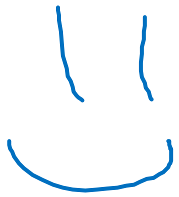


Functions And Graphs

CSE 311 Winter 2024
Lecture 20

Today

Some definitions that will help us over the next few weeks

Function definitions

Graphs and some terms

Finally start on "computation"

Computers with $O(1)$ memory

Functions

Some types of functions

Why?

We'll want to talk about sizes of infinite sets during the last week of classes. It'll help us find problems our computers can't solve.

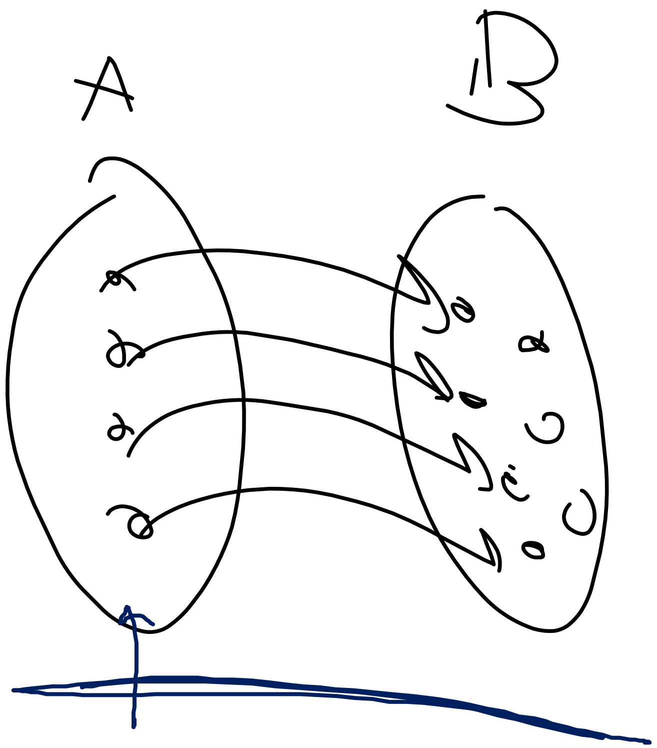
Ok, but why now?

It'll let us practice set proofs a bit more over the next few weeks!

Two Requirements for a Bijection

A function $f: A \to B$ maps every element of A to one element of B

A is the "domain", B is the "co-domain"



One-to-one (aka injection)

A function f is one-to-one iff

$$\forall a \forall b (f(a) = f(b) \to a = b)$$

That is, every output has at most one possible input.

$$f : A \to B$$

One-to-one (injection)

What did that definition say?

$\forall a \forall b (f(a) = f(b) \rightarrow a = b)$

In contrapositive that looks like

$\forall a \forall b \big(a \neq b \rightarrow f(a) \neq f(b)\big)$

So if you get two different inputs, then you get two different outpus.

One-to-one proofs

It's a forall statement! We know how to prove it.

Let $f: \mathbb{Z} \to \mathbb{Z}$ be the function given by $f(x) = x + 5$.

Claim: f is one-to-one

Proof:

What's the outline? What do we introduce, what do we assume, what's our target?

One-to-one proofs

It's a forall statement! We know how to prove it.

Let $f: \mathbb{Z} \to \mathbb{Z}$ be the function given by $f(x) = x + 5$.

Claim: f is one-to-one

Proof: Let a, b be arbitrary elements of our domain, and suppose $f(a) = f(b)$.

…

$a = b$

One-to-one proofs

It's a forall statement! We know how to prove it.

Let $f: \mathbb{Z} \to \mathbb{Z}$ be the function given by $f(x) = x + 5$.

Claim: f is one-to-one

Proof: Let a, b be arbitrary elements of our domain, and suppose $f(a) = f(b)$.

By definition of the function, we have $a + 5 = b + 5$

Subtracting 5 from each side, we have $a = b$, meeting the definition of one-to-one.

Two Requirements for a Bijection

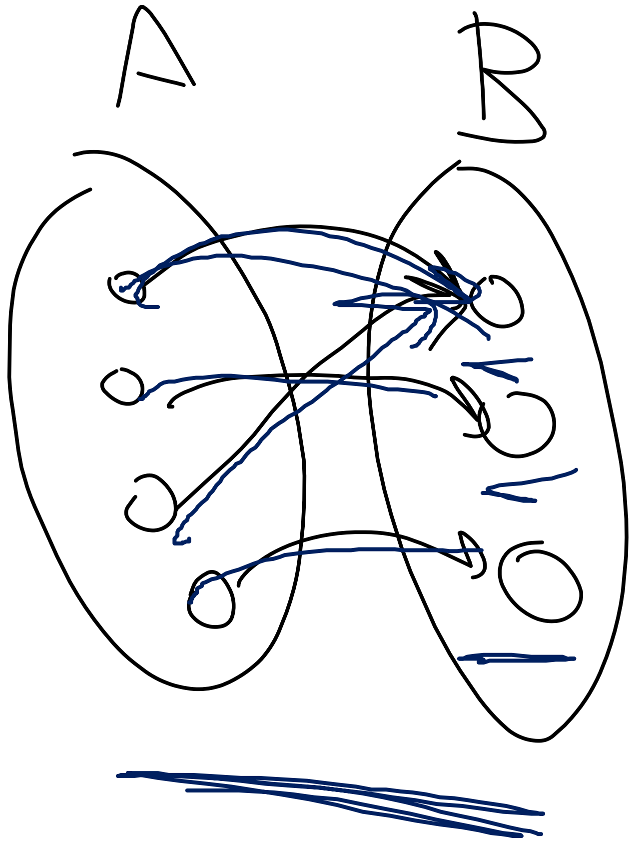
A function $f: A \to B$ maps every element of A to one element of B

A is the "domain", B is the "co-domain"

Onto (aka surjection)

A function $f: A \to B$ is onto iff
$\forall b \in B \exists a \in A(b = f(a))$

Every output has at least one input that maps to it.



~~One-to-one~~ Onto proofs

It’s a forall statement! We know how to prove it.

Let $f: \mathbb{Z} \rightarrow \mathbb{Z}$ be the function given by $f(x) = x + 5$.

Claim: f is onto

Proof:

What’s the outline? What do we introduce, what do we assume, what’s our target?

~~One-to-one~~ Onto proofs

It's a forall statement! We know how to prove it.

Let $f: \mathbb{Z} \to \mathbb{Z}$ be the function given by $f(x) = x + 5$.

Claim: f is ~~one-to-one~~ Onto

Proof: Let b be an arbitrary element of the codomain.

$(b-5)+5$

Consider $a =$...

...

So $f(a) = b$

~~One-to-one~~ proofs

Onto

It's a forall statement! We know how to prove it.

Let $f: \mathbb{Z} \to \mathbb{Z}$ be the function given by $f(x) = x + 5$.

Claim: f is ~~one-to-one~~ onto

Proof: Let b be arbitrary element of the codomain.

Let $a = b - 5$

Observe that $f(a) = a + 5 = b - 5 + 5 = b$.

Since $b \in \mathbb{Z}$, a is also an integer so it is in the domain. Thus f meets the definition of onto.

Bijection

One-to-one (aka injection)

A function f is one-to-one iff
$\forall a \forall b (f(a) = f(b) \rightarrow a = b)$

Onto (aka surjection)

A function $f: A \rightarrow B$ is onto iff
$\forall b \in B \exists a \in A (b = f(a))$

Bijection

A function $f: A \rightarrow B$ is a bijection iff
f is one-to-one and onto

A bijection maps every element of the domain to **exactly** one element of the co-domain, and every element of the domain to **exactly** one element of the domain.

co

Why do we care about bijections?

Bijections create a (confusingly-named) one-to-one correspondence between sets.

There is a bijection $f: A \to B$ if and only if A and B are the same size.

A bijections "matches the elements up"

For finite sets we usually tell which of two sets is bigger by counting the number of elements in each and comparing the numbers.

These functions let you compare set sizes even if you can't count the elements. We'll use that idea for infinite sets in a few weeks.

Graphs

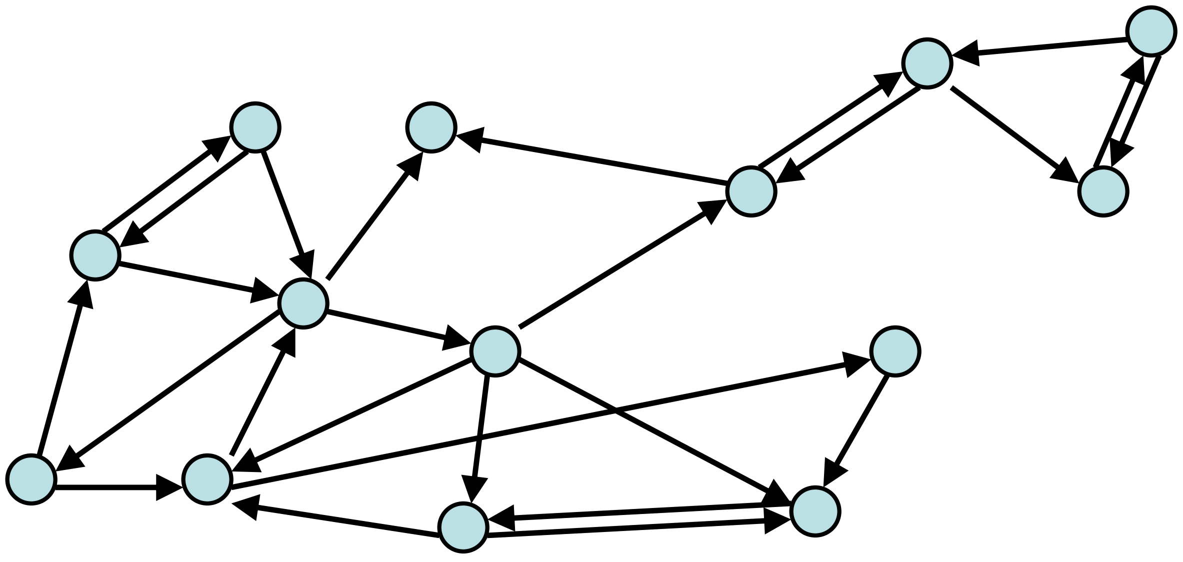
Directed Graphs

$G = (V, E)$

V is a set of vertices (an underlying set of elements)

E is a set of edges (ordered pairs of vertices; i.e. connections from one to the next).

Path $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$
Simple Path: path with all v_i distinct
Cycle: path with $v_0 = v_k$ (and $k > 0$)
Simple Cycle: simple path plus edge (v_k, v_0) with $k > 0$



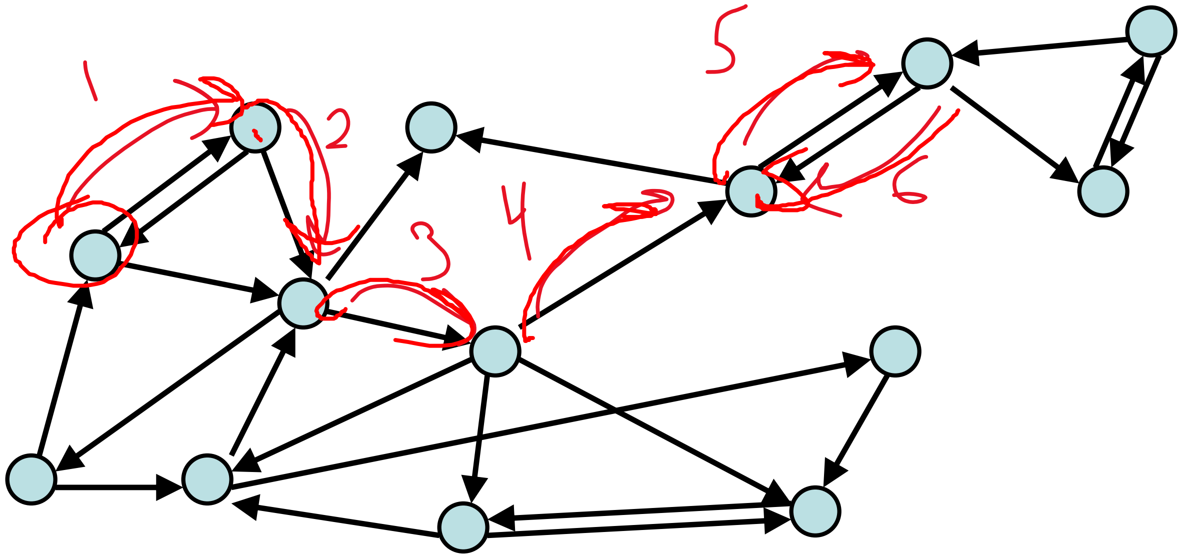
Directed Graphs

$G = (V, E)$

V is a set of vertices (an underlying set of elements)

E is a set of edges (ordered pairs of vertices; i.e. connections from one to the next).

Path $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$
Simple Path: path with all v_i distinct
Cycle: path with $v_0 = v_k$ (and $k > 0$)
Simple Cycle: simple path plus edge (v_k, v_0) with $k > 0$



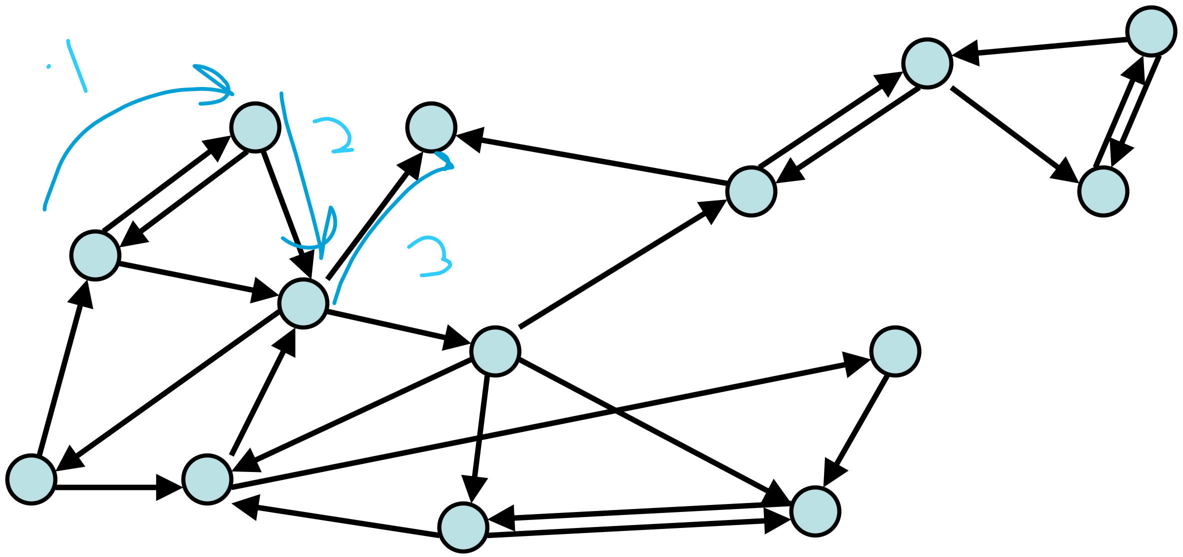
Directed Graphs

$G = (V, E)$

V is a set of vertices (an underlying set of elements)

E is a set of edges (ordered pairs of vertices; i.e. connections from one to the next).

Path $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$
Simple Path: path with all v_i distinct
Cycle: path with $v_0 = v_k$ (and $k > 0$)
Simple Cycle: simple path plus edge (v_k, v_0) with $k > 0$



Directed Graphs

$G = (V, E)$

V is a set of vertices (an underlying set of elements)

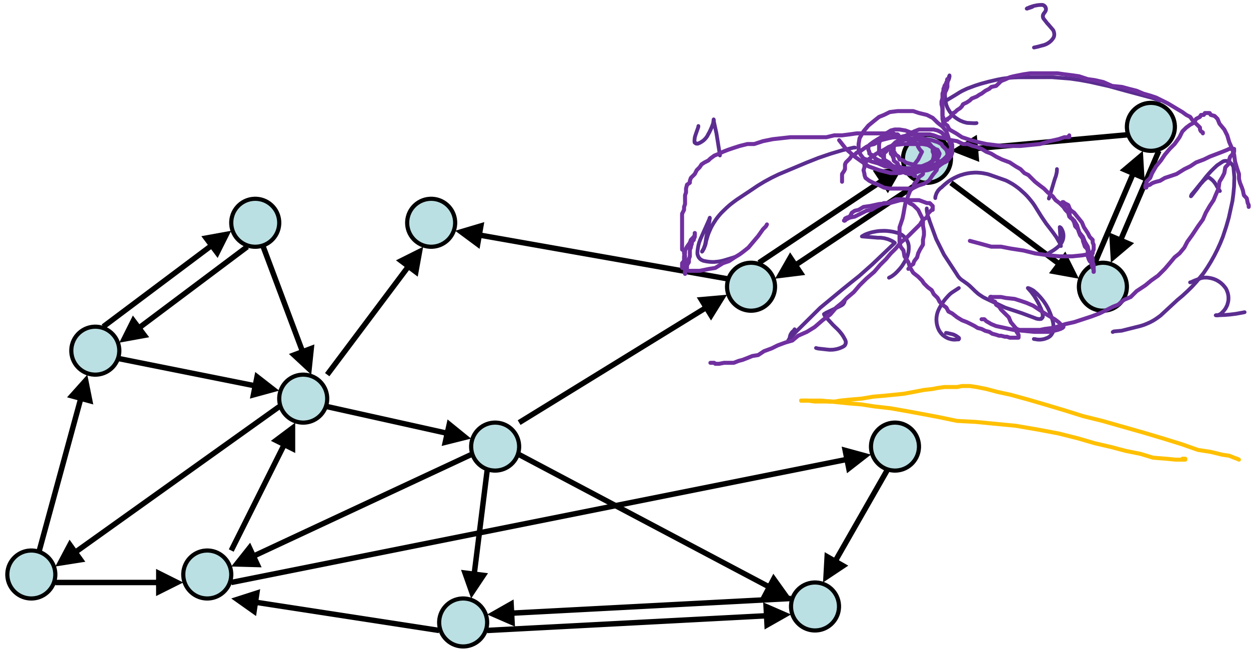
E is a set of edges (ordered pairs of vertices; i.e. connections from one to the next).

Path $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$

Simple Path: path with all v_i distinct

Cycle: path with $v_0 = v_k$ (and $k > 0$)

Simple Cycle: simple path plus edge (v_k, v_0) with $k > 0$



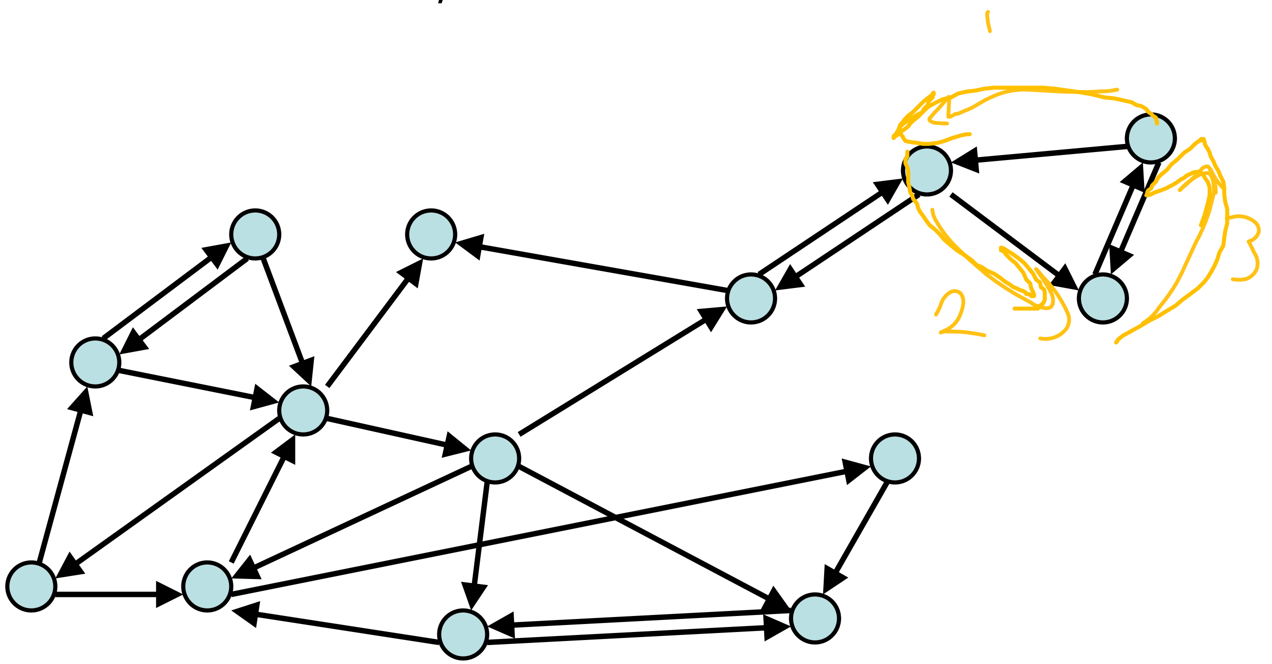
Directed Graphs

$G = (V, E)$

V is a set of vertices (an underlying set of elements)

E is a set of edges (ordered pairs of vertices; i.e. connections from one to the next).

Path $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$
Simple Path: path with all v_i distinct
Cycle: path with $v_0 = v_k$ (and $k > 0$)
Simple Cycle: simple path plus edge (v_k, v_0) with $k > 0$



Finite State Machines

(Tiny little computers)

Last Two 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?

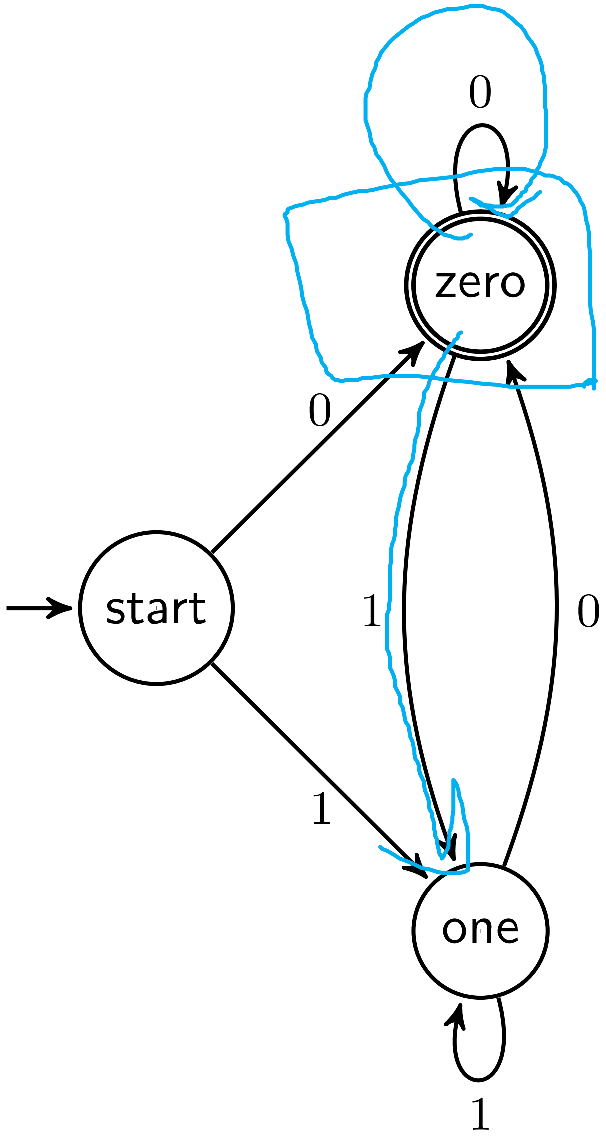
Is this input list sorted?

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).

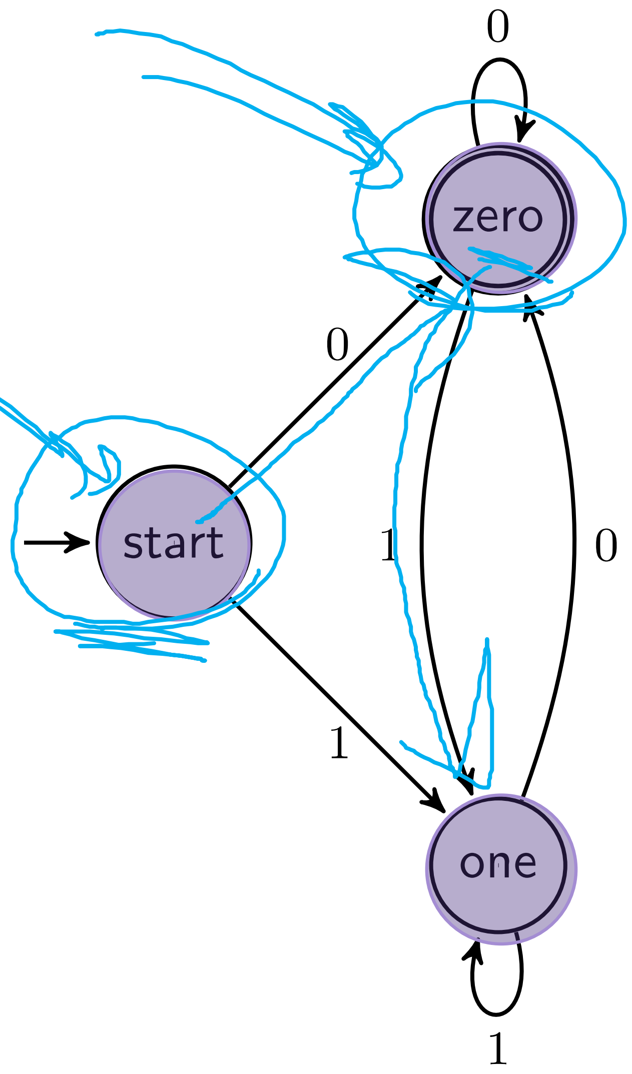


Let's see an example

Input string:

011 — rejected

1010 — accepted

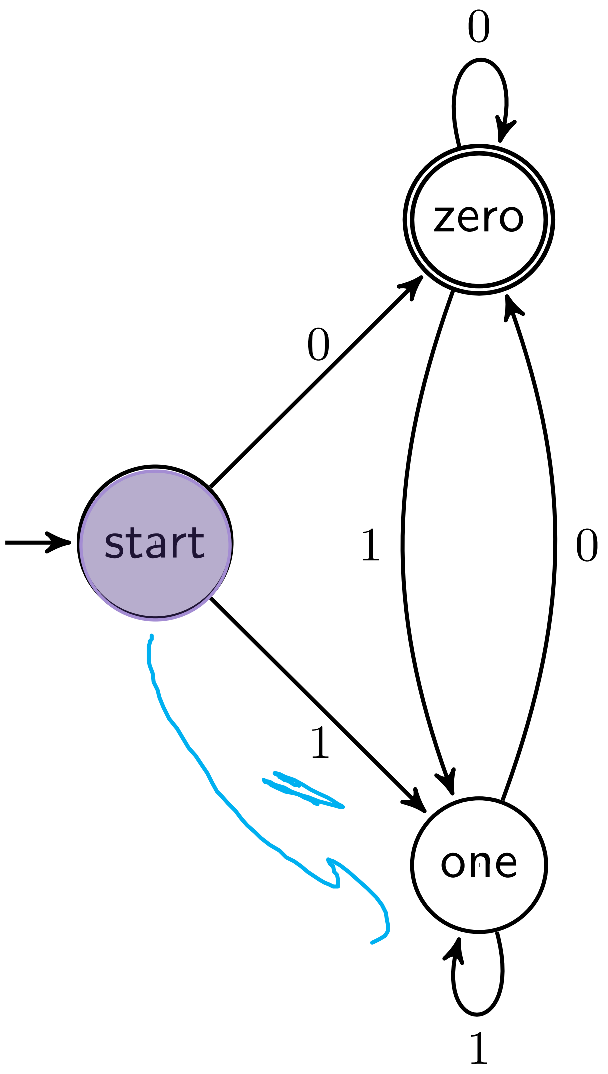


Let's see an example

Input string:

011

1010

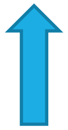


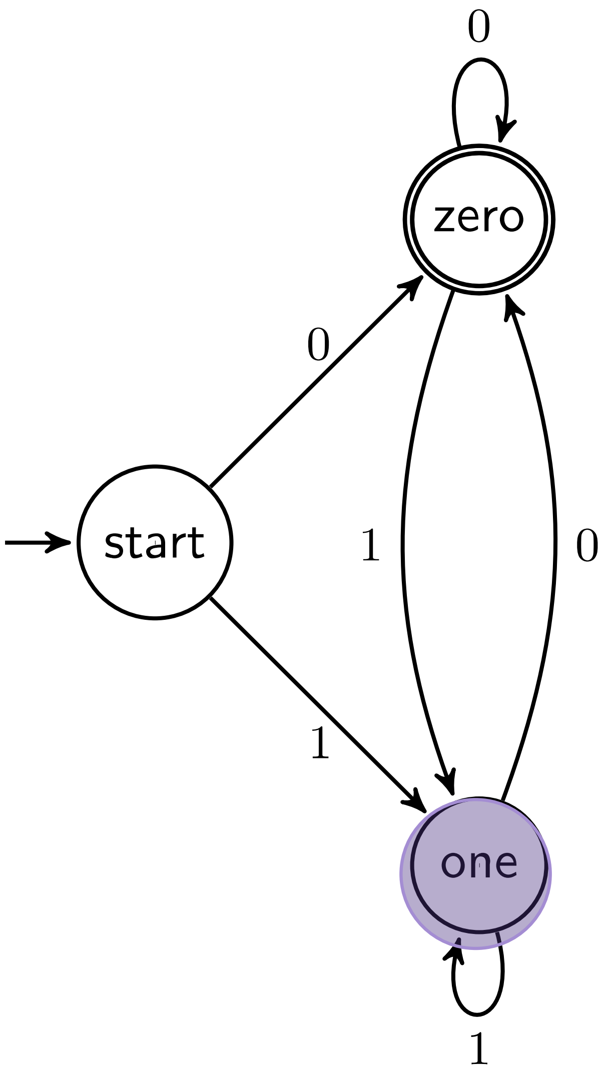
Let’s see an example

Input string:

011

1010



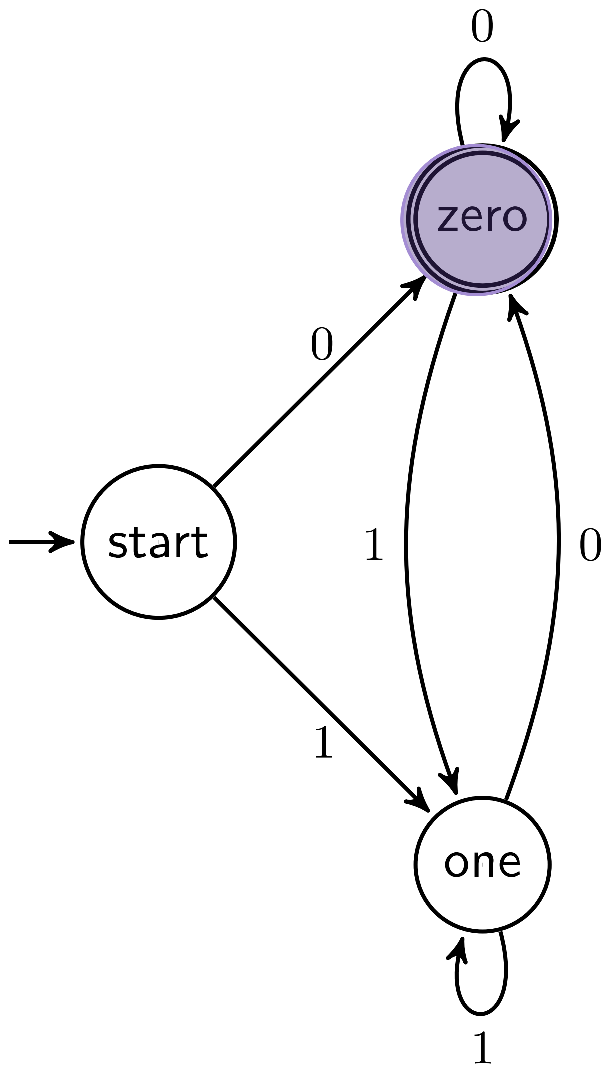


Let’s see an example

Input string:

011

1010

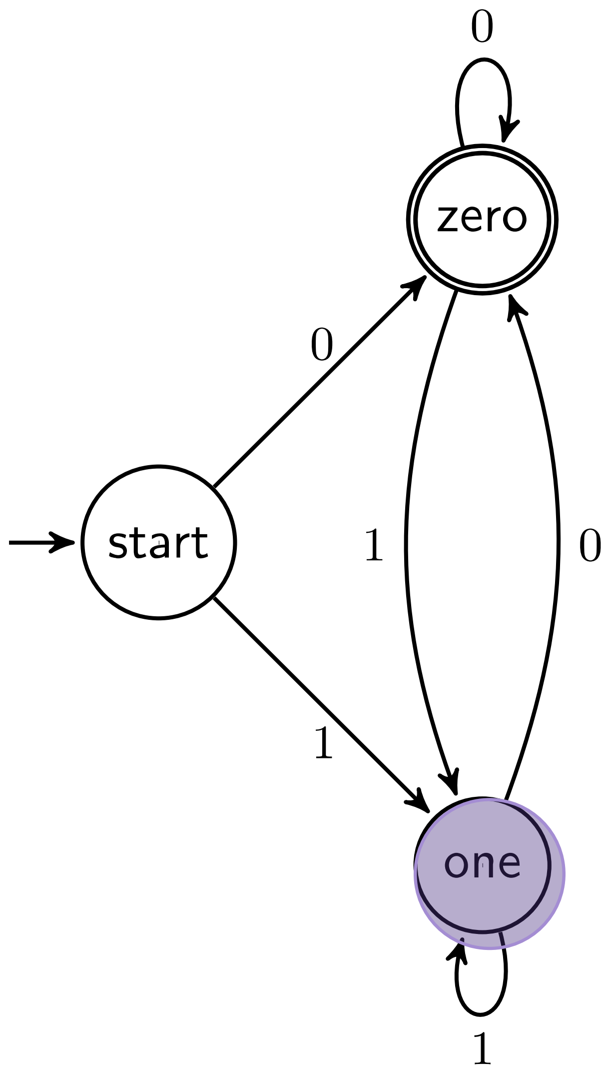


Let's see an example

Input string:

011

1010

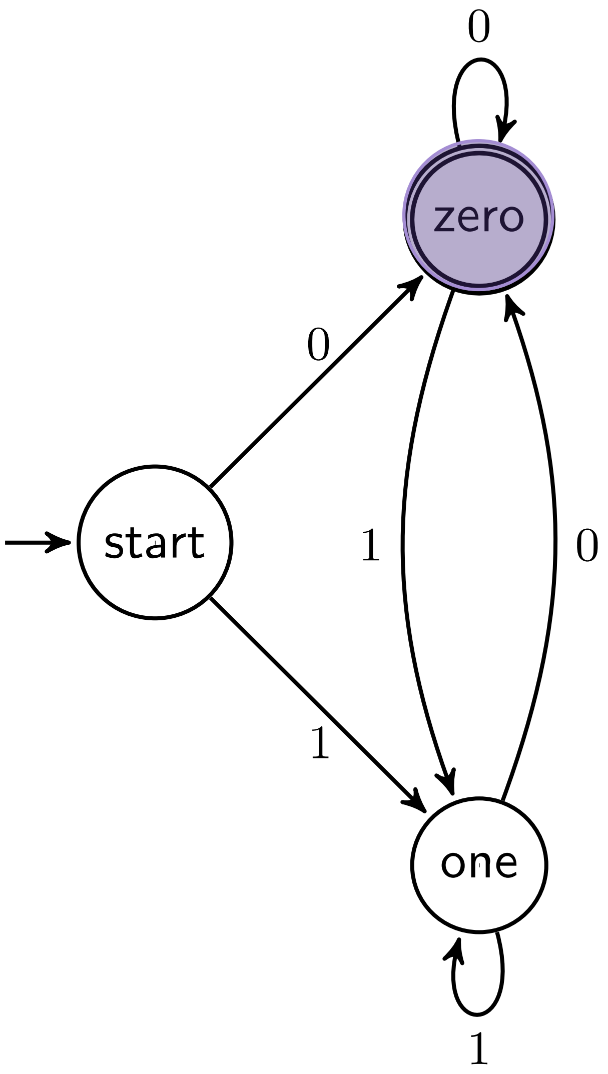


Let’s see an example

Input string:

`011`

`1010` accepted



Deterministic Finite Automata

Some more requirements:

Every machine is defined with respect to an alphabet Σ

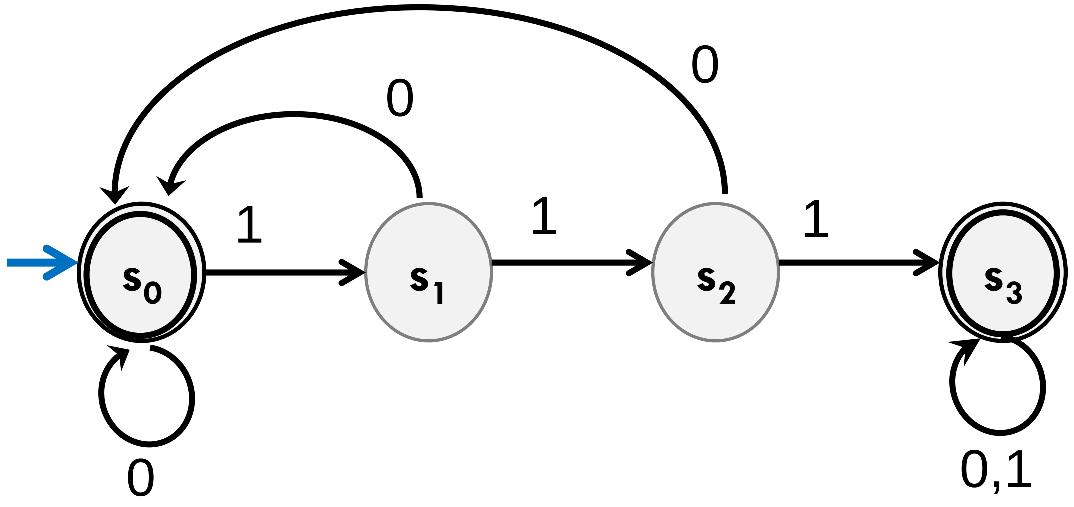
Every state has exactly one outgoing edge for every character in Σ.

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.

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

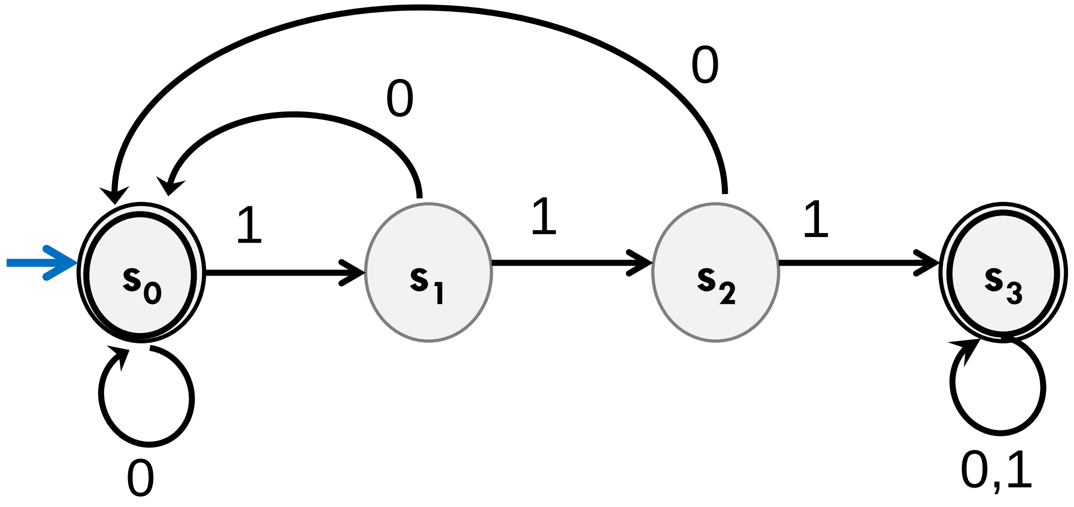


Deterministic Finite Automata

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

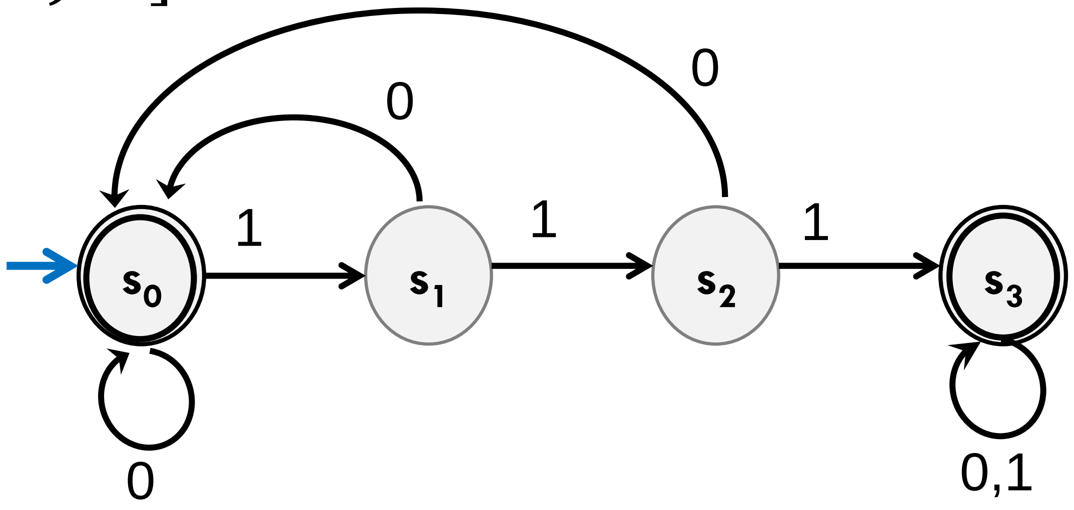
If the string has 111, then you'll end up in s_3 and never leave.

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

And…ε 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.

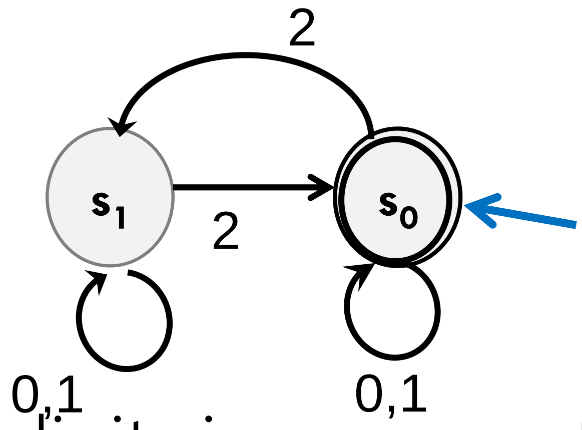
What do you need to remember?

M_2 should recognize “strings where the sum of the digits is congruent to $0\ (mod\ 3)$”

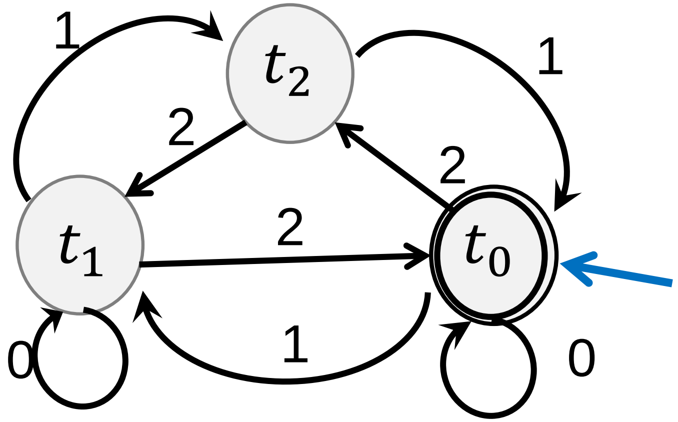
Design some DFAs

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 $0\ (mod\ 3)$"



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.