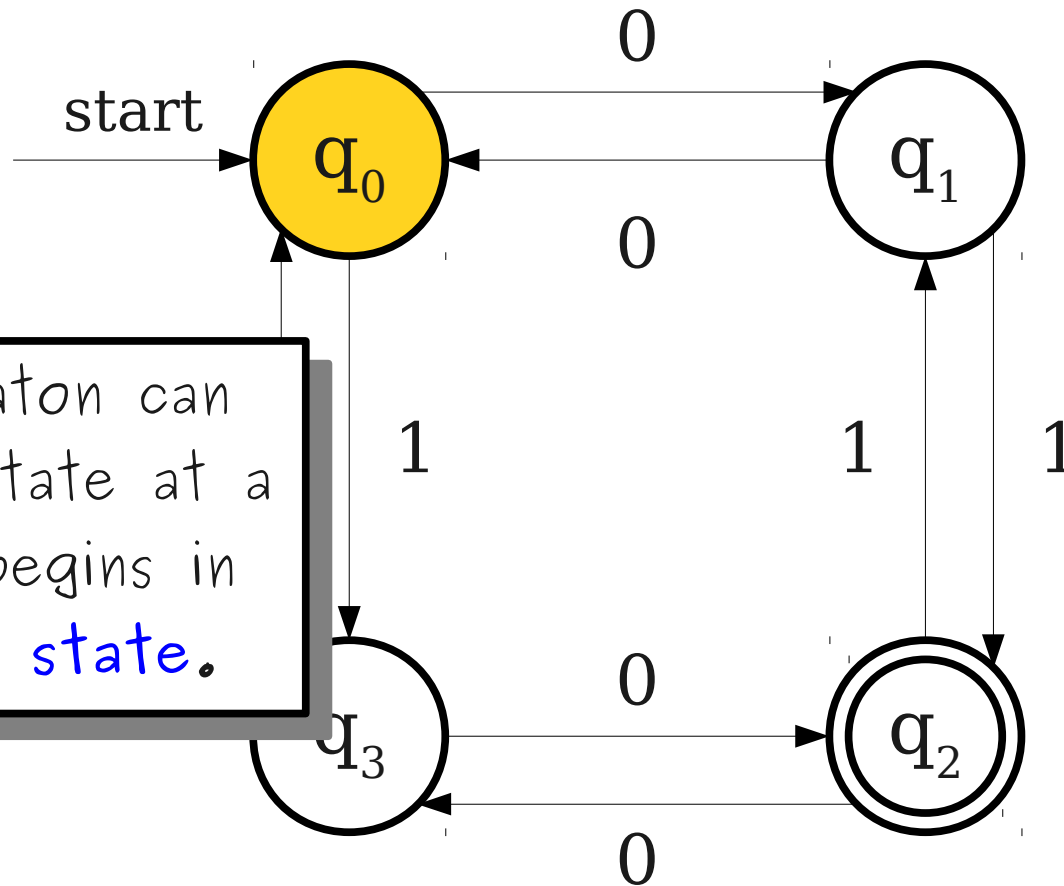
# A Simple Finite Automaton



The automaton is run on an **input string** and answers "yes" or "no."

**0 1 0 1 1 0**

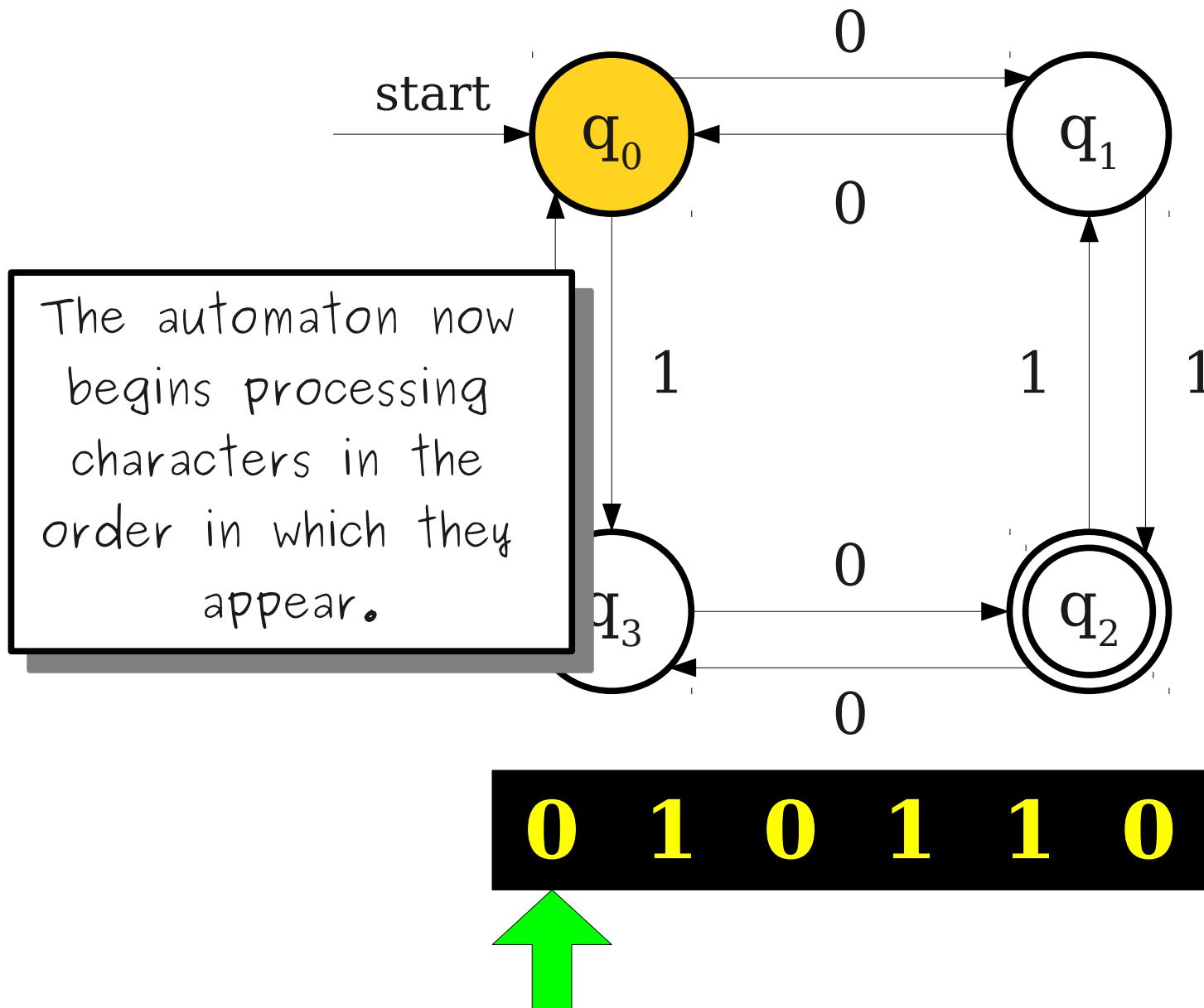
# A Simple Finite Automaton



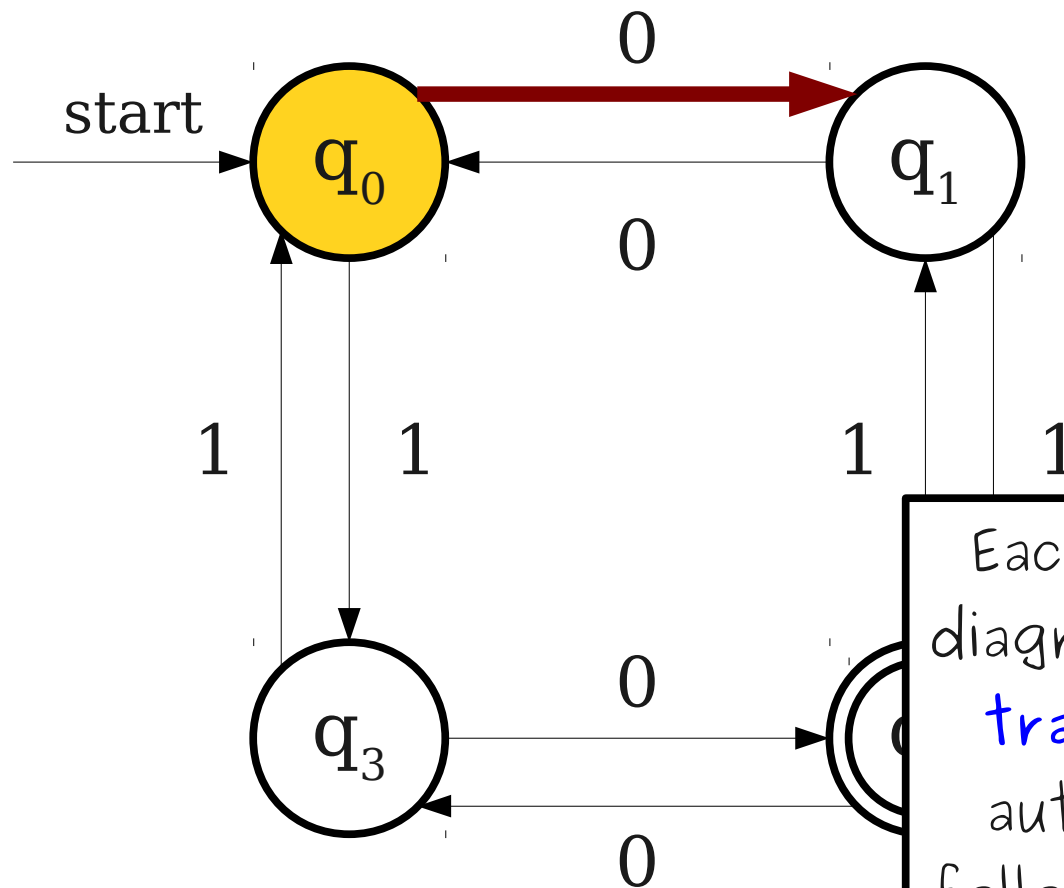
The automaton can be in one state at a time. It begins in the **start state**.

**0 1 0 1 1 0**

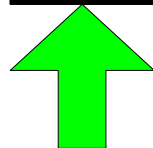
# A Simple Finite Automaton



# A Simple Finite Automaton

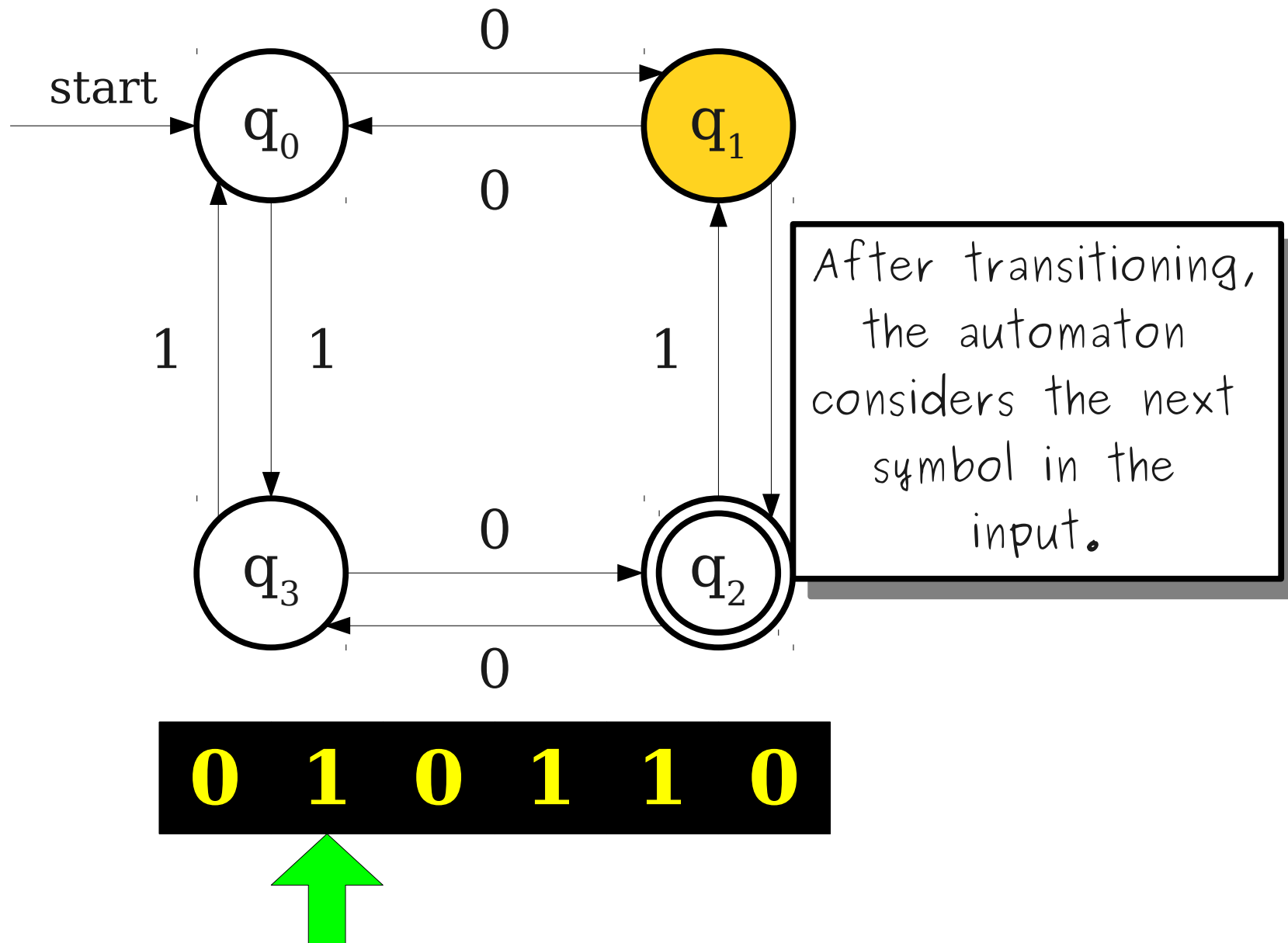


**0 1 0 1 1**

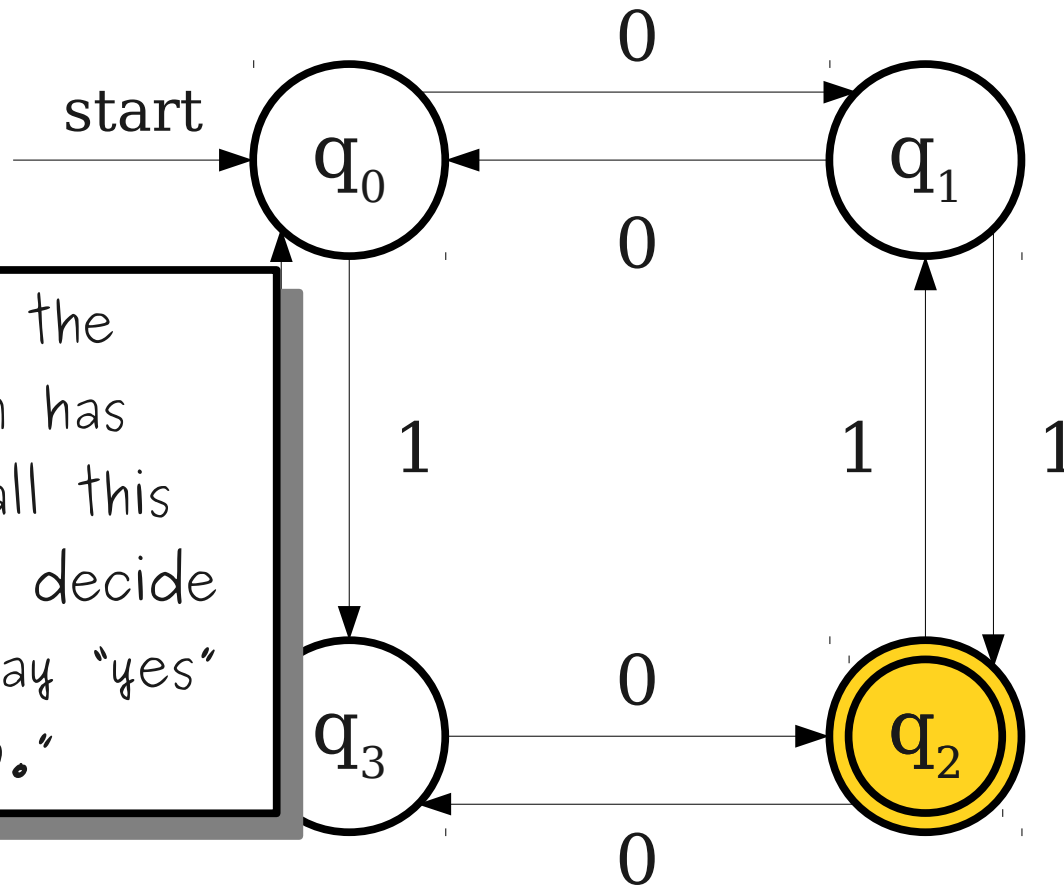


Each arrow in this diagram represents a **transition**. The automaton always follows the transition corresponding to the current symbol being read.

# A Simple Finite Automaton



# A Simple Finite Automaton

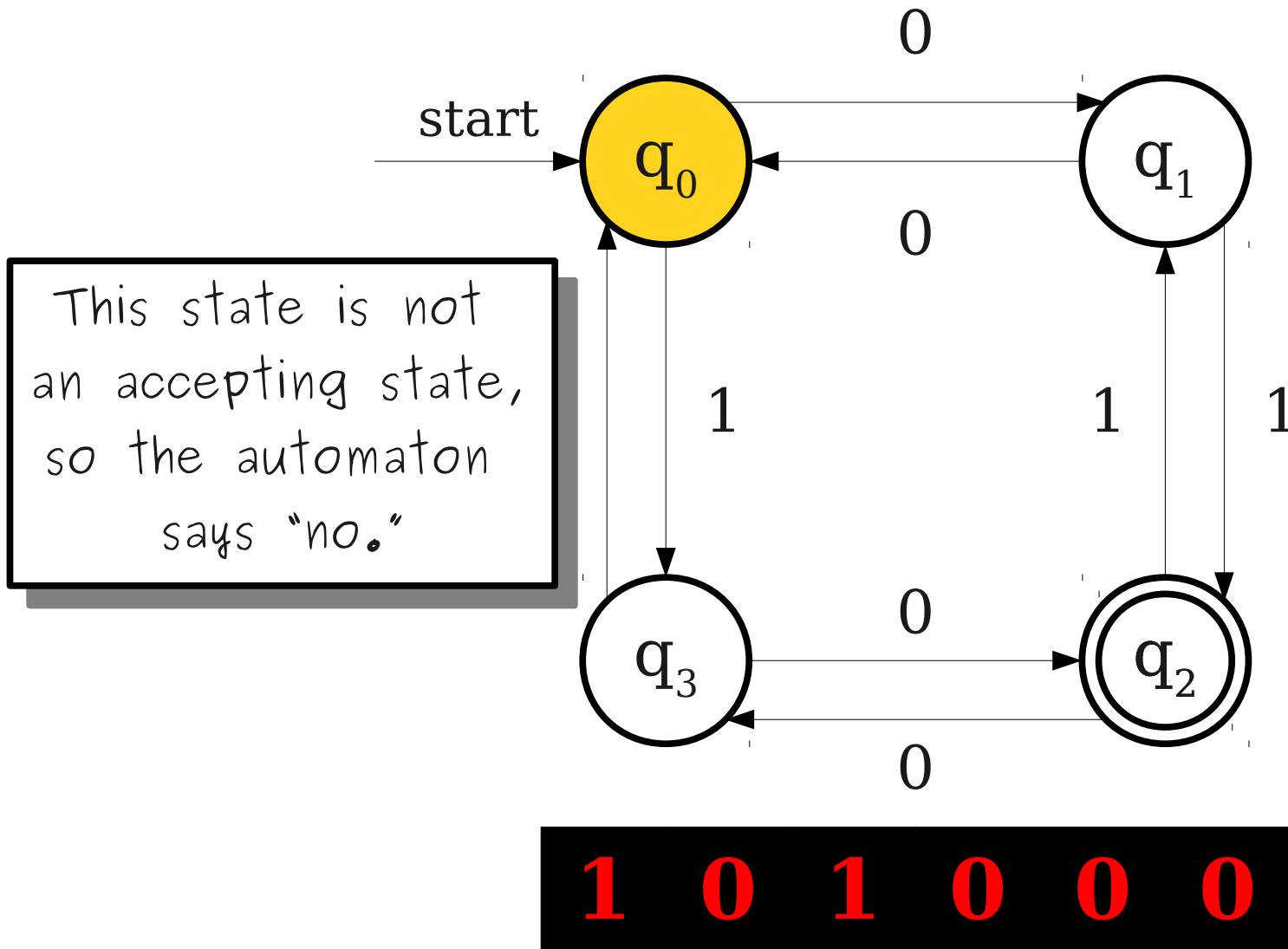


Now that the automaton has looked at all this input, it can decide whether to say "yes" or "no."

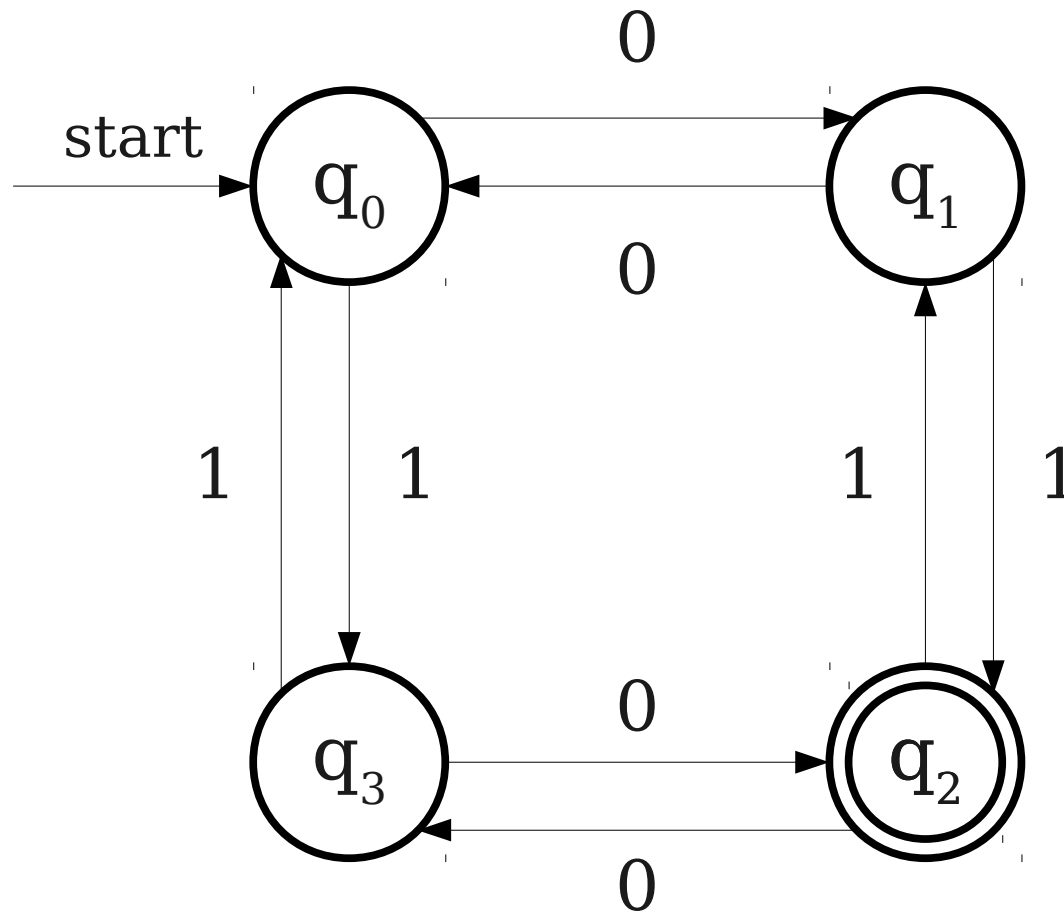
The double circle indicates that this state is an **accepting state**, so the automaton outputs "yes."

**0 1 0 1 1 0**

# A Simple Finite Automaton



# A Simple Finite Automaton



Try it yourself!  
Does the  
automaton **accept**  
(say yes) or  
**reject** (say no)?

**1 1 0 1 1 1 0 0**

# The Story So Far

- A **finite automaton** is a collection of **states** joined by **transitions**.
- Some state is designated as the **start state**.
- Some states are designated as **accepting states**.
- The automaton processes a string by beginning in the start state and following the indicated transitions.
- If the automaton ends in an accepting state, it **accepts** the input.
- Otherwise, the automaton **rejects** the input.

Time-Out For Announcements

# Midterm Exam

- Midterm is October 29 from 7PM – 10PM.
  - Location information TBA.
  - Need to take the exam at an alternate time? Fill out the alternate time form ASAP.
  - Covers material up through and including DFAs.
- Two practice exams available online.
- Handout: Exam Strategies and Policies available online.
- Review session later this week; more details on Wednesday.

Your Questions

“Keith, why did you choose to go into academia as opposed to industry? Do you have any advice for students deciding between the two?”

“What do you think is the coolest  
real-world application of CS103-related  
material?”

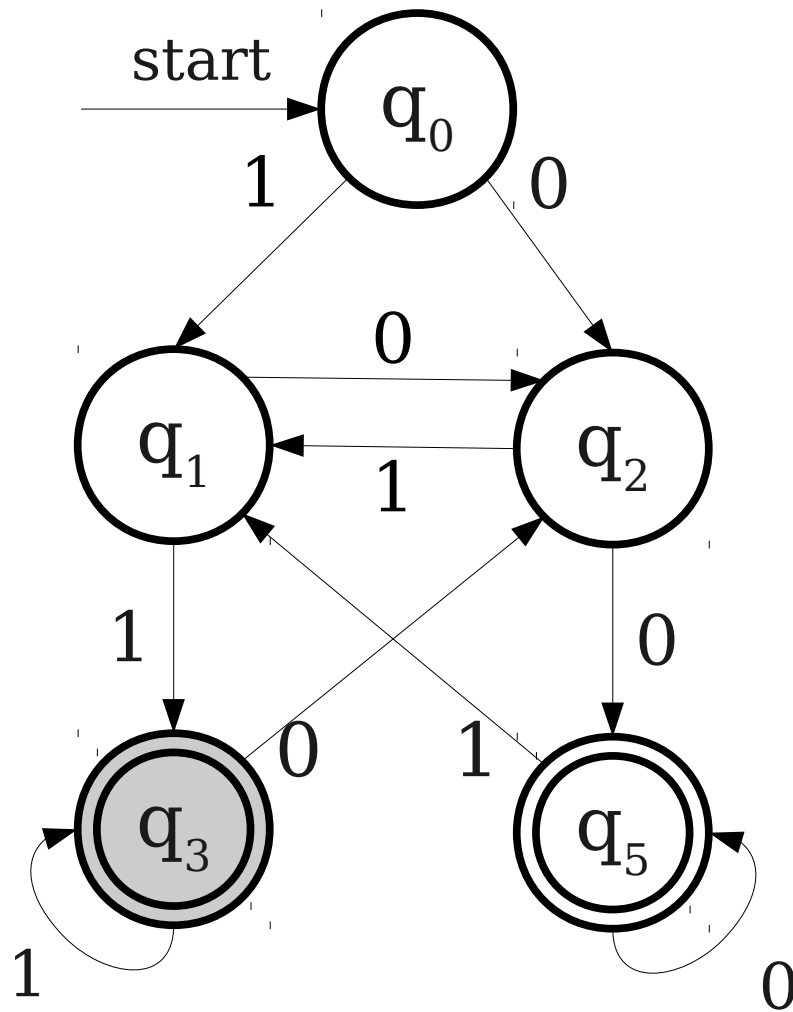
“Why are you taking away the Checkpoint problems... don't?”

Back to CS103!

A finite automaton does **not** accept as soon as the input enters an accepting state.

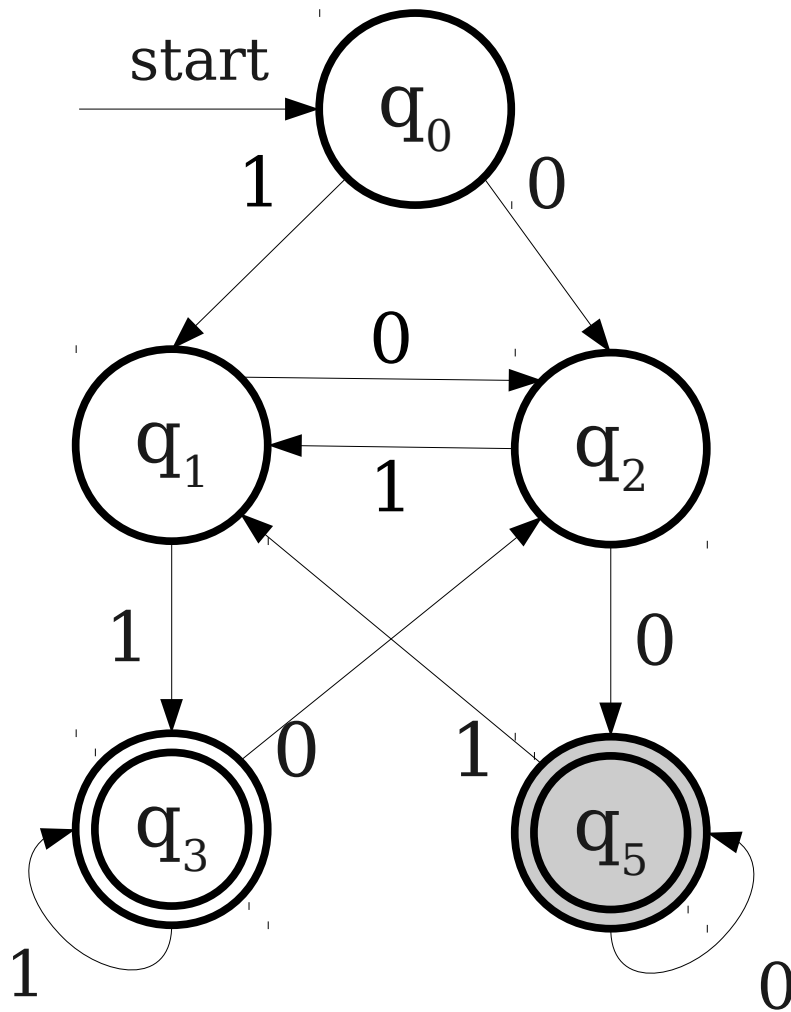
A finite automaton accepts if it **ends** in an accepting state.

# What Does This Accept?



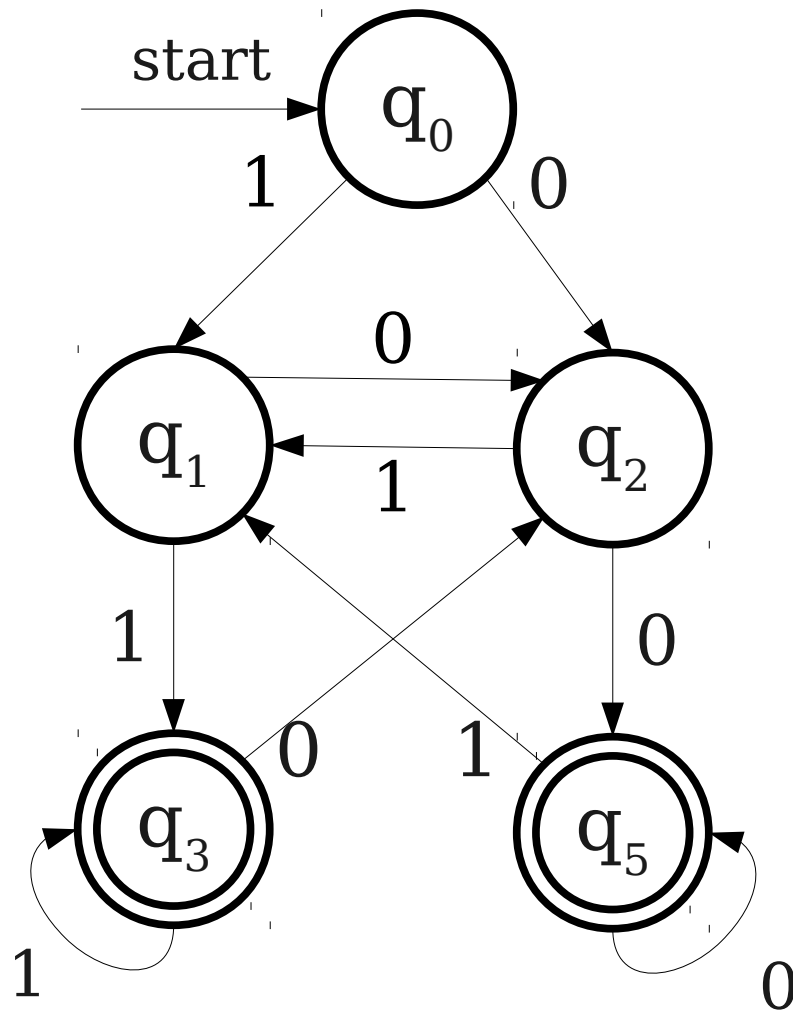
No matter where we start in the automaton, after seeing two 1's, we end up in accepting state  $q_3$ .

# What Does This Accept?



No matter where we start in the automaton, after seeing two 0's, we end up in accepting state  $q_5$ .

# What Does This Accept?



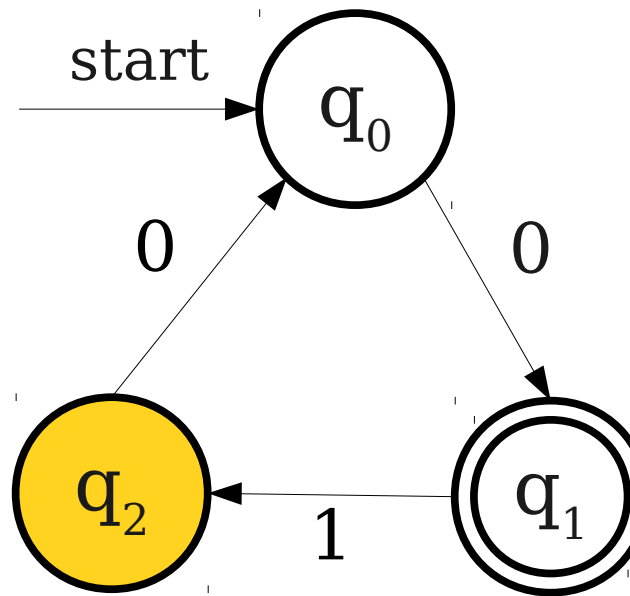
This automaton  
accepts a string iff  
it ends in 00 or 11.

The **language of an automaton** is the set of strings that it accepts.

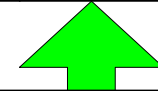
If  $D$  is an automaton, we denote the language of  $D$  as  $\mathcal{L}(D)$ .

$$\mathcal{L}(D) = \{ w \in \Sigma^* \mid D \text{ accepts } w \}$$

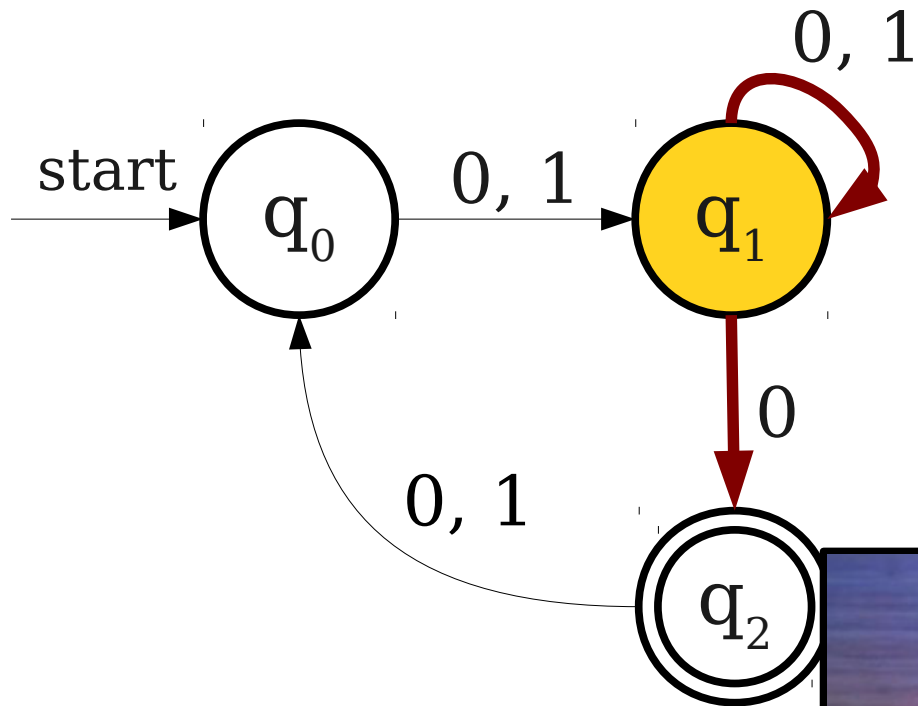
# A Small Problem



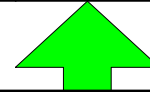
**0 1 1 0**



# Another Small Problem



**0 0 0**



# The Need for Formalism

- In order to reason about the limits of what finite automata can and cannot do, we need to formally specify their behavior in *all* cases.
- All of the following need to be defined or disallowed:
  - What happens if there is no transition out of a state on some input?
  - What happens if there are *multiple* transitions out of a state on some input?

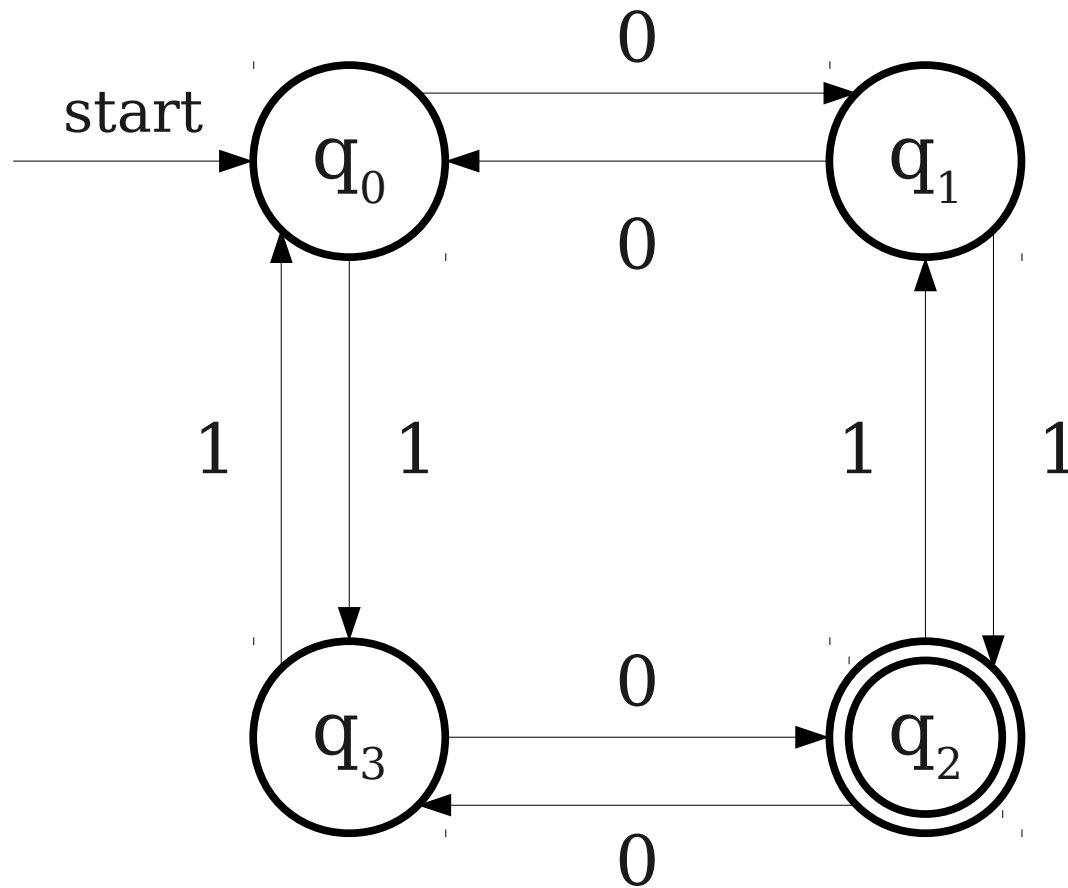
# DFAs

- A **DFA** is a
  - **D**eterministic
  - **F**inite
  - **A**utomaton
- DFAs are the simplest type of automaton that we will see in this course.

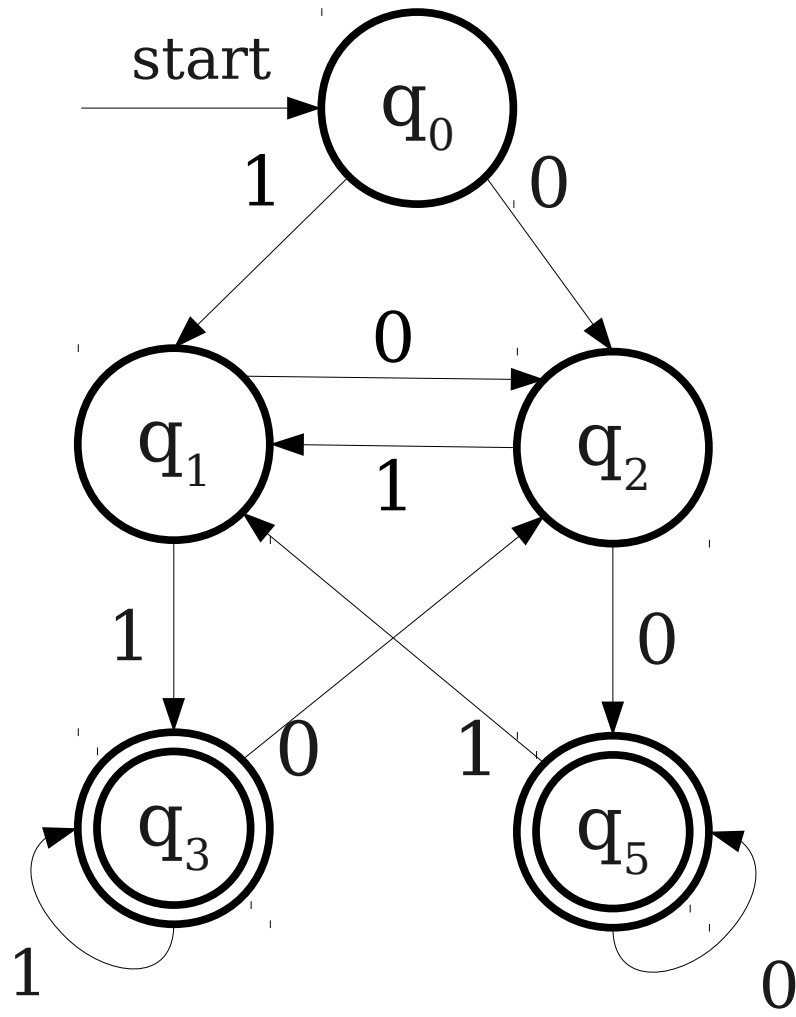
# DFA's, Informally

- A DFA is defined relative to some alphabet  $\Sigma$ .
- For each state in the DFA, there must be **exactly one** transition defined for each symbol in the alphabet.
  - This is the “deterministic” part of DFA.
- There is a **unique** start state.
- There are zero or more accepting states.

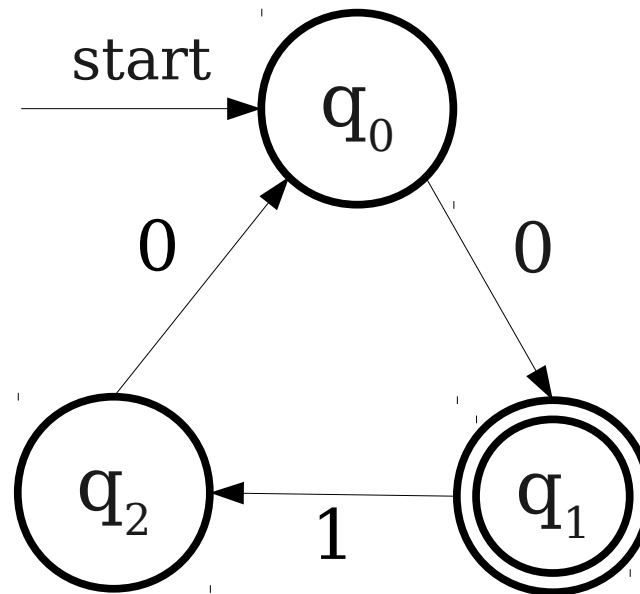
# Is this a DFA?



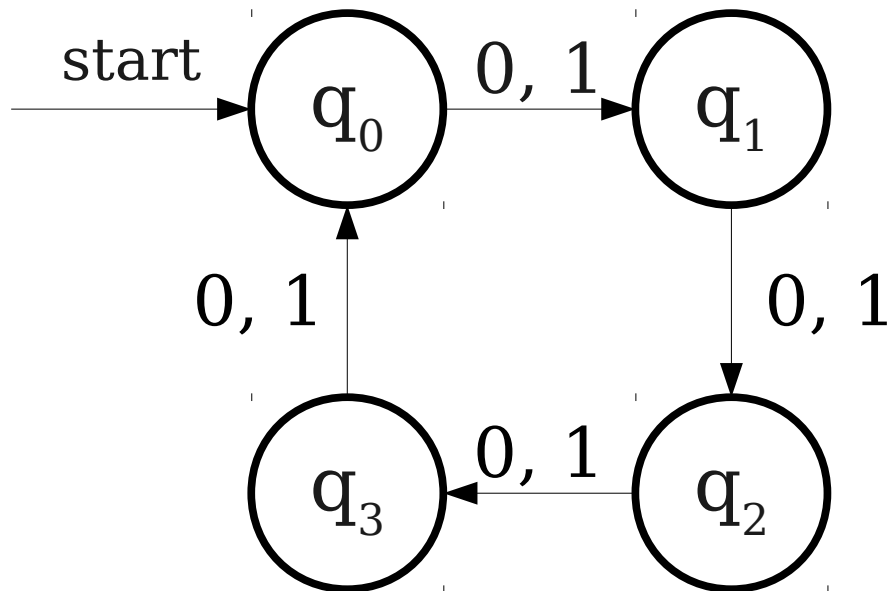
# Is this a DFA?



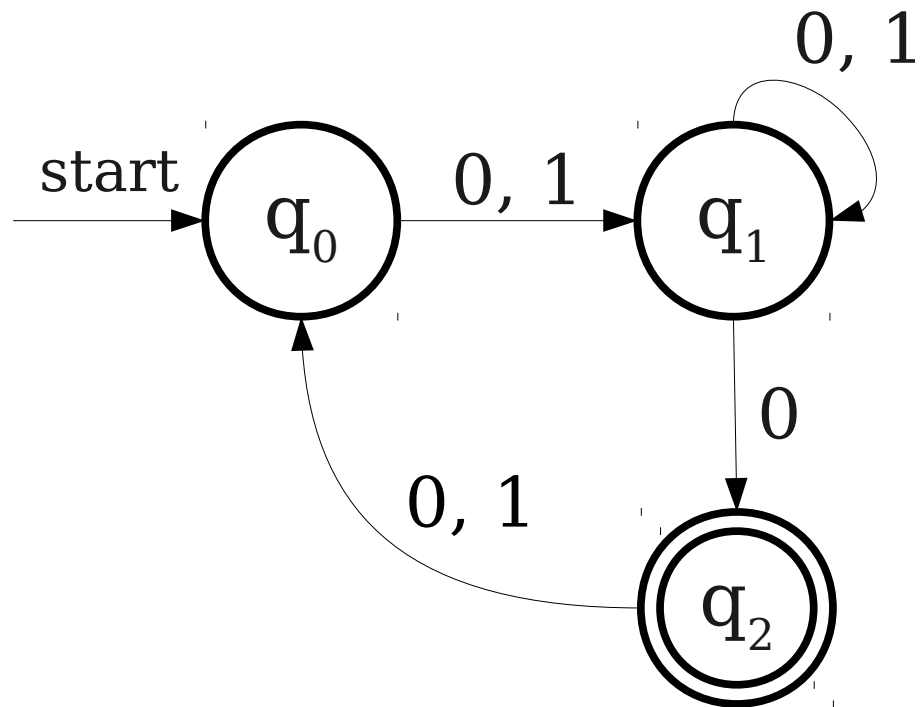
# Is this a DFA?



# Is this a DFA?



# Is this a DFA?

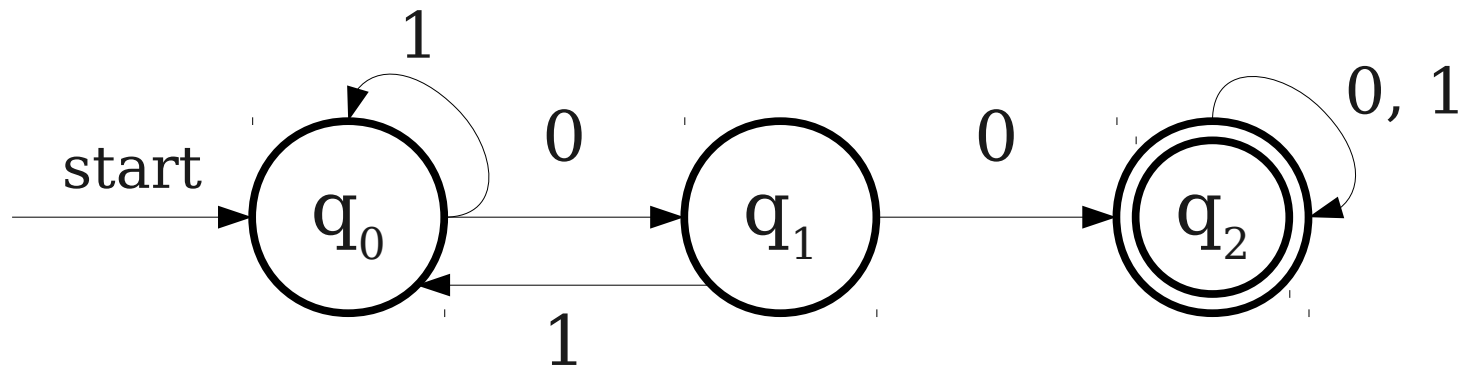


# Designing DFAs

- At each point in its execution, the DFA can only remember what state it is in.
- **DFA Design Tip:** Build each state to correspond to some piece of information you need to remember.
  - Each state acts as a “memento” of what you're supposed to do next.
  - Only finitely many different states  $\approx$  only finitely many different things the machine can remember.

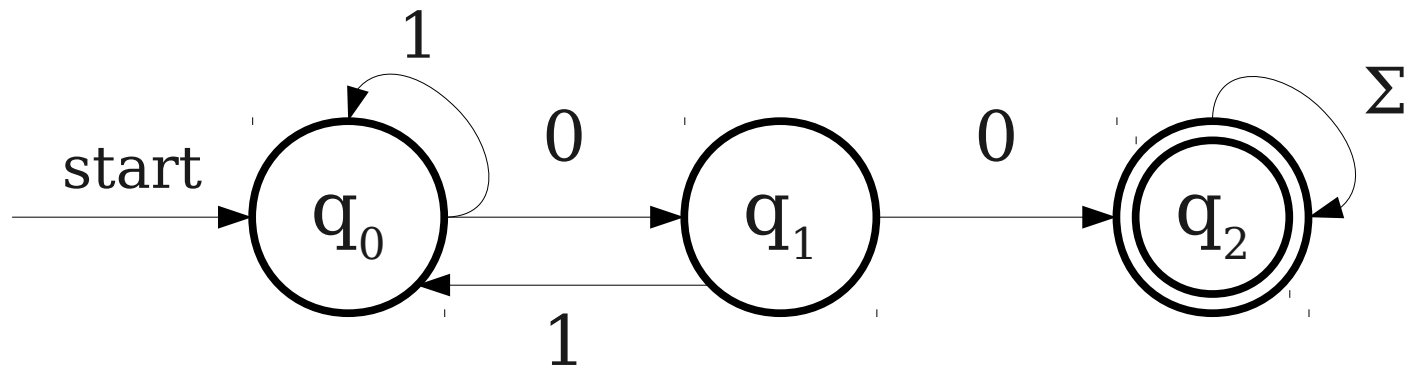
# Recognizing Languages with DFAs

$L = \{ w \in \{0, 1\}^* \mid w \text{ contains } 00 \text{ as a substring} \}$



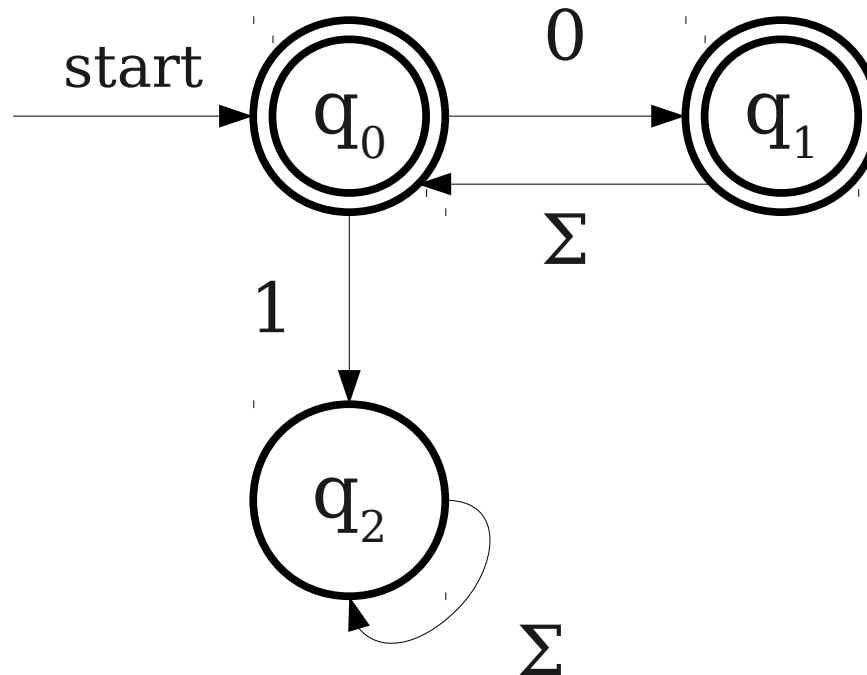
# Recognizing Languages with DFAs

$L = \{ w \in \{0, 1\}^* \mid w \text{ contains } 00 \text{ as a substring} \}$



# Recognizing Languages with DFAs

$L = \{ w \in \{0, 1\}^* \mid \text{every other character of } w, \text{ starting with the first character, is } 0 \}$



# Next Time

- **Regular Languages**
  - What is the expressive power of DFAs?
- **NFAs**
  - Automata with magic superpowers!
- **Nondeterminism**
  - Nondeterministic computation.
  - Intuitions for nondeterminism.
  - Programming with nondeterminism.