Finite Automata

CMPU 240 • Language Theory and Computation • Spring 2019

Last class:

- Introduced the idea of a "computer"
- Review of basic concepts
- Basic notation for talking about languages and strings

Today:

- Review proof techniques
- Introduce automata theory
- Begin studying finite automata and regular languages

Proofs

Proofs

The way to show the truth or falsity of a mathematical statement is with a proof.

- Finding proofs isn't always easy!
- Some helpful strategies:
  - Carefully read the statement you want to prove.
  - Rewrite the statement in your own words.
  - Break the statement down and consider each part separately.
  - Reduction to definitions.

Parts of proofs

The parts of a multi-part statement are not always obvious. E.g.,

P **if and only if** *Q* (also, "*P* iff *Q*" or "$P \Leftrightarrow Q$")

Two parts:

Forward direction: If *P* is true, *Q* is true ($P \Rightarrow Q$)

Reverse direction: If *Q* is true, *P* is true ($P \Leftarrow Q$)

To prove an iff statement, we must prove both directions.

Types of proof

Proof by construction

Theorem states that a particular type of object exists.

Prove by demonstrating how to construct such an object.

Proof by contradiction

Assume the theorem is false and show this leads to an obviously false consequence.

Proof by induction

Truth of a number of statements inferred from the truth of a few specific instances.

Proving the equivalence of sets

Many important facts in language theory are of the form that two sets of strings, described in two different ways, are really the same set.

To prove sets *S* and *T* are the same, prove $x \in S$ if and only if $x \in T$:

1. Assume $x \in S$. Now prove $x \in T$.

2. Assume $x \in T$. Now prove $x \in S$.

Example of proof by construction

Claim: *Every pair of integers has a greatest common divisor.*

Proof approach:

Give an algorithm to find the gcd.

This shows not only that the gcd exists, but also that there exists a method to determine what the gcd is for every pair of integers.

This is a stronger claim than we started with, but in some cases stronger claims are easier to prove!

Example of proof by contradiction

Claim: $\sqrt{2}$ *is irrational.*

Proof:

Assuming $\sqrt{2}$ is rational. In that case, it is the quotient of two integers, i and j. So we have $\sqrt{2} = i/j$.

If i and j have any common factors, we reduce them by those factors. So now we have

$\sqrt{2} = k/n$ where k and n have no common factors.

$2 = k^2/n^2$

$2n^2 = k^2$

Since 2 is a factor of k^2, k^2 must be even and so k is even. Since k is even, we can rewrite it as $2m$ for some integer m. Substituting $2m$ for k, we get:

$2n^2 = (2m)^2$

$2n^2 = 4m^2$

$2n^2 = k^2$

$n^2 = 2m^2$

So n^2 is even and thus n is even. Now both k and n are even and so have 2 as a common factor. But we had reduced them until they had no common factors. The assumption that $\sqrt{2}$ is rational has led to a contradiction. Therefore, $\sqrt{2}$ cannot be rational.

Proof by induction

Method to show that some proposition $\mathcal{P}$ is true for all elements of an *infinite set*, e.g., integers ≥ 0.

Requires two sub-proofs:

Basis: Show that $\mathcal{P}(0)$ is true.

Induction step:

Inductive hypothesis: Assume $\mathcal{P}(i)$ is true.

Use logical or mathematical reasoning to show $\mathcal{P}(i + 1)$ is true.

When both parts are proven, we can conclude $\mathcal{P}$(i) is true for all $i \geq 0$.

Proof by structural induction

Like proof by induction but for any recursive definition, not just numbers.

In this class, we'll more often be working with strings than with numbers.

Recursive definition of Σ^*

Σ^*, the set of all strings over some alphabet Σ, consists of the following:

1. The empty string $\varepsilon \in \Sigma^*$
2. If $y \in \Sigma^*$ and $a \in \Sigma$ is any symbol, then the string $y \cdot a \in \Sigma^*$

No string is in Σ^* unless it can be obtained by these two rules.

E.g., when we write the string *abc*, we interpret it as the structure $((\varepsilon \cdot a) \cdot b) \cdot c$

Recursive definition of *reverse*

The reverse function, *reverse*(*x*), is *x* "read backwards", e.g., *reverse*(*abc*) = *cba*

Formally, define the function *reverse*: $\Sigma^* \rightarrow \Sigma^*$ as follows:

Base case: $reverse(\varepsilon) = \varepsilon$

Recursive case:

Assume that for a given $y \in \Sigma^*$, we have already defined *reverse*(*y*).

Now for each $a \in \Sigma$, define $reverse(y \cdot a) = a \cdot reverse(y)$

This is an example of using the structure of the recursive definition of Σ^* to define a function whose domain is Σ^*.

Example of proof by structural induction

Proposition 1: For any two strings, *x*, *y* over Σ:

$$reverse(x \cdot y) = reverse(y) \cdot reverse(x)$$

Proof. We let $x, y \in \Sigma^*$ and prove by induction on the structure of the string *y* that $reverse(x \cdot y) = reverse(y) \cdot reverse(x)$

Basis: If $y = \varepsilon$ (that is, *y* is the empty string) then

$$\begin{aligned}
reverse(x \cdot y) &= reverse(x \cdot \varepsilon) && \text{(from } y = \varepsilon\text{)} \\
&= reverse(x) && \text{(property of } \cdot\text{)} \\
&= \varepsilon \cdot reverse(x) && \text{(property of } \cdot\text{)} \\
&= \varepsilon\ reverse(\varepsilon) \cdot reverse(x) && \text{(from def. of } reverse\text{)} \\
&= reverse(y) \cdot reverse(x) && \text{(from } y = \varepsilon\text{)}
\end{aligned}$$

Induction step: In this case we have $y = z \cdot a$ for some $z \in \Sigma^*$ and $a \in \Sigma$ (i.e., *y* is ***not*** the empty string).

Inductive hypothesis (IH): $reverse(x \cdot z) = reverse(z) \cdot reverse(x)$, i.e., the property that we want to prove of *y* holds for all "structurally smaller" strings (e.g., the string *z*).

With this assumption, we now prove that the "bigger" string *y* formed from *z* also has the desired property:

$$\begin{aligned}
reverse(x \cdot y) &= reverse(x \cdot (z \cdot a)) && \text{(from } y = z \cdot a\text{)} \\
&= reverse((x \cdot z) \cdot a) && \text{(associativity of } \cdot\text{)} \\
&= a \cdot reverse(x \cdot z) && \text{(from def. of } reverse\text{)} \\
&= a \cdot (reverse(z) \cdot reverse(x)) && \text{(IH)} \\
&= (a \cdot reverse(z)) \cdot reverse(x) && \text{(associativity of } \cdot\text{)} \\
&= reverse(z \cdot a) \cdot reverse(x) && \text{(from def. of } reverse\text{)} \\
&= reverse(y) \cdot reverse(x) && \text{(from } y = z \cdot a\text{)}
\end{aligned}$$

This concludes the proof of Proposition 1.

Modeling computers

A *computational model* is an idealized computer that abstracts away some details but is accurate in others.

Why build models?

- *Mathematical simplicity*: It's significantly easier to manipulate our abstract models of computers than to manipulate actual computers.
- *Intellectual robustness*: If we pick our models correctly, we can make broad, sweeping claims about huge classes of real computers by arguing that they're just special cases of our more general models.

An *abstract machine* reads in an input string, and depending on the input it

- outputs **true** (accept)
- outputs **false** (reject)
- gets stuck in an infinite loop and never outputs anything

We say that a machine *recognizes* a language if it outputs true for every input string in the language and false otherwise.

The artificial restriction to *decision problems* is purely for notational convenience.

Virtually all computational problems can be recast as *language recognition problems*.

Examples:

- To determine whether the integer 97 is prime, ask whether 97 is in the language consisting of all primes, {2, 3, 5, 7, 13, …}
- To determine the decimal expansion of the mathematical constant π, ask whether 7 is the 100th digit of π, and so on.

Fundamental question

Given an alphabet Σ and a language L over Σ, in what cases can we build an automaton that determines which strings are in L?

- The answer depends on both the choice of L and the kind of automaton.

To compare different classes of abstract machines, we define a notion of *power*.

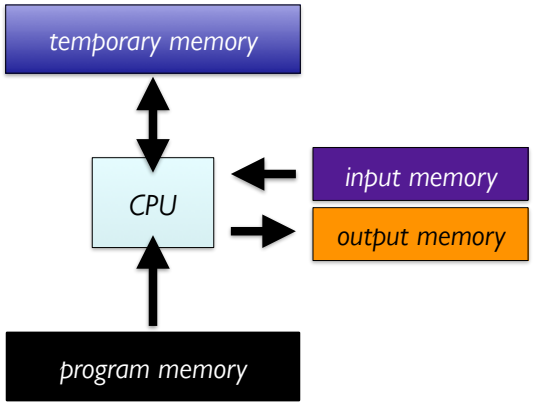
- Machine *A* is at least as powerful as machine *B* if *A* can be programmed to *recognize all of the languages* *B* can.
- Machine *A* is more powerful than *B* if it can be programmed to recognize all the languages *B* can and at least one more.
- Two machines are equivalent if they can be programmed to recognize precisely the same set of languages.

We'll use this definition of power to classify several fundamental machines.

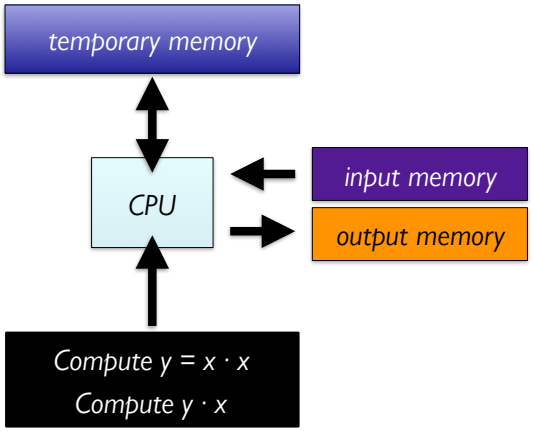
We're interested in designing the most powerful computer, i.e., the one that can solve the widest range of language recognition problems.

- Our notion of power does not say anything about *how fast* a computation can be done.
- It reflects a more fundamental notion of whether or not it is *possible to perform a computation in a finite number of steps*.

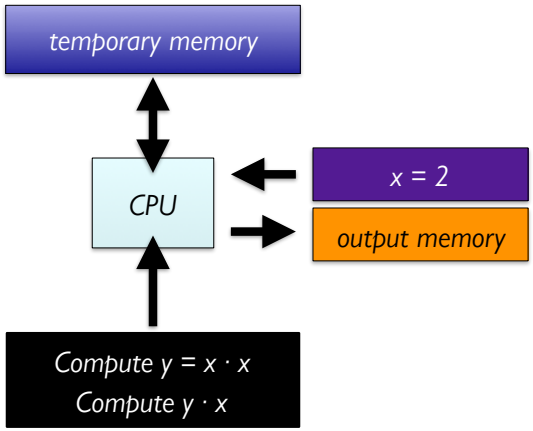
Computing machine



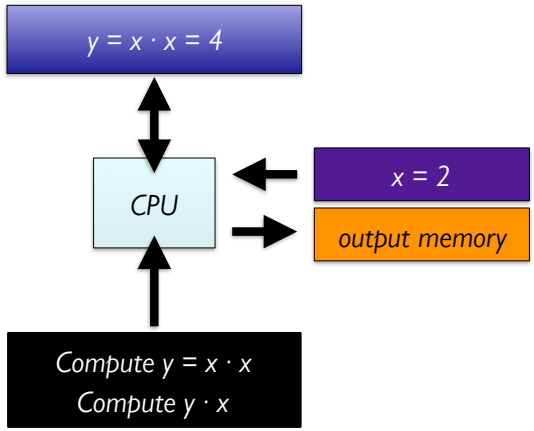
Example: $f(x) = x^3$



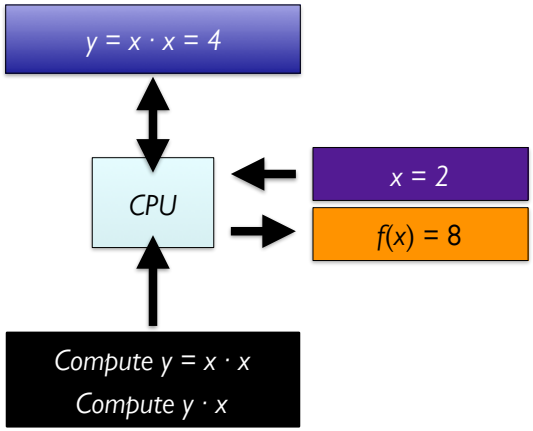
Example: $f(x) = x^3$



Example: $f(x) = x^3$



Example: $f(x) = x^3$

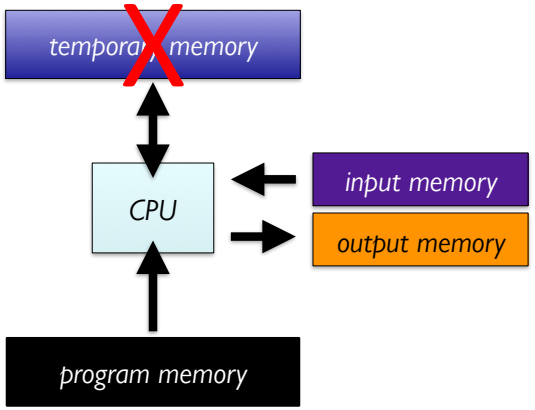


Different kinds of automata

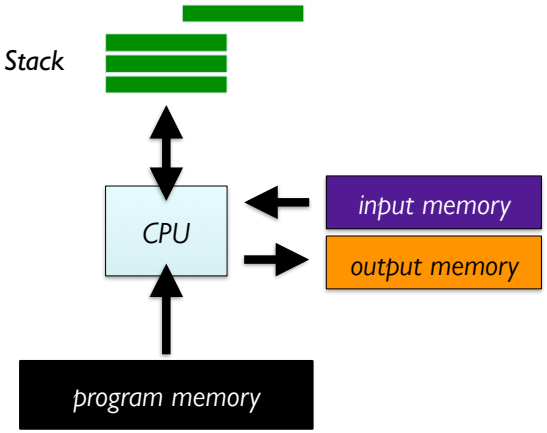
Automata are distinguished by their *temporary memory*:

- *Finite automata*: No temporary memory
- *Pushdown automata*: Stack
- *Turing machines*: Random access memory

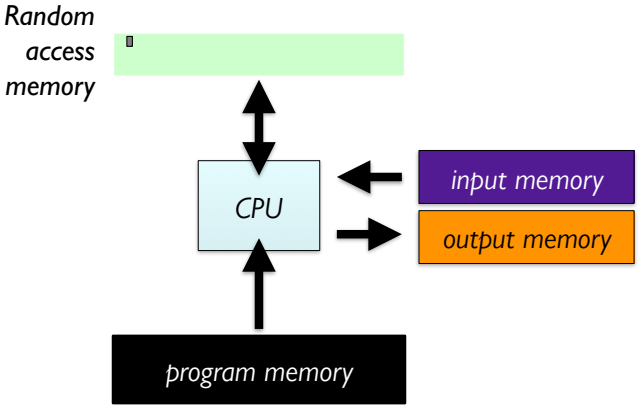
Finite automaton



Pushdown automaton



Turing machine



Power of automata

Finite automata < ***Pushdown automata*** < ***Turing machines***

Less power
Solve fewer computational problems

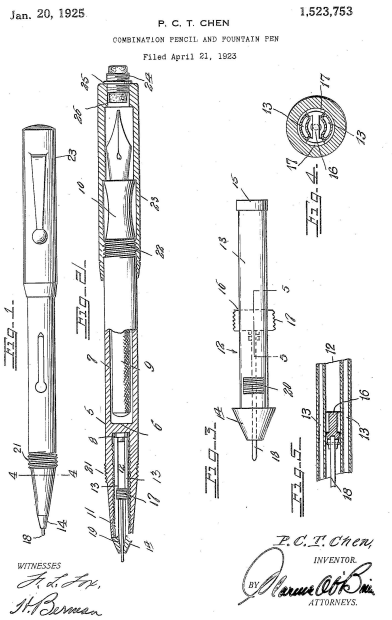
More power
Solve more computational problems

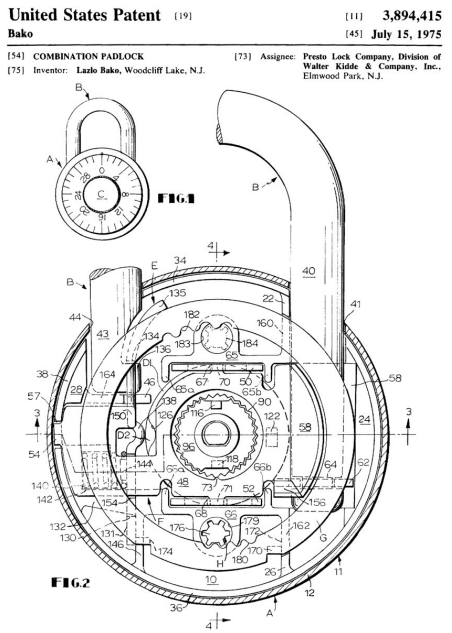
Finite automata

A *finite automaton* – also called a *finite state machine* – is the simplest model of a computer that's still interesting to study.

The essence of a finite state machine is a look-up table and a summary of the past, which is its state.

Once you learn to recognize finite automata, you'll notice them everywhere.





All finite-state machines have

- a fixed set of possible states
- a set of allowable inputs that change the state (e.g., clicking the pen or dialing a number into the lock)
- a set of possible outputs (retracting or extending the pen, opening the lock).
  - The outputs depend only on the state, which in turn depends only on the history of the sequence of inputs.
  - In our abstract machine model for language problems, the only outputs are accepting or rejecting the input.

Finite automaton

A useful practical abstraction:

- Retain sufficient flexibility to perform interesting tasks
- Minimal hardware requirements to building them
- A good model when memory is limited

Other real-world finite automata?

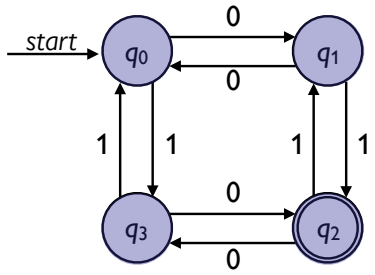
Finite automaton

Captures the basic elements of an abstract machine:

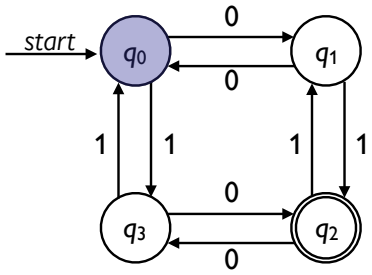
- Reads in a string
- Outputs **true** if the string is in the language it's programmed to recognize.
- Returns **false** if it's not.

Finite automata recognize a class of simple but highly useful languages called *regular languages*.

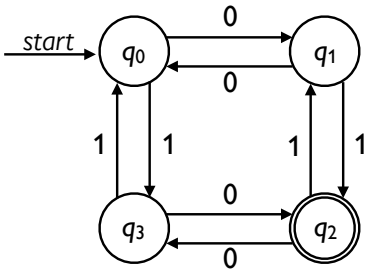
Each finite automaton consists of a set of nodes called *states* connected by *transitions*.



Each circle represents a state of the automaton

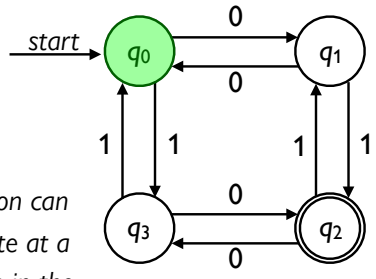


One state is designated as the start state, indicated by an arrow



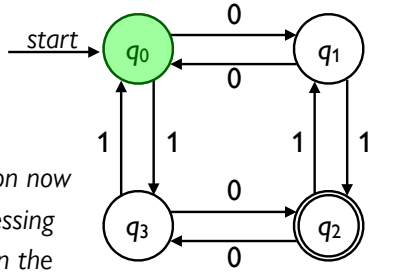
0 1 0 1 1 0

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



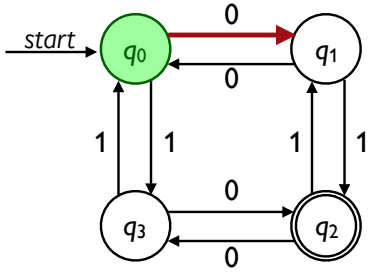
The automaton can be in one state at a time. It begins in the start state.

0 1 0 1 1 0



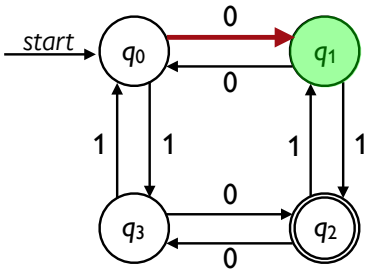
The automaton now begins processing characters in the order in which they appear.

0 1 0 1 1 0

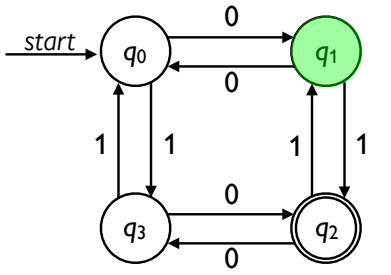


Each arrow in this diagram represents a transition. The automaton always follows the transition for the symbol being read.

0 1 0 1 1 0

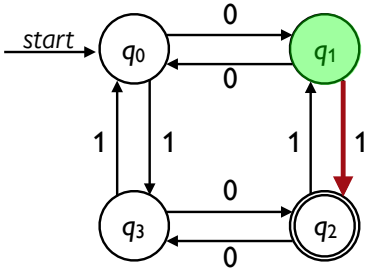


0 1 0 1 1 0

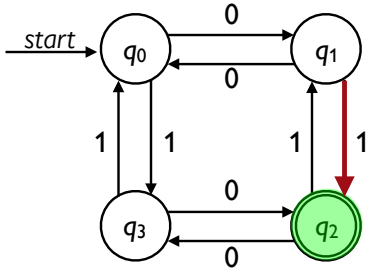


0 1 0 1 1 0

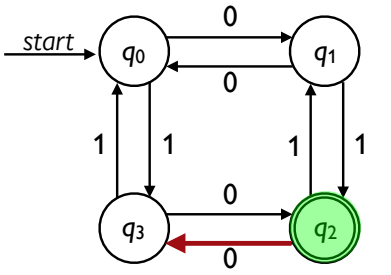
After transitioning, the automaton consider the next symbol in the input.



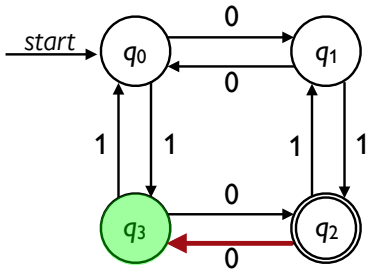
0 1 0 1 1 0



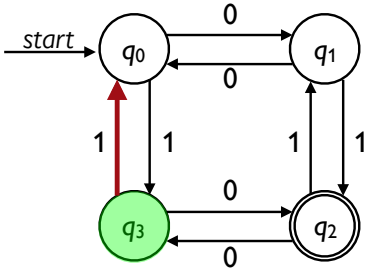
0 1 0 1 1 0



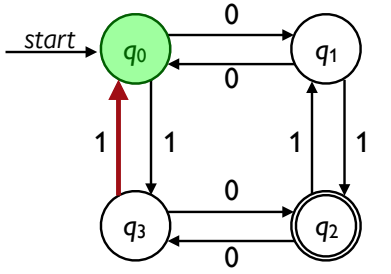
0 1 **0** 1 1 0



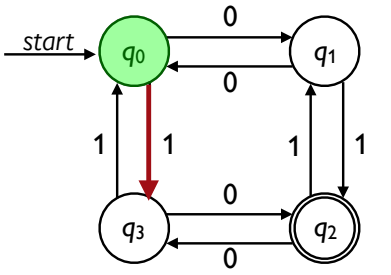
0 1 **0** 1 1 0



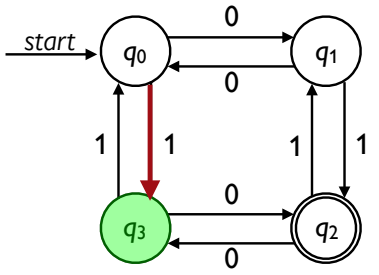
0 1 0 **1** 1 0



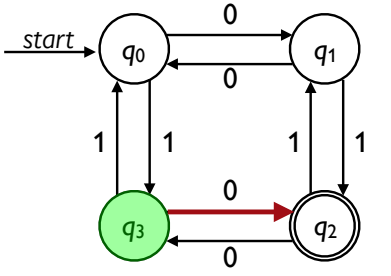
0 1 0 **1** 1 0



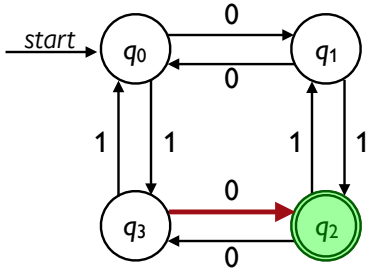
0 1 0 1 **1** 0



0 1 0 1 **1** 0

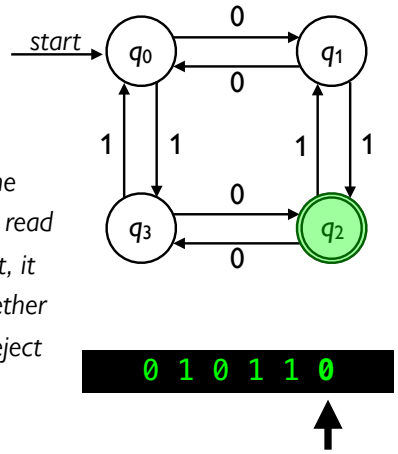


0 1 0 1 1 **0**



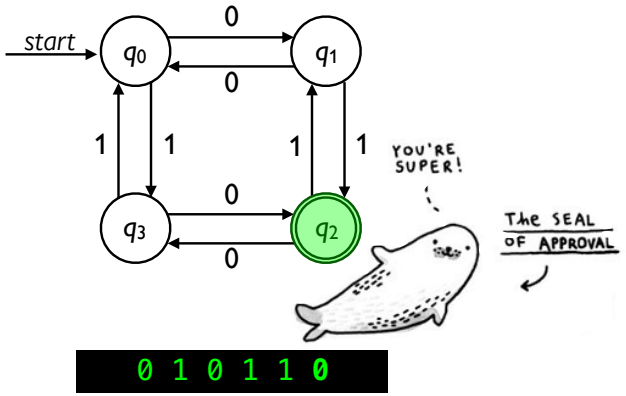
0 1 0 1 1 **0**

Now that the automaton has read all of the input, it can decide whether to accept or reject

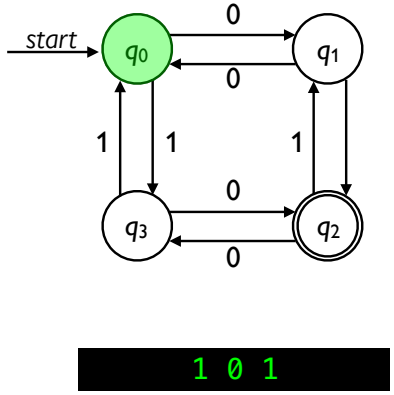


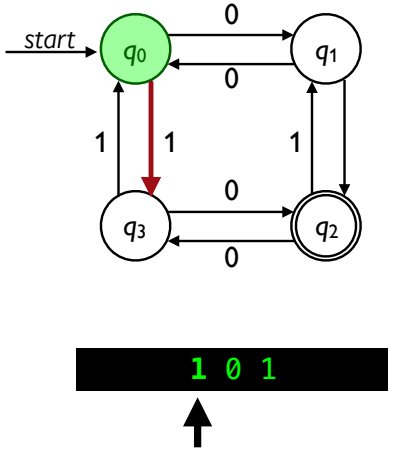
The double circle indicates this is an accepting state, so it accepts!

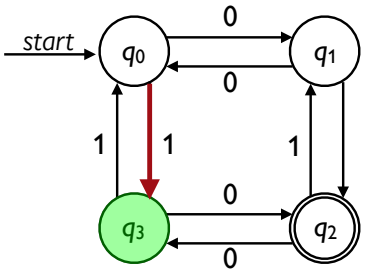
Illustration by Gemma Correll

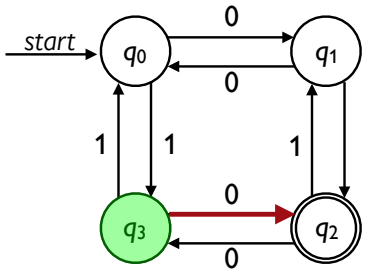


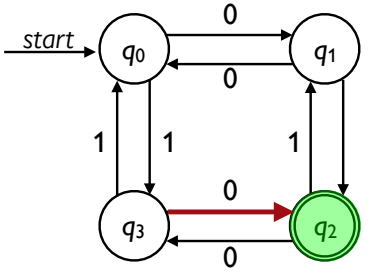
Let's try another input






start
q_0
q_1
q_2
q_3
0
0
1
1
1
0
0
1 0 1


start
q_0
q_1
q_2
q_3
0
0
1
1
1
0
0
1 0 1


start
q_0
q_1
q_2
q_3
0
0
1
1
1
0
0
1 0 1

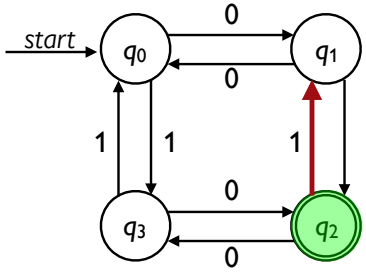
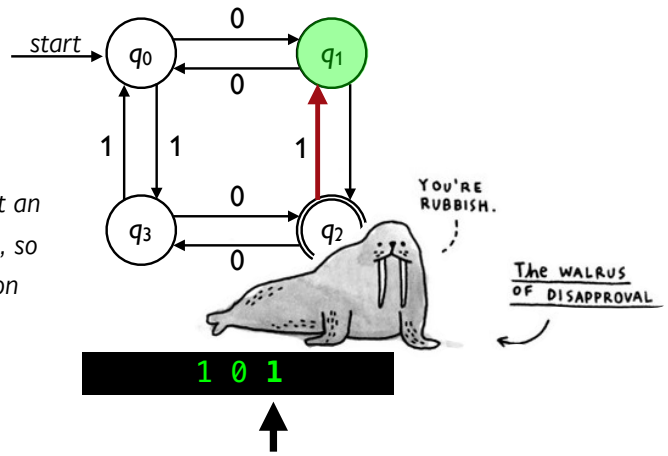

start
q_0
q_1
q_2
q_3
0
0
1
1
1
0
0
1 0 1

Illustration by Gemma Correll



This state is not an accepting state, so the automaton rejects.

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

- One state is designated the ***start state*** or ***initial state***.
- Some states are ***final states*** or ***accepting states***
- Transition arcs are labeled with one or more symbols from some alphabet.

An automaton processes a string by beginning in the start state and following the indicated transitions.

- The new state is completely determined by the current state and the symbol it just read.

When the input is exhausted,

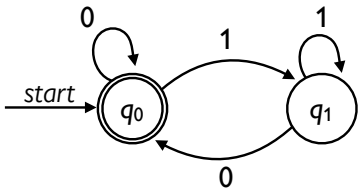
- If the automaton is in an accepting state, it *accepts* the input.
- Otherwise, it *rejects* the input.

Note: An automaton does ***not*** accept as soon as it enters an accepting state. It only accepts if it ***ends*** in an accepting state.

The *language of the FA* is the set of strings that it accepts, i.e., strings that label paths that go from the start state to some accepting state.

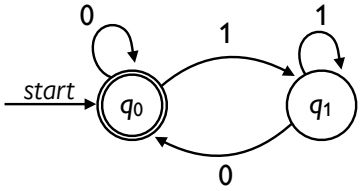
If M is an automaton, we denote its language as $L(M)$.

Example finite automaton



$L(M) = \{ ? \}$

Example 2



$L(M) = \{w \mid w \in \{0, 1\}^*$ and w does not have suffix $1\}$

Acknowledgments

This lecture incorporates material from:

W. Daniel Hillis

Nancy Ide

Keith Schwarz

Michael Sipser

Jeffrey Ullman