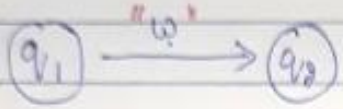
"INTRODUCTION TO NFA"

"Non-Deterministic Finite-Automata":-

The difference b/w "DFA" & "NFA" is that if we are starting from a state on seeing some input say 'w' we goto exactly one state in DFA whereas in "NFA" reverse is also possible.



{ DFA }

whereas in "NFA" on seeing an input might end up in more than "One state" or we might either end-up with nothing.

"Non-determinism means":- On seeing an input we can either go to nothing or we might also end up in more than one state. or we can end up in same state.

In non-deterministic machines many option can be formed. &

Non-deterministic machines are not real, it means no real machine can actually be non-deterministic.

Still we are studying "Non-deterministic machines" because.

Date __/__/__
Page _____

mycompanion

Sometimes, some of the problems can be solved by exhaustive search. It means we have to search for all possible solutions and backtracks.

Note:- "Exhaustive - search" & "backtracking" are the methods, which can give us "better results."

"Non-determinism" is very important if we have to perform "exhaustive search & backtracking"

& "Determinism" is important, if we don't want to do any "backtracking"

Note:- "Non-deterministic Machine" can be implemented using "Deterministic machine" but with difficulty.

→ NFA {Non-Deterministic Finite Automata} :- NFA is a quintuple which is a five-tuple set as shown below:-

$$(Q, \Sigma, \delta, q_0 \& F)$$

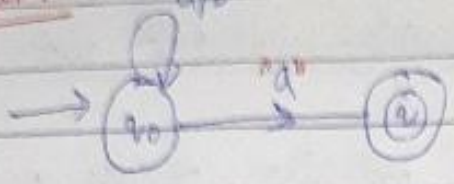
where "Q" -> "Set of states" {finite}
"Σ" -> "input alphabet" {finite}
"q_0" -> "start state"
"F" -> "A set of final states"

Now important point is "δ" Now in order to explain "δ" we will take an example of "NFA":-

Construct an "NFA" which accepts a language such that every string ends with a "a" & let $\Sigma = \{a, b\}$

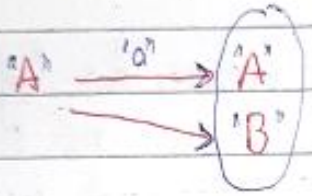
P.T.O

NFA is as :→



string can start with anything but it has to end with an 'a'.

$L = \{a, aa, ba, \dots\}$ infinite.

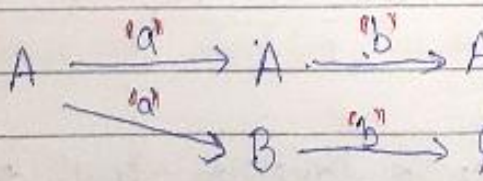


"Non determinism" means on giving an input on one state we can go to more than one state or we might even go to no state at all.

Now, we say that, the string 'a' is accepted by "NFA", if we start from an initial state & if we reach some state, out of which atleast one state is a "final-state".

So, if we reach atleast one 'final state' starting from an "initial-state" on reading a string, we can say that the string is accepted by "NFA".

Now, let us say that the string is "ab" :→



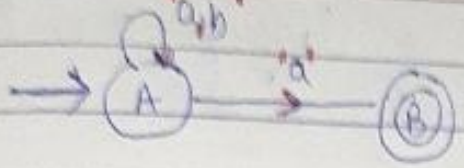
{dead configuration}

So, 'A' is a 'non-accepting state' hence ab is rejected by "NFA".

Date __/__/__
Page

mycompanion

Q. Let us draw transition (state transition) diagram for given NFA :->



δ' is a transition function which will take a state 'A' & take an input symbol from [Σ] & can go to either "one state" or "zero state" or "more than one state".

It is as defined :->

$\delta: Q \times \Sigma \longrightarrow 2^Q$ {any no. of states}

↳ {any subset of states possible}

$Q = \{A, B\}$ & $\Sigma = \{a, b\}$

Now "$Q \times \Sigma$" ↓

$= [A,a \;,\; B,a]$

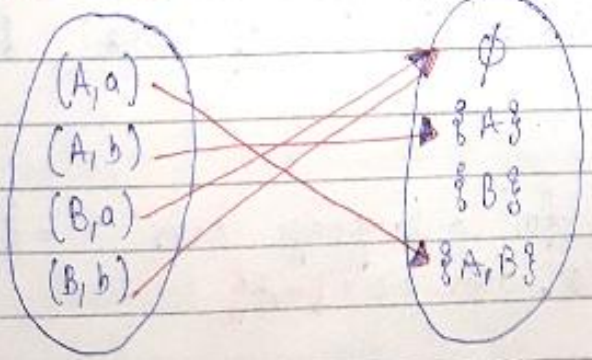
$= [A,b \;,\; B,b]$

Now 2^Q ↓ (Power set of Q)

$(\phi, \{A\}, \{B\}, \{AB\})$ ↓

{"set of all possible subsets"}

[$Q \times \Sigma$] [2^Q]



This is the main difference b/w "NFA" & "DFA".

"DFA" $\delta: Q \times \Sigma \longrightarrow Q$ (only one state)

"NFA" $\delta: Q \times \Sigma \longrightarrow 2^Q$ (some subset of possible states)

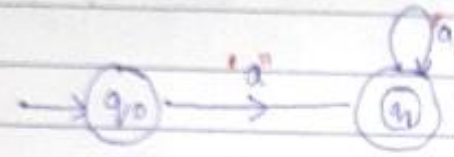
P.T.O.

Note :- In "NFA", we might have a "dead-configuration" but in "DFA" it is not present, instead of that below a "dead state" or "trap state".

→ "Examples of "NFA" & "Non-Deterministic Finite Automata"

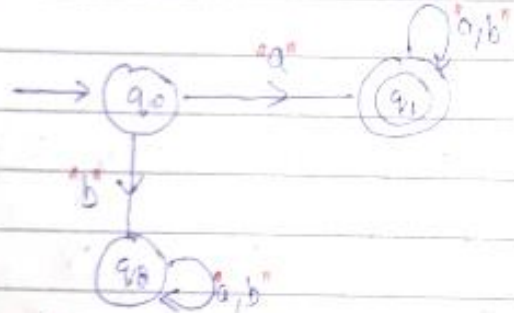
"Example1" :- Construct an NFA for a language over $\Sigma = \{a,b\}$ such that every string starts with 'a'

$L_1 = \{$ starts with 'a' $\}$



"NFA" but not a "DFA" as it is not complete.

Below "DFA" for the same problem is as shown below :->

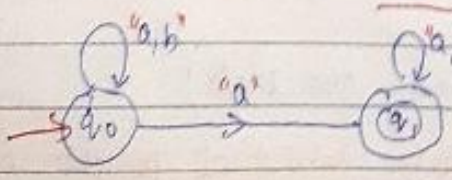


$\{$ Dead-state $\}$

So, the major difference between "DFA" & "NFA" is that in "DFA" we can have a dead-state, whereas in case of "NFA", dead-configuration can be "possible".

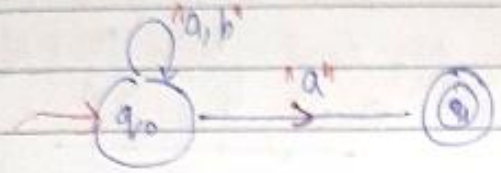
"Example" :- Construct an NFA for a language over $\Sigma = \{a,b\}$ such that every string contains an 'a'. over $\Sigma = \{a,b\}$.

$L_2 = \{$ contains an 'a' $\}$



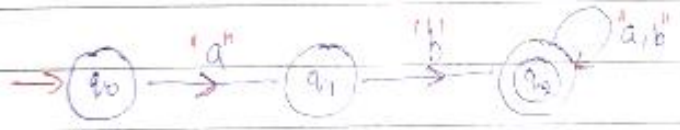
"Example 3" :-> Construct an NFA for a language over $\Sigma = \{a, b\}$ such that each string ends with an 'a'.

$L_3 = \{$ ends with an 'a' $\}$



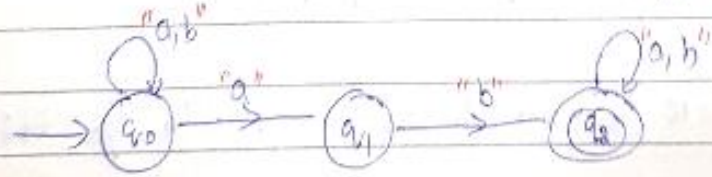
"Example 4" :-> Construct an NFA for a language over $\Sigma = \{a, b\}$ such that every string starts with "ab".

$L_4 = \{$ starts with "ab" $\}$



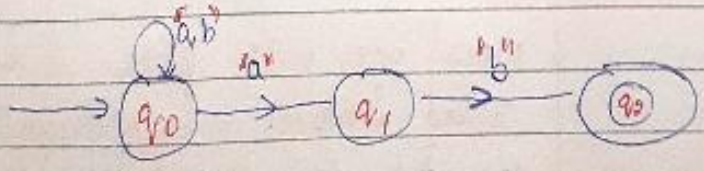
"Example 5" :-> Construct an "NFA" for a language over $\{\Sigma = \{a, b\}\}$ such that every string contains "ab" as a substring.

$L_5 = \{$ Contains "ab" as a substring $\}$.



"Example 6" :-> Construct an "NFA" for a language over $[\Sigma = \{a, b\}]$ such that every string ends with an 'ab'.

$L_6 = \{$ ends with an "ab" $\}$



P.T.O

Construction of 'NFA' is very much easier & simpler than construction of 'DFA'.

"Conversion of 'NFA' to 'DFA'" :=>

Note:=> "Both NFA & DFA are equally powerful"

'NFA' → 'DFA' (Convertible)
& 'DFA' → 'NFA'

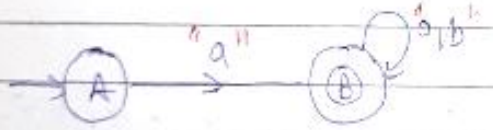
If we are able to do these things we can say that both 'NFA' & 'DFA' are equally powerful.

∴ 'NFA' ≅ 'DFA'

Let us now see an example to convert 'NFA' into 'DFA' :=>

Eg:- 'L' = { starts with an 'a' } & $\Sigma = \{a, b\}$

then NFA is



Now we want to convert this 'NFA' into 'DFA' :=>

There are many methods to convert an 'NFA' into a 'DFA' but we will follow "subset construction" method

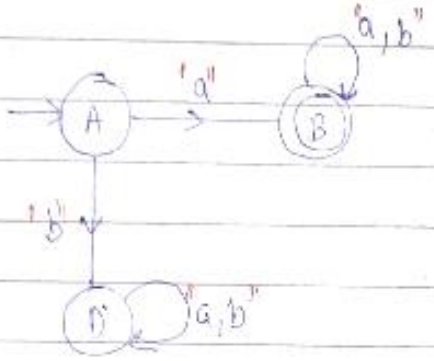
Now let us first draw state transition table for this 'NFA'

	a	b
A	B	ϕ
B	B	B

Now we will follow a method called "subset - construction" to convert this 'NFA' into 'DFA'

Note :- ['∅' is nothing but dead-config in "NFA" & corresponding to it we have dead-state in "DFA"]

	"a"	"b"
→A	{B}	{D} → new state
*B	{B}	{B}
D	{D}	{D}] "dead state"

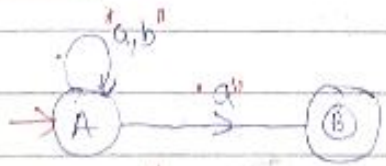


"DFA"

Q:→ Conversion of "NFA" to "DFA" for the example "all strings ends with a".

$L_1 = \{$ ends with an 'a' $\}$ & $\Sigma = \{a, b\}$

Now first "NFA" is :→



Let us now construct "state-transition table" for this "NFA" :→

	"a"	"b"
→A	{"A,B"}	{A}
*B	∅	∅

Now in order to construct "DFA", we should start with the "initial-state" & then construct the transition table.

"P.T.O"

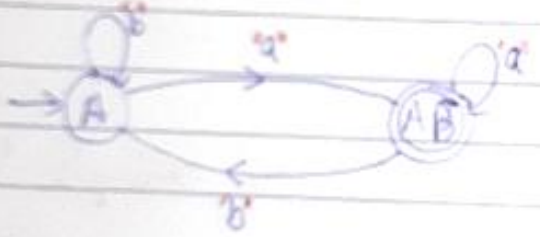
Note:- [AB] ⇒ It is a new state which is a 'Dfa' state.

"Dfa"	"a"	"b"
→[A]	[AB]	[φ]
*[AB]	[AB]	[A]

(Now look for 'A' & 'B' in original table & perform union for [AB] to get the states)

If there are two states in 'DFA' then go to both the states in 'NFA' & take "union".

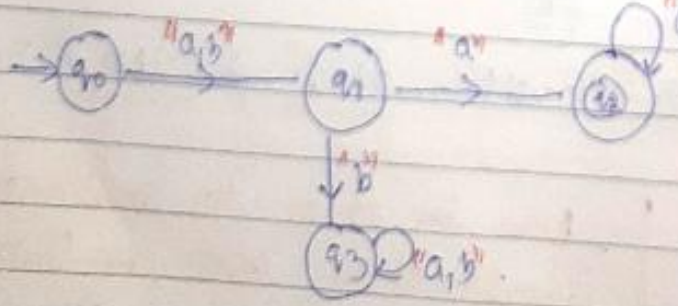
Take "states" only which are "reachable".



Ex:- Conversion of 'NFA' to 'DFA' for the example "all strings in which second symbol from R.H.S is "a"

Eg:- "second symbol" from L.H.S is 'a' over $\Sigma = \{a, b\}$

DFA:-

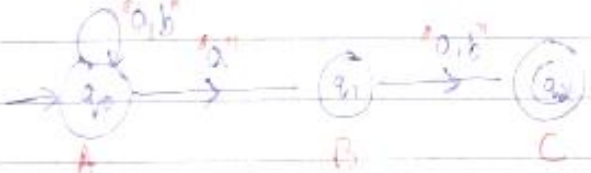


Now NFA is ↗

Now Given Question is that:- 2nd symbol from 'R.H.S' is "a" i.e

Constructing "DFA" for this problem would be a bit complex. So instead of directly constructing a "DFA" we will create an "NFA" first.

"NFA":- $L = \{ aa, ab, baa, aaa, \dots \}$



Let us now convert the "NFA" to "DFA".

First let us draw state transition table for "NFA":

"NFA"	a	b
→A	{B, C}	{A}
B	{C}	{C}
*C	ϕ	ϕ

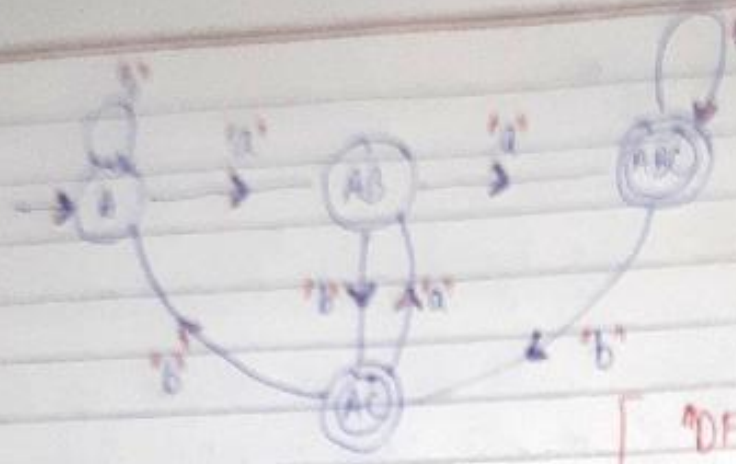
Now let us convert "NFA" transition table to "DFA" transition table.

"DFA"	"a"	"b"
→A	[AB]	[A]
[AB]	[ABC]	[AC]
*[AC]	[AB]	[A]
*[ABC]	[ABC]	[AC]

No-Need

	"a"	"b"

P.T.O

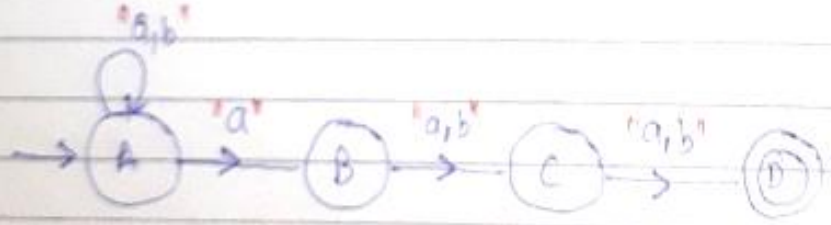


["DFA" required]

Q:→ Conversion of NFA to DFA for the example.

("all strings in which third symbol from R.H.S is a")

Sol:→ Let us first try to construct NFA for this language.

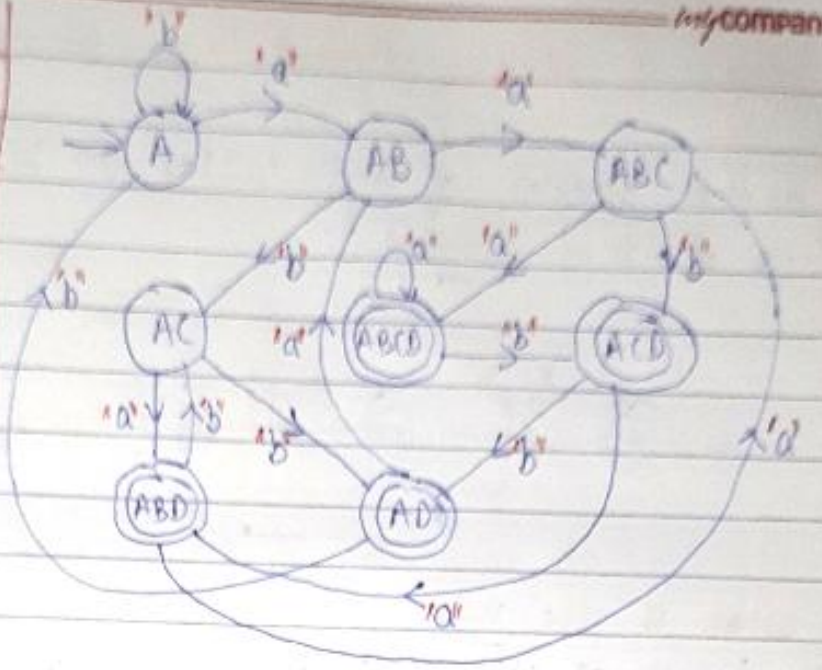


Now "Transition Table" for this NFA is :→

"NFA"	'a'	'b'
→A	{AB}	{A}
B	{C}	{C}
C	{D}	{D}
*D	ϕ	ϕ

[Now let us try to construct "Transition Table" for corresponding "DFA"]

'DFA'	'a'	'b'
→A	[AB]	[A]
[AB]	[ABC]	[AC]
[ABC]	[ABCD]	[ACD]
[AC]	[ABD]	[AD]
* [ABCD]	[ABCD]	[ACD]
* [ACD]	[ABD]	[AD]
* [ABD]	[ABC]	[AC]
* [AD]	[AB]	[A]



→ Examples of NFA's :->

1> 'NFA' for strings of length exactly '2'.

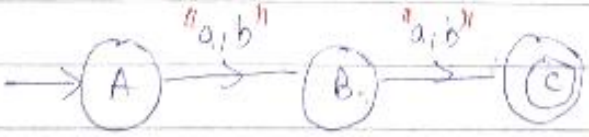
2> 'NFA' for strings of length at most '2'.

3> 'NFA' for strings of length atleast '2'

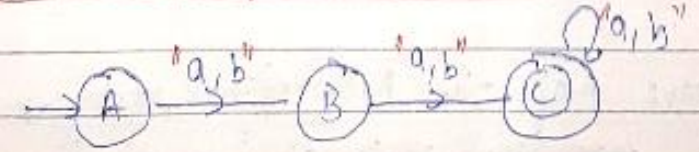
$[\Sigma = \{a, b\}]$

Sol:> 1. "Strings of length exactly '2'"

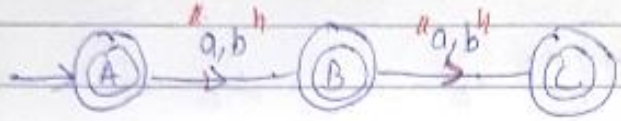
$L = \{aa, ab, ba, bb\}$



2> "Strings of length atleast "2""



3> Strings of length at most '2' :> It means string of length "0" has to be accepted, of length "1" has to be accepted & length "2" has also to be accepted.



Q:> What is the min. no. of states that are present in "NFA" if the length of string is 'n' or atleast 'n' or at most "n".

Ans:> ['n' states are required]. "for NFA"

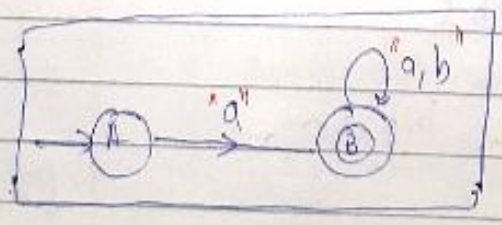
"COMPLEMENTATION OF NFA" :>

In "DFA" if we complement a "DFA", the language accepted by "DFA" is getting complemented. but in case of "NFA" it is not true.

Eg:> NFA, for $\Sigma = \{a, b\}$

"L" = { set of all strings with 'a' } = $\{a, aa, ab, \dots\}$

"NFA" is as :>



Now if we complement this "NFA", let's see what happens :>

↓

It means make the Non-final state as final state & make final state as "Non-final"

$\bar{M}$ =

$L_2 = \{ \in \}$

So, here we can find out that $\boxed{L_1 \neq L_2}$

["L_1 complement" is not equal to "L_2"]

Σ^*

"L"	"$\bar{L}$"
Starts with 'a'	"$\in$" + "starts with b"

But here we get only "$\in$" as $\bar{L}$

for "NFA"

$M \Rightarrow \bar{M}$ (NFA)

$L_1 \neq L_2$

Note :- Complementing an "NFA" is different from complementing a "language."

Note:- When we complement a "DFA", the language accepted by "DFA" also gets complemented.

Note2:- When we complement an "NFA", the language accepted by "NFA" need not get "complemented". (may or may not).

Q1> What is the language accepted by complement of NFA?

&

Q2> What is the complement of language accepted by NFA?

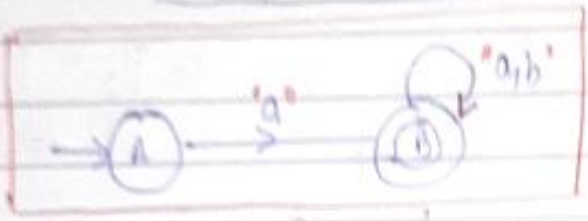
PTO

Aim:

Aim:- Take the 'NFA', find its 'complement' & then find the answer.

Aim:- Take the 'language' accepted by 'NFA', & then find the complement of the 'language'.

Let us assume that our 'NFA' is given as :-



Now suppose the question is:-

Q:- What is the complement of language accepted by the NFA?

So here we need to find the lang. accepted by the NFA & then have to take the complement of that language.

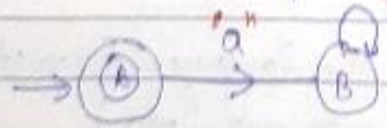
$\therefore L = \{ a, aa, ab, aaa, \ldots \}$ starting with 'a'

Now $\overline{L} = \{ \in, b, bb, ba, bab \ldots \}$

↳ [Complement of above language "L"]

Q:- What is the language accepted by complement of this NFA?

So here we need to take the 'NFA' & complement it..



Now language accepted by complement of this 'NFA' is :-

$L = \{ \in \}$

Date __/__/__
Page _____

mycompanion

"MINIMIZATION OF DFA" :->

Question here is, Can we take any "DFA" & prove that this is minimal 'DFA'

↳ This is known as "Minimization of DFA" :->

So, if a "DFA" is given, then we can apply a procedure using which we can decrease the states of a 'DFA'. This process is known as "Minimization of DFA".

Before going to "Minimization of DFA", let us talk about, what are the properties, we need in a 'DFA' so that it can be minimized :->

So, here we need to talk about "states" & "equivalences b/w the states".

Let us say there are two states (p, q).

we say that these two states are equivalent or these two states can be replaced by a single state if

$$\delta(p, w) \longrightarrow \text{"F"}$$

if 'p' on seeing any input string 'w' goes to final state: then it implies that.

$$\delta(q, w) \longrightarrow \text{'F'}$$

'q' also goes to one of the final states on seeing 'w'.

if $\delta(p, w) \not\longrightarrow$ "F"

if 'p' on seeing an input string 'w' does not go to a "final state" then it implies that

$$\delta(q, w) \not\longrightarrow \text{"F"}$$

So, we can say that, if we have two states let us say "p" & "q". then can we combine them into a single state if both of them are "equivalent".

So we say that (p, q) are equivalents if.

$$[\delta(p, w) \longrightarrow F \Rightarrow \delta(q, w) \longrightarrow F]$$

↓

(need not go to same final state, but can go to any one among the final states.)

or if

$$[\delta(p, w) \not\longrightarrow F \Rightarrow \delta(q, w) \not\longrightarrow F]$$

So, if any two states follow this property, then we can say that such two states are "equivalent".

Note:⇒ if length of 'w' is zero i.e. $|w| = 0$, then both the states 'p & q' are called zero-equivalent.

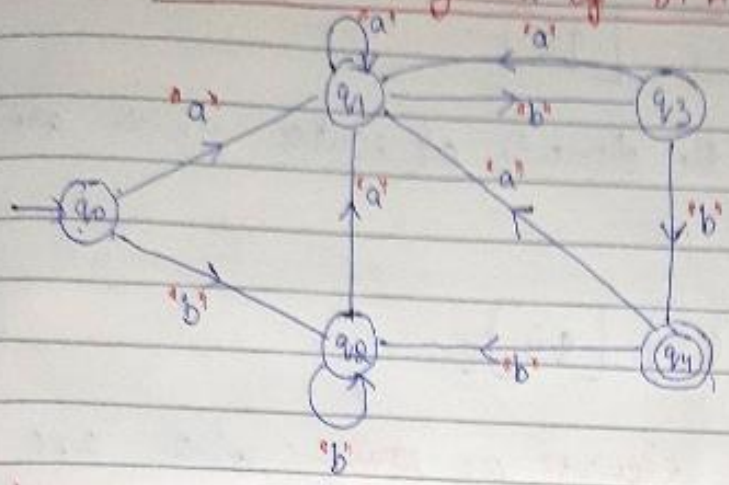
if length of 'w' is one i.e. $|w| = 1$, then both the states 'p & q' are called one-equivalent.

if length of 'w' is two i.e. $|w| = 2$, then both the states 'p & q' are called two-equivalent

So, in general if ⇒

if length of 'w' is 'n' i.e. $|w| = n$, then both the states 'p & q' are n-equivalent.

→ Example of "Minimization of DFA":→



Before we need to minimize this "DFA":→

Step①:→ Identify the "start-state" as well as "final-state".

Step②:→ Try to delete all the states, to which we cannot reach from "initial state".

[Here in this eg. all the states are reachable from initial state]

Step③:→ Draw the "state-Transition" table.

	"a"	"b"
→ "q0"	q1	q2
"q1"	q4	q3
"q2"	q1	q2
"q3"	q1	*q4
*"q4"	q1	q2

Now Try to find out (0-equivalent states) first

0-equivalent:→ Separate out "final states" from "Non final states"

"Non-final" "final"

[q0, q1, q2, q3] [q4]

set of all "final states" & set of all "non-final states"

"0-equivalent" :> $[q_0, q_1, q_2, q_3]$ $[q_4]$

Now we need to find out the elements or states which are one-equivalent to each other.

"1-equivalent" :> $[q_0, q_1, q_2]$ $[q_3]$ $[q_4]$.

Now, we need to find out elements or states which are two-equivalent to each other.

"2-equivalent" :> (It can be found out, by using information from 1-equivalent set.)

"2-equivalent" :> $[q_0, q_2]$, $[q_1]$, $[q_3]$, $[q_4]$

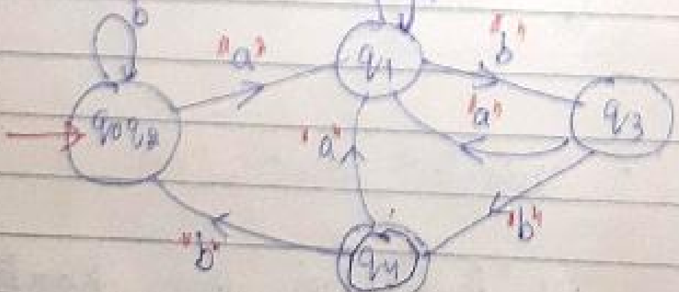
"3-equivalent" :> $[q_0, q_2]$, $[q_1]$, $[q_3]$, $[q_4]$ "final answer"

↓

"cannot be separated"

Continue in this manner until we get "states" like, where nothing will "change".

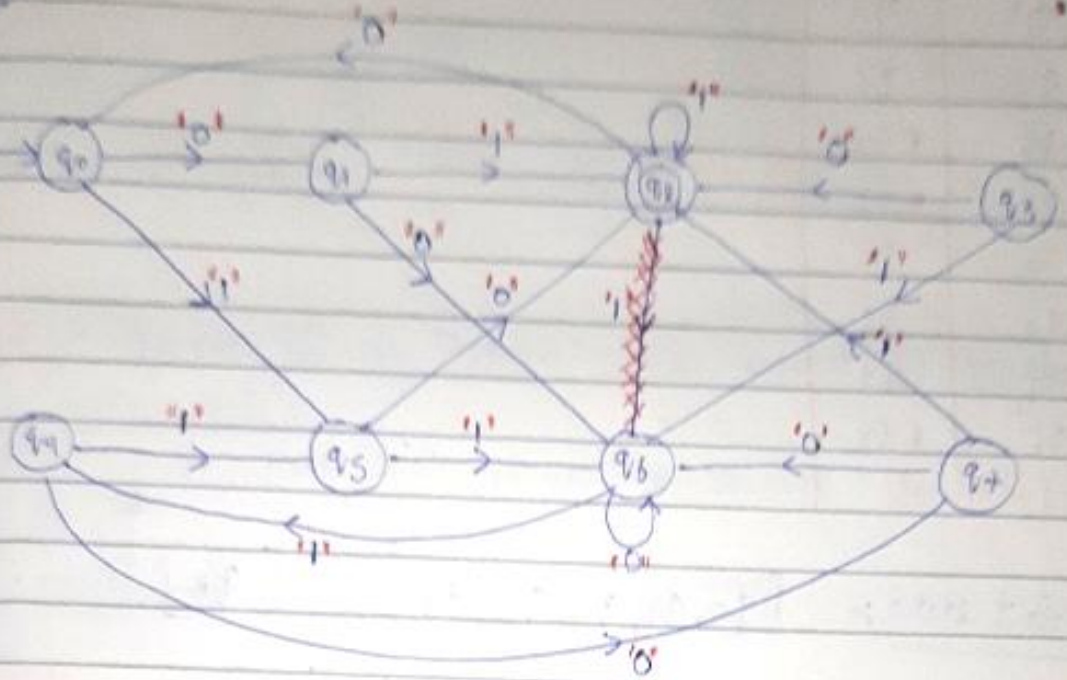
Let us now draw minimized "DFA diagram" :>



—o—

→ Minimization of DFA :→

Example:→



Sol:→ Transition (State-Transition) Table for the "DFA" is :→

	'0'	'1'
→ q_0	q_1	q_5
q_1	q_6	* q_2
* q_2	q_0	* q_2
q_3	* q_2	q_6
q_4	q_7	q_5
q_5	* q_2	q_6
q_6	q_6	q_4
q_7	q_6	* q_2

Step ① :→ We need to remove all those states which cannot be reached from the initial state

Here we can see that "q_3" cannot be reached from initial state "q_0" so "q_3" can be removed along with it transition

So, "transition table" becomes :>

	"0"	"1"
→ q_0	q_1	q_5
q_1	q_6	* q_2
* q_2	q_0	* q_2
q_4	q_7	q_5
q_5	* q_2	q_6
q_6	q_6	q_4
q_7	q_6	* q_2

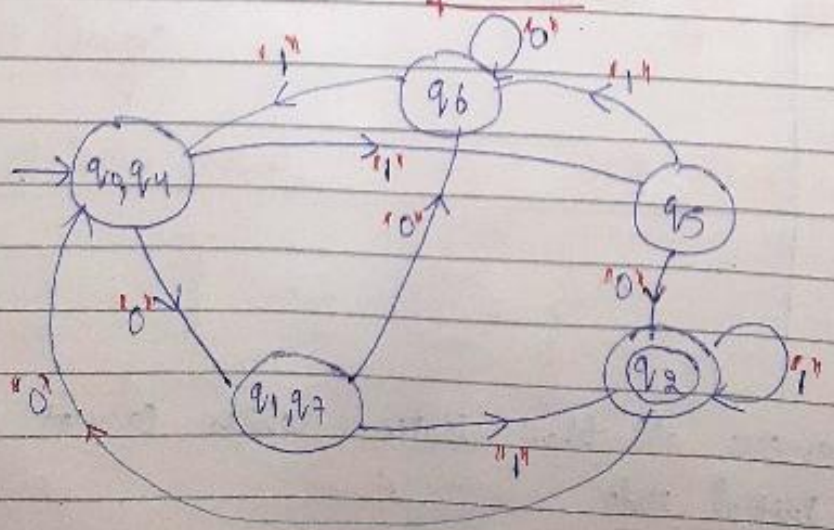
1) 0- Equivalent state :> $[q_0, q_1, q_4, q_5, q_6, q_7]$ $[q_2]$

2) 1- Equivalent states :> $[q_0, q_4, q_6]$ $[q_1, q_7]$ $[q_5]$ $[q_2]$

3) 2- Equivalent states :> $[q_0, q_4]$, $[q_6]$, $[q_1, q_7]$, $[q_5]$ $[q_2]$

4) 3- Equivalent state :> $[q_0, q_4]$, $[q_6]$, $[q_1, q_7]$, $[q_5]$, $[q_2]$
('1') ('2') ('3') ('4') ('5')

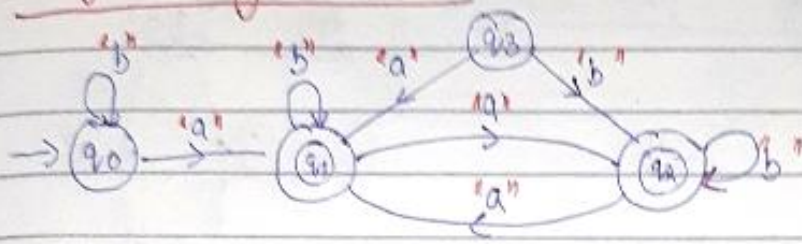
So stop here :>



"This is minimum DFA"

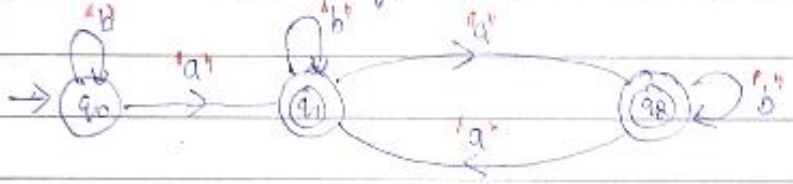
Example ③ :- "GATE - 2003"

Minimize the given "DFA" :-



Step no. ① :- Remove all the "states" which cannot be reached from the "initial-state".

Here "q3" can not be reached so remove it



Transition diagram is as :-

	a	b
→ q0	*q1	q0
*q1	*q2	*q1
*q2	*q1	*q2

① "0-Equivalence" states" :- [q0] [q1, q2]

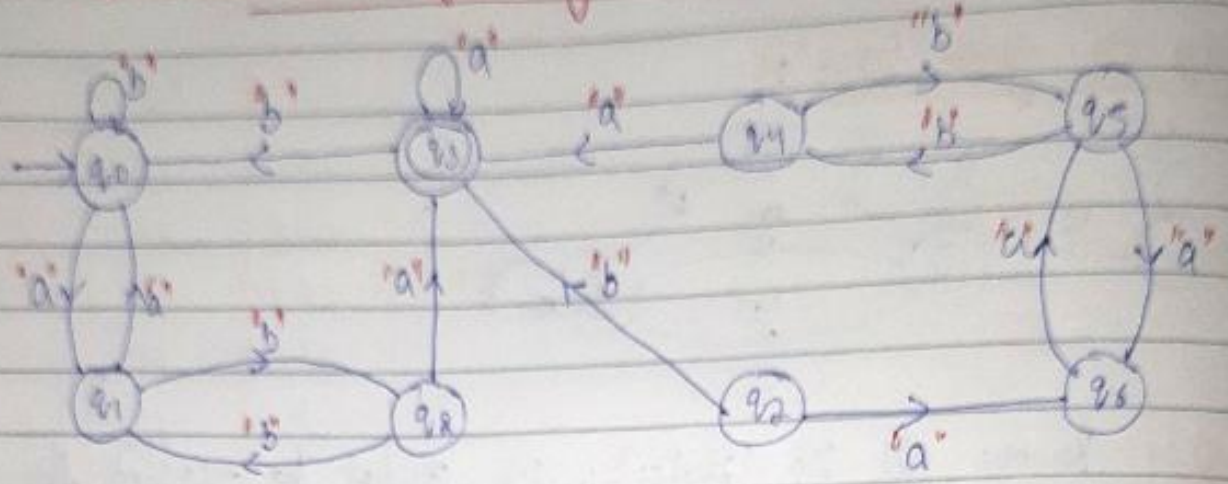
② "1-Equivalent states" :- [q0] [q1, q2]

Stop here.

Now minimized "DFA" is :-

"Minimized DFA"

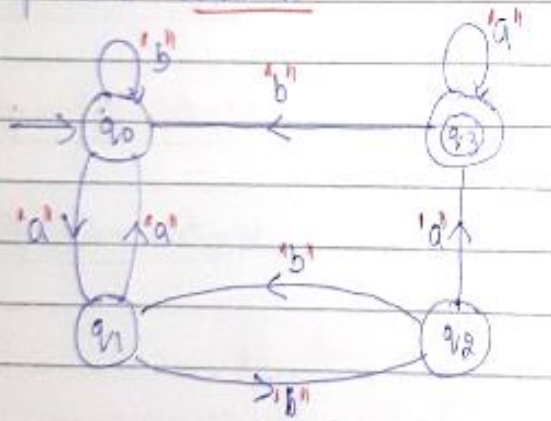
Q:> Minimize the following DFA :>



Sol:> Step 1:> Remove the states which cannot be reached from initial state "q_0".

Here "q_4, q_5, q_6 & q_7" are removed

So, "DFA" becomes :>



Now "Transition Table" is as :>

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

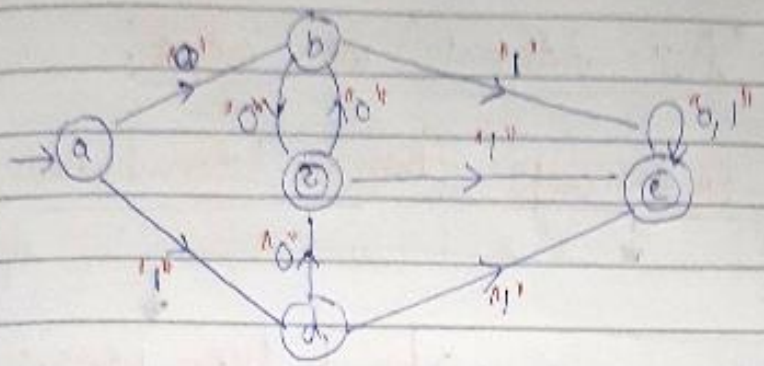
"Zero-equivalent states" :> $[q_0, q_1, q_2]$ $[q_3]$

"One-equivalent states" :> $[q_0, q_1]$ $[q_2]$ $[q_3]$

"Two-equivalent states" :> $[q_0]^1$ $[q_1]^2$ $[q_2]^3$ $[q_3]^4$

This is minimized DFA.

Q:> Minimise the following DFA :>



Sol:> Step1:> Remove the states which cannot be reached from initial state 'a'. There is no state (no such state).

Now state transition diagram is as :>

	"0"	"1"
→a	b	d
b	*c	*e
*c	b	*e
d	*c	*e
*e	e	e

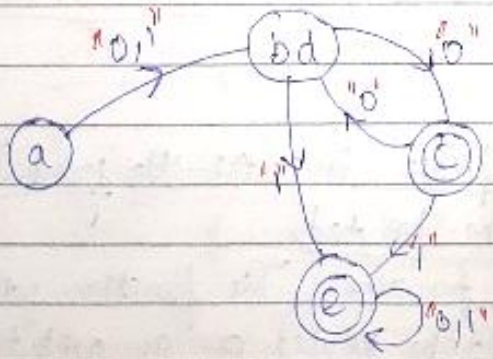
"One-equivalent states" :> [a,b,d] , [c,e]

"One-equivalent states" :> [a] [b,d] [c] [e]

"two-equivalent states" :> [a] [b,d] [c] [e]
('1') ('2') ('3') ('4')

Stop here

Now "Minimised DFA" is as :>



"Minimised- DFA"

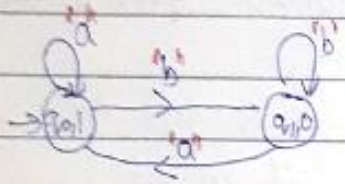
"P.T.O"

"Introduction to Moore & Melay Machine"

Let us now consider "finite-automata" with "output":

"FA with output" (Both are deterministic)

"Moore Machine"



$\lambda : Q \to \Delta$

$(Q, \Sigma, \delta, q_0, \Delta, \lambda)$

Both of these are characterized by above "values".

$Q \to$ "finite set of states"

$\Sigma \to$ "i/p alphabet"

$\delta \to$ "Transition function"

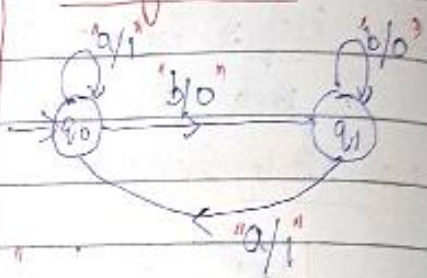
$[Q \times \Sigma \to Q]$

$q_0 \to$ "initial state"

$\Delta \to$ "O/p alphabet"

$\lambda \to$ "O/p function"

"Melay-Machine"



$\lambda : Q \times \Sigma \to \Delta$

Depending upon "current state" & "input symbol", machine is going to change its state.

"Δ" It is "output alphabet", these are the symbols which are supposed to be printed.

"λ" It is "output function". This function tells us, what should be printed as an output.

The only difference b/w "Moore-Machine" & "Melay-Machine" is of the "λ"

In Moore Machine, "λ is a function from 'Q to Δ'"

$$\lambda : "Q \to \Delta"$$

which means for 'every state output is associated'.

for 'q_0' output associated is '1' $\boxed{q_0 \to 1}$

for 'q_1' output associated is '0' $\boxed{q_1 \to 0.}$

In Mealy Machine, 'λ is a function from "$Q \times \Sigma \to \Delta$"

$$\lambda : "Q \times \Sigma \to \Delta"$$

which means for a state for some given input, "some output is associated.".

for $\boxed{q_0}$ for given input $\boxed{a}$ output is $\boxed{1}$. $(q_0, a) \to 1$

for $\boxed{q_0}$ for given input $\boxed{b}$ output is $\boxed{0}$. $(q_0, b) \to 0$

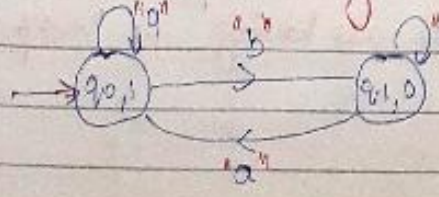
for $\boxed{q_1}$ for given input $\boxed{a}$ output is $\boxed{1}$ $(q_1, a) \to 1$

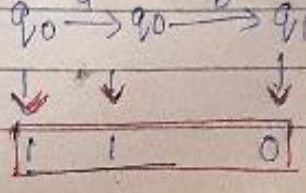
for $\boxed{q_1}$ for given input $\boxed{b}$ output is $\boxed{0}$. $(q_1, b) \to 0$

Note :⇒ "Moore Machine" is going to associate an output with the state itself.

"Mealy Machine" is going to associate an output as a combination of current state & input symbol.

Q:⇒ Given a "Moore machine" as shown below :⇒ let us see what is the output when it read a string "ab"

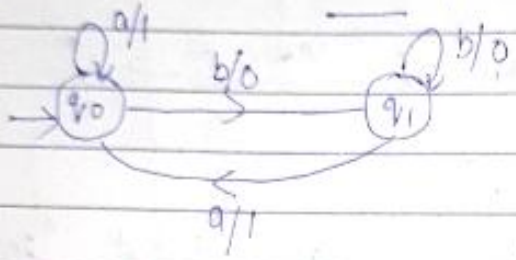




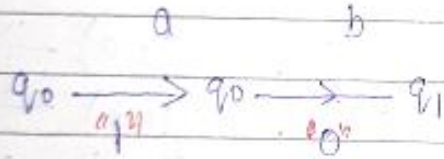
∴ on feeding an input of "ab", the moore machine has produced an output of "110".

Note:⇒ Given a string of length [n], the moore-machine will produce an output of length [n+1]. This is because even without feeding anything [q_0] is going to produce output.

Q:⇒ Given a "Mealy Machine" as shown below:⇒ Let us see what is the output of the machine upon feeding a string "a.b".



String to be checked is "ab" :⇒



Note:⇒ In "Mealy Machine" if string length is [n], then length of output produced is also [n].

→ "Examples of Moore-Machine" :⇒

Ex:⇒ Construct a "Moore Machine" that takes set of all strings over $\Sigma=\{a,b\}$, as input & prints [1] as output for every occurrence of [ab] as a substring.?

Sol:⇒ "Moore Machine" is a machine, for which output is associated with the state: i.e. for "each state", we are having one "output associated".

$\Sigma = \{a, b\}$

$\Delta = \{a, 1\}$

Given that, whenever we see an 'ab' we have to print '1'.

Eg:- "ab" → 1

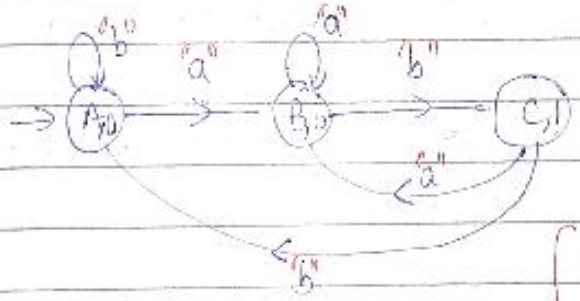
"abab" → 1 1

"ab bb ab" → 1 1

Here we need to count the no. of 'ab's in the "input string".

So, in order to construct this, we have to decide about three things.

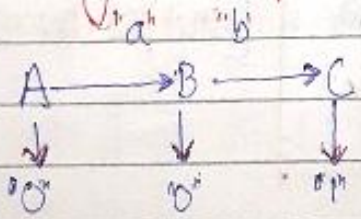
1) We can design the "deterministic finite machine" for set of all strings ending with "ab"



Whenever we see an "ab" we are going to state 'C' & print '1' as an output.

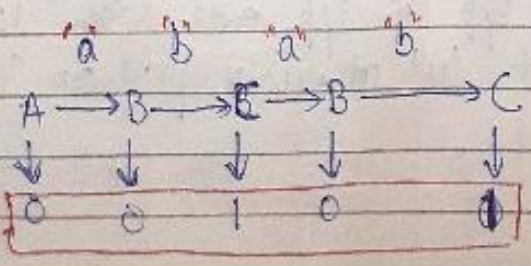
& for remaining states we can give '0' as an output.

Now let us say the input is 'ab'



output = "001"

Now let the input be abab



output = "0101"

Note :- Whenever we have to count some substring occurrences always remember, "endings will help us".

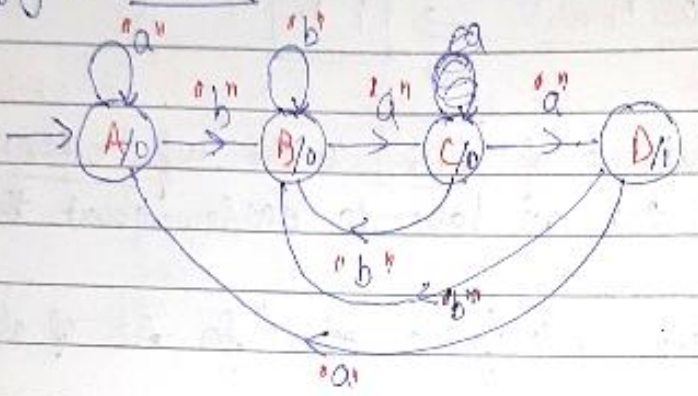
mycompanion

Q:- Construct a Moore machine that takes set of all strings over $\Sigma = \{a, b\}$ & counts no. of occurrences of substring "baa"

Sol:- $\Sigma = \{a, b\}$ $L = \{baa, baabaa, abaa, \dots\}$

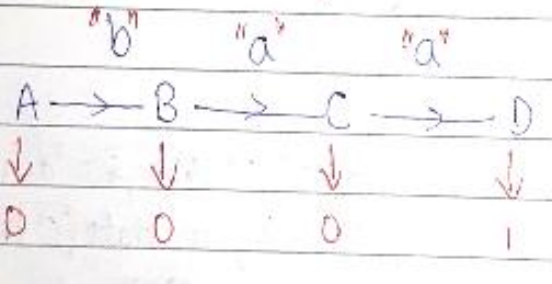
$\Delta = \{0, 1\}$

Ending with "baa".



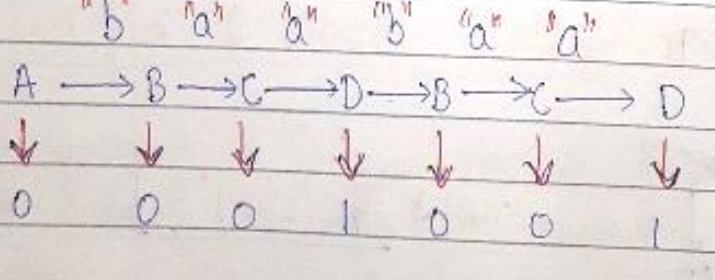
"Moore-machine"

Now suppose the string is "baa" let us see the o/p"



[Output is = "0001"]

Now let the string be "baabaa" let us see the "o/p"

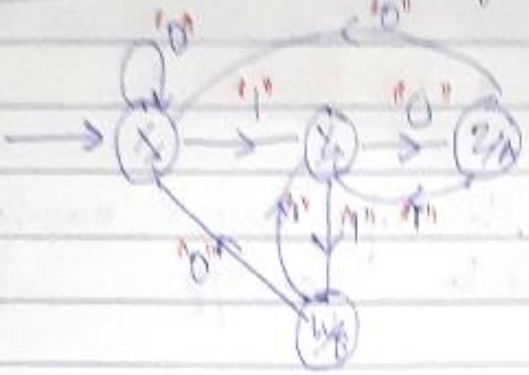


[Output is:- "0001001"]

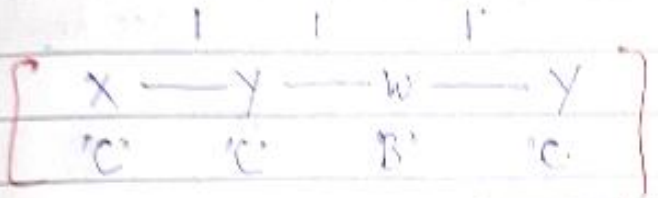
Q:- Construct a "Moore-machine" that takes set of all strings over $\{0,1\}$ and produces 'A' as o/p if i/p ends with '10' or produces 'B' as o/p if i/p ends with '11' otherwise produces 'C'.

Sol:- $\Sigma = \{0, 1\}$ & $\Delta = \{A, B, C\}$

Let us now construct a "DFA" for the language:-



Let the string be "111"



Q:- Construct a 'finite machine' that takes "binary numbers" as input and produces residue modulo '3' as o/p.

Sol:- $\Sigma = \{0, 1\}$

$\Delta = \{0, 1, 2\}$

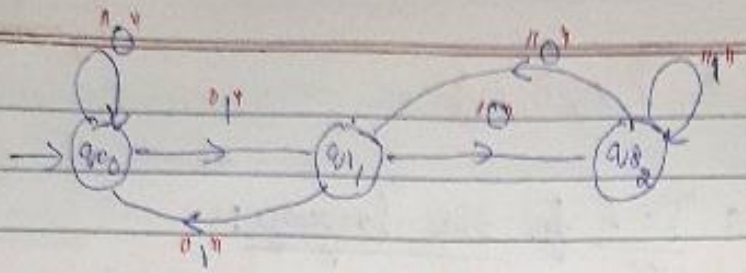
↳ "set of all possible remainders"

Residue modulo '3' means if we take any binary no. & divide it by '3' whatever is the remainder has to be printed

Now we move about the shortcut to draw "state-transition table" :-

	0	1	Δ
→ q_0	q_0	q_1	0
q_1	q_2	q_0	1
q_2	q_1	q_2	2

PTO



Let us extend this question a bit further, as.

Construct a "Moore Machine" that takes base 4 number's as i/p & produces residue modulo '5' as o/p.

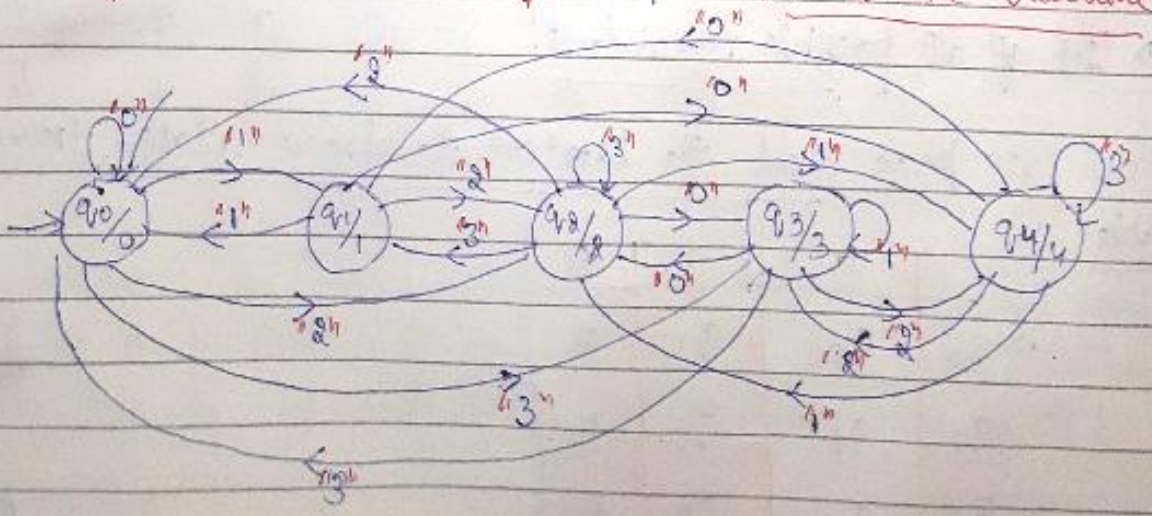
Sol:⇒ $\Sigma = \{0, 1, 2, 3\}$ & $\Delta = \{0, 1, 2, 3, 4\}$

↓ remainders when we divide any no. by '5'

Now "state transition Table"⇒

	0	1	2	3	Δ
→ q_0	q_0	q_1	q_2	q_3	0
q_1	q_4	q_0	q_1	q_2	1
q_2	q_3	q_4	q_0	q_1	2
q_3	q_2	q_3	q_4	q_0	3
q_4	q_1	q_2	q_3	q_4	4

"Now Moore Machine is"



→ "Examples of Mealy Machines":-

Q:- Construct a Mealy machine that takes binary number as 'i/p' and produces '2's complement of that number' as o/p. Assume the string is read 'LSB' to 'MSB' and end once is encountered.

Ans:- Mealy machine is:- "For every state, for every input there is a unique output associated.

$\Sigma = \{0,1\}$ & as output, we need to produce 2's complement of the binary no.

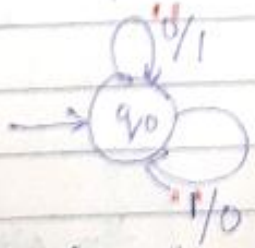
so output $\Delta = \{0,1\}$

[If we say input should be some "binary no" then output to be 2's complement of the no. which is "binary itself".]

Before solving it, let us take some "binary no" as an input & produce its "1's complement" as output

"1011" then its complement is "0100"

[Whenever there is '1' put '0' & wherever there is '0' put '1'.]



"1's complemented"

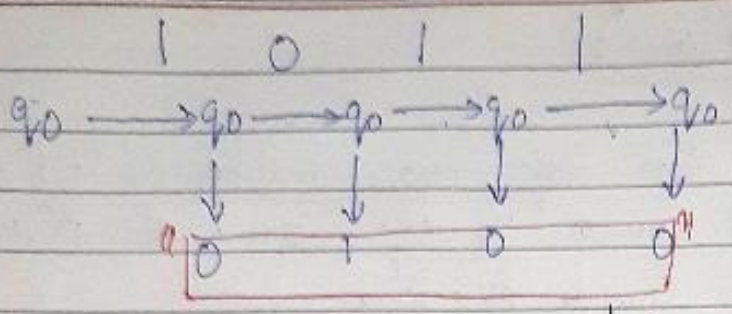
[If it sees a "0" convert it to "1" i.e produce output as '1' & if it sees a '1' it will produce an output of "0"]

'Only one state is required'

Eg:- 1 0 1 1

find its 1's complement using "Mealy Machine".

P.T.O



↳ 1's complement produced as output

Now let us talk about "2's Complement" of a number.

Eg:- Find 2's complement of "1100" ?

Sol:- Find 1's complement of "1100" & then add '1' to it

"1's" ⟶
1100 ⟶ 0011

0011
+ 1
0100 ⟶ 2's Complement of "1100"

Eg:- Find 2's Complement of "11101100" ?

Sol:- 11101100
1'sC = 00010011
+ 1
00010100

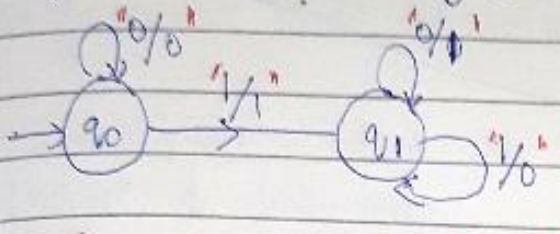
11101100 ← LSB
One's Complement
00010011
[leave 0's & 1st 1]
Then after 1st '1' complement everything that came

Eg:- Find "2's complement" of "11100" ?

Sol:- 00100

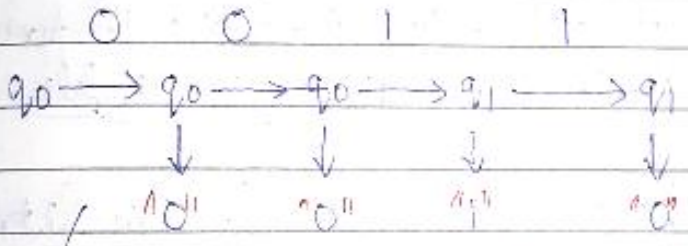
→ Now let us Construct "Mealy Machine" for finding "2's Complement" of a "binary-number."

Assuming that, we are reading from "LSB".



Now Example to understand it :⇒

"1100" we need to find "2's Complement" of this binary number.

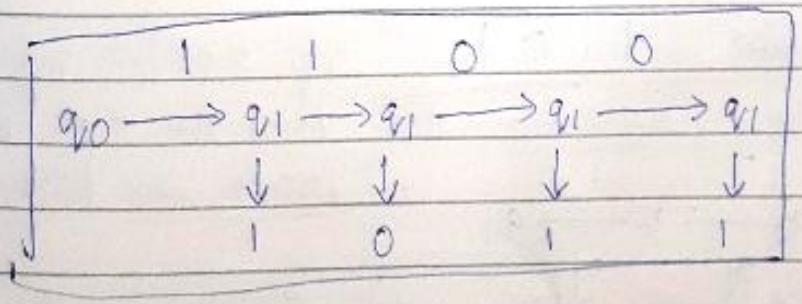


← Read from "R to L"

∴ [2's Complement is "0100"]

— o —

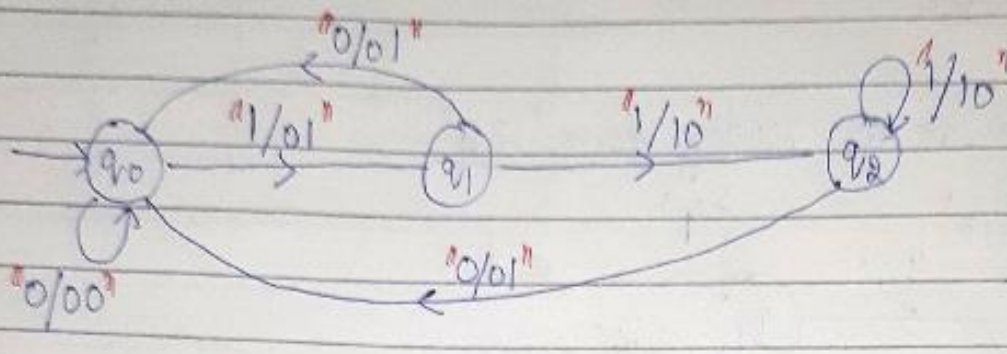
It Can directly be Operated as ⇒



X [As we are supposed to read from ending.]

"P.T.O"

Q:- What is the o/p produced by the following state machine?



a> "11 → 01" ✗ b> "10 → 00" ✗

c> "Sum of present & previous bits" ✓ d> [None of above] "NOT" ✗

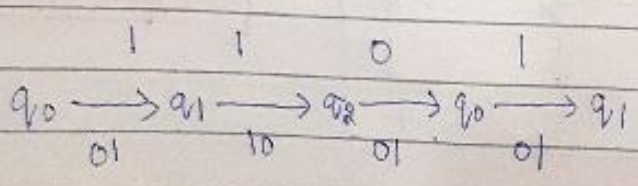
Sol:- a> "11→01" Can not be an output because for each 1 we will get '01' & here there are '11' two one, & output produced should be "0101" which is false.

b> "10 → 00" This is also wrong, because for this 1st's output produced should be "0101".

c> "Sum of present & previous bits":-

Always take an example to find out whether this option is "correct or not".

Eg:- let the string be 1101

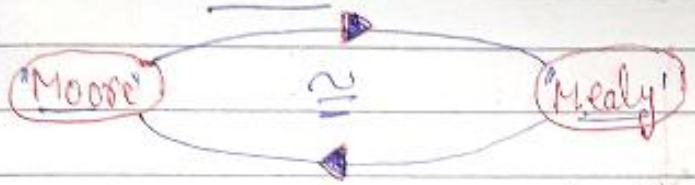


∴ output is 01(10)(01) This is true.

Note:- Both "Moore" & "Mealy" machines are actually equivalent in power.

Now in order to show that both "Moore & Mealy Machine" are equivalent in power, we would be convert them into one another.

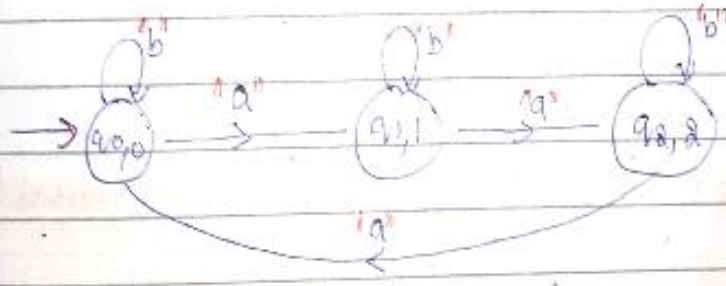
i.e. Given a "Moore Machine" we will be able to convert it to "Mealy Machine" & vice-versa.



"Conversion of Moore Machine to Mealy Machine" :-

Q:- Suppose we want to construct a "Moore Machine" which will count the no of [a's] in the input string. % 3. [a's % 3]

i.e. On [a's] we will count, but we will not count on seeing [b's].



"This is a Moore Machine"

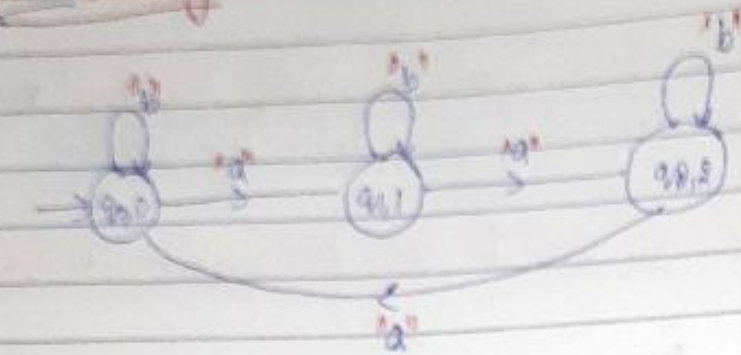
Now, we need to convert this "Moore Machine" into "Mealy Machine".

The major difference b/w "Moore & Mealy Machine" is in mealy machine o/p has to be associated with transitions, whereas in 'Moore machine' o/p is associated with states.

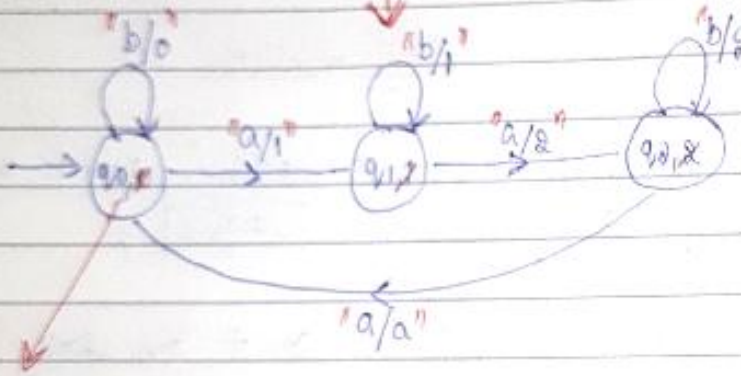
Let us convert it from "Moore to Mealy Machine" using two methods:-

P.T.O

1> Converting "Moore" to "Mealy" machine using "State transition diagram" Jointly :>



"Conversion to Mealy machine"



Now simply remove outputs associated with states.

2> Converting "Moore" to "Mealy" Machine with "State-Transition Table" Only :>

"for Moore Machine" → "output associated"

	"a"	"b"	Δ
→ q_0	q_1	q_0	0
q_1	q_2	q_1	1
q_2	q_0	q_2	2

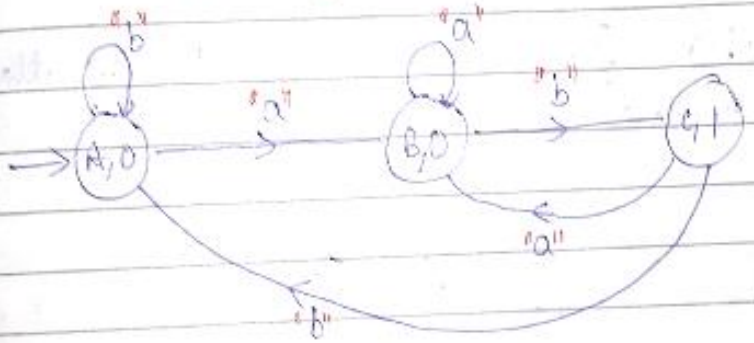
Now in order to convert "Moore to Mealy Machine" using "state transition Table"

the each state ,it where it is going to on seeing an ["input"], then go to that state & see the [O/P] associated with it..

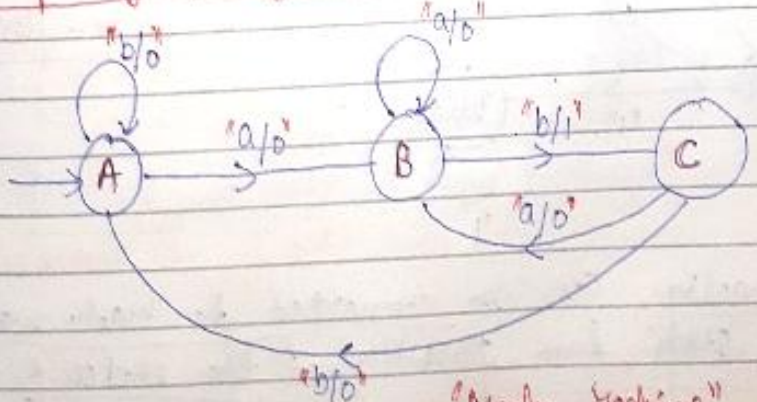
	"a"	"b"
→q_0	$(q_1/1)$	$(q_0/0)$
q_1	$(q_2/0)$	$(q_1/1)$
q_2	$(q_0/0)$	$(q_2/1)$

→ "This is state transition table for Mealy Machine"

Q:→ Convert the following "Moore-Machine" into "Mealy Machine" ?



Now corresponding "Mealy Machine" is :→



"Mealy Machine"

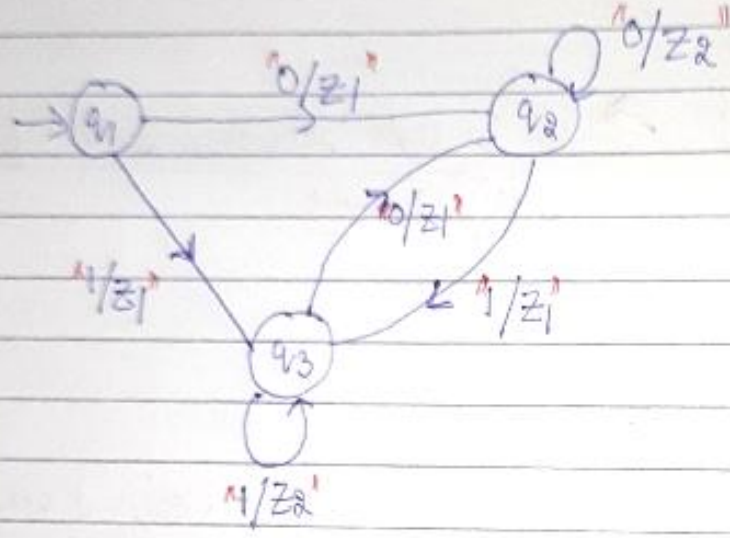
P.T.O.

Note:⇒ "Conversion of "Moore" to "Mealy" machine no of states is same"

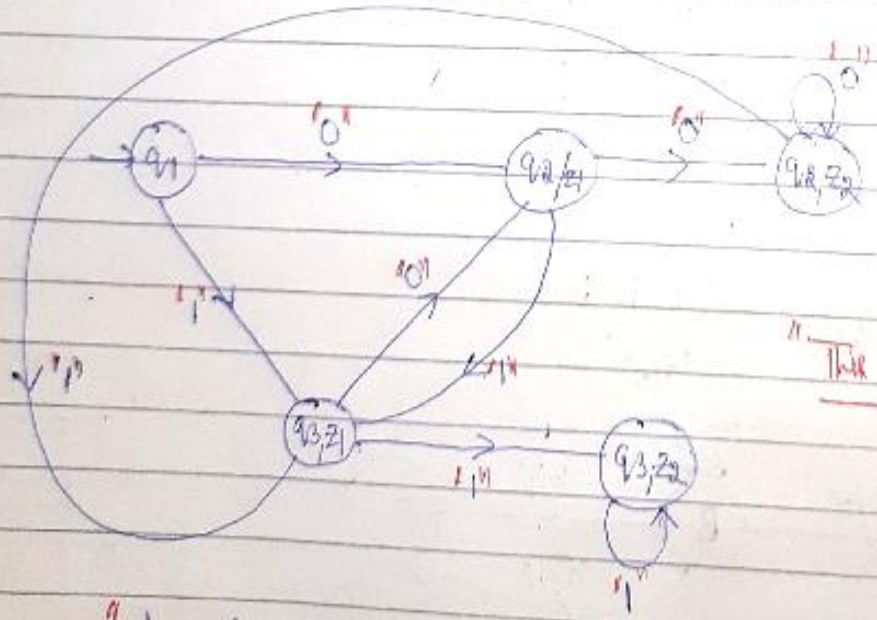
"Conversion of Mealy to Moore machine", in worst case no of states may be double

⇒ "Conversion of Mealy Machine" to "Moore Machine":⇒

Eg:⇒ Given a "Mealy machine" as follows, convert it to "Moore machine":⇒



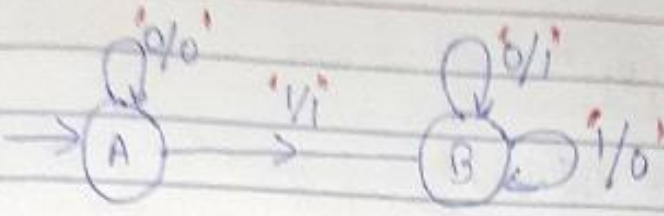
Method is as:⇒ start from initial state & pass the output from transitions to state



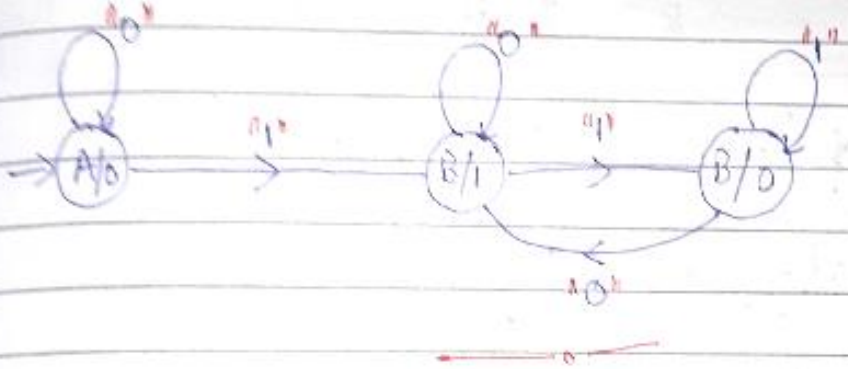
"This is Moore-Machine"

Note:⇒ "A mealy machine can be converted to moore machine by transferring the 'state' from transition to the state"

Q:- Convert the following 'Mealy Machine' to 'Moore Machine' :-



Corresponding Moore Machine is as :-



"Epsilon NFA & Conversion of "Epsilon NFA" to "NFA""

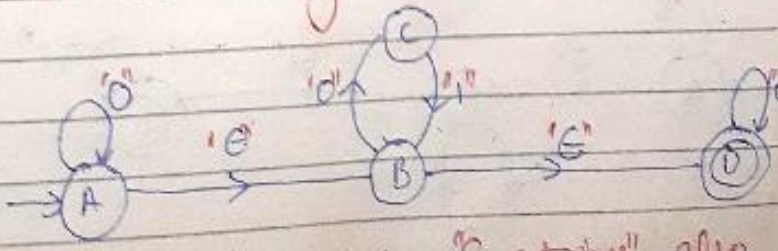
Till now we have seen "NFA, DFA" of finite Automata with output which are "Moore & Mealy Machines".

Now let us see a special kind of NFA called "∈-NFA" (Epsilon-NFA) :-

(∈-NFA) "∈" means "empty string"

Note :- "∈-NFA" is different from ordinary "NFA" in the sense that a transition in "∈ NFA" may even contain a "∈" i.e. an empty string.

Eg :-



Only difference is there are "∈ strings" also.

P.T.O

'∈-NFA' :→ It can be defined as :→

$(Q, \Sigma, \delta, q_0, F)$

Here 'δ' is a bit different

'q_0' is initial state & 'F' is set of final states.

In "$\delta : Q \times \Sigma \longrightarrow 2^Q$" { This is for NFA } "simple NFA"

Now in case of '∈-NFA' 'δ' is defined as :→

$$\delta : Q \times \Sigma \cup \{\in\} \longrightarrow 2^Q$$

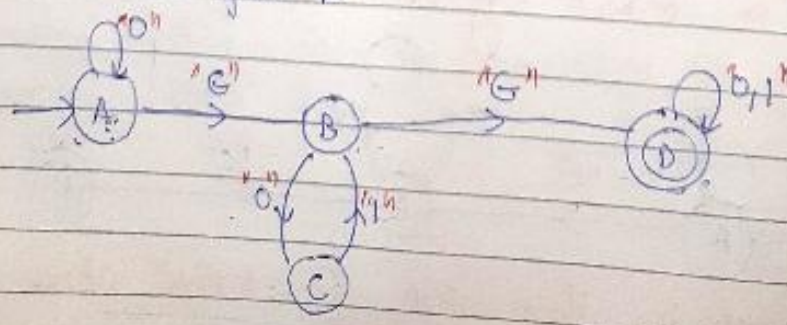
We even have an "empty string"

"Jump"
- It means in a "particular state", on seeing some input we can go to some other state
- Or in a "particular state", without seeing some input we can still go to some other state

∈-Closure property :→

∈-Closure (A) :→ ∈-Closure (A) means what are all the states which we can reach only on seeing an '∈' from 'A'

Eg :→



→ "Always included"

ε-closure (A) :- {A, B, D}

Now, the Question is how to convert ["ε-NFA"] to ["NFA"] :-

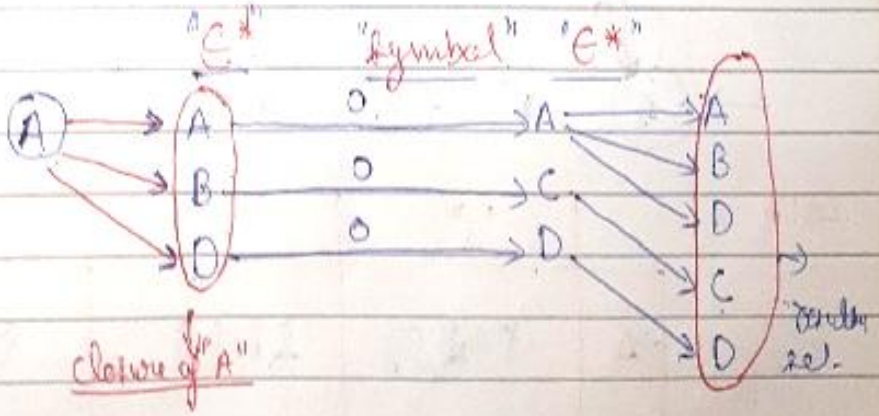
"Conversion of "ε-NFA" to "NFA" :-

So we will convert it using "State-Transition-Table" method :-

In final NFA, we won't be having any "ε" symbol" :-

Now procedure is ε* symbol ε* (Table has to written)

		"0"	"1"
Now	A	{A, B, C, D}	{D}
	B	{C, D}	{D}
	C	{ϕ}	{D}
	D	{D}	{D}



Note: [illegible]

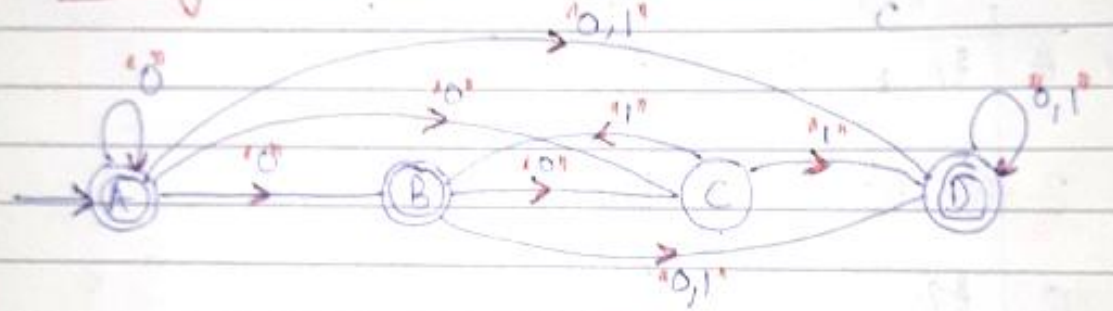
"C"	"Symbol"	"δ"
B	0	C
		D
	1	D

"C"	"Symbol"	"δ"
C	0	ϕ
C	1	B
		D

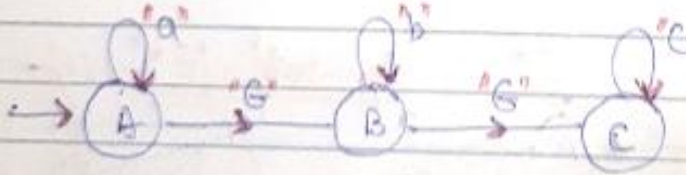
"C"	"Symbol"	"δ"
B	0	ϕ
	1	D
D	0	D
	1	D

"C"	"Symbol"	"δ"
D	0	D
	1	D

Ex: "Resulting NFA with ϵ-Moves like as":

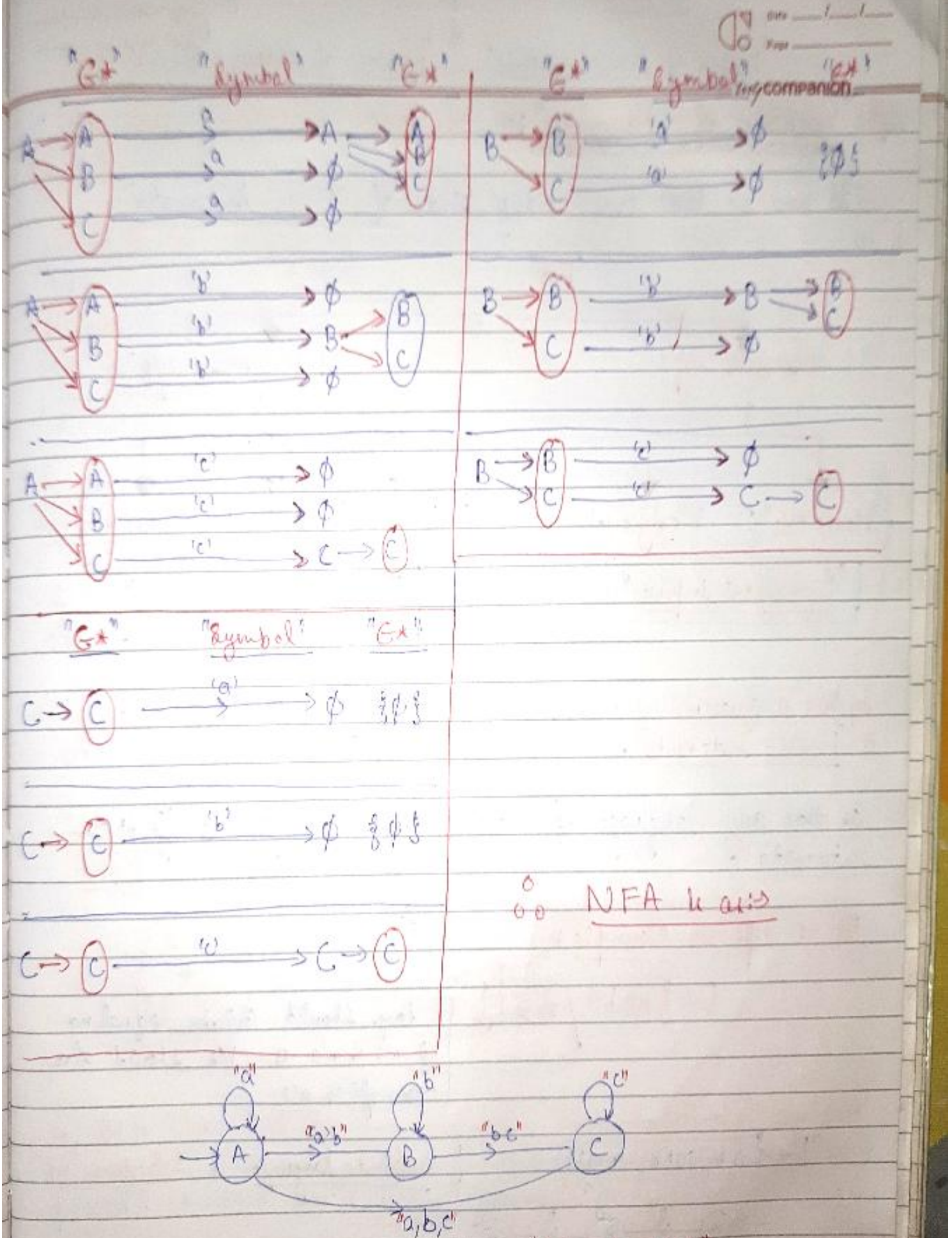


Q:→ Convert given "ϵ-NFA" to simple "NFA":



Sol:→

	"a"	"b"	"c"
→A	{A,B,C}	{B,C}	{C}
B	{ϕ}	{B,C}	{C}
C	{ϕ}	{ϕ}	{C}

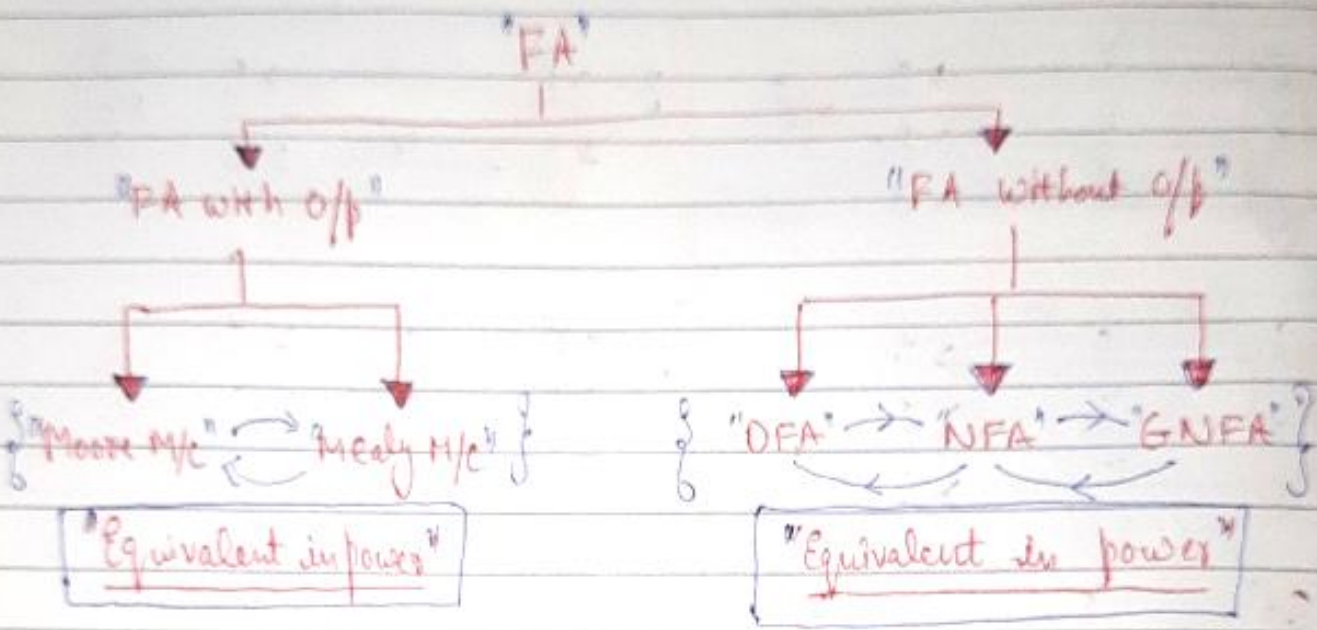

"G*"
"Symbol"
"ϵ*"
"ϵ*"
"Symbol"
"ϵ*"
"G*"
"Symbol"
"ϵ*"
NFA kaise
"a"
"b"
"c"
"a,b"
"b,c"
"a,b,c"
"Required NFA"
P.T.O

Note :- "Power of NFA" = "Power of DFA" = "Power of G-NFA"

"Power of Moore M/c" = "Power of Mealy M/c"

"Families of Languages" :-

Till now we have seen many kind of "Finite Automata" :-



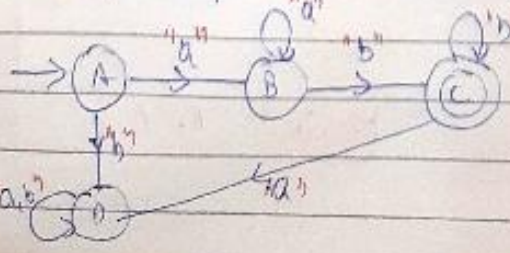
So, here a question arises, Can we accept "any language" using a "Finite-Automata".

Is there any language for which we cannot give a "finite-Automata"

Let us take an example :-

$L = \{ a^n b^n / n \geq 1 \}$ [lang. should contain equal no of a's & b's & b's should always occur after a's

"L" = {ab, aabb, aaabbb - - - -} "infinite language"



This finite automata will not be able to pop a stack of no of 'a's & 'b's.

as it is going to accept 'aab' even but it is not present in the language. as no. of a's should be equal to number of b's.

So here we want be able to count the no. of a's & b's.

using "finite automata" we can count finite no. of elements, not infinite.

So, in order to count "infinite no. of a's" we need "infinite amount of memory".

∴ There are some languages for which constructing a DFA, NFA or a "finite-Automata" is not possible.

↳ $L = \{a^n b^n \mid n \geq 1\}$ is one such example. The reason is "FA" has a finite amount of memory.

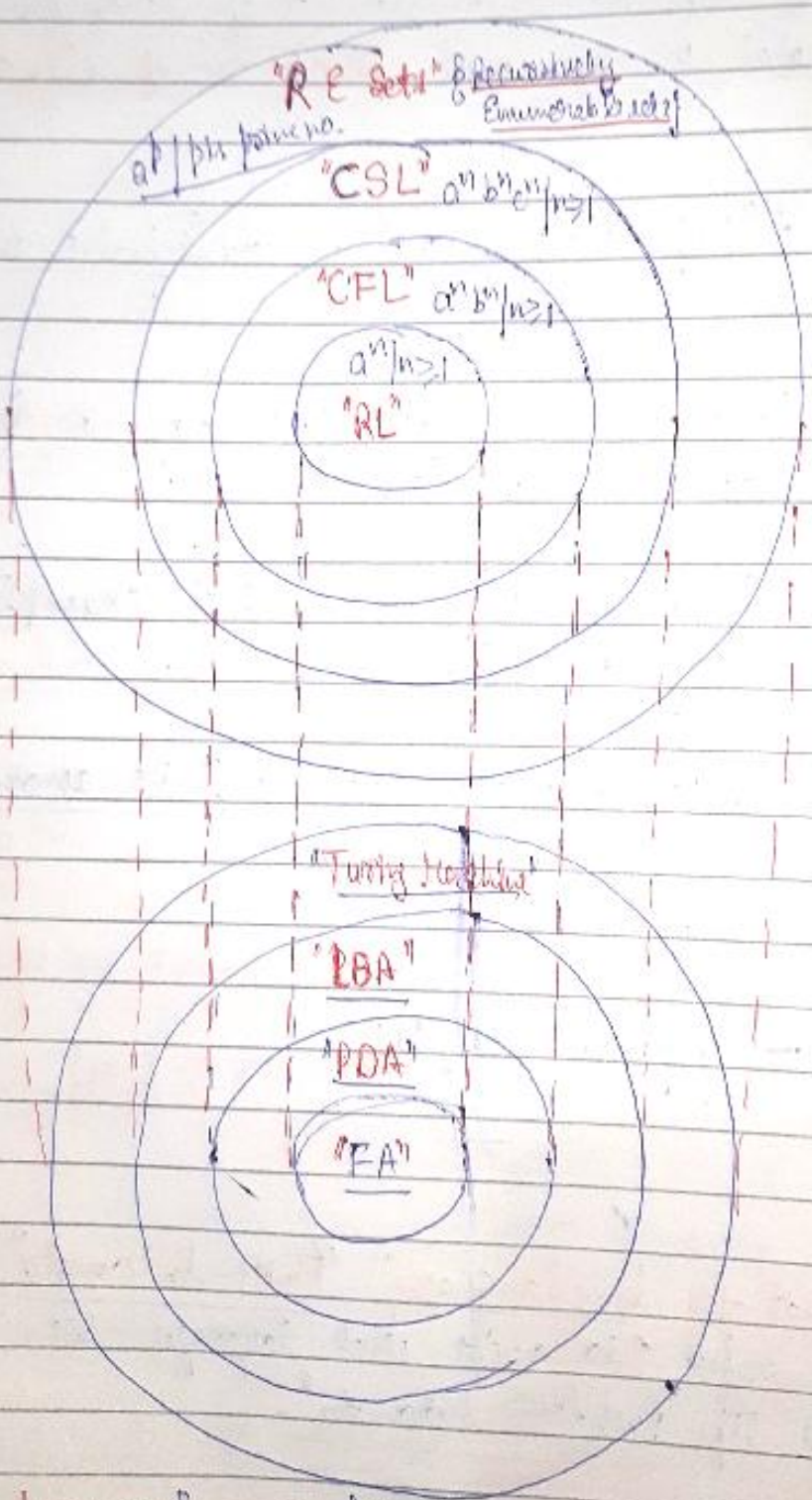
∴ we want a "Finite Automata" with "infinite amount of memory".

"FA" + "Infinite memory" = is known as "PDA".

So, it is clear that for "some languages" "Finite-Automata" can not be constructed. So in order to accept such languages we need some powerful machines like "Pushdown Automata".

PTO.

Now let us Categorize the type of "languages" on the basis of machines that accept them.:

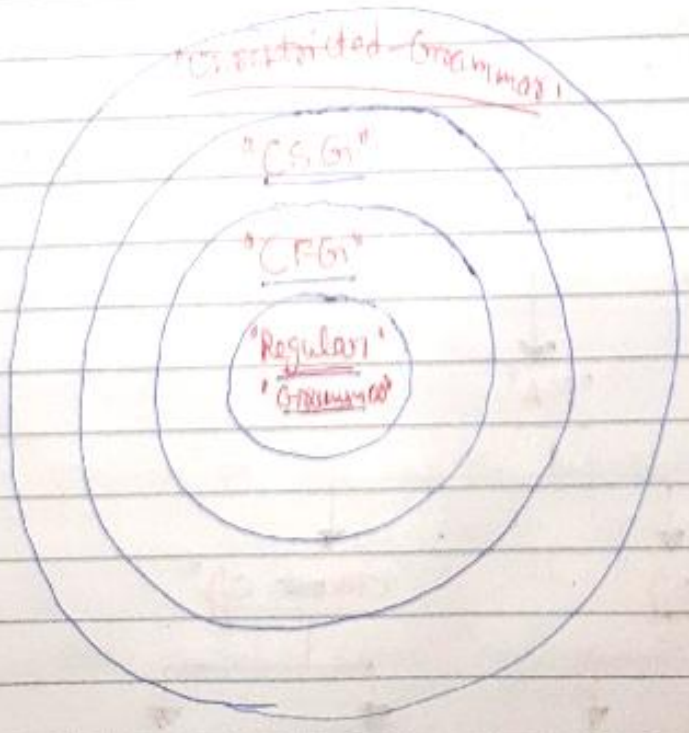


1> "Regular Languages" ⟶ "Finite - Automata"

- "Context - Free - language" → "Pushdown - Automata"
- "Context - Sensitive - Language" → "Linear - Bounded - Automata"
- "Recursively - Enumerable Sets" → "Turing - Machines"

For generating these "languages", we are having "grammars" :-

- "Regular - Languages" → "Regular - Grammars"
- "Context - Free - Language" → "Context - Free - Grammar"
- "Context - Sensitive - Language" → "Context - Sensitive - Grammar"
- "Recursively - Enumerable - Sets" → "Unrestricted - Grammar"



"P.T.O"