Finite Automata

23 February 2021
What problems can we solve with a computer?
What problems can we solve with a computer?

What kind of computer?
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.
Today, we’ll introduce our first computational model, called a \textit{finite automaton} or \textit{finite-state machine}.
What problems can we solve with a computer?
What problems can we solve with a computer?

What’s a “problem”?
Before we can talk about what problems we can solve, we need a formal definition of a “problem”.

We want a definition that

- corresponds to the problems we want to solve,
- captures a large class of problems, and
- is mathematically simple to reason about
Virtually all computational problems can be recast as *language recognition problems*.

E.g., the problem of determining whether an integer is prime:

*Problem*: Is 97 prime?

*Recast*: Is the string 97 in the language of all primes, \{2, 3, 5, 7, 13, \ldots\}?
Formal language theory
Strings
An *alphabet*, denoted $\Sigma$, is a finite, non-empty set of symbols called *characters*, e.g.,

**Binary:** $\Sigma = \{0, 1\}$

**ASCII:** $\Sigma = \{a, b, c, \ldots, 0, 1, \ldots, !, @, #, \ldots\}$
A *string* (or *word*) over an alphabet $\Sigma$ is a finite sequence of characters drawn from $\Sigma$.

For example, if $\Sigma = \{a, b\}$, then

- $a$
- $abbaba$

are strings over $\Sigma$.

The *empty string*, denoted $\varepsilon$, has no characters.
The *length* of a word $w$, written as $|w|$, is the number of symbols in it, e.g.,

$$|\varepsilon| = 0$$

$$|\text{abcde}| = 5$$
String operations

**Reverse:** $w^R$

E.g., if $w = abc$, then $w^R = cba$

**Concatenation:** Given strings $x$ and $y$, their concatenation is $xy$.

E.g., if $x = abc$, and $y = def$, then $xy = abcdef$
String relations

A string $z$ is a **substring** of $w$ if $z$ appears consecutively in $w$, e.g.,

- atm is a substring of batman

**Prefix** and **suffix**: If $w = vu$, $v$ and $u$ are a prefix and a suffix of $w$, respectively, e.g.,

- bat is a prefix of batman
- man is a suffix of batman
Languages
A formal \textit{language} is a set of strings, e.g.,

\[ \emptyset, \text{ the } \textit{empty set}, \text{ is a language of zero strings} \]

\[ \{\text{kitty, cat}\} \text{ is a language of two strings} \]

We say that \( L \) is a \textit{language over} \( \Sigma \) if it is a set of strings over \( \Sigma \), e.g.,

The language of palindromes over 
\( \Sigma = \{a, b, c\} \) is the infinite set

\[ \{\varepsilon, a, b, c, aa, bb, cc, aaa, aba, aca, bab, \ldots\} \]
If a language contains a very large number of strings – whether it is finite or infinite – we don’t write them all out.

We can use an ellipsis, e.g.,

\[
\{a, aa, aaa, \ldots \}
\]

but this is not very precise. E.g., we might mean the language consisting of strings composed of

one or more \textit{as}, or

either one \textit{a} or a prime number of \textit{as}. 

Instead, we use set-builder notation to describe a property of the set, e.g.,

\[ \{w \mid w \text{ consists of 0 or more } a \text{s or } b \text{s in any order} \} \]

“The set of all strings \( w \) such that \( w \)”
Set notation

**Set membership:** \( \in \)

Newt \( \in \) \{Helga, Tonks, Newt\}

**Non-membership:** \( \notin \)

Harry \( \notin \) \{Helga, Tonks, Newt\}

**Proper subset:** \( \subset \)

\{Newt\} \( \subset \) \{Helga, Tonks, Newt\}

**Subset** (or equal to): \( \subseteq \)

\{Helga, Tonks, Newt\} \( \subseteq \) \{Helga, Tonks, Newt\}

\( |X| \) is the number of elements in set \( X \) (its *cardinality*)
Powers

Let $\Sigma = \{a, b\}$.

$\Sigma^k$ is the set of all strings from alphabet $\Sigma$ with length $k$.

$\Sigma^0 = ?$
$\Sigma^1 = ?$
$\Sigma^2 = ?$
Powers

Let $\Sigma = \{a, b\}$.

$\Sigma^k$ is the set of all strings over alphabet $\Sigma$ with length $k$.

- $\Sigma^0 = \{\varepsilon\}$
- $\Sigma^1 = \{a, b\}$
- $\Sigma^2 = \{aa, bb, ab, ba\}$

$\Sigma^* = \text{set of all strings over alphabet } \Sigma$ (*Kleene star*)

$\Sigma^* = ?$
Powers

Let $\Sigma = \{a, b\}$.

$\Sigma^k$ is the set of all strings over alphabet $\Sigma$ with length $k$.

$\Sigma^0 = \{\varepsilon\}$
$\Sigma^1 = \{a, b\}$
$\Sigma^2 = \{aa, bb, ab, ba\}$

$\Sigma^* =$ set of all strings over alphabet $\Sigma$ (Kleene star)

$\Sigma^* = \{\varepsilon, a, b, aa, bb, ab, ba, \ldots\}$

$\Sigma^+ = \Sigma^* - \{\varepsilon\}$
Since languages are sets, we can also use standard set operations, including

union \( (\cup) \),

intersection \( (\cap) \),

difference \( (-) \), and

complement.

More on this later!
Languages are sets of Strings, which are finite sequences of Characters. Alphabets are finite, nonempty sets of Characters.
What problems can we solve with a computer?
What problems can we solve with a computer?
Modeling computers
To compare different classes of abstract machines, we define a notion of \textit{power}.

Machine A is \textit{at least as powerful} as machine B if A can be programmed to recognize all of the languages B can.

Machine A is \textit{more powerful} than B if it can be programmed to recognize all the languages B can and at least one more.

Two machines are \textit{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

http://www.intel.com/design/intarch/prodbref/272713.htm
Computing machine

temporary memory

CPU

input memory

output memory

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

- **CPU**
  - Compute $y = x \cdot x$
  - Compute $y \cdot x$

- **Input Memory**
- **Output Memory**
- **Temporary Memory**
Example: \( f(x) = x^3 \)
Example: $f(x) = x^3$

$y = x \cdot x = 4$

$x = 2$

output memory

Compute $y = x \cdot x$
Compute $y \cdot x$
Example: $f(x) = x^3$

$y = x \cdot x = 4$

$x = 2$

$f(x) = 8$

Compute $y = x \cdot x$

Compute $y \cdot x$
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

- Temporary memory
- Input memory
- Output memory
- Program memory
- CPU
Pushdown automaton

- Stack
- Program memory
- Input memory
- Output memory
- CPU
Turing machine

Random access memory

CPU

program memory

input memory

output memory
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.
Finite-state machines are all around us.
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
- Accepts if the string is in the language it’s programmed to recognize.
- Rejects 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 *states* connected by *transitions*. 
Each circle represents a state of the automaton
One state is designated as the start state, indicated by an arrow.
The automaton is run on an input string and answers “yes” or “no”.

0 1 0 1 1 0
The automaton can be in one state at a time. It begins in the start state.
The automaton now begins processing characters in the order in which they appear.
The image shows a finite automaton with four states: $q_0$, $q_1$, $q_2$, and $q_3$. The transitions are labeled with symbols: 0 and 1. The start state is labeled as $q_0$. The transitions are as follows:

- From $q_0$ on input 0 to $q_1$.
- From $q_0$ on input 1 to $q_3$.
- From $q_2$ on input 1 to $q_2$.
- From $q_2$ on input 0 to $q_1$.
- From $q_1$ on input 0 to $q_0$.
- From $q_1$ on input 1 to $q_2$.
- From $q_3$ on input 1 to $q_2$.

The input sequence at the bottom is 010110.
Each arrow in this diagram represents a transition. The automaton always follows the transition for the symbol being read.
After transitioning, the automaton considers the next symbol in the input.
A finite automaton with states $q_0$, $q_1$, $q_2$, and $q_3$. The transitions are labeled with input symbols:

- From $q_0$: 0 to $q_0$, 1 to $q_3$, 1 to $q_2$.
- From $q_1$: 0 to $q_1$, 1 to $q_2$.
- From $q_2$: 0 to $q_2$, 1 to $q_1$.
- From $q_3$: 0 to $q_3$.

The start state is $q_0$.
Now that the automaton has read all of the input, it can decide whether to accept or reject.
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!
A DFA with states: $q_0$, $q_1$, $q_2$, and $q_3$. The transitions are:

- From $q_0$ to $q_0$ on input 0.
- From $q_0$ to $q_1$ on input 1.
- From $q_1$ to $q_0$ on input 0.
- From $q_1$ to $q_2$ on input 1.
- From $q_2$ to $q_3$ on input 0.
- From $q_3$ to $q_2$ on input 0.

The start state is $q_0$. The accepting state is $q_2$. The input sequence is: 0101110.
Let’s try another input
Let’s try another input
The diagram represents a finite state machine (FSM) with states $q_0$, $q_1$, $q_2$, and $q_3$. The states are connected by transitions labeled with inputs 0 and 1.

- The start state is $q_0$.
- From $q_0$, there is a transition labeled 0 to $q_1$.
- From $q_1$, there is a transition labeled 1 to $q_3$.
- From $q_3$, there is a transition labeled 1 to $q_0$.
- From $q_0$, there is a transition labeled 1 to $q_3$.
- From $q_3$, there is a transition labeled 0 to $q_2$.
- From $q_2$, there is a transition labeled 1 to $q_3$.
- From $q_3$, there is a transition labeled 1 to $q_2$.

The diagram also includes a sequence of 1, 0, 1 at the bottom, indicating a specific input sequence for the FSM.
This state is not an accepting state, so the automaton rejects.
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.
A finite 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 a finite automaton 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 that processes characters from the alphabet $\Sigma$, then its language is defined as

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$$
Example finite automaton

$L(M) = \{ \ ? \ \}$
Example finite automaton

\[ L(M) = \{ w \mid w \in \{0, 1\}^* \text{ and } w \text{ does not have suffix } 1 \} \]
Finite automata in action

Used in

- Text editors and search engines for pattern matching
- Compilers for lexical analysis
- Web browsers for HTML parsing

Serve as the control unit in many physical systems, including

- Elevators, traffic signals, vending machines
- Computer microprocessors
- Network protocol stacks and old VCR clocks

Play a key role in natural language processing and machine learning

*Markov chains* are probabilistic FAs used in part of speech tagging, speech processing, and optical character recognition
Finite automaton for a newspaper vending machine
Acknowledgments

This lecture incorporates material from:

W. Daniel Hillis, *The Pattern on the Stone*
Nancy Ide, Vassar College
Keith Schwarz, Stanford University
Michael Sipser, *Introduction to the Theory of Computation*
Jeffrey Ullman, Stanford University