



FINITE AUTOMATA AND REGULAR LANGUAGES

DISCRETE MATHEMATICS AND THEORY 2

MARK FLORYAN

GOALS!

1. Quick definition of languages...a different way to think about functions AND introduction to the Chomsky Hierarchy.
2. Our first model of computation, the finite automata!! How does it work? How do we prove things about what functions it can compute? Etc...
3. Are there equivalent models / expressions that are equivalent to the finite automata? How do we identify functions that can NOT be computed by these?

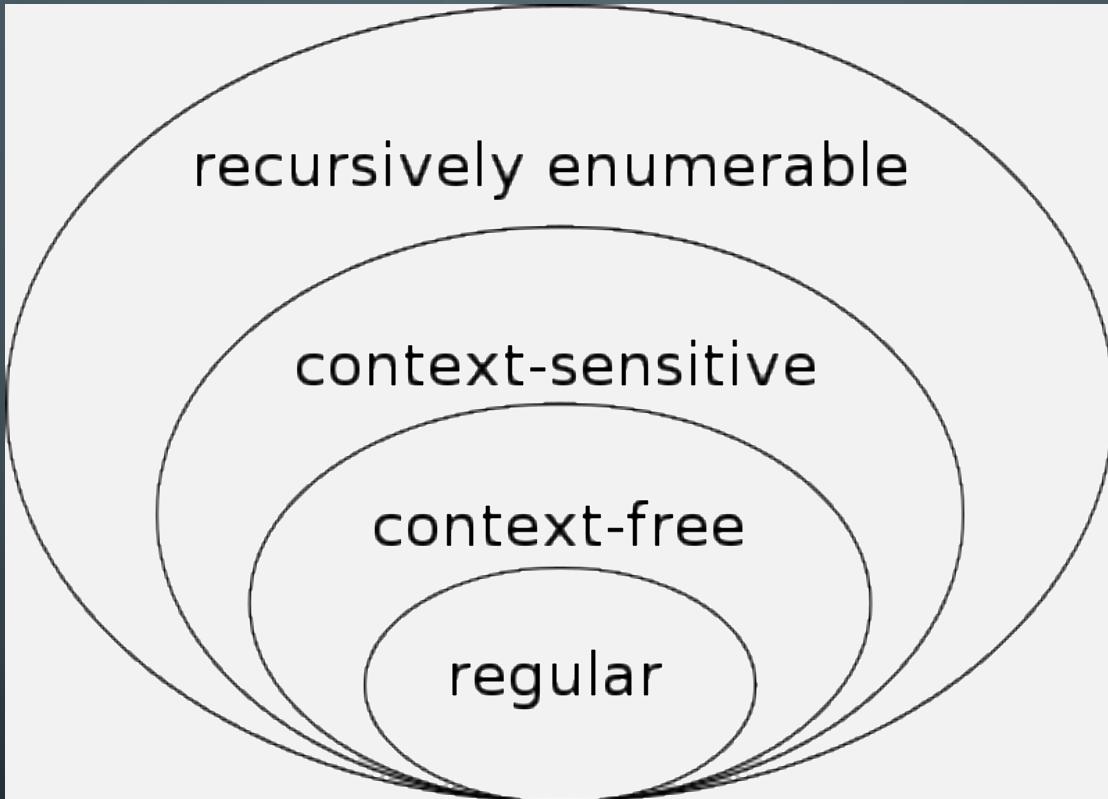
PART 1: FUNCTIONS, LANGUAGES, AND THE CHOMSKY HIERARCHY

TYPES OF PROBLEMS

Name	Decision Problem	Function	Language
XOR	Are there an odd number of 1's?	$f(b) = \begin{cases} 0 & \text{number of 1s is even} \\ 1 & \text{number of 1s is odd} \end{cases}$	$\{b \in \Sigma^* b \text{ has an odd number of 1s}\}$
Majority	Are there more 1s than 0s?	$f(b) = \begin{cases} 0 & \text{more 0s than 1s} \\ 1 & \text{more 1s than 0s} \end{cases}$	$\{b \in \Sigma^* b \text{ has more 1s than 0s}\}$
Thing you want to compute	Does it have have a property?	$f(b) = 1 \text{ if it does have the property}$	$\{b \in \Sigma^* b \text{ has the property}\}$
Is1	Is the string exactly “1”?	$f(b) = 1 \text{ if } b == 1$	{1}
Is_infinite	Is the length of the string infinite?	$f(b) = 0$	\emptyset

CHOMSKY HIERARCHY

A description of languages and their relationship to one another



In this deck, we will see the regular languages and the machines that recognize them

Each language has a computational model that recognizes it

PART 2: FINITE AUTOMATA AND REGULAR LANGUAGES

INTRODUCTION: WHAT IS A FINITE STATE MACHINE

FINITE STATE MACHINES

- First model of computation that we will look at in detail.
- Features:
 - Has a **VERY** limited amount of memory. What can we compute with such limited memory?
 - What input / output does this machine support? Does this matter?
 - Can we find at least one function that this machine **CANNOT** compute?

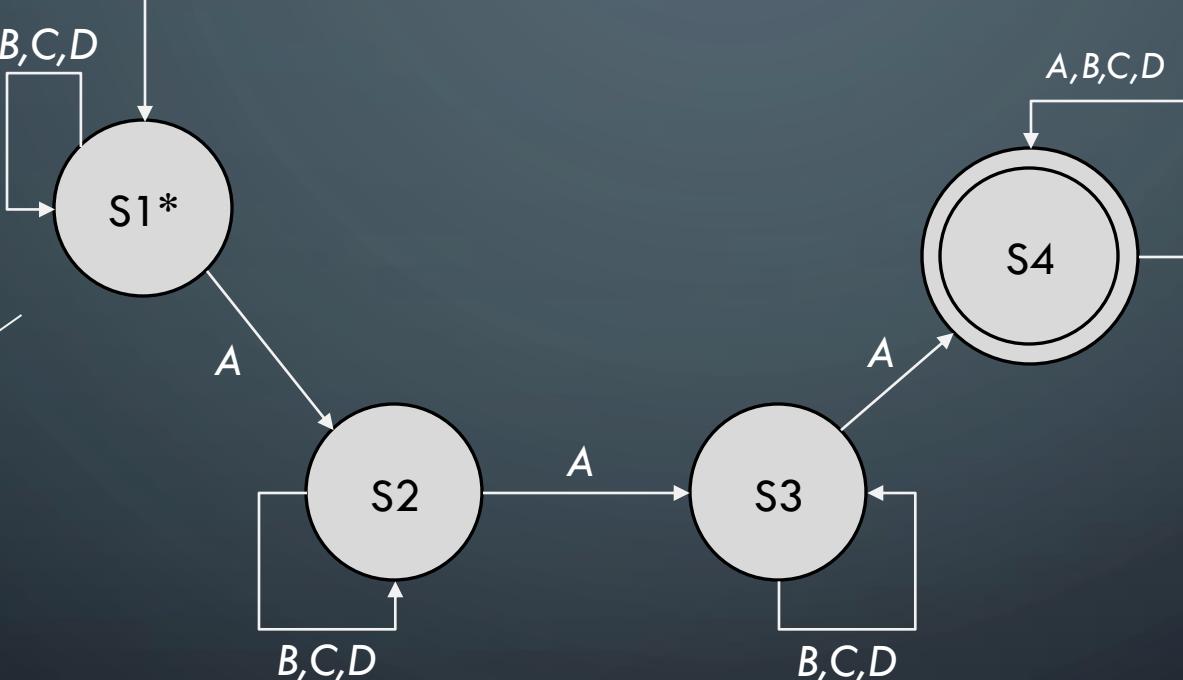
DETERMINISTIC FINITE AUTOMATA (DFA)

Accepts Input as a string

Example Input: AABCDAABCCC

When one character of input is read, machine transitions to a new state

One state is the special start state



One or more states are considered accepting states

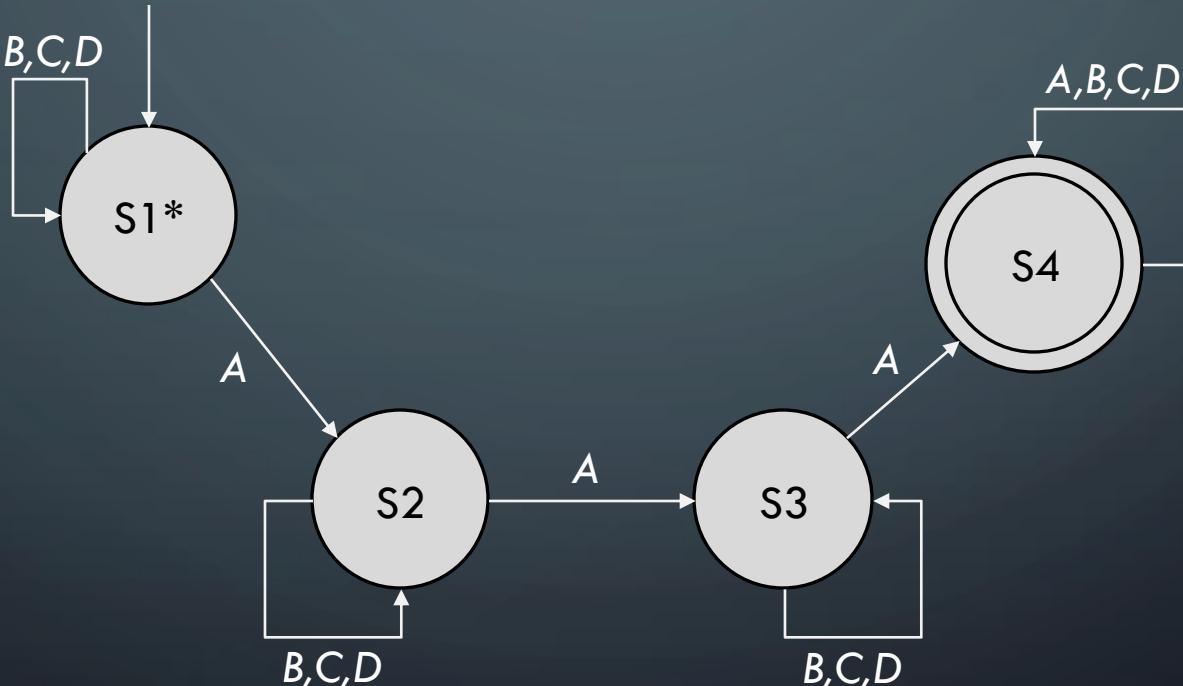
Machine has some number of states (4 in this example) in can be in. Machine is only in one state at a time. This is the ONLY variable / memory DFAs have

DETERMINISTIC FINITE AUTOMATA (DFA)

Accepts Input as a string

Example Input: AABCDAABCCC

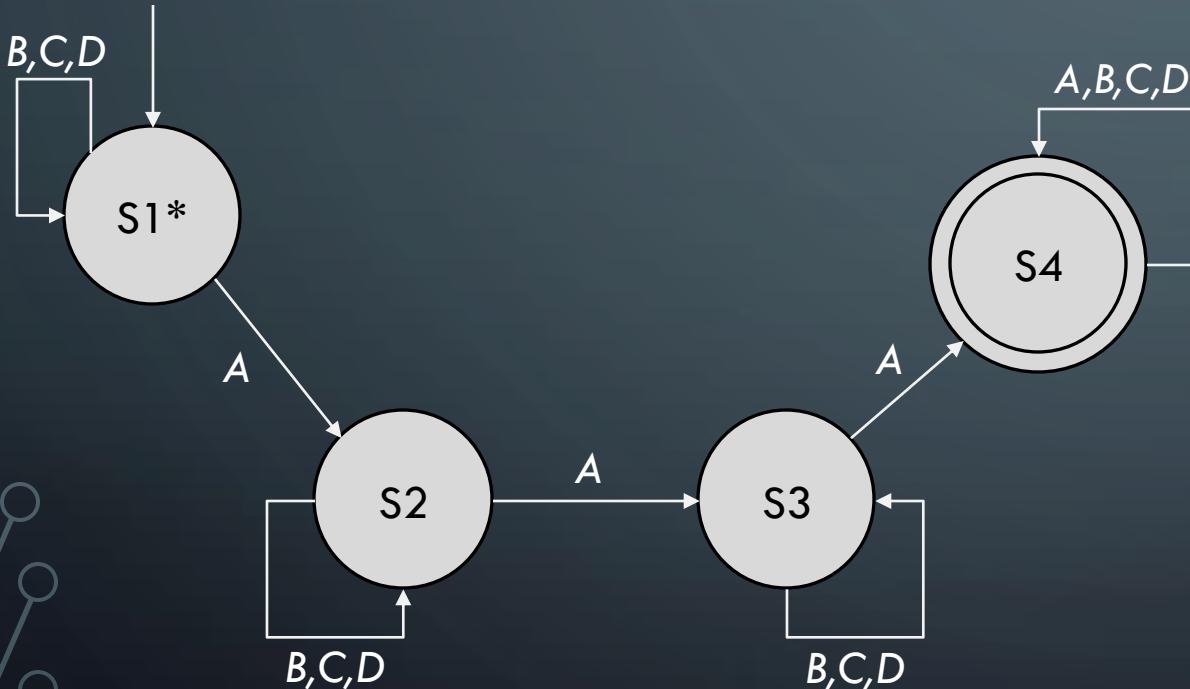
Let's step through the execution of this machine



DETERMINISTIC FINITE AUTOMATA (DFA)

Accepts Input as a string

Example Input: AABCDAAABC



The formal definition of a deterministic finite state machine is:

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

Where:

Q is a finite set called the states

Σ is a finite set called the alphabet

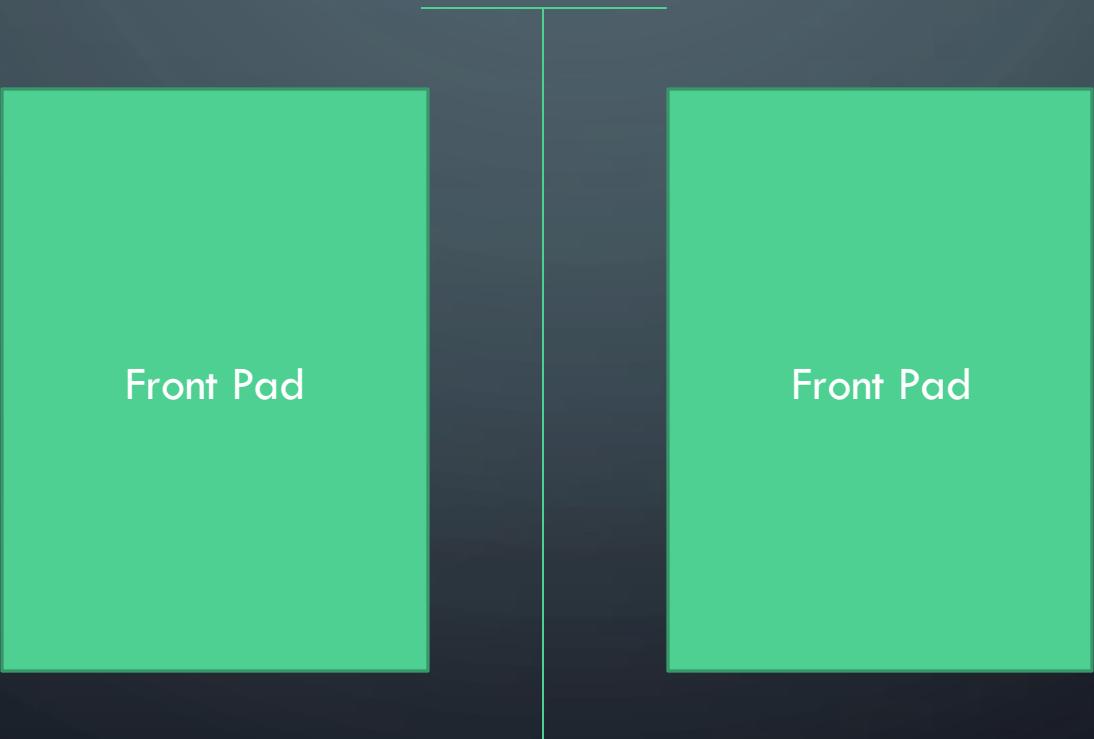
$\delta: Q \times \Sigma \rightarrow Q$ is the transition function

$q_0 \in Q$ is the start state

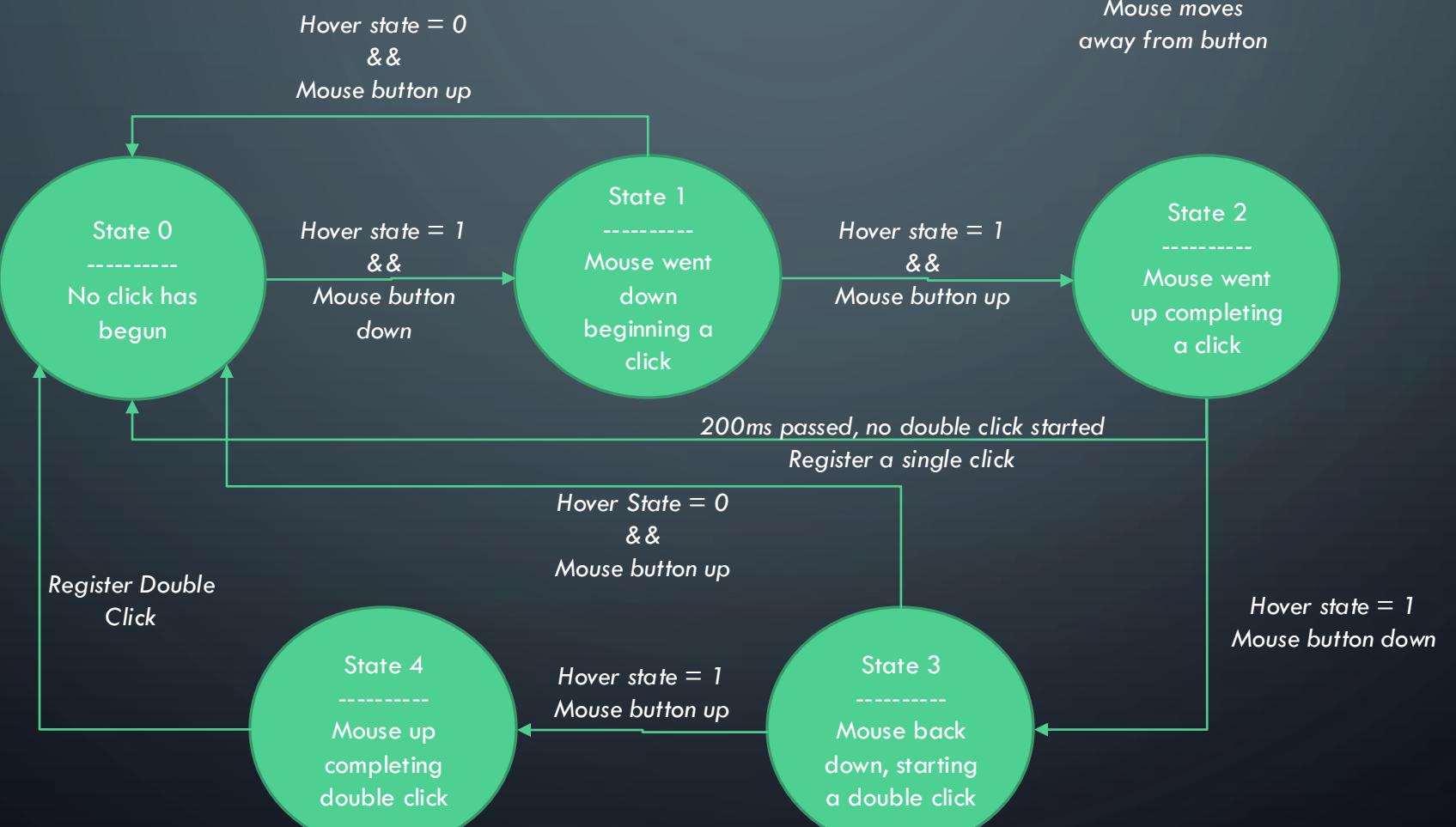
$F \subseteq Q$ is a set of accept states (or final states)

ACTIVITY: DESIGN A STATE MACHINE

- Imaging an automatic sliding door at, say, a grocery store.



ANOTHER EXAMPLE: HOW BUTTONS WORK

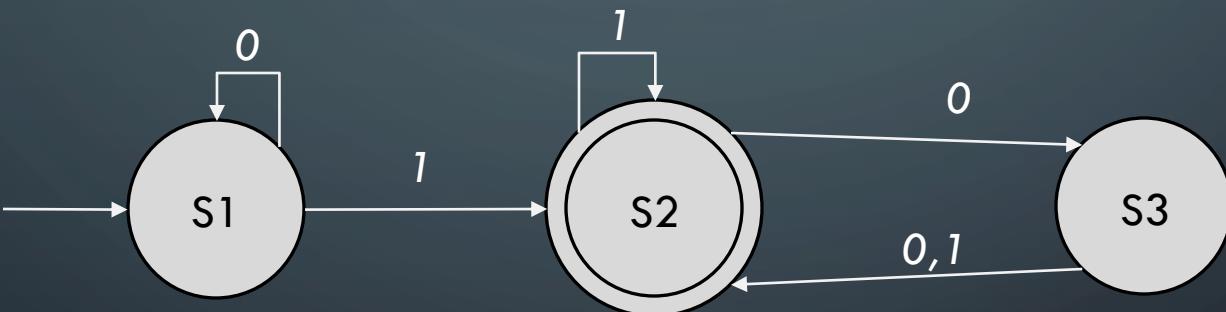


MORE PRACTICE WITH DFA

PRACTICE PROBLEM 1

What set of strings does this machine recognize?

List out the formal description (5-tuple) for this machine



PRACTICE PROBLEM 2

Design a DFA that accepts any binary string that contains “001” as a substring anywhere (possibly multiple times) in the string. The DFA should reject if the string does not contain a contiguous 001 anywhere in the string.

Examples:

01111010101111

REJECT

111100101010111

ACCEPT

FORMAL DEFINITION OF COMPUTATION WITH DFA

FORMAL DEFINITION OF COMPUTATION

*Formal definition of computation on a DFA $M = (Q, \Sigma, \delta, q_0, F)$ on
input string $w = w_1w_2w_3 \dots w_n$ and each $w_i \in \Sigma$*

M accepts the string w if a sequence of states $r_0, r_1, r_2, \dots, r_n \in Q$ exists such that:

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, \dots, n - 1$
3. $r_n \in F$



NON-DETERMINISTIC FINITE STATE AUTOMATA (NFA)

MOTIVATING QUESTION

Does adding a new feature / functionality to our machine / computational model increase the number of functions (or languages) it can recognize?

EXAMPLE: 2-DFA

Imagine a DFA with the following extra feature:

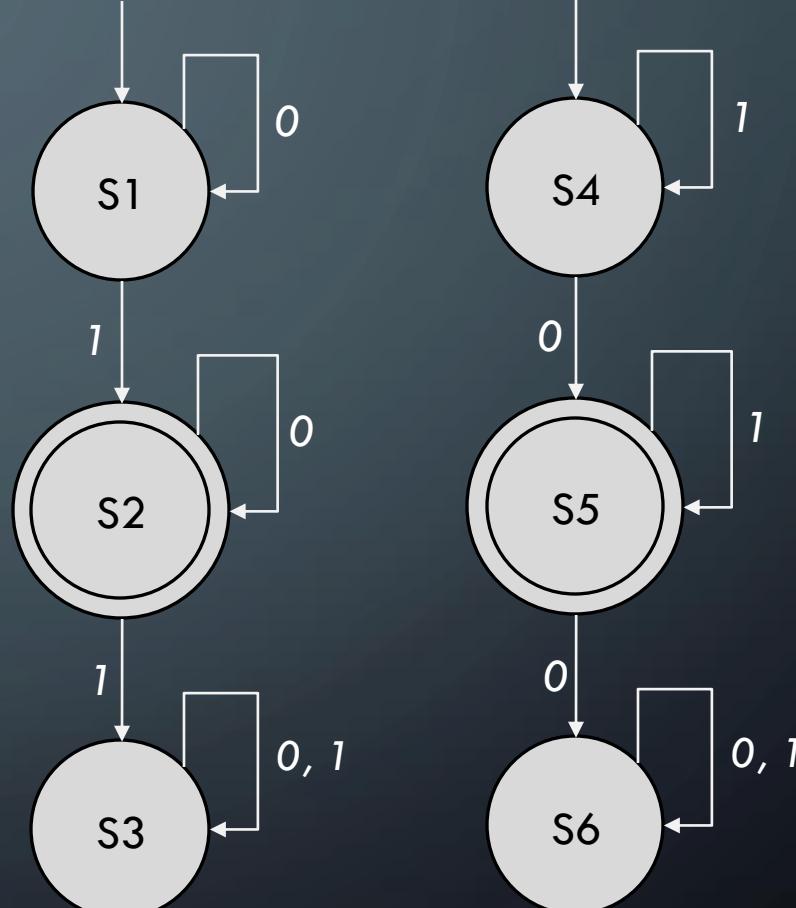
"The machine works exactly as a DFA we have already described, except it can be in up to two states at once."

Notes:

2 start states

Each state transitions after reading each symbol

Machine accepts in either state is in final state at end



In general, what language does this machine recognize?

What does this machine do on
input: 000100

2-DFA VS. DFA?

Which of the following do you think is true?

Possible Theorem 1: A 2-DFA is equivalent in computational power as a traditional DFA

In other words: For any language L , there exists a DFA that accepts it iff there exists a 2-DFA that accepts it (note the if and only if here)

Possible Theorem 2: A 2-DFA has more computational power than a DFA

In other words: There exists at least one language L that can be recognized by a 2-DFA but cannot be accepted by any DFA

2-DFA VS. DFA?

Possible Theorem 1: A 2-DFA is equivalent in computational power as a traditional DFA

In other words: For any language L , there exists a DFA that accepts it iff there exists a 2-DFA that accepts it (note the if and only if here)

Suppose we think
this one is true
(spoiler: it is!)

How do we prove this?

Prove both directions of the claim:

1. If a DFA exists that accepts L , then a 2-DFA exists that accepts L (easy one)
2. If a 2-DFA exists that accepts L , then a DFA exists that accepts L (a little harder)

Basic idea: If one type of machine accepts a language L , can you simulate that machine with the other type?

2-DFA VS. DFA?

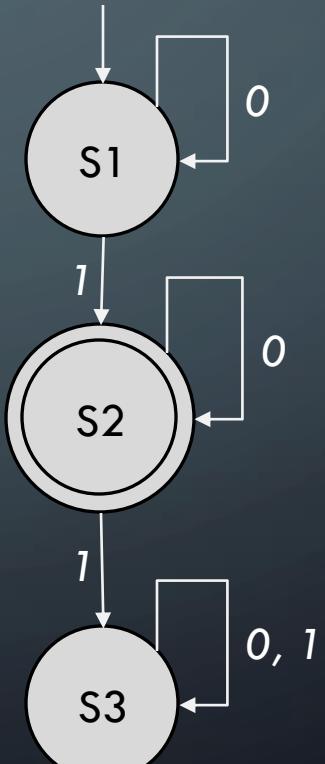
Possible Theorem 1: A 2-DFA is equivalent in computational power as a traditional DFA

Consider direction 1 first:

If a DFA exists that recognizes some language L , then a 2-DFA exists too!



Given a DFA that accepts an arbitrary language L (left), describe the process for turning this into an equivalent 2-DFA (right)



Add a dummy state that the 2-DFA will also be in at all times, doesn't affect the language L that gets recognized

2-DFA VS. DFA?

Possible Theorem 1: A 2-DFA is equivalent in computational power as a traditional DFA

Consider direction 1 first:

If a DFA exists that recognizes some language L , then a 2-DFA exists too!

Consider an arbitrary DFA D that
recognizes an arbitrary language L :

$$D = (Q, \Sigma, \delta, q_0, F)$$

Here is the formal version
of this process

Construct a 2-DFA D' as such:

$$D' = (Q', \Sigma, \delta', \{q_0, q_n\}, F)$$

Such that:

$$Q' = Q \cup \{q_n\}$$

$$\delta' = \delta \cup \{(q_n \times \Sigma) \rightarrow q_n\}$$

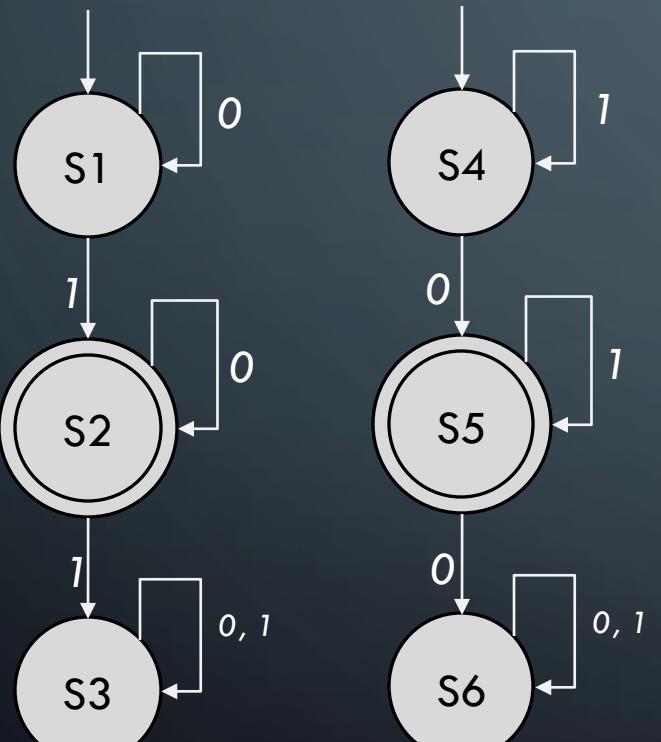
Prove this works: Because D' fulfills all the requirements of a 2-DFA, and executes the exact same way D does (except for being in the dummy second state at all times). Thus, any string that D accepts will also be accepted by D'

2-DFA VS. DFA?

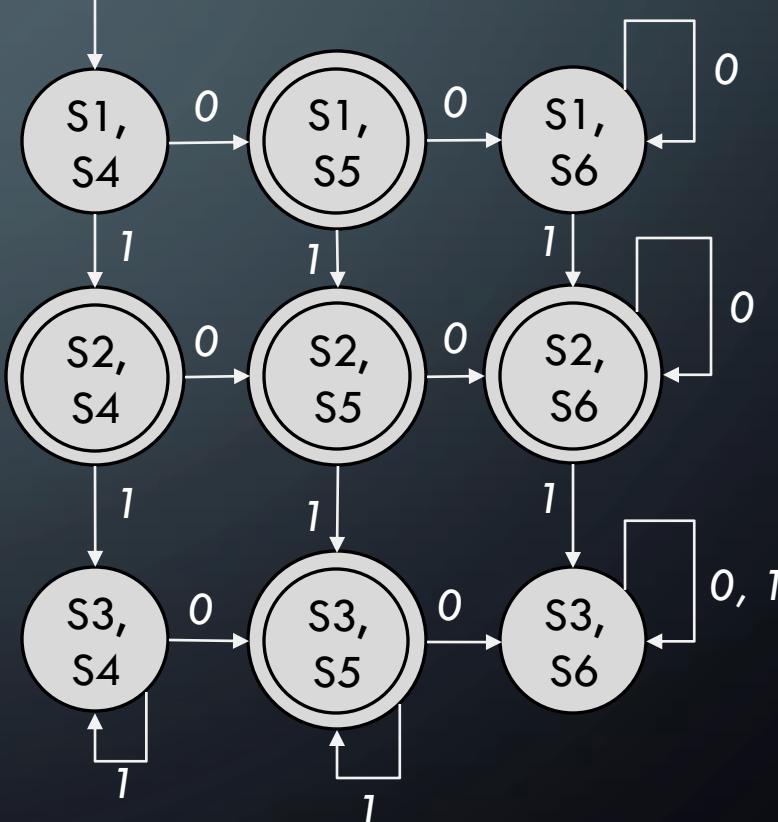
Possible Theorem 1: A 2-DFA is equivalent in computational power as a traditional DFA

Consider direction 2 next:

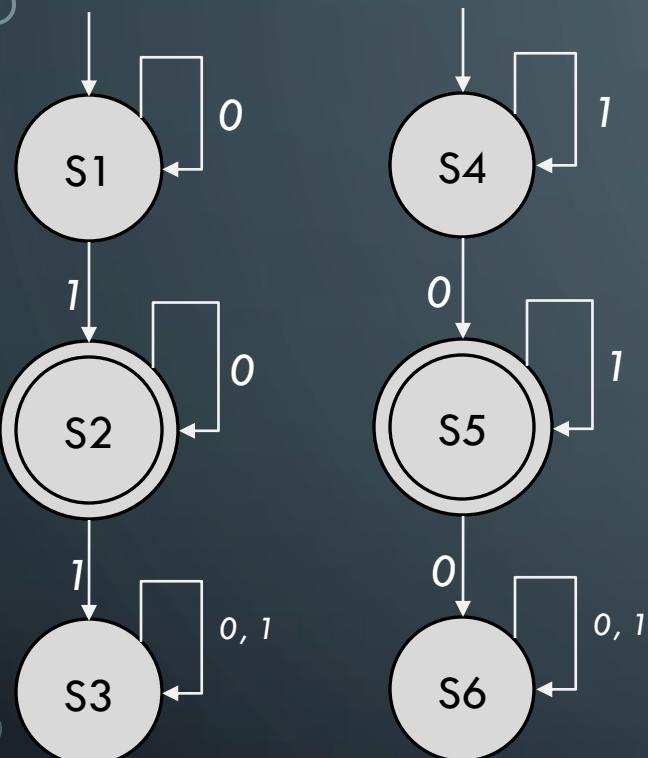
If a 2-DFA exists that recognizes some language L , then a DFA exists too!



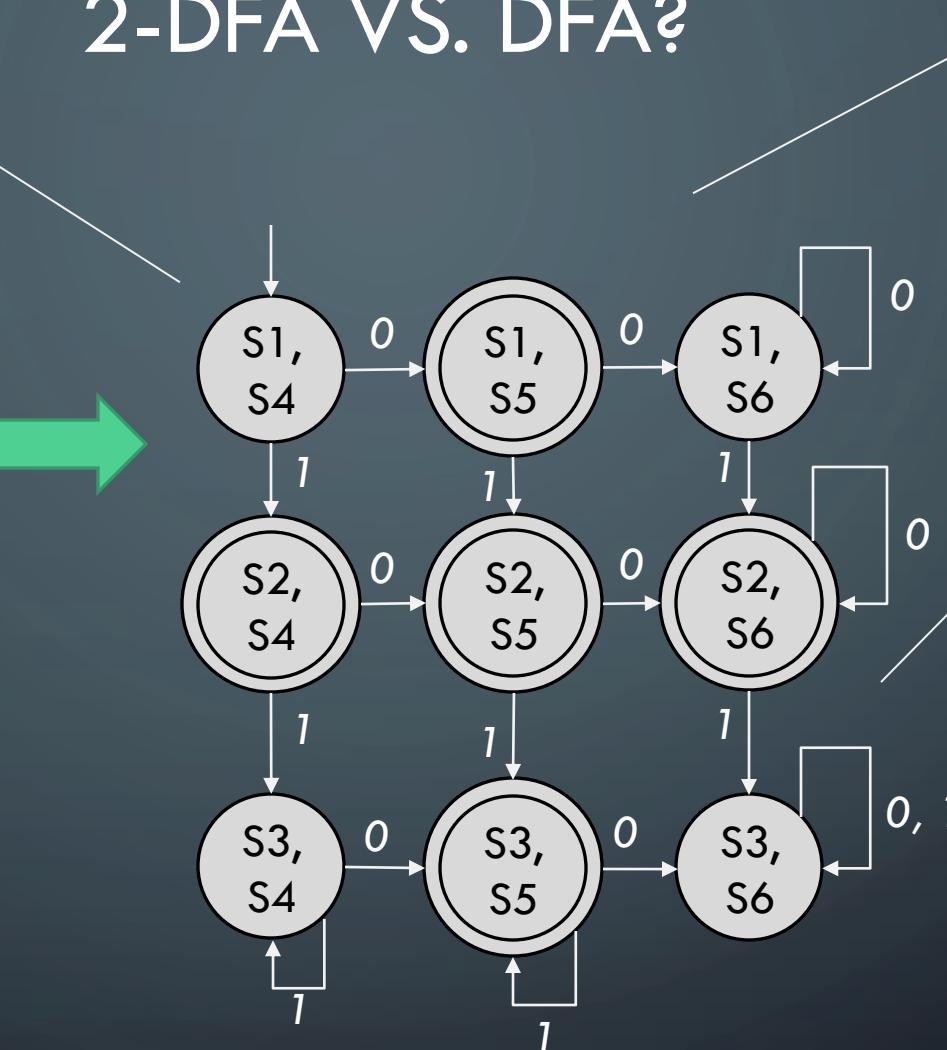
Given a 2-DFA that accepts an arbitrary language L (left), describe the process for turning this into an equivalent DFA (right)



2-DFA VS. DFA?



One start state
(combination of
two start states in
original DFA)



Each state represents
a combination of
states the 2-DFA can
be in

State is final if ONE
of the two was final in
the original DFA

Transitions show how
one symbol leads to a
new combination of 2
states in original DFA

Prove that it works: The new DFA will always accept if the original 2-DFA does because each state in the new DFA represents exactly one pair of states of the original 2-DFA. Thus, with a traditional DFA, we can simulate exactly what the 2-DFA would have done and accept if and only if the original 2-DFA accepts.

2-DFA VS. DFA?

Possible Theorem 1: A 2-DFA is equivalent in computational power as a traditional DFA

Consider direction 2 next:

Here is the formal version of the process

Consider an arbitrary 2-DFA D that recognizes an arbitrary language L :

$$D = (Q, \Sigma, \delta, \{q_0, q_1\}, F)$$



Construct a DFA D' as such:

$$D' = (Q', \Sigma, \delta', q_{0,1}, F')$$

Such that:

$$Q' = \{q_{i,j} \mid q_i, q_j \in Q\}$$

$$\delta' = \{(q_{i,j}, \sigma) \rightarrow q_{i',j'} \mid \sigma \in \Sigma, (q_i, \sigma) \rightarrow q_{i'} \in \delta, (q_j, \sigma) \rightarrow q_{j'} \in \delta\}$$

$$F' = \{q_{i,j} \mid q_i \in F \vee q_j \in F\}$$

2-DFA VS. DFA?

Possible Theorem 1: A 2-DFA is equivalent in computational power as a traditional DFA

Thus, it is proven!!!!

NON-DETERMINISM

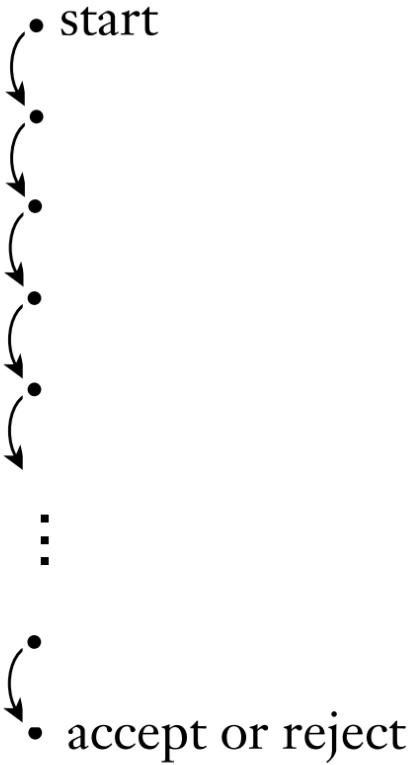
*Let us now look at a different, but similar functionality we can add to the DFA: **Non-Determinism***

Non-Determinism: Is a feature of computational models that allows them to exist in multiple states at one time. The machine can be in any number of it's states simultaneously.

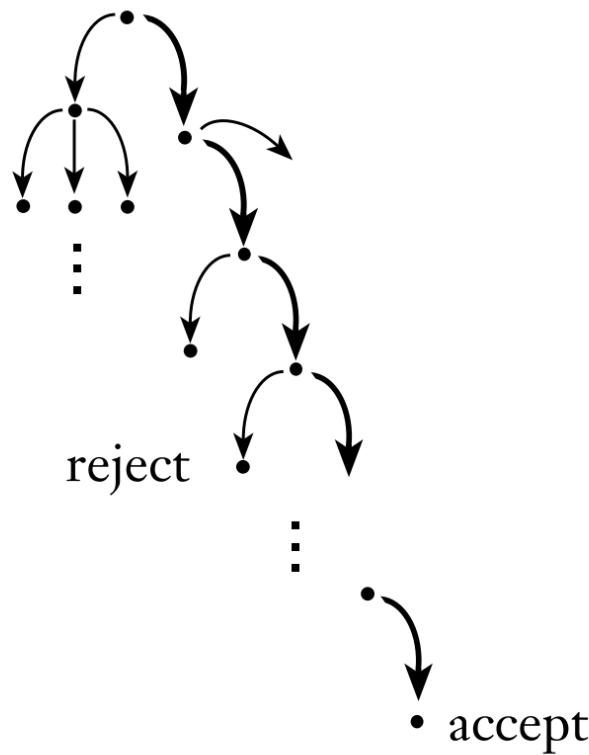
Note that this will be an extension of the 2-DFA. Effectively, an n-DFA where the DFA has n states. The main difference being that an NFA does not HAVE to be in multiple states.

NON-DETERMINISM: INTUITION

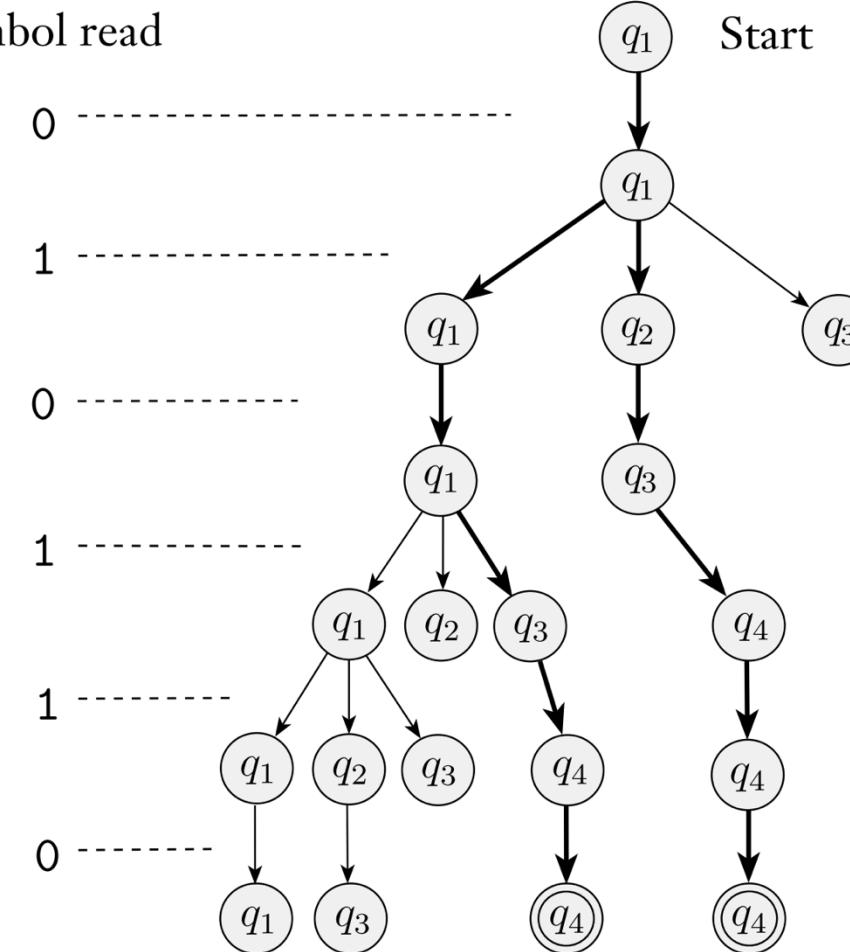
Deterministic computation



Nondeterministic computation



Symbol read



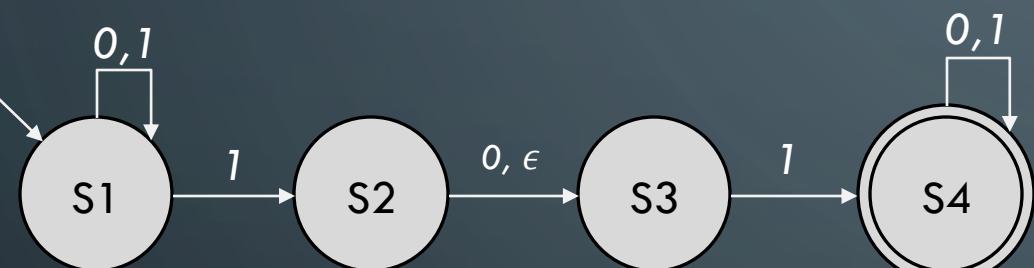
NON-DETERMINISM DEFINITION AND EXAMPLE

A Non-Deterministic Finite Automaton (NFA) is a DFA than can be in multiple states at once.

Formally:

A non-deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

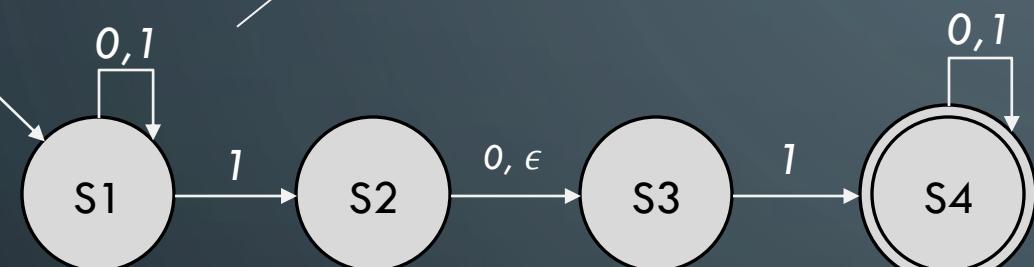
1. Q is a finite set of states
2. Σ is a finite alphabet
3. $\delta: (Q \times \Sigma_\epsilon) \rightarrow \mathcal{P}(Q)$ is the transition function
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept states



Σ_ϵ is the alphabet plus epsilon
(i.e., $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$)

$\mathcal{P}(Q)$ is the power set of Q

NON-DETERMINISM DEFINITION AND EXAMPLE



Note that S_1 has two transitions for input character 1. This means if a 1 is read, two copies of the machine will split off. One transitions to S_1 and the other transitions to S_2 .

ϵ is called an empty transition or an epsilon transition. Machine splits into multiple copies of itself (is in multiple states at once) and the copy transitions to the new state without reading input.

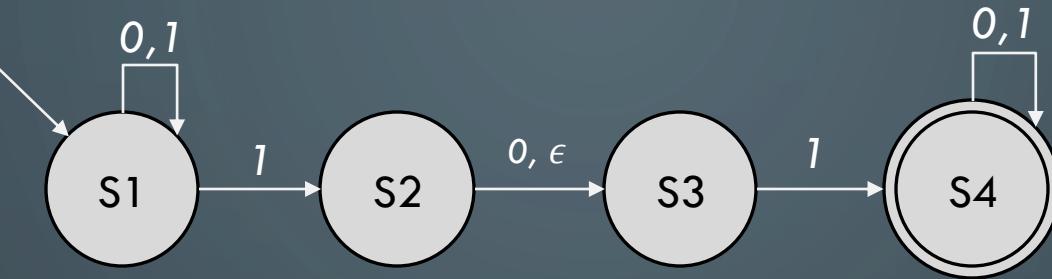
A Non-Deterministic Finite Automaton (NFA) is a DFA than can be in multiple states at once.

Formally:

A non-deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

1. Q is a finite set of states
2. Σ is a finite alphabet
3. $\delta: (Q \times \Sigma^*) \rightarrow P(Q)$ is the transition function
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept states

NON-DETERMINISM DEFINITION AND EXAMPLE



Let's run this machine on some sample input.

Inputs to try:

0000

1101

11

10

10001

In general, what language does this machine recognize?

Answer: Strings that contain 11 or 101 somewhere in them.

NON-DETERMINISM EXAMPLE

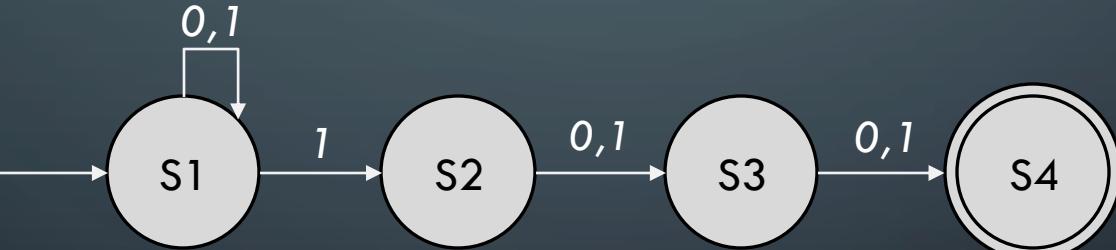
Practice: Develop an NFA that accepts the following language:

Let A be the language consisting of all strings over $\{0,1\}^$ containing a 1 in the third position from the end
(e.g., 000100 is in A but 0011 is not).*

NON-DETERMINISM EXAMPLE

Practice: Develop an NFA that accepts the following language:

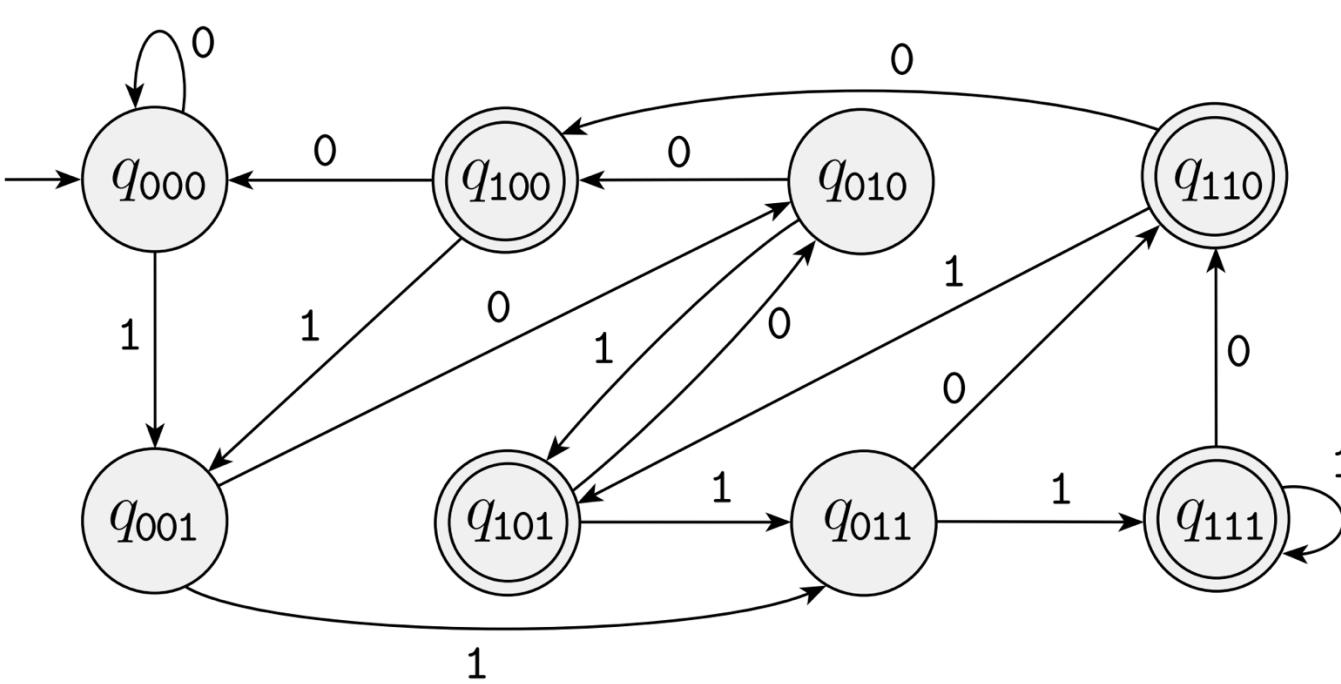
Let A be the language consisting of all strings over $\{0,1\}^*$ containing a 1 in the third position from the end
(e.g., 000100 is in A but 0011 is not).



NON-DETERMINISM EXAMPLE

Practice: Develop an NFA that accepts the following language:

Let A be the language consisting of all strings over $\{0,1\}^*$ containing a 1 in the third position from the end
(e.g., 000100 is in A but 0011 is not).



FYI: Here is a DFA that accepts A .
Oftentimes, using an NFA is much simpler

NON-DETERMINISM EXAMPLE

Practice: Develop an NFA that accepts the following language:

Let A be the language consisting of all strings over $\{0,1\}^$ ending in 101 or ending in 010*

EQUIVALENCE OF NFA AND DFA?

Which of the following do you think is true?

Possible Theorem 1: An NFA is equivalent in computational power to a DFA

In other words: For any language L , there exists a DFA that accepts it iff there exists an NFA that accepts it (note the if and only if here)

Possible Theorem 2: An NFA has more computational power than a DFA

In other words: There exists at least one language L that can be recognized by an NFA but cannot be accepted by any DFA

NFA VS. DFA?

Possible Theorem 1: An NFA is equivalent in computational power to a DFA

In other words: For any language L , there exists a DFA that accepts it iff there exists an NFA that accepts it (note the if and only if here)

Suppose we think
this one is true
(spoiler: it is!)

How do we prove this?

Prove both directions of the claim:

1. If a DFA exists that accepts L , then an NFA exists that accepts L (easy one)
2. If an NFA exists that accepts L , then a DFA exists that accepts L (harder)

Basic idea: If one type of machine accepts a language L , can you simulate that machine with the other type? It is the same (similar) proof as the 2-DFA example!!!

NFA VS. DFA?

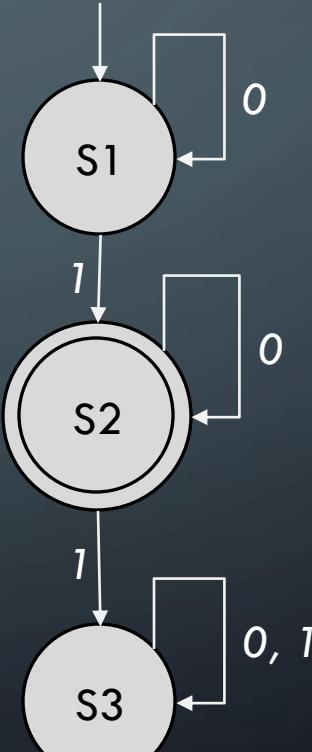
Possible Theorem 1: An NFA is equivalent in computational power to a DFA

Consider direction 1 first:

If a DFA exists that recognizes some language L , then an NFA exists too!



Given a DFA that accepts an arbitrary language L (left), describe the process for turning this into an equivalent NFA (right)



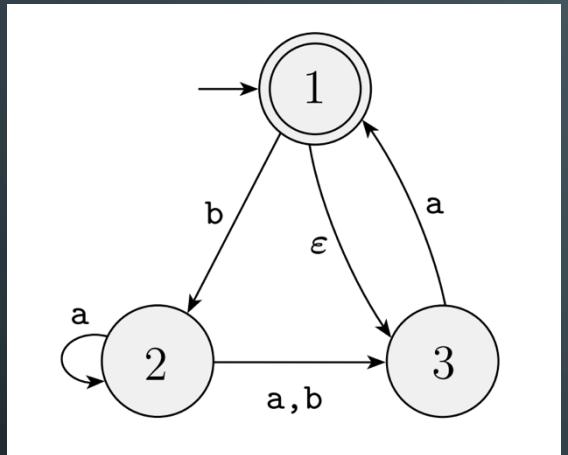
Trivial, a DFA is a valid NFA by definition, so don't actually need to change anything!!

NFA VS. DFA?

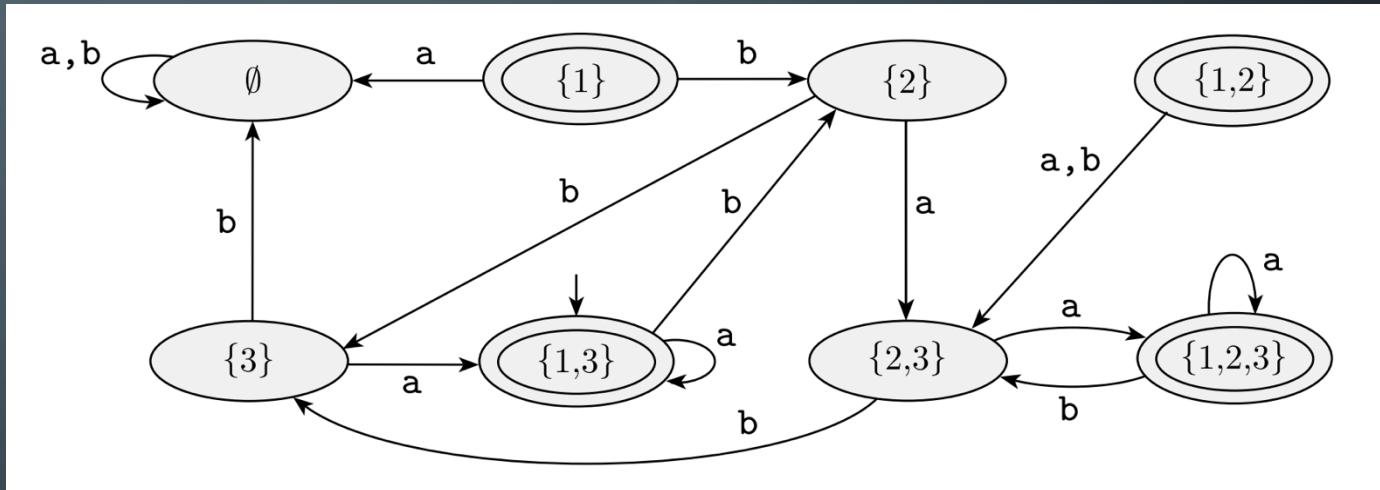
Possible Theorem 1: An NFA is equivalent in computational power to a DFA

Consider direction 2 next:

If an NFA exists that recognizes some language L , then a DFA exists that also accepts L !



Given an NFA that accepts an arbitrary language L (left), describe the process for turning this into an equivalent DFA (right)



States are every possible subset of states in the original NFA.

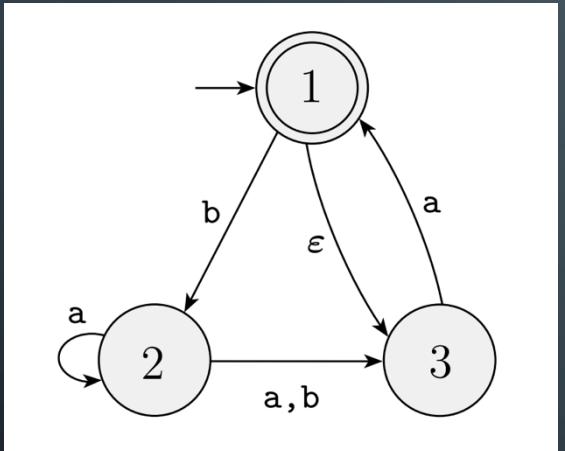
When character is read from input, we transition to a new subset of states the NFA was in

NFA VS. DFA?

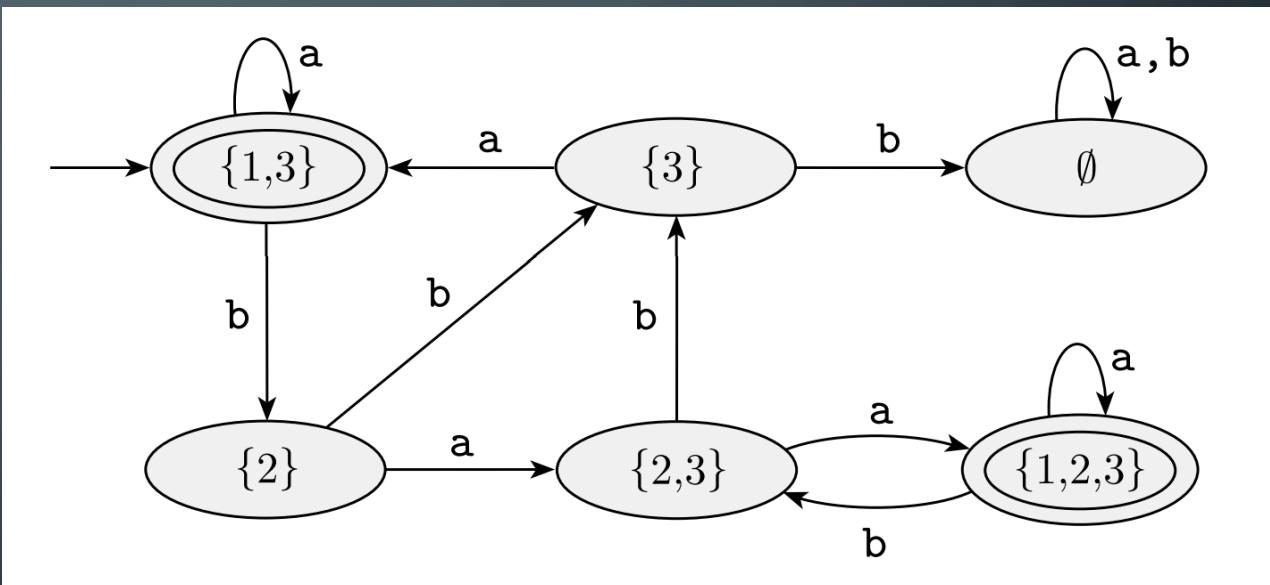
Possible Theorem 1: An NFA is equivalent in computational power to a DFA

Consider direction 2 next:

If an NFA exists that recognizes some language L , then a DFA exists that also accepts L !



Given an NFA that accepts an arbitrary language L (left), describe the process for turning this into an equivalent DFA (right)



Some state combinations from previous slide are impossible (have no incoming edges), so they can be removed. This is the simpler version

NFA VS. DFA?

Possible Theorem 1: An NFA is equivalent in computational power to a DFA

Consider direction 2 next:

Here is the formal version of the process

Consider an arbitrary NFA N that recognizes an arbitrary language L :

$$N = (Q, \Sigma, \delta, q_0, F)$$



Construct a DFA D as such:

$$D = (Q', \Sigma, \delta', q_0', F')$$

Such that:

$$Q' = \mathcal{P}(Q)$$

$$\delta' = (R \in Q', a \in \Sigma) \rightarrow \{q \in Q, r \in R \mid q \in E(\delta(r, a))\}$$

$$q_0' = E(\{q_0\})$$

$$F' = \{R \in Q' \mid \exists_{r \in R} r \in F\}$$

What is $E()$ here? Given a set of states R , let $E(R)$ be the set of states that we can reach by traveling along epsilon transitions only, including the original states in the return value.

NFA VS. DFA?

Theorem: An NFA is equivalent in computational power to a DFA

Thus, it is proven!!!!

NON-DETERMINISM SUMMARY

What did we learn in this section:

1. An NFA is a *different type of machine that extends the functionality of a DFA*
2. NFAs are *often more convenient for recognizing languages because of the parallelism*
 3. NFAs and DFAs are *equivalent in computational power*
4. *Do we know how to build an NFA in the real world? Kind of...*

REGULAR LANGUAGES

MOTIVATING QUESTIONS

Ok, we have NFAs and DFAs (and they are equivalent in computational power). What is the exact set of languages that these machines can recognize?

Can we find at least one language that NFAs and DFAs cannot recognize (spoiler: yes)? How do we find one?

DEFINITION: REGULAR LANGUAGE

A language is called a **regular language** if there exists some DFA that recognizes it

...and by equivalence,
there exists an NFA that
recognizes it too!

This definition is a bit tautological
right? Don't worry, it is a good
definition for now and we will analyze
what falls into this category and what
doesn't soon.

PROPERTIES OF REGULAR LANGUAGES

A language is called a **regular language** if there exists some DFA that recognizes it

We are interested in the following (potential) properties of Regular Languages:

Let A and B be languages, we define the regular operations as follows:

Union:

$$A \cup B = \{x | x \in A \vee x \in B\}$$

Concatenation:

$$A \circ B = \{xy | x \in A \wedge y \in B\}$$

Star:

$$A^* = \{x_1 x_2 \dots x_k | k \geq 0 \wedge \forall_i x_i \in A\}$$

CLOSURE REGULAR LANGUAGES

Suppose now that **A and B are regular languages**. What does it mean to say that regular languages are **closed under Union**?

For example, regular languages are closed under Union if the following is true:

If A and B are regular languages, then A Union B is also a regular language

Closure under some operation means that if the inputs are both in a set, then the output is also in that set. Are the regular languages closed under union, concatenation, and star?

Union:

$$A \cup B = \{x | x \in A \vee x \in B\}$$

Concatenation:

$$A \circ B = \{xy | x \in A \wedge y \in B\}$$

Star:

$$A^* = \{x_1 x_2 \dots x_k | k \geq 0 \wedge \forall_i x_i \in A\}$$

Suppose the regular languages ARE closed under these three operations. What does that tell us about regular languages and/or about how they can be constructed?

REG. LANGUAGES ARE CLOSED UNDER UNION

Theorem: *The regular languages are closed under union*

Union:

$$A \cup B = \{x | x \in A \vee x \in B\}$$

REG. LANGUAGES ARE CLOSED UNDER UNION

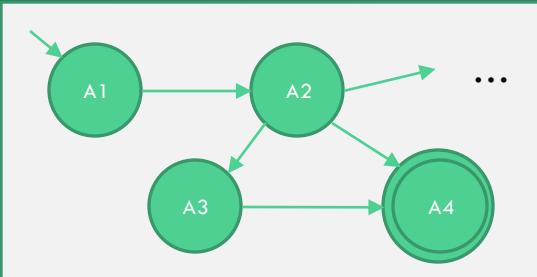
Theorem: *The regular languages are closed under union*

Union:

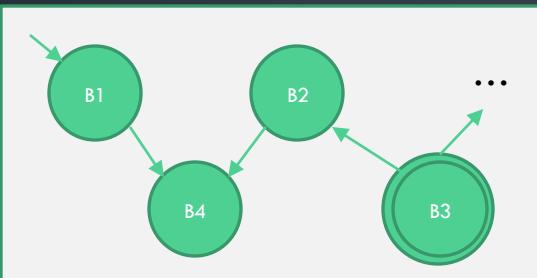
$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

Claim: If A and B are regular, then A union B is regular

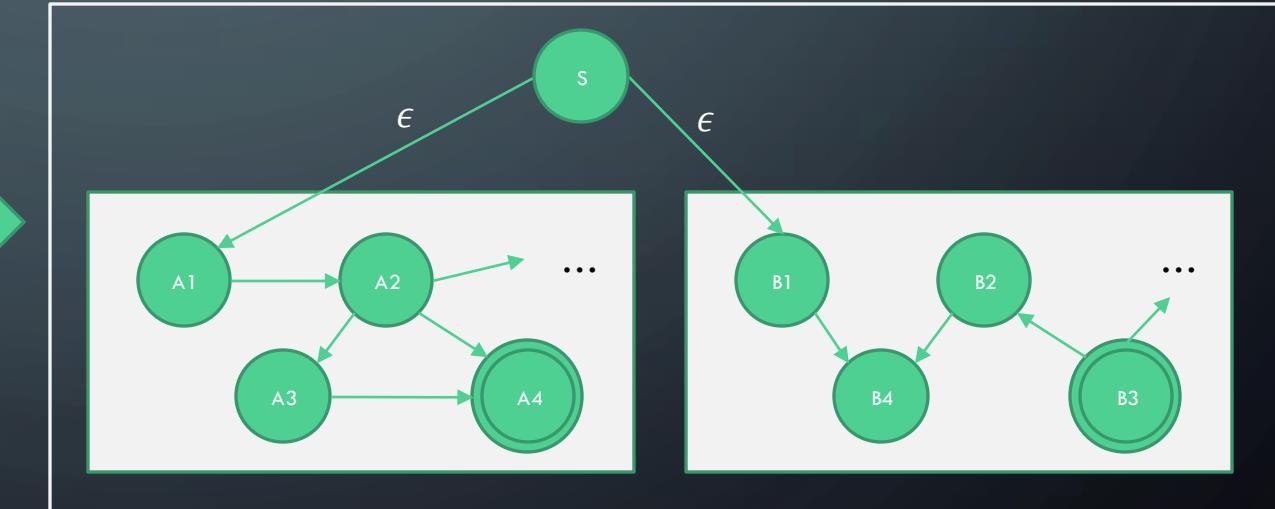
If A is regular,
then some DFA
 M_1 accepts it



If B is regular,
then some DFA
 M_2 accepts it



Proof: Given the DFAs for A and B (see left), construct a machine that accepts A union B (right)



See how easy the NFA makes things sometimes? Non-determinism allows us to simply spawn two threads, each of which runs one of the original DFAs.

REG. LANGUAGES ARE CLOSED UNDER CONCATENATION

Theorem: *The regular languages are closed under concatenation*

Concatenation:

$$A \circ B = \{xy \mid x \in A \wedge y \in B\}$$

REG. LANGUAGES ARE CLOSED UNDER CONCATENATION

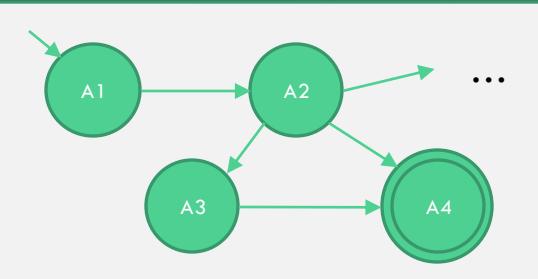
Theorem: The regular languages are closed under concatenation

Concatenation:

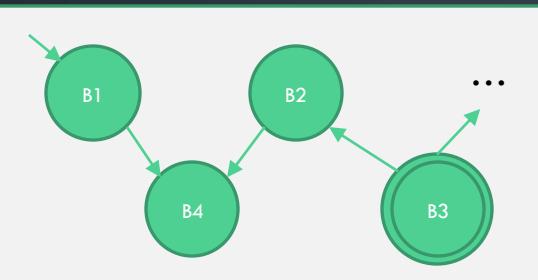
$$A \circ B = \{xy \mid x \in A \wedge y \in B\}$$

Claim: If A and B are regular, then A conc. B is regular

If A is regular,
then some DFA
 M_1 accepts it

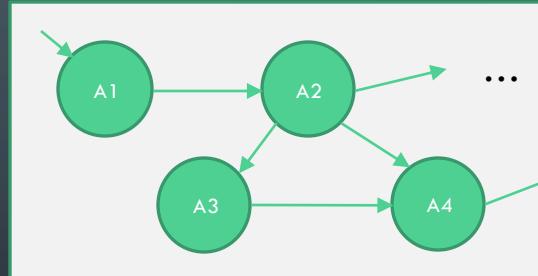


If B is regular,
then some DFA
 M_2 accepts it

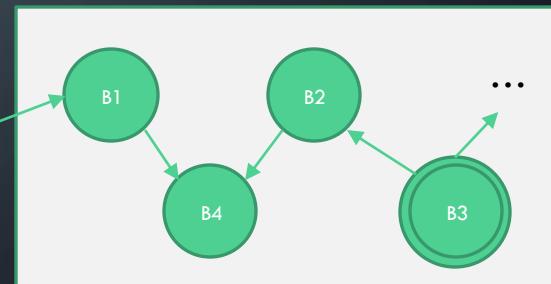


Proof: Given the DFAs for A and B (see left), construct a machine that accepts A conc. B (right)

New NFA N



ϵ



Key idea: Add an epsilon transition from every final state in M_1 to every start state in M_2

REG. LANGUAGES ARE CLOSED UNDER STAR

Theorem: *The regular languages are closed under star*

Star:

$$A^* = \{x_1 x_2 \dots x_k | k \geq 0 \wedge \forall_i x_i \in A\}$$

REG. LANGUAGES ARE CLOSED UNDER STAR

Theorem: The regular languages are closed under star

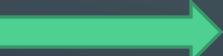
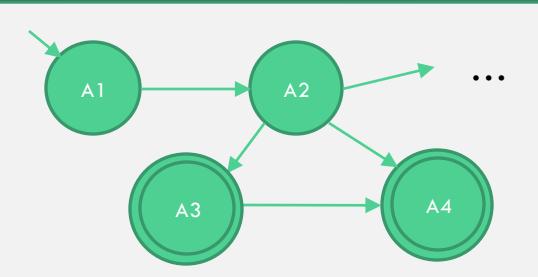
Star:

$$A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \wedge \forall_i x_i \in A\}$$

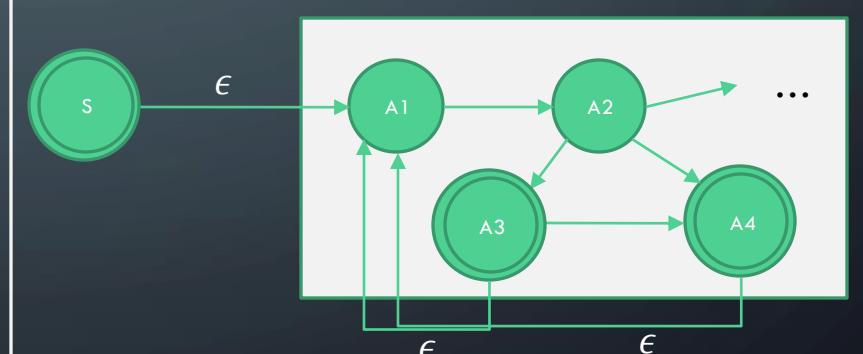
Claim: If A is regular, then A^* is regular

Proof: Given the DFA for A (see left), construct a machine that accepts A^* (right)

If A is regular,
then some DFA
 M_1 accepts it



New NFA N



Key idea: Add new start state (to handle empty string case), then run M_1 as before. Anytime we hit a final state of M_1 , epsilon transition to M_1 's old start state (because a new string in language A might be coming up next)

REGULAR LANGUAGES SUMMARY

What did we learn in this section:

1. A regular language is any language for which an NFA or DFA exists that recognizes it.
2. Regular languages are closed under Union, Concatenation, and Star (What does this mean for building up regular languages from smaller ones?)

PART 3: REGULAR EXPRESSIONS AND NON-REGULAR LANGUAGES

MOTIVATING QUESTIONS

Ok, NFA and DFAs both recognize regular languages. How can we succinctly express regular languages using more natural expressions?

Can we prove that this new notation for expressing regular languages is in fact complete and sound?

REGULAR EXPRESSIONS

In arithmetic, we have operations that allow us to build up larger items in a set from smaller ones. One example:

$$(5 + 3) \times 4$$

Because of the closure properties we just proved, we should be able to build up regular languages in a similar way. These expressions are called regular expressions.

$$(0 \cup 1)0^*$$

This is the language of binary strings that start with a 0 or 1 and then end in any number of 0s

MORE EXAMPLES OF REGULAR EXPRESSIONS

$$R = 0^*10^*$$

$L(R) = ???$

$$R = \Sigma^*1\Sigma^*$$

$L(R) = ???$

$$R = (\Sigma\Sigma)^*$$

$L(R) = ???$

$$R = 0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$$

$L(R) = ???$

*Note that here the alphabet is $\Sigma = \{0,1\}$

MORE EXAMPLES OF REGULAR EXPRESSIONS

$$R = 0^* 1 0^*$$

$L(R) = \{w \mid w \text{ contains a single } 1\}$

$$R = \Sigma^* 1 \Sigma^*$$

$L(R) = \{w \mid w \text{ contains at least one } 1\}$

$$R = (\Sigma\Sigma)^*$$

$L(R) = \{w \mid w \text{ is a string of even length}\}$

$$R = 0\Sigma^* 0 \cup 1\Sigma^* 1 \cup 0 \cup 1$$

$L(R) = \{w \mid w \text{ starts and ends with the same character}\}$

*Note that here the alphabet is $\Sigma = \{0,1\}$

FORMAL DEFINITION OF REGULAR EXPRESSIONS

We say that an expression R is a regular expression if R is...

1. a for some $a \in \Sigma$
2. ϵ
3. ϕ
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*) , where R_1 is a regular expression

1-3 are base cases. Regular expression can be one character, epsilon (the set containing only the empty string), or the empty set.

4-6 use the closed operations to build up larger expressions from smaller ones

FORMAL DEFINITION OF REGULAR EXPRESSIONS

We say that an expression R is a regular expression if R is...

1. a for some $a \in \Sigma$
2. ϵ
3. ϕ
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*) , where R_1 is a regular expression

Important Question:

Using this definition, can we build EVERY possible regular expression? How can we formally express this question and try to prove it?

FORMAL DEFINITION OF REGULAR EXPRESSIONS

We say that an expression R is a regular expression if R is...

1. a for some $a \in \Sigma$
2. ϵ
3. ϕ
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*) , where R_1 is a regular expression

Important Question:

Using this definition, can we build **EVERY** possible regular expression? How can we formally express this question and try to prove it?

Here is a formal description of this question

Possible Theorem: A language is regular (i.e., an NFA or DFA exists that accepts it) if and only if some regular expression describes it.

...and by extension, this means that finite automata and regular expressions are equivalently expressive (they can describe the exact same set of functions).

REG. EXPRESSIONS TO FINITE AUTOMATA

Possible Theorem: A language is regular (i.e., an NFA or DFA exists that accepts it) if and only if some regular expression describes it.

...and by extension, this means that finite automata and regular expressions are equivalently expressive (they can describe the exact same set of functions).

Proof Overview:

Direction 1: If given a regular expression, then the language is regular

Strategy: Given a regular expression, show how to construct the NFA that is equivalent to it

Direction 2: If given a regular language (or DFA/NFA), then some regular expression describes it.

Strategy: Given an arbitrary DFA/NFA, describe the process for generating an equivalent reg. expression

REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 1: If given a regular expression, then the language is regular

Strategy: Given a regular expression, show how to construct the NFA that is equivalent to it

We say that an expression R is a regular expression if R is...

1. a for some $a \in \Sigma$
2. ϵ
3. ϕ
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*) , where R_1 is a regular expression



Proof by induction:

Cases 1-3 are the base cases

Inductive Hypothesis: Assume R_1 and R_2 are regular expressions and have DFAs

Cases 4-6 are the inductive steps

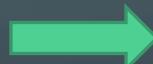
REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 1: If given a regular expression, then the language is regular

Strategy: Given a regular expression, show how to construct the NFA that is equivalent to it

We say that an expression R is a regular expression if R is...

1. a for some $a \in \Sigma$
2. ϵ
3. ϕ
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*) , where R_1 is a regular expression



Proof by induction:

Base Case 1: NFA is below:



REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 1: If given a regular expression, then the language is regular

Strategy: Given a regular expression, show how to construct the NFA that is equivalent to it

We say that an expression R is a regular expression if R is...

1. a for some $a \in \Sigma$
2. ϵ
3. ϕ
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*) , where R_1 is a regular expression



Proof by induction:

Base Case 2: NFA is below:



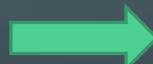
REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 1: If given a regular expression, then the language is regular

Strategy: Given a regular expression, show how to construct the NFA that is equivalent to it

We say that an expression R is a regular expression if R is...

1. a for some $a \in \Sigma$
2. ϵ
3. ϕ
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*) , where R_1 is a regular expression



Proof by induction:

Base Case 3: NFA is below:



REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 1: If given a regular expression, then the language is regular

Strategy: Given a regular expression, show how to construct the NFA that is equivalent to it

We say that an expression R is a regular expression if R is...

1. a for some $a \in \Sigma$
2. ϵ
3. ϕ
4. **$(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions**
5. **$(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions**
6. **(R_1^*) , where R_1 is a regular expression**



Inductive Hypothesis:

Assume that R_1 and R_2 are regular languages and have at least one DFA/NFA that recognizes them each.

Inductive Step:

Use R_1 and R_2 to construct cases 4-6 in exactly the same way we did to prove closure for regular languages. Will not repeat here.

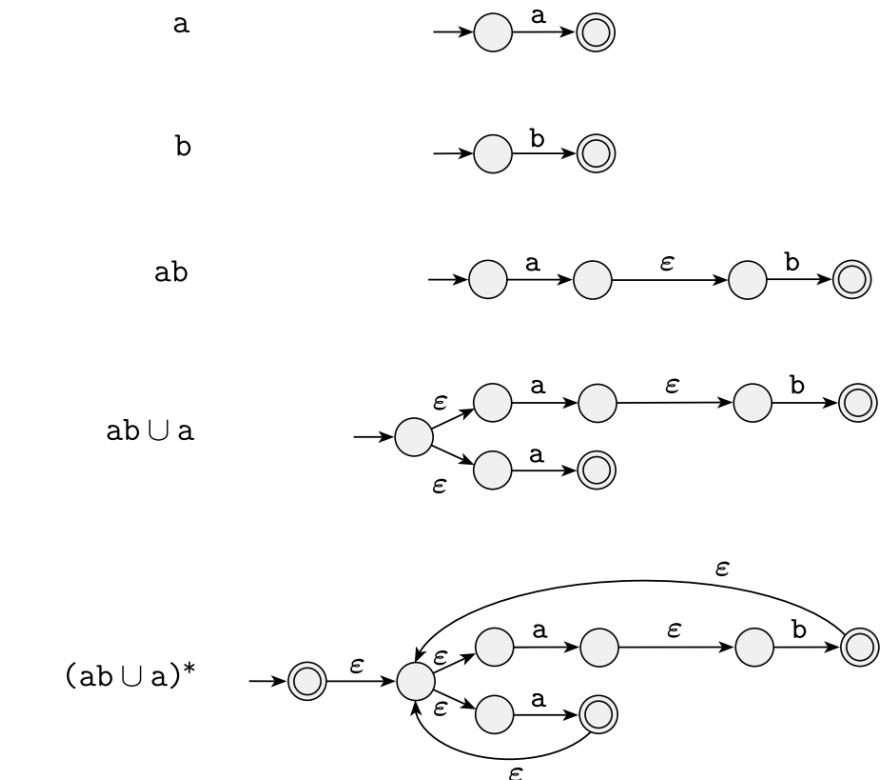
Thus, it is proven in one direction. All regular expressions are also a regular language (and thus have at least one DFA that recognizes them)

REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 1: If given a regular expression, then the language is regular

Strategy: Given a regular expression, show how to construct the NFA that is equivalent to it

Example conversion of
 $(ab \cup a)^*$ to help with intuition.
Remember that an example is
not a proof, even if it helps with
your understanding.



REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 2: If given a regular language (or DFA/NFA), then some regular expression describes it.

Strategy: Given an arbitrary DFA/NFA, describe the process for generating an equivalent reg. expression

This direction is much harder, so we will provide an overview

Given an NFA, need to convert it into an equivalent regular expression

Convert to a GNFA (an NFA with regular expressions as transitions)

Make GNFA smaller one state at a time until there is only a start and final state (one transition)

The one transition in the GNFA is the regular expression we need

REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 2: If given a regular language (or DFA/NFA), then some regular expression describes it.

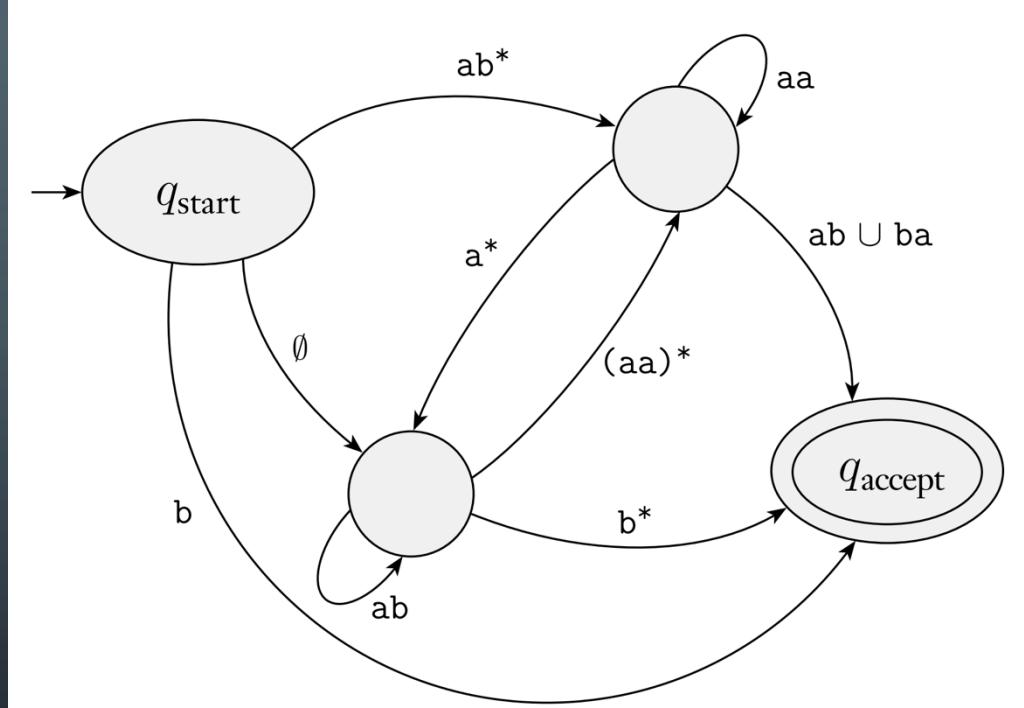
Strategy: Given an arbitrary DFA/NFA, describe the process for generating an equivalent reg. expression

Given an NFA, need to convert it into an equivalent regular expression

Convert to a GNFA (an NFA with regular expressions as transitions)

A Generalized NFA (GNFA) is an NFA that allows for patterns of strings as inputs instead of just once character as inputs (can read multiple symbols at once and then transition. Also:

1. Is a complete graph (there is a transition from every node to every other except:
2. Start node (only one allowed) has no incoming edges
3. Final node (only one allowed) has no outgoing edges



REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 2: If given a regular language (or DFA/NFA), then some regular expression describes it.

Strategy: Given an arbitrary DFA/NFA, describe the process for generating an equivalent reg. expression

Given an NFA, need to convert it into an equivalent regular expression



Convert to a GNFA (an NFA with regular expressions as transitions)

To convert a generic NFA into a GNFA, simply:

1. Add dummy start node with epsilon transition to old start node
2. Add dummy final node with epsilon transitions from all final nodes in NFA
3. For any pair of nodes with no connection, add one with null set as transition requirement (will never be used)
4. If any pair of nodes have multiple transitions, combine them into one where the transition is now the Union of the two (or more) old labels.

REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 2: If given a regular language (or DFA/NFA), then some regular expression describes it.

Strategy: Given an arbitrary DFA/NFA, describe the process for generating an equivalent reg. expression

Convert to a GNFA (an NFA with regular expressions as transitions)

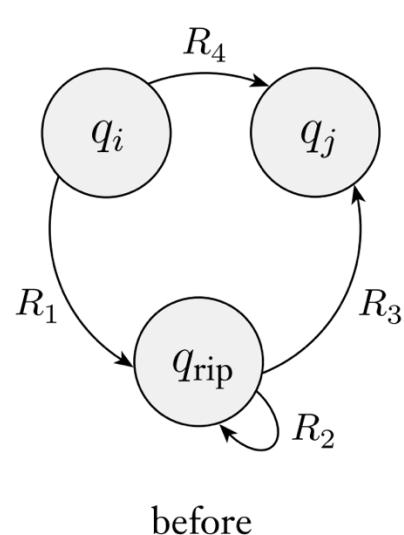


Make GNFA smaller one state at a time until there is only a start and final state (one transition)

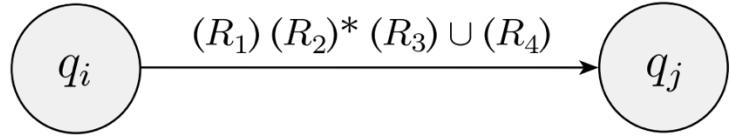
Process: Given a state you want remove (q_{rip}) every pair of any two other states that you will not remove (q_i, q_j), change the transition from q_i to q_j to:

$$(R_1)(R_2)^*(R_3) \cup (R_4)$$

**See diagram for definitions of R_1 through R_4



before



after

**Repeat this process until there are only two states left (start and final state).

REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 2: If given a regular language (or DFA/NFA), then some regular expression describes it.

Strategy: Given an arbitrary DFA/NFA, describe the process for generating an equivalent reg. expression

Make GNFA smaller one state at a time until there is only a start and final state (one transition)



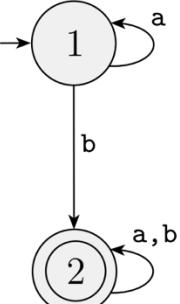
The one transition in the GNFA is the regular expression we need

Once there is only one state left, the one transition label IS the regular expression that is equivalent to the original NFA.

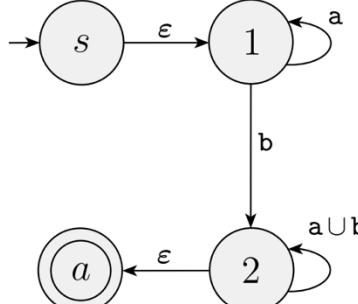
REG. EXPRESSIONS TO FINITE AUTOMATA

Direction 2: If given a regular language (or DFA/NFA), then some regular expression describes it.

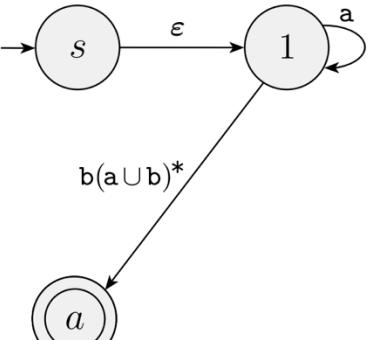
Strategy: Given an arbitrary DFA/NFA, describe the process for generating an equivalent reg. expression



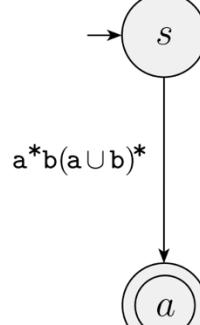
(a)



(b)



(c)



(d)

SUMMARY SO FAR!!!

These are all equivalent in their expressive power...and we proved it!

Finite Automata: First computation model.

Limited memory (just one state and input)

DFA_s are equivalent to NFA_s

Regular Languages:

*Class of languages DFA/NFA can
recognize*

Regular Expressions:

*A different but equivalent way to express
the regular languages*

Just one last thing to do...

*Can we find languages that are not
regular??*

FINDING NON-REGULAR LANGUAGES

MOTIVATING QUESTIONS

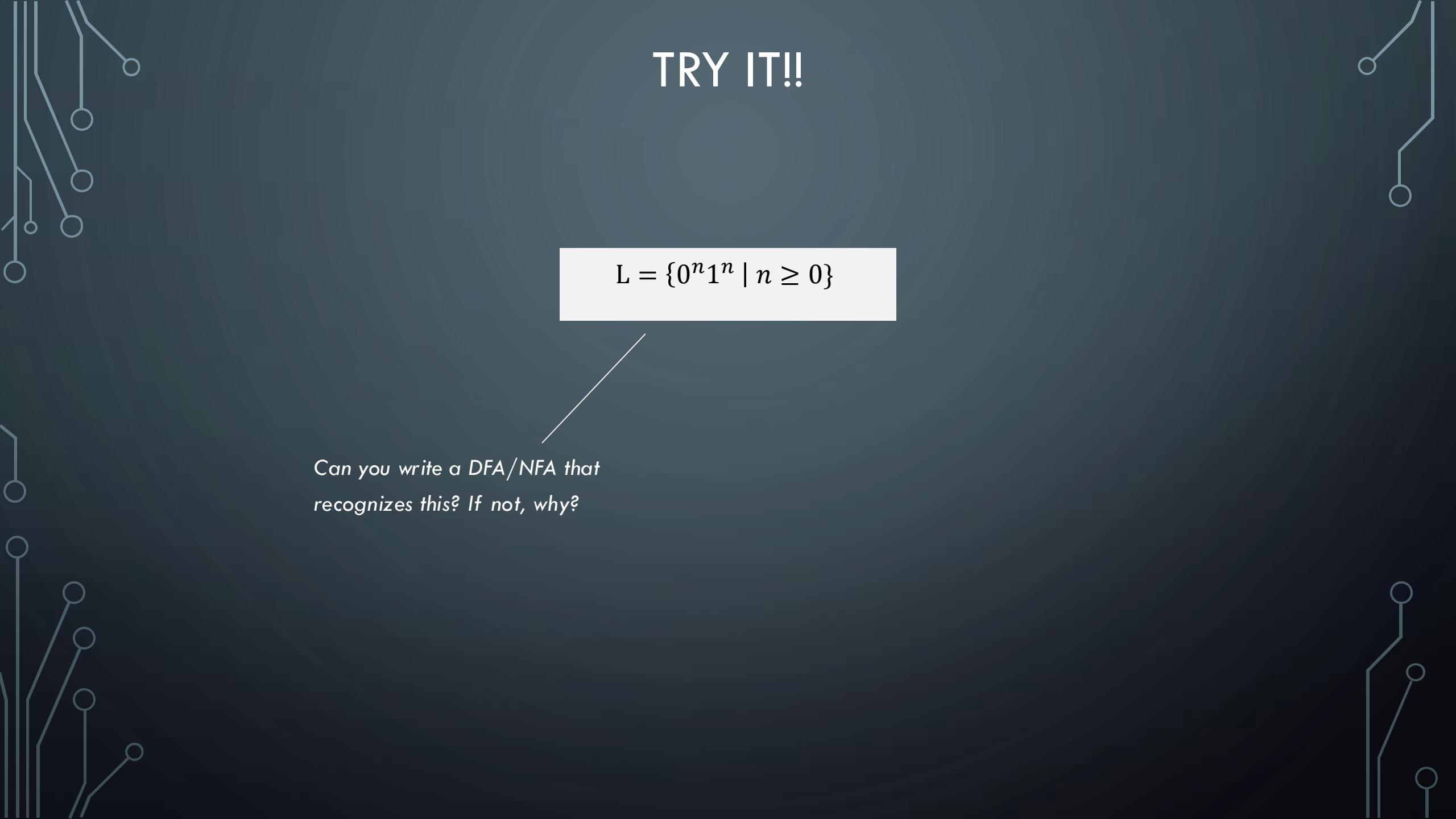
Can we find at least one non-regular language? What seems to be the limiting factor that prevents a DFA/NFA from recognizing it?

Do we have a good mechanism for proving a language is not regular?

TRY IT!!

Can you come up with a simple language that is not regular?

HINT: Think about what a DFA/NFA does not have / cannot do and try to exploit it. What does a DFA have very limited amount of...



TRY IT!!

$$L = \{0^n 1^n \mid n \geq 0\}$$

*Can you write a DFA/NFA that
recognizes this? If not, why?*

WHAT ABOUT THESE TWO?

$C = \{w \mid w \text{ has an equal number of } 0\text{s and } 1\text{s}\}$

$D = \{w \mid w \text{ has an equal number of } 01 \text{ and } 10 \text{ substrings}\}$

WHAT ABOUT THESE TWO?

$C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$

$D = \{w \mid w \text{ has an equal number of 01 and 10 substrings}\}$

This one is NOT regular

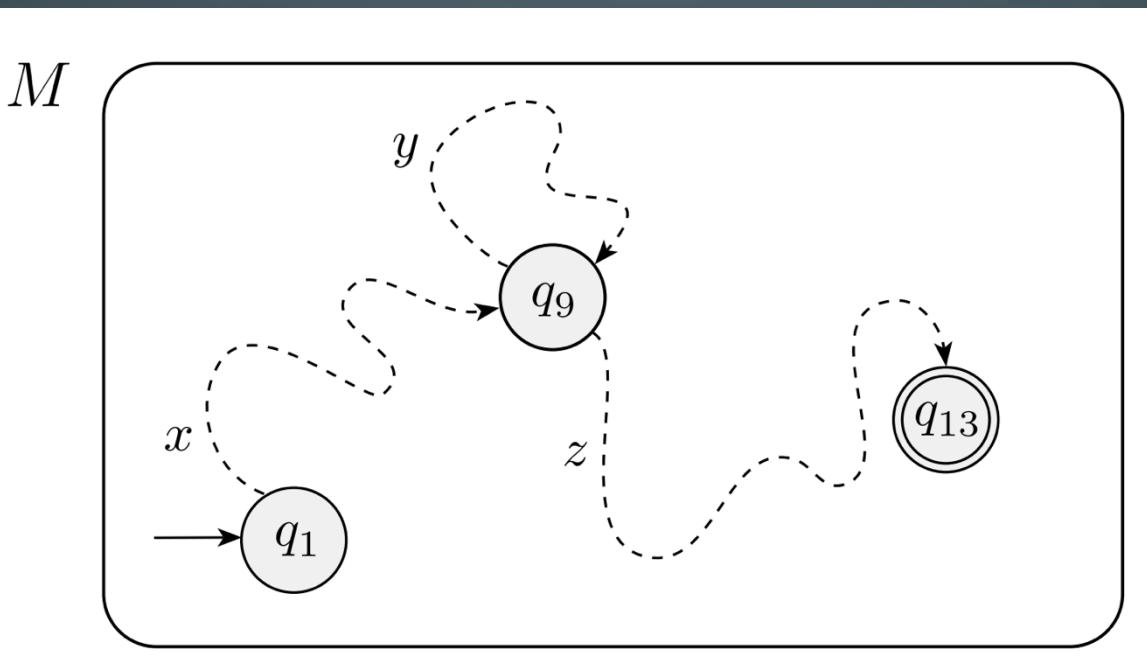
The point here is that sometimes we cannot rely on intuition alone. We need a mathematical proof to be SURE our intuition is correct.

Surprisingly, this one IS regular

THE PUMPING LEMMA

The pumping lemma for regular languages is a technique for proving that a language is NOT regular. It relies on the idea that certain substrings of input must be repeated in order for a DFA to continue to run.

Consider the DFA M (to the right). The string x gets us from state q_1 to q_9 (possibly with states in between).



The string y gets us from q_9 , through other states possibly, and eventually back to q_9 .

The string z gets us from state q_9 eventually to final state q_{13}

So...in this case, notice that the string y can be pumped (repeated) over and over and still be accepted by M .

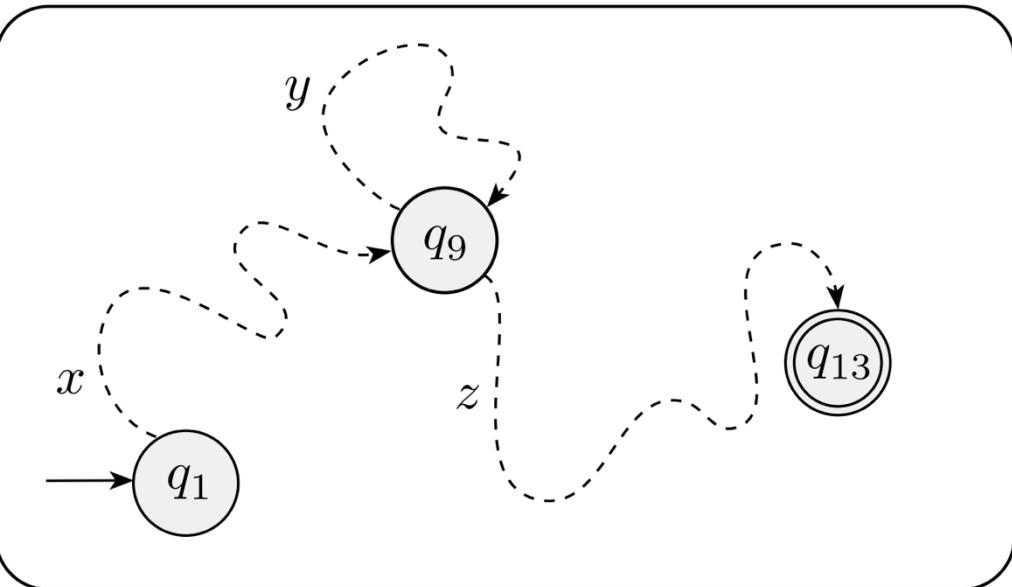
Any string of the form:
 xy^*z

Will be accepted

THE PUMPING LEMMA

The pumping lemma for regular languages is a technique for proving that a language is NOT regular. It relies on the idea that certain substrings of input must be repeated in order for a DFA to continue to run.

M



So...in this case, notice that the string y can be pumped (repeated) over and over and still be accepted by M .

Any string of the form:

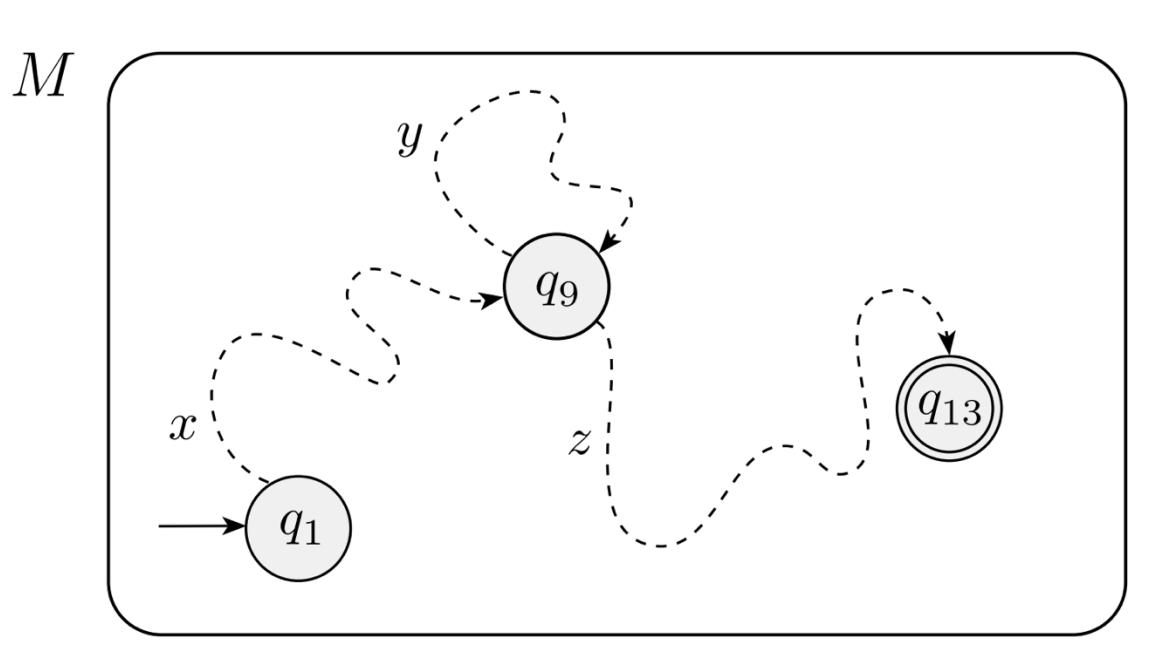
$$xy^*z$$

Will be accepted

Important Observation: The string y (that can be pumped) is not guaranteed to exist for all accepted strings. They must be long enough that it is guaranteed that at least one state in the DFA was used multiple times (e.g., if the string is longer than the number of states in the DFA, then some string y is guaranteed to exist).

THE PUMPING LEMMA

The **pumping lemma for regular languages** is a technique for proving that a language is NOT regular. It relies on the idea that certain substrings of input must be repeated in order for a DFA to continue to run.



Pumping Lemma:

If A is a regular language, then there is a number p (called the *pumping length*) where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$ satisfying the following conditions:

1. for each $i \geq 0, xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

FYI, p is going to be the number of states in the theoretical DFA that accepts the language A . Any string of length p or greater is guaranteed to have a repeated state within the first p characters of input

THE PUMPING LEMMA

Example 1: Let's show the following language is NOT regular:

$$B = \{0^n 1^n \mid n \geq 0\}$$

Pumping Lemma:

If A is a regular language, then there is a number p (called the pumping length) where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$ satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Proof overview (Proof by Contradiction):

1. Assume that language B is regular!
2. Select a string s' that is in language B and is long enough. Because of the pumping lemma we should be able to split it, find a string y , and pump that string y .
3. Show that there is no way to pump s' (every way we try to divide it up is not pumpable)
4. Contradiction: Language is not regular because we found a string s' that cannot be pumped

THE PUMPING LEMMA

Example 1: Let's show the following language is NOT regular:

$$B = \{0^n 1^n \mid n \geq 0\}$$

Pumping Lemma:

If A is a regular language, then there is a number p (called the pumping length) where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$ satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Proof overview (Proof by Contradiction):

1. Assume that language B is regular!
2. Select a string s' that is in language B . Because of the pumping lemma we should be able to split it, find a string y , and pump that string y .

Step 1: Just assume B is regular (DONE!)

Step 2: Select $s' = 0^p 1^p$

**Note that p must exist by pumping lemma!

THE PUMPING LEMMA

Example 1: Let's show the following language is NOT regular:

$$B = \{0^n 1^n \mid n \geq 0\}$$

Pumping Lemma:

If A is a regular language, then there is a number p (called the pumping length) where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$ satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Proof overview (Proof by Contradiction):

3. Show that there is no way to pump s' (every way we try to divide it up is not pumpable)

We selected $s' = 0^p 1^p$

So we want to divide it such that $s' = xyz$

Three cases:

1. y contains only 0s
2. y contains some 0s and some 1s (e.g., 0000111111)
3. y contains all 1s

None of these three cases lead to valid pumping. Cases 1 and 3 will lead to more 0s or 1s in the string respectively. Case 2 (if pumped) will lead to the format of the string being invalid (e.g., 00110011). Proof complete!!

THE PUMPING LEMMA

Example 2: Let's show the following language is NOT regular:

$$C = \{w \mid w \text{ has an equal number of } 0\text{s and } 1\text{s}\}$$

Pumping Lemma:

If A is a regular language, then there is a number p (called the pumping length) where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$ satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Let's do this one on the board together!

THE PUMPING LEMMA

Example 3: Let's show the following language is NOT regular:

$$C = \{0^i 1^j \mid i > j\}$$

Pumping Lemma:

If A is a regular language, then there is a number p (called the pumping length) where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$ satisfying the following conditions:

1. for each $i \geq 0, xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Let's do this one on the board together!

CONCLUSIONS

WHAT YOU LEARNED IN THIS DECK!

These are all equivalent in their expressive power...and we proved it!

Finite Automata: First computation model.

Limited memory (just one state and input)

DFA_s are equivalent to NFA_s

Regular Languages:

*Class of languages DFA/NFA can
recognize*

Regular Expressions:

*A different but equivalent way to express
the regular languages*

Using the pumping lemma, we can find languages that are non-regular.

*We will need an updated computational model to handle these. Next
time we will see a model that adds memory to our machines to increase
the functions (languages) that can be computed.*