

# Finite State Machines

CSE 311 Autumn 2024  
Lecture 23

# Last 2.5 Weeks

{ What computers can and can't do...  
Given any finite amount of time.

We'll start with a simple model of a computer – finite state machines.

What do we want computers to do? Let's start very simple.

We'll give them an input (in a string format), and we want them to say "yes" or "no" for that string on a certain question.

Example questions one might want to answer.

Does this input java code compile to a valid program?

Does this input string match a particular regular expression?

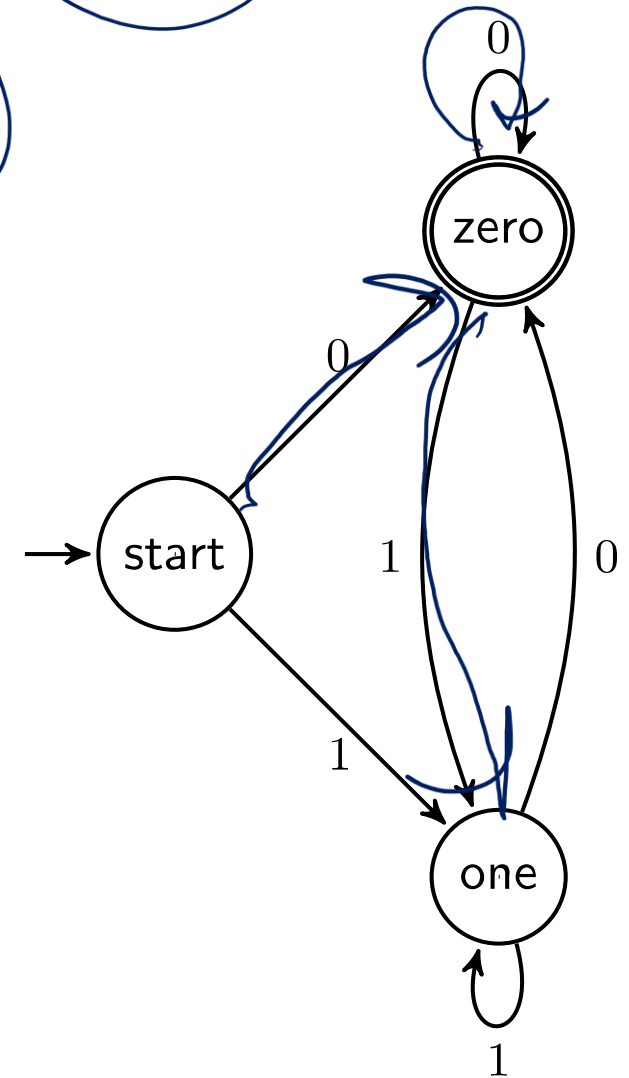
Is this input list sorted?

Depending on the "computer" some questions might be out of reach.

# Deterministic Finite Automaton

Our machine is going to get a string as input. It will read one character at a time and update "its state." At every step, the machine thinks of itself as in one of the (finite number) vertices. When it reads the character it follows the arrow labeled with that character to its next state.

Start at the "start state" (unlabeled, incoming arrow). After you've read the last character, accept the string if and only if you're in a "final state" (double circle).

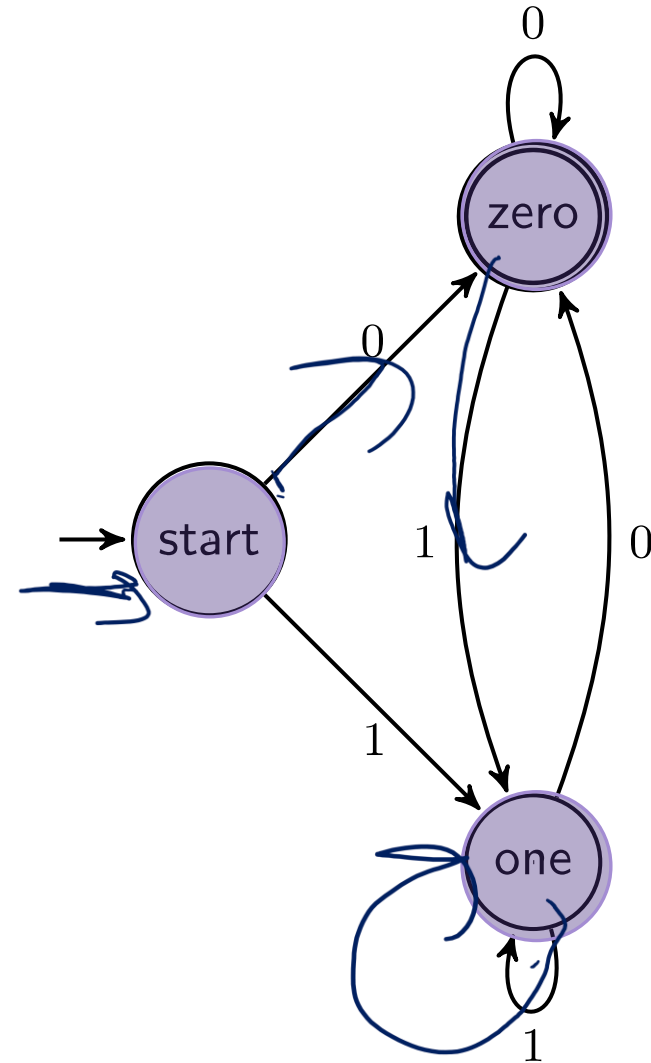


# Let's see an example

Input string:

011  
↑  
1010

*reject*

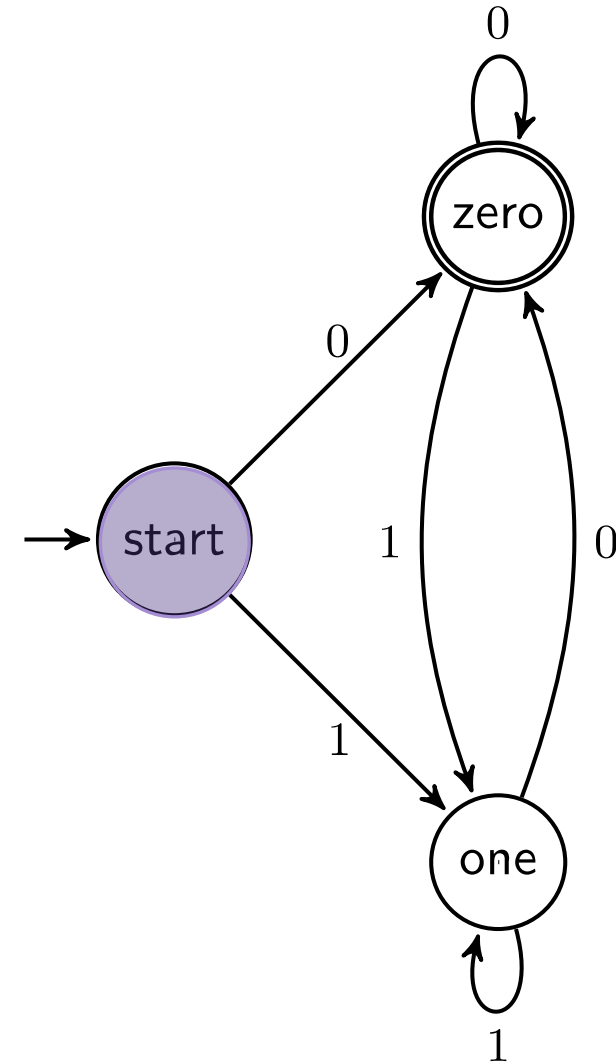


# Let's see an example

Input string:

011

1010

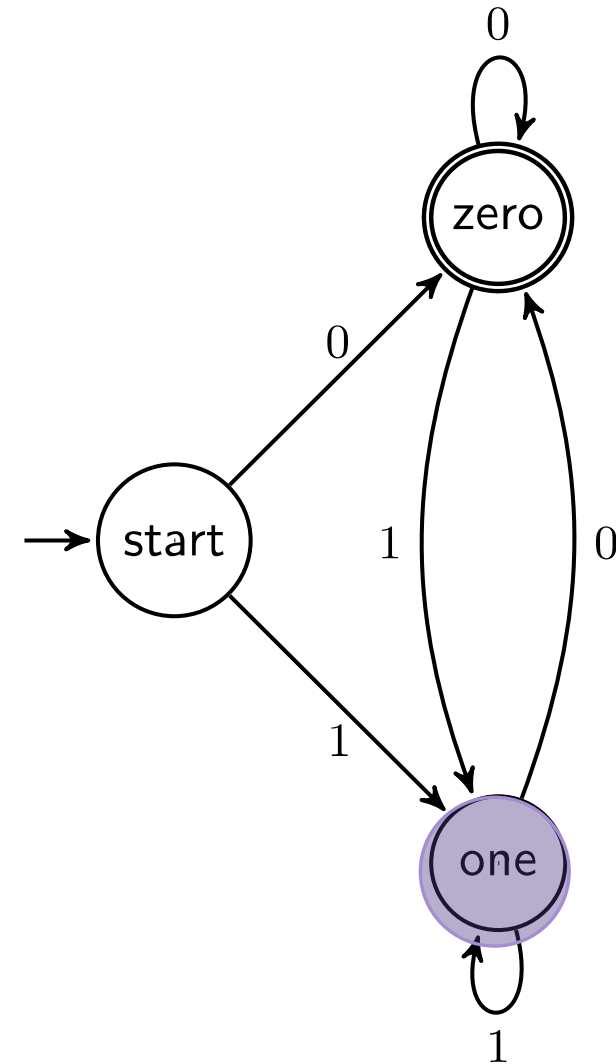


# Let's see an example

Input string:

011

1010

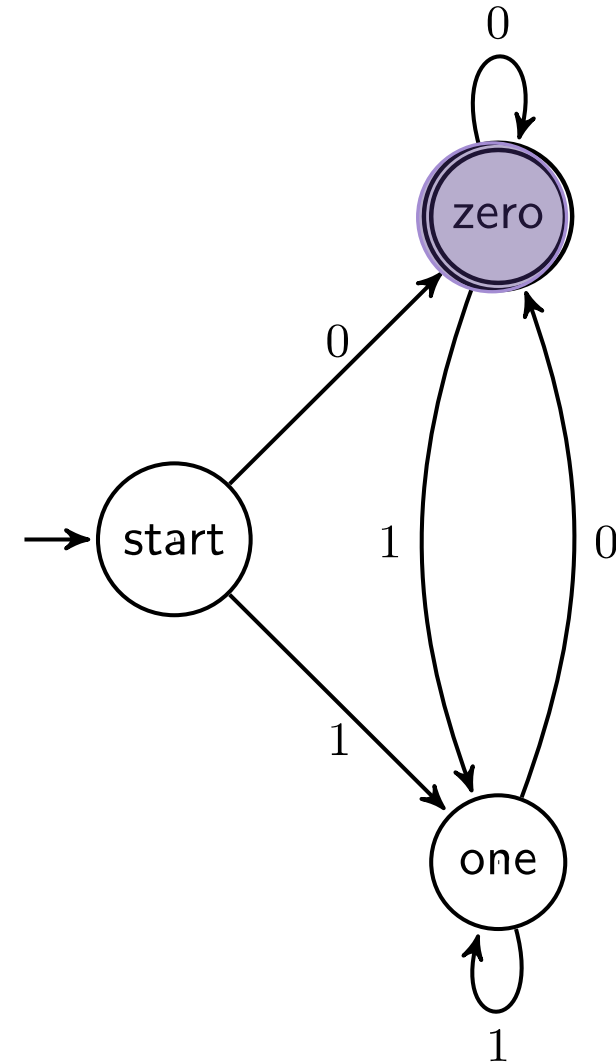


# Let's see an example

Input string:

011

1010

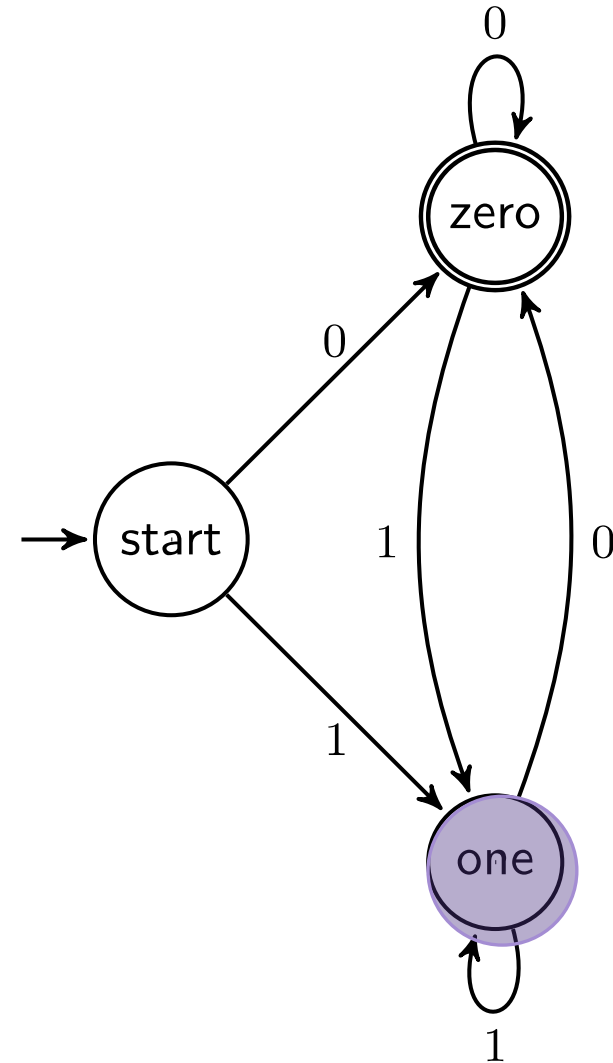


# Let's see an example

Input string:

011

1010



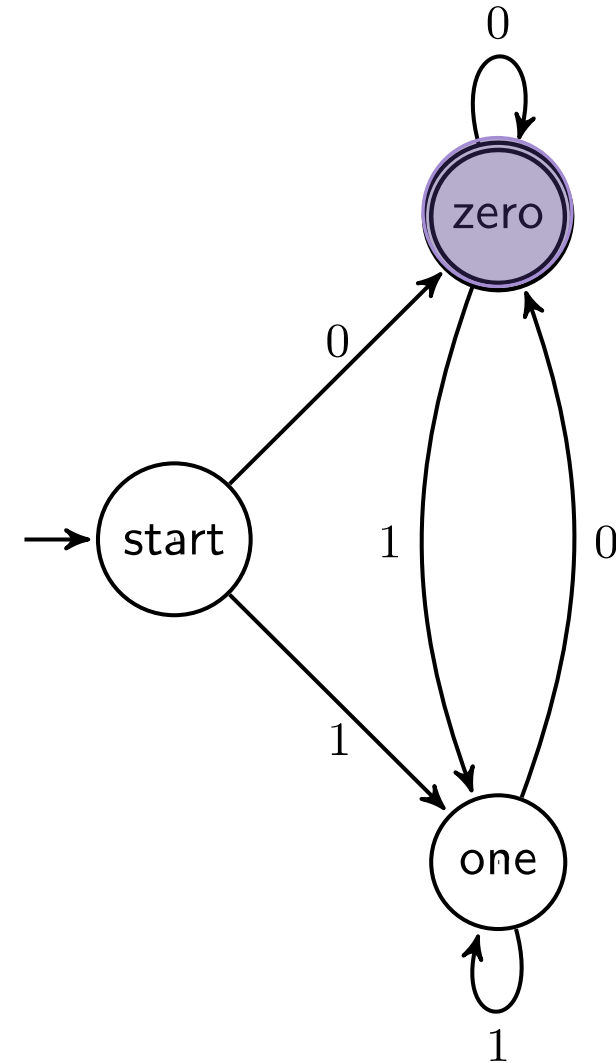
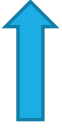


# Let's see an example

Input string:

011 *rejected*

1010 *accepted*



# Deterministic Finite Automata

Some more requirements:

Every machine is defined with respect to an alphabet  $\Sigma$

Every state has exactly one outgoing edge for every character in  $\Sigma$ .

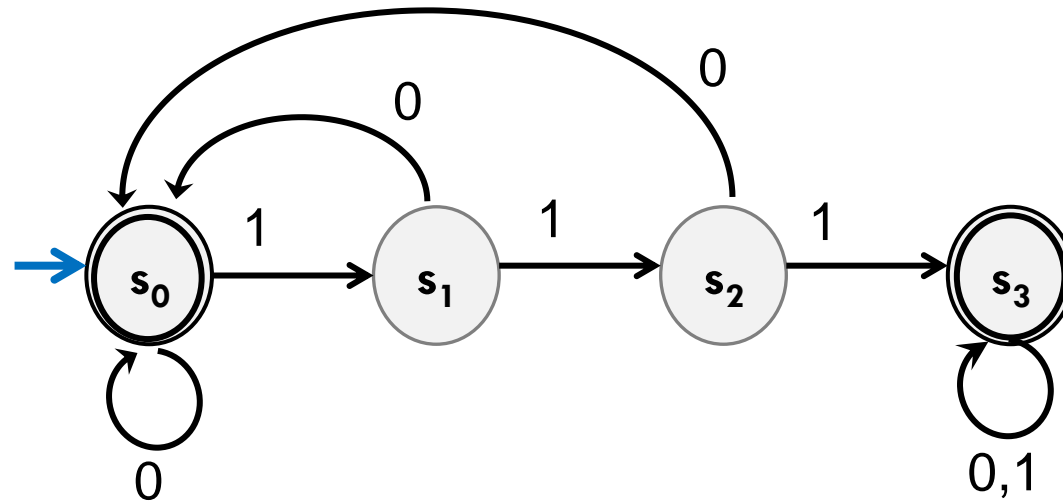
There is exactly one start state; can have as many accept states (aka final states) as you want – including none.

# Deterministic Finite Automata

Can also represent transitions with a table.

111      6✓      ~~001~~ 0      0✓      E

Old State	0	1
$s_0$	$s_0$	$s_1$
$s_1$	$s_0$	$s_2$
$s_2$	$s_0$	$s_3$
$s_3$	$s_3$	$s_3$

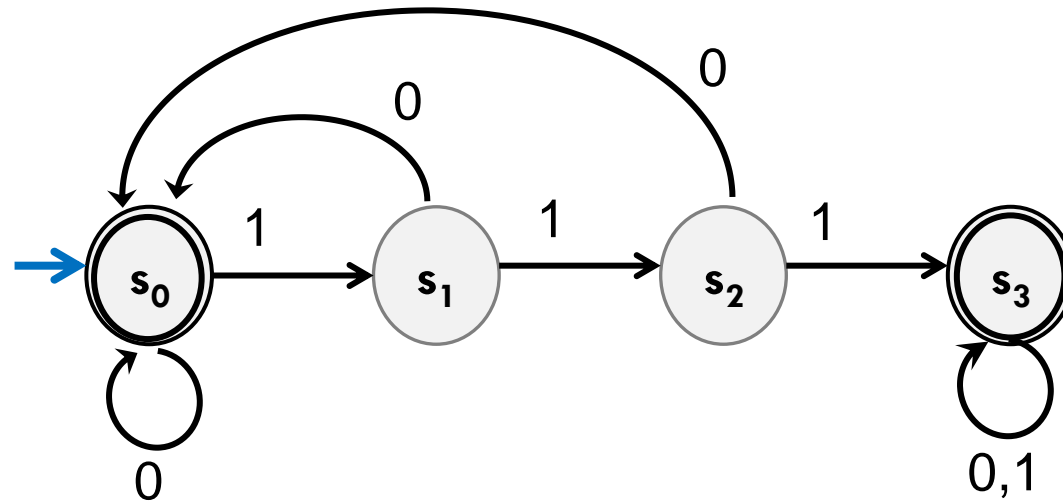


# Deterministic Finite Automata

What is the language of this DFA?

I.e. the set of all strings it accepts?

Old State	0	1
$s_0$	$s_0$	$s_1$
$s_1$	$s_0$	$s_2$
$s_2$	$s_0$	$s_3$
$s_3$	$s_3$	$s_3$



# Deterministic Finite Automata

0110111000...

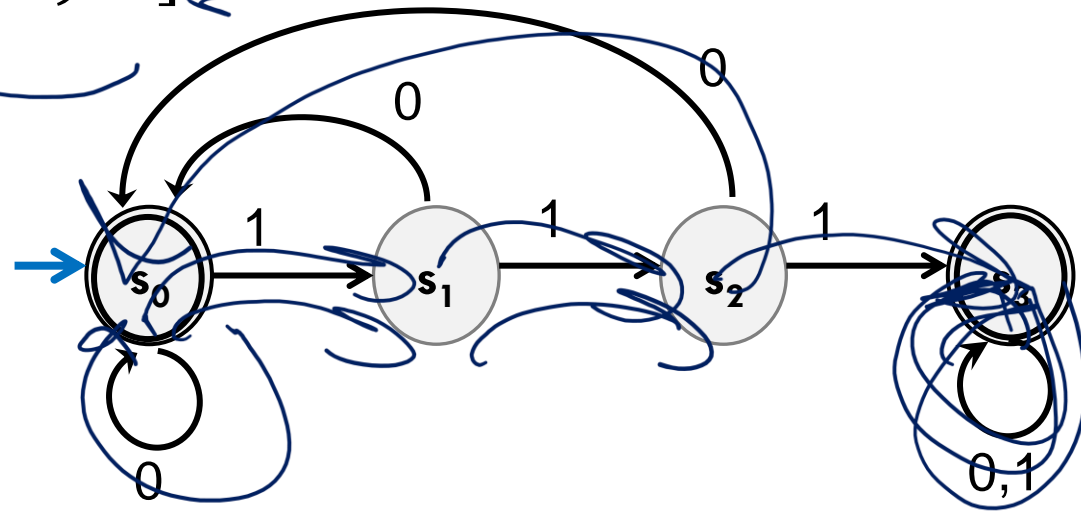
If the string has 111, then you'll end up in  $s_3$  and never leave.

If you end with a 0 you're back in  $s_0$  which also accepts.

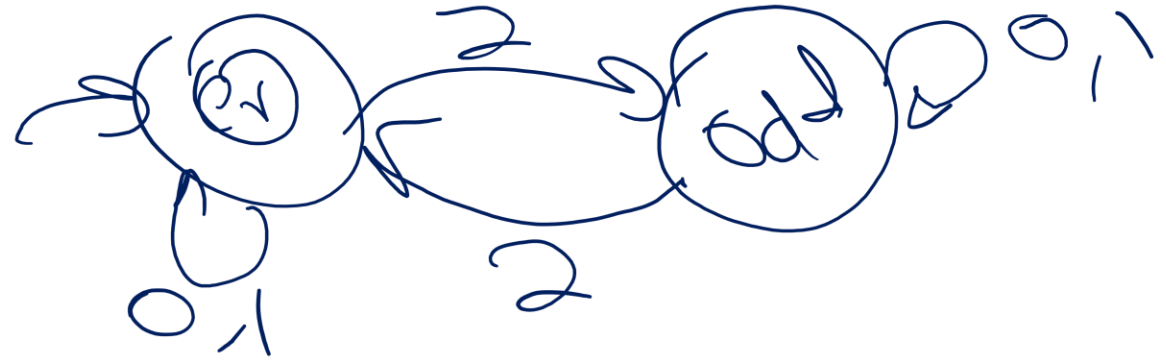
And... $\epsilon$  is also accepted

$$[(0 \cup 1)^* 111(0 \cup 1)^*] \cup [(0 \cup 1)^* 0]^*$$

Old State	0	1
$s_0$	$s_0$	$s_1$
$s_1$	$s_0$	$s_2$
$s_2$	$s_0$	$s_3$
$s_3$	$s_3$	$s_3$



# Design some DFAs



Let  $\Sigma = \{0,1,2\}$

$M_1$  should recognize "strings with an even number of 2's."

What do you need to remember?

$M_2$  should recognize "strings where the sum of the digits is congruent to 0 (*mod* 3)"



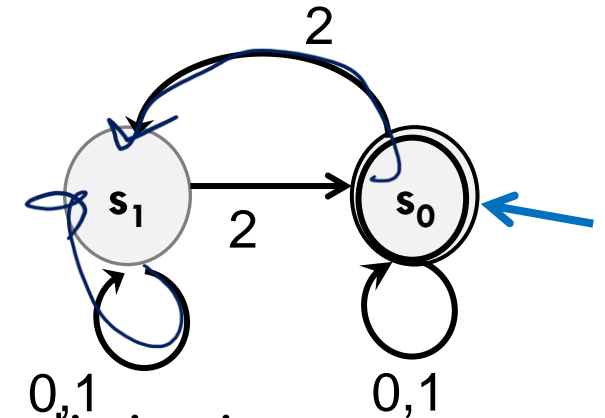
1021

$$1 + 0 + 2 + 1 = 4$$

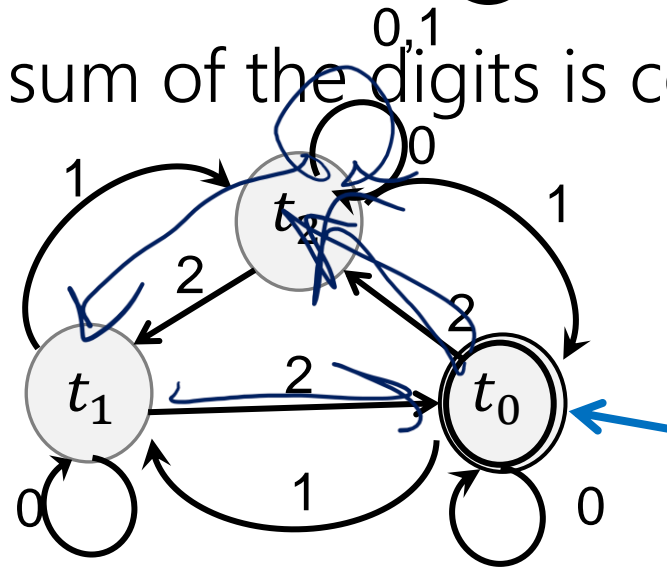
# Design some DFAs

Let  $\Sigma = \{0,1,2\}$

$M_1$  should recognize "strings with an even number of 2's."



$M_2$  should recognize "strings where the sum of the digits is congruent to 0 ( $\text{mod } 3$ )"



# Designing DFAs notes

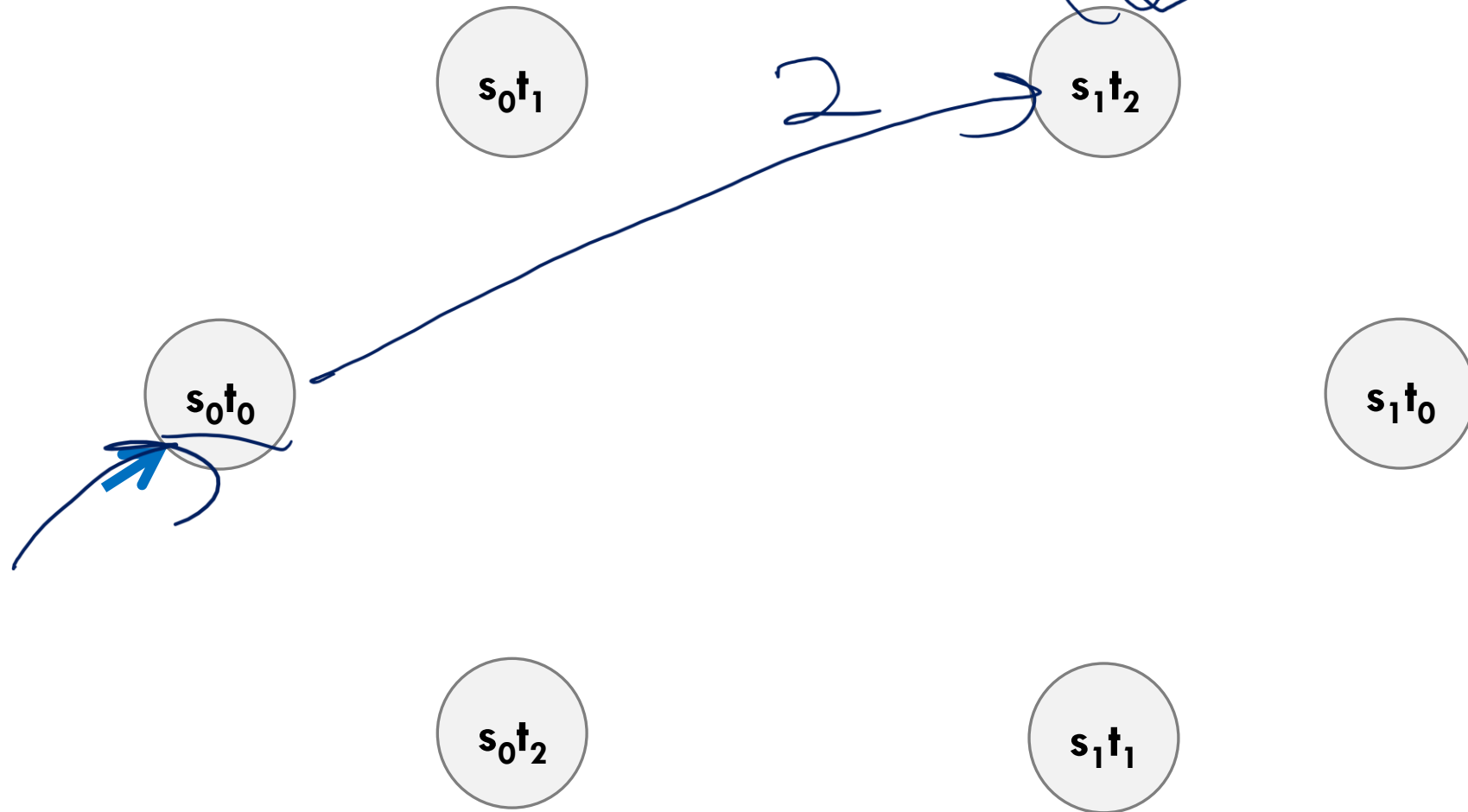
DFAs can't "count arbitrarily high"

For example, we could not make a DFA that remembers the overall sum of all the digits (not taken % 3)

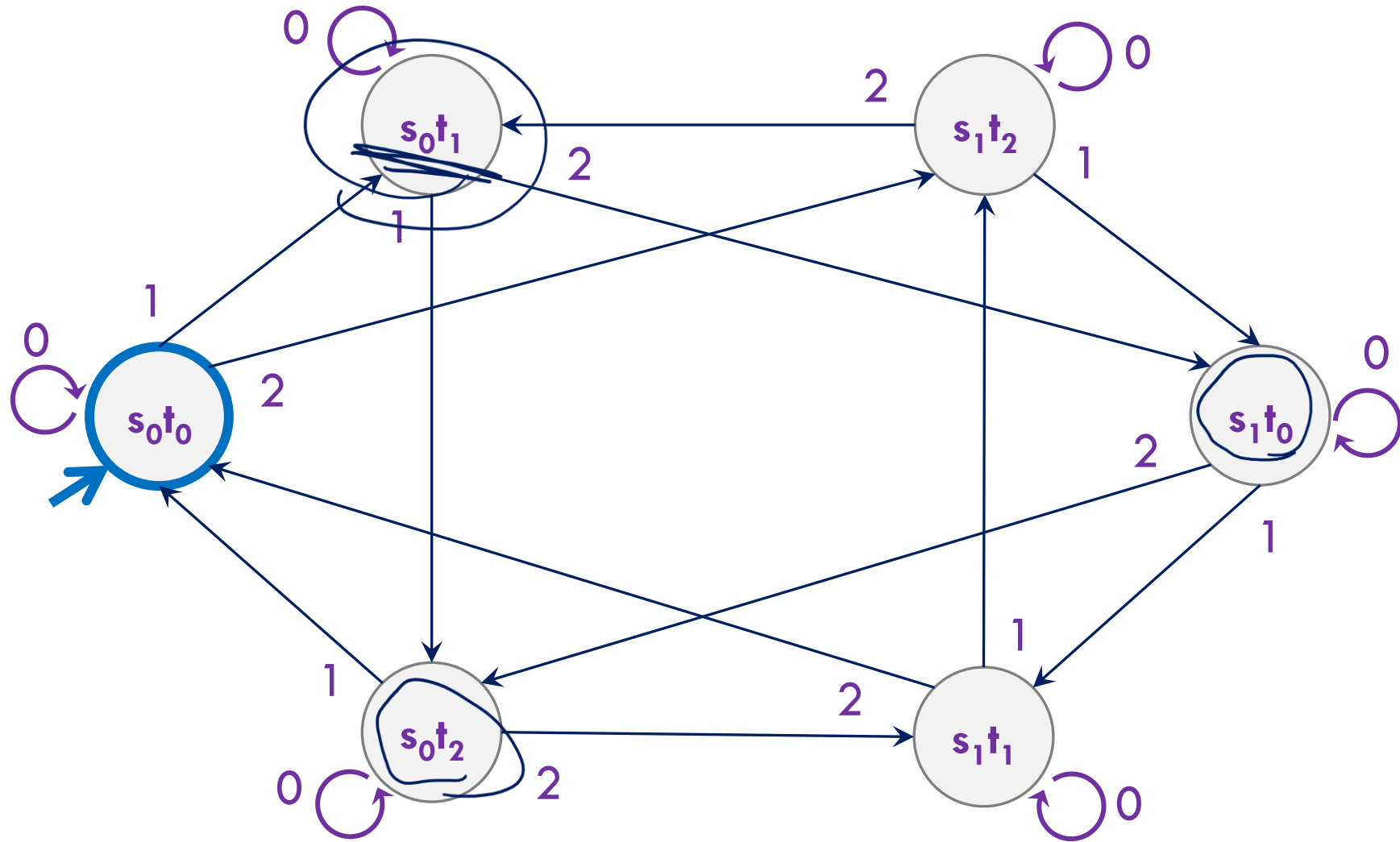
That would have infinitely many states! We're only allowed a finite number.



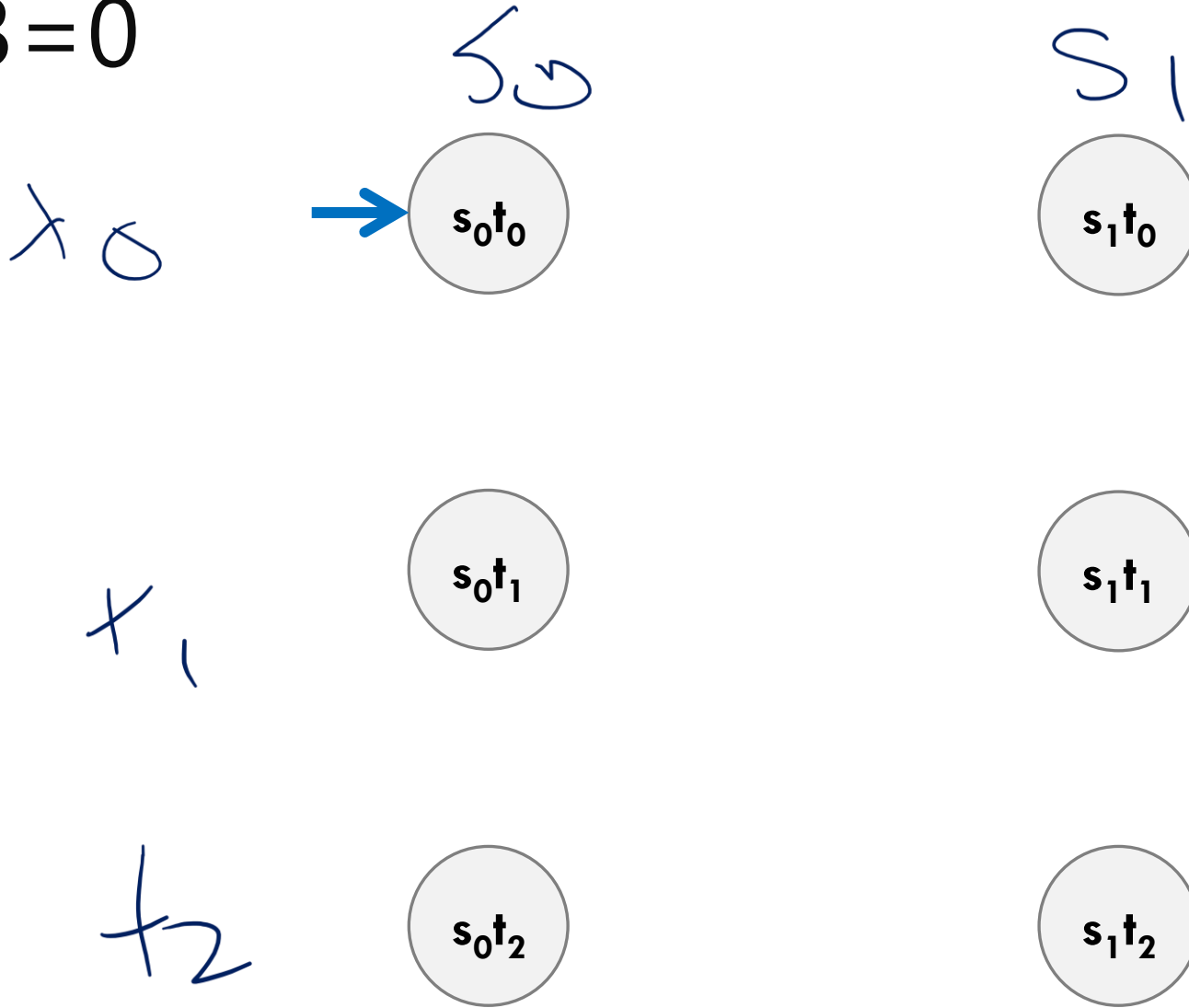
Strings over  $\{0,1,2\}$  w/ even number of 2's  
**and**  $\text{sum} \% 3 = 0$



Strings over  $\{0,1,2\}$  w/ even number of 2's **and**  $\text{sum} \% 3 = 0$



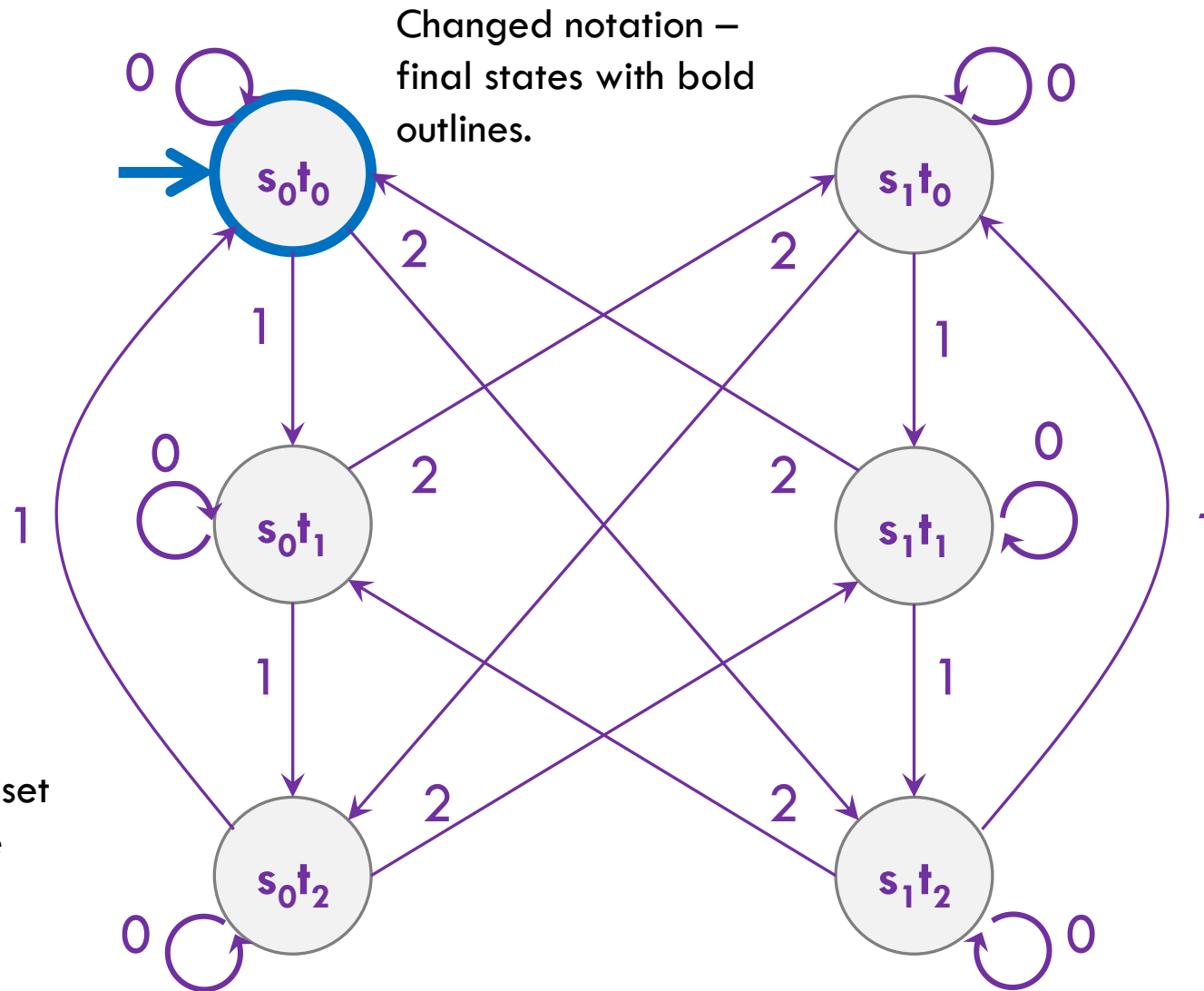
Strings over  $\{0,1,2\}$  w/ even number of 2's **and**  
 $\text{sum} \% 3 = 0$



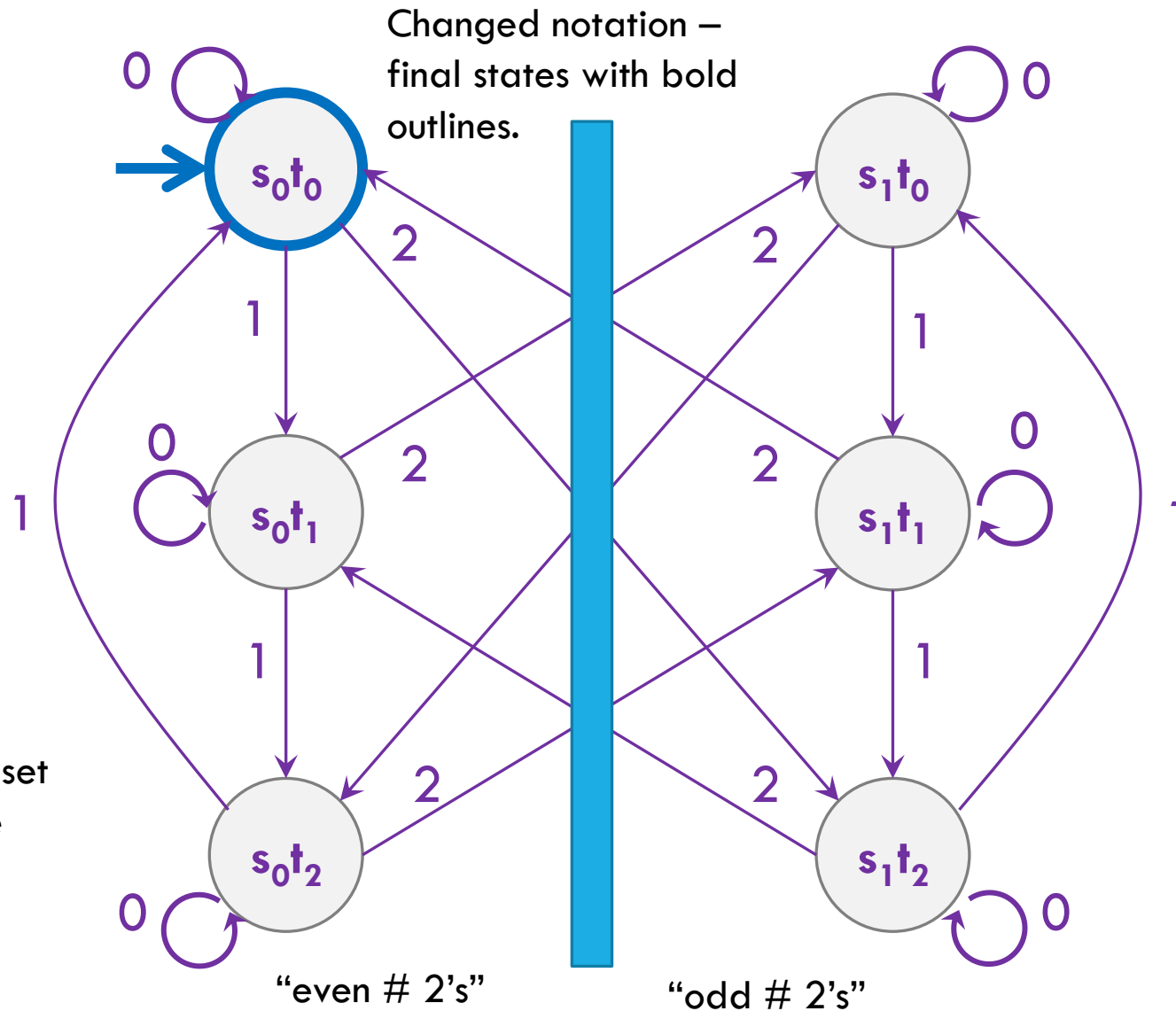
# Strings over $\{0,1,2\}$ w/ even number of 2's **and** $\text{sum} \% 3 = 0$

$S_{M_1} \times S_{M_2}$

Called the “cross product” construction (because you have a set of states equal to  $Q_1 \times Q_2$  where first two DFAs had states  $Q_1, Q_2$ . A very common trick to combine DFAs.

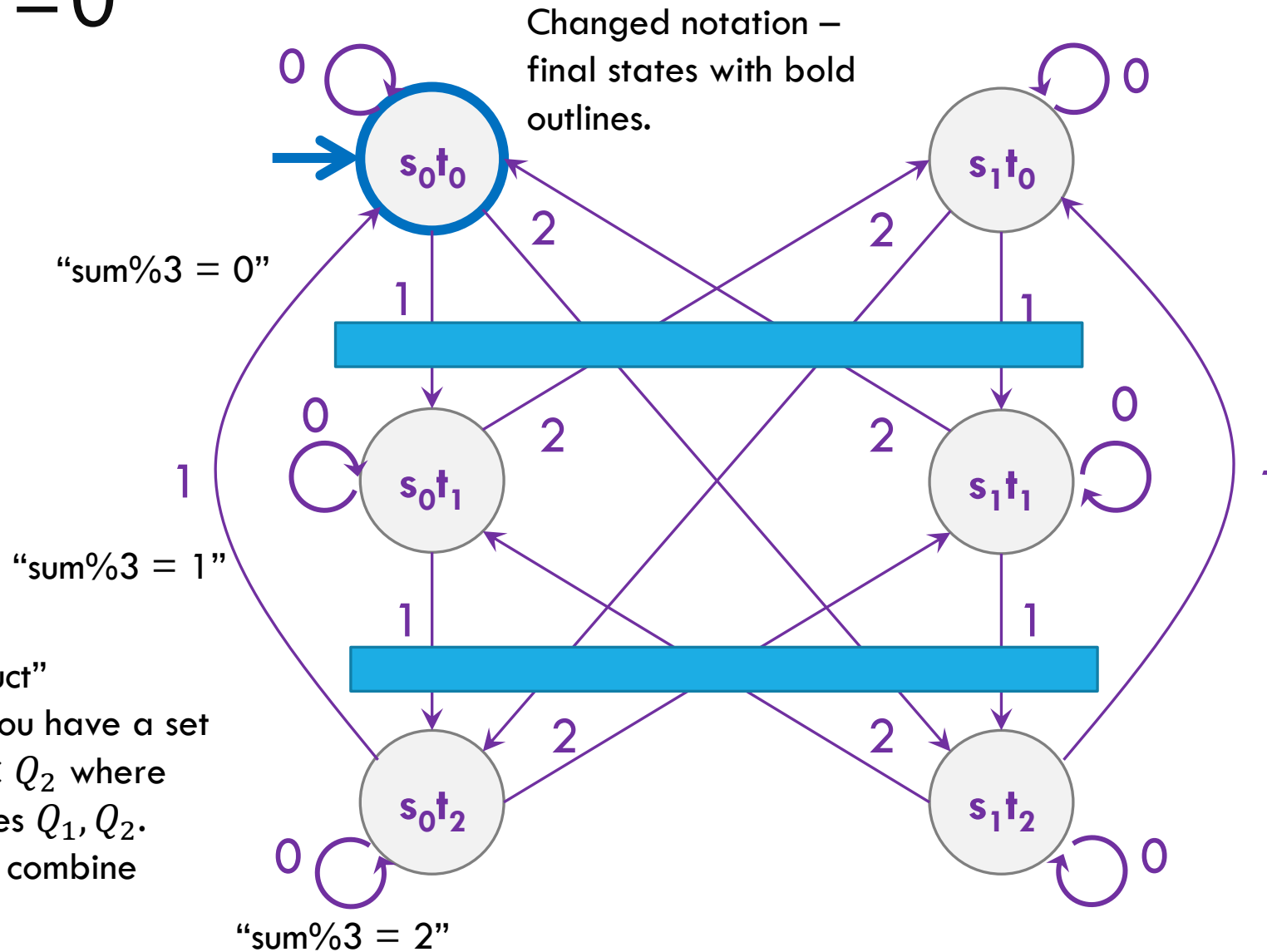


# Strings over $\{0,1,2\}$ w/ even number of 2's **and** $\text{sum} \% 3 = 0$



Called the “cross product” construction (because you have a set of states equal to  $Q_1 \times Q_2$  where first two DFAs had states  $Q_1, Q_2$ . A very common trick to combine DFAs.

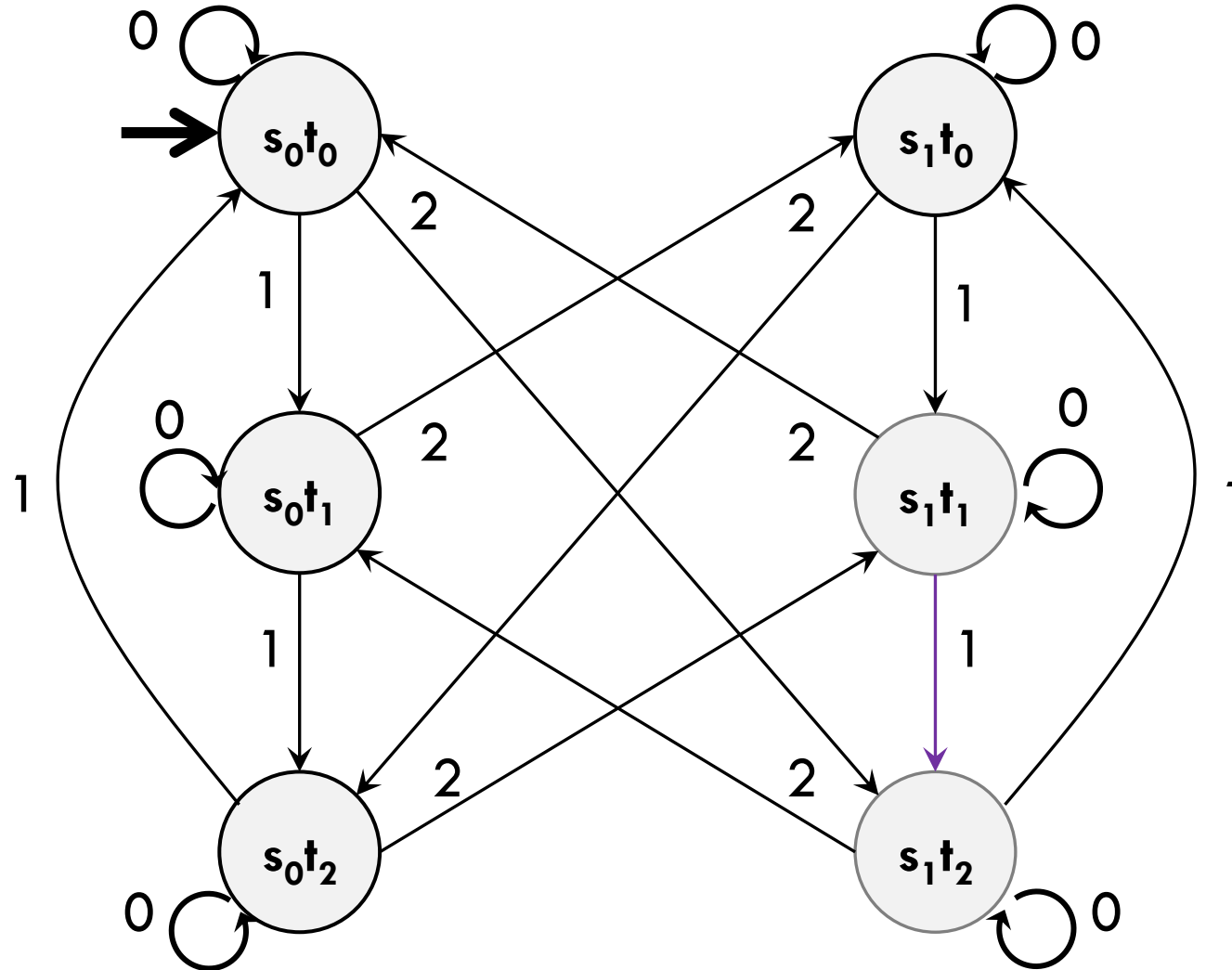
# Strings over $\{0,1,2\}$ w/ even number of 2's **and** $\text{sum} \% 3 = 0$



Called the “cross product” construction (because you have a set of states equal to  $Q_1 \times Q_2$  where first two DFAs had states  $Q_1, Q_2$ . A very common trick to combine DFAs.

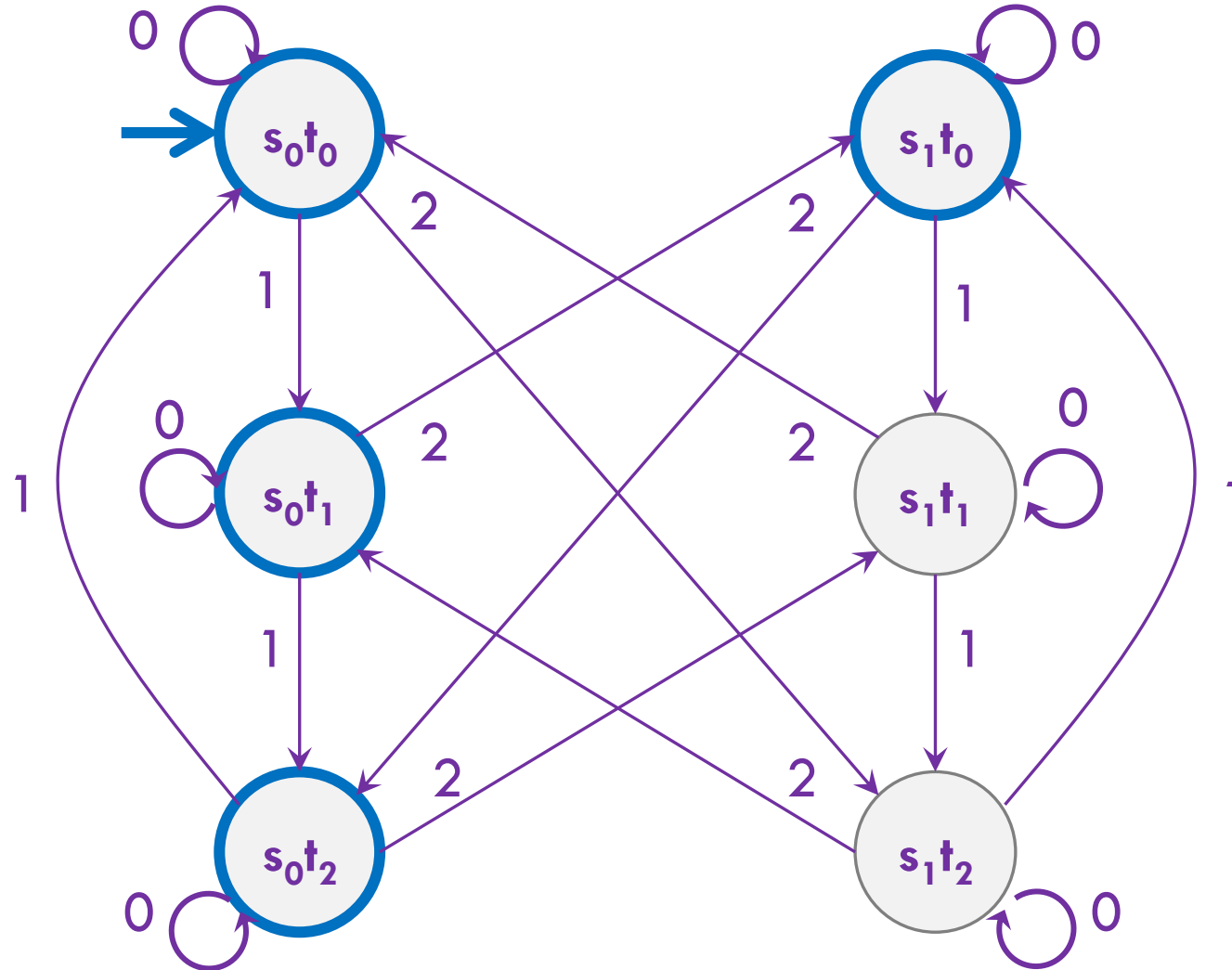
# Strings over $\{0,1,2\}$ w/ even number of 2's **OR** $\text{sum} \% 3 = 0$

Want to  
change the  
and to or –  
don't need to  
change states  
or transitions...



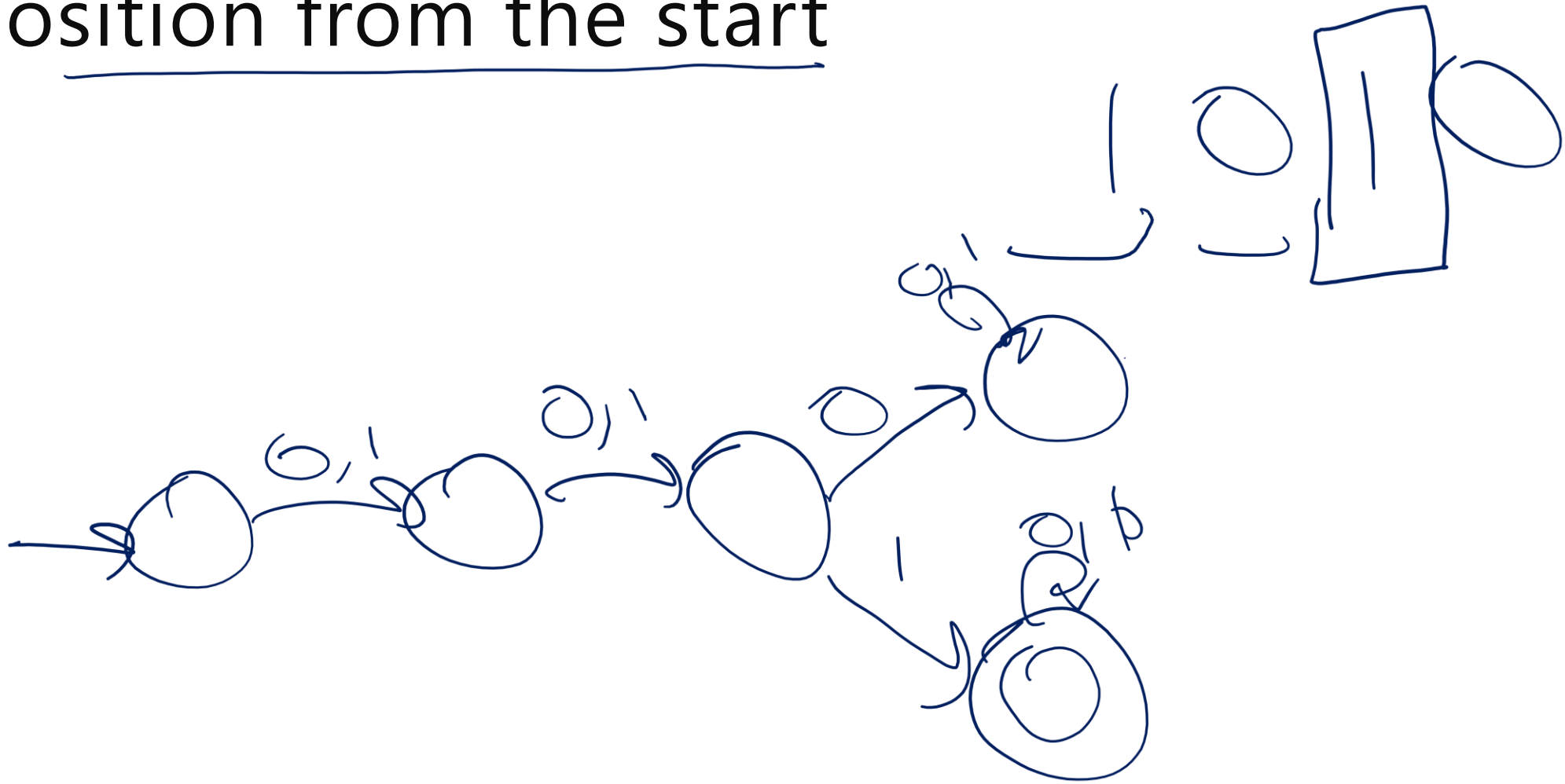
# Strings over $\{0,1,2\}$ w/ even number of 2's **OR** $\text{sum} \% 3 = 0$

Want to change  
the and to or –  
don't need to  
change states or  
transitions...  
Just which accept.

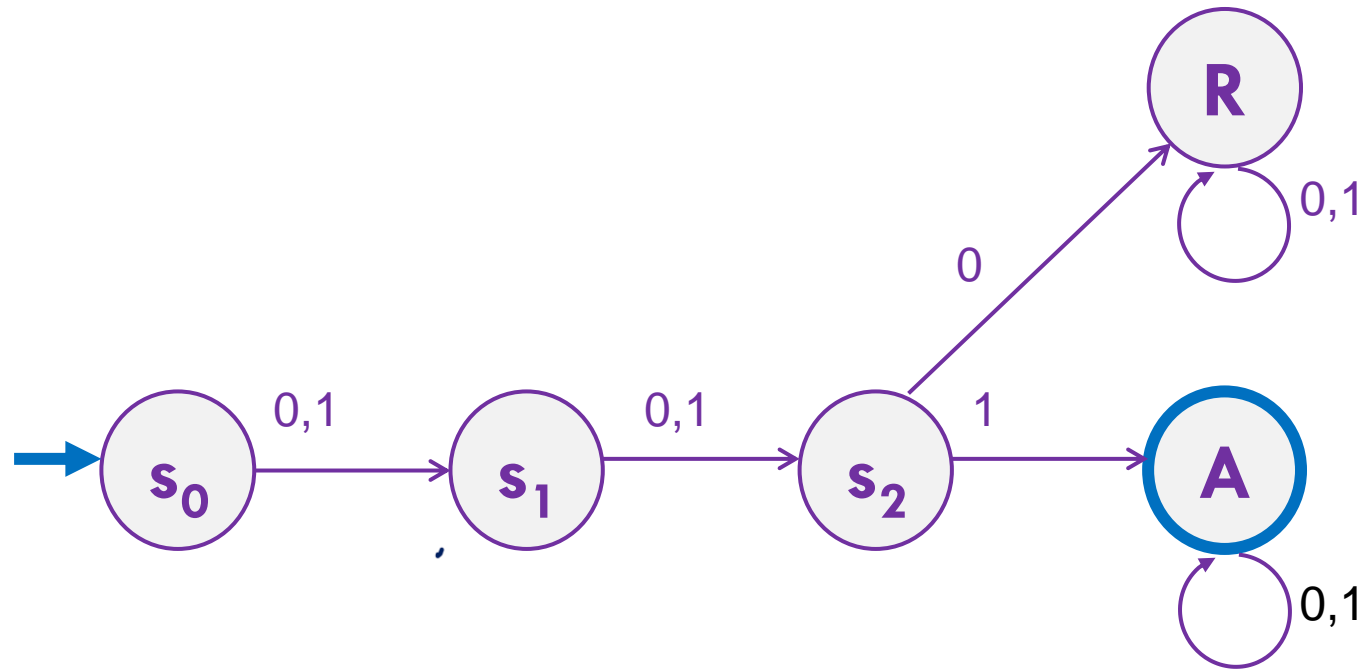




The set of binary strings with a 1 in the 3<sup>rd</sup> position from the start



The set of binary strings with a 1 in the 3<sup>rd</sup> position from the start



# The set of binary strings with a 1 in the 3<sup>rd</sup> position from the end

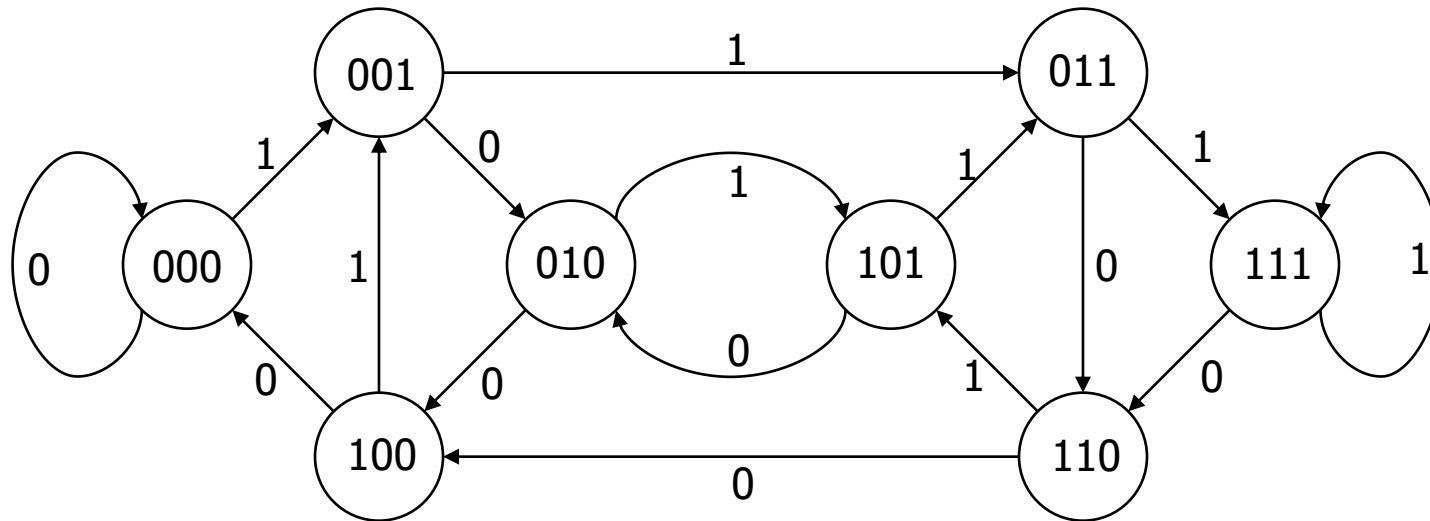
What do we need to remember?

We can't know what string was third from the end until we have read the last character.

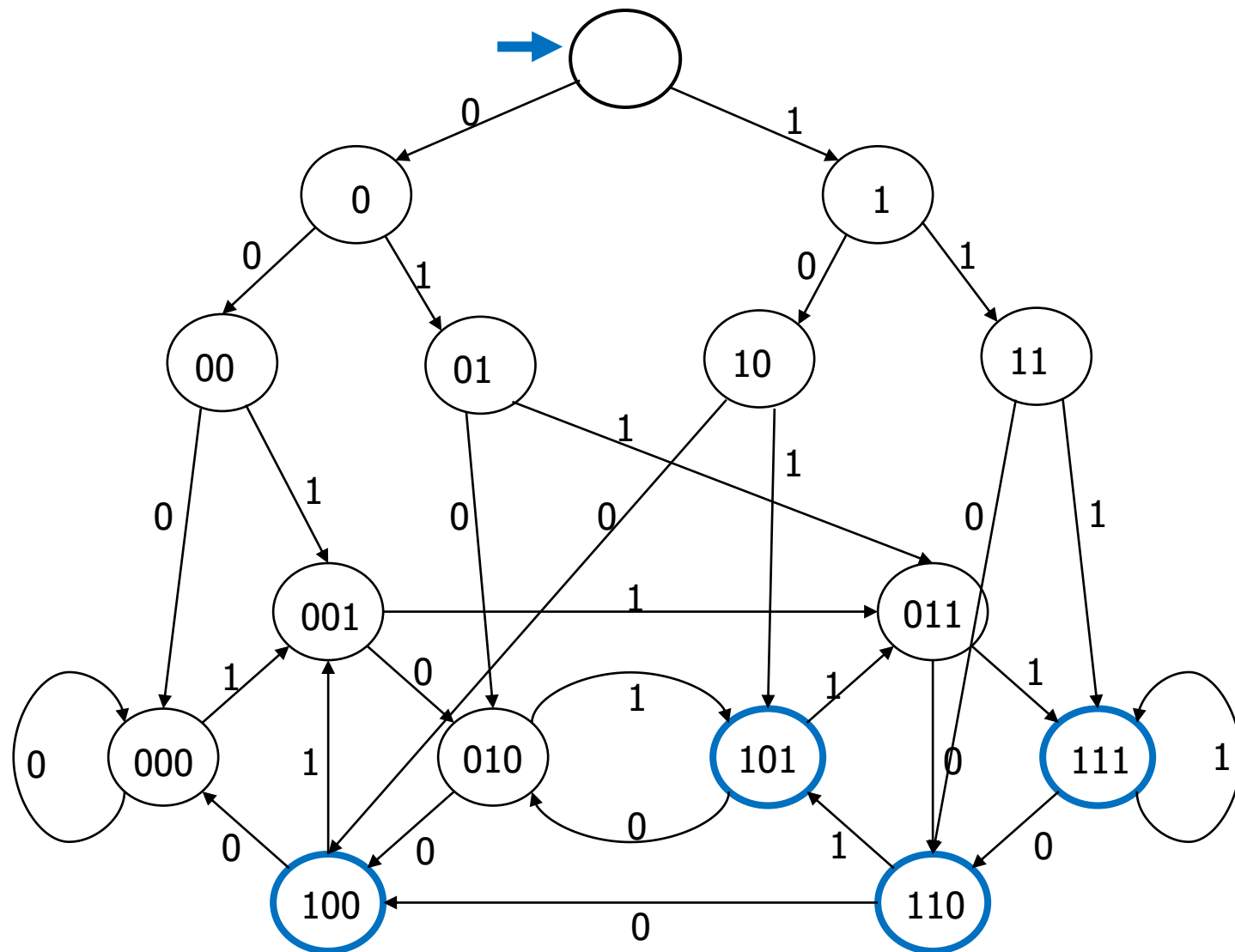
So we'll need to keep track of "the character that was 3 ago" in case this was the end of the string.

But if it's not...we'll need the character 2 ago, to update what the character 3 ago becomes. Same with the last character.

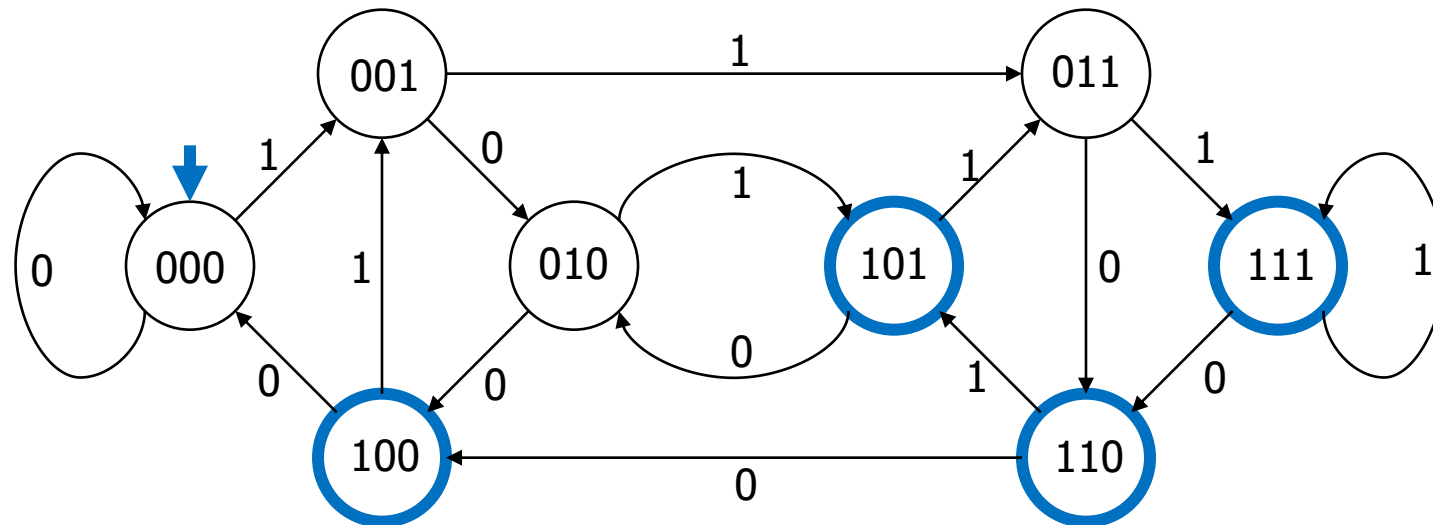
# 3 bit shift register “Remember the last three bits”



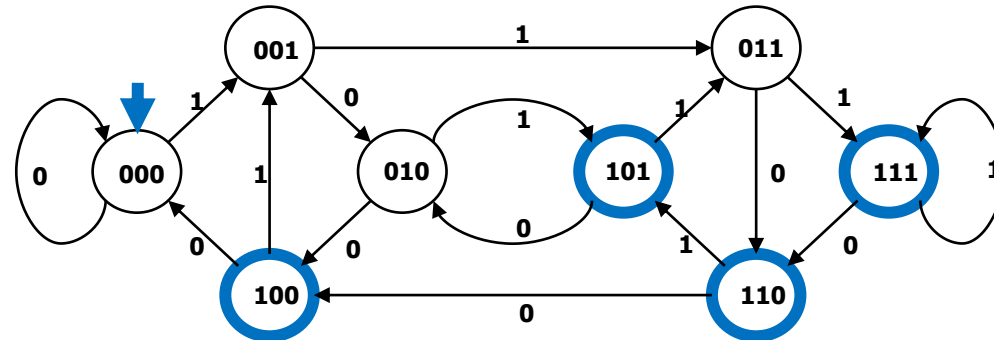
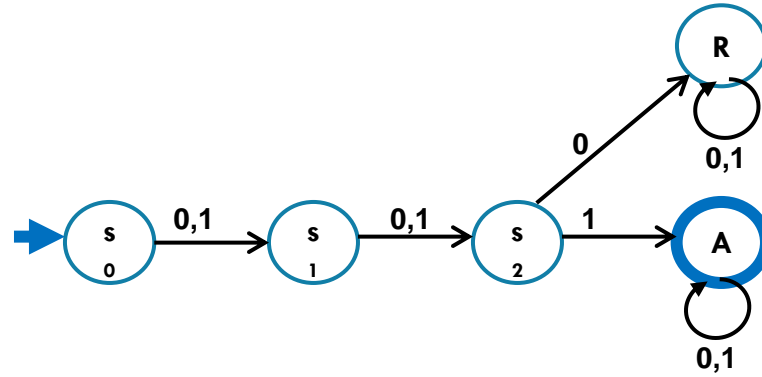
The set of binary strings with a 1 in the 3<sup>rd</sup> position from the end



The set of binary strings with a 1 in the 3<sup>rd</sup> position from the end



# The beginning versus the end



# From the beginning was “easier” than “from the end”

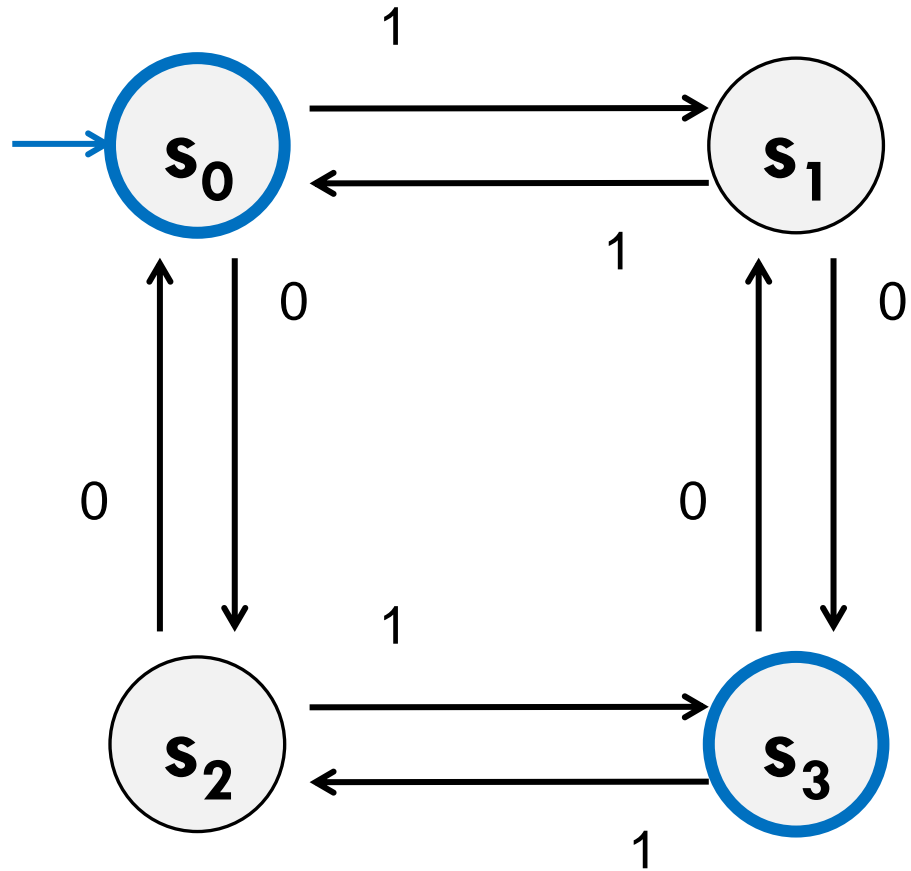
At least in the sense that we needed fewer states.

That might be surprising since a java program wouldn't be much different for those two.

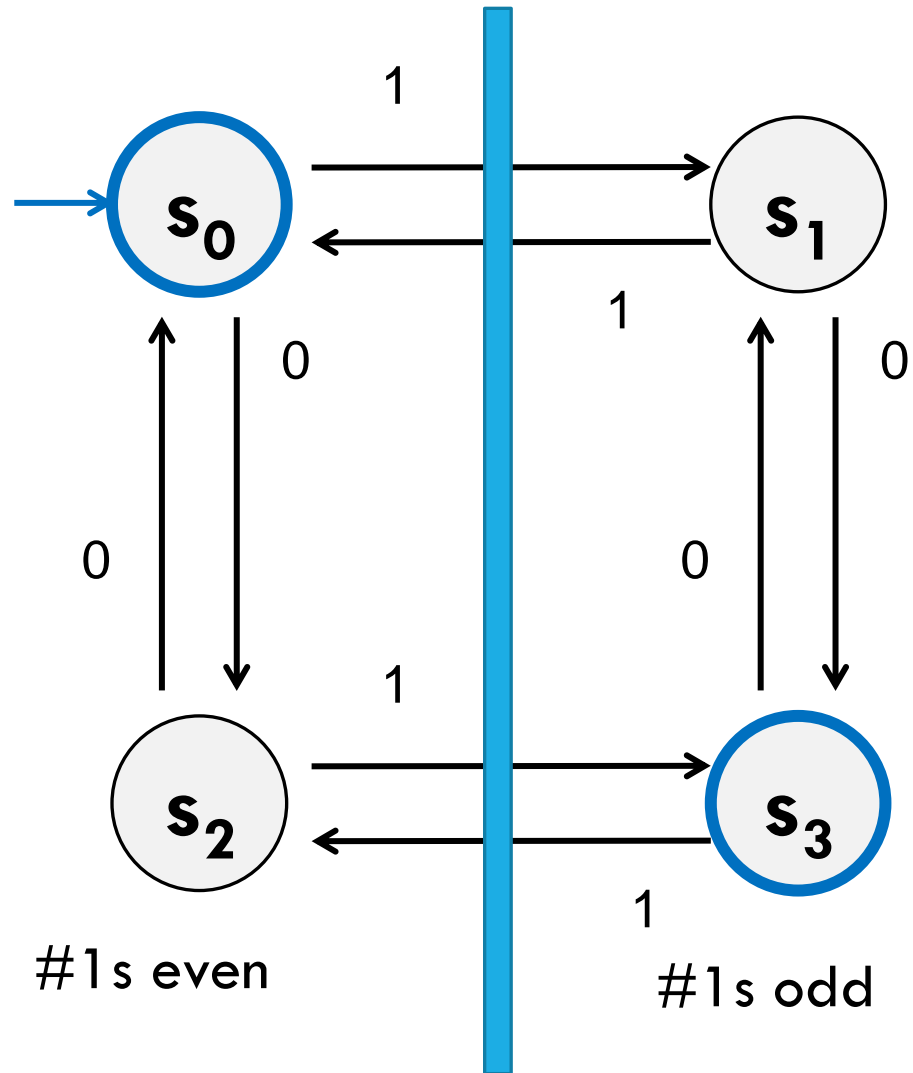
Not being able to access the full input at once limits your abilities somewhat and makes some jobs harder than others.



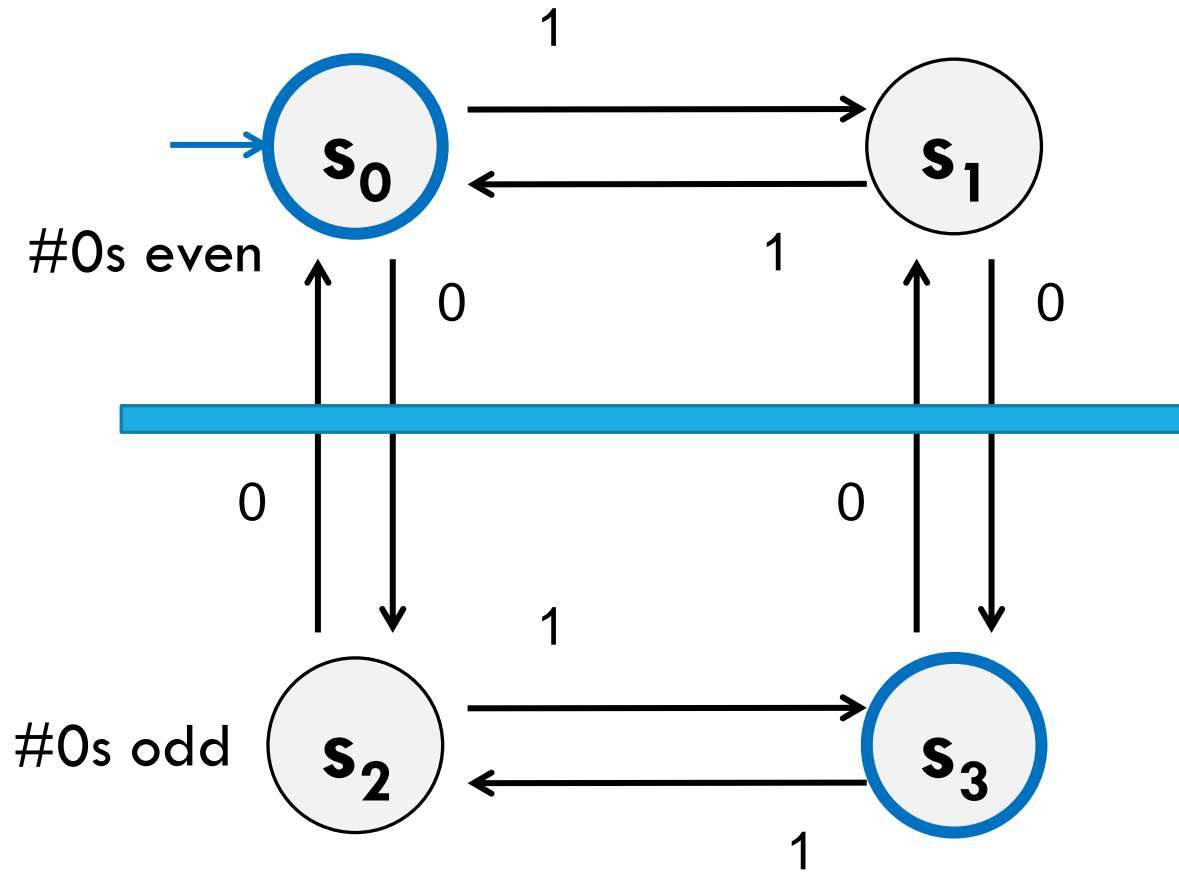
# What language does this machine recognize?



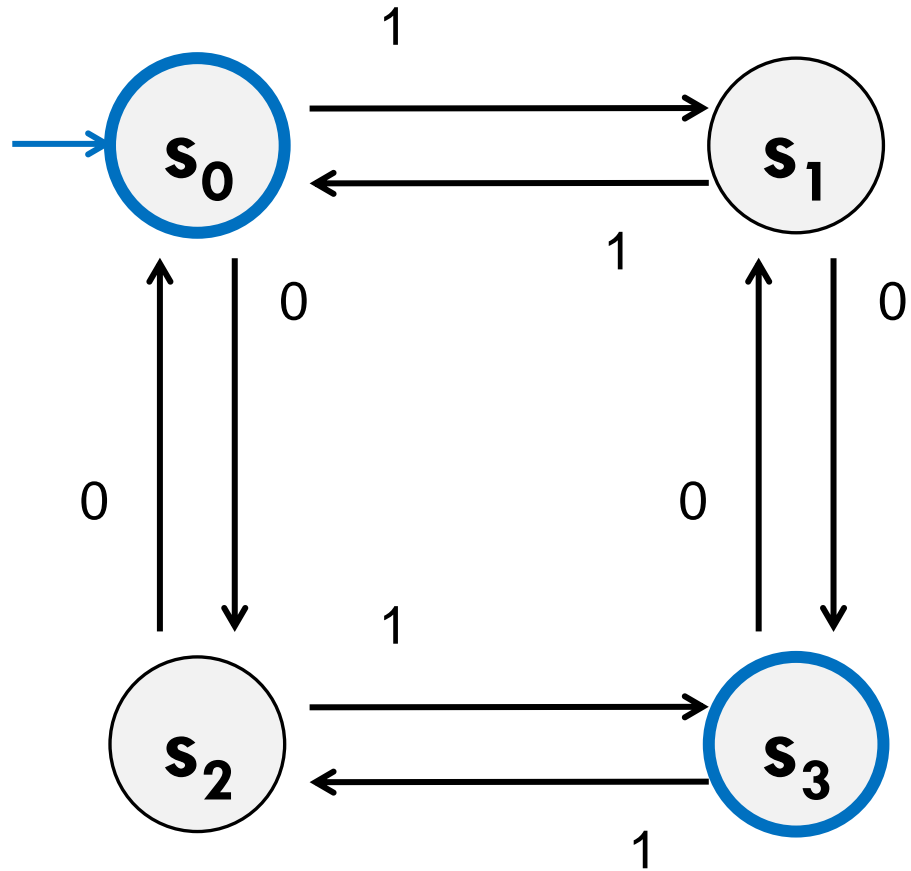
# What language does this machine recognize?



# What language does this machine recognize?



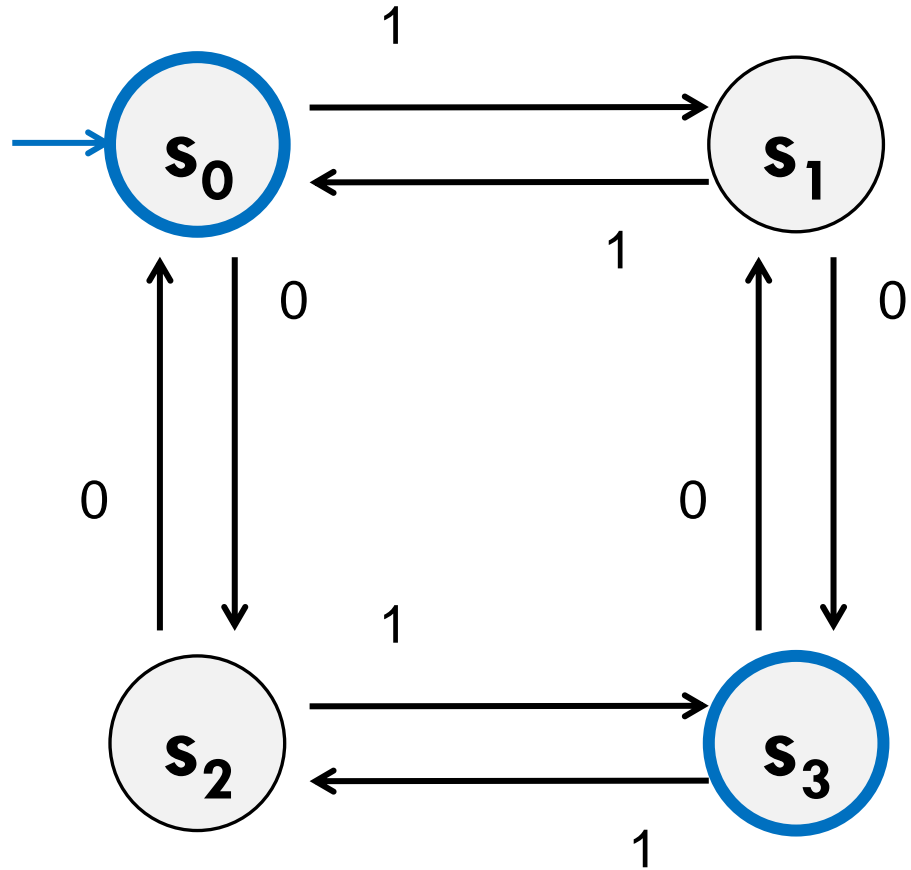
# What language does this machine recognize?



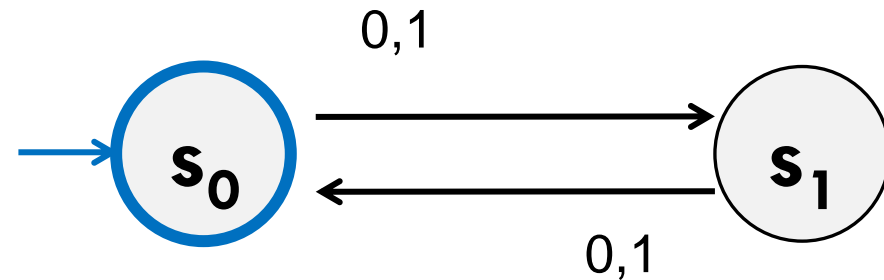
#0s is congruent to #1s (mod 2)

Wait...there's an easier way to describe that....

# What language does this machine recognize?



That's all binary strings of even length.



# Takeaways

The first DFA might not be the simplest.

Try to think of other descriptions – you might realize you can keep track of fewer things than you thought.

Boy...it'd be nice if we could know that we have the smallest possible DFA for a given language...

# DFA Minimization

We can know!

Fun fact: there is a **unique** minimum DFA for every language (up to renaming the states)

High level idea – final states and non-final states must be different.

Otherwise, hope that states can be the same, and iteratively separate when they have to go to different spots.

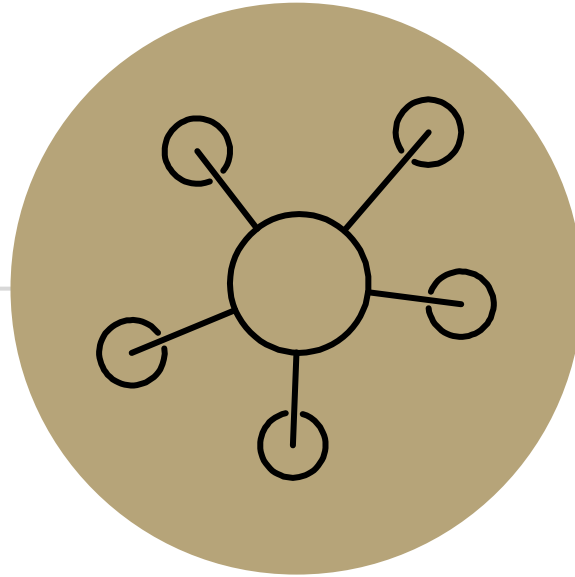
In some quarters, we cover it in detail. But...we ran out of time.

Optional slides will be posted – won't be required in HW or final but you might find it useful/interesting for your own learning.

# Next Time

What if we give the DFAs a little more power...try to get them to do more things.





Optional Content:  
Machines with output

# What are FSMs used for?

“Classic” hardware applications:

Anything where you only need to remember a very small amount of information, and have very simple update rules.

Vending machines

Elevators: need to know whether you’re going up or down, where people want to go, where people are waiting, and whether you’re going up or down. Simple rules to transition.

These days...general hardware is cheap, less likely to use custom hardware. BUT the programmer was probably still thinking about FSMs when writing the code.

# What are FSMs used for?

Theoretically – still lots of applications.

`grep` uses FSMs to analyze regular expressions (more on this later).

Useful for modeling situations where you have minimal memory.

Good model for simple AI (say simple NPCs in games).

**Technically** all of our computers are finite state machines...

But they're not usually how we think about them...more on this next week.

# Adding Output to Finite State Machines

So far we have considered finite state machines that just accept/reject strings

called "Deterministic Finite Automata" or DFAs

One can also consider finite state machines that with output

These are often used as controllers



# Vending Machine

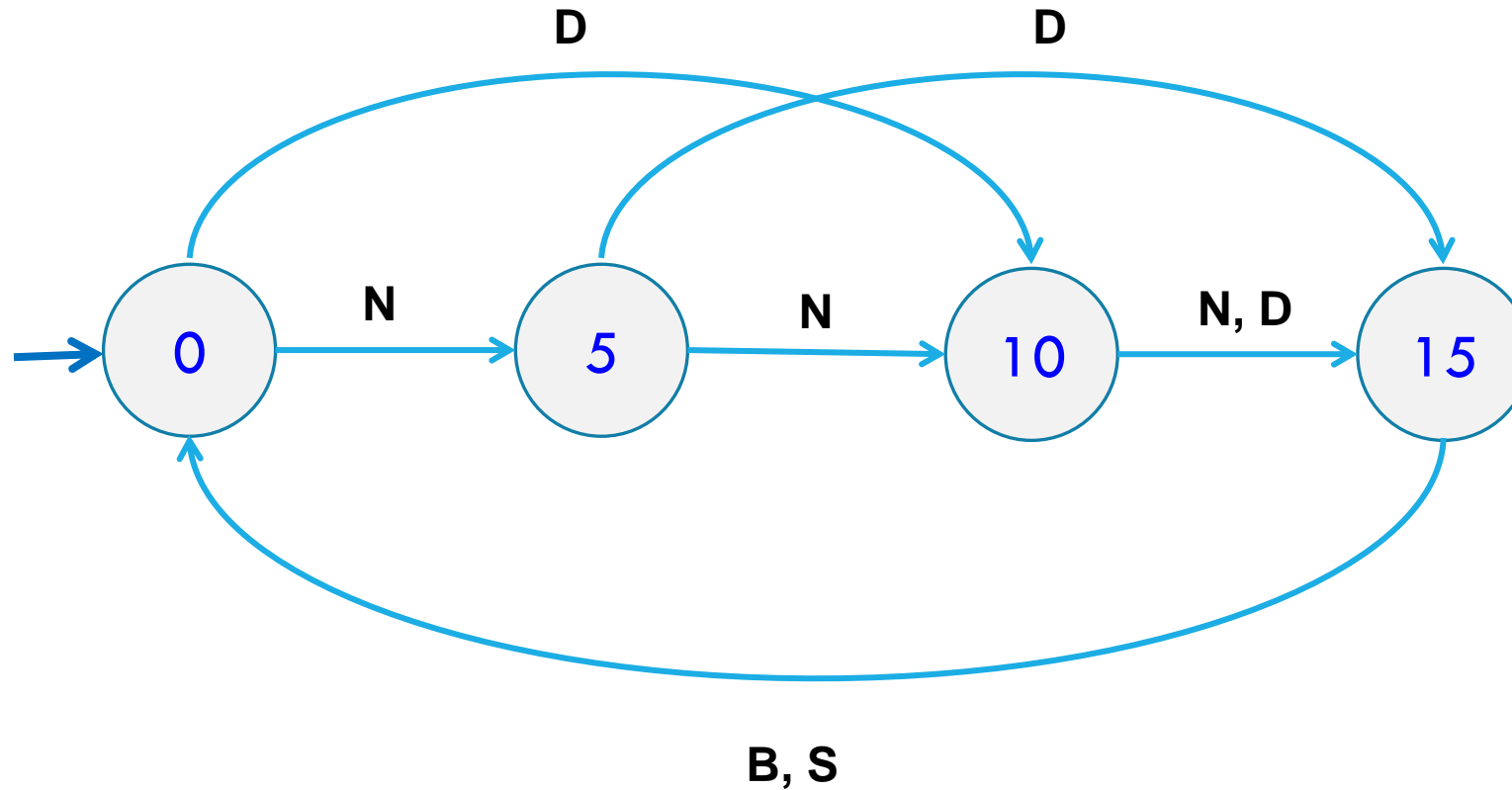


Enter 15 cents in dimes or nickels  
Press S or B for a candy bar



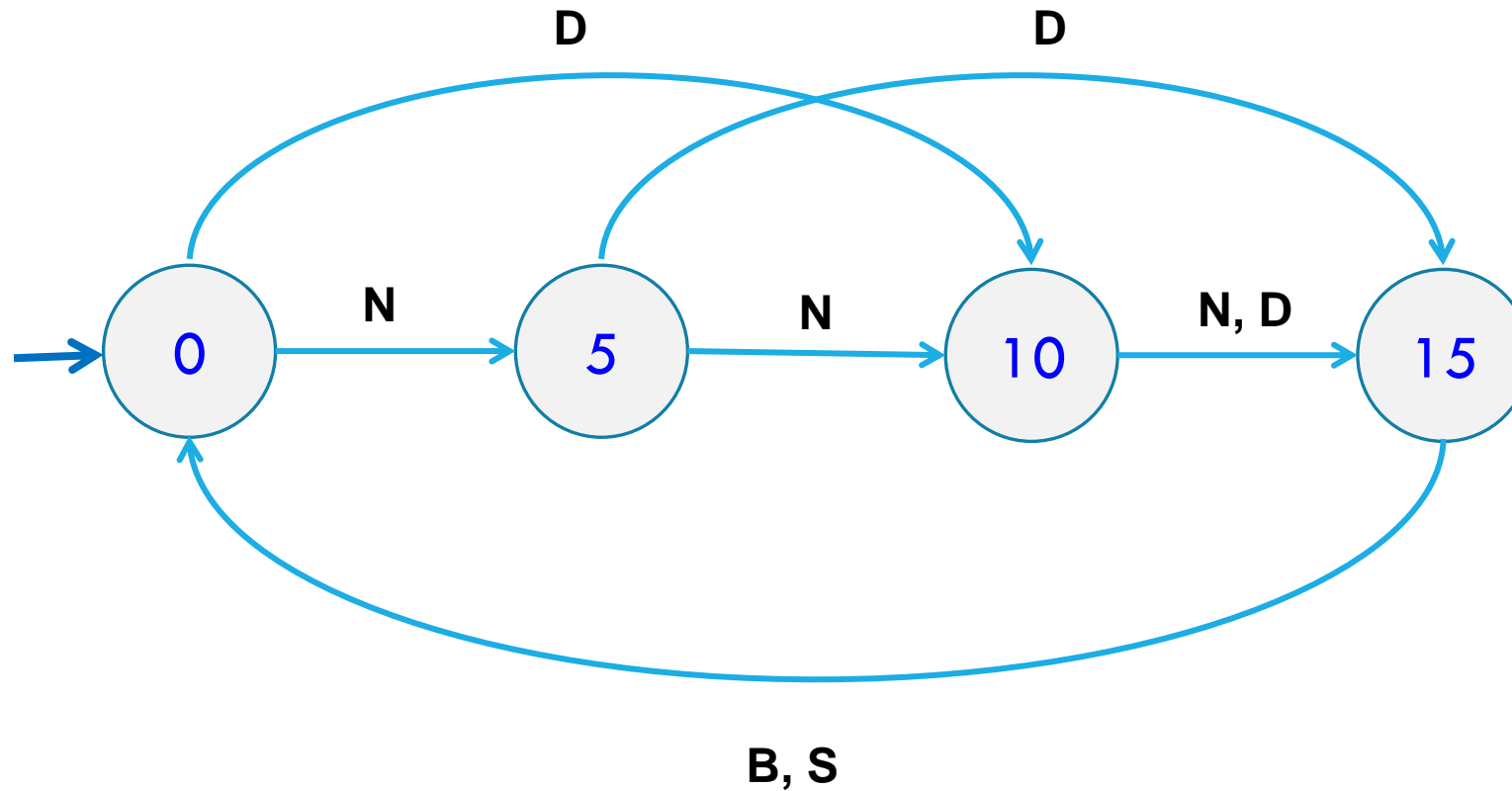
# Vending Machine v0.1

# Vending Machine, v0.1



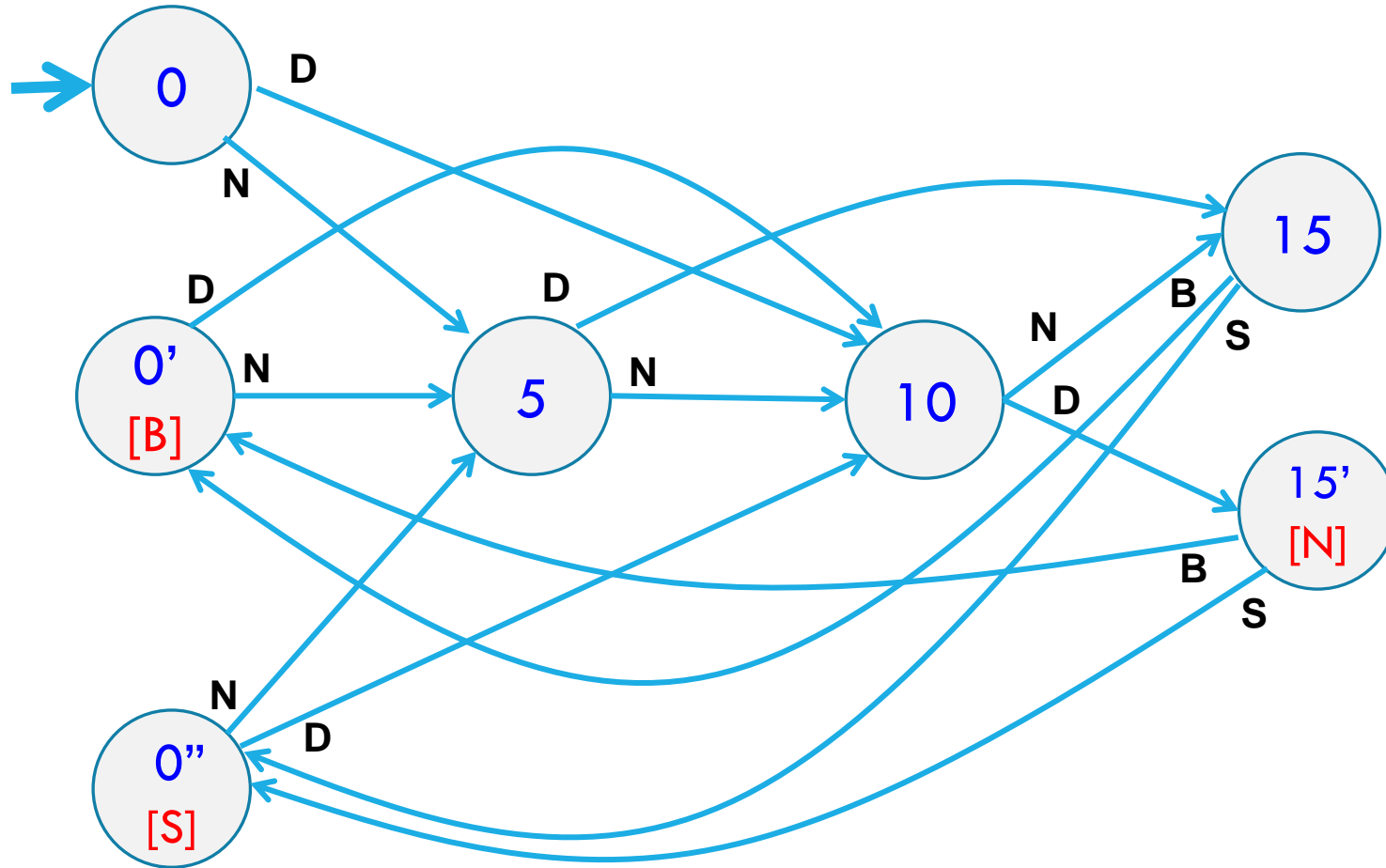
Basic transitions on **N** (nickel), **D** (dime), **B** (butterfinger), **S** (snickers)

# Vending Machine v0.2



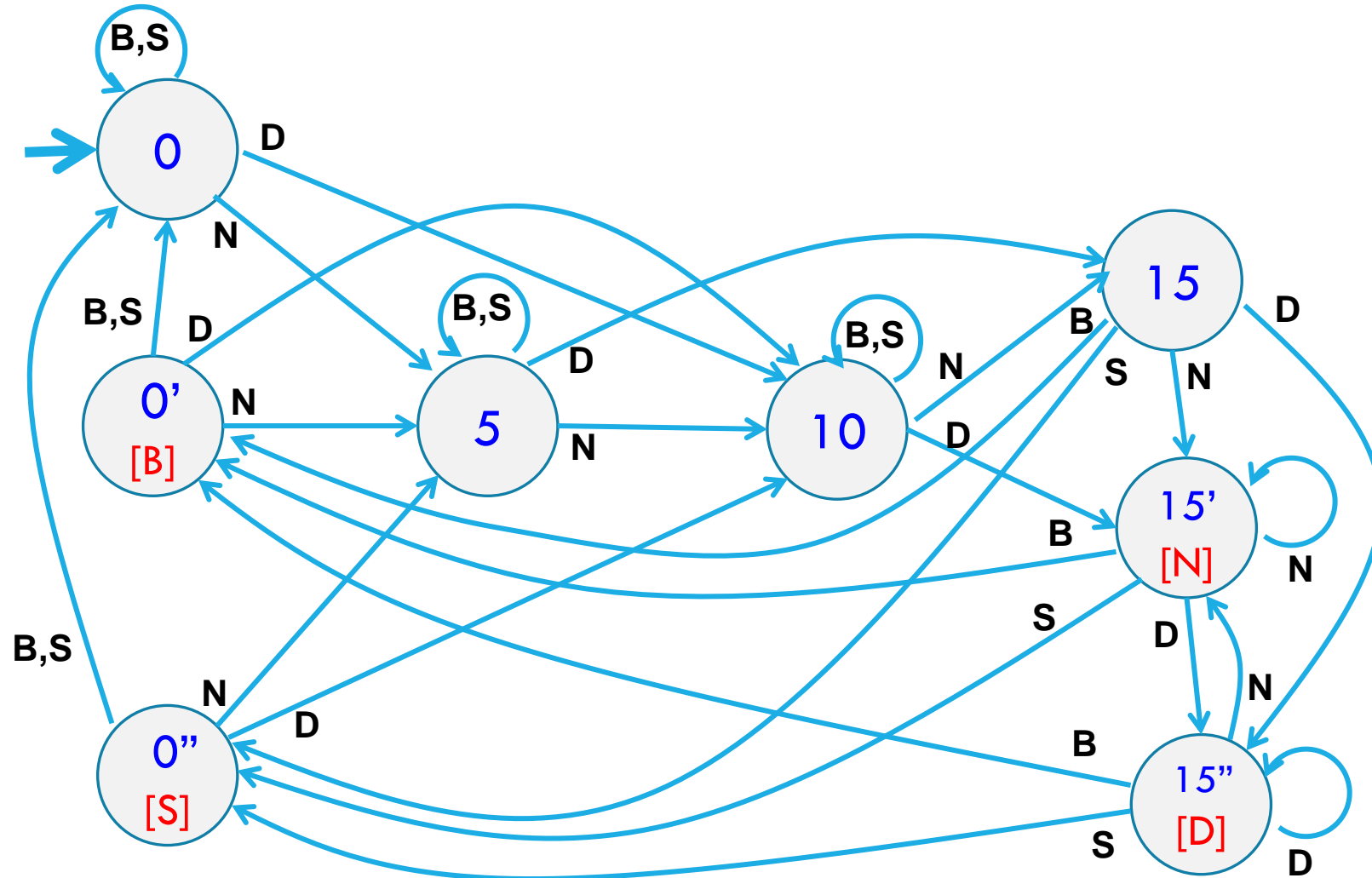


# Vending Machine, v0.2



Adding output to states: **N** – Nickel, **S** – Snickers, **B** – Butterfinger

# Vending Machine, v1.0



Adding additional “unexpected” transitions to cover all symbols for each state