Patterns and Finite Automata

A *pattern* is a set of objects with a recognizable property.

- In computer science, we’re typically interested in patterns that are sequences of character strings
  - I think “Halpern” a very interesting pattern
  - I may want to find all occurrences of that pattern in a paper
- Other patterns:
  - **if** followed by any string of characters followed by **then**
  - all filenames ending with “.doc”

Pattern matching comes up all the time in text search.

A *finite automaton* is a particularly simple computing device that can recognize certain types of patterns, called *regular languages*

- The text does not cover finite automata; there is a separate handout on CMS.
Finite Automata

A *finite automaton* is a machine that is always in one of a finite number of states.

- When it gets some input, it moves from one state to another
  - If I’m in a “sad” state and someone hugs me, I move to a “happy” state
  - If I’m in a “happy” state and someone yells at me, I move to a “sad” state

**Example:** A digital watch with “buttons” on the side for changing the time and date, or switching it to “stopwatch” mode, is an automaton

- What are the states and inputs of this automaton?

- A certain state is denoted the *start* state
  - That’s how the automaton starts life

- Other states are denoted *final* state
  - The automaton stops when it reaches a final state
  - (A digital watch has no final state, unless we count running out of battery power.)
A finite automaton can be represented by a labeled directed graph.

- The nodes represent the states of the machine
- The edges are labeled by inputs, and describe how the machine transitions from one state to another
Example:

There are four states: $s_0, s_1, s_2, s_3$
- $s_0$ is the start state (denoted by “start →”, by convention)
- $s_0$ and $s_3$ are the final states (denoted by double circles, by convention)

The labeled edges describe the transitions for each input
- The inputs are either 0 or 1
  - in state $s_0$ and reads 0, it stays in $s_0$
  - If the machine is in state $s_0$ and reads 1, it moves to $s_1$
  - If the machine is in state $s_1$ and reads 0, it moves to $s_1$
  - If the machine is in state $s_1$ and reads 1, it moves to $s_2$
What happens on input 00000? 0101010? 010101? 11?

- Some strings move the automaton to a final state; some don’t.
- The strings that take it to a final state are accepted.
A Parity-Checking Automaton

Here’s an automaton that accepts strings of 0s and 1s that have even parity (an even number of 1s).

We need two states:

- $s_0$: we’ve seen an even number of 1s so far
- $s_1$: we’ve seen an odd number of 1s so far

The transition function is easy:

- If you see a 0, stay where you are; the number of 1s hasn’t changed
- If you see a 1, move from $s_0$ to $s_1$, and from $s_1$ to $s_0$
Finite Automata: Formal Definition

A *(deterministic)* finite automaton is a tuple $M = (S, I, f, s_0, F)$:

- $S$ is a finite set of states;
- $I$ is a finite input alphabet (e.g. $\{0, 1\}$, $\{a, \ldots, z\}$)
- $f$ is a transition function; $f : S \times I \rightarrow S$
  - $f$ describes what the next state is if the machine is in state $s$ and sees input $i \in I$.
- $s_0 \in S$ is the initial state;
- $F \subseteq S$ is the set of final states.
Example:

- $S = \{s_0, s_1, s_2, s_3\}$
- $I = \{0, 1\}$
- $F = \{s_0, s_3\}$
- The transition function $f$ is described by the graph:
  - $f(s_0, 0) = s_0; f(s_0, 1) = s_1; f(s_1, 0) = s_0; \ldots$

You should be able to translate back and forth between finite automata and the graphs that describe them.
Describing Languages

The language accepted (or recognized) by an automaton is the set of strings that it accepts.

- A language is a set of strings

We need tools for describing languages.

- If $A$ and $B$ are sets of strings, then $AB$, the concatenation of $A$ and $B$, is the set of all strings $ab$ such that $a \in A$ and $b \in B$.
  - **Example:** If $A = \{0, 11\}$, $B = \{111, 00\}$, then
    - $AB = \{0111, 000, 1111, 1100\}$
    - $BA = \{1110, 11111, 000, 0011\}$

- Define $A^{n+1}$ inductively:
  - $A^0 = \{\lambda\}$: $\lambda$ is the empty string
  - $A^1 = A$
  - $A^{n+1} = AA^n$
  - $A^* = \bigcup_{n=0}^{\infty} A^n$. 

What's $\{0, 1\}^n$? $\{0, 1\}^*$?

$\{11\}^*$?
Describing Languages

The *language* accepted (or *recognized*) by an automaton is the set of strings that it accepts.

- A *language* is a set of strings

We need tools for describing languages.

- If $A$ and $B$ are sets of strings, then $AB$, the *concatenation* of $A$ and $B$, is the set of all strings $ab$ such that $a \in A$ and $b \in B$.

  - **Example:** If $A = \{0, 11\}$, $B = \{111, 00\}$, then
    - $AB = \{0111, 000, 1111, 1100\}$
    - $BA = \{1110, 11111, 000, 0011\}$

- Define $A^{n+1}$ inductively:
  - $A^0 = \{\lambda\}$: $\lambda$ is the empty string
  - $A^1 = A$
  - $A^{n+1} = AA^n$

- $A^* = \bigcup_{n=0}^{\infty} A^n$.

- What’s $\{0, 1\}^n$? $\{0, 1\}^*$? $\{11\}^*$?
Regular Expressions

A regular expression is an algebraic way of defining a pattern

**Definition:** The set of regular expressions over \( I \) (where \( I \) is an input set) is the smallest set \( S \) of expressions such that:

- the symbol \( \emptyset \in S \) (that should be a boldface \( \emptyset \))
- the symbol \( \lambda \in S \) (that should be a boldface \( \lambda \))
- the symbol \( x \in S \) is a regular expression if \( x \in I \);
- if \( E_1 \) and \( E_2 \) are in \( S \), then so are \((E_1E_2)\), \((E_1 \cup E_2)\) and \( E_1^* \).

That is, we start with the empty set, \( \lambda \), and elements of \( I \), then close off under union, concatenation, and \( * \).

- A regular set is a syntactic object: a sequence of symbols.
- Concatenation, union, and \( * \) are overloaded; they’re used for both languages (sets of strings) and regular expressions (sequences of symbols)
- The parens are used for disambiguation
  \[ (((ab) \cup c) \neq (a(b \cup c)) \]
- There is an equivalent inductive definition (see homework).

Those of you familiar with the programming language Perl or Unix searches should recognize the syntax ...
Each regular expression $E$ over $I$ defines a subset of $I^*$, denoted $L(E)$ (the *language* of $E$) in the obvious way:

- $L(\emptyset) = \emptyset$;
- $L(\lambda) = \{\lambda\}$;
- $L(x) = \{x\}$;
- $L(E_1E_2) = L(E_1)L(E_2)$;
- $L(E_1 \cup E_2) = L(E_1) \cup L(E_2)$;
- $L(E^*) = L(E)^*$.

**Examples:**

- What’s $L(0^*10^*10^*)$?
- What’s $L((0^*10^*10^*)^n)$? $L(0^*(0^*10^*10^*)^*)$?
- $L(0^*(0^*10^*10^*)^*)$ is the language accepted by the parity automaton!
- If $\Sigma = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9\} \cup Punctuation$, what is $\Sigma^*\{H\}{a}\{l\}\{p\}\{e\}\{r\}\{n\}\Sigma^*$?
  - *Punctuation* consists of the punctuation symbols (comma, period, space, etc.)
  - $\Sigma$ is the input alphabet
  - Note that $L(\Sigma^*\text{Halpern}\Sigma^*) = \Sigma^*\{H\}{a}\{l\}\{p\}\{e\}\{r\}\{n\}\Sigma^*$. 

\[11 / 37\]
Can you define an automaton that accepts exactly the strings in $\Sigma^* \text{Halpern}\Sigma^*$?

- How many states would you need?
Can you define an automaton that accepts exactly the strings in $\Sigma^* Halpern\Sigma^*$?

- How many states would you need?

What language is represented by the automaton in the original example:

![Automaton Diagram]

1. $s_1$
2. $s_2$
3. $s_3$

States:
- $s_0$: start
- $s_1$
- $s_2$
- $s_3$

Transitions:
- $s_0 \xrightarrow{0} s_0$
- $s_0 \xrightarrow{0,1} s_2$
- $s_0 \xrightarrow{1} s_1$
- $s_1 \xrightarrow{1} s_3$
- $s_2 \xrightarrow{1,0} s_3$

Language represented: $((0 \cup 1)^* 0 \cup 111)^*$

It's not easy to prove this formally!
Can you define an automaton that accepts exactly the strings in $\Sigma^* Halpern\Sigma^*$?

- How many states would you need?

What language is represented by the automaton in the original example:

- $((0^* 0^*((110) \cup (111))^* ))^*$
- Perhaps clearer: $((0 \cup 1)^* 0 \cup 111)^*$
- It’s not easy to prove this formally!
What language is accepted by the following automata:
What language is accepted by the following automata:

$L(1^*) = \{1\}^*$
What language is accepted by the following automata:

$L(1^*) = \{1\}^*$

$L(1 \cup 01) = \{1, 01\}$
\[ L(0 \ast \{0\} \ast \{10\} \ast \{0,1\}) = \{0\} \ast \{10\} \ast \{0,1\} \]
\( L(0^*10(0 \cup 1)^*) = \{0\}^*\{10\}\{0, 1\}^* \)
So far we’ve considered *deterministic* finite automata (DFA)

- what happens in a state is completely determined by the input symbol read

*Nondeterministic* finite automata allow several possible next states when an input is read.

Formally, a nondeterministic finite automaton is a tuple $M = (S, I, f, s_0, F)$. All the components are just like a DFA, except now $f : S \times I \rightarrow 2^S$ (before, $f : S \times I \rightarrow S$).

- if $s' \in f(s, i)$, then $s'$ is a possible next state if the machines is in state $s$ and sees input $i$. 

We can still use a graph to represent an NFA. There might be several edges coming out of a state labeled by $i \in I$, or none. In the example below, there are two edges coming out of $s_0$ labeled 0, and none coming out of $s_4$ labeled 1.

- Can either stay in $s_0$ or move to $s_2$
- On input 111, get stuck in $s_4$ after 11, so 111 not accepted.
An NFA $M$ accepts (or recognizes) a string $x$ if it is possible to get to a final state from the start state with input $x$.

The language $L$ is accepted by an NFA $M$ consists of all strings accepted by $M$.

What language is accepted by this NFA:
An NFA $M$ accepts (or recognizes) a string $x$ if it is possible to get to a final state from the start state with input $x$.

The language $L$ is accepted by an NFA $M$ consists of all strings accepted by $M$.

What language is accepted by this NFA:

$L(0^*01 \cup 0^*11)$
Equivalence of Automata

Every DFA is an NFA, but not every NFA is a DFA.

- Do we gain extra power from nondeterminism?
  - Are there languages that are accepted by an NFA that can’t be accepted by a DFA?
  - Somewhat surprising answer: NO!

Define two automata to be *equivalent* if they accept the same language.

Example:

![Diagram](image-url)
**Theorem:** Every nondeterministic finite automaton is equivalent to some deterministic finite automaton.

**Proof:** Given an NFA $M = (S, I, f, s_0, F)$, let $M' = (S', I, f', \{s_0\}, F')$, where

- $S' = 2^S$
- $f'(A, i) = \{ t : t \in f(s, i) \text{ for some } s \in A \} \in 2^S$
- $f' : 2^S \times I \rightarrow 2^S$ (i.e., $f' : S' \times I \rightarrow S'$)
- $F' = \{ A : A \cap F \neq \emptyset \}$

Thus,

- the states in $M'$ are subsets of states in $M$;
- the final states in $M'$ are the sets which contain a final state in $M$;
- in state $A$, given input $i$, the next state consists of all possible next states from an element in $A$.

$M'$ is **deterministic**.

- This is called the *subset* construction.
- The states in $M'$ are subsets of states in $M$. 
We want to show that $M$ accepts $x$ iff $M'$ accepts $x$.

- Let $x = x_1 \ldots x_k$.
- If $M$ accepts $x$, then there is a sequence of states $s_0, \ldots, s_k$ such that $s_k \in F$ and $s_{i+1} \in f(s_i, x_{i+1})$.
  - That's what it means for an NFA $M$ to accept $x$
  - $s_0, \ldots, s_k$ is a possible sequence of states that $M$ goes through on input $x$
    - It's only one possible sequence: $M$ is an NFA

- Define $A_0, \ldots, A_k$ inductively:
  $A_0 = \{s_0\}$ and $A_{i+1} = f'(A_i, x_i)$.
  - Intuitively, $A_i$ is the set of states that $M$ could be in after seeing $x_1 \ldots x_i$
    - Remember: a state in $M'$ is a set of states in $M$.
    - $M'$ is deterministic: this sequence is unique.
  - An easy induction shows that $s_i \in A_i$.
  - Therefore $s_k \in A_k$, so $A_k \cap F \neq \emptyset$.
  - Conclusion: $A_k \in F'$, so $M'$ accepts $x$. 
For the converse, suppose that $M'$ accepts $x$

- Let $A_0, \ldots, A_k$ be the sequence of states that $M'$ goes through on input $x$.
- Since $A_k \cap F \neq \emptyset$, there is some $t_k \in A_k \cap F$.
- By induction, if $1 \leq j \leq k$, can find $t_{k-j} \in A_{k-j}$ such that $t_{k-j+1} \in f(t_{k-j}, x_{k-j})$.
- Since $A_0 = \{s_0\}$, we must have $s_0 = t_0$.
- Thus, $t_0 \ldots t_k$ is an “accepting path” for $x$ in $M$
- Conclusion: $M$ accepts $x$
Notes:

▶ Michael Rabin and Dana Scott won a Turing award for defining NFAs and showing they are equivalent to DFAs
▶ This construction blows up the number of states:
  ▶ $|S'| = 2^{|S|}$
  ▶ Sometimes you can do better; in general, you can't
Regular Languages and Finite Automata

Some notation:

- Language $L$ is regular iff $L = L(E)$ for some regexp $E$.
- $L(M)$ is the language accepted by the automaton $M$

**Theorem:** $L = L(M)$ for some automaton $M$ iff $L$ is regular.

First we’ll show that every regular language is accepted by some finite automaton:

**Proof:** We show that $L(E)$ is regular by induction on the (length/structure) of $E$. We need to show that

- $\emptyset = L(\emptyset) = L(M)$ for some finite automaton $M$
  - Easy: build an automaton where no input ever reaches a final state
- $\{\lambda\} = L(\lambda) = L(M)$ for some finite automaton $M$
  - Easy: $M$ has two states, $s_0$ and $s_1$, $s_0$ is the only accepting state, but every non-empty string ends leads to $s_1$.
- For each $x \in I$, $\{x\} = L(x) = L(M)$ for some automaton $M$
  - Easy: an automaton with states $\{s_0, s_1, s_2\}$, only $s_1$ is an accepting state, $x$ leads from $s_0$ to $s_1$, all other nonempty strings lead to $s_2$. 
\[ L(E_1 E_2) = L(M) \text{ for some } M. \text{ Suppose that } L(E_1) = A, \]
\[ L(E_2) = B, A = L(M_A), \text{ and } B = L(M_B). \text{ We must show that } \]
\[ AB = L(M) \text{ for some automaton } M \]

**Proof:** Suppose that \( M_A = (S_A, I, f_A, s_A, F_A) \) and \( M_B = (S_B, I, f_B, s_B, F_B) \). Suppose that \( M_A \) and \( M_B \) and NFAs, and \( S_A \) and \( S_B \) are disjoint (without loss of generality).

Idea: We hook \( M_A \) and \( M_B \) together. Let NFA \( M_{AB} = (S_A \cup S_B, I, f_{AB}, s_A, F_{AB}) \), where

\[
F_{AB} = \begin{cases}
F_B \cup F_A & \text{if } \lambda \in B; \\
F_B & \text{otherwise}
\end{cases}
\]

\( t \in f_{AB}(s, i) \) if either

\[
\begin{align*}
&\quad s \in S_A \text{ and } t \in f_A(s, i), \text{ or} \\
&\quad s \in S_B \text{ and } t \in f_B(s, i), \text{ or} \\
&\quad s \in F_A \text{ and } t \in f_B(s_B, i) \text{ ("switch" from } M_A \text{ to } M_B) \\
\end{align*}
\]

Idea: given input \( xy \in AB \), the machine “guesses” when to switch from running \( M_A \) to running \( M_B \).

\[ L(M_{AB}) = AB. \]
Proof: There are two parts to this proof:

1. Showing that if \( x \in AB \), then \( x \in L(M_{AB}) \).
2. Show that if \( x \in L(M_{AB}) \), then \( x \in AB \).

For part 1, suppose that \( x = ab \in AB \), where \( a = a_1 \ldots a_k \) and \( b = b_1 \ldots b_m \). Then there exists an accepting path for \( a \) and one for \( b \); that is, a sequence of states \( s_0, \ldots, s_k \in S_A \) and a sequence of states \( t_0, \ldots, t_m \in S_B \) such that

- \( s_0 = s_A \) and \( t_0 = s_B \);
- \( s_{i+1} \in f_A(s_i, a_{i+1}) \) and \( t_{i+1} \in f_B(t_i, b_{i+1}) \);
- \( s_k \in F_A \) and \( t_m \in F_B \).

That means that after reading \( a \), \( M_{AB} \) could be in state \( s_k \). If \( b = \lambda \), \( M_{AB} \) accepts \( a \) (since \( s_k \in F_A \subseteq F_{AB} \) if \( \lambda \in B \)). Otherwise, \( M_{AB} \) can continue to \( t_1, \ldots, t_m \) when reading \( b \), so it accepts \( ab \) (since \( t_m \in F_B \subseteq F_{AB} \)).

- is, \( s_0, \ldots, s_k, t_1, \ldots, t_m \) is an accepting path for \( ab \)
  - Note that there is no \( t_0 \); we go from \( s_k \) to \( t_1 \)
For part 2, suppose that $x = c_1 \ldots c_n$ is accepted by $M_{AB}$. That means that there is a sequence of states $s_0, \ldots, s_n \in S_A \cup S_B$ such that

- $s_0 = s_A$
- $s_{i+1} \in f_{AB}(s_i, c_{i+1})$
- $s_n \in F_{AB}$

If $s_n \in F_A$, then $\lambda \in B$, $s_0, \ldots, s_n \subseteq S_A$ (since once $M_{AB}$ moves to a state in $S_B$, it never moves to a state in $S_A$), so $x$ is accepted by $M_A$. Thus, $x \in A \subseteq AB$. 
For part 2, suppose that \( x = c_1 \ldots c_n \) is accepted by \( M_{AB} \). That means that there is a sequence of states \( s_0, \ldots, s_n \in S_A \cup S_B \) such that

- \( s_0 = s_A \)
- \( s_{i+1} \in f_{AB}(s_i, c_{i+1}) \)
- \( s_n \in F_{AB} \)

If \( s_n \in F_A \), then \( \lambda \in B \), \( s_0, \ldots, s_n \subseteq S_A \) (since once \( M_{AB} \) moves to a state in \( S_B \), it never moves to a state in \( S_A \)), so \( x \) is accepted by \( M_A \). Thus, \( x \in A \subseteq AB \).

If \( s_n \in F_B \), let \( s_j \) be the first state in the sequence in \( S_B \). Then \( s_0, \ldots, s_{j-1} \subseteq S_A \), \( s_{j-1} \in F_A \), so \( c_1 \ldots c_{j-1} \) is accepted by \( M_A \), and hence is in \( A \). Moreover, \( s_B, s_j, \ldots, s_n \subseteq S_B \) (once \( M_{AB} \) is in a state of \( S_B \), it never moves to a state of \( S_A \)), so \( c_j \ldots c_n \) is accepted by \( M_B \), and hence is in \( B \). Thus, \( x = (c_1 \ldots c_{j-1})(c_j \ldots c_n) \in AB \).
if $A = L(M_A)$ and $B = L(M_B)$, then $A \cup B = L(M)$ for some automaton $M$.

**Proof:** Suppose that $M_A = (S_A, I, f_A, s_A, F_A)$ and $M_B = (S_B, I, f_B, s_B, F_B)$. Suppose that $M_A$ and $M_B$ and NFAs, and $S_A$ and $S_B$ are disjoint.

Idea: given input $x \in A \cup B$, the machine “guesses” whether to run $M_A$ or $M_B$.

- $M_{A \cup B} = (S_A \cup S_B \cup \{s_0\}, I, f_{A \cup B}, s_0, F_{A \cup B})$, where
  - $s_0$ is a new state, not in $S_A \cup S_B$
  - $f_{A \cup B}(s, i) = \begin{cases} f_A(s, i) & \text{if } s \in S_A \\ f_B(s, i) & \text{if } s \in S_B \\ f_A(s_A, i) \cup f_B(s_B, i) & \text{if } s = s_0 \end{cases}$
  - $F_{A \cup B} = \begin{cases} F_A \cup F_B \cup \{s_0\} & \text{if } \lambda \in A \cup B \\ F_A \cup F_B & \text{otherwise.} \end{cases}$

We have to prove that $L(M_{A \cup B}) = A \cup B$; this is straightforward.
if $A = L(M_A)$, then $A^* = L(M)$ for some $M$.

Suppose that $M_A = (S_A, I, f_A, s_A, F_A)$

$M_A^* = (S_A \cup \{s_0\}, I, f_A^*, s_0, F_A \cup \{s_0\})$, where

$s_0$ is a new state, not in $S_A$;

$f_{A^*}(s, i) = \begin{cases} f_A(s, i) & \text{if } s \in S_A - F_A; \\ f_A(s, i) \cup f_A(s_A, i) & \text{if } s \in F_A; \\ f_A(s_A, i) & \text{if } s = s_0 \end{cases}$

We now have to prove that $L(M_{A^*}) = A^*$.

Homework!
Next we’ll show that every language accepted by a finite automaton is regular:

**Proof:** Fix an automaton $M$ with states $\{s_0, \ldots, s_n\}$. Can assume wlog (without loss of generality) that $M$ is deterministic.

- a language is accepted by a DFA iff it is accepted by a NFA.

Let $S(s_i, s_j, k)$ be the set of strings that force $M$ from state $s_i$ to $s_j$ on a path such that every intermediate state is $\{s_0, \ldots, s_k\}$.

- E.g., $S(s_4, s_5, 2)$ consists of all strings that force $M$ from $s_4$ to $s_5$ on a path that goes through only $s_0$, $s_1$, and $s_2$ (in any order, perhaps with repeats).

Note that a string $x$ is accepted by $M$ iff $x \in S(s_0, s, n)$ for some final state $s$. Thus, $L(M)$ is the union over all final states $s$ of $S(s_0, s, n)$.
We will prove by induction on $k$ that $S(s_i, s_j, k)$ is regular.

- Why not just take $s_i = s_0$?
  - We need a stronger induction hypothesis
We will prove by induction on $k$ that $S(s_i, s_j, k)$ is regular.

- Why not just take $s_i = s_0$?
  - We need a stronger induction hypothesis

Base case:

**Lemma 1**: $S(s_i, s_j, -1)$ is regular.

**Proof**: For a string $\sigma$ to be in $S(s_i, s_j, -1)$, it must go directly from $s_i$ to $s_j$, without going through any intermediate strings. Thus, $\sigma$ must be some subset of $I$ (possibly empty) together with $\lambda$ if $s_i = s_j$. Either way, $S(s_i, s_j, -1)$ is regular.
Lemma 2: If $s_j \neq s_{k+1}$, then $S(s_i, s_j, k + 1) = S(s_i, s_j, k) \cup S(s_i, s_{k+1}, k)(S(s_{k+1}, s_{k+1}, k))^* S(s_{k+1}, s_j, k)$. 
Lemma 2: If \( s_j \neq s_{k+1} \), then \( S(s_i, s_j, k + 1) = S(s_i, s_j, k) \cup S(s_i, s_{k+1}, k)(S(s_{k+1}, s_{k+1}, k))^* S(s_{k+1}, s_j, k) \).

Proof: If a string \( \sigma \) forces \( M \) from \( s_i \) to \( s_j \) on a path with intermediates states all in \( \{s_0, \ldots, s_{k+1}\} \), then the path either does not go through \( s_{k+1} \) at all, so is in \( S(s_i, s_j, k) \), or goes through \( s_{k+1} \) some finite number of times, say \( m \). That is, the path looks like this:

\[
\begin{align*}
\quad s_i & \ldots s_{k+1} \ldots s_{k+1} \ldots s_{k+1} \ldots \ldots \ldots s_j \\
where all the states in the & \ldots \text{part are in } \{s_0, \ldots, s_k\}. \quad \text{Thus, we can split up the string } \sigma \text{ into } m + 1 \text{ corresponding pieces:} \\
& \quad \sigma_0 \text{ that takes } M \text{ from } s_i \text{ to } s_{k+1}, \\
& \quad \text{each of } \sigma_1, \ldots, \sigma_m \text{ take } M \text{ from } s_{k+1} \text{ back to } s_{k+1} \\
& \quad \sigma_{m+1} \text{ takes } M \text{ from } s_{k+1} \text{ to } s_j. \\
\text{Thus,} \\
& \quad \sigma_0 \in S(s_i, s_{k+1}, k) \\
& \quad \sigma_1, \ldots, \sigma_m \text{ are all in } S(s_{k+1}, s_{k+1}, k) \\
& \quad \sigma_{m+1} \in S(s_{k+1}, s_j, k) \\
& \quad \text{So } \sigma = \sigma_0 \sigma_1 \ldots \sigma_{m+1} \in \quad S(s_i, s_j, k) \cup S(s_i, s_{k+1}, k)(S(s_{k+1}, s_{k+1}, k))^* S(s_{k+1}, s_j, k)
Lemma 3: If $s_j = s_{k+1}$, then
$S(s_i, s_j, k + 1) = S(s_i, s_j, k) \cup S(s_i, s_j, k)(S(s_j, s_j, k))^*$.

Proof: Same idea as previous proof.
Lemma 3: If \( s_j = s_{k+1} \), then 
\[
S(s_i, s_j, k + 1) = S(s_i, s_j, k) \cup S(s_i, s_j, k)(S(s_j, s_j, k))^*.
\]

Proof: Same idea as previous proof.
**Lemma 3:** If \( s_j = s_{k+1} \), then
\[
S(s_i, s_j, k + 1) = S(s_i, s_j, k) \cup S(s_i, s_j, k)(S(s_j, s_j, k))^*.
\]

**Proof:** Same idea as previous proof.

**Lemma 4:** \( S(s_i, s_j, N) \) is regular for all \( N \) with \(-1 \leq N \leq n\).

**Proof:** An easy induction. Lemma 1 gives the base case; Lemmas 2 and 3 give the inductive step.
Lemma 3: If $s_j = s_{k+1}$, then
$S(s_i, s_j, k + 1) = S(s_i, s_j, k) \cup S(s_i, s_j, k)(S(s_j, s_j, k))^*.$

Proof: Same idea as previous proof.

Lemma 4: $S(s_i, s_j, N)$ is regular for all $N$ with $-1 \leq N \leq n$.

Proof: An easy induction. Lemma 1 gives the base case; Lemmas 2 and 3 give the inductive step.

The language accepted by $M$ is the union of the sets $S(s_0, s', n)$ such that $s'$ is a final state. Since regular languages are closed under union, the result follows.
We can use the ideas of this proof to compute the regular language accepted by an automaton.

We can methodically build up $S(s_0, s_0, 2)$, which is what we want (since $s_3$ is unreachable).
A Non-Regular Language

Not every language is regular (which means that not every language can be accepted by a finite automaton).

**Theorem:** $L = \{0^n1^n : n = 0, 1, 2, \ldots \}$ is not regular.

**Proof:** Suppose, by way of contradiction, that $L$ is regular. Then there is a DFA $M = (S, \{0, 1\}, f, s_0, F)$ that accepts $L$. Suppose that $M$ has $N$ states. Let $s_0, \ldots, s_{2N}$ be the set of states that $M$ goes through on input $0^N1^N$.

- Thus $f(s_i, 0) = s_{i+1}$ for $i = 0, \ldots, N$.

Since $M$ has $N$ states, by the pigeonhole principle (remember that?), at least two of $s_0, \ldots, s_N$ must be the same. Suppose it’s $s_i$ and $s_j$, where $i < j$, and $j - i = t$.

**Claim:** $M$ accepts $0^N0^t1^N$, and $0^N0^{2t}1^N$, $0^N0^{3t}1^N$.

**Proof:** Starting in $s_0$, $0^i$ brings the machine to $s_i$; another $0^t$ bring the machine back to $s_i$ (since $s_j = s_{i+t} = s_i$); another $0^t$ bring machine back to $s_i$ again. After going around the loop for a while, the can continue to $s_N$ and accept.
The Pumping Lemma

The techniques of the previous proof generalize. If $M$ is a DFA and $x$ is a string accepted by $M$ such that $|x| \geq |S|$

- $|S|$ is the number of states; $|x|$ is the length of $x$
then there are strings $u$, $v$, $w$ such that

- $x = uvw$,
- $|uv| \leq |S|$,
- $|v| \geq 1$,
- $uv^i w$ is accepted by $M$, for $i = 0, 1, 2, \ldots$.

The proof is the same as on the previous slide.

- $x$ was $0^n1^n$, $u = 0^i$, $v = 0^t$, $w = 0^{N-t-i}1^N$.

We can use the Pumping Lemma to show that many languages are not regular

- $\{1^{n^2} : n = 0, 1, 2, \ldots\}$: homework
- $\{0^{2n}1^n : n = 0, 1, 2, \ldots\}$: homework
- $\{1^n : n \text{ is prime}\}$
- ...
More Powerful Machines

Finite automata are very simple machines.

- They have no memory
- Roughly speaking, they can’t count beyond the number of states they have.

*Pushdown automata* have states and a *stack* which provides unlimited memory.

- They can recognize all languages generated by *context-free grammars* (CFGs)
  - CFGs are typically used to characterize the syntax of programming languages
- They can recognize the language $\{0^n1^n : n = 0, 1, 2, \ldots\}$, but not the language $L’ = \{0^n1^n2^n : n = 0, 1, 2, \ldots\}$

*Linear bounded automata* can recognize $L’$.

- More generally, they can recognize *context-sensitive grammars* (CSGs)
- CSGs are (almost) good enough to characterize the grammar of real languages (like English)
Most general of all: Turing machine (TM)

- Given a *computable* language, there is a TM that accepts it.
- This is essentially how we define computability.

If you’re interested in these issues, take CS 4810!