UNIT-I: Introduction

Table of Content

Introduction of Theory of Computation/ Theory of Automata

Automata theory (also known as Theory Of Computation) is a theoretical branch of Computer Science and Mathematics, which mainly deals with the logic of computation with respect to simple machines, referred to as automata.
Automata* enables the scientists to understand how machines compute the functions and solve problems. The main motivation behind developing Automata Theory was to develop methods to describe and analyse the dynamic behaviour of discrete systems.

Automata Theory is a branch of computer science that deals with designing abstract self-propelled computing devices that follow a predetermined sequence of operations automatically. An automaton with a finite number of states is called a Finite Automaton. This is a brief and concise tutorial that introduces the fundamental concepts of Finite Automata, Regular Languages, and Pushdown Automata before moving onto Turing machines and Decidability.

The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

- **Symbol:** Symbol is the smallest building block, which can be any alphabet, letter or any picture.

a, b, c, 0, 1, ...

- **Alphabets (Σ):** Alphabets are set of symbols, which are always ***finite***.

Examples:
Σ = {0, 1} is an alphabet of binary digits
Σ = {0, 1, ... , 9} is an alphabet of decimal digits
Σ = {a, b, c}
Σ = {A, B, C, ..., Z}

- **String:** String is a ***finite*** sequence of symbols from some alphabet. String is generally denoted as ***w*** and length of a string is denoted as ***|w|***.

```
Empty string is the string with
zero occurrence of symbols,
represented as ε.
```

```
Number of Strings (of length 2)
that can be generated over the alphabet {a, b} -
                    -   -
                    a   a
                    a   b
                    b   a
                    b   b

Length of String |w| = 2
Number of Strings = 4


Conclusion:
For alphabet {a, b} with length n, number of
strings can be generated = 2^n.
```

Note – If the number of Σ's is represented by |Σ|, then number of strings of length n, possible over Σ is $|\Sigma|^n$.

- **Language:** A language is a *set of strings*, chosen form some Σ* or we can say- 'A language is a subset of Σ* '. A language which can be formed over ' Σ ' can be **Finite** or **Infinite**.

Example-
L_1 = { Set of all strings of length 2}
= { aa, ab, ba, bb } ----------→ *Finite Language*

L_2 = { Set of all strings which starts with 'a' }
= { a, aa, ab, aba, aba, aaa, abb, ... } ----------→ *Infinite Language*

Powers of ' Σ ' :

Say Σ = {a,b} then

Σ^0 = Set of all strings over Σ of length 0. {ε}

Σ^1 = Set of all strings over Σ of length 1. {a, b}

Σ^2 = Set of all strings over Σ of length 2. {aa, ab, ba, bb}

i.e. $|\Sigma^2|$= 4 and Similarly, $|\Sigma^3|$ = 8

```
Σ* is a Universal Set.
Σ* = Σ⁰ U Σ¹ U Σ² .........
   = {ε} U {a, b} U {aa, ab, ba, bb}
   = .............   //infinite language.
```

Kleene Star

The Kleene star, ∑*, is a unary operator on a set of symbols or strings, ∑, that gives the infinite set of all possible strings of all possible lengths over ∑ including λ.

Representation − ∑* = ∑0 ∪ ∑1 ∪ ∑2 ∪……. where ∑p is the set of all possible strings of length p.

Example − If ∑ = {a, b}, ∑* = {λ, a, b, aa, ab, ba, bb,……..}

Kleene Closure / Plus

Definition − The set ∑+ is the infinite set of all possible strings of all possible lengths over ∑ excluding λ.

Representation − ∑+ = ∑1 ∪ ∑2 ∪ ∑3 ∪…….

∑+ = ∑* − { λ }

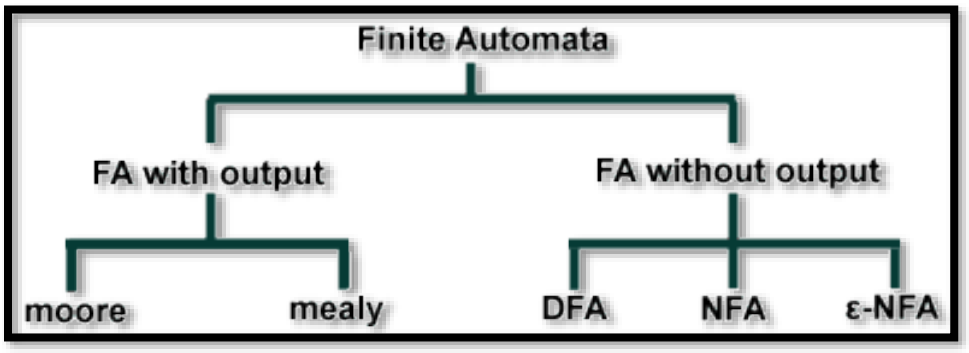
Example − If ∑ = { a, b } , ∑+ = { a, b, aa, ab, ba, bb,……..}

Language

Definition − A language is a subset of ∑* for some alphabet ∑. It can be finite or infinite.

Example − If the language takes all possible strings of length 2 over ∑ = {a, b}, then L = { ab, aa, ba, bb }

Finite Automata



Finite Automata (FA) is the simplest machine to recognize patterns.
An automaton can be represented by a 5-tuple (Q, ∑, δ, q0, F), where –

Q : Finite set of states.
∑ : set of Input Symbols.
q : Initial state.
F : set of Final States.
δ : Transition Function.

FA is characterized into two types:

1) Deterministic Finite Automata (DFA)

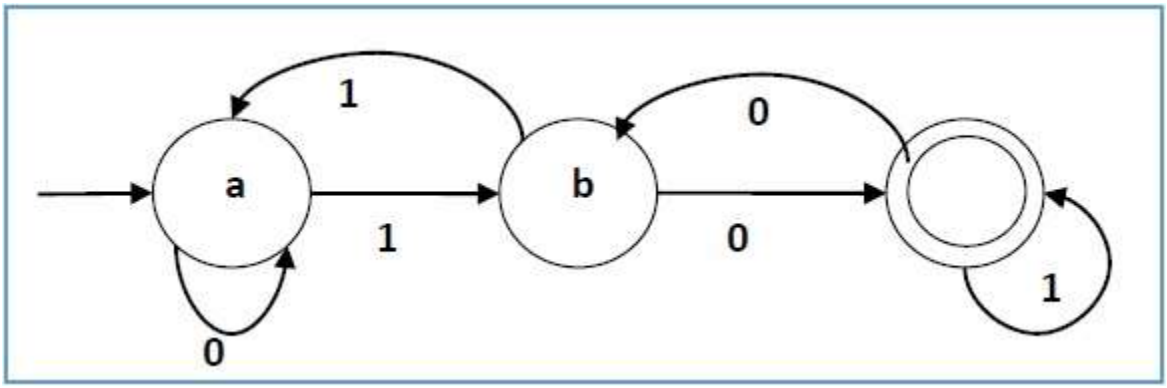
In a DFA, for a particular input character, machine goes to one state only. A transition function is defined on
every state for every input symbol. Also in DFA null (or ε) move is not allow, i.e., DFA cannot change state
without any input character.

DFA consists of 5 tuples {Q, ∑, q, F, δ}.
Q : set of all states.
∑ : set of input symbols. (Symbols which machine takes as input)
q : Initial state. (Starting state of a machine)
F : set of final state.
δ : Transition Function, defined as δ : Q X ∑ --> Q.

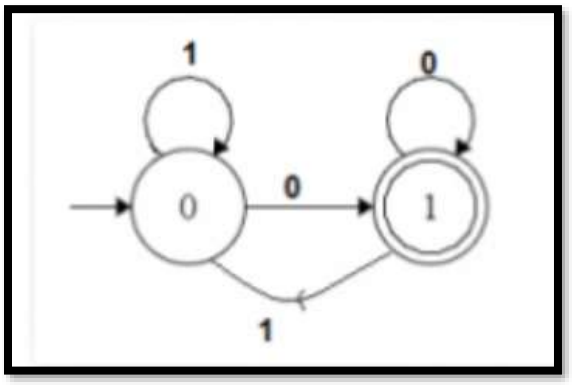
Transition function δ as shown by the following table –

Present State	Next State for Input 0	Next State for Input 1
a	A	B
b	C	A
c	B	C

Its graphical representation would be as follows –



For example, below DFA with ∑ = {0, 1} accepts all strings ending with 0.



One important thing to note is, there can be many possible DFAs for a pattern. A DFA with minimum number of states is generally preferred.

2) Nondeterministic Finite Automata(NFA)

1. Null (or ε) move is allowed i.e., it can move forward without reading symbols.
2. Ability to transit to any number of states for a particular input.

However, these above features don't add any power to NFA. If we compare both in terms of power, both are equivalent.
Due to above additional features, NFA has a different transition function, rest is same as DFA.

δ: Transition Function
δ: Q X (∑ U ε) --> 2 ^ Q.

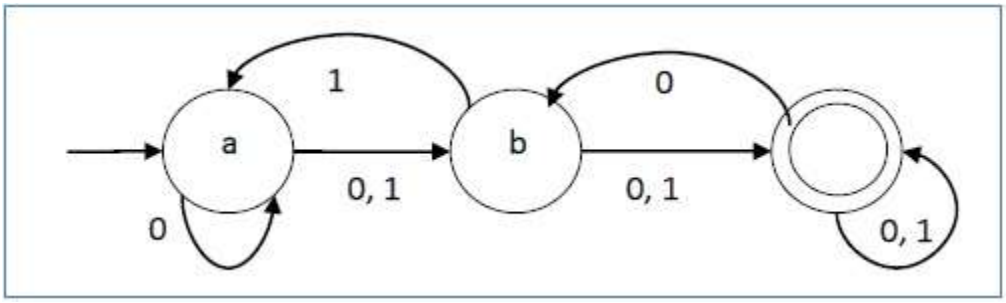
As you can see in transition function for any input including null (or &epsilon), NFA can go to any state number of states.

The transition function δ as shown below –

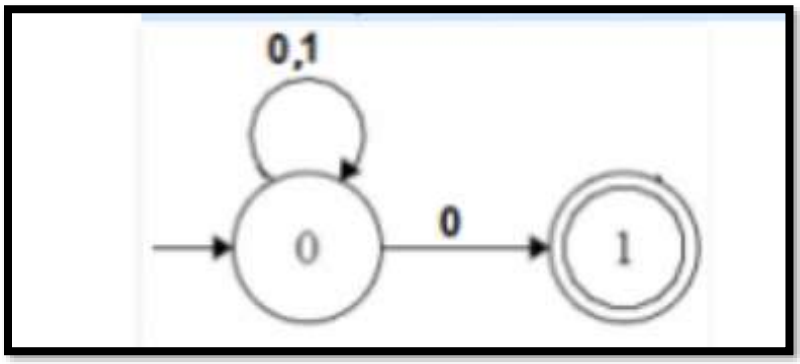
Present State	Next State for Input 0	Next State for Input 1
a	a, b	b

b	C	a, c
c	b, c	c

Its graphical representation would be as follows –



For example, below is a NFA for above problem



One important thing to note is, in NFA, if any path for an input string leads to a final state, then the input
string accepted. For example, in above NFA, there are multiple paths for input string "00". Since, one of the
paths lead to a final state, "00" is accepted by above NFA.

Some Important Points:

1. Every DFA is NFA but not vice versa.
2. Both NFA and DFA have same power and each NFA can be translated into a DFA.
3. There can be multiple final states in both DFA and NFA.

3. NFA is more of a theoretical concept.
4. DFA is used in Lexical Analysis in Compiler.

DFA vs NDFA

The following table lists the differences between DFA and NDFA.

DFA	NDFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called *deterministic*.	The transition from a state can be to multiple next states for each input symbol. Hence it is called *non-deterministic*.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

Epsilon nondeterministic finite automaton (NFA)

An **epsilon nondeterministic finite automaton (NFA)** has null or epsilon transitions from one state to another. Epsilon NFA is also called a **null NFA** or an **NFA lambda**.

A regular expression for a language forms an epsilon NFA. This epsilon NFA then converts to a simple NFA. We then use the simple NFA to make a **deterministic finite automaton (DFA)**.

Conversion of Epsilon-NFA to NFA

To remove the epsilon move/Null move from epsilon-NFA and to convert it into NFA, we follow the steps mentioned below.

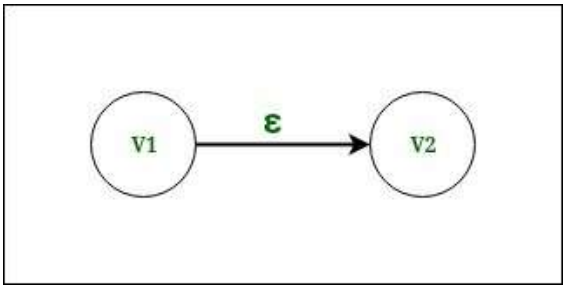


Figure – Vertex v1 and Vertex v2 having an epsilon move

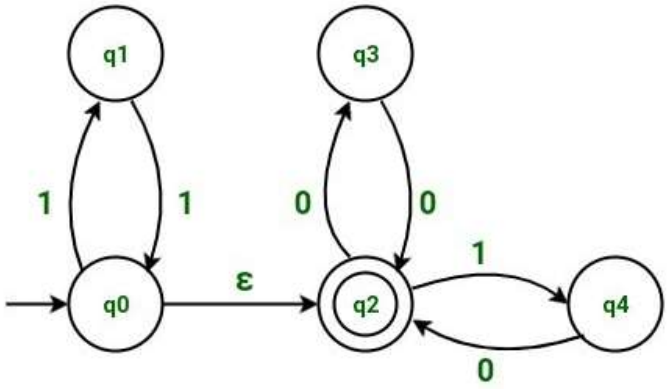
Step-1: Consider the two vertexes having the epsilon move. Here in Fig.1 we have vertex v1 and vertex v2 having epsilon move from v1 to v2.

Step-2: Now find all the moves to any other vertex that start from vertex v2 (other than the epsilon move that is considering). After finding the moves, duplicate all the moves that start from vertex v2, with the same input to start from vertex v1 and remove the epsilon move from vertex v1 to vertex v2.

Step-3: See that if the vertex v1 is a start state or not. If vertex v1 is a start state, then we will also make vertex v2 as a start state. If vertex v1 is not a start state, then there will not be any change.

Step-4: See that if the vertex v2 is a final state or not. If vertex v2 is a final state, then we will also make vertex v1 as a final state. If vertex v2 is not a final state, then there will not be any change. Repeat the steps(from step 1 to step 4) until all the epsilon moves are removed from the NFA. Now, to explain this conversion, let us take an example.

Example: Convert epsilon-NFA to NFA. Consider the example having states q0, q1, q2, q3, and q4.



In the above example, we have 5 states named as q0, q1, q2, q3 and q4. Initially, we have q0 as start state and q2 as final state. We have q1, q3 and q4 as intermediate states.

Transition table for the above NFA is:

States/Input	Input 0	Input 1	Input epsilon
q0	–	q1	q2
q1	–	q0	–
q2	q3	q4	–
q3	q2	–	–
q4	q2	–	–

According to the transition table above,

state q0 on getting input 1 goes to state q1.

State q0 on getting input as a null move (i.e. an epsilon move) goes to state q2.

State q1 on getting input 1 goes to state q0.

Similarly, state q2 on getting input 0 goes to state q3, state q2 on getting input 1 goes to state q4.

Similarly, state q3 on getting input 0 goes to state q2.

Similarly, state q4 on getting input 0 goes to state q2. We can see that we have an epsilon move from state q0 to state q2, which is to be removed. To remove epsilon move from state q0 to state q1, we will follow the steps mentioned below.

Step-1: Considering the epsilon move from state q0 to state q2. Consider the state q0 as vertex v1 and state q2 as vertex v2.

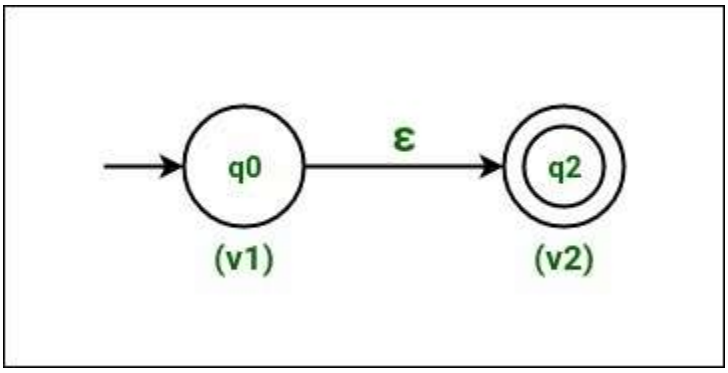


Figure – State q0 as vertex v1 and state q2 as vertex v2

Step-2: Now find all the moves that starts from vertex v2 (i.e. state q2). After finding the moves, duplicate all the moves that start from vertex v2 (i.e state q2) with the same input to start from vertex v1 (i.e. state q0) and remove the epsilon move from vertex v1 (i.e. state q0) to vertex v2 (i.e. state q2). Since state q2 on getting input 0 goes to state q3. Hence on duplicating the move, we will have state q0 on getting input 0 also to go to state q3.

Similarly state q2 on getting input 1 goes to state q4. Hence on duplicating the move, we will have state q0 on getting input 1 also to go to state q4.

So, NFA after duplicating the moves is:

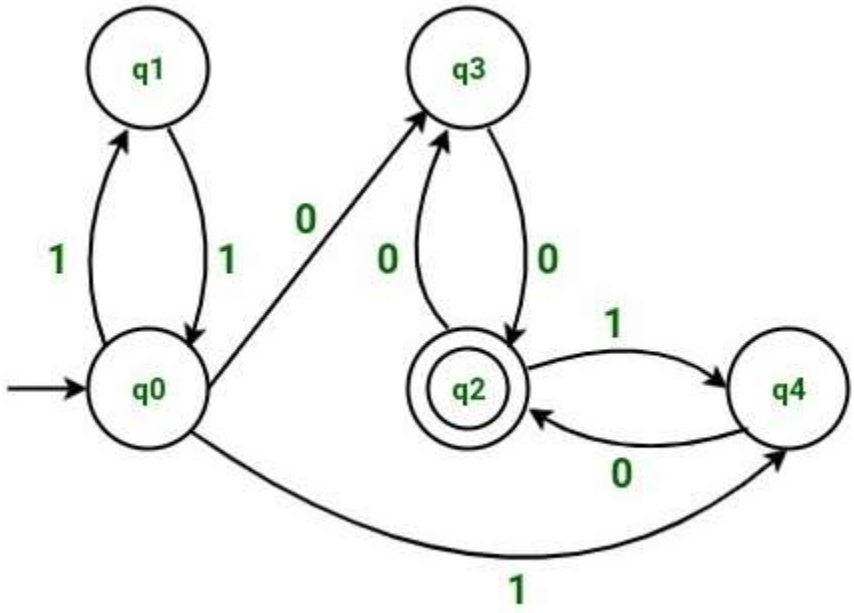


Figure – NFA on duplicating moves

Step-3: Since vertex v1 (i.e. state q0) is a start state. Hence we will also make vertex v2 (i.e. state q2) as a start state. Note that state q2 will also remain as a final state as we had initially. NFA after making state q2 also as a start state is:

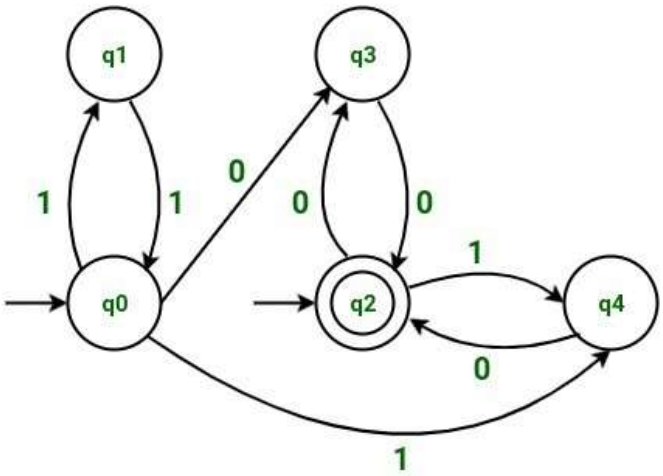
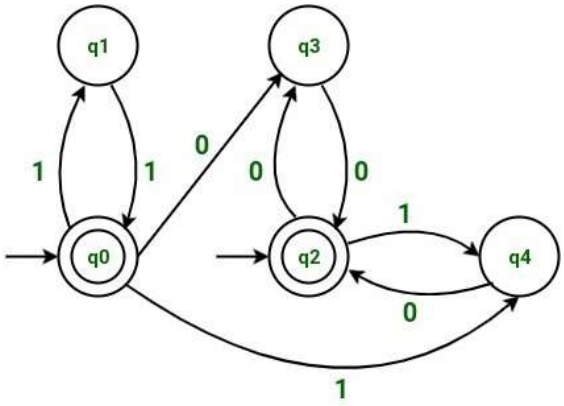


Figure – NFA after making state q2 as a start state

Step-4: Since vertex v2 (i.e. state q2) is a final state. Hence we will also make vertex v1 (i.e. state q0) as a final state. Note that state q0 will also remain as a start state as we had initially. After making state q0 also as a final state, the resulting



NDFA to DFA Conversion

Algorithm

Input – An NDFA

Output – An equivalent DFA

Step 1 – Create state table from the given NDFA.

Step 2 – Create a blank state table under possible input alphabets for the equivalent DFA.

Step 3 – Mark the start state of the DFA by q0 (Same as the NDFA).

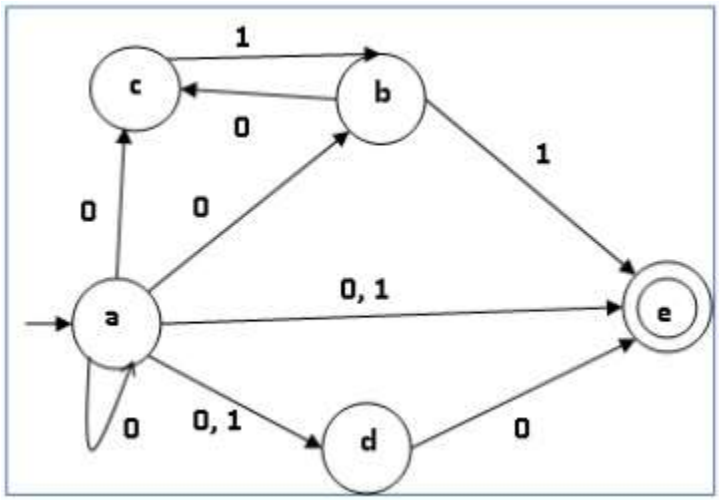
Step 4 – Find out the combination of States $\{Q_0, Q_1, \ldots, Q_n\}$ for each possible input alphabet.

Step 5 – Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 – The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

Example

Let us consider the NDFA shown in the figure below.



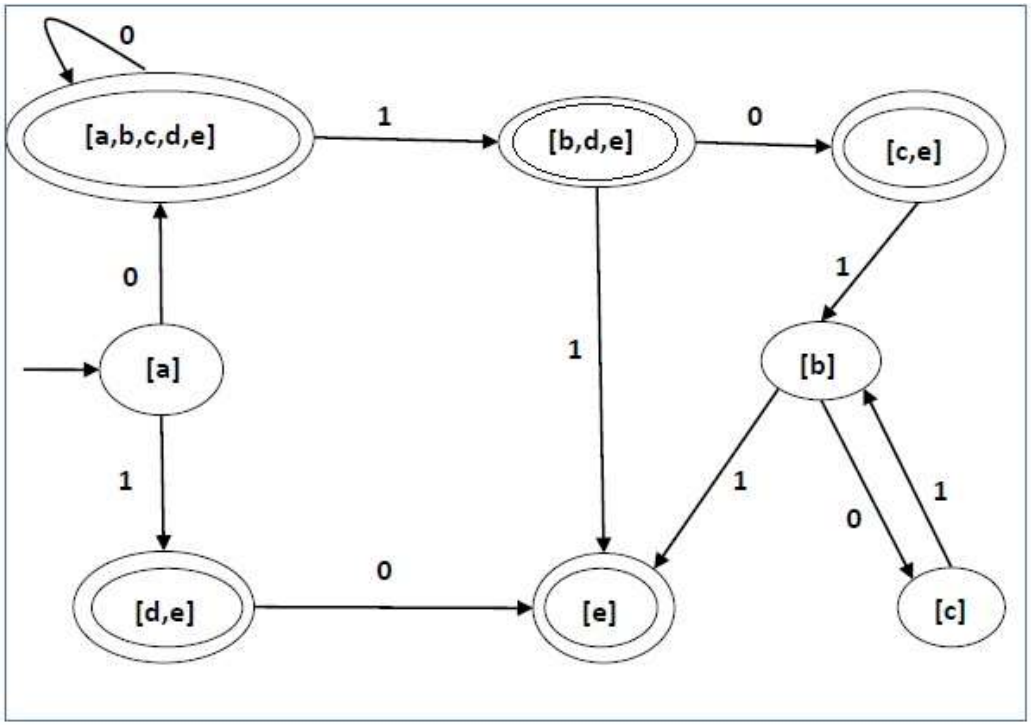
q	δ(q,0)	δ(q,1)
a	{a,b,c,d,e}	{d,e}
b	{c}	{e}
c	∅	{b}
d	{e}	∅
e	∅	∅

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

Q	δ(q,0)	δ(q,1)
[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	∅
[b,d,e]	[c,e]	[e]
[e]	∅	∅
[c, e]	∅	[b]

[b]	[c]	[e]
[c]	∅	[b]

The state diagram of the DFA is as follows –



DFA Minimization

Algorithm

Step 1 – Remove aal dead state from given DFA.

Step 2 – All the states **Q** are divided in two partitions – **final states** and **non-final states** and are denoted by P_0. All the states in a partition are 0^{th} equivalent. Take a counter **k** and initialize it with 0.

Step 3 – Increment k by 1. For each partition in P_k, divide the states in P_k into two partitions if they are k-distinguishable. Two states within this partition X and Y are k-

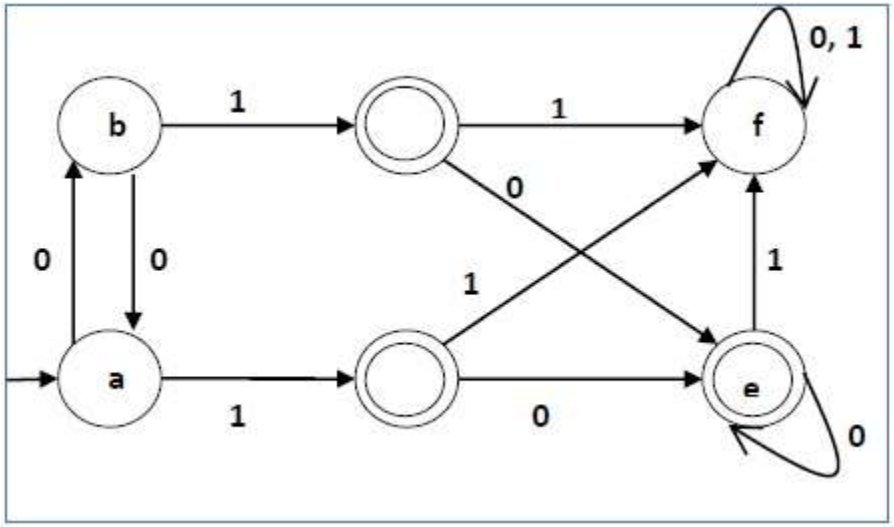
distinguishable if there is an input **S** such that **δ(X, S)** and **δ(Y, S)** are (k-1)-distinguishable.

Step 4 – If $P_k \neq P_{k-1}$, repeat Step 2, otherwise go to Step 4.

Step 5 – Combine k^{th} equivalent sets and make them the new states of the reduced DFA.

Example

Let us consider the following DFA –



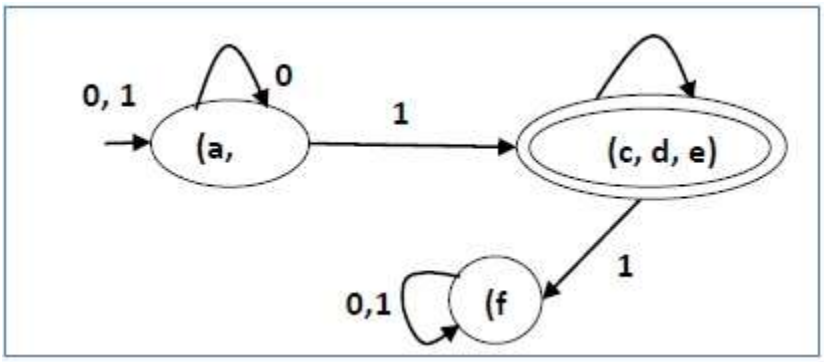
q	δ(q,0)	δ(q,1)
a	B	c
b	A	d
c	E	f
d	E	f
e	E	f
f	F	f

Let us apply the above algorithm to the above DFA –

- P_0 = {(c,d,e), (a,b,f)}
- P_1 = {(c,d,e), (a,b),(f)}
- P_2 = {(c,d,e), (a,b),(f)}

Hence, $P_1 = P_2$.

There are three states in the reduced DFA. The reduced DFA is as follows –



Moore and Mealy Machines

Mealy Machine

A Mealy Machine is an FSM whose output depends on the present state as well as the present input.

It can be described by a 6 tuple $(Q, \sum, O, \delta, X, q_0)$ where –

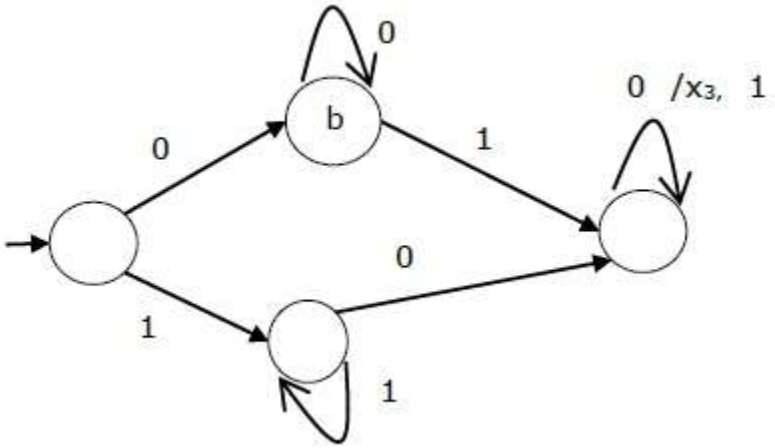
- **Q** is a finite set of states.
- **∑** is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where $\delta: Q \times \sum \rightarrow Q$
- **X** is the output transition function where $X: Q \times \sum \rightarrow O$
- $\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).

The state table of a Mealy Machine is shown below –

Present state	Next state			
	input = 0		input = 1	
	State	Output	State	Output
→ a	B	x_1	c	x_1
B	B	x_2	d	x_3

C	D	x_3	c	x_1
D	D	x_3	d	x_2

The state diagram of the above Mealy Machine is –



Moore Machine

Moore machine is an FSM whose outputs depend on only the present state.

A Moore machine can be described by a 6 tuple (Q, ∑, O, δ, X, q_0) where –

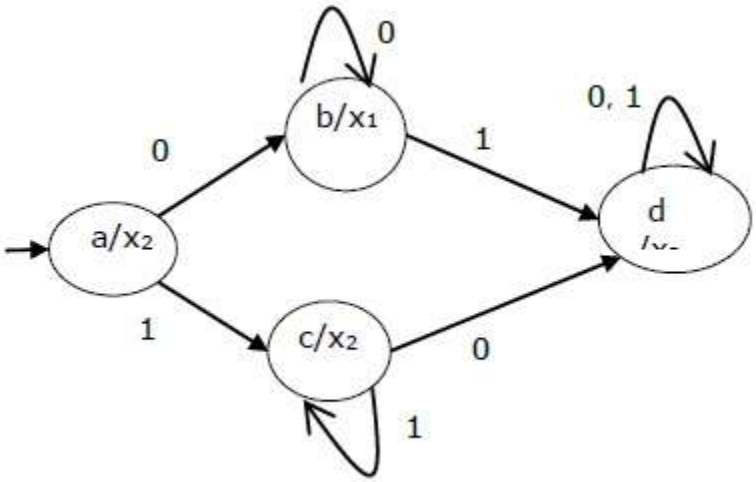
- **Q** is a finite set of states.
- **∑** is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where δ: Q × ∑ → Q
- **X** is the output transition function where X: Q → O
- **q_0** is the initial state from where any input is processed (q_0 ∈ Q).

The state table of a Moore Machine is shown below –

Present state	Next State		Output
	Input = 0	Input = 1	
→ a	B	c	x_2
b	B	d	x_1
c	C	d	x_2

d	D	d	x_3

The state diagram of the above Moore Machine is –



Mealy Machine vs. Moore Machine

The following table highlights the points that differentiate a Mealy Machine from a Moore Machine.

Mealy Machine	Moore Machine
Output depends both upon the present state and the present input	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
The value of the output function is a function of the transitions and the changes, when the input logic on the present state is done.	The value of the output function is a function of the current state and the changes at the clock edges, whenever state changes occur.
Mealy machines react faster to inputs. They generally react in the same clock cycle.	In Moore machines, more logic is required to decode the outputs resulting in more circuit delays. They generally react one clock cycle later.