CS351 - Finite Automata

This handout will describe finite automata, a mechanism that can be used to construct regular languages. We'll describe regular languages in an upcoming set of lecture notes. We will study two types of finite automata:

Deterministic (DFA) – There is a fixed number of states and we can only be in one state at a time

Nondeterministic (NFA) –There is a fixed number of states but we can be in multiple states at one time

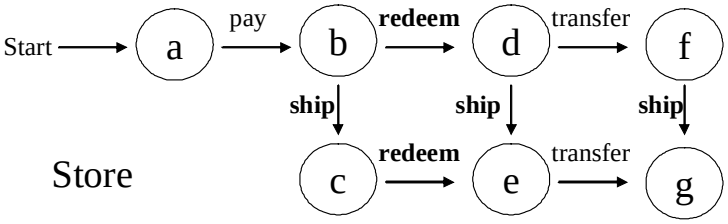
While NFA's are going to be more expressive than DFA's, we will see that adding nondeterminism does not let us define any language that cannot be defined by a DFA. One way to think of this is we might write a program using a NFA, but then when it is "compiled" we turn the NFA into an equivalent DFA.

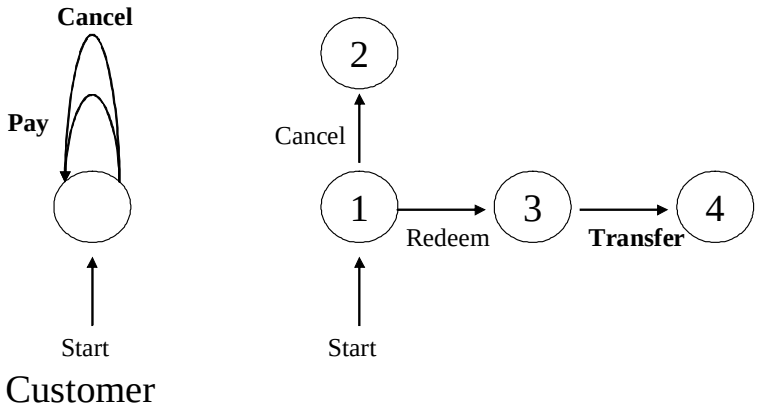
Informal Automata Example

Let's use the example from the textbook as an informal introduction to automata. Let's model a customer shopping at a store using some form of electronic money that can change hands at a bank

The customer may *pay* the e-money or *cancel* the e-money at any time.
The store may *ship* goods and *redeem* the electronic money with the bank.
The bank may *transfer* any redeemed money to a different party, say the store.

We can model the electronic money protocol through three automata, each from the perspective of the entities. The edges on the arcs from one state to another are actions that are initiated by one of the entities. Let's say that in this example, we would like the store to have the flexibility of shipping, redeeming, and transferring e-money in different orders.





In the automata, actions in bold are initiated by the entity. Otherwise, the actions are initiated by someone else and received by the specified automata.

First, consider the automata for the bank. The bank starts in state 1. It sits in state 1 until a cancel or redeem is received. The cancel will be issued from the customer, while the redeem is issued from a store. If a cancel is received, the bank restores the e-money to the customer's account, and sits in state 2, which does not allow anything else to happen with this money (i.e. the customer can't spend it later). If a redeem is received, we move to state 3 and then issue a transfer with new e-money now belonging to the store.

The customer's automata is simple; the customer may pay or cancel at any and in any order, and always stays in the lone state after each action. It will be up to the other entities to not allow money to be paid or cancelled multiple times.

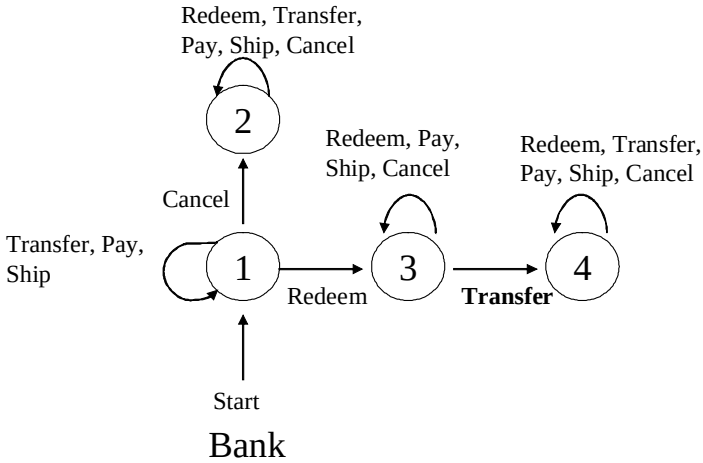
The store's system is set up to allow the shipping and financial operations to be separate processes. This allows the ship action to be done either before, after, or during the e-money transaction. Such flexibility could be useful for different types of purchases. Once a store receives the pay action from a customer, it may then redeem/ship/transfer, ship/redeem/transfer, or redeem/transfer/ship ultimately ending up in state g.

Ignoring Actions

The three automata have been set up to reflect the behaviors of interest. To be more precise, with a DFA (deterministic finite automaton) we should specify arcs for all possible inputs.

For example, what should the customer automaton do if it receives a "redeem"? What should the bank do if it is in state 2 and receives a "redeem"? These behaviors are unspecified currently.

The typical behavior if we receive an unspecified action is for the automaton to **die**. The automaton enters no state at all, and further action by the automaton would be ignored. The best method though is to specify a state for all behaviors, as indicated below for the bank automaton.

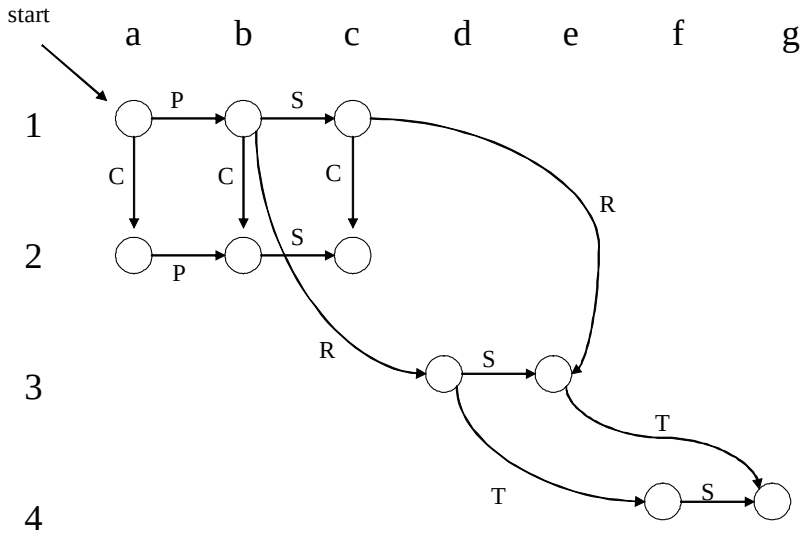


The Entire System as an Automaton

When there are multiple automata for a system, it is useful to incorporate all of the automata into a single one so that we can better understand the interaction. This can be accomplished by constructing the *product* automaton. The product automaton creates a new state for all possible states of each automaton.

Since the customer automaton only has one state, we only need to consider the pair of states between the bank and the store. For example, we start in state (a,1) where the store is in its start state, and the bank is in its start state. From there we can move to states (a,2) if the bank receives a cancel, or state (b,1) if the store receives a pay.

To construct the product automaton, we run the bank and store automaton "in parallel" using all possible inputs and creating an edge on the product automaton to the corresponding set of states. From the start state, the reachable states in the product automaton are shown below where the column and row labels indicate the state in the bank and store automata:



How is this useful? It can help validate our protocol. First, it tells us that not all states are reachable from the start state. For example, we should never be in state (g,1) where we have shipped and transferred cash, but the bank is still waiting for a redeem.

Second, the product automata allows us to see if potential errors can occur. From the automata we can see that we can reach state (c, 2). This is problematic because it allows a product to be shipped but the money has not been transferred to the store.

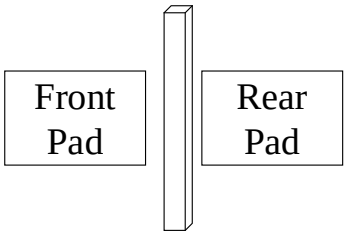
In contrast, we can see that if we reach state (d, 3) or (e, 3) then the store should be okay – a transfer from the bank must occur (assuming the bank automaton doesn't "die" which is why it is useful to add arcs for all possible inputs to complete the automaton).

Formal Definition of a Deterministic Finite Automaton (DFA)

1. Finite set of states, typically Q.
2. Alphabet of input symbols, typically Σ
3. One state is the start/initial state, typically q_0
4. Zero or more final/accepting states; the set is typically F.
5. A transition function, typically δ. This function:
   - Takes a state and input symbol as arguments.
   - Returns a state.
   - One "rule" of δ would be written δ(q, a) = p, where q and p are states, and a is an input symbol.
   - Intuitively: if the FA is in state q, and input a is received, then the FA goes to state p (note: q = p OK).
6. A FA is represented as the five-tuple: A = (Q, Σ, δ, q_0; F). Here, F is a set of accepting states.

Example : One-Way Automatic Door

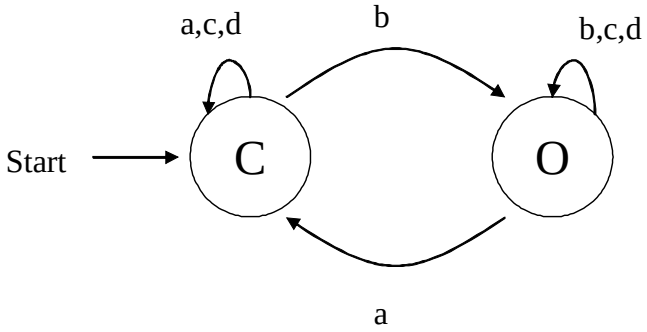
As an example, consider a one-way automatic door. This door has two pads that can sense when someone is standing on them, a front and rear pad. We want people to walk through the front and toward the rear, but not allow someone to walk the other direction:



Let's assign the following codes to our different input cases:

a - Nobody on either pad
b - Person on front pad
c - Person on rear pad
d - Person on front and rear pad

We can design the following automaton so that the door doesn't open if someone is still on the rear pad and hit them:



Using our formal notation, we have:

Q = {C, O} (usually we'll use q_0 and q_1 instead)
F = {} There is no final state
q_0 = C This is the start state
Σ = {a,b,c,d}
The transition function, δ, can be specified by the table:

	a	**b**	**c**	**c**
→ **C**	C	O	C	C
O	C	O	O	O

The start state is indicated with the →
If there are final accepting states, that is indicated with a * in the proper row.

Exercise:

Using Σ={0,1} a "clamping" circuit waits for a 1 input, and forever after makes a 1 output regardless of the input. However, to avoid clamping on spurious noise, design a DFA that waits for two 1's in a row, and "clamps" only then.

Write the transition function in table format as well as graph format.

Extension of δ to paths

Intuitively, a FA accepts a string w = $a_1a_2\ldots a_n$ if there is a path in the transition diagram that:
1. Begins at the start state,
2. Ends at an accepting state(s), and
3. Has sequence of labels $a_1a_2\ldots a_n$

Formally, we extend transition function δ to δ^(q, w), where w can be any string of input symbols. δ^ is called the extended transition function. We can define δ^ by induction as follows:

Basis: δ^(q,ε) = q (i.e., on no input, the FA doesn't go anywhere.)

Induction: $\delta\hat{}(q, wa) = \delta(\delta\hat{}(q,w),a)$, where w is a string, and a is a single symbol (i.e., see where the FA goes on w, then look for the transition on the last symbol from that state).

Important fact with a straightforward, inductive proof: $\delta\hat{}$ really represents paths. That is, if $w = a_1a_2...a_n$, and $\delta(p_i, a_i) = p_{i+1}$ for all i = 0,1, …, n-1, then $\delta\hat{}(p_0, w) = p_n$.

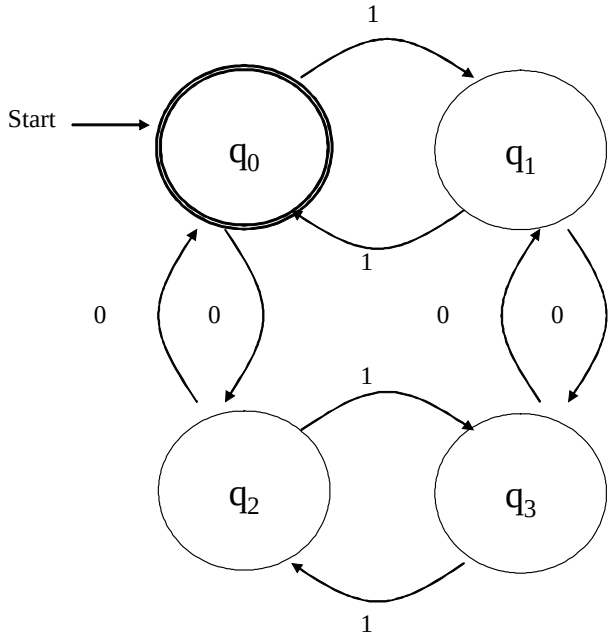
Acceptance of Strings

A finite automata $A = (Q, \Sigma, \delta, q_0, F)$ accepts string w if $\delta\hat{}(q_0, w)$ is in F.

Language of a Finite Automata

A finite automata A accepts the language $L(A) = \{w \mid \delta\hat{}(q_0, w) \text{ is in } F\}$
In other words, the language is all of those strings that are accepted by the finite automata.

For example, here is a DFA for the language that is the set of all strings of 0's and 1's whose numbers of 0's and 1's are both even:



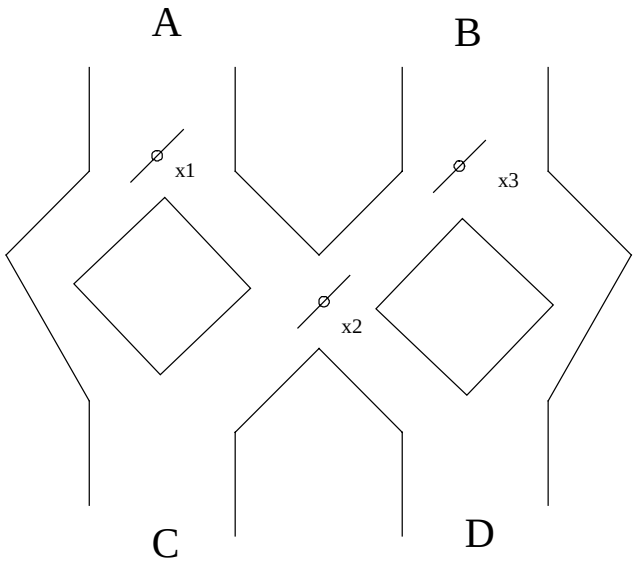
Aside : Type Errors

A major source of confusion when dealing with automata (or mathematics in general) is making "type errors."

- Don't confuse A, a FA, i.e., a program, with L(A), which is of type "set of strings."
- The start state q_0 is of type "state," but the accepting states F is of type "set of states."
- *a* could be a symbol or *a* could be a string of length 1 depending on the context

DFA Exercise: The figure below is a marble-rolling toy. A marble is dropped at A or B. Levers x1, x2, and x3 cause the marble to fall either to the left or to the right. Whenever a marble encounters a lever, it causes the lever to reverse after the marble passes, so the next marble will take the opposite branch.

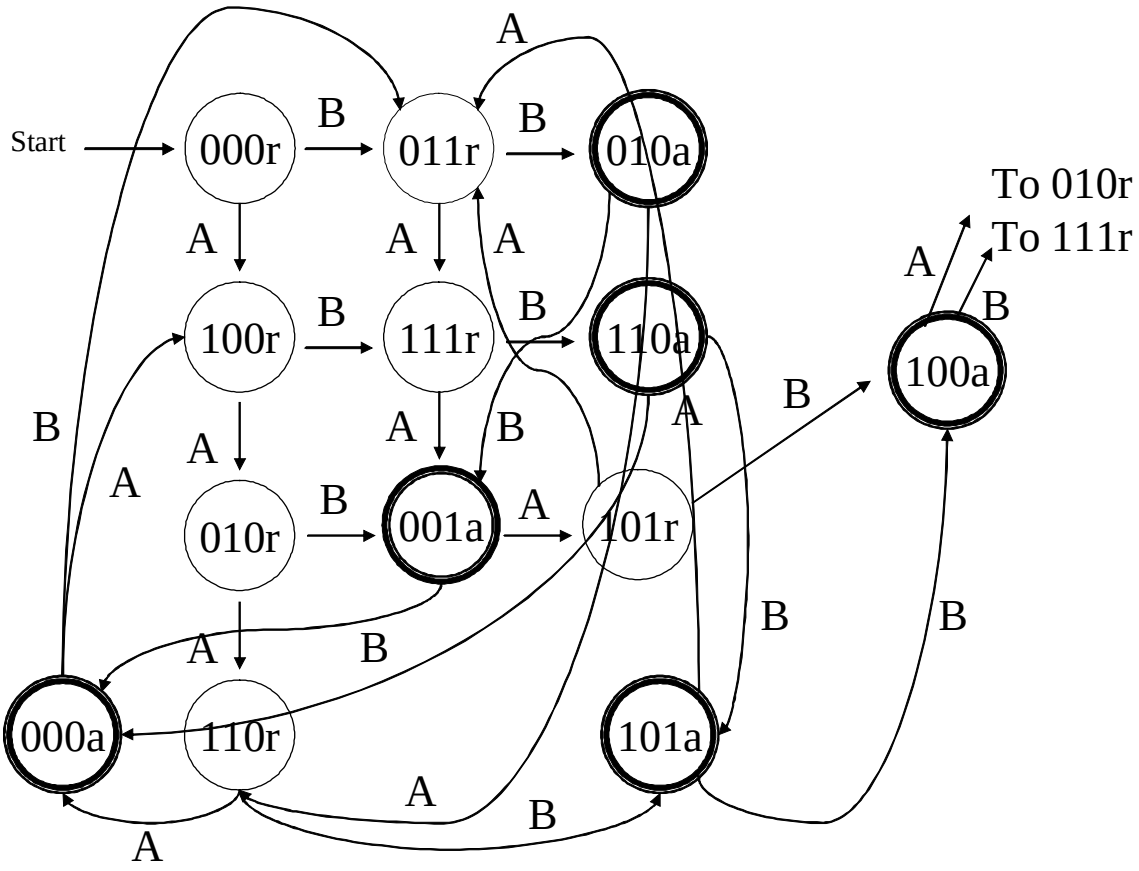
Model this game by a finite automaton. Let acceptance correspond to the marble exiting at D. Non-acceptance represents a marble exiting at C.



The key to solving this problem is to recognize that the inputs and outputs (A-D) become the alphabet of the automaton, while the levers indicate the possible states. If we define the initial status of each lever to be a 0, then if the levers change direction they are in state 1.

Let's use the format $x_1x_2x_3$ to indicate a state. The initial state is 000. If we drop a marble down B, then the state becomes to 011 and the marble exits at D.

Since we have three levers that can take on binary values, we have a total of 8 possible states, 000 to 111. We further identify the states by appending an "a" for acceptance, or "r" for rejection. This leads to a total of 16 possible states. All we need to do is start from the initial state and draw out the new states we are led to as we get inputs from A or B.



This may be a bit easier to view in table format. Note that not all of the 16 states are accessible.

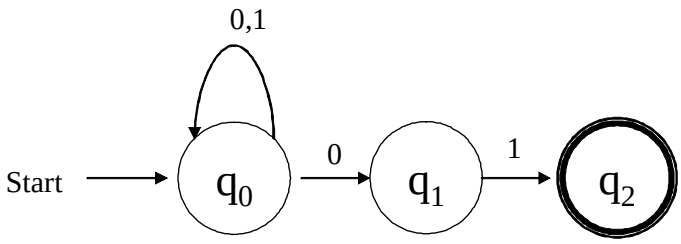
	A	B
->000r	100r	011r
*000a	100r	011r
*001a	101r	000a
010r	110r	001a
*010a	110r	001a
011r	111r	010a
100r	010r	111r
*100a	010r	111r
101r	011r	100a
*101a	011r	100a
110r	000a	101a
*110a	000a	101a
111r	001a	110a

Nondeterministic Finite Automata

A NFA (nondeterministic finite automata) is able to be in several states at once. A particular state of a DFA may accept only one destination state for a particular input. However, an NFA may accept multiple destination states for the same input. You can think of this as the NFA "guesses" something about its input and will always follow the proper path if that can lead to an accepting state.

Another way to think of the NFA is that it travels all possible paths, and so it remains in many states at once. As long as at least one of the paths results in an accepting state, the NFA accepts the input. This is a useful tool to have and is more expressive than a DFA. However, to actually implement the NFA, we must implement it deterministically. That is, we will see that for any NFA we can construct a corresponding DFA.

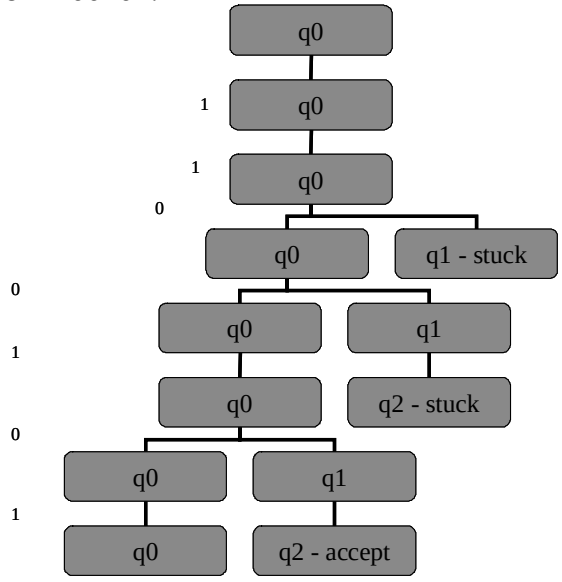
Consider the following NFA, whose job is to accept all and only the strings of 0's and 1's that end in 01:



Note that we haven't specified all possible inputs for each state. If a state receives an unspecified input, the automaton "dies."

Also, note that from state q_0 we have two possible states if the input is 0. We can go back to state q_0 or we can move to state q_1.

The way to read this automaton is that we can enter multiple states whenever possible. You can think of this as parallel "threads" of execution. The tree below shows what happens for an input of 1100101:



Formal Definition of an NFA

An NFA is defined similarly to a DFA:

1. Finite set of states, typically Q.
2. Alphabet of input symbols, typically Σ
3. One state is the start/initial state, typically q_0
4. Zero or more final/accepting states; the set is typically F.
5. A transition function, typically δ. This function:
   - Takes a state and input symbol as arguments.
   - Returns a **set of states** instead of a single state, as a DFA
6. A FA is represented as the five-tuple: A = ($Q, \Sigma, \delta, q_0, F$). Here, F is a set of accepting states.

The previous NFA could be specified formally as:

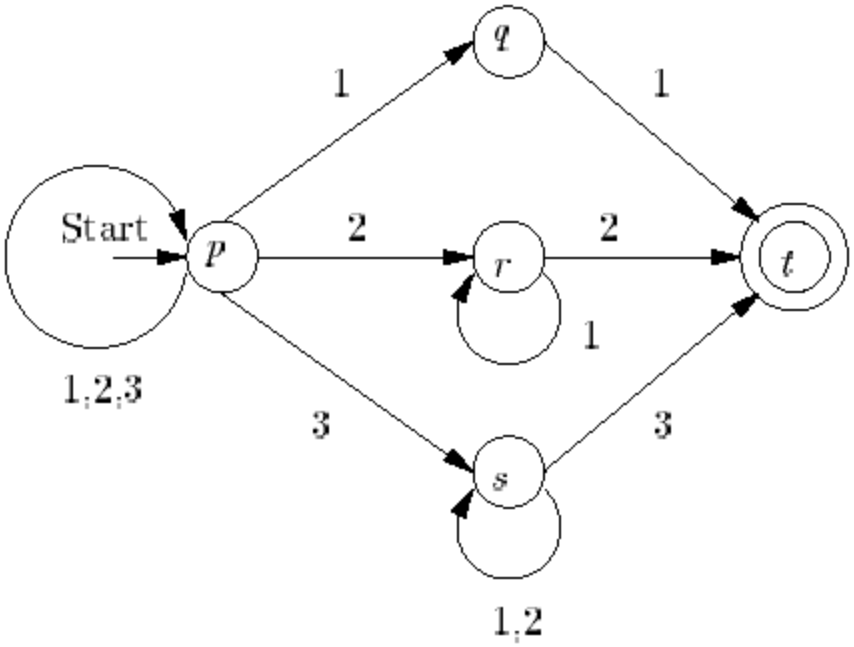
($\{q_0,q_1,q_2\}$, $\{0,1\}$, δ, q_0, $\{q_2\}$)

The transition table is:

	0	**1**
→ **q0**	{q0,q1}	{q0}
q1	**Ø**	{q2}
***q2**	**Ø**	**Ø**

Example: Here is an NFA that will accept strings over alphabet {1, 2, 3} such that the last symbol appears at least twice, but without any intervening higher symbol, in between:
e.g., 11, 2112, 123113, 3212113, etc.

Trick: use start state to mean "I guess I haven't seen the symbol that matches the ending symbol yet." Use three other states to represent a guess that the matching symbol has been seen, and remembers what that symbol is.



What is the transition table for this NFA?

The Extended Transition Function

Just as we had an extended transition function for a DFA, we can have the same thing for an NFA and extend it to support strings. The difference from the extended transition function for the DFA is that we can be in a set of states instead of a specific, deterministic state upon processing some string.

For example, consider the tree for the NFA we constructed earlier that recognizes strings that end in 01. After processing 1100, we can be in one of two states: q_0 or q_1. With a DFA, we would only be in one state.

Formally, we define $\delta\hat{}$, the extended transition function for an NFA, inductively as follows:

Basis: $\delta\hat{}(q,\varepsilon) = \{q\}$. That is, without reading any input symbols, we are in the same set of states we begin in.

Induction: Let

- $\delta\hat{}(q,w) = \{p_1, p_2, \ldots p_k\}$
- $\delta(p_i,a) = S_i$ for i=1,2,…k

Then $\delta\hat{}(q, wa) = S_1 \cup S_2 \cup \ldots \cup S_k$

Less formally, we compute $\delta\hat{}(q,w)$ by first computing $\delta\hat{}(q,x)$ and then follow any transition from one of these states to complete w, where w = xa, where a is a single symbol in the alphabet.

For example, returning to the NFA that accepts strings ending in 01:

$\delta\hat{}(q_0, \varepsilon) = \{ q_0 \}$
$\delta\hat{}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
$\delta\hat{}(q_0, 00) = \delta(q_0,0) \cup \delta(q_1, 0) = \{q_0, q_1\}$
$\delta\hat{}(q_0,001) = \delta(q_0,1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2 \}$

Language of an NFA

An NFA accepts w if any path from the start state to an accepting state is labeled w. Formally:

$$L(N) = \{ w \mid \delta\hat{}(q_0, w) \cap F \neq \emptyset \}$$

That is, for an NFA N, L(N) is the set of strings w in Σ^* such that $\delta\hat{}(q_0, w)$ contains at least one accepting state.

The language for the example we have been using is informally:

$$L = \{w \mid w \text{ ends in } 01 \}$$

Equivalence of DFA and NFA

Although there are many languages for which an NFA is easier to construct than a DFA, it is perhaps a surprising fact that every language that can be described by some NFA can also be described by some DFA. That is, we can construct a DFA that accepts the same strings as a NFA. The downside is there may be up to 2^n states in the turning a NFA into a DFA. However, for most problems the number of states is approximately equivalent.

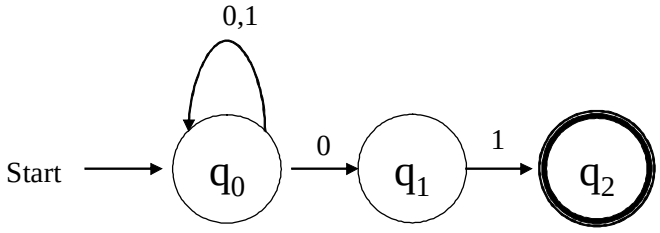
The process of turning an NFA into a DFA is called *subset construction* because it involves constructing up to all of the set of states of the NFA.

Let an NFA N be defined as $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$.
The equivalent DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ where:

1. $Q_D = 2^{Qn}$; i.e. Q_D is the set of all subsets of Q_N; that is, it is the power set of Q_N. Often, not all of these states are accessible from the start state; these states may be "thrown away."
2. F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N's states that include at least one accepting state of N.
3. For each set $S \subseteq Q_N$ and for each input symbol a in Σ:
$$\delta_D(S,a) = \bigcup_{p \in S} \delta_N(p,a)$$

That is, to compute $\delta_D(S, a)$ we look at all the states p in S, see what states N goes to starting from p on input a, and take the union of all those states.

Consider once again the NFA that accepts strings ending in 01:



The power set of these states is: { Ø, $\{q_0\}$, $\{q_1\}$, $\{q_2\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$,$\{q_1, q_2\}$, $\{q_0, q_1, q_2\}$ }

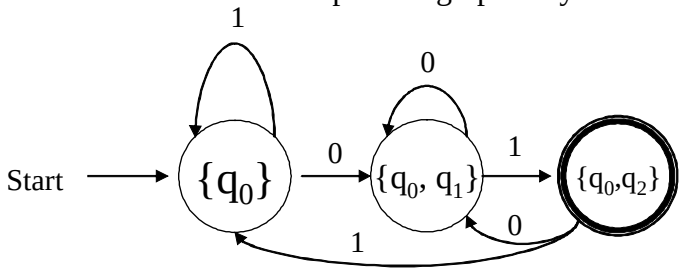
We construct a new transition function with all of these states and go to the set of possible inputs:

	0	**1**
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	Ø	$\{q_2\}$
*$\{q_2\}$	Ø	Ø
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
*$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
*$\{q_1, q_2\}$	Ø	$\{q_2\}$
*$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

The start state is $\{q_0\}$ and the final states are any of those that contain q_2.
Note that many of these states are unreachable from our start state. A good way to construct the equivalent DFA from an NFA is to start with the start states and construct new states on the fly as we reach them:

	0	**1**
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
*$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

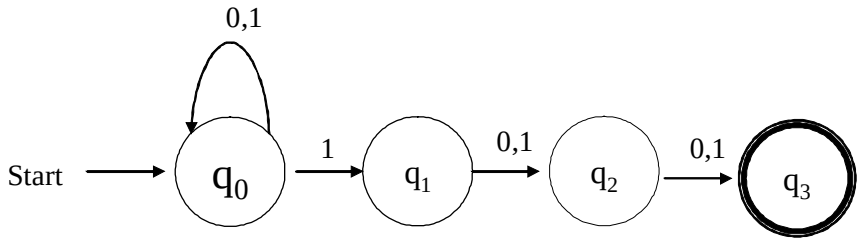
This new transition function can be expressed graphically as:



Theorem: L(D) = L(N) for an appropriately constructed DFA from an NFA.
See the book for the inductive proof.

Theorem: A language L is accepted by some DFA if and only if L is accepted by some NFA.
Informal Proof: The "if" part is the subset construction proof from the theorem above. The "only if" part is proved since it is trivial to turn a DFA into an NFA (in effect there is nothing to do since it is valid to have an NFA with a deterministic set of states).

A bad case for subset construction: We can see that a bad case for subset construction is when we can actually reach all (or most) of the 2^n possible subsets of states. Here is one example, a NFA that accepts any string of 0's and 1's such that the nth symbol from the end is 1. In this case, we have n=3:



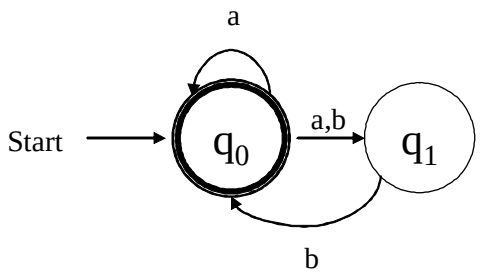
For the string that ends in 100, 101, 110, or 111 the NFA stays in the start state until we reach one of these suffixes and proceed to the final state at q_3. Any other input will "die" out at q_3 when we receive further input.

Here is a table representing the conversion to a DFA:

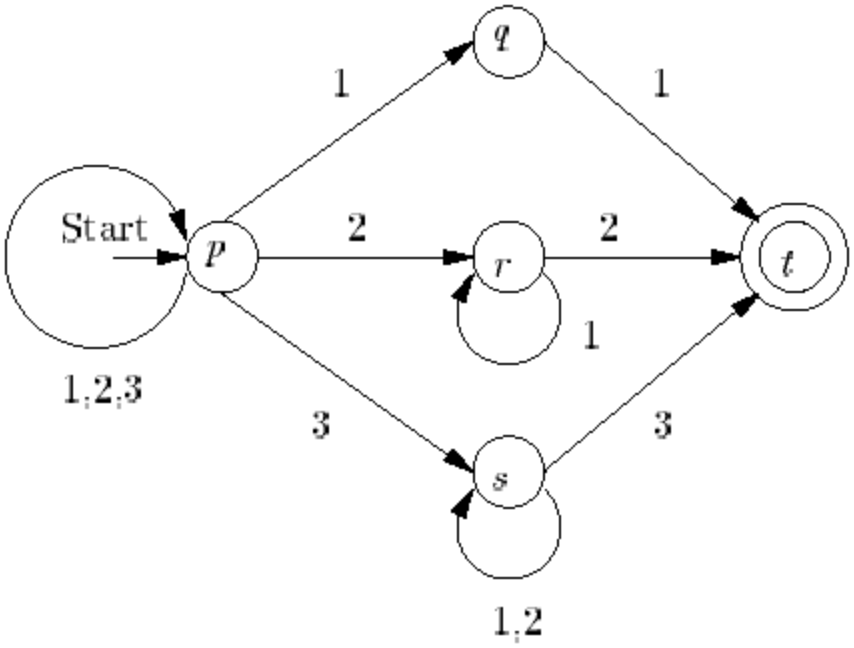
	0	**1**
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$
*$\{q_0, q_3\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$
*$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$
*$\{q_0, q_2, q_3\}$	$\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$
*$\{q_0, q_1, q_3\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$

In this case, we have 2^n states (2^3 in this example, or 8 states).

Exercise: Convert the following NFA to a DFA.



Exercise: Convert the following NFA into a DFA (this is the NFA that recognizes strings with smaller numbers between the last value and an earlier value)



There should be 15 possible states (might be easier to represent in table format)

Finite Automate with Epsilon Transitions

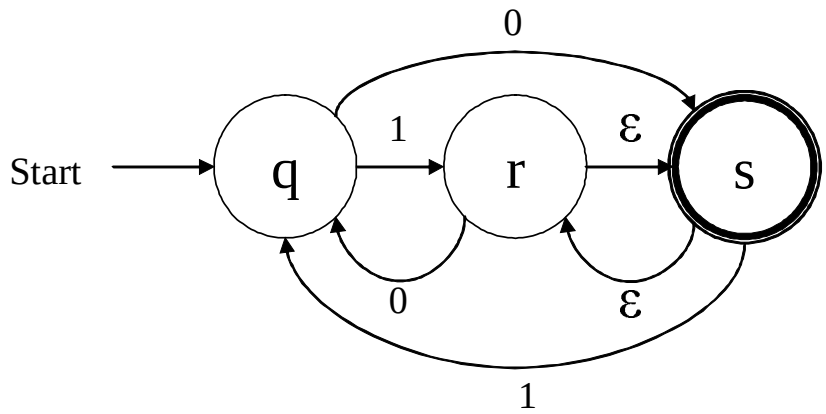
We can extend an NFA by introducing a "feature" that allows us to make a transition on ε, the empty string. All the ε transition lets us do is spontaneously make a transition, without receiving an input symbol. This is another mechanism that allows our NFA to be in multiple states at once. Whenever we take an ε edge, we must fork off a new "thread" for the NFA starting in the destination state.

Just as nondeterminism made NFA's more convenient to represent some problems than DFA's but were not more powerful, the same applies to ε-NFA's. While more expressive, anything we can represent with an ε-NFA we can represent with a DFA that has no ε-transitions.

The formal notation of an ε-NFA is the same as a regular NFA A = {Q, ∑, δ, q_0, F) except that the transition function δ is now a function that takes as arguments:

1. A state in Q and
2. A member of ∑ ∪ {ε}; that is, an input symbol or the symbol ε. We require that ε not be a symbol of the alphabet ∑ to avoid any confusion.

Example:



In this ε-NFA, the string "001" is accepted by the path qsrqrs, where the first qs matches 0, sr matches ε, rq matches 0, qr matches 1, and then rs matches ε. In other words, the accepted string is 0ε0ε1ε.

Epsilon Closure

Epsilon closure of a state is simply the set of all states we can reach by following the transition function from the given state that are labeled ε. Generally speaking, a collection of objects is *closed* under some operation if applying that operation to members of the collection returns an object still in the collection.

In the above example:

ε-closure(q) = { q }
ε-closure(r) = { r, s}

Epsilon closure lets us define the extended transition function for an ε-NFA. For a regular, NFA we said for the induction step:

Let

- $\delta\hat{}(q,w) = \{p_1, p_2, \ldots p_k\}$
- $\delta(p_i,a) = S_i$ for i=1,2,…k

Then $\delta\hat{}(q, wa) = S_1 \cup S_2 \cup \ldots \cup S_k$

For an ε-NFA, we change for $\delta\hat{}(q, wa)$:
Union[ε-closure(Each state in $S_1, S_2, \ldots S_k$)]

This includes the original set $S_1 \cup S_2 \cup \ldots \cup S_k$ as well as any states we can reach via ε.

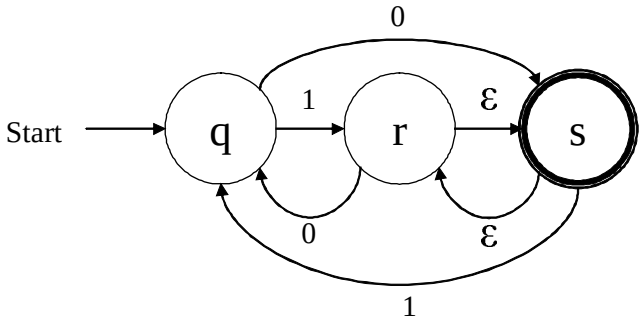
When coupled with the basis that $\delta\hat{}(q, \varepsilon)$ = ε-closure(q) lets us inductively define an extended transition function for a ε-NFA.

Eliminating ε-Transitions

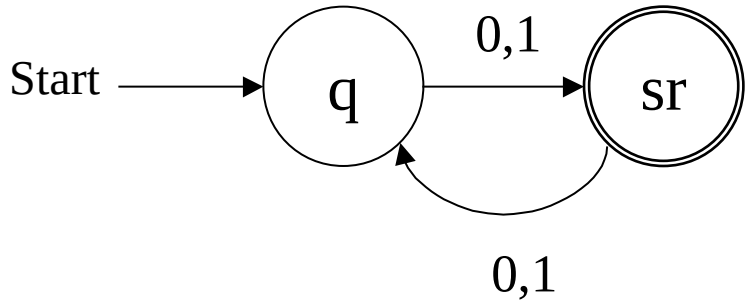
ε-Transitions are a convenience in some cases, but do not increase the power of the NFA. To eliminate them we can convert a ε-NFA into an equivalent DFA, which is quite similar to the steps we took for converting a normal NFA to a DFA, except we must now follow all ε-Transitions and add those to our set of states.

1. Compute ε-closure for the current state, resulting in a set of states S.
2. $\delta_D(S,a)$ is computed for all a in Σ by
    a. Let $S = \{p_1, p_2, \ldots p_k\}$
    b. Compute $\bigcup_{i=1}^{k} \delta(p_i, a)$ and call this set $\{r_1, r_2, r_3 \ldots r_m\}$ This set is achieved by following input a, not by following any ε-transitions
    c. Add the ε-transitions in by computing $\delta(S,a) = \bigcup_{i=1}^{m} \varepsilon - closure(r_i)$
3. Make a state an accepting state if it includes any final states in the ε-NFA.

Example



Converts to:



In this case the DFA is actually a bit simpler!

Exercise: Design an ε-NFA for the language consisting of zero or more a's followed by zero or more b's followed by zero or more c's.