One Minute To Learn Programming:
Finite Automata
Let me teach you a programming language so simple that you can learn it in less than a minute.
Meet “ABA” The Automaton!

<table>
<thead>
<tr>
<th>Input String</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba</td>
<td>Accept</td>
</tr>
<tr>
<td>aabb</td>
<td>Reject</td>
</tr>
<tr>
<td>aabba</td>
<td>Accept</td>
</tr>
<tr>
<td>ε</td>
<td>Accept</td>
</tr>
</tbody>
</table>
The Simplest Interesting Machine:

Finite State Machine

OR

Finite Automaton
**Finite Automaton**

<table>
<thead>
<tr>
<th>Finite set of states</th>
<th>![States]</th>
<th>$Q = {q_o, q_1, q_2, \ldots, q_k}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A start state</td>
<td>![Start State]</td>
<td>$q_o$</td>
</tr>
<tr>
<td>A set of accepting states</td>
<td>![Accepting States]</td>
<td>$F = {q_{i_1}, q_{i_2}, \ldots, q_{i_r}}$</td>
</tr>
<tr>
<td>A finite alphabet</td>
<td>$a \ b \ # \ x \ 1$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>State transition instructions</td>
<td>![Transition]</td>
<td>$\delta : Q \times \Sigma \rightarrow Q$</td>
</tr>
<tr>
<td></td>
<td>$q_i \ a \ q_j$</td>
<td>$\delta(q_i, a) = q_j$</td>
</tr>
</tbody>
</table>
How Machine $M$ operates.

$M$ “reads” one letter at a time from the input string (going from left to right)

$M$ starts in state $q_0$.
If $M$ is in state $q_i$ reads the letter $a$ then

If $\delta(q_i, a)$ is undefined then CRASH.

Otherwise $M$ moves to state $\delta(q_i, a)$
Let $M=(Q, \Sigma, F, \delta)$ be a finite automaton.

$M$ accepts the string $x$ if when $M$ reads $x$ it ends in an accepting state.

$M$ rejects the string $x$ if when $M$ reads $x$ it ends in a non-accepting state.

$M$ crashes on $x$ if $M$ crashes while reading $x$. 
The set (or language) accepted by $M$ is:

$$L_M = \{ x \in \Sigma^* \mid M \text{ accepts } x \}$$

$$\Sigma^k \equiv \text{All length } k \text{ strings over the alphabet } \Sigma$$

$$\Sigma^* \equiv \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$$

Notice that this is $\{ \varepsilon \}$
What is the language accepted by this machine?

$L = \{a,b\}^* = \text{all finite strings of } a\text{'s and } b\text{'s}$
What is the language accepted by this machine?

$L = \text{all even length strings of } a \text{'s and } b \text{'s}$
What machine accepts this language?

$L = \text{all strings in } \{a, b\}^* \text{ that contain at least one } a$
What machine accepts this language?

$$L = \text{strings with an odd number of } b\text{'s and any number of } a\text{'s}$$
What is the language accepted by this machine?

$L = \text{any string ending with } a \ b$
What is the language accepted by this machine?

$L = \text{any string with at least two } a\text{'s}$
What machine accepts this language?

$L =$ any string with an $a$ and a $b$
What machine accepts this language?

$L = \text{strings with an even number of } ab \text{ pairs}$
\( L = \) all strings containing \textit{ababb} as a consecutive substring

Invariant: I am state \textit{s} exactly when \textit{s} is the longest suffix of the input (so far) that forms a prefix of \textit{ababb}.
The “grep” Problem

Input:
  • text $T$ of length $t$
  • string $S$ of length $n$

Problem:
  Does the string $S$ appear inside the text $T$?

Naïve method:

$\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_t$

Cost: $O(nt)$ comparisons
Automata Solution

• Build a machine $M$ that accepts any string with $S$ as a consecutive substring.

• Feed the text to $M$.

• **Cost:** $t$ comparisons + time to build $M$.

• As luck would have it, the Knuth, *Morris*, Pratt algorithm builds $M$ quickly.

• By the way, it can be done with fewer than $t$ comparisons in the worst case!
Real-life uses of finite state machines

- grep
- coke machines
- thermostats (fridge)
- elevators
- train track switches
- lexical analyzers for parsers
Any $L \subseteq \Sigma^*$ is defined to be a language.

$L$ is just a set of strings. It is called a language for historical reasons.
Let \( L \subseteq \Sigma^* \) be a language.

\( L \) is called a regular language if there is some finite automaton that accepts \( L \).

In this lecture we have seen many regular languages.

- \( \Sigma^* \)
- even length strings
- strings containing \( ababb \)
**Theorem:** Any finite language is regular.

**Proof:** Make a machine with a “path” for each string in the language.

**Example:** $L = \{a, bcd, ac, bb\}$
Are all languages regular?
Consider the language

\[ a^n b^n = \{ \varepsilon, ab, aabb, aaabbb, \ldots \} \]

i.e., a bunch of \( a \)'s followed by an equal number of \( b \)'s.

No finite automaton accepts this language.

Can you prove this?
$a^n b^n$ is not regular. No machine has enough states to keep track of the number of $a$’s it might encounter.
That is a fairly weak argument. Consider the following example...
$L =$ strings where the # of occurrences of the pattern $ab$ is equal to the number of occurrences of the pattern $ba$

Can't be regular. No machine has enough states to keep track of the number of occurrences of $ab$. 
Remember “ABA”?

ABA accepts only the strings with an equal number of ab’s and ba’s!
Let me show you a professional strength proof that $a^n b^n$ is not regular....
Theorem: $a^n b^n$ is not regular.

Proof: Assume that it is. Then $\exists M$ with $k$ states that accepts it.

For each $0 \leq i \leq k$, let $S_i$ be the state $M$ is in after reading $a^i$.

$\exists i, j \leq k$ s.t. $S_i = S_j$, but $i \neq j$

$M$ will do the same thing on $a^i b^i$ and $a^j b^i$.

But a valid $M$ must reject $a^j b^i$ and accept $a^i b^i$. $\implies \iff$
MORAL:

Finite automata can't count.
You can learn much more about these creatures in the FLAC course.

Formal Languages, Automata, and Computation

- There is a unique smallest automaton for any regular language
- It can be found by a fast algorithm.
Cellular Automata

• Line up a bunch of identical finite automata in a straight line.

• Transitions are based on the states of the machine’s two neighbors or an indicator that a neighbor is missing. (There is no other input.)

\[
Q \times Q \times Q \rightarrow Q
\]

\[
(q_k, q_i, q_j) = q_l
\]

• All cells move to their next states at the same time: synchronous transition
The Firing Squad Problem

- Five “soldiers” all start in the sleep state. You change the one on the left to the wake state.
- All five must get to the fire state at the same time (for the first time).
Shorthand

\[ Q,\{a,b,d\} \]

Means use this transition when your left neighbor is in any state at all and your right neighbor is in state \( a,b, \) or \( d. \)
The diagram shows a state transition diagram with states labeled as `sleep`, `wake`, and `fire`. The states are connected by edges labeled with sets `{sleep, fire, end}`, `{w}`, `{w2}`, `{w3}`, `{w4}`, and `{sleep, fire, end}`. The table below lists the transitions for each state:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>2</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>3</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>4</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>5</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
</tbody>
</table>
{sleep, fire, end}, Q

{w}, Q
{w2}, Q
{w3}, Q
{w4}, Q

wake
wake2
wake3
wake4

fire!

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>wake</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
</tbody>
</table>
{sleep, fire, end}, Q

sleep

{w}, Q

{w2}, Q

{w3}, Q

{w4}, Q

wake

Q, Q

wake2

Q, Q

wake3

Q, Q

wake4

Q, Q

fire!

Q, Q

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>wake</td>
<td>wake</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>wake2</td>
<td>wake2</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
</tbody>
</table>
The figure shows a state diagram with states labeled as 'sleep', 'wake', 'fire!', and a transition diagram between them.

States:
- Sleep
- Wake
- Wake2
- Wake3
- Wake4

Transitions:
- From sleep to wake
- From wake to wake2
- From wake2 to wake3
- From wake3 to wake4
- From wake3 to fire

The table below represents the transitions:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>wake</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>wake2</td>
<td>wake2</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>wake3</td>
<td>wake3</td>
<td>wake3</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
</tbody>
</table>
A diagram showing states and transitions labeled with different events:

- **States:**
  - Sleep
  - Wake
  - Wake2
  - Wake3
  - Wake4

- **Events:**
  - Sleep
  - Fire

- **Transitions:***
  - From Sleep to Sleep
  - From Sleep to Wake
  - From Sleep to Wake2 (with event {w}, Q)
  - From Sleep to Wake3 (with event {w2}, Q)
  - From Sleep to Wake4 (with event {w3}, Q)
  - From Wake to Wake2 (with event {w}, Q)
  - From Wake2 to Wake3 (with event {w2}, Q)
  - From Wake3 to Wake4 (with event {w3}, Q)
  - From Wake4 to Wake (with event {w4}, Q)
  - From Fire to Sleep

A table showing the sequence of events:

<table>
<thead>
<tr>
<th>Event</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wake</td>
<td>Sleep</td>
<td>Sleep</td>
<td>Sleep</td>
<td>Sleep</td>
<td>Sleep</td>
</tr>
<tr>
<td>Wake2</td>
<td>Wake2</td>
<td>Sleep</td>
<td>Sleep</td>
<td>Sleep</td>
<td>Sleep</td>
</tr>
<tr>
<td>Wake3</td>
<td>Wake3</td>
<td>Wake3</td>
<td>Sleep</td>
<td>Sleep</td>
<td>Sleep</td>
</tr>
<tr>
<td>Wake4</td>
<td>Wake4</td>
<td>Wake4</td>
<td>Wake4</td>
<td>Wake4</td>
<td>Sleep</td>
</tr>
</tbody>
</table>
sleep
{sleep,fire,end},Q

wake
{w},Q

wake2
{w2},Q

wake3
{w3},Q

wake4
{w4},Q

fire!
{w2},Q
{w3},Q
{w4},Q

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>wake</td>
<td>wake</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>wake2</td>
<td>wake2</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>wake3</td>
<td>wake3</td>
<td>wake3</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>wake4</td>
<td>wake4</td>
<td>wake4</td>
<td>wake4</td>
<td>wake4</td>
<td>sleep</td>
</tr>
<tr>
<td>fire</td>
<td>fire</td>
<td>fire</td>
<td>fire</td>
<td>fire</td>
<td>fire</td>
</tr>
</tbody>
</table>
Question

Can you build the soldier’s finite automaton brain before you know how many soldiers will be in the line?

No. Finite automata can’t count!
Don’t jump to conclusions! It is possible to design a single cellular automaton that works for any number of soldiers!