University of sulaimani
College of science
Department of Computer Science

Computation

Finite Automata

Lecture One

Mzhda Hiwa Hama
2024-2025

Theory of Computation

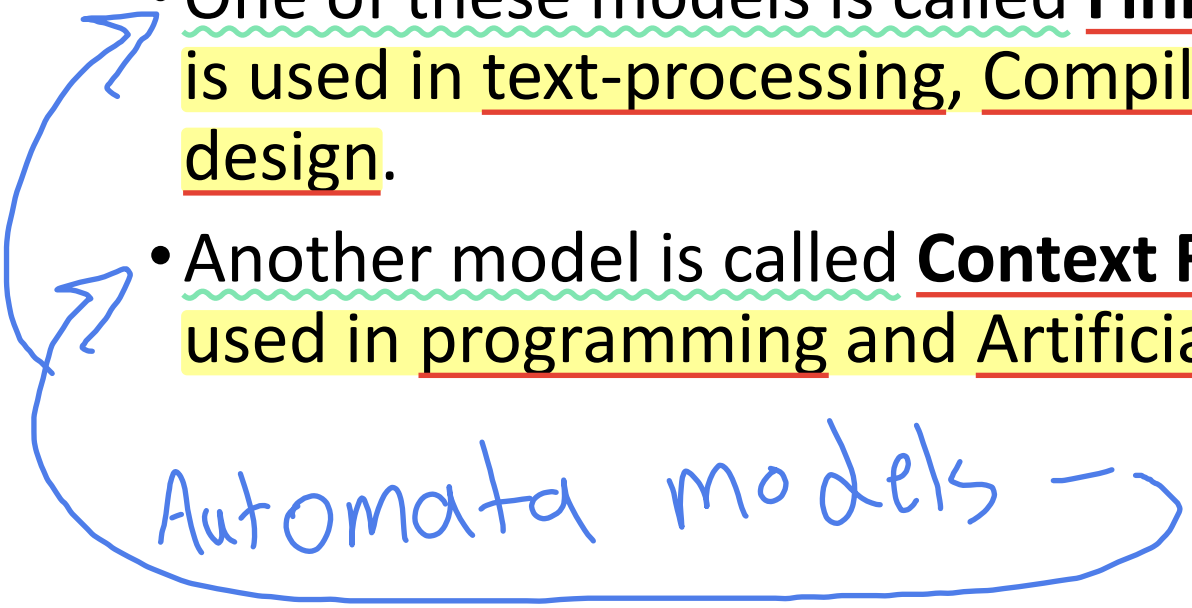
- The Theory of Computation deals with how effectively problems can be solved on a model of computation using an algorithm.

This field is divided into three major areas:

1. Automata Theory (deals with the mathematical model of computation)
2. Complexity Theory (How hard a certain problem is?).
3. Computability Theory (Is a certain problem solvable?)

Automata Theory

- Deals with the definitions and the properties of mathematical models of computation.
- These models play a role in several applied areas of computer science.
- One of these models is called **Finite Automata** which is used in text-processing, Compiler and hardware design.
- Another model is called **Context Free Grammar** that's used in programming and Artificial Intelligence.



Complexity Theory

- Computer problems :
  easy
  hard

- What makes some problems computationally hard and others easy ?
- We don't know what make them easy and hard but we know how to classify each problems with an elegant scheme.
  – Cryptography is supposed to be a hard problem.

Computability Theory

- There are some problems which can't be solved by computers, e.g., determining whether a mathematical statement is true or false.

- The object of the Computability Theory is to classify the problems whether they are solvable by computers or not.

Terminologies related to Automata Theory

Alphabets: A finite set of symbol, and are typically thought of as representing letters, characters, or digits

Example: {a,b,c,…x.y.z}. {0,1},{0,1,2,3,4,5,6,7,8,9}

We use Greek letter ∑ to designate an Alphabet.

String over an alphabet: A finite sequence of symbols from the alphabet

Example: 01011 - over {0,1}

3786 - over {0,1,2,3,4,5,6,7,8,9}

A string with one symbol only = symbol itself

Length of a string - The number of symbols

if w = **abcd**, $|w|$ = 4

Empty string - no symbols ,the string of length zero is called the empty string and is written **ϵ**.

Operations on strings

- **Concatenation:** combines two strings by putting them one after the other.

Eg: ***x*** **= united**, ***y*** **= states**, then ***xy*** **= unitedstates**, or simply ***yx*** **= statesunited**

- The concatenation of the empty string with any other string gives the string itself:

$x \epsilon = \epsilon x = x$

- **Substring:** String x is a Substring of y if x appears in y as a single element or a serious of related elements.

Eg: for y=abcde , x= bcd. x is a substring of y but

w=ac is not a substring of y

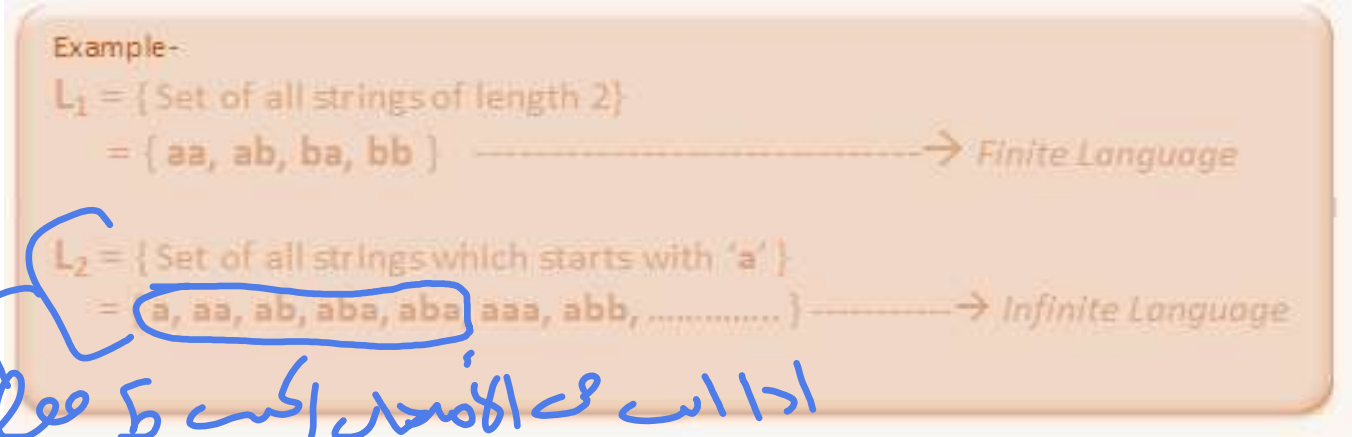
- **Prefix & Suffix**: a string x is a **prefix** of string y if x appears as a substring in the beginning of y. correspondingly, if x appears in the end of y then x is a **Suffix** of y.

Example:

for y= abcde then we have abc as a **prefix** of y, and de as a **Suffix** of y.

Languages

- Language is any set of strings over an alphabet ∑.
- If ∑ is an alphabet, then ∑* is the set of all strings over ∑.
- Eg: if ∑= {a, b, c} then ∑*= {ϵ, a, b, c, aa, ab, ac, ba,bb … , aaaaaabbbaababa, …}
- let L be language is any subset of ∑*.
- A language which can be formed over ' Σ ' can be **Finite** or **Infinite**.



Some Operations on Languages

- **Concatenation**

  If L_1 and L_2 are languages, then $L = L_1 \circ L_2$ (or simply L_1L_2) is the set: $L = \{xy , x \in L_1, y \in L_2\}$

- **Union**

  $L1 \cup L2 = \{x|\ x \in L1 \text{ or } x \in L2\}$

- **Kleene star(Kleene closure)** of a language L is the set of all strings obtained by concatenating zero or more strings from L. It is denoted by L*.

Examples

- Let L1= {white, black} and

  L2 = {chocolate, cream} from the alphabet ∑ of the 26 English letters, then :

- L1∪L2 = {white, black, chocolate, cream}
- L1.L2 = {whitechocolate, whitecream, blackchocolate, blackcream}.
- L1*={ϵ, white, black, whitewhite, whiteblack,blackwhite, blackblack, whitewhitewhite, whitewhitewhietblack,…… }

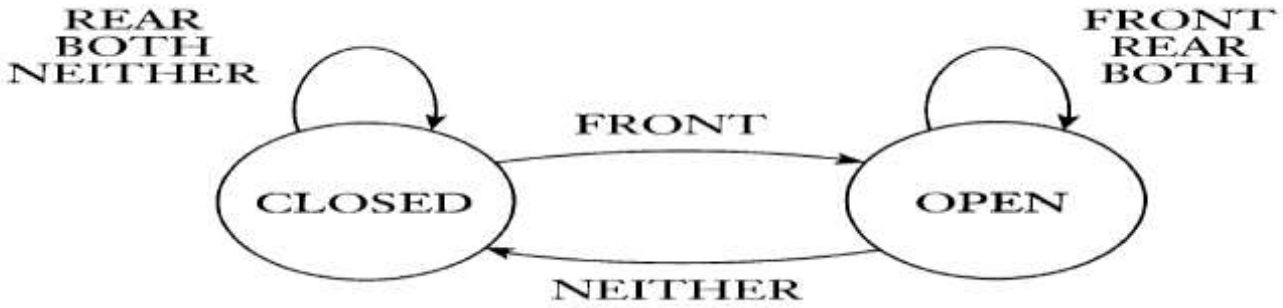
Finite State Machine or Finite Automaton

Finite state machine is the simplest model of computation and generally they are good models for computers with an extremely limited amount of memory. It is a machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs and/or a condition is satisfied; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

- The controller of an automatic door is an example of such a machine

Top view of an Automatic door

- The controller is in either of two states: "OPEN" or "CLOSED".
- There are four possible input conditions: "FRONT", "REAR", "BOTH", and "NEITHER"
- This controller is a computer with a single bit of memory, capable of recording which of the two states the controller is in.



State Diagram of an Automatic door

State transition table

- The controller moves from state to state depending on the input it receives.

	input signal			
state	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

Finite state automaton – Cont'd

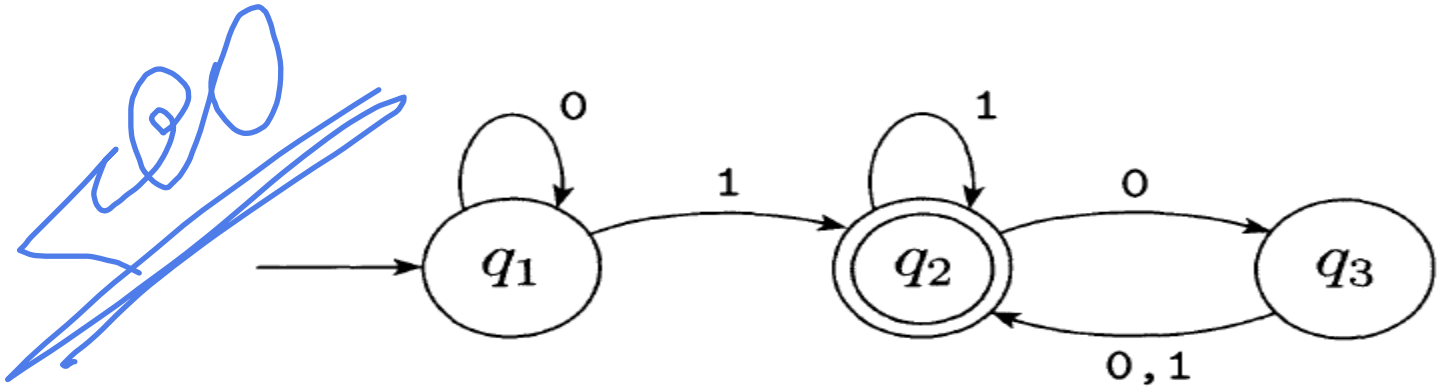
- Generally, Finite Automata (or Finite State Machine)is a useful computational model for both hardware and certain types of software.

- It processes a string of sequences as input and produces an output by **Accepting (Recognizing)** or **Rejecting** that string which belongs to a particular language.

- Eg. Designing a Finite Automata to accept the strings that are legal variable or method names in Java.

Finite state automaton – Cont'd

- Without reference to any particular application, the following figure is used to describe the mathematical theory of Finite Automata.



A finite automata called M1 which has three states

- The figure is called the **state diagram** of M1. It has three states, labeled q1,q2, and q3.

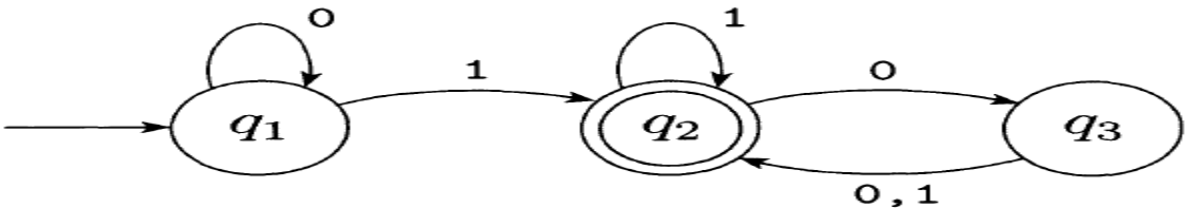
- The **start** state, q1, is indicated by the arrow pointing to it from nowhere.

- The **accept** state, q3, is the one with a double circle.

- The arrows going from one state to another are called, **transitions**.

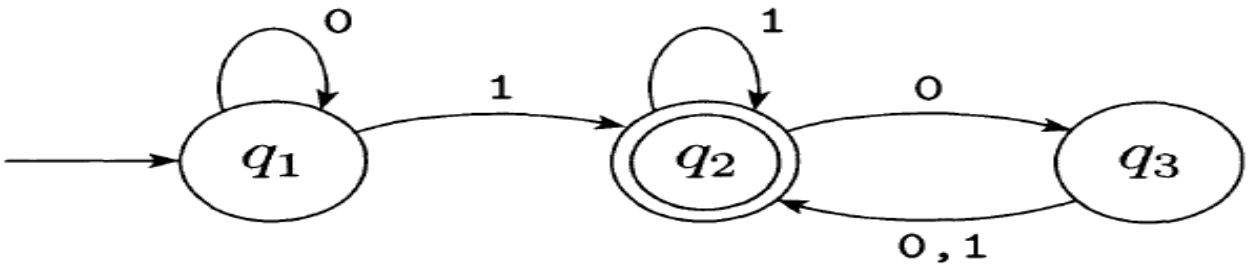
How it works

- When the automaton receives an input string such as 1101, it processes that string and produces an output. The output is either accept or reject. The processing begins in M1's start state. The automaton receives the symbols from the input string one by one from left to right. After reading each symbol, M1 moves from one state to another along the transition that has that symbol as label. When it reads the last symbol, M1 produces its output. The output is accept if M1 is now in an accept state and reject if it is not.

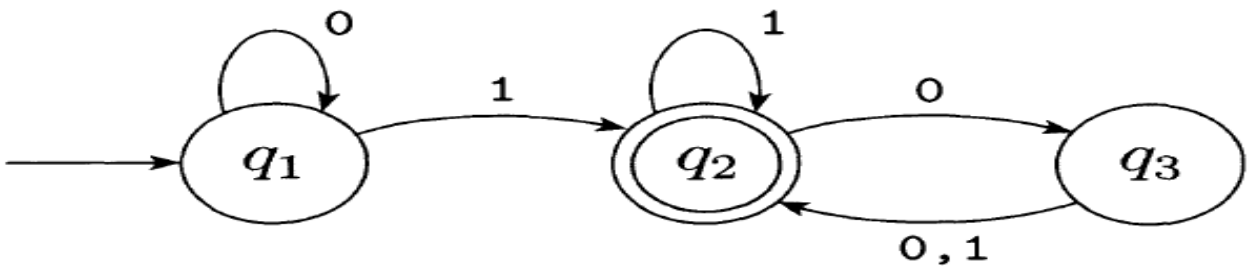


Example: 1101

1. Start in state q_1.
2. Read 1, follow transition from q_1 to q_2.
3. Read 1, follow transition from q_2 to q_2.
4. Read 0, follow transition from q_2 to q_3.
5. Read 1, follow transition from q_3 to q_2.
6. *Accept* because M_1 is in an accept state q_2 at the end of the input.



- Experimenting this machine with different input strings shows that it can accept strings like 1, 01, 11, and 0101010101.
- M1 accepts any string that ends with a 1.
- It also accepts 100, 0100, 110000, and 0101000000, and any string that ends with an even number of 0s following the last 1.
- But It rejects other strings, such as 0, 10, 101000.



Formal definition of Finite Automata

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Note: Finite automaton can have more than one accepting states

Formal definition of Finite Automata- Cont'd…

- Despite using State Diagrams to introduce Finite Automata, we also need to use formal definition for two reasons:

1. Its is certain, i.e It resolves any uncertainty about what is allowed in a Finite Automaton.
2. It provides notations, which help to express your thoughts clearly.

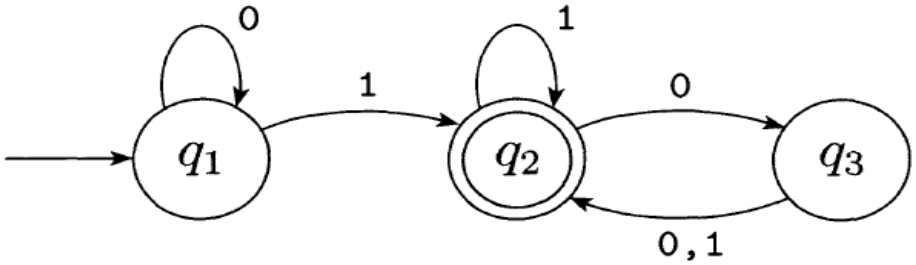
- **Now, to describe M1 formally**

$$M_1 = (Q, \Sigma, \delta, q_1, F)$$

1. $Q = \{q_1, q_2, q_3\}$,

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

3. δ is described as



	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and

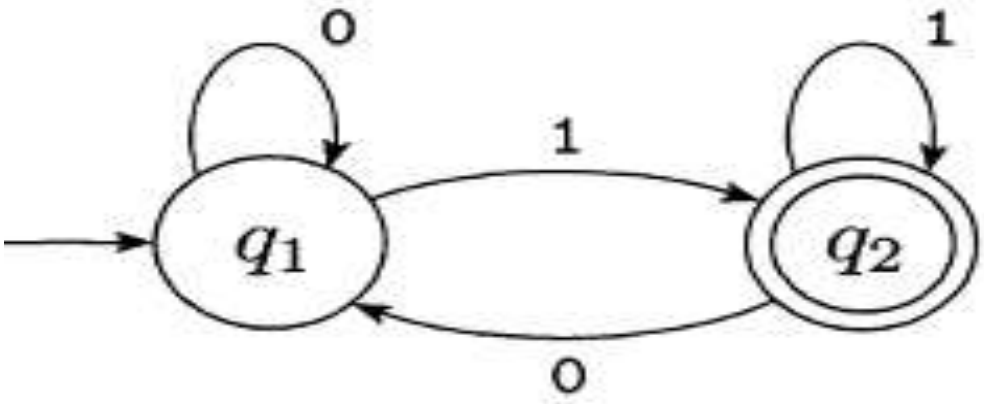
5. $F = \{q_2\}$.

- If A is the set of all strings that machine M accepts, we say that A is the language of machine M, and write L(M)=A.
- A machine can only recognize one language which may comprised of several strings. If the machine accepts no strings, then we say it accepts the empty language $\emptyset$.

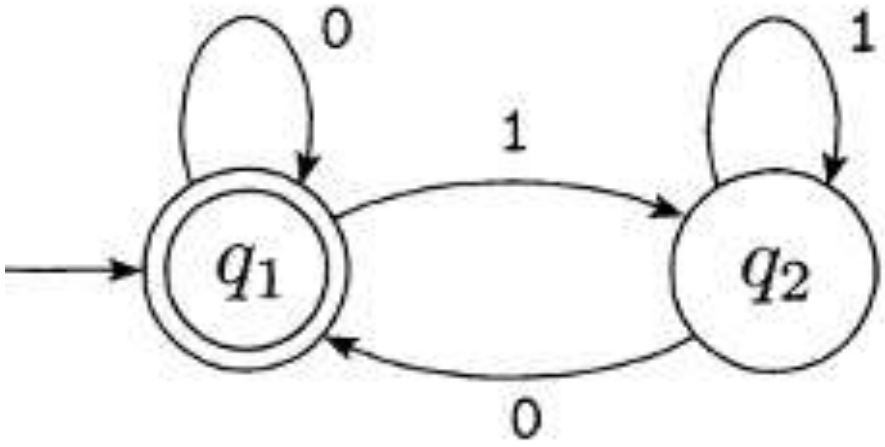
Exercise

Define the following state diagrams formally:

- Example 1



- Example 2



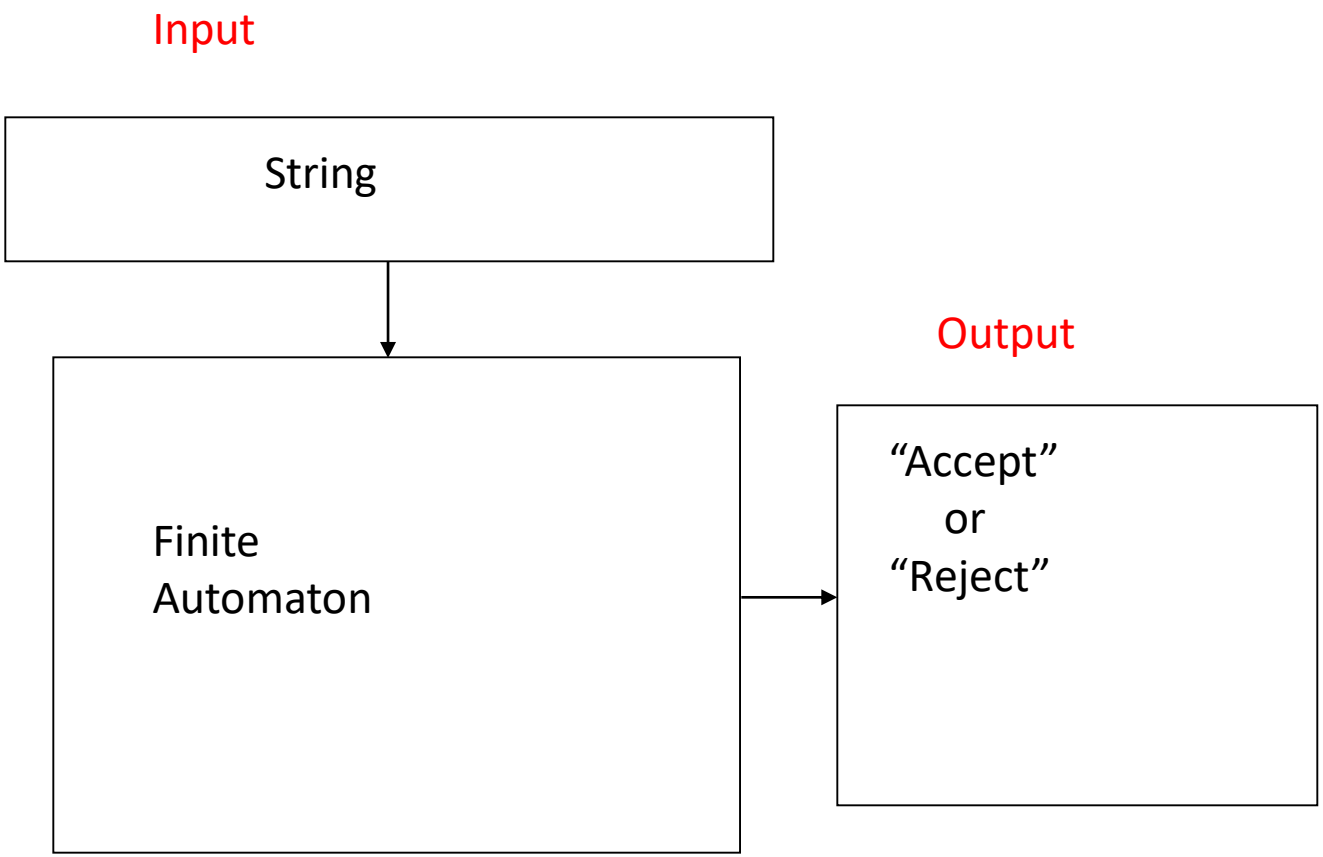
نوع الأول من Finite Automata

Deterministic Finite Automata (DFA)

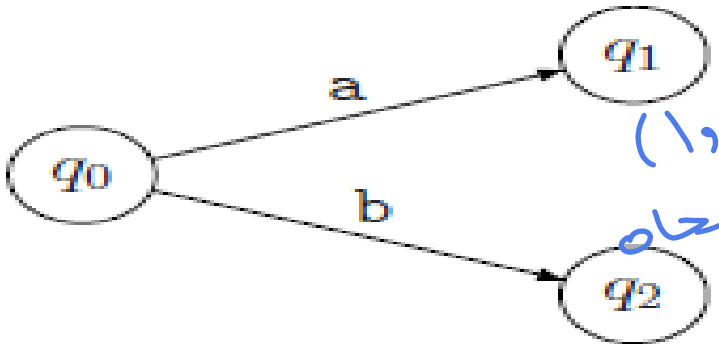
- Also known as **deterministic finite state machine**— is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation of the automaton for each input string.

- Each node in a deterministic machine has exactly one outgoing transition for each character in the alphabet.

Deterministic Finite Automaton (DFA)



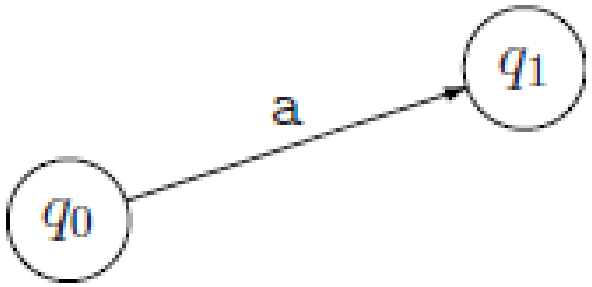
Let ∑= {a,b} be an input alphabet, then all nodes in a DFA for this alphabet must look like the this.



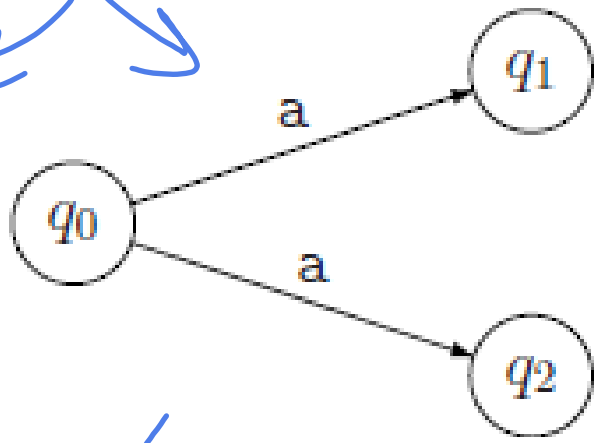
*يجب ان يكون لكل State (a او b) 1 او (1,0)

*ما يطلع اكثر من بعض الحرف والاتجاه

While these are incorrect DFAs

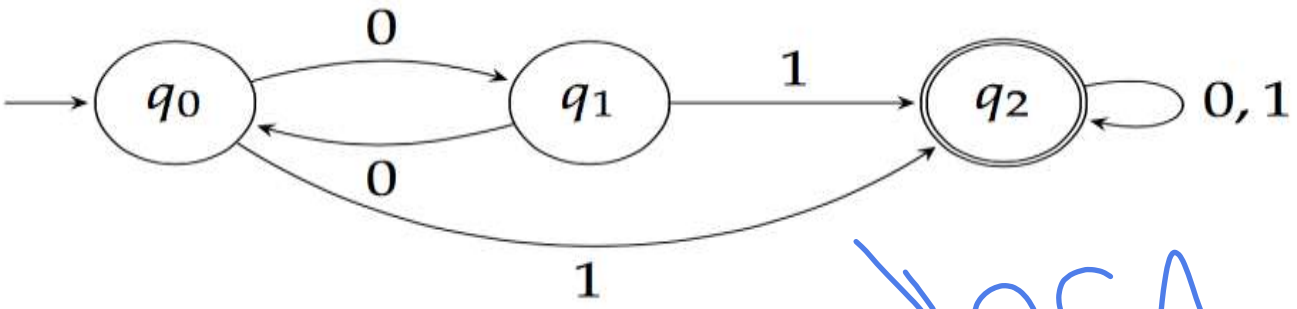


لا يوجد منه b

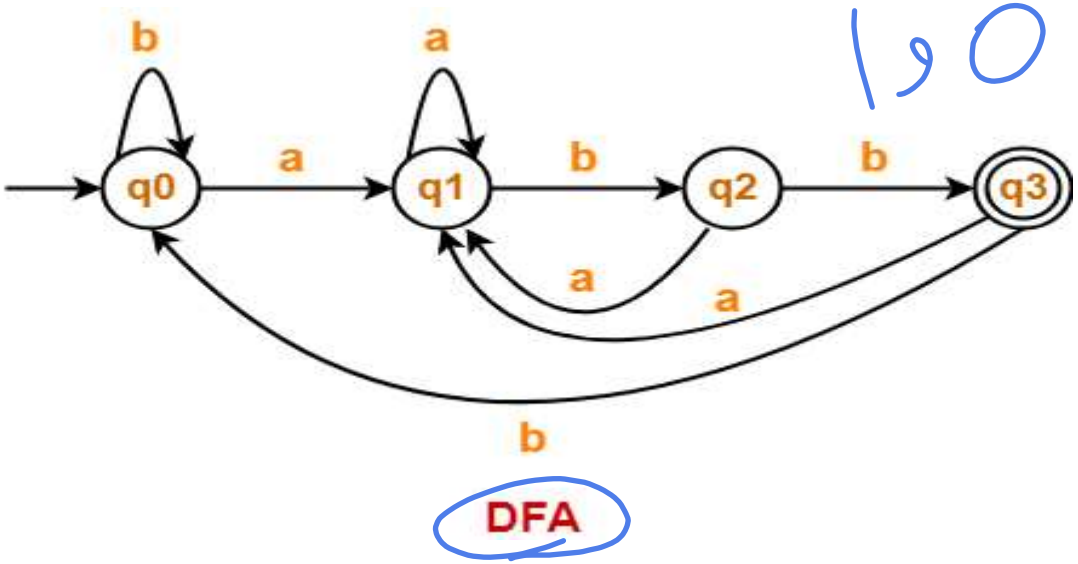


نفس الحرف مكرر

Examples



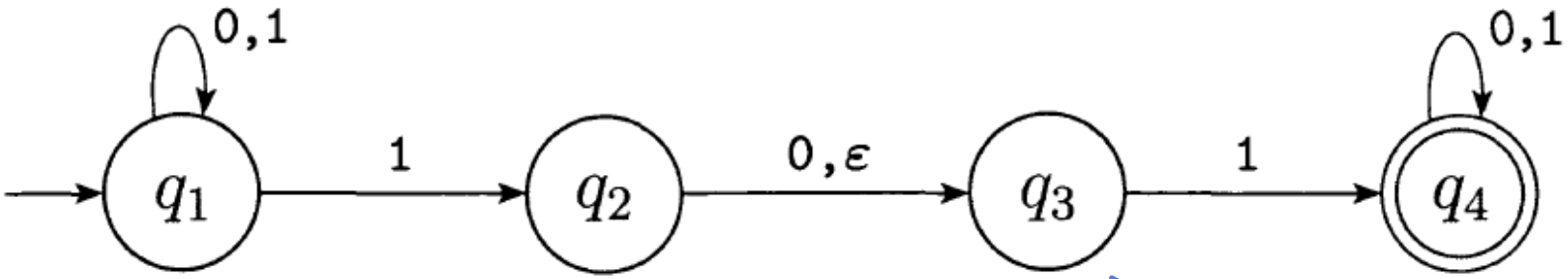
DFA



Nondeterministic Finite Automata (NFA)

- A NDF can have transitions to more than one state on the same symbol.
- It can also contain **ϵ** –transition, means it can jump from one state to another without reading any symbols.
- Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. DFA is subset of NFA.

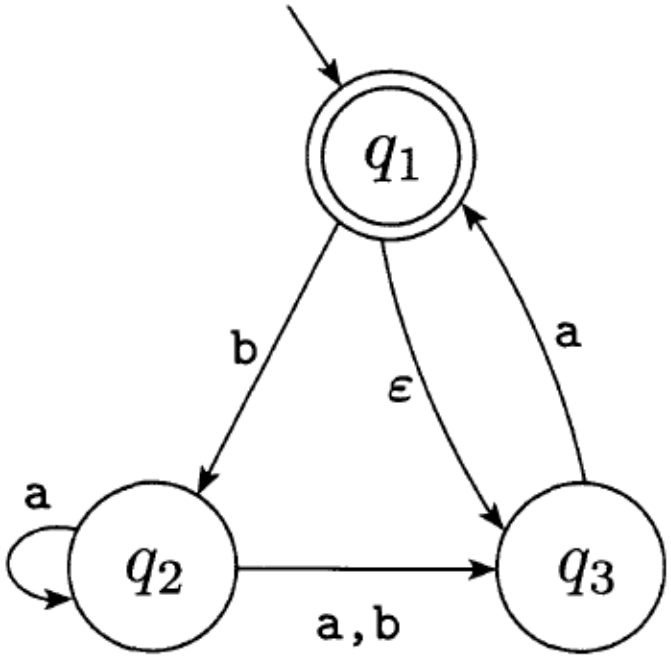
Example 1

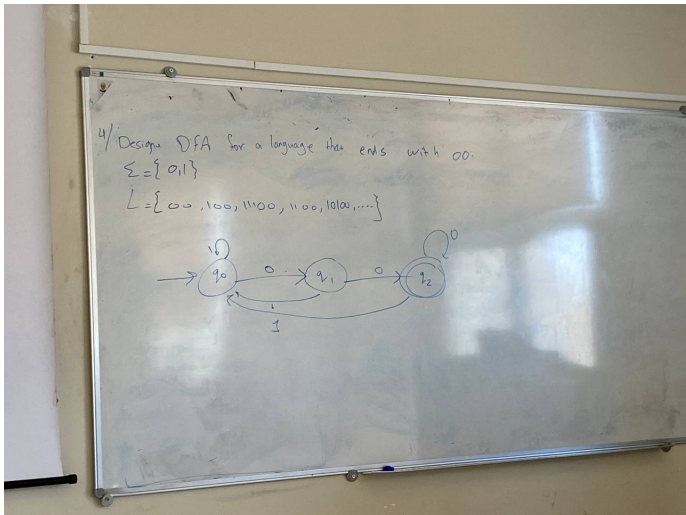


$q_1 \xrightarrow{0} q_1 \;/\; q_1 \xrightarrow{1} q_2 \;/\; q_2 \xrightarrow{1} q_4 \;/\; q_4 \xrightarrow{0} q_4$

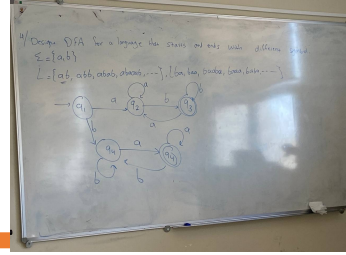
Example 2

- Accepts ϵ, a, baba, and baa. But doesn't accept b, bb.



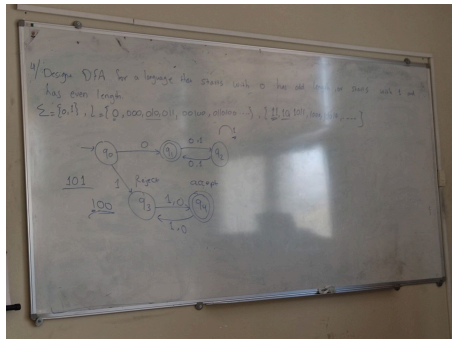


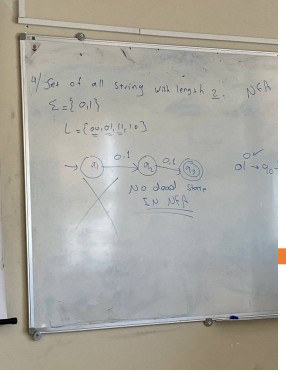
Difference between DFA and NFA

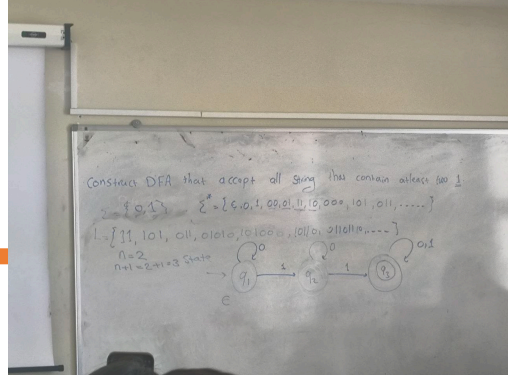


- When processing a string in a DFA, there is always a unique state to go next when each character is read, but NFA breaks this rule.

- In an NFA, several choice (or no choice) may exist for the next state
  - Can move to more than 1 states, or nowhere
  - Can move to a state without reading anything

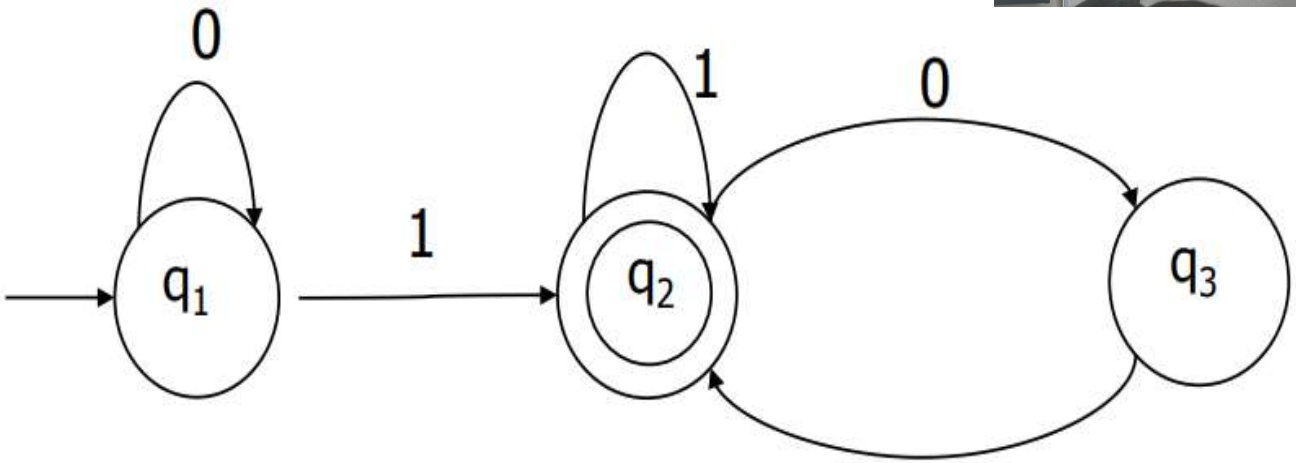




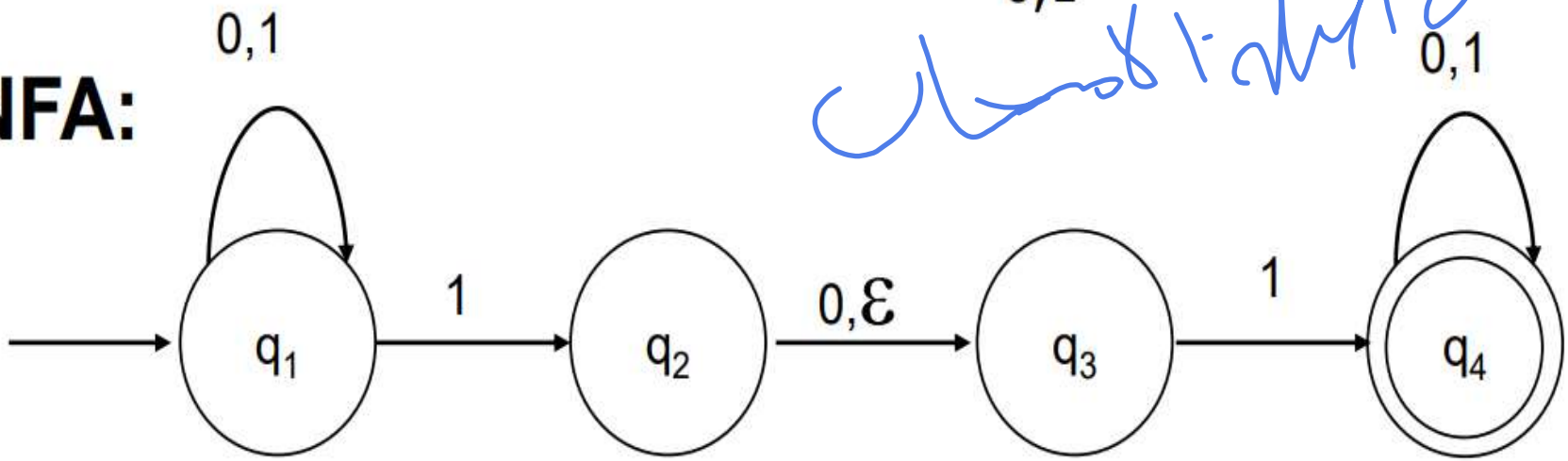


Example of DFA vs. NFA

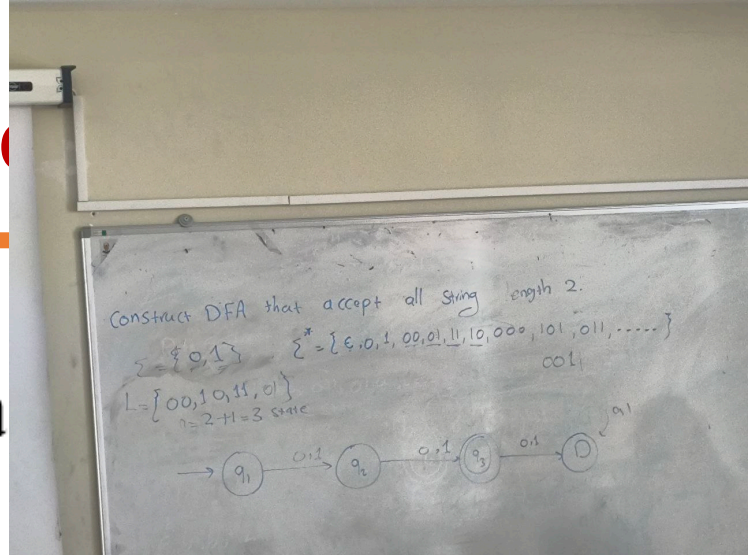
DFA:



NFA:

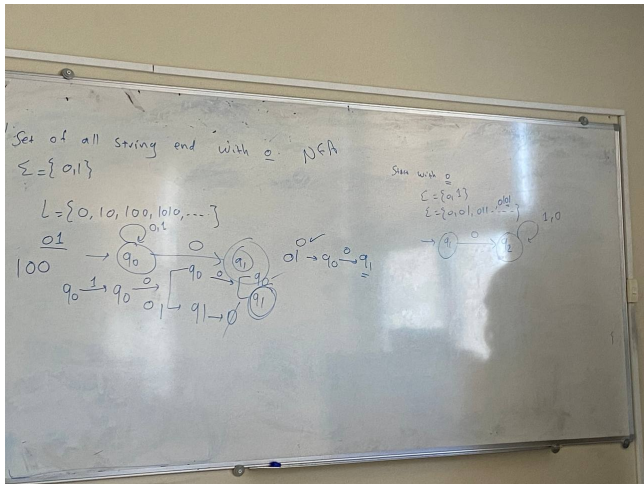


Formal definition



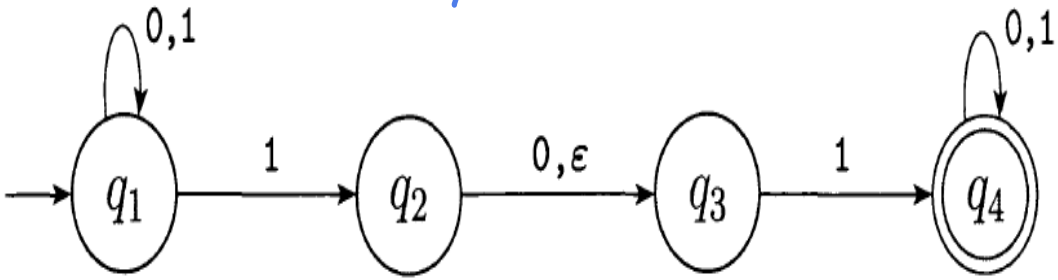
A ***nondeterministic finite automaton*** is a
where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.



Formal definition of example 1

N_1 is $(Q, \Sigma, \delta, q_1, F)$



1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

,

4. q_1 is the start state, and
5. $F = \{q_4\}$.

Contract DFA that accept all string starts with '0'?

$\Sigma = \{0,1\}$ $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 11, 10, 000, \ldots\}$

$L = \{0, 00, 01, 000, 011, 010 \ldots\}$

Regular Languages

- A language is called a **regular language** if it is recognized by some finite automaton.

- Both deterministic and nondeterministic finite automat a are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the regular languages, that regular expressions can describe.

Equivalence of NFAs and DFAs

- NFA and DFA recognize the same class of language. This is **surprising** and **useful** at the same time. **Surprising**, because NFA seems to have more power, so we it is expected to recognize more languages. **Useful**, because some times describing a NFA for a given language is easier than describing DFA for the same language.

- So, two machines are equivalent if they recognize the same language.

NFA to DFA Conversion

- Every DFA is NFA but not vice versa. But there is an equivalent DFA for every NFA it means that NFA can be converted to DFA.

Given: Non Deterministic Finite State Machine

Goal: Convert to an equivalent Deterministic Finite State Machine

Why: faster recognizer

Steps of Converting NFA to DFA Conversion

Step 1:

- Let Q' be a new set of states of the DFA. Q' is null in the starting.
- Let T' be a new transition table of the DFA.

Step 2:

- Add start state of the NFA to Q'.
- Add transitions of the start state to the transition table T'.
- If start state makes transition to multiple states for some input alphabet, then treat those multiple states as a single state in the DFA.

Note: In NFA, if the transition of start state over some input alphabet is null, then perform the transition of start state over that input alphabet to a dead state in the DFA.

Step 3:

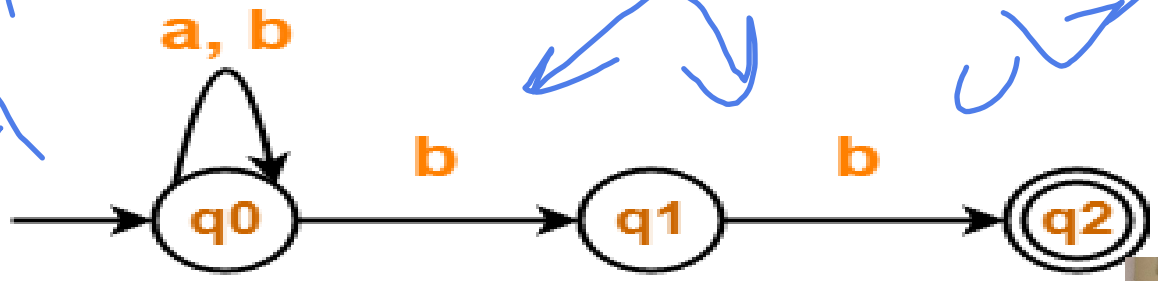
If any new state is present in the transition table T',
- Add the new state in Q'.
- Add transitions of that state in the transition table T'.

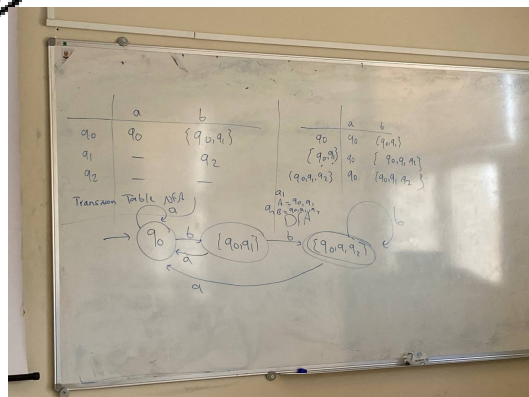
Step 4:

Keep repeating Step 3 until no new state is present in the transition table T'.
Finally, the transition table T' so obtained is the complete transition table of the required DFA.

Example 1

Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA).





Solution

Transition table for the given Non-Deterministic Finite Automata (NFA) is:

State / Alphabet	a	b
→q0	q0	q0, q1
q1	–	*q2
*q2	–	–

Step 1:

Let Q’ be a new set of states of the Deterministic Finite Automata (DFA).
Let T’ be a new transition table of the DFA.

Step 2:

Add transitions of start state q0 to the transition table T’.

State / Alphabet	a	b
→q0	q0	{q0, q1}

Step 3:

New state present in state Q’ is {q0, q1}.
Add transitions for set of states {q0, q1} to the transition table T’.

State / Alphabet	a	b
→q0	q0	{q0, q1}
{q0, q1}	q0	{q0, q1, q2}

Step 4:

New state present in state Q’ is {q0, q1, q2}.
Add transitions for set of states {q0, q1, q2} to the transition table T’.

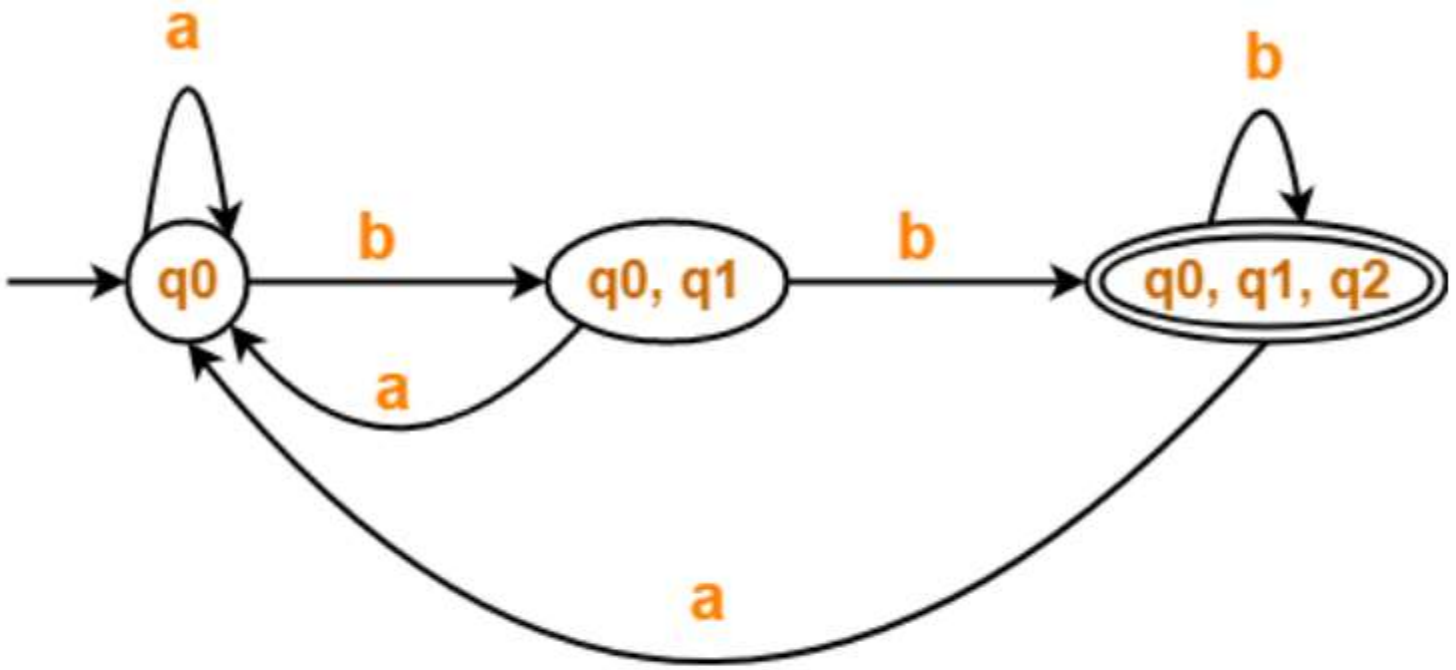
State / Alphabet	a	b
→q0	q0	{q0, q1}
{q0, q1}	q0	{q0, q1, q2}
{q0, q1, q2}	q0	{q0, q1, q2}

Step 5:

Since no new states are left to be added in the transition table T', so we stop.
States containing q2 as its component are treated as final states of the DFA.
Finally, Transition table for Deterministic Finite Automata (DFA) is:

State / Alphabet	a	b
→q0	q0	{q0, q1}
{q0, q1}	q0	*{q0, q1, q2}
*{q0, q1, q2}	q0	*{q0, q1, q2}

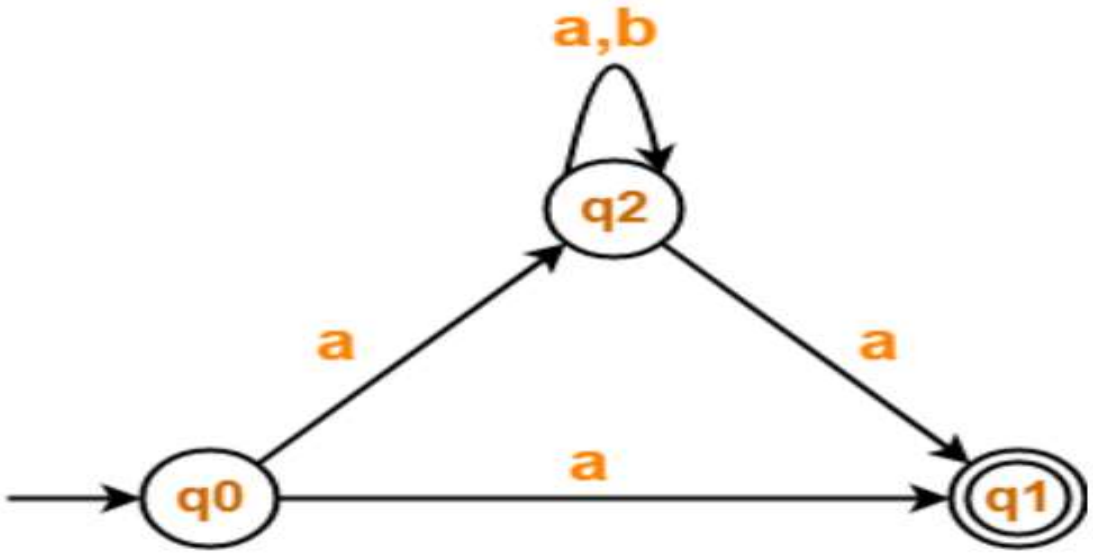
Now, Deterministic Finite Automata (DFA) is drawn as:



Deterministic Finite Automata (DFA)

Example 2

Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA).



Solution

Transition table for the given Non-Deterministic Finite Automata (NFA) is:

State / Alphabet	a	b
→q0	*q1, q2	–
*q1	–	–
q2	*q1, q2	q2

Step 1:

Let Q’ be a new set of states of the Deterministic Finite Automata (DFA).
Let T’ be a new transition table of the DFA.

Step 2:

Add transitions of start state q0 to the transition table T’.

State / Alphabet	a	b
→q0	{q1, q2}	Ø (Dead State)

Step 3:

New state present in state Q’ is {q1, q2}.
Add transitions for set of states {q1, q2} to the transition table T’.

State / Alphabet	a	b
→q0	{q1, q2}	∅
{q1, q2}	{q1, q2}	q2

Step 4:

New state present in state Q’ is q2.
Add transitions for state q2 to the transition table T’.

State / Alphabet	a	b
→q0	{q1, q2}	Ø
{q1, q2}	{q1, q2}	q2
q2	{q1, q2}	q2

Step 5:

Add transitions for dead state {Ø} to the transition table T’.

State / Alphabet	a	b
→q0	{q1, q2}	Ø
{q1, q2}	{q1, q2}	q2
q2	{q1, q2}	q2
Ø	Ø	Ø

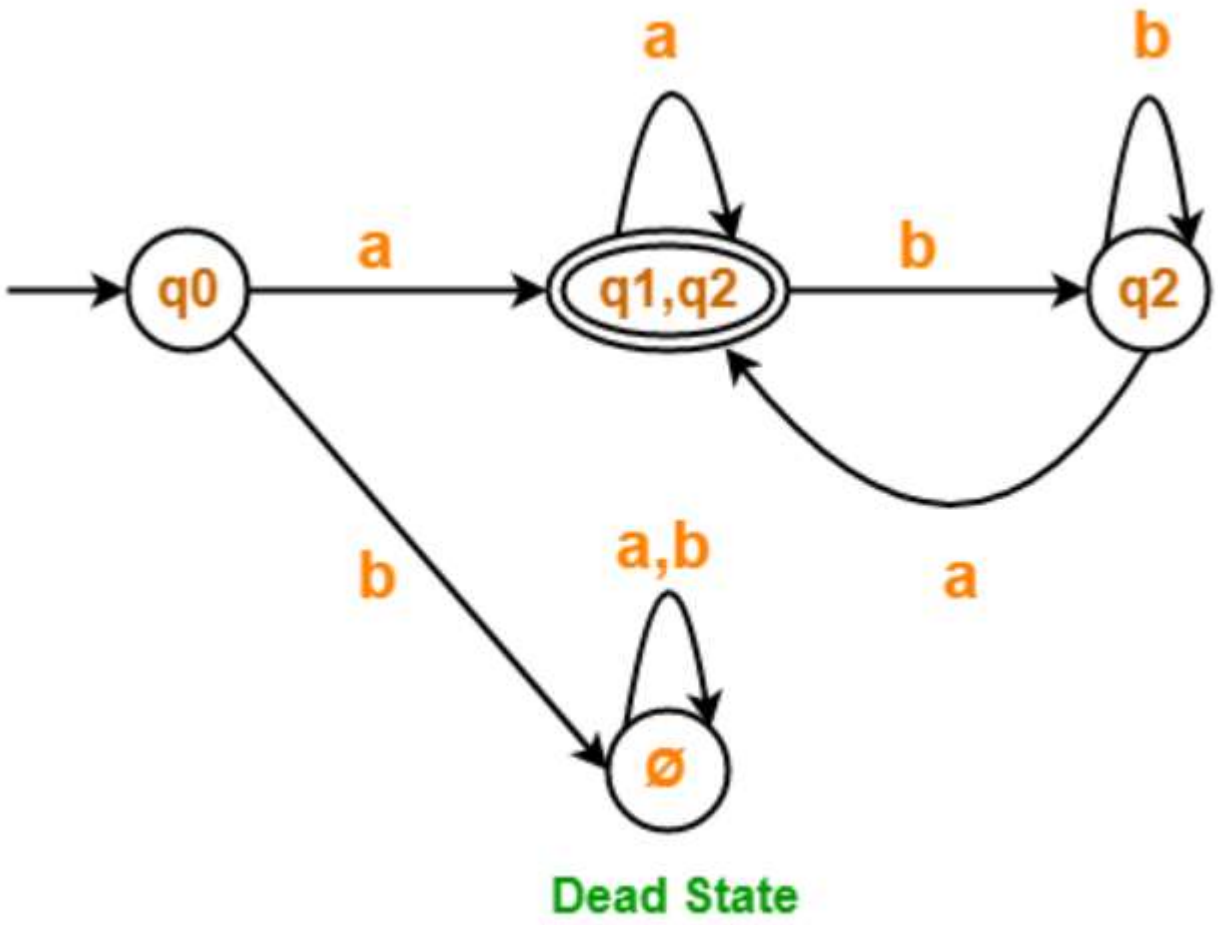
Step 6:

Since no new states are left to be added in the transition table T', so we stop.
States containing q1 as its component are treated as final states of the DFA.

Finally, Transition table for Deterministic Finite Automata is:

State / Alphabet	a	b
→q0	*{q1, q2}	Ø
***{q1, q2}**	*{q1, q2}	q2
q2	*{q1, q2}	q2
Ø	Ø	Ø

Now, Deterministic Finite Automata (DFA) may be drawn as:



Deterministic Finite Automata (DFA)

Notes

- After conversion, the number of states in the resulting DFA may or may not be same as NFA.
  The maximum number of states that may be present in the DFA are $2^{\text{Number of states in the NFA}}$.

- In general, the following relationship exists between the number of states in the NFA and DFA:

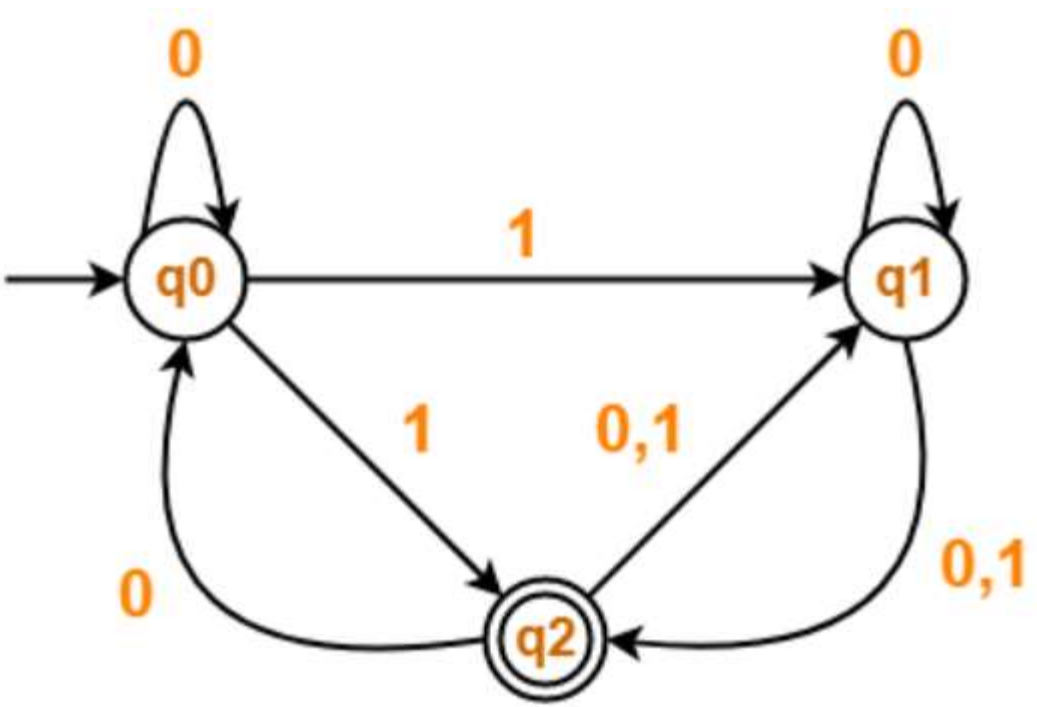
$$1 <= n <= 2^m$$

  n = Number of states in the DFA
  m = Number of states in the NFA

- In the resulting DFA, all those states that contain the final state(s) of NFA are treated as final states.

Homework

Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)



NFA to DFA conversion with epsilon - closure

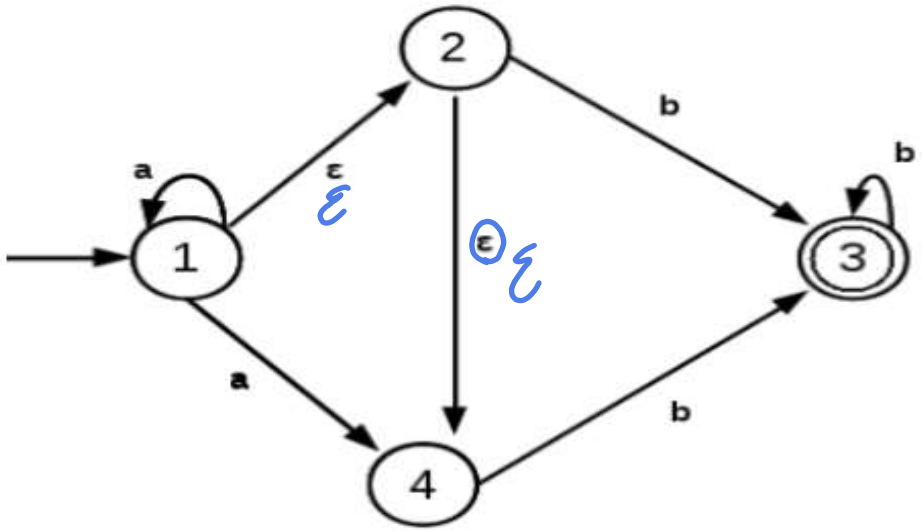
For converting an NFA to DFA, we first need to learn how to find the epsilon closure of a state.

Epsilon closure (ε- closure) of a state q is the set that contains the state q itself and all other states that can be reached from state q by following ε transitions.

We use the term, ECLOSE(q_O) to denote the Epsilon closure of state q_O.

Example1

Consider the following NFA,



Find the epsilon closure of all states.

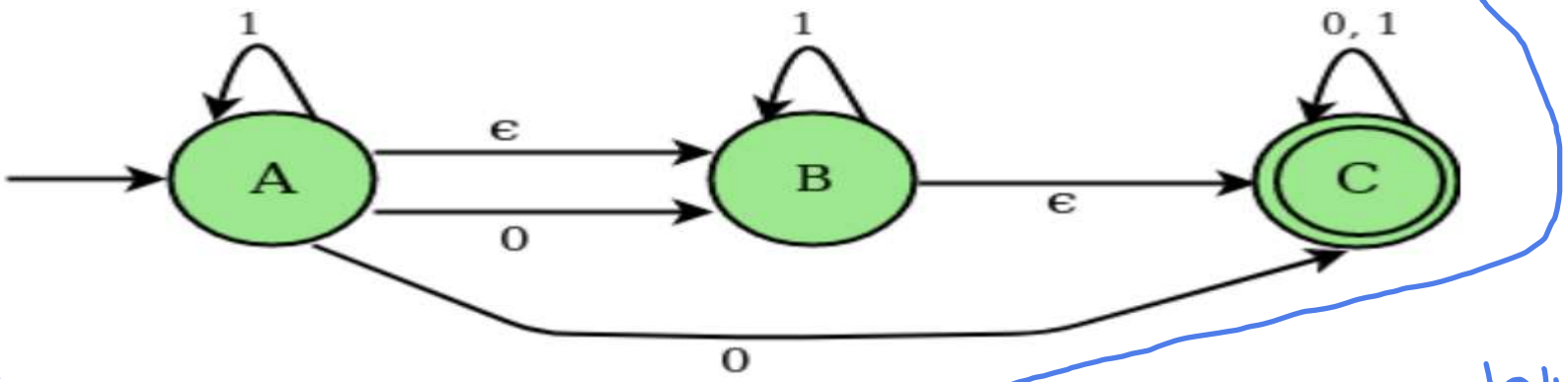
ECLOSE(1) = {1, 2, 4}

ECLOSE(2) = {2, 4}

ECLOSE(3) = {3}

ECLOSE(4) = {4}

Example2 (NFA to DFA conversion with epsilon)




```
∈ closure(A) : {A, B, C}
∈ closure(B) : {B, C}
∈ closure(C) : {C}
```

هذه الطريقة سنستخدمها عندما يوجد

State/ Alphabet	0	1
{A,B,C}	{B,C}	{A,B,C}
{B,C}	{c}	{B,C}
{C}	{c}	{c}

DFA STATE DIAGRAM