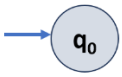
Department of Computer Science and Engineering
PES University, Bangalore, India

UE23CS242A: Automata Formal Language and Logic

Prakash C O, Associate Professor, Department of CSE

FSM Components

- **Input** - A string fed to a machine which the machine will determine whether it is part of the language that the machine was designed for. The string must only be made up of symbols from the machine's alphabet. An input is read by a machine in a forward fashion, one symbol at a time.
- **Return** - The results of running the machine on a given input. Initially, this will either be **accepted** or **rejected**, indicating whether the input string is respectively part of the language or not.
- **State** - A resting place while the machine reads more input, if more input is available. States are typically named. Canonical names for states (which we will get to later) commonly consist of the lowercase letter 'q' followed by a number, e.g. q_0. State names must be unique. Graphically, states are represented by a circle with the name inside.
- **Start State** - For programmers, this is known as the *program entry point*. It is the state that the machine naturally starts in before it reads any input. The name for the start state will usually either be *S* or, canonically, q_0 (it is numbered with the lowest number of all the states). Graphically, the start state is represented as a state with an unlabeled arrow pointing from nothing to it.



- **Accepting State** or **Final State** - **A set of states which the machine may halt in, provided it has no input left, in order to accept the string as part of the language**. Graphically, accepting states are indicated distinctly from other states using a double circle instead of a single.

It is worth noting that a machine with no accepting states does not accept any string as part of the language (not even the empty string) - it is essentially the Empty Language, ∅.

- **Rejecting State** - **Any state in the machine which is not denoted as an accepting state**. The string is only rejected if the machine *halts* in a rejecting state with no more input left. Rejecting states have no special representation, because **every state is either accepting or rejecting.**

- **Dead State** - A rejecting state that is essentially a dead end. Once the machine enters a dead state, there is no way for it to reach an accepting state, so we already know that the string is going to be rejected.

  A machine may have multiple dead states, but at most only one dead state is needed per machine.

- **Transition** - A way for a machine to go from one state to another given a symbol from the input. Graphically, **a transition is represented as an arrow pointing from one state to another**, **labeled with the symbol or symbols that it can read** in order to move the machine from the state at the tail of the arrow to the tip of the arrow.

  Transitions may even point back to the same state that they came from, which is called a self **loop**.

Using this terminology, we can logically deduce that a **Finite State Automaton** or **Finite State Machine** (FSM) is a machine that has a finite number of states.

Why NFA?

DFA may be cumbersome to specify for complex systems and may even be unknown.
NFA provides a nice way of 'abstraction of information' - a concept applied in so many aspects of computing.

NFA are often more convenient to design than DFA. Examples below can be easily designed using NFA than DFA.

1. Strings containing 1 in the third last position.
2. Strings accepting multiples of 2 or 3. (General union)

Why Use NFAs?

Reason	Explanation
Conciseness	NFAs often require fewer states than DFAs for the same language. They're more compact and easier to construct, especially for complex patterns.
Expressive Construction	Easier to build from regular expressions or for language composition operations (like union, concatenation, Kleene star).
Conversion is Always Possible	Every NFA can be converted to an equivalent DFA — so NFAs serve as a flexible design layer before optimization.
Useful in Parsing & Pattern Matching	Tools like regex engines often build NFAs first, then optimize to DFA-like representations for execution speed.

Nondeterministic finite automaton (NFA)

Formal Definition of an NFA

Formally, an NFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$

where:

- Q is a finite set of states
- Σ is the input alphabet (symbols which are part of input alphabet)
- δ is a transition function which maps $Q \times \Sigma \rightarrow 2^Q$
- q_0 is the initial (or start) state
- F is the set of accepting (or final) states ($F \subseteq Q$). If any state of F is reached, input string is accepted.

In automata theory, a finite-state machine is called a deterministic finite automaton (DFA), if

- each of its transitions is *uniquely* determined by its source state and input symbol, and
- reading an input symbol is required for each state transition.

A nondeterministic finite automaton (NFA), does not need to obey these restrictions. In particular, every DFA is also an NFA.

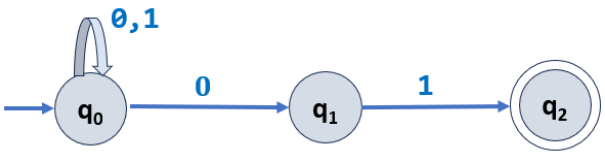
Note:

1. NFA provides a nice way of 'abstraction of information'. **In an NFA, a (state, symbol) combination may lead to several states simultaneously.**
2. **DFAs have been generalized to nondeterministic finite automata (NFA) which may have several arrows of the same label starting from a state.**
3. **Using subset construction method, every NFA can be translated to a DFA that recognizes the same language. DFAs, and NFAs as well, recognize exactly the set of regular languages.**

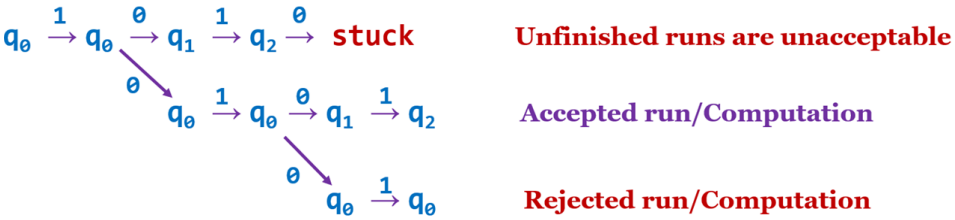
Computation or Run of an NFA

- **For an input string, NFA will create a 'computation tree' rather than a 'computation sequence' in case of DFA.**
- An NFA accepts a string if any one of the paths in the tree leads to some accept state.
- Nondeterminism causes multiple runs. There can be several or no runs for a given input word. If there is at least one run that leads to an accepting state, the machine will accept the input string.

Example 1: Consider L = { x ∈ {0, 1}* | where x ends with 01}



Let 10101 be an input word. The following are potential runs.



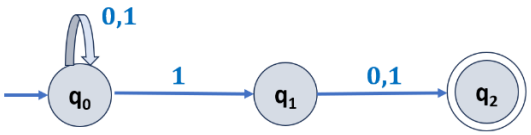
If the word has 01 at the end, there exists an accepting run!

Exercise 1:

Give runs for word 1110.

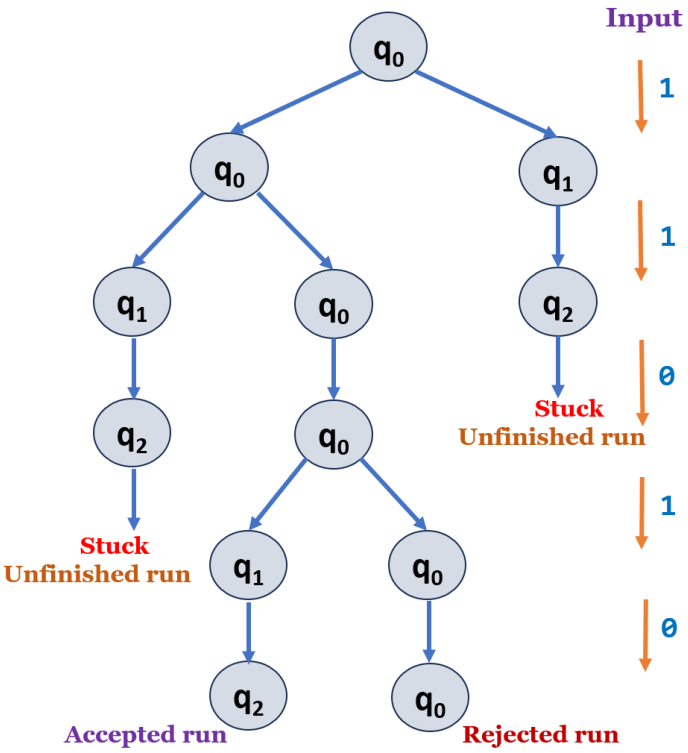
Example 2:

Consider L = {x ∈ {0, 1}* | 2nd symbol from the right is 1}



In the above NFA, the movement from state q_0 when input is '1' is nondeterministic.

Computation tree for the input string 11010

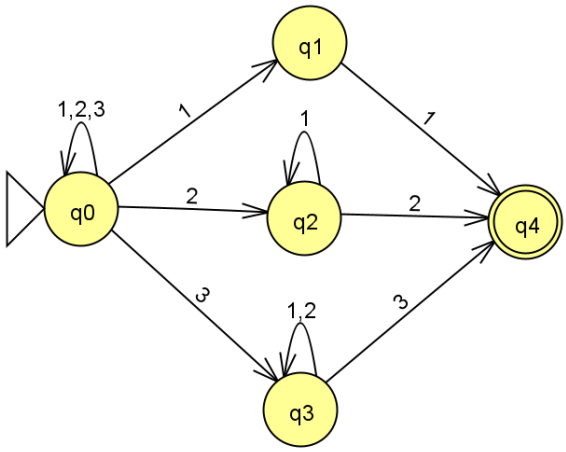


The NFA consumes a string of input symbols, one by one. In each step, whenever two or more transitions are applicable, it "clones" itself into appropriately many copies, each one following a different transition.

If no transition is applicable, the current copy is in a dead end (or stuck), it "dies". If, after consuming the complete input, any of the copies is in an accept state, the input is accepted, else, it is rejected.

Example 3:

Construct 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.



Equivalence of DFA and NFA

- **Surprisingly, for any NFA N there is a DFA D, such that L(D) = L(N), and vice versa.** Given an NFA N we will construct a DFA D such that L(D) = L(N)
- NFAs and DFAs are equivalent in that if a language is recognized by an NFA, it is also recognized by a DFA and vice versa. The establishment of such equivalence is important and useful.

- It is useful because constructing an NFA to recognize a given language is sometimes much easier than constructing a DFA for that language.
- It is important because NFAs can be used to reduce the complexity of the mathematical work required to establish many important properties in the theory of computation.
  For example, it is much easier to prove closure properties of regular languages using NFAs than DFAs.

Note:

- **NFAs have been generalized in multiple ways**, e.g., nondeterministic finite automata with ε-moves, finite-state transducers, pushdown automata, alternating automata, ω-automata, and probabilistic automata.
- Every NFA has an equivalent DFA (accepting the same language).
- The languages accepted by Finite State Machines (DFA, NFA, **ε-NFA**) are referred to as Regular languages.

NFA with ε-transitions (or λ-transitions)

Nondeterministic finite automaton with ε-moves (ε-NFA) is a further generalization to NFA. In this kind of automaton, the transition function is additionally defined on the empty string ε. A transition without consuming an input symbol is called an ε-transition.

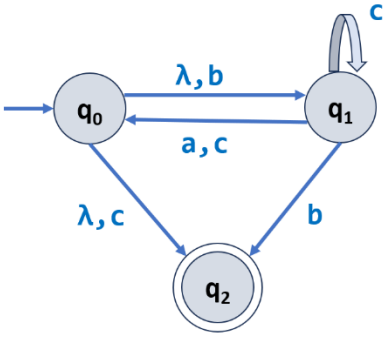
Let us add another feature to our automaton.

Let us allow it to jump states without reading inputs. We will call such moves ε-transitions (or λ-transitions).

ε-closure (or λ-closure) of a state or set of states

For a state q, let ε-closure(q) (or λ-closure(q)) denote the set of states that are reachable from q by ε-transitions (or λ-transitions) including the state q itself.

Example:



λ-closure(q_0) = {q_0 q_1 q_2}

λ-closure(q_1) = { q_1 }

λ-closure(q_2) = { q_2 }

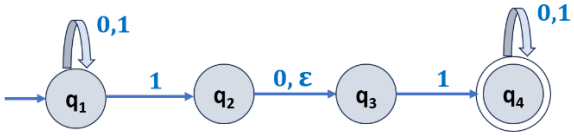
Formal Definition of an ε-NFA (or λ-NFA)

ε-NFA can be represented by a 5-tuple M = (Q, ∑, δ, q_0, F) where

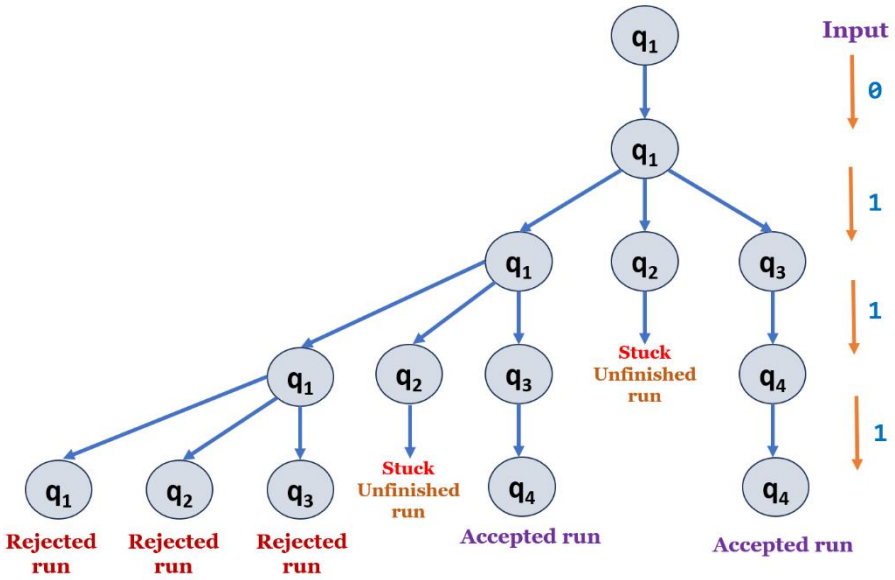
- Q is a finite set of states.
- ∑ is a finite non-empty set of input symbols called the alphabet.
- δ is the transition function where δ: Q × (**Σ** ∪ {ε}) → 2^Q or δ: Q × (**Σ** ∪ {λ}) → 2^Q
- q_0 is the initial state from where any input is processed (q_0 ∈ Q).
- F is a set of final state/states (F ⊆ Q).

Computation or Run of an ε-NFA

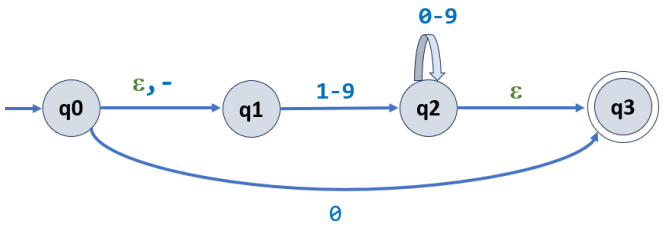
Example 1: Consider L = { x ∈ {0, 1}* | where x contains the substring 101 or 11}



Computation tree for the input string 0111



Example 1: Consider the following automaton with ε -transitions that recognizes integers without leading zeros.

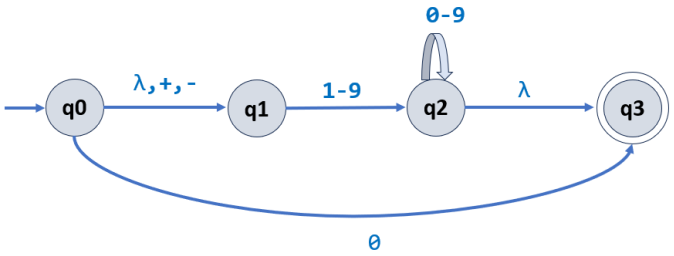


Note:

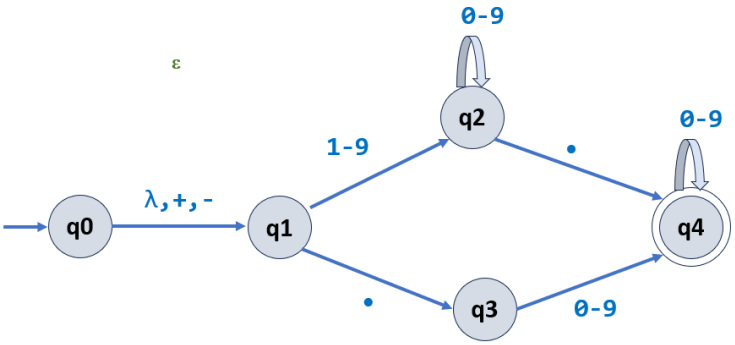
- '-' is used to specify range of symbols.
- 0-9 means {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Construct ε-NFA for

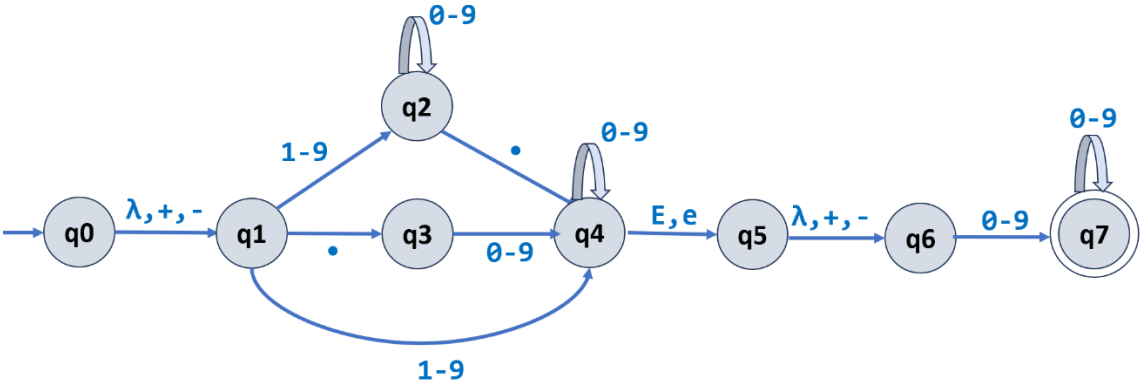
- L1 is the set of all strings that are decimal integer numbers. Specifically, L1 consists of strings that start with an optional sign, followed by one or more digits.
  Examples of strings in L1 are “22”, “+9”, and “-241”.



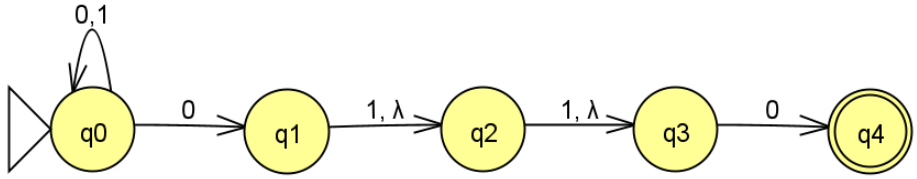
- L2 is the set of all strings that are floating-point numbers that are not in exponential notation. Specifically, L2 consists of strings that start with an optional sign, followed by zero or more digits, followed by a decimal point, and end with zero or more digits, where there must be at least one digit in the string.
  Examples of strings in L2 are “13.231”, “-28.” and “.124”. All strings in L2 have exactly one decimal point.
  Σ = {0,1,2,3,4,5,6,7,8,9, . , +, - }



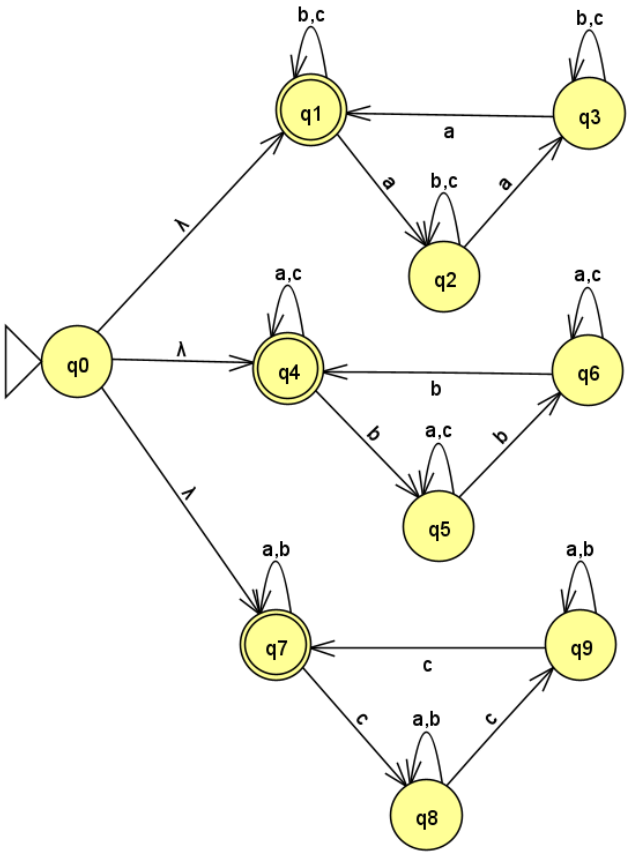
- L3 is the set of all strings that are floating-point numbers in exponential notation. Specifically, L3 consists of strings that start with a string from L1 or L2, followed by “E” or “e”, and end with a string from L1.
  Examples of strings in L3 are “-80.1E-083”, “+8.E5” and “1e+31”.



- An NFA for the language of all strings over {0, 1} that end with one of 0110, 010, and 00.



- An NFA for the language of all strings over {a, b, c} for which at least one of the number of occurrences of a or b or c is a multiple of 3.



Converting an NFA to a DFA

Given: A non-deterministic finite state machine (NFA)

Goal: Convert to an equivalent deterministic finite state machine (DFA)

Why?

- Faster recognizer!

IDEA: Each state in the DFA will correspond to a set of NFA states.

Worst-case:

- There can be an exponential number $\mathbf{O}(2^N)$ of sets of states.
- **The DFA can have exponentially many more states than the NFA... but this is rare.**

The subset construction method

The Subset Construction Method (**also known as the powerset construction**) is the standard algorithm for converting a Nondeterministic Finite Automaton (NFA) into an equivalent Deterministic Finite Automaton (DFA).

The idea behind subset construction method:

- Given an NFA N = (Q, Σ, δ, q0, F) we construct the equivalent DFA D:
- The basic idea of this construction is to define a DFA whose states are sets of NFA states.
- A set of possible NFA states thus becomes a single DFA state.

- The DFA transition function is given by considering all reachable NFA states for each of the current possible NFA states for each input symbol.
- The resulting set of possible NFA states is again just a single DFA state.
- A DFA state is final if that set contains at least one final NFA state.

The subset construction method (for NFA to DFA) follows the following steps:

Convert the given NFA transition diagram to its equivalent DFA transition table by following the below steps.

Step 1: Make the start state of NFA as the start state of DFA.

Step 2: Find the transition of the start state to another set of states on a set of input symbols.

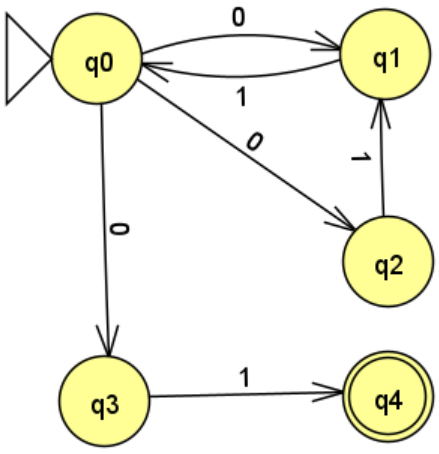
Step 3: Add the newly emerged state as a state of DFA and find the set of states which the current state transit on the input alphabets.

Step 4: This process is repeated until no new states emerge.

Step 5: Make the state that contains any of the final states of NFA as the final state of its equivalent DFA.

Examples:

1. Convert the following NFA to DFA using subset construction method.



Solution:

Given NFA Transition table:

δ	0	1
→q_0	q_1q_2 q_3	Φ
q_1	Φ	q_0
q_2	Φ	q_1
q_3	Φ	q_4
*q_4	Φ	Φ

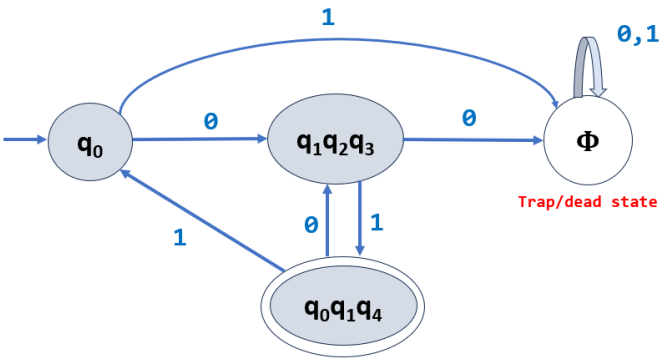
Note:

1) Φ in transition table represents empty set and the corresponding Φ in FSM represents Trap state.
2) Φ in formal languages represents empty language.

Transition table for DFA:

δ	0	1
→q_0	{q_1 q_2 q_3}	Φ
{q_1 q_2 q_3}	Φ	{q_0 q_1 q_4}
*{q_0 q_1 q_4}	{q_1 q_2 q_3}	q_0

DFA State Transition Diagram:

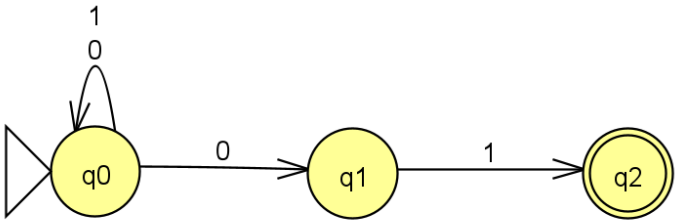


Note:

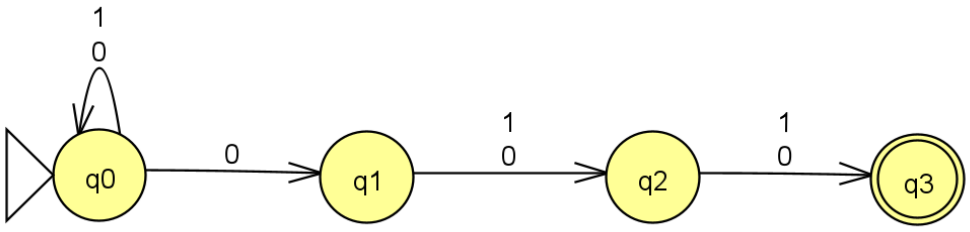
- A state in a DFA will be a subset of the set of states of the equivalent NFA. So, the maximum number of states in the equivalent DFA of an NFA, will be 2^n, where n is the number of states in NFA, as a set with n items has maximum 2^n subsets.

Exercise:

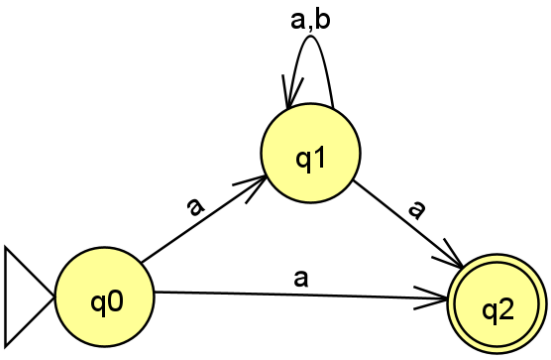
1. Convert the following NFA to DFA using subset construction method.



2. Convert the following NFA to DFA using subset construction method.



3. Convert the following NFA to DFA using subset construction method.



Converting an ε-NFA (or λ -NFA) to a DFA

The subset construction method (for ε-NFA to DFA) follows the following steps:

Convert the given ε-NFA (or λ-NFA) transition diagram to its equivalent DFA transition table by following the below steps.

Step 1: Start with ε-closure of the start state of ε-NFA

- Compute the ε-closure of q_0: all states reachable from q0 via ε-transitions.
- This becomes the start state of the DFA.

Step 2: Define DFA states as sets of NFA states

- Each DFA state represents a set of ε-NFA states.
- These sets are formed by tracking all possible states the ε-NFA could be in after reading each input symbol.

Step 3: For each DFA state and input symbol

- For each symbol a ∈ Σ, compute:
  - The set of ε-NFA states reachable from any state in the current DFA state on input symbol a
  - Then take the ε-closure of that set
- This new set becomes a new DFA state (if not already created)

Step 4: Repeat until no new states are added

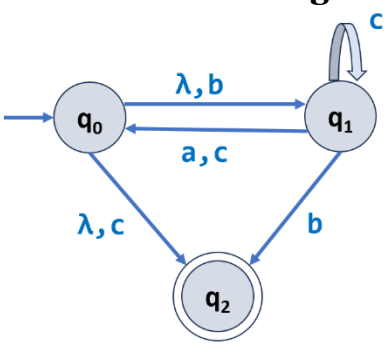
- Continue expanding transitions until all reachable sets of ε-NFA states are accounted for.

Step 5: Define DFA accept states

- Any DFA state that includes at least one ε-NFA accept state becomes a DFA accept state.

Examples:

1. Convert the following λ-NFA to DFA using subset construction method.



Solution:

λ-NFA Transition table:

δ	a	b	c
→q_0	Φ	q_1	q_2
q_1	q_0	q_2	{q_0 q_1 }
q_2*	Φ	Φ	Φ

	λ-closure
q_0	{q_0 q_1 q_2}
q_1	{ q_1 }
q_2	{ q_2 }

DFA Transition table:

The start state of DFA is the λ-closure of the start state of λ-NFA.

λ-closure(q_0) = {q_0 q_1 q_2}

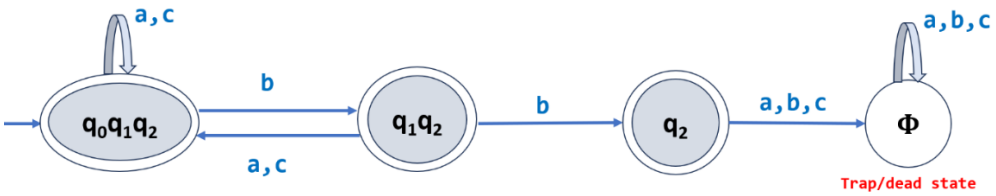
δ	a	b	c
→{q_0 q_1 q_2}*	δ({q_0 q_1 q_2}, a) = q_0 λ-closure(q_0) = {q_0 q_1 q_2}	δ({q_0 q_1 q_2}, b) = {q_1 q_2} λ-closure({q_1 q_2}) = { q_1 q_2} ←new state	δ({q_0 q_1 q_2}, c) = {q_0 q_1 q_2} λ-closure({q_0 q_1 q_2}) = {q_0 q_1 q_2}
{q_1 q_2}*	δ({q_1 q_2}, a) = q_0 λ-closure(q_0) = {q_0 q_1 q_2}	δ({q_1 q_2}, b) = q_2 λ-closure(q_2) = q_2 ←new state	δ({q_1 q_2}, c) = {q_0 q_1} λ-closure({q_0 q_1}) = {q_0 q_1 q_2}
q_2 *	δ(q_2, a) = Φ λ-closure(Φ) = Φ	δ(q_2, b) = Φ λ-closure(Φ) = Φ	δ(q_2, c) = Φ λ-closure(Φ) = Φ

Note:

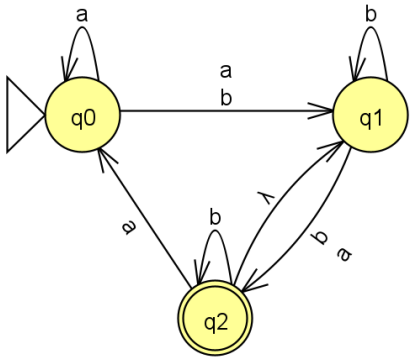
δ({q_0 q_1 q_2},a) = δ(q_0,a) ∪ δ(q_1,a) ∪ δ(q_2,a)

λ-closure({q_0 q_1 q_2}) = λ-closure(q_0) ∪ λ-closure(q_1) ∪ λ-closure(q_2)

DFA - Transition Diagram:



2. Convert the following λ-NFA to DFA using subset construction method.



Solution:

λ-NFA Transition table:

δ	a	b	λ-closure
→q_0	q_0q_1	q_1	{ q_0 }
q_1	q_2	q_1q_2	{ q_1 }
q_2*	q_0	q_2	{ q_1 q_2 }

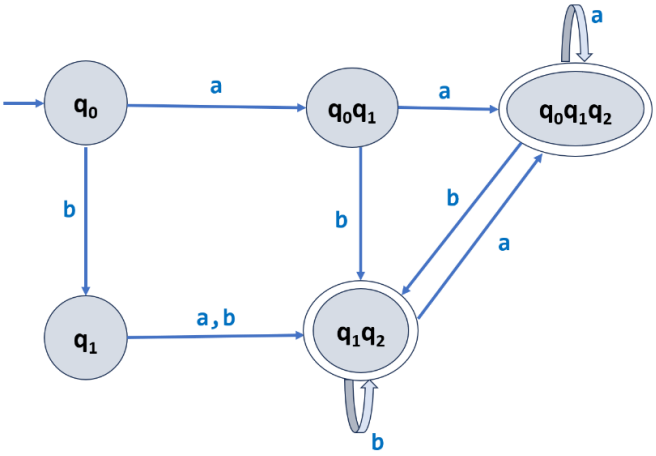
DFA Transition table:

The start state of DFA is the λ-closure of the start state of λ-NFA.

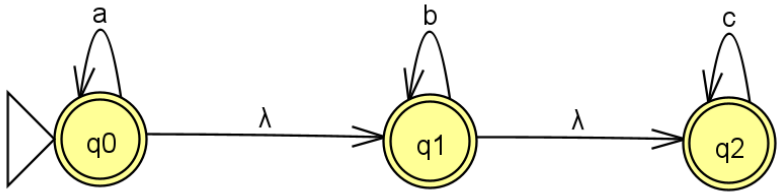
λ-closure(q_0) = q_0

δ	a	b
→q_0	δ({q_0}, a) = { q_0 q_1 } λ-closure({ q_0 q_1 }) = {q_0 q_1} ←new state	δ({q_0}, b) = q_1 λ-closure(q_1) = q_1 ←new state
{q_0 q_1}	δ({q_0 q_1}, a) = {q_0 q_1 q_2} λ-closure({q_0 q_1 q_2}) = {q_0 q_1 q_2} ←new state	δ({q_0 q_1}, b) = {q_1 q_2} λ-closure({q_1 q_2}) = { q_1 q_2} ←new state
q_1	δ(q_1, a) = q_2 λ-closure(q_2) = { q_1 q_2}	δ(q_1, b) = {q_1 q_2} λ-closure({q_1 q_2}) = { q_1 q_2}
{q_0 q_1 q_2}*	δ({q_0 q_1 q_2}, a) = {q_0 q_1 q_2} λ-closure({q_0 q_1 q_2}) = {q_0 q_1 q_2}	δ({q_0 q_1 q_2}, b) = {q_1 q_2} λ-closure({q_1 q_2}) = { q_1 q_2}
{q_1 q_2}*	δ({q_1 q_2}, a) = {q_0 q_2} λ-closure({q_0 q_2}) = {q_0 q_1 q_2}	δ({ q_1 q_2}, b) = {q_1 q_2} λ-closure({q_1 q_2})= { q_1 q_2}

DFA - Transition Diagram:



3. Convert the following λ-NFA to DFA using subset construction method.



Solution:

λ-NFA transition table:

δ	a	b	c	ε
→ q_0*	q_0	∅	∅	q_1
q_1*	∅	q_1	∅	q_2
q_2*	∅	∅	q_2	∅

Let us obtain λ-closure of each state.

λ-closure $\{q_0\} = \{q_0, q_1, q_2\}$

λ-closure $\{q_1\} = \{q_1, q_2\}$

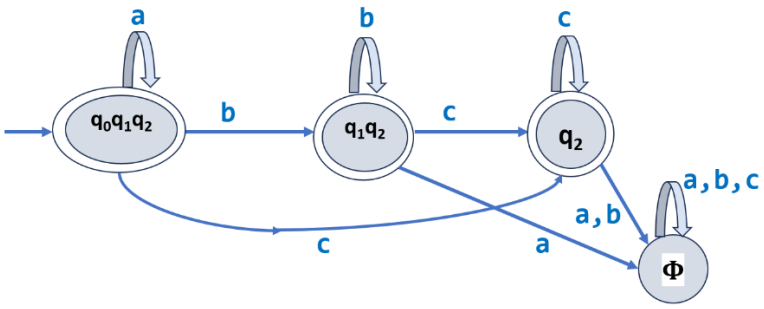
λ-closure $\{q_2\} = \{q_2\}$

DFA transition table:

Start state of DFA = λ-closure of start state of λ-NFA
= λ -closure(q_0)
= {q_0, q_1, q_2}

δ	a	b	c
→ {q_0 q_1 q_2}*	δ({q_0 q_1 q_2}, a) = q0 ε-closure(q_0) = {q_0 q_1 q_2}	δ({q_0 q_1 q_2}, b) = q1 ε-closure(q_1) = {q_1 q_2} ←new state	δ({q_0 q_1 q_2}, c) = q_2 ε-closure(q_2) = q_2 ←new state
{q_1 q_2}*	δ({q_1 q_2}, a) = ∅ ε-closure(∅) = ∅	δ({q_1 q_2}, b) = q_1 ε-closure(q_1) = {q_1 q_2}	δ({q_1 q_2}, c) = q_2 ε-closure(q_2) = q_2
q_2*	δ(q_2, a) = ∅ ε-closure(∅) = ∅	δ(q_2, b) = ∅ ε-closure(∅) = ∅	δ(q_2, c) = q2 ε-closure(q_2) = q_2

DFA - Transition Diagram:



DFA Minimization by Table filling Algorithm:

Why Minimize a DFA?

1. Efficiency in Computation

- A minimized DFA has fewer states, which means faster processing and less memory usage.
- This is especially useful in real-time systems, compilers, and pattern matching algorithms.

2. Simplicity and Clarity

- A smaller DFA is easier to understand, debug, and maintain.
- It helps to reveal the essential structure of the language being recognized.

3. Uniqueness

- For any regular language, the minimized DFA is unique. This makes it a canonical representation of the language.

4. Optimization in Applications

- In lexical analyzers (like those used in compilers), minimized DFAs reduce the overhead of token recognition.
- In network security or text search, smaller automata lead to faster scanning of input streams.

Note: The goal is to identify and merge equivalent states—states that behave identically for all input strings—so the DFA becomes as compact as possible without changing the language it accepts.

DFA Minimization by Table filling Algorithm:

Let the DFA be D= (Q, Σ, δ, q_0, F),

I. Remove the unreachable states, if any.

II. The table filling algorithm for minimizing the given DFA is as follows:

1) **Construct a table (triangular matrix table) of pairs (x, y) where x and y are the states in DFA, x,y ∈ Q, initially all the entries in the table are unmarked.**
   Note: While writing state names horizontally (i.e., from left to right) in table, consider only first state to last but one state, and while writing state names vertically (i.e., from top to bottom) in table, consider only from second state to last state.

2) **Repeat the following computation upwards in table for all the cells.**
   Mark the state pair (x, y) as distinguishable (using the cross symbol ×), iff x ∈ F and y ∉ F or the other way around. This marked pairs are called distinguishable (i.e., not same or not similar) pairs.

3) **Repeat the below computation upwards in table for all the unmarked cells following the step-2 process.**
   For each unmarked pair (x, y) and there exists a symbol a ∈ Σ such that a pair {δ(x, a), δ(y, a)} = (p, q). If (p, q) is already marked as distinguishable then mark the pair (x, y) as distinguishable.

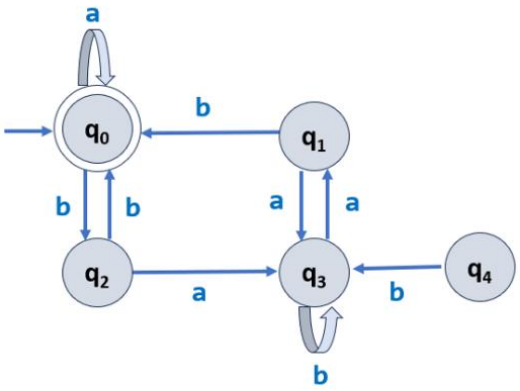
4) **After step-3 completion, if any pair (x,y) is not marked (i.e., indistinguishable), that pair can be merged and considered as a single state.**

Note:

- **If pairs (x1, y1), (x1, y2) and (x2, y2) are indistinguishable and they have common states among them, then we make single merged new state as (x1, y1, x2, y2)**
- If 2 states are not distinguished by table-filling algorithm, then they are equivalent.
- **DFA Minimization by Table filling Algorithm is known as Myhill-Nerode Theorem.**

Example 1:

1) Minimize the following DFA using table filling algorithm (aka Myhill-Nerode Theorem). Clearly mention the start and final state(s).



Solution:

Given DFA transition table:

δ	a	b
→ *q0	q0	q2
q1	q3	q0
q2	q3	q0
q3	q1	q3
q4	∅	q3

I. Eliminate unreachable state q4.

II. The table filling algorithm for minimizing the given DFA is as follows: (4-Marks)

1) Construct a table (triangular matrix table) of pairs (x, y) where x and y are the states in DFA, x,y ∈ Q, initially all the entries in the table are unmarked.

Note: While writing state names horizontally (i.e., from left to right) in table, consider only first state to last but one state, and while writing state names vertically (i.e., from top to bottom) in table, consider only from second state to last state.

q1			
q2			
q3			
	q0*	q1	q2

2) Repeat the following computation upwards in table for all the cells.

Mark the state pair (x, y) as distinguishable (using the symbol ×), iff x ∈ F and y ∉ F or the other way around. This marked pairs are called distinguishable (i.e., not same or not similar) pairs.

q1	×		
q2	×		
q3	×		
	q0*	q1	q2

3) Repeat the below computation upwards in table for all the unmarked cells following the step-2 process. For each unmarked pair (x, y) and there exists a symbol a ∈ Σ such that a pair {δ(x, a), δ(y, a)} = (p, q). If (p, q) is already marked as distinguishable then mark the pair (x, y) as distinguishable.

Consider unmarked cell (q1, q3):

{δ(q1, a), δ(q3, a)} = **(q3, q1)** [Getting the same pair (just ignore), then check with other input symbols]

{δ(q1, b), δ(q3, b)} = **(q0, q3)**

The pair (q0, q3) is already marked distinguishable then mark the pair (q1, q3) also as distinguishable.

Consider unmarked cell (q2, q3):
{δ(q2, a), δ(q3, a)} = **(q3, q1)**

The pair (q3, q1) is already marked distinguishable then mark the pair (q2, q3) also as distinguishable.

We came to know that the pair (q2, q3) is distinguishable, then no need to check the pair (q2, q3) for other input symbols.

Consider unmarked cell (q1, q2):
{δ(q1, a), δ(q2, a)} = **(q3, q3)** ← this pair does not exist, just ignore

{δ(q1, b), δ(q2, b)} = **(q0, q0)** ← this pair does not exist, just ignore

q_1	×		
q_2	×		
q_3	×	×	×
	q_0*	q_1	q_2

4)After step-3 completion, if any pair (x, y) is not marked (i.e., indistinguishable), that pair can be merged and considered as a single state in the minimized DFA.

The pair (q1, q2) is not marked (i.e., indistinguishable), that pair can be merged and considered as a single state.

q_1	×		
q_2	×		
q_3	×	×	×
	q_0*	q_1	q_2

Given DFA Transition table:

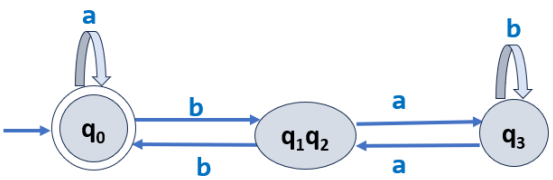
δ	a	b
→ q_0*	q_0	q_2
q_1	q_3	q_0
q_2	q_3	q_0
q_3	q_1	q_3

The indistinguishable pair (q1, q2) is merged as a single state and its transitions are also merged. The all occurrences of q1 and q2 in other state transitions are replaced by the pair {q1 q2}

Minimized DFA Transition table:

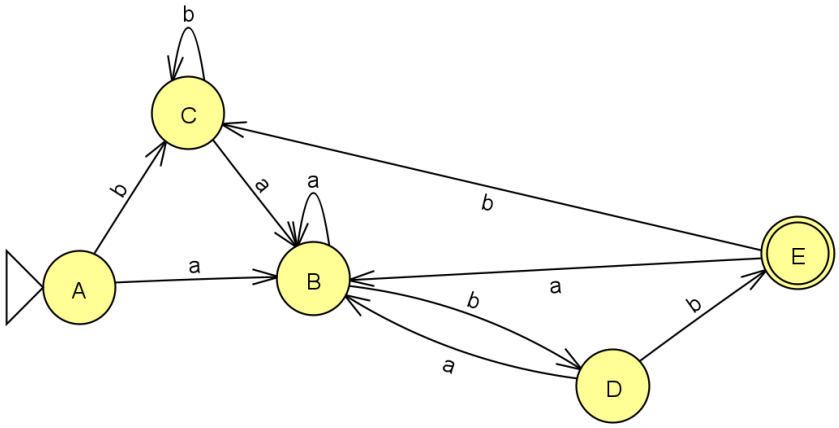
δ	a	b
→q_0*	q_0	{q_1 q_2}
{q_1 q_2}	q_3	q_0
q_3	{q_1 q_2}	q_3

Minimized DFA:

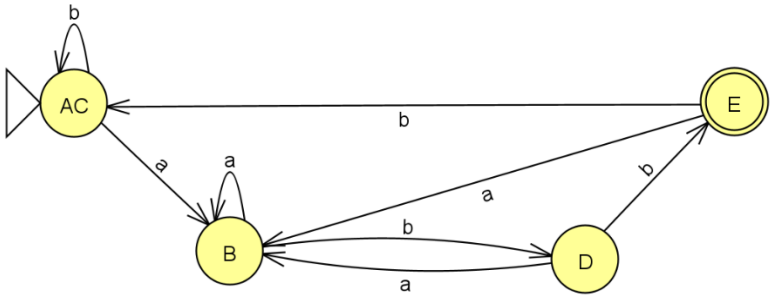


Exercise:

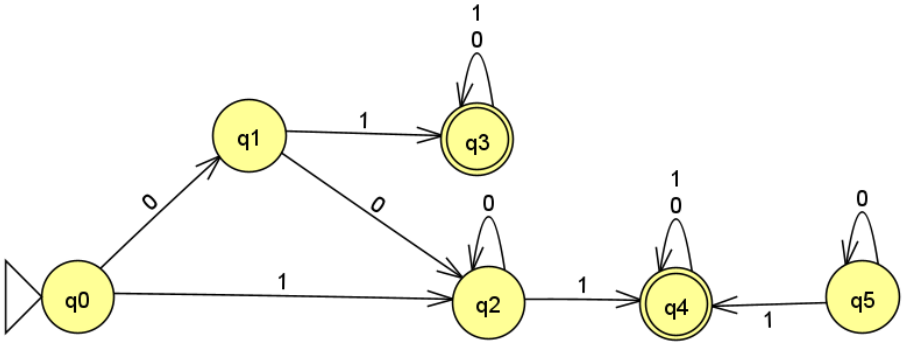
1) Minimize the following DFA using table filling algorithm (aka Myhill-Nerode Theorem). Clearly mention the start and final state(s).



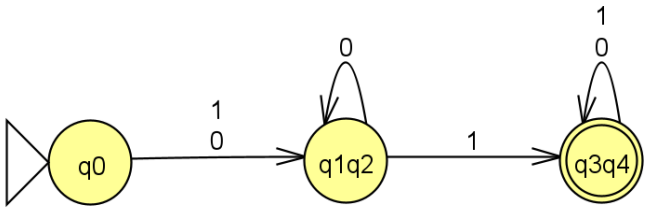
Minimized DFA:



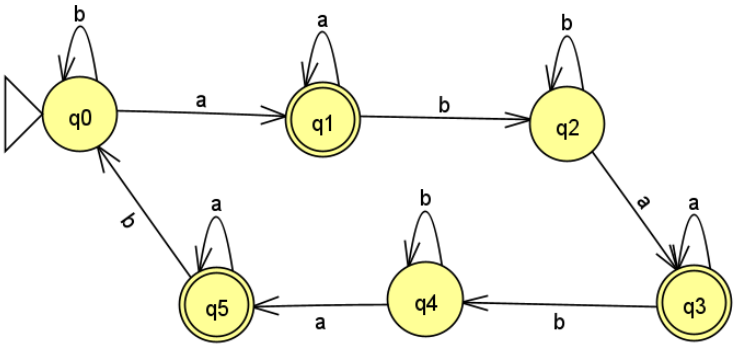
2) Minimize the following DFA using table filling algorithm (aka Myhill-Nerode Theorem). Clearly mention the start and final state(s).



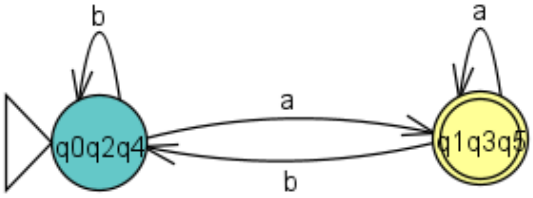
Minimized DFA:



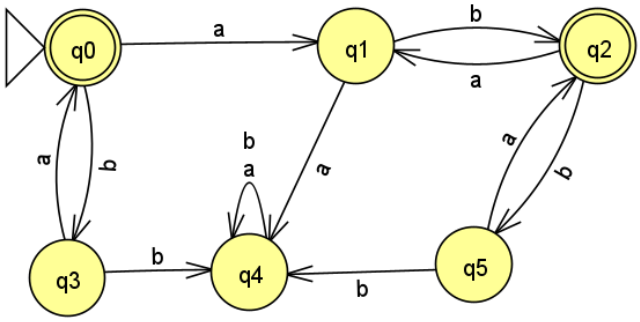
3) Minimize the following DFA using table filling algorithm (aka Myhill-Nerode Theorem). Clearly mention the start and final state(s).



Minimized DFA:



4) Minimize the following DFA using table filling algorithm (aka Myhill-Nerode Theorem). Clearly mention the start and final state(s).



Equivalence of Finite State Machines

The two finite automata (FA) are said to be equivalent if both accept the same set of strings over an input set Σ.

When two FA's are equivalent then, there is some string w over Σ. On acceptance of that string, both FA's should be in final state.

Method

The method for comparing two FA's is explained below:

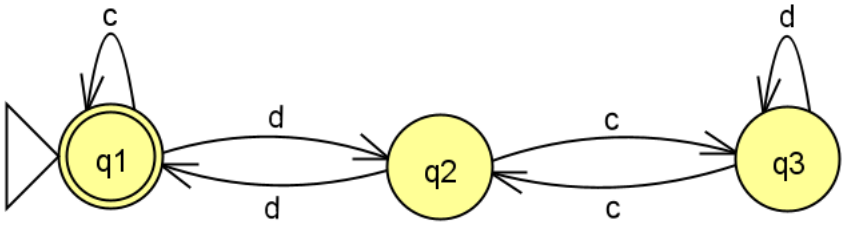
Let M1 and M2 be the two FA's and Σ be a set of input symbols.

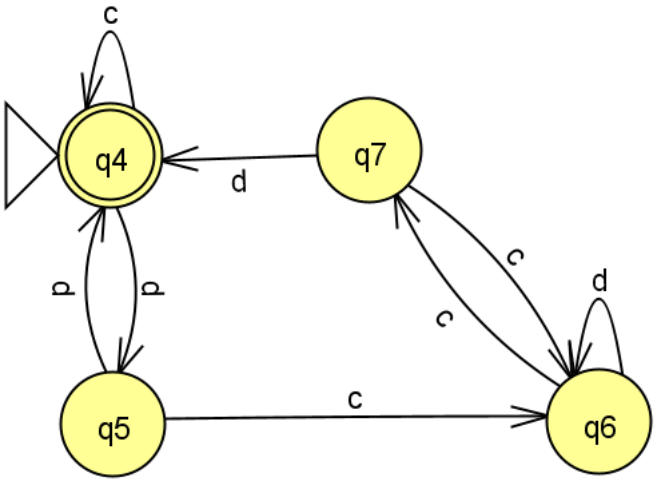
- **Step 1: Construct a transition table that has pairwise entries (p, q) where p ∈ M1 and q ∈ M2 for each input symbol. Start from the start state pair.**
- **Step 2: If we get in a pair as one final state(FS) and other non-final state(NFS) then we terminate the construction of transition table declaring that two FA's are not equivalent**

- **Step 3: The construction of the transition table gets terminated when there is no new pair appearing in the transition table.**

Example:

Consider the two Deterministic Finite Automata (DFA) and check whether they are equivalent or not.





The transition table is as follows:

δ	c	d
{q1, q4}	{q1, q4} FS FS	{q2, q5} ← new pair NFS NFS
{q2, q5}	{q3, q6} ← new pair NFS NFS	{q1, q4} FS FS
{q3, q6}	{q2, q7} ← new pair NFS NFS	{q3, q6} NFS NFS
{q2, q7}	{q3, q6} NFS NFS	{q1, q4} FS FS

Here, FS is the Final State and NFS is the Non-final State.

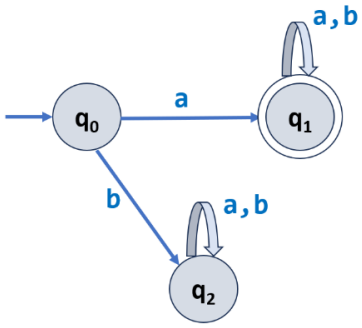
Therefore, by seeing the above table it is clear that we don't get one Final State and one Non-final State in any pair. Hence, we can declare that two DFA's are equivalent.

Complement of FSM's

Example:

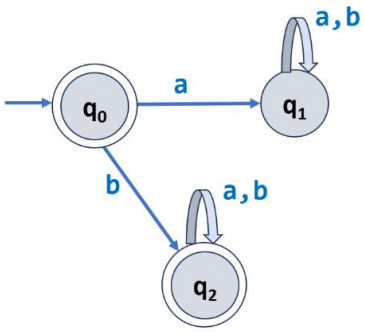
1) Find the complement of a given Deterministic Finite Automata (DFA).

M:



Solution: Ensure the DFA is complete and toggle final and non-final states.

M^C:



The complement DFA will accept all strings not accepted by the original DFA.

Note:

1) Complement of machine should also complement language recognised by FSM.
L={ a, ab, aa, aba, abb, aaa, aab, ……..}
L^c={ ε, b, ba, bb, bba, bbb, baa, bab, ……..}
The complement of a language L is the set of all strings not in L, over the same alphabet.

2) Complement of NFA may or may not work (NFAs are not closed under complementation directly.)
Suppose your NFA accepts strings over {0,1} that end in "01". Its complement would accept all strings except those ending in "01".

Reversal of a language through Finite Automata

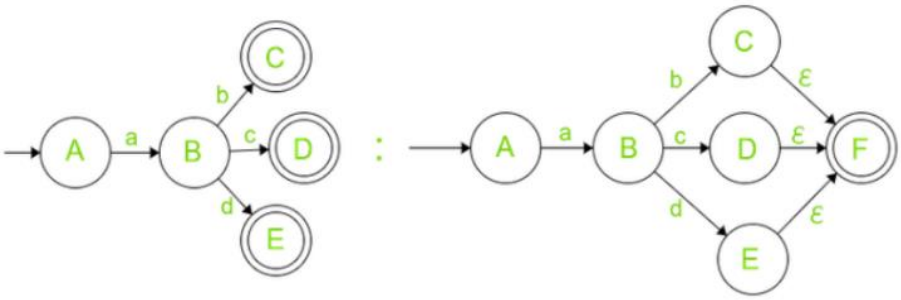
If a language L contains strings like “abcd”, “xyz” then its reversal L^R contains “dcba”, “zyx”—every string in L reversed character by character.

Let’s say you have a regular language L. To get L^R through finite automata follow these steps:

1) **Construct NFA for the given regular language L.**

Note:

- DFA is deterministic, but reversal requires nondeterminism (i.e., NFA).
- You Can’t Reverse a NFA having multiple final states.
  If the Automata have multiple final states it is recommended to strip their status of being final states and add outgoing Ɛ-transition to new and only final state with no outgoing transitions. For example,



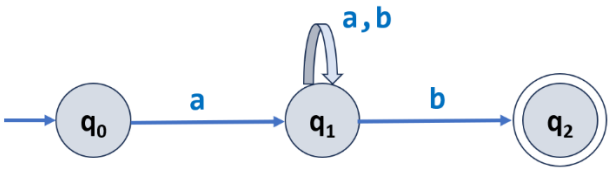
2) Swap Start and Final State.

3) Reverse all transitions

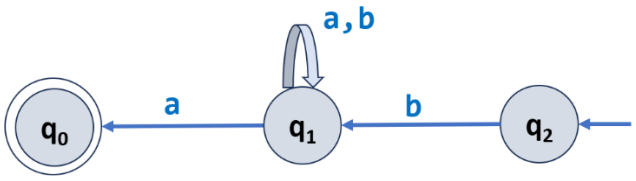
Example: Find the reversal of a below language through Finite Automata.

a) L = { awb | w∈{a,b}* }

Solution: Construct NFA for the given language.



Toggle initial and final state and reverse the arrows.



L^R = { bwa | w∈{a,b}* }

Exercise: Find the language reversal of below Finite Automata.

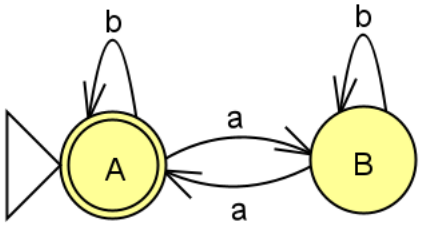
a) L = { wa | w∈{a,b}* }

Cross Product Operation in DFA

Let's understand the cross-product operation in Deterministic Finite Automata (DFA) with help of the below example

Designing a DFA-1 over {a, b} such that strings of the language contains even number of a's.
Designing a DFA-2 over {a, b} such that strings of the language contains even number of b's.

Step-1: Let's form a DFA-1 which count even number of a's



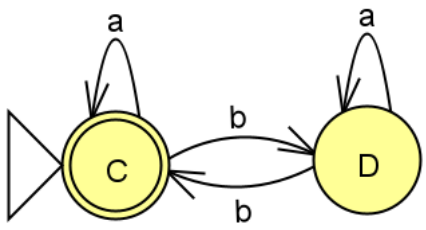
The language accepted by above DFA-1 is-

L = {ε, aab, b, baa, aabbbbb, aaaab, ...}

The language that does not accepted by the above DFA-1 is-

L = {aaa, abbb, baaa, bbaaba, ...}

Step-2: Let's form a DFA which count even number of b's



The language accepted by above DFA-2 is-

L = {ε, bba, a, abb, bbbbaaaa, bbbba, ...}

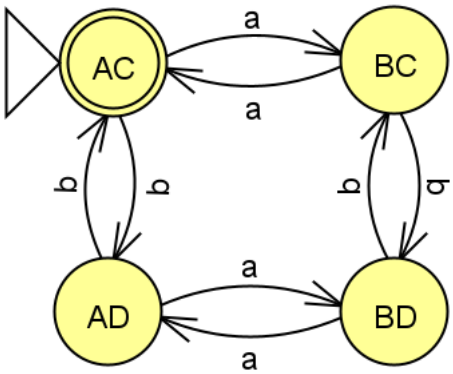
The language that does not accepted by the above DFA-2 is-

L = {bbb, bbba, abbb, aaba, ...}

Step-3: Here the states of step-1 and step-2 get cross multiplied and produce a result like below

{A, B} × {C, D} = {AC, AD, BC, BD}

And in the below, state transition diagram four states {AC, AD, BC, BD} used is the result of the cross product of step 1 and 2 states out of which 'AC' is the initial and final state too because in step 1 'A' is the initial and final state and in step 2 'C' is the initial and final state and rest are normal states.
Then the resultant state transition diagram after cross product operation becomes like below-



Thus the above DFA accepts all the language of an even number of a's and b's string and the language which is accepted and not accepted by above DFA is given below-

L1 = {ε, aa, bb, abab, aabb, baba, bbaa, ...}
L2 = {aaa, aaabb, aaabaabb, aaabb, baaba, bbbaa, ...}
L1 is accepted by above DFA but L2 does not.

Additional Info:

Note 1: Do finite automata have memory?

- **FSM doesn't have memory, It can't remember the things unlike Push Down Automata (i.e nothing but a FSM with Memory). Anything which require memory can't accepted by FSM.**
- All finite automata have “state” which is a form of memory. Any memory in addition to the state, make the finite automaton belong to a different class. So, strictly speaking DFAs and NFAs, which is what most people mean by **finite automata have no memory other than their state and in that sense are considered to have no memory.**
- Each state in a finite automaton represents a specific condition or configuration of the system. When an input is given, the automaton transitions from one state to another according to predefined rules. This transition is influenced by the current state and the input.
- While finite automata have memory in the form of states, they have limited memory capacity. They can only store information about their current state and cannot remember details about past states. This limited memory makes finite automata efficient for solving specific computational problems that do not require extensive memory.
- It's important to note that there are different types of finite automata with varying memory capabilities. For example, deterministic finite automata (DFA) have a fixed memory and can only be in one state at a time. On the other hand, nondeterministic finite automata (NFA) can be in multiple states simultaneously, allowing for more flexible memory representation.
- While finite automata have memory in the form of states, their memory capacity is limited and restricted to the current state.

Note 2:

- A scanner (i.e., lexical analyser) is a concrete realization of a finite automaton, a type of formal machine.
- A parser (i.e., syntax analyser) is a concrete realization of a push-down automaton. Just as context-free grammars add recursion to regular expressions, push-down automata add a stack to the memory of a finite automaton.
- It can be proven, constructively, that regular expressions and finite automata are equivalent: one can construct a finite automaton that accepts the language defined by a given regular expression, and vice versa.